ComDEX: A Context-aware Federated Platform for IoT-enhanced Communities

Nikolaos Papadakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assoc. Prof. *K. Magoutis*

Thesis Supervisor: Assoc. Prof. *G. Bouloukakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

This work has been partially supported by the Greek Research Technology Development and Innovation Action "RESEARCH - CREATE - INNOVATE", Operational Programme on Competitiveness, Entrepreneurship and Innovation (ΕΠΑνΕΚ) 2014–2020 through the SmartCityBus project (T2EΔK-02848).

University of Crete
Computer Science Department

ComDEX: A Context-aware Federated Platform for IoT-enhanced Communities

Thesis submitted by
Nikolaos Papadakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: [illegible]
Nikolaos Papadakis

Committee approvals: [illegible]
Kostas Magoutis
Associate Professor, Thesis Advisor

Georgios
Georgios Bouloukakis
Associate Professor, Thesis Supervisor, Committee Member

DIMITRIOS PLEXOUSAKIS
Digitally signed by DIMITRIOS PLEXOUSAKIS
Date: 2022.09.26 14:03:42 +03'00'
Dimitrios Plexousakis
Professor, Committee Member

Departmental approval: [illegible]
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, September 2022

ComDEX: A Context-aware Federated Platform for IoT-enhanced Communities

Abstract

This thesis presents *ComDEX*, a context-aware federated architecture and IoT platform for enabling data exchange between IoT-enhanced communities. Today, smart communities are highly heterogeneous and siloed as they can offer IoT applications and services only to their local community inhabitants. Though some aspects of the creation of an ideal context-aware IoT platform for smart communities can be covered by existing technologies, a solution in a federation of brokers that fully addresses the needs of a smart IoT community, as well as the solutions proposed for individual smart spaces, has not yet been properly defined. *ComDEX* relies on property graphs to represent smart community entities that include high-level context-based information (occupancy of rooms, etc.) and are automatically mapped to context-aware publish/subscribe messages. Such messages can be discovered and exchanged between communities via a hierarchical federated topology and an advertisement-based mechanism. The *ComDEX* prototype is implemented using well-known IoT technologies such as MQTT and NGSI-LD. In order to include existing smart buildings for future experiments, an NGSI-LD smart data model for smart spaces and devices was created to include missing entities like rooms and to properly create their relationships. To enable the quick generation of realistic NGSI-LD compatible building data, we built a parsing tool that converts IFC-standardized building information to our extended NGSI-LD entities. Finally, *ComDEX* is evaluated using a realistic smart port scenario and compared against different federation topologies. The experimental results demonstrate that the approach presented in this thesis outperforms existing HTTP-based solutions in IoT scenarios with synthetically generated workloads, with low impact in larger deployments where the number of hops between brokers in the federation increases.

***ComDEX*: Μια Κατανεμημένη Πλατφόρμα με Επίγνωση Συμφραζομένων για Κοινότητες που χρησιμοποιούν το Διαδίκτυο των Πραγμάτων.**

Περίληψη

Καθώς οι έξυπνες κοινότητες IoT στις σύγχρονες πόλεις ωριμάζουν, αυξάνεται και η πολυπλοκότητα τους και οι απαιτήσεις τους. Η διατήρηση σημαντικών πτυχών των ¨έξυπνων χώρων' όπως η προστασία της ιδιωτικότητας των δεδομένων που ανταλλάσσονται, η διαλειτουργικότητα, η υψηλή διαθεσιμότητα των συστημάτων καθώς και ο εντοπισμός δεδομένων συμφραζομένων (context) σε μια συνεργασία από έξυπνες κοινότητες, αποτελεί πρόκληση. Σήμερα, τέτοιες έξυπνες κοινότητες είναι ιδιαίτερα ετερογενείς και απομονωμένες, και προσφέρουν τις υπηρεσίες και εφαρμογές τους μόνο στα τμήματα της τοπικής κοινότητας τα οποία αποκλειστικά εξυπηρετούν. Πλατφόρμες διαδικτύου των πραγμάτων με επίγνωση συμφραζομένων (context awareness) υποστηρίζονται σε κάποιο βαθμό από υπάρχουσες τεχνολογίες, ωστόσο μια λύση ομοσπονδίας "μεσιτών" δεδομένων που να καλύπτει τις όλες τις προαναφερθείσες απαιτήσεις σε μεγαλύτερη κλίμακα είναι ακόμα υπό διερεύνηση. Σε αυτή τη διατριβή παρουσιάζουμε το *ComDEX*, μια αρχιτεκτονική και IoT πλατφόρμα. ομοσπονδίας με επίγνωση συμφραζομένων που παρέχει την δυνατότητα ανταλλαγής πληροφορίας μεταξύ έξυπνων κοινοτήτων. Αρχικά δημιουργήσαμε ένα NGSI-LD μοντέλο δεδομένων για έξυπνους χώρους και συσκευές IoT σε συνδυασμό με έναν αναλυτή που μετατρέπει πληροφορία κτιρίων από το διαδεδομένο σχήμα IFC στο μοντέλο μας. Τα δεδομένα αυτά στο *ComDEX* δομούνται χρησιμοποιώντας ένα σχήμα που στηρίζεται στους γράφους ιδιοτήτων με οντότητες που περιλαμβάνουν πληροφορίες που αναπαριστούν πληροφορίες υψηλού επιπέδου (π.χ πληρότητα δωματίων σε ένα κτίριο). Για την μετατροπή των οντοτήτων αυτών σε μηνύματα με επίγνωση συμφραζομένων (context) χρησιμοποιήθηκε ένα σχήμα δημοσίευσης/εγγραφής που βασίζεται σε θεματικό διαχωρισμό των μηνυμάτων. Η ανταλλαγή δεδομένων στο *ComDEX* γίνεται διαμέσου ενός ομόσπονδου συστήματος δημοσίευσης/εγγραφής όπου τα προσφερόμενα μηνύματα γίνονται διαθέσιμα με την χρήση τεχνικών διαφήμισης πληροφορίας. Για την προσαρμόσιμη δρομολόγηση των μηνυμάτων εισάγεται μια υβριδική και ιεραρχική τοπολογία ομοσπονδίας για τους μεσίτες δεδομένων. Το πρωτότυπο του *ComDEX* υλοποιείται με χρήση γνωστών τεχνολογιών του διαδικτύου των πραγμάτων όπως το MQTT και το NGSI-LD. Τέλος έγινε αξιολόγηση του *ComDEX* χρησιμοποιώντας ένα ρεαλιστικό σενάριο που αναπαριστά ένα έξυπνο λιμάνι σε σύγκριση με υπάρχοντες μεσίτες NGSI-LD και χρησιμοποιώντας για σύγκριση διαφορετικές τοπολογίες ομοσπονδίας. Τα πειραματικά αποτελέσματα δείχνουν ότι η προσέγγιση που παρουσιάζεται σε αυτή την εργασία υπερτερεί των υφιστάμενων λύσεων HTTP σε σενάρια IoT με συνθετικά παραγόμενο φόρτο εργασίας, με μικρό αντίκτυπο σε μεγαλύτερες υλοποιήσεις όπου ο αριθμός των δικτυακών συνδέσεων μεταξύ των μεσαζόντων στην ομοσπονδία αυξάνεται. Το *ComDEX* θα μπορούσε να θεωρηθεί η αρχή μιας αρκετά φιλόδοξης πλατφόρμας με μια πιθανή μελλοντική δυνατότητα αυτόματης διαμόρφωσης της με βάση τις απαιτήσεις του κάθε αποδέκτη των δεδομένων που προσφέρει.

Acknowledgements

I'd want to express my gratitude to both of my supervisors for helping me in the conception and completion of this thesis. I'd like to express my sincere thanks to Prof. Kostas Magoutis for initially giving me the opportunity to collaborate with him on the scope of my thesis. I'd like to express my appreciation to Prof. Georgios Bouloukakis for his personal guidance during my research. I'd want to convey my gratefulness to both for their encouragement, understanding, excellent partnership, and willingness to lend their time to mentor me. I would also like to thank Prof. Dimitris Plexousakis for partaking in the examining committee of this thesis. Additionally, I want to thank my junior high school and high school teachers of information technology, especially Dimitris Antonoglou and Emilianos Evangelinos, for inspiring me to pursue a higher education in their field. Finally, I want to convey my sincere thanks to my parents, siblings, and friends for their unwavering support throughout my academic career. None of this would be achievable without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The advent of the Internet of Things (IoT) has gradually transformed cities into intelligent communities (residential areas, universities, smart ports, smart hospitals, etc.) that have changed how people approach everyday activities. Such smart communities offer important services such as smart governance and public safety, environmental monitoring, smart utilities, smart transportation, to name a few. To offer such services, it is essential to build IoT applications that leverage diverse IoT devices (sensors/actuators), data exchange systems, IoT protocols, Application Programming Interfaces (APIs), data models, and interoperability/-data processing software nodes, etc. To develop such applications, the common practice is to leverage open-source platforms (Orion-LD, RabbitMQ, EMQ), commercial IoT platforms (EvryThng, Google Cloud IoT, Azure IoT, Cisco Kinetic), as well as visual programming tools (Node-RED, ThingsBoard, IFTTT).

While the above platforms expose APIs for enabling Web or mobile clients to access IoT data from applications, their backend APIs are usually developed to provide IoT data access to specific spaces (a city port, university building, etc) and IoT devices. Therefore, IoT applications are not portable but constrained to be used in specific smart spaces. In addition, since IoT devices belong to different individuals/organizations, a smart community typically leverages the owners' IoT resources to provide applications and services to the same community inhabitants (students, port authority, etc.). This leads to the creation of vertical and siloed communities.

Because multiple smart communities can be deployed in wide-scale areas (e.g., university campus, urban port), different IoT sources deployed in multiple communities can be exploited to build applications. For example, an air-quality monitoring application may encompass sensors deployed in the city port, smartphone sensors from city inhabitants, or even cameras from private security companies. Such applications are defined as ***widespread IoT applications***. To enable the development of such applications, different individuals/organizations must have access to communities' IoT data. For example, a city port collects WiFi connectivity data in a management system to infer occupancy levels and provides them to its port-authority personnel via a "room finder" application. In case of an emergency, emergency responders (ERs) would like to obtain situational awareness information (e.g., floor plans, occupancy levels, etc.) to their dashboards. However, to receive such data, ERs dashboards must exchange data with the port's community and automatically discover the port situation.

Exchanging information between smart communities requires the usage of common data models, IoT protocols and APIs. This is unrealistic in today's IoT systems due to the lack of portable and extensible IoT applications, standard methodologies for data exchange between communities, high-level data discovery mechanisms, and adaptable Edge infrastructures for new applications, devices and community changes. Additionally, community administrators must be supported with secured and trustworthy approaches when providing access to their community IoT data. Data sharing policies for advertising specific community information must be designed, along with discovery and data exchange mechanisms.

Existing state-of-the-art approaches have adopted **federated designs** to create smart-city platforms such as MARGOT [26], Fogflow [11], Trustyfeer [21], ALMANAC [10], CPaaS.io [12], and WiseIoT [16]. However, these mainly provide cross-domain communication, interoperability, and IoT resource discovery for specific communities/cities. The Space Broker [3] and the SemIoTic [35] framework define smart space characteristics and contextual data such as spatial representation of spaces. However, both works have focused on a single smart space and do not offer a solution for all of these important qualities in a federation of smart communities/smart spaces.

This thesis introduces *ComDEX*, a context-aware federated architecture and IoT platform for enabling data exchange between smart communities. In *ComDEX*, smart community data are structured using a *property graph schema* [5] with *entities* that include high-level context-based information (occupancy of rooms, etc.). A topic/type-based publish/subscribe subscription scheme [13] is leveraged to convert *entities* to *context-aware messages*. Data exchange between communities is enabled by relying on a federated publish/subscribe system where offered messages (smart community entities) are shared via *advertisement techniques.*For the adaptable routing of messages a hybrid and hierarchical federation topology is introduced.

1.1 Research Contributions

The main contributions of this thesis can be briefly displayed as follows:

1. The mapping of property graphs to topic/type-based subscriptions for enabling context-aware data exchange in smart communities via a publish/subscribe-based advertisement-based policy.

2. A hybrid and hierarchical federation topology, as well as an algorithm for message routing between brokers and support for the deployment of widespread

IoT applications in smart communities.

3. The creation of NGSI-LD data models for smart spaces (buildings, transportation systems) combined with a parser from IFC data entities and the formation of NGSI-LD data models for smart devices.

4. A prototype implementation of *ComDEX* and its federation architecture using state-of-the-art technologies such as NGSI-LD, MQTT and existing message broker implementations.

5. An evaluation of the *ComDEX* prototype and topology using emulated data for realistic communities in a smart port scenario.

1.2 Thesis Outline

This thesis is organized as follows:

We begin Chapter 2 by providing a background on the important concepts and technologies that facilitate this thesis. In this chapter, we also acknowledge the existing technologies that exist and can be used to satisfy some objectives that have been outlined expressly for the development of the optimum context-aware IoT platform for smart communities. We additionally, point out the limitations of such endeavors, and explain how our work is going to differ by handling all these highlighted limitations.

We resume in chapter 3 by presenting a scenario involving communication between different organizations on the scope of handling of a smart port in order to showcase the benefits of achieving cross-smart-community data exchange as a main motivation of *ComDEX*. A high-level overview of the architecture of our system is outlined, with the function of its various components explained.

In chapter 4, we continue with the representation of the data within *ComDEX* and, more specifically, with the modeling of smart spaces and devices. In summary, we present a new NGSI-LD data model for smart buildings that we accompany with a parser from the IFC schema to our data model. Next, we present similarly made models for smart public transportation, in particular for buses. Lastly, we present a data model for devices, in which we separate the modeling of the device itself from its observations, and we showcase with examples how the previous models can be conjoined with this one to model certain aspects of a smart community.

In chapter 5 the *ComDEX* formal model, which consists of a property graph based data schema linked to a publish/subscribe subscription mechanism for the production of context-aware messages, is provided. The formal definitions of a federated system for data exchange among smart communities are then outlined. Then, formally presented are the actions that can be carried out in such a system.

Continuing in chapter 6 from the formal model, we present the *ComDEX* federation topology, whose featured architectural solution is based on propagating advertisements of context-aware messages in the federation.

In chapter 7, having the previous 2 sections covering the system design, we proceed with the description of an implemented prototype, which is based on technologies such as NGSI-LD (the models we created in chapter §4) and MQTT as the main communication protocol, using mosquitto as the underlying broker. We explain how the prototype would work in the port example, and how the federation can be modified to enable resilience.

In the last of the main chapters, chapter 8, we evaluate *ComDEX* and the overall design approach using our prototype presented the previous chapter. Using straightforward synthetic data models of a typical NGSI-LD entity, we first assess the performance of the *ComDEX* prototype with various topology sizes and compare it to existing NGSI-LD brokers in both a best- and worst-case scenario. On the basis of current NGSI-LD data models and actual IoT device traces, we then assess the effects of altering the federation topology and advertisement granularity using randomly created entities of smart buildings. Our findings from every experiment confirm that our current *ComDEX* prototype performs well under a variety of deployment configurations and generated workloads.

Finally, in chapter 10 this thesis is briefly concluded and related future work is also described.

Chapter 2

Background

2.1 Useful Background Information

The following section contains a collection of useful background information related to this thesis, for the interested reader that is not well versed in the pub/sub domain. An experienced reader can skip this chapter to go directly to section 9.
The concept of hypertext links, which allow a link on one web page to direct the browser to loading another page from a known location, should be recognizable to readers. While computers can comprehend relationship discoverability and how links function, they find this to be considerably challenging and need a clear protocol to go from one data element to another stored in a different location. **Linked Data** is such a method of connecting disparate documents and Web sites through a network of machine-interpretable data based on industry standards. It enables an application to start with one piece of Linked Data and then follow embedded links to other pieces of Linked Data located on various sites throughout the Internet.
JSON-LD [30] is a simple JSON syntax for serializing Linked Data. Its design makes it easy to convert existing JSON to Linked Data with little changes. JSON-LD is primarily intended for usage in Web-based programming environments, the development of interoperable Web services, and the storing of Linked Data in JSON-based storage engines. The enormous number of JSON parsers and libraries available today can be reused since JSON-LD is 100 percent compatible with JSON.
The **NGSI-LD** specification [1] has been proposed by the ETSI Industry Specification Group for CrossCutting Context Information Management (ISG CIM), which comprises an information model with semantic characteristics connected to Linked Data and ontologies. In short it is an API and data model for publishing, querying, and subscribing to context data. Its purpose is to make the open exchange and sharing of structured data amongst various parties easier. It's utilized in Smart Cities, Smart Industry, and Smart Agriculture, as well as the Internet of Things, Cyber-Physical Systems, Systems of Systems, and Digital Twins more generally.

Entities, relationships and properties are the key components of the NGSI-LD information model Fig. 2.1. A real-world item, such as a building or a person, is represented by an entity. A relationship connects two or more entities, such as a person who works in a building. A property connects values to elements,[4] such as identifying that an entity corresponds to a real person. JSONLD is used to represent NGSI-LD entities. Instead of using RDF triples, which is typically used in Linked Data, JSON-LD tries to serialize entity data in a simple and effective method.

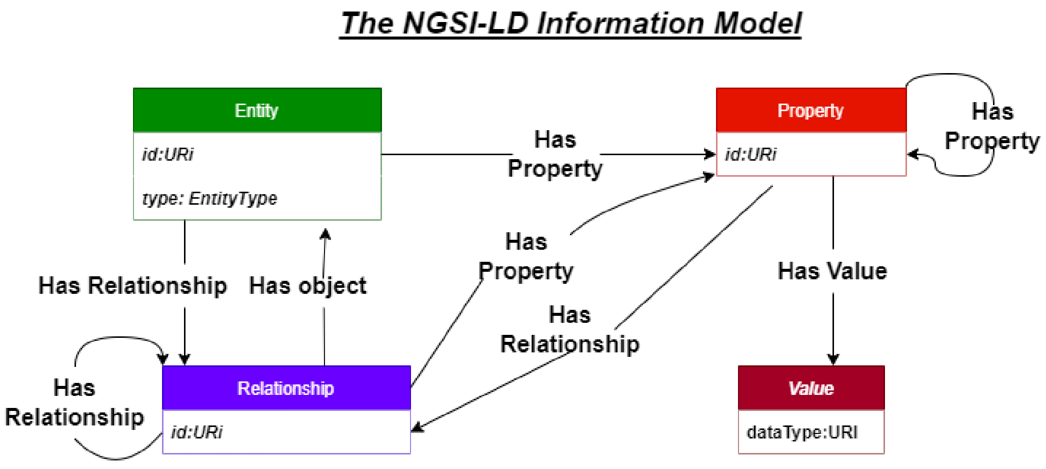


Figure 2.1: The NGSI-LD Information Model

The **Industry Foundation Classes (IFC)** [1] is a digital standard for describing the building asset domain. It supports vendor-neutral, or agnostic, and usable capabilities across a wide range of hardware devices, software platforms, and interfaces for many different use cases and is an open, international standard (ISO 16739-1:2018). It offers a comprehensive, standardized data format for the vendor-neutral interchange of digital building models, are a crucial foundation [9]. for the development of Big Open BIM (Building Information Models)

It is a sophisticated data model Fig. 2.2 that enables object-oriented representation of a building model's geometry and semantic structure. The structure is divided into its structural elements and its interior spaces, both of which are thoroughly delineated along with how they correlate to one another.

More explicitly, the IFC schema is a standardized data model that logically codifies numerous aspects of buildings and their parts, such as identity and semantics (name, machine-readable id, object type or function), characteristics or attributes (like material, color, and thermal properties), and relationships (including locations, connections, and ownership) of objects (like doors,windows), as well as abstract concepts (performance, costing) and people (owners, designers, contractors, suppliers, etc.).

[1]https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes/

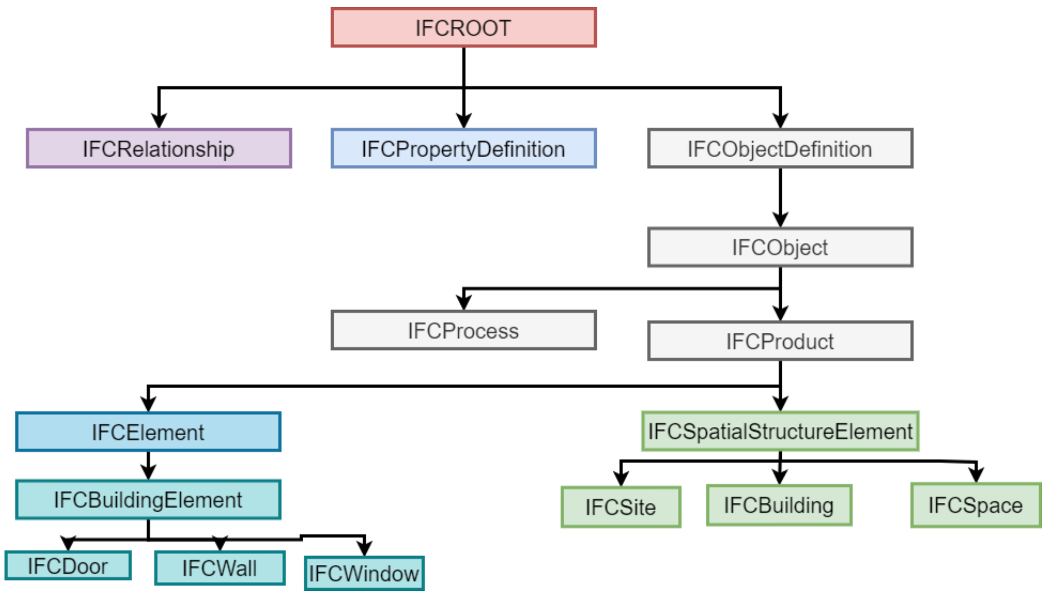


Figure 2.2: Some of the most important entities of the IFC hierachy

It may be utilized for practically any data exchange situation throughout the life cycle of a building due to this extensive data structure. **Message brokers** are commonly used to assist message collection and delivery to IoT services and applications in real time. There are several message broker implementations that can be used to implement such data collection brokers. Mosquitto is a lightweight broker written in C that is mostly used for prototyping. It allows data to be exchanged with MQTT-based clients but does not allow for clustering. RabbitMQ is the most extensively used Erlang-based open source message broker. It has clustering capabilities and supports MQTT and AMQP data exchange protocols. Apache Kafka is a centralized data pipeline with scalability, durability, and high throughput (millions of messages). Consumers subscribe to a specific topic and partition, while producers publish data to the topics of their choice.
Context brokers are considered to be context aware messaging middle-ware. Context aware is when a system uses context to give relevant information and/or services to a user, relevancy is determined by the user's task. [2].
Pub/Sub systems [13] are systems in which subscribers can indicate their interest in an event or a pattern of events using the publish/subscribe interaction paradigm, and they will be alerted of each event generated by a publisher that matches their registered interest. In other words, producers publish data on a software bus (an event manager), and consumers subscribe to the data they want to receive from that bus. The term event is commonly used to describe this information, while notification is used to describe the act of sending it.
NGSI-LD context brokers are Pub/Sub brokers that allow for the management and requesting context of information in a structured manner based on linked data

standards following the NGSI-LD specification of the ETSI standard. However the specification is a living, changing document (on version 1.5 as of July 2021), with features added in a pace that is hard for the NGSI-LD implementations to keep up. Currently there are 4 different NGSI-LD broker implementations, namely Scorpio, Orion-LD, Djane and Stellio. Orion-LD is the only context broker which can currently service both NGSI-v2 and NGSI-LD.

Broker federations

In broker federations [14] message routes in one broker (the source broker) are automatically routed to another broker, allowing messaging networks to be built by constructing message routes (the destination broker). These routes can be defined between two brokers' exchanges (the source exchange and the destination exchange), or between a queue in the source broker (the source queue) and an exchange in the destination broker. When bidirectional flow is required, one route in each direction is built. Routes might be long-lasting or just exist for a short period of time. A durable route survives broker restarts, allowing it to be restored as soon as both the source and destination brokers are available. The destination for a route is always a destination broker exchange. Message routes are formed by default by specifying the destination broker, which then contacts the source broker to subscribe to the source queue. This is referred to as a **pull route**. A route can also be built by specifying the source broker, which then contacts the destination broker to send messages. This is known as a **push route**, and it's especially beneficial when the destination broker isn't available at the time the messaging route is set up, or when a large number of routes with the same destination exchange are set up. Broker Federation can be used to establish huge communications networks, one route at a time, with numerous brokers. A full distributed messaging network can be configured from a single place if network connectivity allows it. Routing rules can be updated dynamically as servers, responsibilities, and times of day vary, as well as to accommodate other changing conditions.

Context providers and consumers may be geographically dispersed in a real-world deployment of a broker-based context aware system [19]. Multiple brokers in the system split into administrative, network, geographic, contextual, or load-based domains are preferable to reduce administration and communication overheads. Context providers and consumers can be set up to only communicate with the brokers who are closest, most relevant, and most convenient to them. This system, however, necessitates inter-broker federation so that providers and consumers affiliated with different brokers can work together effortlessly.

As for the topologies,[14] federated network is typically shaped as a tree, star, or line Fig. 2.3, with bidirectional linkages (implemented as two unidirectional links) connecting any two brokers. If just unidirectional linkages are employed, a ring topology is also viable. It takes time to send and receive messages (number of "hops"). The number of brokers between the message origin and final destination should be kept to a minimum for optimum performance. Tree or star topologies work best in most applications. In a federated network, usually there should be only one path from A to B for any pair of nodes A,B. Message loops can produce

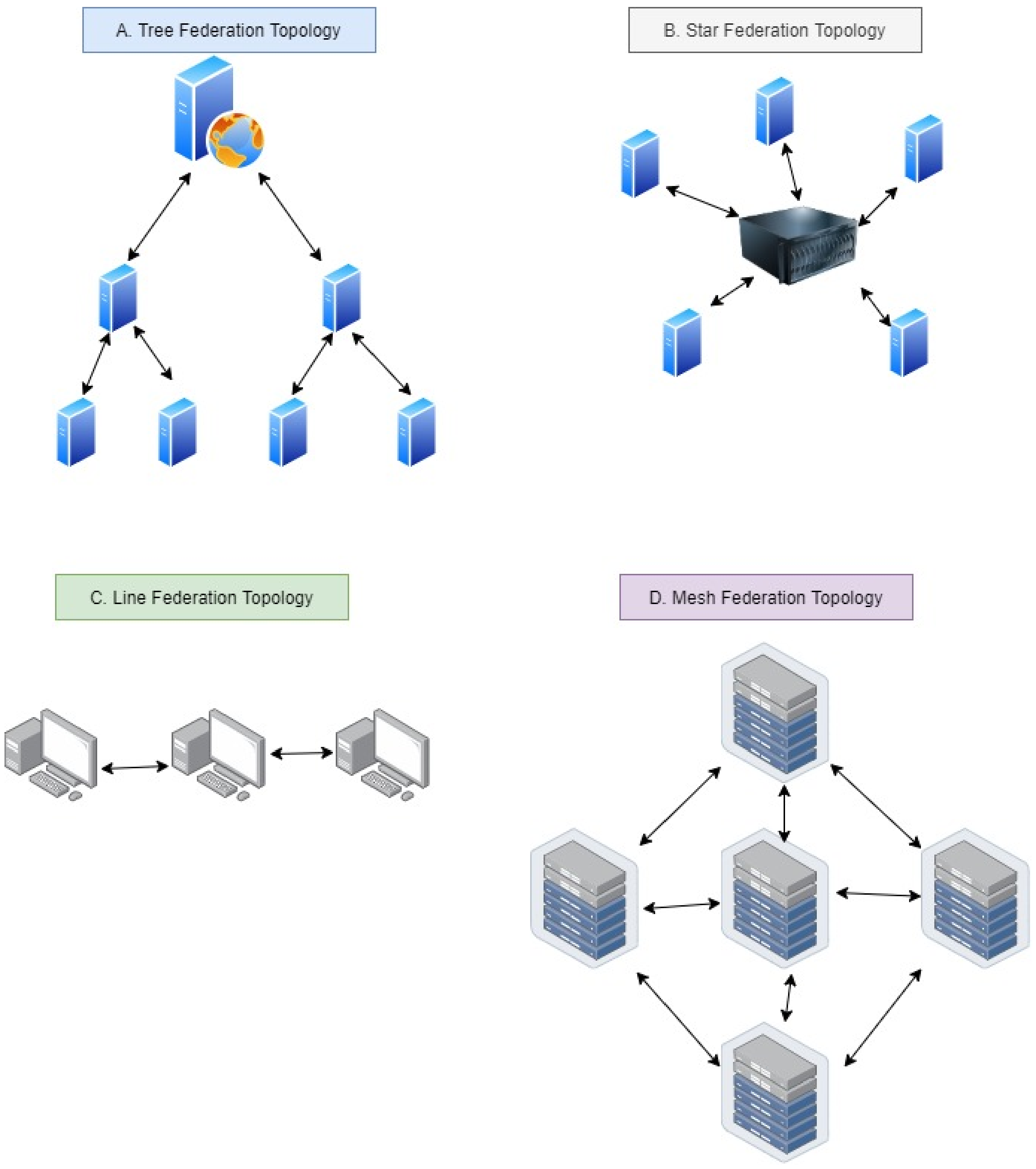

A. Tree Federation Topology
B. Star Federation Topology
C. Line Federation Topology
D. Mesh Federation Topology

Figure 2.3: Classic federation topology variations (a.tree), (b.star), (c.line), (d.mesh)

redundant message transmission and overload the federated network if there are several paths. Message loops are not present in the topologies described above. Due to the fact that a given broker can receive the same message from two separate brokers, a ring topology with bidirectional linkages or a mesh topology are examples of a topology that does cause this problem.
This concludes the background information chapter.

Chapter 3

Overview

3.1 Motivating scenario

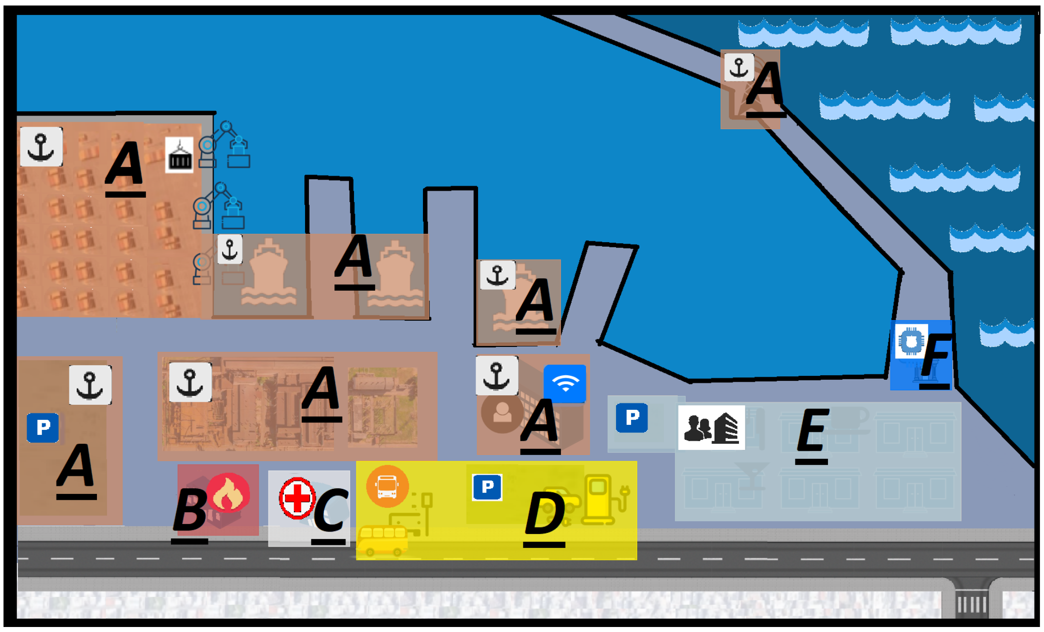


Figure 3.1: Smart communities in a modern city port

Large city ports such as shown in Fig. 3.1 comprise a variety of different organizations (e.g., maritime authority, port authority, city authorities, etc.) and communities that own or manage data, sensors, and applications. Modern such ports are already evolving into smart communities [25, 29] that stand to benefit from IoT applications, such as monitoring the occupancy of passenger stations, monitoring air quality using environmental sensors, marine and urban traffic management applications, etc. In such a heterogeneous environment, diverse sensors and actuators transmit data relevant to discrete data recipients. An example of an application relevant to a smart port is *smart urban transportation*. In what follows, the benefits of achieving cross-smart-community communication in this context will be highlighted, as a motivation for *ComDEX*.

For simplicity, lets assume the participation of just 3 different stakeholders in a smart port (Fig. 3.1): the port authority (zone Ⓐ), city firefighting department (zone Ⓑ), and Smart InterCity Bus organization (zone Ⓓ), each storing and managing the IoT data they own. The smart transportation application could benefit from information-sharing between these different stakeholders. Each of them however would only want to share the part of their data that it considers relevant. The Intercity bus organization and port authority would benefit by exchanging information (such as real-time vehicle and ship positions); the firefighting department may at times want to request information, such as the current occupancy of the passenger station in case there is a fire emergency, but not necessarily provide any to others. The connection topology between the various brokers should be one that helps in maintaining this. Information exchange would require data format standards to ensure interoperability and data discovery of dynamic smart transportation properties. Lastly, there ought to be ways to handle data and applications differently according to specific context information or data criticality. Considering the wealth of diverse static and dynamic contextual information available across smart communities, a key goal is to allow multiple IoT-enhanced communities/stakeholders to collaborate by exchanging information across their (currently siloed) IoT systems efficiently and in a QoS-aware manner, while preserving data sovereignty.

The main motivation for this work is to design a federated IoT platform architecture for smart communities that addresses this goal, building upon state-of-the-art technologies. Section 7.5 of chapter 7 provides details on how the proposed architecture, *ComDEX*, achieves these objectives in a smart port deployment. As a result, *ComDEX* was born.

3.2 The *ComDEX* Architecture

Here a high-level overview of the architecture of *ComDEX*, a context-aware federated platform for IoT-enhanced Communities, is provided. The *ComDEX* prototype implementation is provided as an open-source platform at `https://github.com/SAMSGBLab/\textit{ComDEX}--ngsi2mqtt`. The high-level architecture of *ComDEX* (see Fig. 3.2) consists of multiple components that are directly correlated with the port example in §3.1. The architecture presented here, facilitates the data exchange between IoT devices and IoT applications in smart communities based on three major components: i) The Federation Topology component, which handles the connection between various brokers; ii) the knowledge base component, which corresponds to the information model of *ComDEX*; and iii) the Action Handler component, along with its sub-components, which interacts with clients for various data exchange operations.

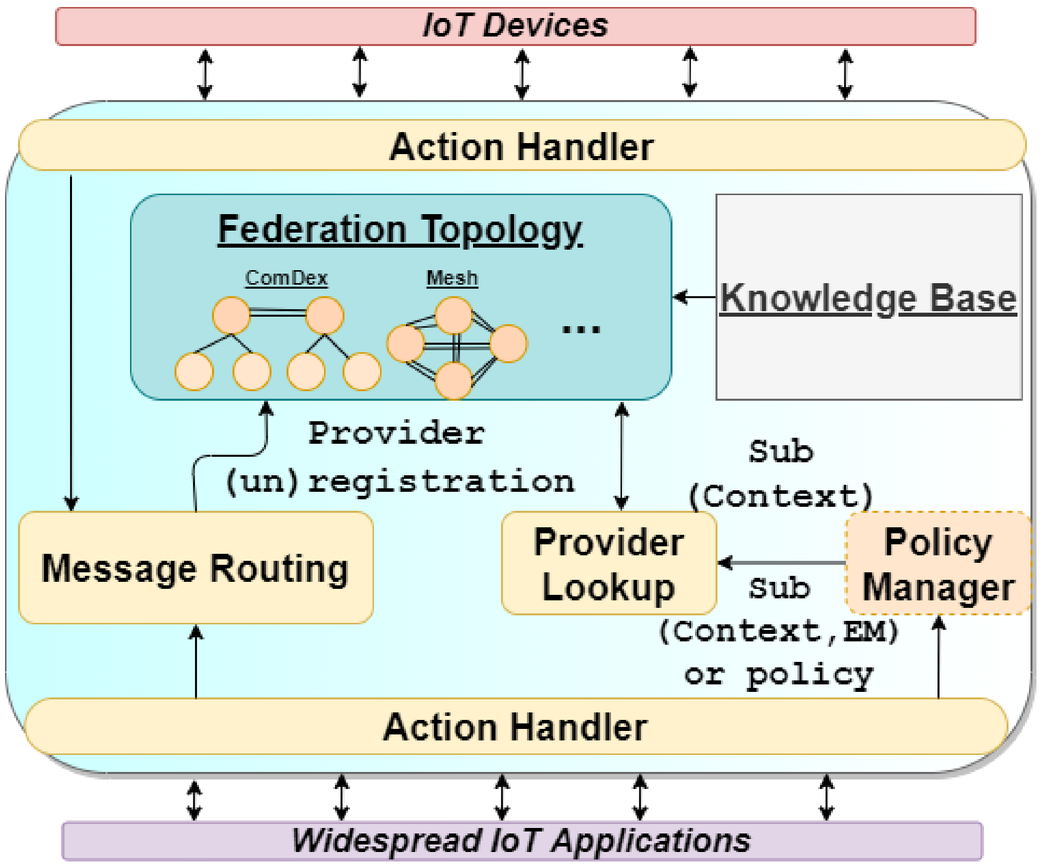


Figure 3.2: High level view of the *ComDEX* Architecture

ComDEX works with a **federation of brokers**, a group of autonomous brokers that collaborate to conduct data discovery operations. To connect and co-operate, brokers must follow a predefined **topology**. *ComDEX* proposes **a hierarchical based hybrid topology** for the system architecture. This topology is comprised of a collection of hierarchical/tree sub-topologies belonging to different data stakeholders, connected to each other at appropriate hierarchical levels based on either smart domain separation, geographical areas, or both. This topology is presented in detail in chapter 6.

The **knowledge base** is made up of schema files that specify the concepts/-data models that make up *ComDEX*'s information model, which is in turn based on property graphs. As a starting point, new data models can be developed by extending existing ones, although certain values may be redundant while other needed elements may be lacking. To produce proper digital twins specific to the scenario being modeled, the base models will need to be adjusted. Another thing the knowledge base offers is information about the current broker topology that can be used by new brokers that wish to join the system and for any potential modification. There is also an optional mapping of brokers to extra metadata, which is useful for grouping brokers based on virtual or geographical areas. For example, in Fig. 6.2 brokers B1,B2,B3 belong to the same community, and thus this metadata can be something like B1,B2,B3 belong to the smart port authority organization.

The Action Handler provides an API for the various clients (producers/consumers) to conduct diverse "Actions". "Action" can be defined as any operation inside the architecture that is necessary for the exchange of information between clients and the brokers. It offers high-level functions like data context discovery, both synchronous and asynchronous. It is responsible for executing commands and managing data flows using various sub-components, such as the **message routing component** (handles the different data flows), the **provider lookup component** (enables the discovery of remote data providers when requesting data), and the **policy manager**, which could affect the provider lookup and message routing components by taking into account certain policies, such as a change in the network or an emergency scenario. Although defined in the *ComDEX* architecture, determining how to approach the handling of different QoS requirements of applications in different scenarios (e.g. emergencies) in the general case is out of context for this thesis and part of future work. Other actions include the **publish-data action**, which is performed using the **message routing** component. Each time a content producer creates data at an edge broker, the data is stored locally at the edge, and in the hierarchical network of brokers, a **provider-registration operation** is also performed to showcase what data are available at which broker. Finally the **request/subscribe-to-data action** is performed using **the provider lookup and message routing components**. The provider lookup action is used to find where in the topology is the information required by the client, in order to route the client command to the appropriate broker for data exchange.

Action	**Components**	**Description**
Choose Action	Action Handler Main	Chooses which operation to execute and components to use according to client input.
Publish Data	Message Routing	Each time a content producer creates data at an edge broker the data is stored locally at the edge and in the hierarchical network of brokers, a provider registration operation is also performed.
Provider Registration	Message Routing	Stores and showcases which data information is available at which broker. These provider registrations can also be propagated to brokers of the topology in accordance with a federation contract.
Provider Lookup	Provider Lookup	Used by clients to find where in the topology is which data.
Request/Subscriber to Data (synchronous/asynchronous)	Provider Lookup Message Routing	Use provider lookup to find where in the topology is the information requested by the client, to route the client command to the appropriate broker to get the information requested.

Table 3.1: Summary of the various operations of the *ComDEX* architecture

Chapter 4

NGSI-LD data models for smart communities

4.1 Modeling of Smart Buildings

We found the NGSI-LD information model to be fitting to our initial *ComDEX* architecture design (see previous section 3.2). When we first started looking into NGSI-LD as a technology, one of the first things we did was look at the data models available from the NGSI-LD community, specifically the Smart Data Models project [1]. While there are many "official" data models provided by the NGSI-LD community as openly available information, a lot of them are still in an incubation stage and lack important elements. One such example was the data models related to the modelling of buildings. While buildings have been properly defined in great detail as entities with a generic model that can cover different building types, from farms to office buildings, the various sub-structural elements of a building have not been considered as a proper entity yet. The only entity that has been contemplated as of the time this thesis was written is the floor entity, which is still in an incubation stage. Since we wanted to have information that corresponded to smart buildings, we needed to extend this data model to include everything we considered missing and meet our needs. Obviously, not every building in a community will be "smart," and not every room will have devices, so the elements of a building must be modeled regardless of the presence or absence of sensors/actuators. Thus, we consider separating the modeling of smart buildings into the modeling of the structures and spaces and the modeling of the devices (sensors and actuators) present in a smart building.

4.1.1 Modeling of Spaces

The first thing is what entities we want to be able to represent with our data model. It is surely important to have a separation of the various main components

[1]https://smartdatamodels.org/

of a building to allow for easier querying of different entity types. While pondering what information one should have about the buildings, there are a few clear-cut distinct elements we want to be able to depict:

- **The building as a whole.**
- **The surrounding community where the building is located.**
- **The floors.**
- **The zones.**
- **The rooms and their sub-elements (doors, windows, and stairs).**

The digital depiction of those entities in our data model needs to contain various relationships between the different entities. These relationships should allow for easy querying of important distinct information. For example, let's say that we have a floor entity type and a building entity type. If only the relationship "Building has Floors" exists, having a floor with only its id and searching for what building that room belongs to would require searching all the buildings present in a broker and doing a content check of the "Building has floors" relationship in order to find it. By establishing bidirectional relationships wherever logical and possible, such as "Floor is in Building," one can avoid unnecessary processing. The relationships of the previously identified smart building structural components can be seen in Fig. 4.1

Let us now see the various entities of our model in greater detail, from the broadest in scope to the smallest. First we examine the community that a building is located. We have been talking about the idea of a platform for smart communities since the beginning of our work, so it makes sense that we start with this division. The ability to categorize buildings in different communities for presentation and a more distinct division of the different structures makes it vital to have as well. A real smart community would have many more buildings than the one we have in the "HasBuildings" relationship in the example of listing 4.1. Also take note that obviously the GPS position features do not match reality because the listings presented in this section deliberately do not much real information.

```
  "id": "urn:ngsi-ld:Community:Test:SmartCitiesdomain:SmartBuildings:ExampleCommunity",
  "type": "Community",
  "HasBuildings": ["urn:ngsi-ld:Building:Test:SmartCitiesdomain:SmartBuildings:3isw_NcDz2ghLEYGeHmBHm"]
  "location": {
      "coordinates": [
        [
          [25.0750599,35.3070706],[25.0704902,35.3069743],[25.0705004,35.3041812],[25.0753909,35.3044001],[25.0750476,35.3070881],[25.0750599,35.3070706]
        ]
      ],
      "type": "Polygon"
```

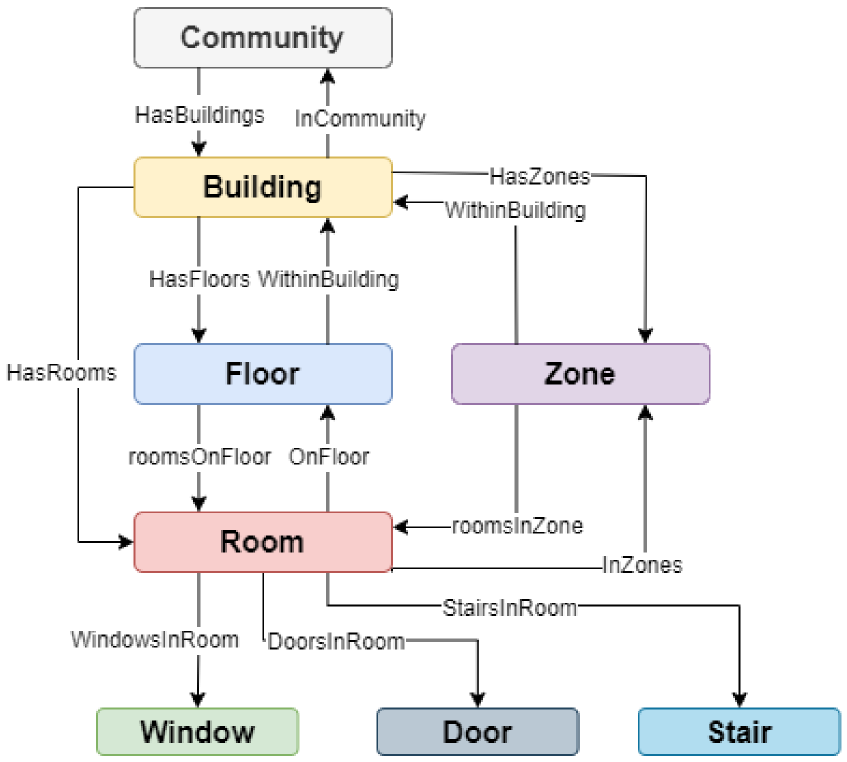


Figure 4.1: Relationships of building model entities.

```
        }
    },
    "name": "Test Building Community Area"
    ...
```

Listing 4.1: Community example

Then we have the building entity which as the name suggests is the general information about the entirety of a building. This model is pretty much derived from the building data model [2] of the smart-data-models project, with the main extensions by us being the addition of the relationships with the various entities listed before (InCommunity,HasRooms,HasZones,HasFloors). A simplified example of a building with our NGSI-LD data model can be seen in listing .4.2

```
"id": "urn:ngsi-ld:Building:Test:SmartCitiesdomain:SmartBuildings:3
    isw_NcDz2ghLEYGeHmBHm",
"type": "Building",
"name":   "Default Building",
"category": ["civic"],
"dataProvider": "ICS_Forth",
"description": "A fake office building",
"floorsAboveGround": 3,
"floorsBelowGround": 1,
```

[2]https://github.com/smart-data-models/dataModel.Building/tree/master/Building

```
"InCommunity": ["urn:ngsi-ld:Community:Test:SmartCitiesdomain:
    SmartBuildings:ExampleCommunity"],
"HasFloors": ["urn:ngsi-ld:Floor:Test:SmartCitiesdomain:SmartBuildings:38
    vC2rMpPDpQ1cy52XqxrF",...],
"HasZones": ["urn:ngsi-ld:Zone:Test:SmartCitiesdomain:SmartBuildings:0
    aJ7egOIn66uiXCDk6uTq7",...],
"HasRooms": ["urn:ngsi-ld:Room:Test:SmartCitiesdomain:SmartBuildings:1
    OKccvw796O94cljFtqofp",...]
...
```

Listing 4.2: Building example

Next we have floors and zones, which like the community entity, are brand new entities not present in the smart data models repository. Floor, as the name reflects, is the entity that models each floor of a building. Its main relationships include the building the floor is contained in and the rooms the floor encompasses. Zones are areas within a floor or across multiple floors that belong in the same category. There are multiple zoning techniques. Normally, a house (a smart home) is not divided into zones, but because we are discussing smart buildings in general, an office building, for example, could have many zones, each of which could possibly correspond to offices of the same lab group, and so on. As with the floor entity, the zone entity has a relationship that points to what building it is in and a relationship that shows what rooms are included in each zone.

The most intriguing of the bunch is the room entity . Rooms are the smallest depiction of spaces in a building, therefore having information about every room in a building means having practically most of its information—all but high-level metadata. We also regard hallways as rooms, and consider that rooms can be found in different zones. There can be a geometrical depiction of the shape of each component of a building entity. Since many buildings have a 3D representation, we permit rooms to have an attribute called ”relative position” that covers both the 3D and 2D depiction of a room.We refer to it as such because each point in the geometry corresponds to a point relative to the coordinates of its surroundings rather than a GPS coordinate (in this case, relative to the building). The room has relationships with its sub-elements (doors,stairs,windows) and with the floor and zone(s) it belongs to. There is no need to have a relationship directly with the building it belongs to, as the 1-1 relationship ”onFloor” also leads to a 1-1 relationship ”WithinBuilding”. In Listing . 4.3 we can see an example of a specific room.

```

"id": "urn:ngsi-ld:Room:Test:SmartCitiesdomain:SmartBuildings:1
    OKccvw796O94cljFtqofp",
"type": "Room",
"name": "K-3",
"onFloor": "urn:ngsi-ld:Floor:Test:SmartCitiesdomain:SmartBuildings:38
    vC2rMpPDpQ1cy52XqxrF",
"inZone": ["urn:ngsi-ld:Zone:Test:SmartCitiesdomain:SmartBuildings:0
    aJ7egOIn66uiXCDk6uTq7"],
"relativePosition": {
    "type": "Trimesh",
    "measurementUnit": "m",
    "Dimensions": "3D",
    "coordinates": [[0.35,9.68,-2.685],[0.35,9.68,-0.18000000000000016],[3.575,9.68,-2.685],[3.575,9.68,-0.18000000000000016],[3.575,9.075,-2.685],[3.575,9.075,-0.18000000000000016],[4.85,9.075,-2.685],[4.85,9.075,-0.18000000000000016],[4.85,6.055,-2.685],[4.85,6.055,-0.18000000000000016],[0.35,6.055,-2.685],[0.35,6.055,-0.18000000000000016]],
    "faces": [[2,0,1],[2,1,3],[4,2,3],[4,3,5],[6,4,5],[6,5,7],[8,6,7],[8,7,9],[10,8,9],[10,9,11],[0,10,11],[0,11,1],[4,6,8],[10,4,8],[0,2,4],[0,4,10],[9,7,5],[9,5,11],[5,3,1],[11,5,1]]
    } ,
"DoorsInRoom":["urn:ngsi-ld:Door:Test:SmartCitiesdomain:SmartBuildings:10ELxINu1AtQJwHiDxs4mK"] ,
"windowsInRoom":["urn:ngsi-ld:Window:Test:SmartCitiesdomain:SmartBuildings:1romkx9sb2mfYEttxEHWPc"] ,
"numberOfDoors": 1 ,
"numberOfWindows": 1 ,
...
```

Listing 4.3: Room example

Finally we have the "smallest" entities which correspond to various important entities present in rooms, such as windows,doors and staircases. Their model mostly consists of their geometrical representation in space (either 2D or 3D).

The Building data models presented here are publicly available at `https://github.com/SAMSGBLab/iotspaces-DataModels/tree/main/Building`

4.1.2 IFC2NGSI-LD parser

Following on from the previous subsection, we now have a way to properly represent smart buildings using NGSI-LD. But there was a small caveat in our building modeling endeavors. Do we want to manually create data models of existing buildings from scratch? Obviously, such a thing is very time-consuming and prone to human errors. For this reason, we started to lookup in what data form we could find existing buildings and what is a popular data type for buildings in general. Thus, we ended up finding the **Industry Foundation Classes (IFC)** [3] (see chapter 2) which is a digital standard for describing the building asset domain. It supports vendor-neutral, or agnostic, and usable capabilities across a wide range of hardware devices, software platforms, and interfaces for many different use cases and is an open, international standard. Consequently, we decided that a solution to quickly generating data compatible with our data models, derived from information from existing smart buildings, was to create a parser from .IFC to our NGSI-LD data models. Handling directly .IFC data from textual STEP physical files isn't easy, however, as it requires extensive knowledge of all the different entities in the IFC schema, their relationships and their hierarchies. We decided to work with the ifcOpenShell [4] library to process this format more easily. This is an open source (LGPL) software library that makes it easier for users and software developers to work with the IFC file format. The biggest advantage of using this library is that IFC entities and relationships can be queried and retrieved by name,

[3]https://www.buildingsmart.org/standards/bsi-standards/industry-foundation-classes/

[4]http://www.ifcopenshell.org/

without requiring to search around a bunch of nested elements in the original text file. For example, to get the windows present in an IFC file, a simple ”windows = ifc.by_type('IfcWindows')” function can be used. Skimming through the list of entities present in the IFC schema, one can correspond them to the entities present in our data model. This can be seen in Fig. 4.2 The inner workings of

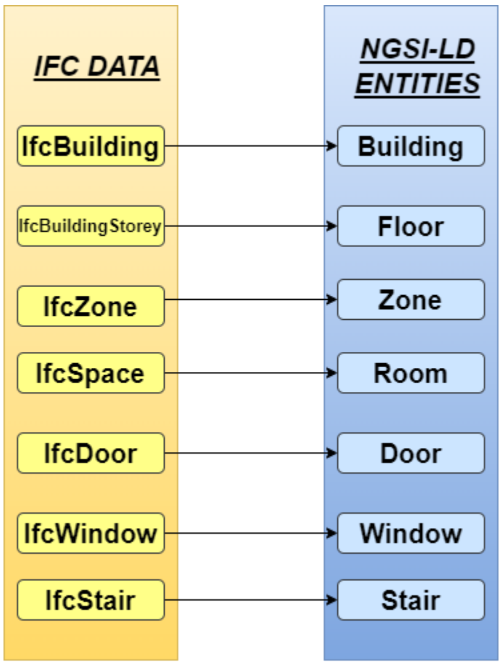


Figure 4.2: Mapping of .IFC data to ngsi-ld entities.

our parser are fairly simple. Basically, we query for every different entity that we are interested in 4.2 from the ”largest” entity ”Building” to the ”small ones” (doors,windows,stairs), we create new NGSI-LD entities and for each relationship between them or attribute they contain, we create a matching NGSI-LD property (attribute or relationship) using the function that can be seen in listing 4.4

```
def create_ngsi_ld_attribute(Dictionary,Key,Value,Attribute_type):
    if(Value!='' and Value!=[]):
        if(Attribute_type=="Relationship"):
            Dictionary.update({Key: {"type":"Relationship","object":Value}})
        elif(Attribute_type=="Property"):
            Dictionary.update({Key: {"type":"Property","value":Value}})
        else:
            print("This␣is␣an␣error␣message")
```

Listing 4.4: Creation of NGSI-LD attributes from IFC attributes

Each different entity type is then ”packaged” in the same file Fig. 4.3 (e.g all Room entities are written in a Rooms.ngsi-ld file). In hindsight, because we first created the data models and then considered the IFC format, our data model doesn't contain all the information a IFC file is supposed to contain, and this

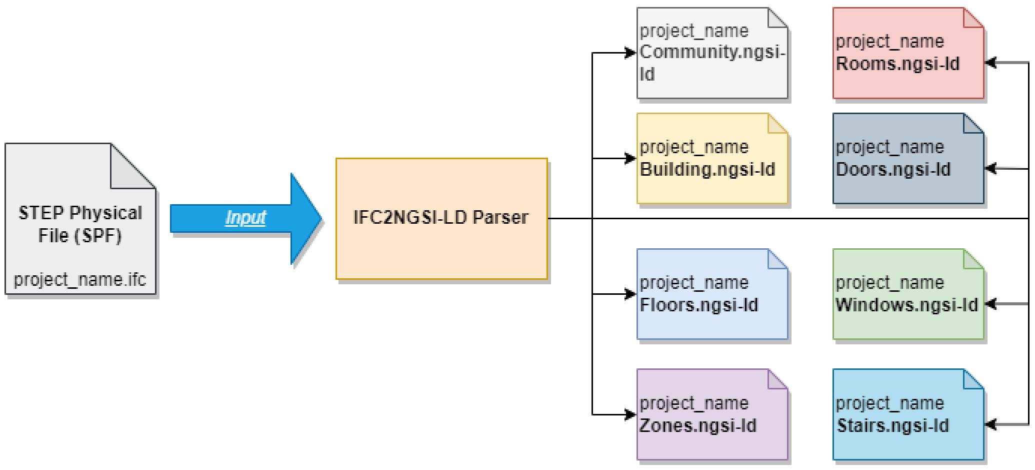

STEP Physical File (SPF)
project_name.ifc
Input
IFC2NGSI-LD Parser
project_name Community.ngsi-ld
project_name Building.ngsi-ld
project_name Floors.ngsi-ld
project_name Zones.ngsi-ld
project_name Rooms.ngsi-ld
project_name Doors.ngsi-ld
project_name Windows.ngsi-ld
project_name Stairs.ngsi-ld

Figure 4.3: IFC2NGSI-LD parser

leads to a loss of information from the IFC file. This means that while our parser is capable of converting IFC to NGSI-LD files compatible with our data model, the reverse operation would lead to a much more barebones IFC file than the original. A one-to-one mapping from IFC to a fully compliant NGSI-LD data model can be done in the future, but would require designing a much more complex data model in order to include every potential piece of information such a file might contain.

4.2 Modeling smart vehicles for public transportation systems

Smart buildings was not the only NGSI-LD data model domain we explored. Similarly to the previous NGSI-LD models, we decided to extend what we considered to be a very generic model for smart vehicles. While existing NGSI-LD models provide models for vehicles, these are too generic to be used for modeling more specific smart vehicle types, for example buses. Thus, bus transportation system developers have to arbitrarily define properties related to buses. We extend the NGSI-LD *Transportation* models by adding properties related to static and dynamic properties of the interior bus space, as well as to the space of bus stations.Our properties enable the use of existing IoT devices that can sense and actuate dynamic properties of bus transportation systems while having their observed properties decoupled. While the relationship diagrams of the smart spaces relevant to this section contain a glimpse of the device data model design, the decisions concerning the modeling of devices are thoroughly explained in section 4.3. It is very important to note that system designers can follow the approach presented in this thesis to introduce similar models for other smart vehicle types.

4.2.1 The NGSI-LD Bus model

We use the NGSI-LD *Transportation* data model [5] to extend the following entities: (i) *Vehicle Model*: to model a particular vehicle model, including all properties which are common to multiple vehicle instances; (ii) *Vehicle*: to model a specific vehicle. Fig. 4.4 presents the extended *Bus Models*[6] and their relationships with other existing as well as with new entities.

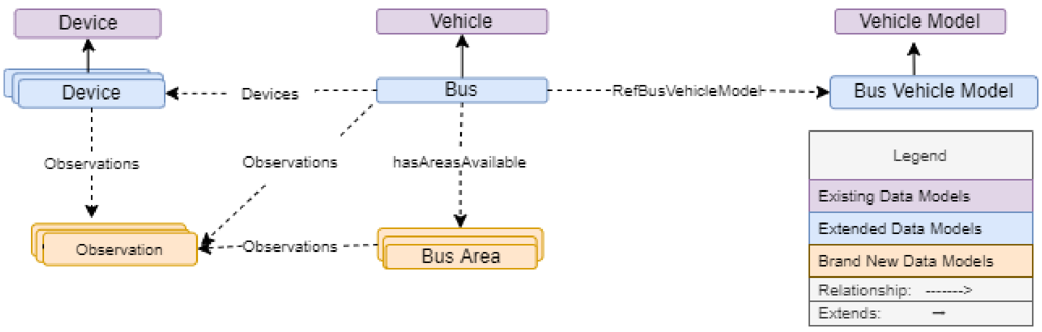


Figure 4.4: The proposed NGSI-based transport bus system data model.

The *Bus Vehicle Model* entity includes all properties (length, width, etc.) which are common to multiple buses belonging to the same model. The *Bus* entity includes properties for a specific bus (e.g., the Citaro G Centre Bus in Roma 3). To accurately model specific areas of buses, we must take into consideration the vehicle types used in the targeted region. In Europe, buses constructed for urban transportation must follow strict standards [32]. In this work, we select the *Mercedes Citaro G (3 door) version* since it is commonly used in EU bus transportation systems.

As depicted in Fig. 4.5, we first define the bus extent (i.e., its spatial properties) which corresponds to the *busExtent* property. To make this property flexible and generic, each extent can be defined using diverse coordinates such as 2D, 25D and 3D (see Listing 4.5 lines [4-9]). Then, the Bus entity has relationships to entities used to further define specific areas (*hasAvailableAreas* relationship) of the bus (e.g., driver area, seating areas, doors, etc., see Listing 4.5 lines [10-12]), as well as IoT devices placed in specific locations of the bus (e.g., WiFi, GPS, etc., see Listing 4.5 lines [14-16]). Finally, a Bus static (capacity) and dynamic (occupancy) attributes. Dynamic attributes can be associated with observations captured from IoT devices (Listing 4.5 lines [18-21]).

```
1  ....
2  "type": "Bus",
3  "name": "Centre Bus Roma 3",
4  "busExtent": {
5      "type": "Polygon",
6      "measurementUnit": "m",
```

[5]https://github.com/smart-data-models/dataModel.Transportation
[6]https://github.com/SAMSGBLab/iotspaces-DataModels/tree/main/bus-models

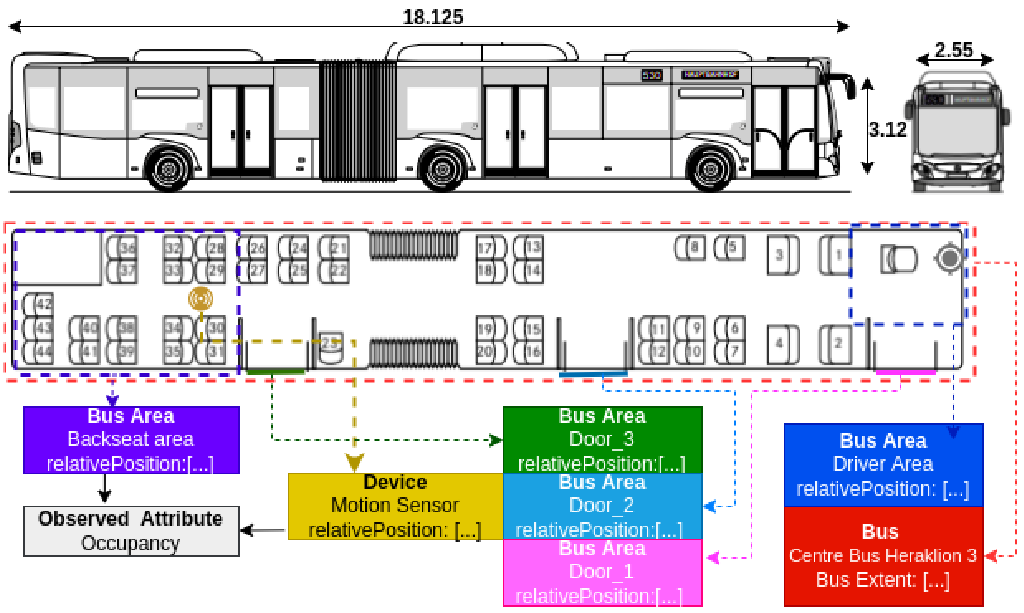


Figure 4.5: Bus areas to model their entities and attributes.

```
    "dimensions": "2D",
    "coordinates":
    [[[0,0],[18.125,0],[18.125,2.55],[0,2.55],[0,0]]] },
"hasAvailableAreas": [
    "urn:ngsild:Area:id:1",
    "urn:ngsild:Area:id:door1"],
"RefBusVehicle Model": [urn:ngsild:BusVehicleModel:YZX02],
"Devices": [
    "urn:ngsild:device_N01","urn:ngsild:device_N02",
    "urn:ngsild:device_N03"],
"capacity": "117",
"Observations": [
    "urn:ngsild:observation_N01",
    "urn:ngsild:observation_N02",
    "urn:ngsild:observation_N03"],
...
```

Listing 4.5: The NGSI-LD Bus Entity.

Bus Area[7] entities are used to define multiple areas of the bus (driver area, door area, etc.). As depicted in Fig. 4.5, such entities can be defined using the *relativePosition* property (polygon extent for the driver area, see Listing 4.6 lines [4-9]; or line extent for the door 3 area, see Listing 4.6 lines [13-20]). Finally, similar to Bus entities, bus areas have dynamic attributes that can be associated with observations (e.g., occupancy, people count, etc.) captured from IoT devices (Listing 4.6 lines [10-11]).

```
1 ...
2 "type": "Bus Area",
```

[7]https://github.com/SAMSGBLab/iotspaces-DataModels/tree/main/bus-models/BusArea

```
"name": "Driver Area",
"relativePosition": {
    "type": "Polygon",
    "measurementUnit": "m",
    "dimensions": "2D",
    "coordinates":
    [[[17.5,0],[18.125,0],[18.125,1],[17.5,1],[17.5,0]]] },
"Observations": [
    "urn:ngsild:observation_N01"],
...
"type": "Bus Area",
"name": "Door_3",
"relativePosition": {
    "type": "line",
    "measurementUnit": "m",
    "dimensions": "2D",
    "coordinates":
    [[[5.5,2.55],[7,2.55]] }
...
```

Listing 4.6: The NGSI-LD Bus Area Entity (driver example).

4.2.2 The NGSI-LD Bus Station model

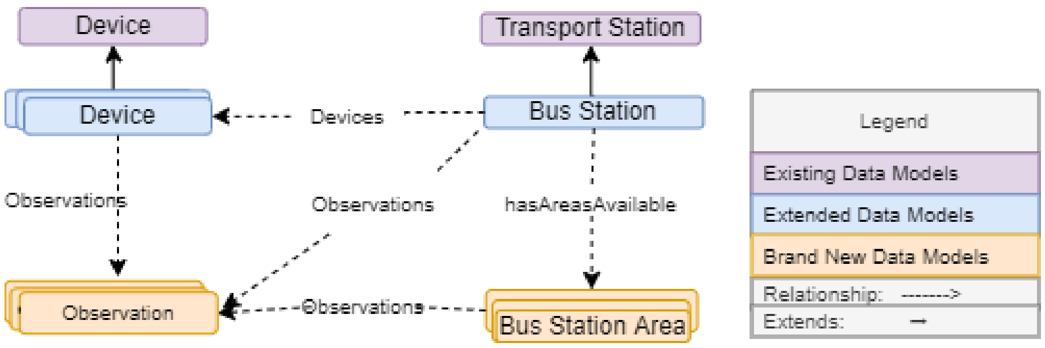


Figure 4.6: The proposed NGSI-LD bus station model.

By following a similar approach as the one presented in the previous subsections, we now extend the *Transport Station* entity of the NGSI *Transportation* data model. In a bus transportation system, this entity could represent a bus station or a bus stop and thus, we introduce the *Bus Station* entity. The extent of a bus station can be splitted to multiple areas such as bus stop, ticket booth, bus parking, etc., using the *Bus Station Area* entity. Then, static and dynamic properties can be assigned to a *Bus Station*[8] or *Bus Station Area*[9] entities. Dynamic properties are updated using IoT devices and their observed attributes.

[8]https://github.com/SAMSGBLab/iotspaces-DataModels/tree/main/bus-models/BusStation

[9]https://github.com/SAMSGBLab/iotspaces-DataModels/tree/main/bus-models/BusStationArea

4.3 Modeling of Devices

We cannot discuss smart spaces (smart buildings,smart vehicles) without considering what makes them smart, the various devices (sensors/actuators) that accompany them. The devices present in smart spaces vary in type and function [7] and many differences can be found even between devices of the same variety. For example, an environmental humidity sensor from two different manufacturers may vary in physical characteristics and the technology used to send the information (LoRaWAN,MQTT). The thing these two sensors both measure is the same. The humidity reading is the observed attribute of these 2 devices. There is a need to separate the physical device from the thing it measures or does. Developers will be able to create portable apps utilizing high-level observations rather than properties of particular devices by treating an observation of a device as an independent object. Multiple observations can also be associated with current or upcoming IoT devices. Thus, we want in our model to have a separation between the physical device properties and its observations/actuation. For this reason, we promote the observation/actuation attribute to a full-blown entity and create appropriate relationships with the device it originates from and the smart space it belongs to.

To clarify what we have been saying, let us examine a specific example of a temperature sensor in a room Fig .4.7. The physical device is obviously the device that can be seen mounted on the wall of the room. The observation of the device is the temperature reading, and the space it belongs to is the room entity. Obviously,

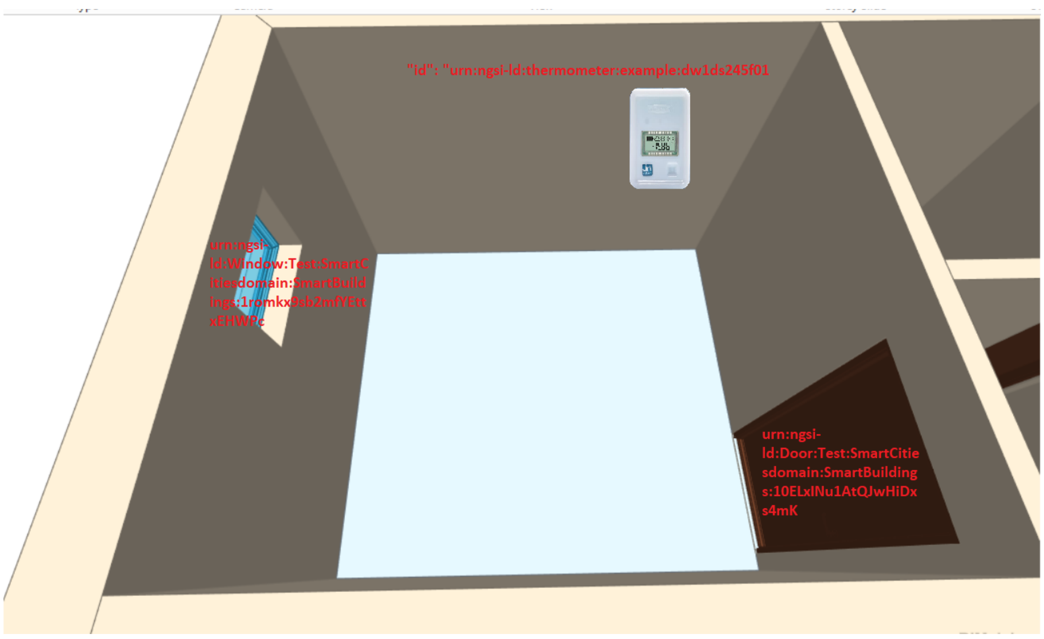

"id": "urn:ngsi-ld:thermometer:example:dw1ds245f01
urn:ngsi-ld:Window:Test:SmartCitiesdomain:SmartBuildings:1romkx9sb2mfYEttxEHWPc
urn:ngsi-ld:Door:Test:SmartCitiesdomain:SmartBuildings:10ELxINu1AtQJwHiDxs4mK

Figure 4.7: A temperature sensor on a room wall.

proper relationships need to be created between these entities. Allow us to include a camera in this room to demonstrate how multiple device, observation, and smart space relationships are handled Fig .4.8 .

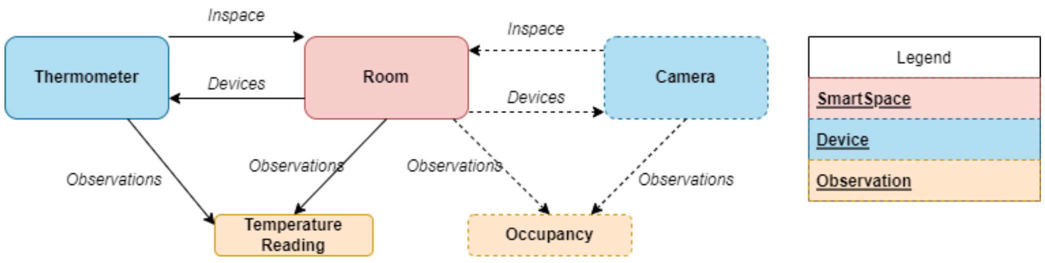


Figure 4.8: Diagram of relationships of entities depicted in Fig. 4.7.

First we have the smart space, which in this example is the room. The room has a relationship called Devices that shows which physical devices are present in the room (Thermometer, Camera). The room also has the high-level observations that are derived from the devices. These are showcased with the Observations relationship, in this case (Temperature Reading,Occupancy). The devices have the Inspace relationship, which points to the smart space the devices are in and, similarly to the room, the observation relationship. Our NGSI-LD model representations of this example can be seen in listings: (.4.7, .4.8, .4.9) If one looks back to the previous section's figures, one can see how the relationships between smart spaces, devices, and observations are handled similarly in smart public transportation modeling.

```
"id": "urn:ngsi-ld:thermometer:example:dw1ds245f01",
"type": "Thermometer",
"ControlledObservations": ["Temperature"],
"Observations": ["urn:ngsi-ld:Observation:example:obs314s12"],
"InSpace": "urn:ngsi-ld:Room:Test:SmartCitiesdomain:SmartBuildings:1
    0Kccvw796094cljFtqofp"
....
```

Listing 4.7: Temperature Sensor Device

```
"id":"urn:ngsi-ld:Observation:example:obs314s12",
"type": "Observation",
"MeasurementType":"Temperature",
"Measurement": {"value":"34","Unit":"'C"}
"description": "Measurement of the temperature of a room",
"Coverage": "65%",
....
```

Listing 4.8: Temperature Measurement of room

```
"id": "urn:ngsi-ld:Room:Test:SmartCitiesdomain:SmartBuildings:1
    0Kccvw796094cljFtqofp",
"type": "Room",
"name": "K-3",
"onFloor": "urn:ngsi-ld:Floor:Test:SmartCitiesdomain:SmartBuildings:38
    vC2rMpPDpQ1cy52XqxrF"
"inZone": "urn:ngsi-ld:Zone:Test:SmartCitiesdomain:SmartBuildings:0
    aJ7egOIn66uiXCDk6uTq7"
```

```
"DoorsInRoom": "urn:ngsi-ld:Door:Test:SmartCitiesdomain:SmartBuildings:10
    ELxINu1AtQJwHiDxs4mK"
"windowsInRoom": "urn:ngsi-ld:Window:Test:SmartCitiesdomain:SmartBuildings
    :1romkx9sb2mfYEttxEHWPc"
"numberOfDoors": 1,
"numberOfWindows": 1,
"Observations": ["urn:ngsi-ld:Observation:example:obs314s12","urn:ngsi-ld:
    Observation:example:occupancy7421wwa"],
"Devices":["urn:ngsi-ld:thermometer:example:dw1ds245f01","urn:ngsi-ld:
    camera:example:as31312d"]
 ...
```

Listing 4.9: Smart Space that houses the temperature sensor

As it can be obviously deduced, there has been the decision to not have a generic device model that covers every device but a dedicated model for each different device. For example, having a screen size attribute might be relevant for a smartphone device, but for a simple motion sensor, that would be a redundant attribute.

In addition, we consider that not every device is just a sensor of sorts, some devices could be used for actuations. For instance, there could be a panel on a smart waste bin that lights up red when the bin is almost full Fig .4.9.

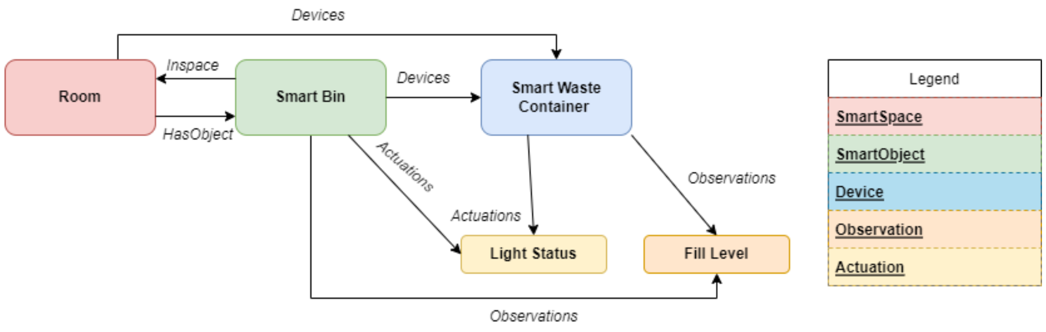


Figure 4.9: Example of smart waste bin inside a room.

Considering all the above and specifically the various relationships present in the device data model can be summarised in the following figure Fig .4.10

In essence, the observations the device makes and any applications that utilise these measurements won't be impacted by altering the physical device (which will probably lead to new device attributes). With the necessary entities for our system being modelled there was a need to define the whole architecture formally before continuing with utilizing these data models.

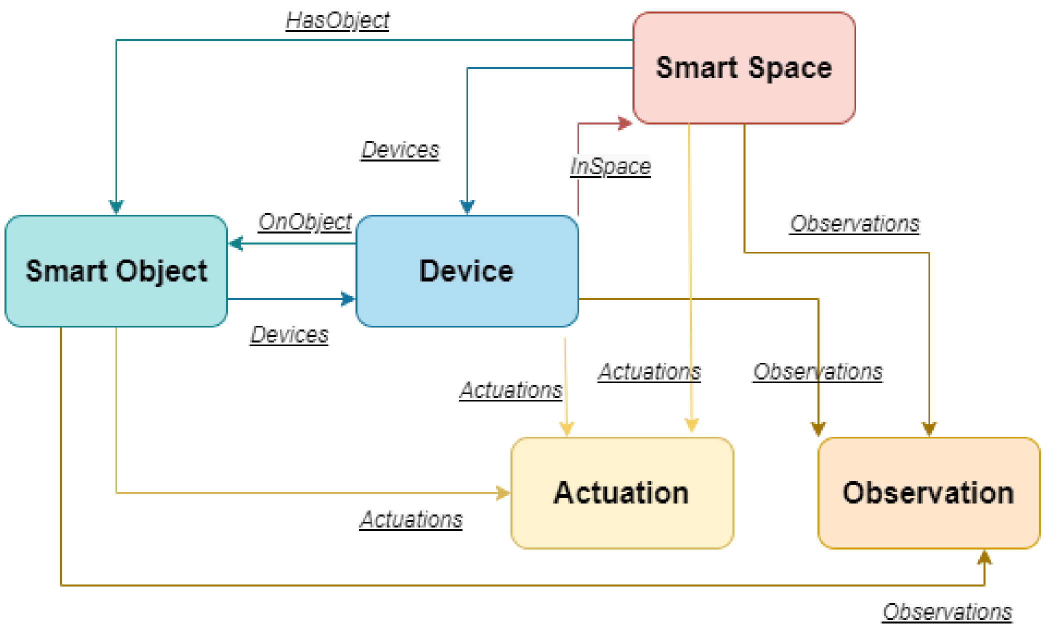


Figure 4.10: Relationship of devices in a smart space.

Chapter 5

The *ComDEX* Formal Model

In this chapter the formal model of *ComDEX* is provided, which consists of a data schema that is mapped to a publish/subscribe subscription scheme for the creation of context-aware messages. Then, the formal definitions of a federated system for the data exchange between smart communities are presented. Finally, the actions that can be performed in such a system are formally presented. In smart communities, information can be represented via separation into entities. To model entities in smart communities, *ComDEX* relies on property graphs. Property graphs [28] are a type of graph data model that focuses on a graph structure. In such data models, each graph consists of a collection of elements: vertices connected by a set of directed, labeled edges. Every element has a unique identifier and can have any number of key-value pairs called properties annotated on it [5]. A data schema is a strong data modeling feature that enables describing data structures and enforcing consistency. In this way, a graph schema allows establishing the graph structure by identifying the types of nodes, edges, and their properties.

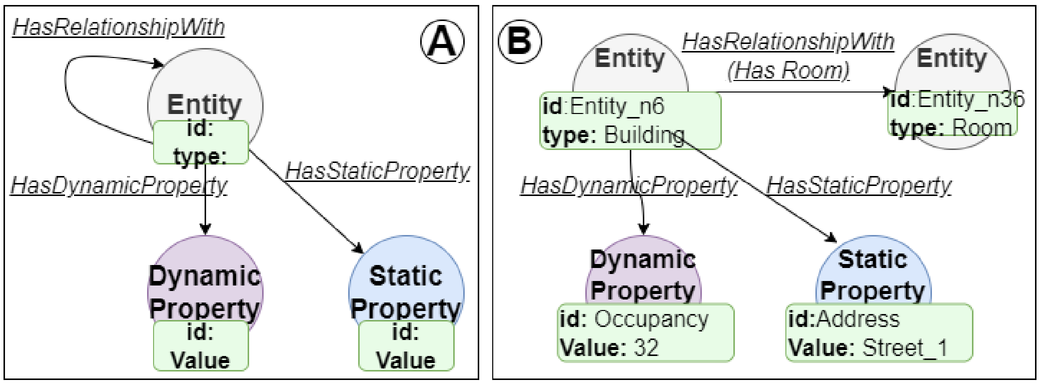


Figure 5.1: Property graph schema for smart communities

Using property graph schema, the information of an entity, is modeled using three aspects: entity details (static and dynamic attributes), entity type, and entity relationships (see Fig. 5.1A). An entity's "entity type" is a word or phrase in the information of the relevant knowledge base that indicates the entity's category information. Relationships between entities are known as "entity connections". Let $E = \{\epsilon_j : j \in [1..|E|]\}$ be the set of entities where each ϵ_j has features $\{id, type, attr\}$. Each ϵ_j's feature (e.g., type) is referred as $\epsilon_j.type$. Let $\epsilon_j.attr = \{\epsilon_j.attr.ea_i : i \in [1..|\epsilon_j.attr|]\}$ be a set of attributes with features $\{type, value\}$. $\epsilon_j.attr.ea_i.type = \{stprop, dnprop, rel\}$ is denoted as the specific types of the i_{th} attribute that an entity ϵ_j can have static properties ($stprop$), dynamic ($dnprop$) properties and/or relationships with other entities (rel). The notation of attributes for an entity ϵ_j simplified and $\epsilon_j.attr.ea_i$ is referred as ea_i. For example, suppose that a building entity exists, ϵ_{n6} shown in Fig. 5.1B with information about its occupancy (dynamic property, $ea_i.type = dnprop$), its address (static property, $ea_i.type = stprop$) and that it is related to a room (HasRoom, $ea_i.type = rel$).

In order to continue working on *ComDEX*, a specific pub/sub data detection technique had to be chosen. Different approaches of defining the data of interest have resulted in the discovery of distinct pub/sub variations [6]. The expressive power of the subscription models that have surfaced in the literature is distinguished by their ability to precisely match subscribers' interests, i.e. getting just the messages that they are interested in. In topic-based pub/sub systems, notifications are grouped in topics i.e., a context subscriber declares its interest for a particular topic and will receive messages related to that topic. Let $T = \{t_j : j \in [1..|T|]\}$ be the set of topics available in the system. By relying on pub/-sub, *ComDEX* represents entities as stored messages that have been published to certain topics. Let $M = \{m_j : i \in [1..|T|]\}$ be the set of *ComDEX* messages where each m_j has features $\{topic, payload\}$.

To enable clients performing context-aware actions (data requests and subscriptions), entities representing the context of smart communities are mapped to *ComDEX* messages. For this, it is essential to define a general rule as follows:

For every entity ϵ_j with static and dynamic properties, as well as with its relationships, a message is created and stored using Algorithm 1. In this way, *ComDEX* messages are now *context-aware*. For instance, given as input the building entity of Fig. 5.1B in Algorithm 1, messages are created as shown:

```
"message1" : {
    "topic" : "Building/Entity_n6/HasDynamicProperty/Occupancy"
    "payload" : "value:32" }
"message2" : {
    "topic" : "Building/Entity_n6/HasStaticProperty/address"
    "payload" : "value:Street_1" |
"message3" : {
    "topic" : "Building/Entity_n6/HasRelationshipWIth/HasRoom"
    "payload" : "id:Entity_n36"
```

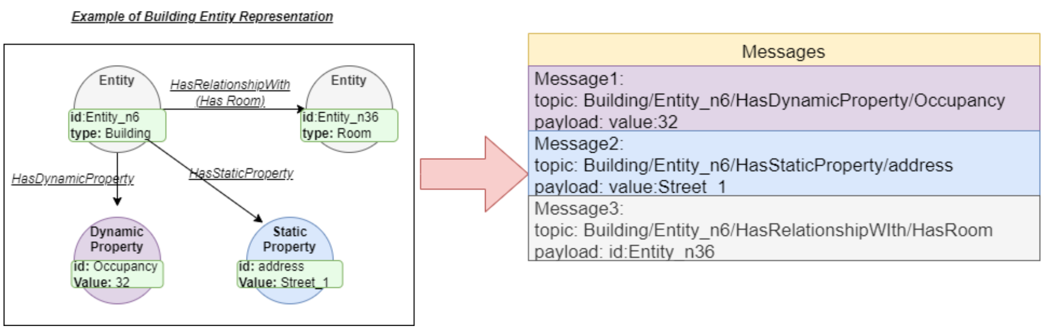
Listing 5.1: Entities mapped to pub/sub messages

Note that topics mapped from dynamic properties aim to be updated by IoT devices attached to the corresponding entity.

To enable the exchange of context-aware topic-based messages among smart

Algorithm 1 Algorithm to split of data in property graph into messages on specific topics

```
//Input: Property Graph //Output: *ComDEX* Messages
**procedure** SPLITTING P.GRAPH:
    **for each** node $x$ where $node_{type}$ equals "Entity" **do**
        $\epsilon_j \leftarrow x$
        $\epsilon_j.attr.ea_x \leftarrow edges.of.x$
        **for each** $\epsilon_j.attr.ea_x$ **do**
            $t_j \leftarrow \epsilon_j.type + \epsilon_j.id + ea_x.type + ea_x.id$
            $m_j.topic \leftarrow t_j$
            $m_j.payload \leftarrow ea_x.value$
            $print(m_j)$
        **end for**
    **end for**
**end procedure**
```



Message1:
topic: Building/Entity_n6/HasDynamicProperty/Occupancy
payload: value:32

Message2:
topic: Building/Entity_n6/HasStaticProperty/address
payload: value:Street_1

Message3:
topic: Building/Entity_n6/HasRelationshipWIth/HasRoom
payload: id:Entity_n36

Figure 5.2: Entities mapped to pub/sub messages.

communities, *ComDEX* is built based on a distributed pub/sub system where context-related components serve as context brokers, publishers and subscribers (see Fig. 5.3). Let $P = \{p_i : i \in [1..|P|]\}$ be the set of publishers that correspond to IoT devices placed in smart communities publishing entities to a set of topics. $M_{p_i} \subseteq M$ are denoted as the set of context-aware messages (i.e., entity values) that p_i publishes to a set of topics $T_{p_i} \subseteq T$ (i.e., characterized by entity types). Similarly $S = \{s_j : j \in [1..|S|]\}$ are denoted as the set of subscribers that correspond to community occupants interested in receiving messages. They subscribe to a set of topics, denoted by $T_{s_j} \subseteq T$ (including entity types, attributes and values) or perform direct on-demand queries.

A context broker can be deployed in a smart community providing context-aware messages. For the cooperation between brokers (i.e., communities), such brokers can form a federation offering routing, message management, query resolution and service discovery. Context brokers may be geographically dispersed in a real-world deployment. Multiple brokers split into administrative, network,

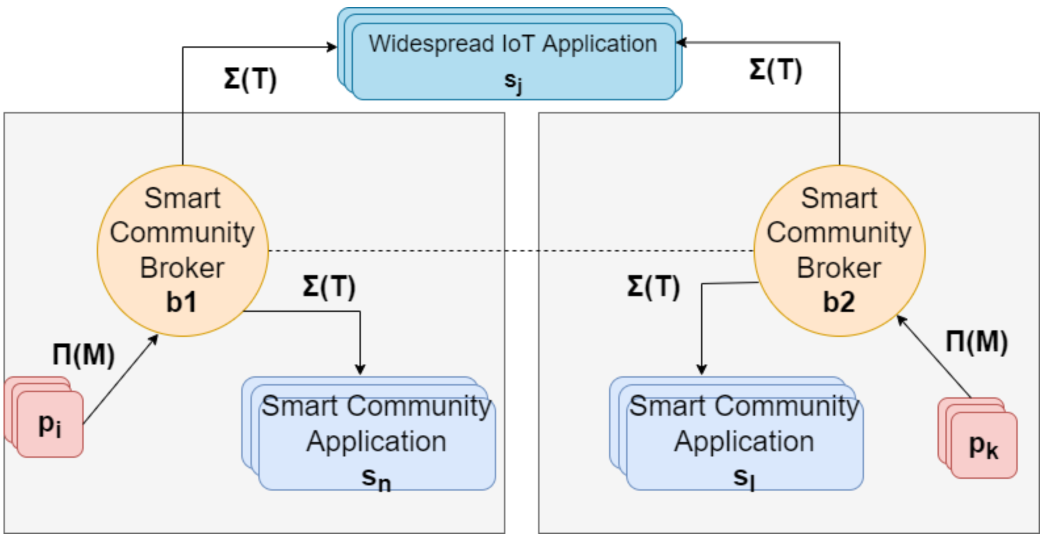

Widespread IoT Application
sj
Σ(T)
Σ(T)
Smart Community Broker b1
Smart Community Broker b2
Σ(T)
Σ(T)
Π(M)
Π(M)
pi
pk
Smart Community Application sn
Smart Community Application sl

Figure 5.3: A federation broker-based model

geographic, contextual, or load-based domains are ideal to reduce administration and communication overheads. *ComDEX* necessitates inter-broker federation so that data providers and clients affiliated with various brokers can communicate with one another. This requirement can be met using a basic message system for data relaying implemented by an overlay network of distributed brokers.

$B = \{b_k : k \in [1..|B|]\}$ are denoted as, the set of federation brokers. A broker forwards messages from publishers to interested subscribers or to other brokers (in this case advertisement messages, more details below). An assumption is made, that each publisher/subscriber connects with a single broker that it is referred to as its *home broker*: b_{p_i} in case of publisher p_i, b_{s_j} in case of subscriber s_j. Furthermore, the set of publishers and subscribers connected with b_k is defined as $P_{b_k} = \{p_i \in P : b_k == b_{p_i}\}$ and $S_{b_k} = \{s_j \in S : b_k == b_{s_j}\}$. In addition the set of Brokers connected in the federation to a b_k is defined as $B_{b_l} = \{b_l \in B : b_k == b_{b_l}\}$

In *ComDEX*, message routing between federated brokers is accomplished by relying on advertisements. Let $A = \{a_j : j \in [1..|A|]\}$ be the set of advertisements that are disseminated and stored in the broker federation. Advertisements are used by subscribers to find a broker that offers context-aware messages. An a_j is basically information about a broker that has messages published (entity types, attributes and values). Advertisements are composed from the following features: (i) $a_j.addr$ is the broker connection information-address where the advertisement was originally generated and (ii) $a_j.type$ is the entity type it advertises. An advertisement is created for each distinct entity type published in a broker. The set of topics matching an advertisement a_j is denoted as $T_{a_j} \subseteq T$ and the entities matching a_j as $E_{a_j} \subseteq E$. To make clear how many advertisements a singular broker has at most, in the case that it receives advertisements from every other broker (e.g., mesh topology), the following is defined,

$|A_{max}| = \sum_{k=1}^{B} b_k * (distinct.\epsilon_j.type : j \in [1..|E|])$. To evaluate the efficacy of *ComDEX*, the following performance metrics are defined: Let $\Delta_{gen}(a_j, p_i, b_e)$ be the generation time of an advertisement a_j by a p_i at an edge broker, b_e. Let $\Delta_{rec}(a_j, s_j, b_t)$ be the reception time of an advertisement a_j by a s_j a a top broker b_t. The time taking for an advertisement to be installed in the entire federation of *ComDEX* is $\Delta_{ins} = \Delta_{rec}(a_j, s_j, b_t) - \Delta_{gen}(a_j, p_i, b_e)$. The subscription notification latency metric is defined as the time, from the creation of a Publication π_j at a broker b_k until its reception by an interested subscriber s_j

When *ComDEX* federation is setup (e.g., brokers in smart communities), community inhabitants can leverage the provided IoT devices and applications. A set of diverse ***actions*** is used to push/pull or subscribe to receive context-aware messages in the smart community (see Fig. 6.1). To enable the registration of context information (buildings, vehicles, etc.), community inhabitants can advertise the entity types that they provide to their home broker. Let $REG(a_j)$ be the *provider registration* action which creates an advertisement to a context broker. Then, publication actions including the advertised information may be performed. Let $\Pi = \{\pi_j : j \in [1..|\Pi|]\}$ be the set of publications with context-aware messages related to the smart community (e.g., Buildings of the port community). Publications are assumed to be generated from real-world activity or traces of existing IoT devices deployed in smart communities.

Now the definition of actions used from data subscribers willing to discover entities in a smart community is presented. Let $LUP(a_j)$ be the *provider lookup* action with which a context subscriber can discover what kind of data is available and where. A subscriber performs this using the stored advertisements in its local broker. Having discovered the community entities, a subscriber can setup a subscription action which defines the required instances of entities and the validity period of the subscription. Let $\Sigma = \{\sigma_i : i \in [1..|\Sigma|]\}$ be the set of subscriptions. To satisfy a subscription, each broker filters the entity type generated by the publishers based on the topics locally, and forwards the subscriptions to other community brokers that have relevant information according to the provider lookup action. Subscriptions are usually made to entities with dynamic properties (e.g., the temperature of a room). To receive the messages stored in brokers and especially entities with static properties,the data request action is defined. Let $D = \{d_i : i \in [1..|D|]\}$ be the set of data requests. This is the action where a subscriber requests data synchronously from its home broker. Similarly, to satisfy the request the broker will also forward the inquiry to community brokers that have relevant information according to provider lookup.

Having defined the *ComDEX* system model, the basic design principles and IoT platform requirements presented in chapter 3 have been matched and thus proceeding a working Context-aware Federated Platform can be created for IoT-enhanced Communities.

Chapter 6

The *ComDEX* Federation Topology

To design *ComDEX*'s federation topology, the displayed architecture solution is based on advertising context-aware messages in the federation.

Existing pub/sub distributed systems usually rely on naive event/message flooding across the broker overlay. In *flooding-based routing* [6], each message from the publisher is propagated to all the brokers in the system. *Filtering-based routing* [6], in which subscription tables are shared across a collective of brokers, cannot be used for this situation since it assumes a push-only based architecture. While a publisher using the subscription tables can route its information to interested subscribers, a context subscriber that wants to get the latest state of a message stored in a remote broker, would have to subscribe and wait for a publisher to push the message to the topic of interest. A rendezvous-based algorithm [6] is also not suitable, since it does not handle well dynamicality over a broker-based architecture.

In *ComDEX*, instead of subscribers conveying the topics they are interested in, each broker advertises the context-aware messages it can provide. When a new message is pushed, the *ComDEX* action handler creates a *provider advertisement message*. Brokers forward these advertisements subject to topic restrictions to allow broker owners to limit what advertisements are propagated and where.

Using Fig. 6.1, how the federation model operates is now described for the following sets of brokers: (i) one broker; (ii) two brokers and; (iii) n brokers. $E_{b_k} \subseteq E$ are defined as the entities stored at a broker b_k; $A_{b_k} \subseteq A$ as the advertisements stored at a broker b_k; and $FWD(a_j, b_k, b_l)$ as the action with which a broker b_k forwards an advertisement a_j to one of its connected brokers $b_l \in b_{b_k}$.

For the trivial case of a single broker, consider that the local subscriber s_1 of broker b_1 requests messages via a subscription on topic t_1. A publisher p_1 publishes j messages of various entity types in different topics, and creates appropriate advertisements (if they do not exist) using the provider registration action. When a message is published to a topic and it matches the subscription, it notifies the

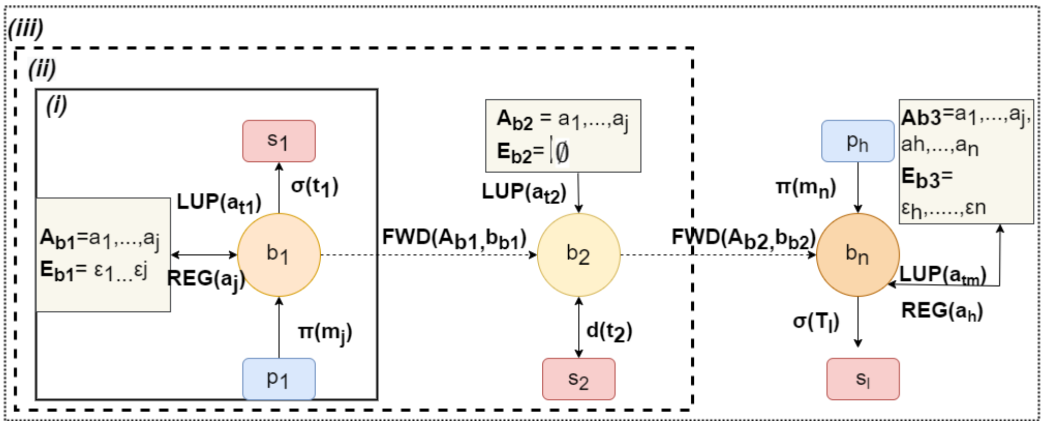


Figure 6.1: Actions in a number n of connected brokers

subscriber and sends it the match context-aware messages.

For the case of 2 brokers, consider that there is one more broker (b_2) connected to the previous broker and a context subscriber s_2 which requests messages using d_i on a specific topic (i.e., entity type) T_{s_2}. Advertisements are now propagated along connected brokers. When the data request action is performed in b_2, the broker first checks for available messages stored locally that match the request. Then, using the provider lookup, b_2 searches for advertisements matching the messages requested. If a matching advertisement is found, then it forwards the data request to the relevant broker. If matching messages are found in b_1, these are sent to s_2.

Finally, for the case of n brokers, consider an arbitrary number of brokers connected along with the previous ones. When a message is published to b_1, its matching advertisement is propagated along the federation of brokers. Consider now a subscriber s_m and a publisher p_h connected to b_n. Since the advertisements are forwarded in one-way, from "left to right", clients connected to broker b_1 are not aware of the available messages in b_n. Similarly, a subscriber s_m becomes aware of the messages stored at b_n and the provider of messages that matches their subscribed topic, through the stored advertisements at broker b_n.

Distributed solutions are better than centralised ones in big cross-community deployments, as they help with: flexibility, since brokers can be inserted and removed on the fly; modularity, as a stakeholder modifying their broker set will not affect the entire system; and privacy aspects of a system, since the data is maintained in their owner's machines. As for topology [14], federated systems are typically shaped as a tree, star, or mesh with bidirectional linkages (implemented as two unidirectional links) connecting any two brokers. A ring topology is also viable if just unidirectional linkages are employed. As the number of "hops" affects the time to send and receive messages, in most cases for optimum performance the number of brokers between the message origin and the final destination should be kept to a minimum. In a mesh topology, every broker is connected to one another via dedicated direct connections. In a star topology, all brokers are connected to a single broker. The tree topology can be seen as a variation of this (multiple star topologies) that has a hierarchical flow of data. As already mentioned in section 3, *ComDEX*'s topology is a hierarchical-based hybrid topology. Having hierarchies when exchanging data flows is convenient as it resembles the reality of smart communities, -e.g., a community has buildings, a building has floors, etc. In addition, context information can be separated in a privacy-preserving manner - e.g., each community chooses what data to advertise to the upper levels of communities. The proposed topology maintains the disadvantages inherent to a tree/hierarchical topology. As the system becomes more complex and the number of "hierarchical levels" increases, so does the number of hops for the traversal of information. As levels increase, so does the complexity of handling the entire system. If a node at one level is erroneous, nodes at higher levels could face issues as well. Having multiple nodes in order to separate the data flows can also be quite costly. On the other hand, a mesh topology would require that every data provider in a community share its (possibly private) information with every other community, which is not realistic. Lastly, as the complexity and number of brokers increase, adding new brokers to a mesh topology becomes a complex task.

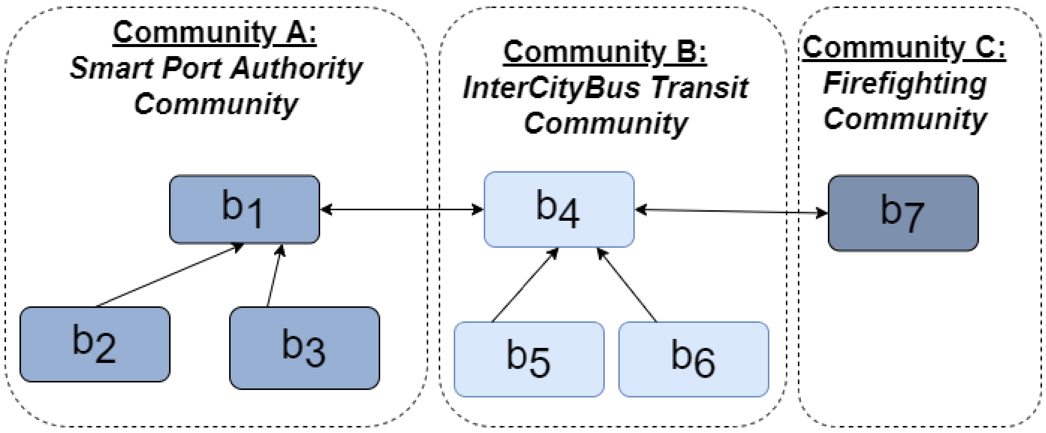


Figure 6.2: The *ComDEX* Topology

In Fig. 6.2 a generic example of the proposed topology can be seen, matched with the port application example from section 3. Community A is the *Smart Port Authority Community*, with a central broker b_1 and two sub-brokers b_2 and b_3 that cover different geographical areas in the smart port. Community B is the *InterCityBus Transit Community* and the separation between brokers is by smart domain, e.g., b_5 as the smart buildings broker, b_6 as the smart transportation broker, etc. Lastly, community C is the Firefighting Community, with broker b_7 its central broker. Advertisements are forwarded according to connections, e.g., b_2 forwards to b_1, b_1 forwards to b_4.

The algorithm for advertisement propagation in the federation (Alg. 2) is pretty straightforward: Suppose there is a broker b_k that has a set of directly connected brokers b_{b_k} in the federation (e.g, $b_1 \rightarrow b_2$ in Fig. 6.2). Broker b_k has a process that runs in parallel with all its other operations, which simply waits for an advertisement a_j to arrive. When an advertisement arrives at the broker, it checks all its connections with other brokers. These connections may have restrictions: For example, b_7 could only want to exchange information about specific entity types (e.g., smart_building), not all its information. It checks if the advertisement, in each connection, passes the connection's restriction, and if it does, the advertisement a_j is propagated to the appropriate connected brokers.

Algorithm 2 Algorithm for advertisement management by broker

```
1: procedure ADVERTISEMENT PROPAGATION AT $b_k$:
2:     while 1 do
3:         $a_j \leftarrow wait.for.advertisement.arrival()$
4:         $A_{b_k} \leftarrow A_{b_k} + a_j$
5:         for each $b_l \in b_{b_k}$ do
6:             $pass \leftarrow Restriction.Check(a_j, b_l)$
7:             if $(pass == TRUE)$ then
8:                 $FWD(a_j, b_k, b_l)$
9:             end if
10:        end for
11:    end while
12: end procedure
```

The main advantage of the presented approach over a purely hierarchical one is the ability to connect high-level brokers of different smart communities while treating them as "first-class citizens" (the existing hierarchy of the highest in each community broker does not change) and avoiding creating new higher-level nodes in the hierarchy. For example, b_1, b_4 and b_7 (highest in hierarchy brokers in each community) can be connected directly without necessarily having to create a new b_8 broker at a higher hierarchy level to connect them. Applications of existing communities will discover new entities without having to connect to a new central broker. It is important to note that if no bidirectional connections exist throughout the topology (i.e. see community A sub-topology as the whole topology), it is no different than the standard hierarchical topology, since that would mean that there is a root node.

Chapter 7

Prototype Implementation

In chapter 5, *ComDEX* 's model was defined as a property graph of entities that are mapped and stored using a topic-based messaging scheme. This model was implemented, taking as a basis the NGSI-LD specification [1], which combines linked-data entities with property graphs and defines an API that covers most actions required by *ComDEX*. MQTT was chosen as a lightweight topic-based pub-sub message communication protocol to use in *ComDEX* due to its broad deployment and acceptance in IoT applications. The *ComDEX* implementation includes a lightweight NGSI-LD federated broker harnessing open-source MQTT brokers at its core. This prototype improves upon existing heavyweight NGSI-LD brokers. This is because the NGSI-LD specification currently favors HTTP as a communication protocol with NGSI-LD CRUD operations naturally mapping to HTTP verbs. This means that current NGSI-LD brokers lack end-to-end MQTT capabilities such as QoS delivery guarantees (forcing them to use IoT-Agents for MQTT compatibility).

To enable context-aware data exchange among smart communities, *ComDEX* implements a federated architecture. The NGSI-LD specification aims to support federated and distributed broker topologies through definitions of *context sources* and *context-source registrations*. Context-source registrations include details on the types of context information a context-source can provide, but not actual values. While the context-information API operations are supported by many existing NGSI-LD brokers, context-source registrations and discovery operations are not yet there. Without data discovery and forwarding, federation topologies cannot yet be supported by existing NGSI-LD implementations. What is more, the different solutions put in place by current NGSI-LD broker implementations, still under heavy development, could be dismissed by the NGSI-LD community if/when an official forwarding solution is defined. As *ComDEX* proves however, the implementation of federation in an NGSI-LD service naturally fits and can be embedded into a native MQTT topic-based architecture.

The *ComDEX* prototype implementation is provided as an open-source platform at `https://github.com/SAMSGBLab/\textit{ComDEX}--ngsi2mqtt`.

7.1 NGSI-LD as *ComDEX*'s information model

Recall that the *ComDEX* information model consists of modeling entities, their static and dynamic properties, and relationships using property graphs. The NGSI-LD information model is a good fit for it as it also derives from property graphs [27]: The core element is the *entity* (known as a *node* in property-graph language), which corresponds to a real-world concept. Every entity must have a unique identifier, which must be a URI (typically a URN), as well as a type, likewise a URI. This URI should point to a Web-based data model. Entities are associated with *properties* and *relationships*. To enable context awareness, each property's name should ideally be a well-defined URI that corresponds to a widely used notion on the Web. This knowledge graph is well-defined and infinitely expandable. Property graphs are versatile, scalable representations that have been widely used in the IT industry.

7.2 NGSI-LD to MQTT mapping

Moving from theory to practice in mapping from a property-graph data representation to MQTT topic messages raised a number of design choices. From the NGSI-LD specification, it has indeed been understood that API "Actions" can be mapped to the previously presented architecture. However, to ensure an efficient implementation the *granularity* of various NGSI-LD API commands had to be considered.

NGSI-LD data entities can be managed via a namespace structured hierarchically as `/NGSI-LD/v1/entities/<entity-id>/attrs/<attr-id>`, offering four endpoints upon which one can apply fine-grain or coarser-grain CRUD actions (e.g., modify an individual entity or attribute, or query all entities of a certain type). NGSI-LD context information API operations are divided into two main categories: *provision* (creating entities and modifying their attributes) and *consumption* (querying and subscribing to entities).

All NGSI-LD entities necessarily have an id, a type, attributes that describe the entity, and what is called "context", derived from JSON-LD. A context is a URL that points to the description of the data model used/requested and is used to expand and compact the shortnames that are part of the payload data. Requesting entities of the short-type "vehicle", for example, without specifying the context in which the vehicle entity is described will not work.

To describe the scheme for mapping NGSI-LD operations to MQTT and how NGSI-LD data map to MQTT messages the following example from a smart-city application reporting air-quality (entity) is used, with three attributes, the level of NO_2, the date it was observed and a reference to a point of interest:

```
NGSI-LD entity:
{
    "id": "urn:NGSI-LD:AirQualityObserved:RZ:Obsv4567",
    "type": "AirQualityObserved",
    "dateObserved": {
        "type": "Property",
        "value": {
            "@type": "DateTime",
            "@value": "2018-08-07T12:00:00Z"
        }
    },
    "NO2": {
        "type": "Property",
        "value": 22,
        "unitCode": "GP"
    },
    "refPointOfInterest": {
        "type": "Relationship",
        "object": "urn:NGSI-LD:PointOfInterest:RZ:MainSquare"
    },
    "@context": [
        "https://schema.lab.fiware.org/ld/context",
        "https://uri.etsi.org/NGSI-LD/v1/NGSI-LD-core-context.jsonld"
    ]
}

-----------------------------------------------
MQTT Messages:
"message1":
topic: unknown_area/entities/https%3A%2F%2Fschema.lab.fiware.org/
    AirQualityObserved/urn:NGSI-LD:AirQualityObserved:RZ:Obsv4567/
    dateObserved
payload:
"type": "Property", "value": {
        "@type": "DateTime",
        "@value": "2018-08-07T12:00:00Z" }

"message2":
topic: unknown_area/entities/https%3A%2F%2Fschema.lab.fiware.org/
    AirQualityObserved/urn:NGSI-LD:AirQualityObserved:RZ:Obsv4567/NO2
payload:
"type": "Property",
        "value": 22,
        "unitCode": "GP"

"message3":
topic: unknown_area/entities/https%3A%2F%2Fschema.lab.fiware.org/
    AirQualityObserved/urn:NGSI-LD:AirQualityObserved:RZ:Obsv4567/
    refPointOfInterest
payload:
"type": "Relationship",
        "object": "urn:NGSI-LD:PointOfInterest:RZ:MainSquare"

"Advertisement":
topic:provider/broker_address/broker_port/https%3A%2F%2Fschema.lab.fiware.
    org/AirQualityObserved
```

Listing 7.1: NGSI-LD entity message separation

An entity in NGSI-LD format is mapped to a number of MQTT messages, one for each attribute as shown in Listing 7.1. In this example, the name of the MQTT topic for each message is `(area + '/entities/' + context + type + id +'/'+ attribute)` and the value of the attribute is the payload of the message. The term "area" defines (in a human-friendly way, as a simple string) the geographical scope of the broker, for example, CommunityA_CityB. Each Action Handler-Broker pair is assumed to be aware of the name of the "area" it covers. The area value is nonetheless optional and may be omitted. The string "entities" is just to convey that that the information described by this message is part of an entity. This is useful as a self-reference to NGSI-LD endpoints in the development and maintenance of the prototype. It is also useful to determine when information that does not correspond to entities is inserted at a broker, as in the future information about context other than entities might be available (e.g. which subscriptions are currently active in the broker). Context, type, id, and attribute are all (one-to-one) parts of the entity described above. When requesting for entities, messages that fit a specified request are recombined back into whole parts using the entity-id part of the topic to recognise that a message represents the same entity. This is the reverse of the data split. Since there is a message for each different attribute, operations can be performed at the attribute level as needed.

In general, the aim was to achieve close compliance with the latest NGSI-LD specification. This means supporting pull-based applications (i.e., those issue GETs for data) as specified by the NGSI-LD API. This raised an issue as in MQTT typically if a publisher publishes a message to a topic with no-one subscribed to it, the message is simply discarded by the broker. Thus MQTT brokers had to be configured to store entities by having the data publisher instruct the broker to keep the last message on that topic (setting the retained message flag). Deletion of these messages can be then performed with an empty-payload message.

In what follows the implementation of key NGSI-LD API commands, also known as Actions, are described in detail: **Creation of an entity** happens with the "POST" entity command (PUSH DATA Action,Provider Registration Action) to an NGSI-LD broker, in the format: `'POST' 'http://broker-address/:port/NGSI-LD/v1/entities/' 'entity_data.file'`. The prototype uses a similar format for the selection of every possible command, e.g `python3 actionhandler.py -b broker-address -p port -c POST/entities -f 'entity_data.file'`. The action handler first checks if the command/action inserted is valid (here, the POST/entities command is a valid command). Next, it tries to connect with the broker specified to check if the file given for entity creation is valid JSON, and if necessary parts of an entity (id, type, context) are included. Then it checks if the entity with the specified entity-id already exists in the broker. If so, it notifies the publisher. An *advertisement*[1], a notification of new context information available in a federated *ComDEX* service, should also be created for this entity if it does not already exist for this broker. A new advertisement is created and published with the topic `"provider/' + broker_address + broker_port + /area + /entity context + /entity type + (/entity id)"` according to the configured advertisement granularity (§6.3). Finally, for each different attribute present in the entity a new MQTT message is created (Listing 7.1), with payload the content

[1]The creation of *advertisements*, an important aspect of the federation solution implementation, is thoroughly covered in section 6.3

of each attribute and the topic `area+'/entities/' +entity_context+entity_type+entity_id+'/'+ attribute`. To also enable temporal queries for each attribute, messages for attributes *createdAt* and *modifiedAt* are created.

Listing 7.2 provides pseudocode of this process.

```
def Action_handler(argv):
    if(command==POST/entities):
        connect(broker,port)
        file=open_file("entity_data.file")
        if(is_valid_json(file) and is_valid_ngsild(file):
            if(entity_exists(entity.id)):
                print("already␣exists")
                exit()
            if(!advert_exists(broker,port,entity.context,entity.type)):
                publish(advertisement)
            for each attribute in entity_file:
                publish(attribute.message)
                publish(attribute.time_message)
            close_connection()
        else:
            print("invalid_file")
            exit()
```

Listing 7.2: POST/entities command

Creation of a subscription happens with a 'POST' (GET Data Action, Provider Lookup Action) `'http://broker-address/:port/NGSI-LD/v1/Subscriptions/'` 'subscription.file' , which follows the same logic as the creation of an entity with some additional complexity. A client interested to find information about entities must check the advertisements to find where this information is located. Thus, in the creation of a subscription the action handler first checks if the command inserted is valid and then if the file given for entity creation is valid JSON with all parts of a subscription (at least one of the following: id, type, watched_attributes) are included. It then tries to connect with the specified broker to subscribe to the provider messages of the broker that correspond to the subscription file. For every provider-message/advertisement that is received and has not been received previously, a new parallel connection/subscription is created to the broker that advertised that has relevant information. If an advertisement is deleted while a subscriber is still connected, terminate the relevant subscription. The received data is recreated accordingly, and exit is performed if a subscription exceeds a timeout.

Listing 7.3 provides the pseudocode of this process.

```
def Action_handler(argv):
    if(command==POST/Subscriptions):
        file=open_file("subscription.file")
        if(is_valid_json(file) and is_valid_ngsild_sub(file):
        subscribe_for_advert_notif(broker,port,sub_data)

def subscribe_for_advert_notif(broker,port,sub_data):
    known_adverts=[]
    connect(broker,port)
    subscribe(advertisements)
    on_message():
        advert=message.topic
```

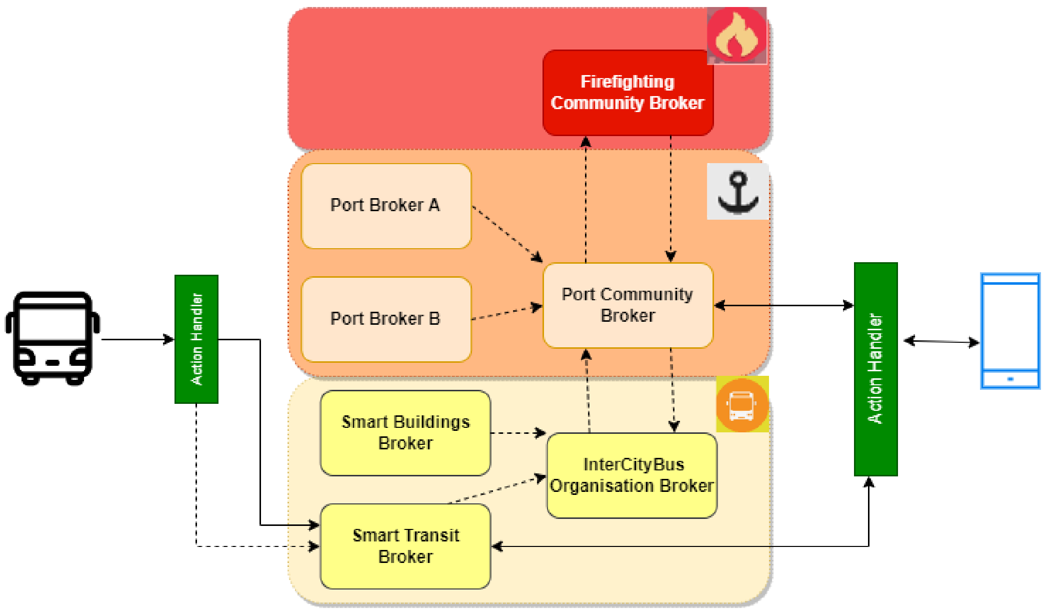


Figure 7.1: Smart Transportation Application using *ComDEX*

```
        if(message.payload==Null):
            known_adverts.remove(advert)
            close_appropriate_connection()
        if(!advertisement_already_known(message.topic)):
            known_adverts.add(advert)
            create_sub(advert.broker,advert.port,sub_data)
```

Listing 7.3: POST/Subscriptions command

The implementation of *ComDEX* prototype is in Python. While a C implementation may have resulted to a more efficient system, with python a fully functional federated NGSI-LD broker was able to be prtotyped according to *ComDEX* design principles. The prototype is nonetheless easily re-targetable to other programming languages.

7.3 *ComDEX* advertisements and bridging

The implementation of broker federation is based on advertisements of exported context, which (just like entities) are propagated as MQTT messages. An advertisement message is composed of a topic in the following form: *(provider/' + broker_address + broker_port + area + NGSI-LD entity context + NGSI-LD entity type)*, in accordance with the mappings described in §7.2. The *'provider/'* string at the start of the topic is used to denote a data-provider advertisement message. The *broker address* and *broker port* identify the broker that has the data described by the *context* and *type*, and *area* (a string) is the region the data provider covers. A client interested in specific context information uses the advertisements available at the broker it initially connects with to create new connections and get all the information it is interested in, which could be at a remote broker. The subscription is described in §7.2 and formally in §5. Advertisements follow paths summarily to Fig. 6.1. Advertisements may be created for new each entity type, or (at a finer grain) for each new entity id. The prototype allows selection of advertisement granularity (type or id). Advertisement per-type is currently the default.

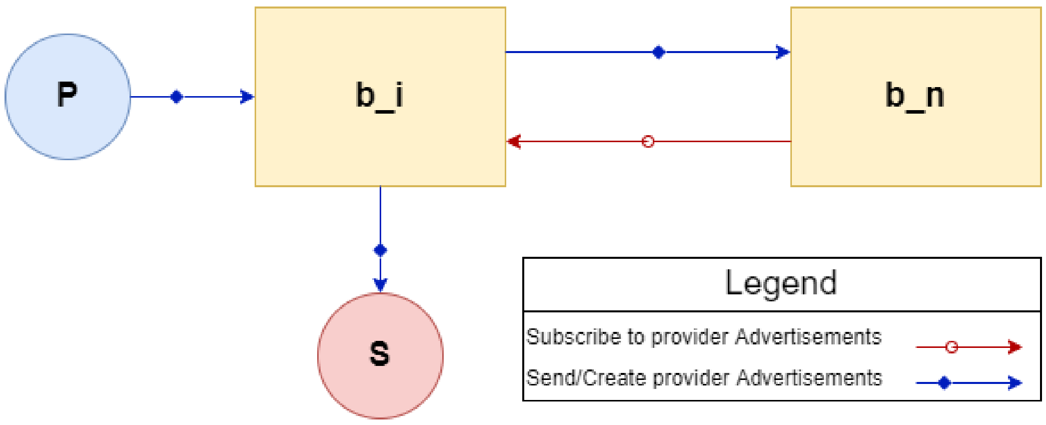

P
b_i
b_n
S
Legend
Subscribe to provider Advertisements
Send/Create provider Advertisements

Figure 7.2: Advertisement messages paths.

Provider messages are propagated throughout the federation of brokers with the use of **MQTT bridges**. For example, 'topic provider/# out 2 "" ""' can be used as a bridge that propagates all the provider messages that are created at the broker to the address specified. Filtering the data that moves between each bridge is also an important part of this type of bridging. Care should be taken when creating MQTT bridges, as bidirectional linkages can produce redundant message transmissions and overload the federated network. Using the bridge 'topic provider/# both 2 "" ""' from a b_A to a b_B, is an example of a bad bidirectional bridge as b_A would send a message to the b_B and vice versa. An alternative to the automated advertisement system described would be to perform manual context source registration by simply creating advertisements manually (akin to what NGSI-LD dictates) according to the context source information they wish to "register". The showcased automated advertisement system has clear usability advantages leading to an all-around functional NGSI-LD broker federation.

7.4 Smart transportation in the port scenario

The *ComDEX* prototype implementation can support deployment of the smart transportation motivational scenario (section 3) depicted in Fig. 7.1. In this scenario, the brokers of each stakeholder (firefighters, port community, InterCity buses) are connected in a hierarchical manner with the use of MQTT bridges, with high-level brokers "seeing" all the information-provider sources advertised to them via these bridges. In the interest of simplicity, let us assume there is only one data publisher and only one subscriber.

The data publisher (e.g., a bus GPS) sends NGSI-LD data about a vehicle to the action-handler-broker (SmartTransit). The action-handler converts the NGSI-LD information into appropriate MQTT messages and sends them to the MQTT broker. The brokers can be any MQTT compliant broker (for example, Mosquito). At the same time, it checks if an entity of type vehicle (a type defined within the smart transportation data context) has already been advertised to this broker. If not, it creates a data-provider-advertisement message and pushes it to the broker as described in §7.2.

The broker receives the messages and through bridging, forwards the advertisement to the IntercityBus and on to the Port Community Broker, where it is stored. The paths that advertisements can traverse are visible with dotted lines in Fig. 7.1. A subscriber (e.g., a mobile application) wants to receive information about all the buses available throughout the federation, and it happens that the closest broker is the Port Community Broker. It issues the relevant NGSI-LD query to the Port Community Broker with the use of the action handler, which checks the query and searches for provider advertisements matching the entity-type requested and, if specified, the location of an entity or area of interest (e.g., Port Area A). The interface finds out that the InterCityBus Broker has information about buses, so it connects to it to receive the relevant data. The messages are structured back as NGSI-LD entities; ultimately, the information requested is returned to the subscriber.

7.5 *ComDEX*'s resilience to failure

While fault tolerance isn't directly explored in the scope of this work, there are numerous ways a system designer can connect the various components in their community and a federation of communities to as mentioned in §9, enable resistance to system errors if a node or multiple, for some reason, terminate, are unresponsive, or their network is unreachable. This can be achieved using backup MQTT bridges. The clients probe the brokers in order to determine whether the brokers are reachable and able to publish or subscribe to their messages. The backup link to the next broker is automatically used if such a check for a connection to a broker fails. [24].

While it might not be the most optimal solution due to the fact that to achieve a semi robust federation, one would need a backup for each advertisement bridge in the federation, it is fairly simple to setup. For simplicity, let us explore a small example. On the left hand side of Fig. 7.3 are 3 brokers, 2 connected with data producers Ⓐ Ⓑ at the bottom of the hierarchy and 1 Ⓒ connected with the data consumers at the top of the hierarchy. This could also be a small part of a much larger system with brokers even higher in the hierarchy. On the right hand side of Fig. 7.3 we can see that there is a backup MQTT bridge for every connection, both for connections that transfer publisher information and for advertisements. Each

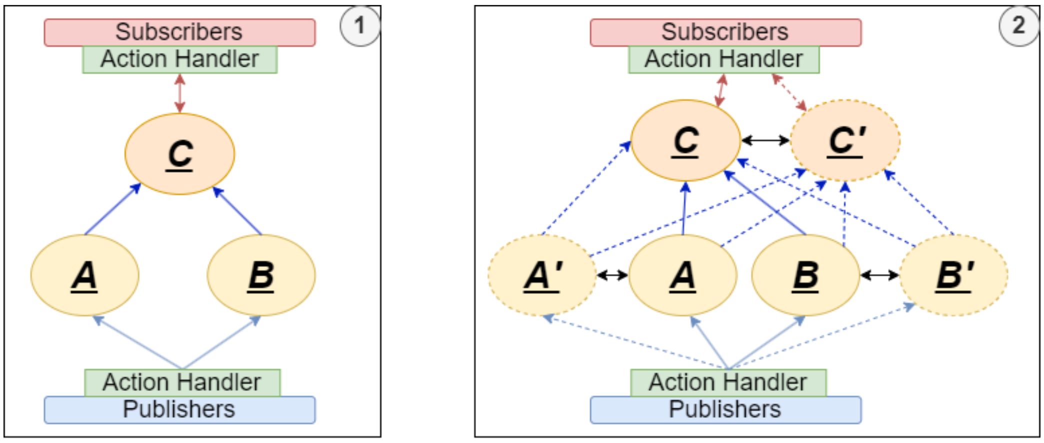


Figure 7.3: An example of a system comprised of just 3 brokers.

broker has a ”partner” backup broker which keeps a copy of everything the initial broker has by subscribing to the '#' topic, which is a wildcard that includes every single topic message. Let us examine broker Ⓐ and backup broker Ⓐ' . First, there is the connection between the two brokers. The Broker Ⓐ' subscribes to everything that is published/stored in the Broker large Ⓐ thus being an exact replica. Then there is the connection between Ⓐ' and the publishers. The publishers that publish in the area covered by broker Ⓐ consider him the primary broker and broker Ⓐ' the fallback broker. For the advertisements in the federation, broker Ⓐ is bridged with broker Ⓒ and has a backup bridge with broker Ⓒ' which is broker's Ⓒ ”partner”. Now the connection that requires the most care is between the backup broker Ⓐ' and its higher ups. Since everything it receives is a carbon copy of what broker Ⓐ has, handling the bridges of advertisements is somewhat tricky. A way to deal with this is to remap the advertisement topics so that the advertisements that arrive from broker Ⓐ and have its address in the topic are changed to have broker's Ⓐ' address, for example topic ”provider/+ broker_A_address + / broker_A_port +/...” is converted to ””provider/+ broker_A'_address + / broker_A'_port +/...”. If the advertisements remained unchanged, in case broker Ⓐ failed, broker Ⓐ' which is supposed to replace him in the downtime, would also propagate advertisements that point to an unavailable broker. We also want to avoid having broker Ⓐ' from propagating advertisements to brokers Ⓒ or Ⓒ' while broker Ⓐ is still available, as it would be an unnecessary load and might lead to duplicate information. Thus broker Ⓐ' sends its advertisements to broker Ⓐ as a primary MQTT bridge (which broker Ⓐ discards) and by having backup advertisement bridges with brokers Ⓒ and Ⓒ', propagates advertisements when it considers broker Ⓐ to be unavailable. For all this to be better understood, let us see what happens in 2 scenarios. First scenario: Broker Ⓒ malfunctions and becomes unavailable. Subscribers that are interested in the information provided by the whole system try to contact broker

Ⓒ. Since they fail to connect, they fallback to broker Ⓒ'. Broker Ⓒ' has all the data that was available in Ⓒ and thus the subscribers can be serviced as if nothing happened. Second scenario: Broker Ⓐ fails. This means that publishers that want to publish data to broker Ⓐ can't reach it and they fallback to sending data to broker Ⓐ'. Broker Ⓐ' tries to send its new advertisements to broker Ⓐ but fails and thus propagates the advertisements to broker Ⓒ and if that is also unavailable (scenario 1) then to broker Ⓒ'. Subscribers that are connected to the top broker (either Ⓒ or Ⓒ') become aware that there is a new source of information via the advertisements received from Ⓐ'.

Of course the design of the connections of the MQTT backup bridges ultimately falls to the choice of the system designer. For example if the edge brokers are unavailable the sensor information could be directly sent to the next in hierarchy broker of the federation, in this case brokers Ⓒ,Ⓒ'

Chapter 8

Experimental Evaluation

Here we evaluate *ComDEX* and overall design approach using our prototype. We utilise multiple AWS EC2 VM instances within a single region, with each broker, publisher and subscriber hosted on a different VM with the specifications listed in Table 8.1. Each broker is a c5.large instance and each subscriber or producer is a t3.nano instance. For consistent time measurements, the VMs are synchronized using AWS's Time Sync Service over the Network Time Protocol (NTP).

Functionallity	**Instance Type**	**Instance Family**	**Instance Size**	**VCPUs**	**Memory (GIB)**	**Network Performace**
Brokers	c5.large	c5	large	2	4	Up to 10 Gigabit
Subscribers/Publishers	t3.nano	t3	nano	2	0.5	Up to 5 Gigabit
Publisher (§8.2)	t3.large	t3	large	2	8	Up to 5 Gigabit

Table 8.1: Configuration of experimental testbed

We first evaluate the performance of the *ComDEX* prototype with different topology sizes and compare it against other NGSI-LD brokers, for a normal case and a worst-case scenario, using simple synthetic data models of a generic NGSI-LD entity. Then, we evaluate the impact of changing the federation topology and advertisement granularity using randomly generated entities of smart buildings based on existing NGSI-LD data models and real IoT device traces. Our results throughout all experiments validate the performance of our current *ComDEX* prototype under a broad range of deployment configurations and generated workloads.

8.1 Comparison with other NGSI-LD brokers

8.1.1 Normal-case scenario

We first evaluate the performance of the *ComDEX* prototype with various topology sizes against current state-of-the-art NGSI-LD brokers: Orion-LD[1] and Scorpio[2].

[1]https://github.com/FIWARE/context.Orion-LD

[2]https://github.com/ScorpioBroker/ScorpioBroker

Name	Type	Generation Strategy
id	URN	A urn string generated by concatenating “urn:ngsi-ld:entity” and a unique number between 0 to total number of entities requested.
Type	String	A type that represent various different entity types, it is generated by concatenating the string ”Dummy_entity” and a random number between 0 and 10
Attribute1	String	A string value generated by contacting “value” and a randomly generated number between 0 to total number of entities requested.
Attribute2	Integer	Random Integer between 0 and total number of entities requested
Attribute3	Interger	Random Integer between 0 and total number of entities requested
Context	NGSI-LD context	Use the default NGSI-LD context.

Table 8.2: Virtual data model of generated entities (§8.1)

While the NGSI-LD specification is still evolving, these two brokers incorporate most up-to-date functionalities of the NGSI-LD specification. *ComDEX* adopts 1, 3, 6 broker-*wide* topologies, where width is the distance (network hops) from the broker where the data is published to the broker where interested subscribers connect.

For this set of experiments, we created a data model with simple attributes with no semantic meaning, following a similar approach to [15]. The structure of the data model can be seen in Table 8.2. Developing and using a simple data generator in Python that creates entities of the above data model, we generate 2000 entities with 10 different entity types and 3 different attributes.

The generated entities are entered into the system and stored at the edge brokers. Advertisements for each of the entity types are generated and propagated through each setup seen in Fig. 8.1. We then deploy a workload generating synthetic sensor requests on 10 VMs with each selecting a random attribute of a random entity approximately every 100ms and patching its value to a random value (e.g., entity_500, Attribute3, 42) on every different edge broker of each configuration (Orion-LD, Scorpio, *ComDEX* (Mosquitto) 1-, 3- and 6-broker-wide). In each setup we deploy one subscriber VM for each data type (e.g., Orion-LD has 10 different subscribers). We consider this a ”normal scenario” in the sense that it is setup in a way where the load isn’t very big and data transfers between publishers and subscribers are performed without any special circumstances or settings.

We let the experiment run for a few hours to warm up the system. We then modify the code of both the sensors and subscribers as follows: instead of patching the value of the attribute of a random entity to a random value, we patch it to the current time T_1 and immediately send it. Then, as soon as a subscriber receives a notification about an entity, it marks the current time T_2 and exports T_1, T_2 along with the entity, to calculate subscription notification latency. Fig. 8.2

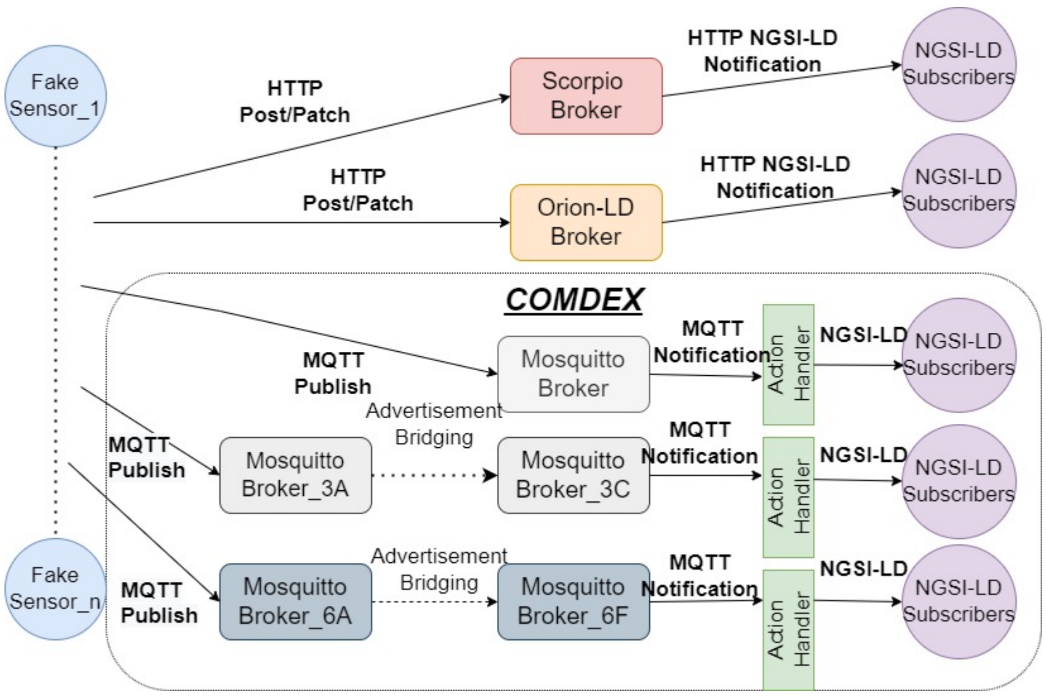


Figure 8.1: Normal-case experimental setup (§8.1.1)

reports a random sample of 50 observations of the subscription notification latency in milliseconds.

We observe that *ComDEX* performs better in comparison with native HTTP NGSI-LD solutions since MQTT is used as the data exchange protocol,. This result is in line with previous comparisons of HTTP and MQTT [33, 34] and highlights an advantage of the *ComDEX* design. We can also notice from Fig. 8.2 that since the system has been running for a while and all the advertisements have been installed, there is no substantial difference between different-sized topologies of *ComDEX*, another one of its advantages. The same result can also be seen in the first half of Fig .8.4 (nc) which shows the mean subscription notification latency across multiple runs.

8.1.2 Worst-case scenario for our system.

Next, we examined a worst-case scenario, performance-wise, for our prototype. This happens when a client needs to ”re-discover” the data source and connect to it for every piece of information published to a broker. Recall from §7.2 that each time a subscriber connects to a broker searching for available data throughout the system, it must search through the existing advertisements to find where to connect. One way to emulate such a scenario is to have all data requested be composed of single entities of different types, which is feasible but not directly compatible with the single type subscription of NGSI-LD. Another way to emulate

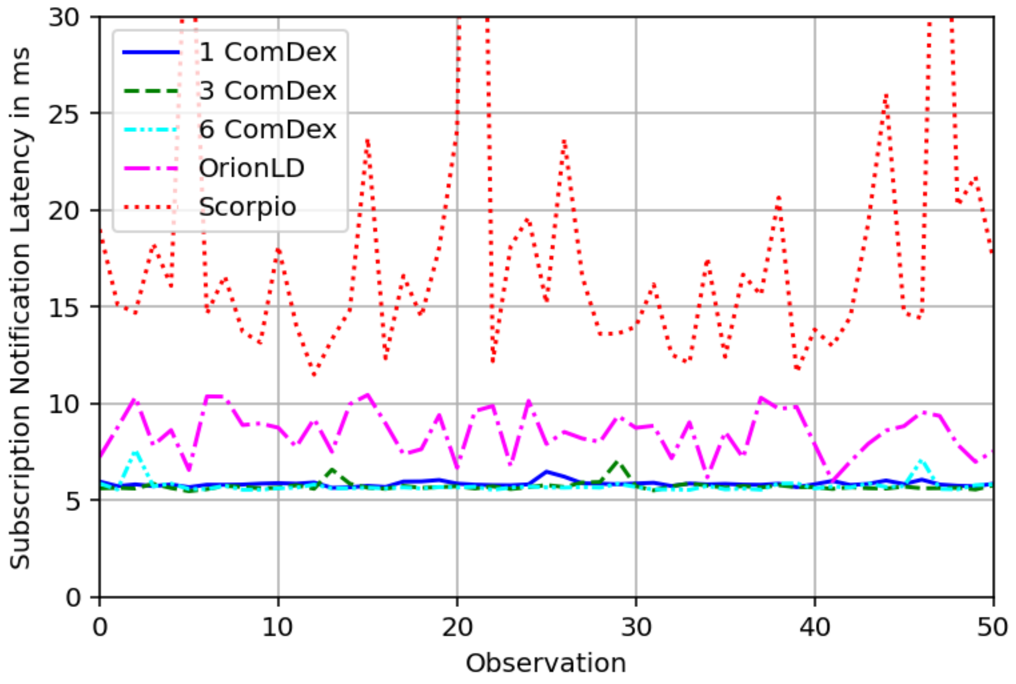


Figure 8.2: Subscription notification latencies (§8.1.1)

this is to have the data requested (subscribed to) be comprised of an entity-type for which its data and subsequently its advertisement are constantly being deleted and reinserted, which is the direction we chose for this experiment (shown in Fig. 8.3). These worst-case emulated setups are luckily not expected in realistic systems. Again, we compare *ComDEX* to Orion-LD and Scorpio in similar conditions; since the latter two do not use advertisements, we expect them to not be affected as much.

We use the data generator from the first experiment to create an entity of type "DummyEntity". This entity is then inserted into each broker along with a timestamp of when it was sent. The subscribers receive this information and mark the time it arrives. The entity is then deleted from the broker it was inserted into, and consequently, the advertisement for that entity type is also removed from the entire system. We repeat the process a large number of times and report our results in Fig. 8.4, where the metrics from experiment 2 (wc, §8.1.2) are displayed side by side in comparison with the first experiment (nc, §8.1.1). We observe that the performance of our prototype in the worst-case scenario declines, as expected. This delay can only happen when a subscriber consistently wants an entity whose type does not exist and waits for its advertisement to arrive to discover it. We also use the setup to monitor the *ComDEX* advertisement installation[3] times, as

[3]Advertisement installation time is the time it takes for an advertisement from its reception

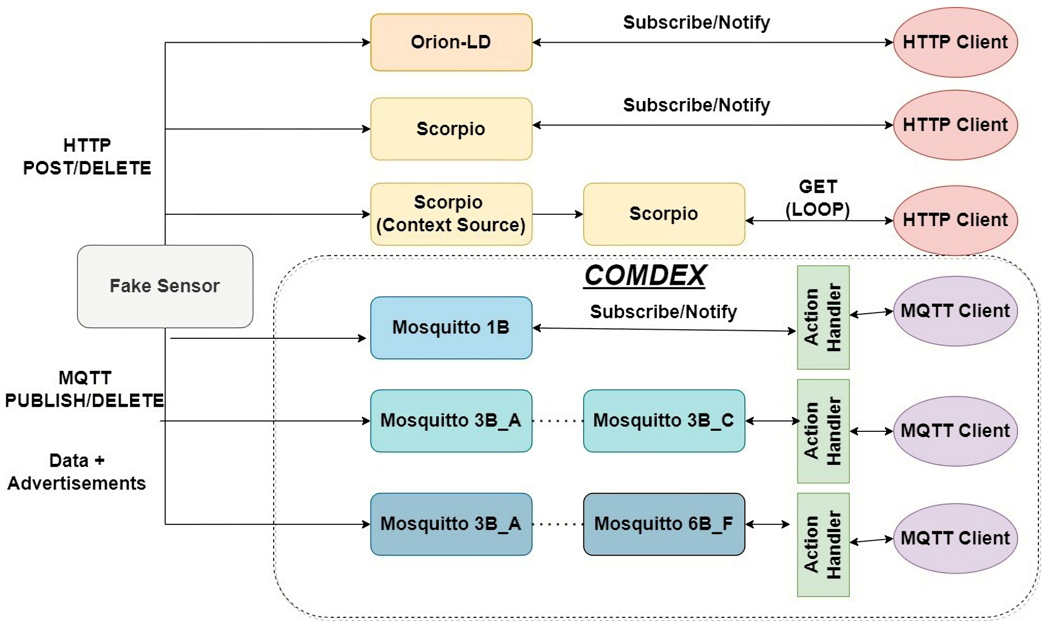


Figure 8.3: Worst-case scenario experimental setup (§8.1.2)

depicted in Fig. 8.5. As expected, when the distance between the edge broker and the top broker increases, advertisement installation time increases as well, but not dramatically.

8.2 Evaluating *ComDEX* using diverse advertisement granularities and real IoT device traces

Next, we examine the impact of topology and advertisement granularity on the performance of our prototype. We consider type-based advertisement granularity, where an advertisement is created and propagated for every different entity type, and id-based granularity, where an advertisement is created for each different entity (§7.3). Being able to discover exact entities with their id allows for greater flexibility, such as remote actions on single entities. Having only advertisements on entity types reduces the number of advertisement messages, lowering traffic through system brokers. The downside of this is having to check if an advertisement already exists when creating an entity, which takes time. For this experiment, we implemented a generator of smart-building entities and IoT devices. The tool receives as input the number of federated brokers and generates communities, buildings, floors, rooms, and devices, each contained in the other in a logical way, by relying on an extended NGSI-LD based building and device data models. To create a realistic scenario for the experiment, the IoT devices and publishers are modeled using

from the edge broker to arrive at the highest broker in the hierarchy

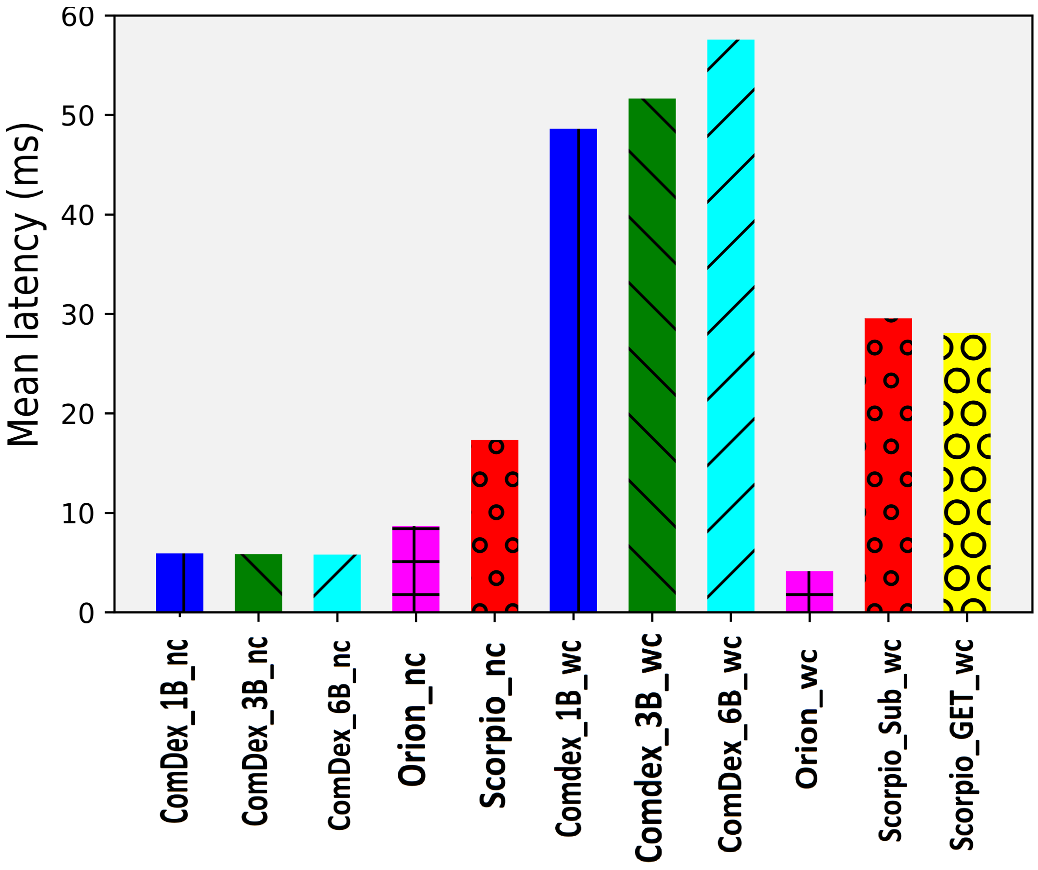


Figure 8.4: Comparison of end-to-end delays from data publication to subscriber notification

the characteristics of devices in the work by Kumar et al. [20] where a dataset of 20 days of network traces generated by 20 IoT devices, is processed and their significant features extracted.

Here we simulate the environment described in the motivating scenario using 3 different topologies (mesh, *ComDEX* topology, and a single central broker). In the mesh topology, the brokers are connected to each other, thus every broker is aware of entities present in the other brokers through direct advertisements, unlike *ComDEX* where there is a hierarchy between brokers. Each of the different communities in each setup has a set of subscribers that are interested in changes in the values of devices across all smart communities. Such a subscriber could be, for example, a firefighting application that wants to know if a smoke detector detects a fire in a building regardless of location. The synthetic device sensors in this experiment work as follows: the devices are generated with a set of characteristics (message size, message frequency) taken from real device traces. Each device sends

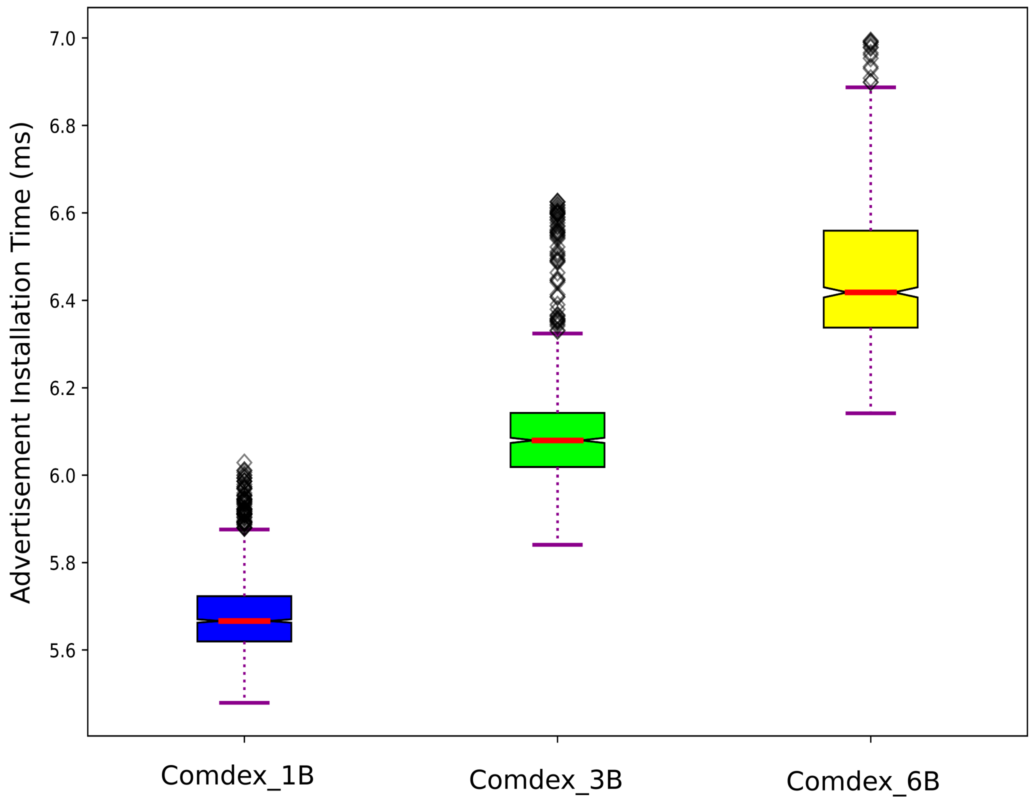


Figure 8.5: Advertisement-installation times for *ComDEX* in 1, 3, 6-broker-wide setups

a message to its value in the appropriate broker according to these features continuously. To calculate the various metrics of this experiment, we used the tshark[4] network monitoring tool at each broker, similar to Bertrand-Martinez et al.[8], to avoid modifying the content of the entities as we did in previous experiments.

The results of this experiment are depicted in Figs. 8.7-8.9. In Fig. 8.7, we observe the difference in the number of messages needed for the creation of the entities of each community, clearly seeing the impact of topology. In the mesh topology where "everyone knows everything", especially in the case of id granularity where the advertisements are equal to the number of entities inserted, the number of messages exchanged between brokers increases, since in this case every advertisement must be propagated to every broker.
In Fig. 8.8 we observe advertisement-installation times for different topologies and advertisement granularities. As expected, there is not much variance between topologies since the difference in the number of hops of each advertisement propagation does not exceed 2. The difference between the single broker and the two topologies (mesh,*ComDEX*) is similar to the difference between "broker widths"

[4]https://tshark.dev/

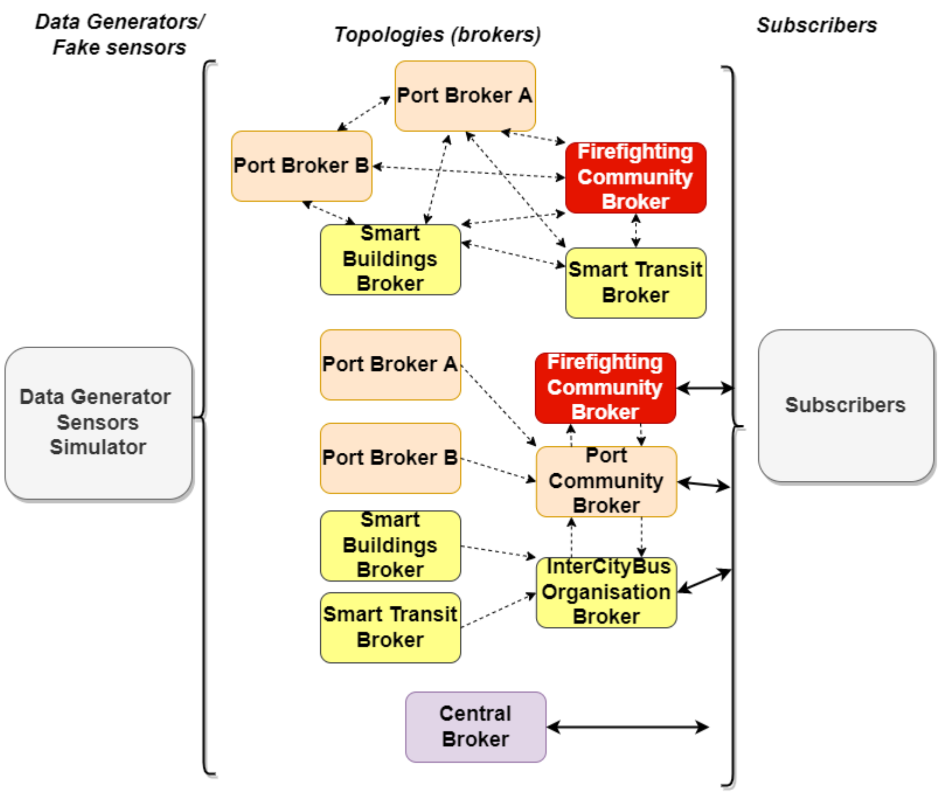


Figure 8.6: A more realistic setup, using real data models simulating three smart communities

in Fig. 8.5. The variability between granularities is not significant and the slight difference between them can be attributed to the larger sample inherent to the nature of the id granularity (1 advertisement for each entity).

Lastly, we evaluated how each system performs under intense load conditions when MQTT is set for QoS 0 (at-most-once delivery). Note that the QoS of the propagation of advertisement messages from broker to broker remains the same across all experiments (QoS 2, exactly once). To create stress-testing conditions we increased the number of generated devices across all communities. Our results in Fig. 8.9 show that under heavy load all systems drop messages. *ComDEX* outperforms the others, while the single-broker case performs the worst. *ComDEX* achieved better message-delivery rate compared to the mesh topology since in *ComDEX*, a subscriber requests information about where to connect (advertisements) from a high-level broker that in this experiment does not deal directly with generated data and thus has a lighter load. In the mesh topology, the subscribers interact directly with the brokers that are under heavy load, and that is causing

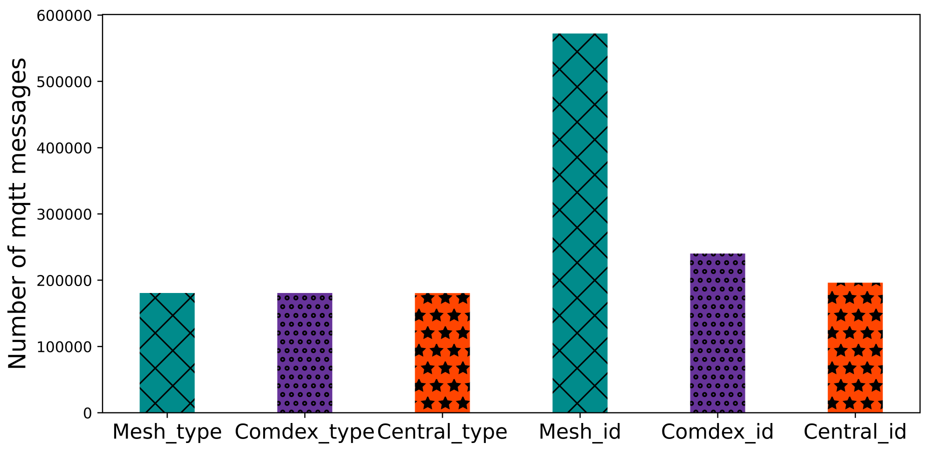


Figure 8.7: Number of messages required for the creation of 15000 entities

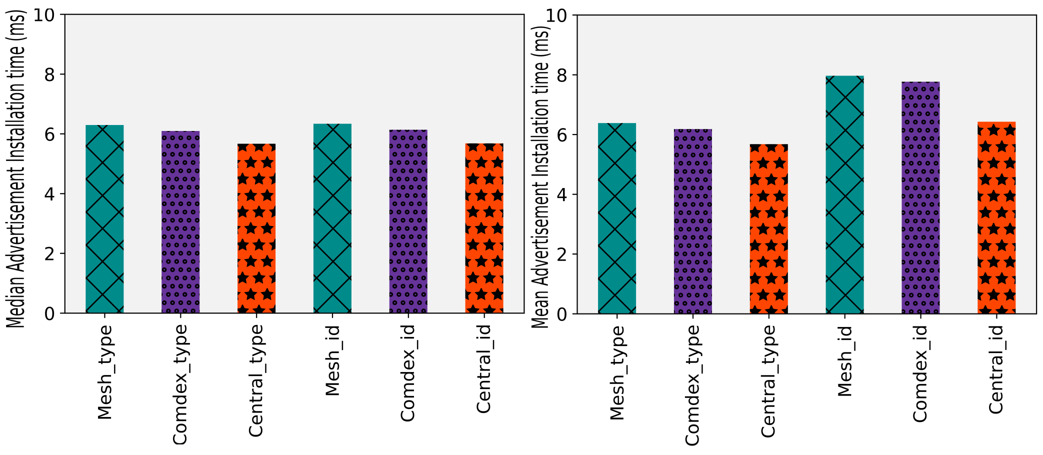


Figure 8.8: Advertisement installation times for different topologies and advertisement granularities

advertisement messages requested by the subscriber to get dropped (QoS 0), leading in turn to the subscriber missing out on data that interests them (does not know where to connect to get it).

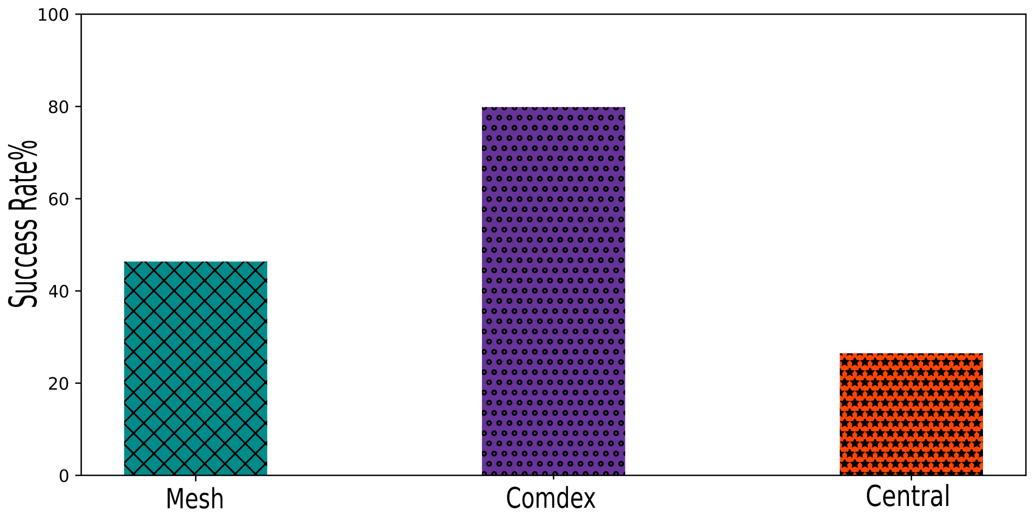


Figure 8.9: Success rate of messages for different topologies under heavy load (MQTT-QoS 0)

Chapter 9

Related Work

As smart IoT communities mature, so do their complexity and overall requirements. Maintaining important aspects of smart spaces, such as data privacy, interoperability, high availability, and context data discovery, across a partnership of diverse smart communities becomes a challenging endeavor. IoT platforms have been characterized by the distributed deployment of real-time applications and high-speed data dissemination. For asynchronous and cross-platform communication, IoT applications most commonly use publish/subscribe middleware [18]. However, in traditional pub/sub solutions, data exchange between brokers is unrestricted as their main goal is to maximize performance. *ComDEX* acknowledges the limitations of traditional pub/sub when designing a cross-community collaboration platform and builds upon this.

For this cross-community collaboration, many initiatives connected to the creation of smart-city platforms have adopted **federated designs**. For example, to establish optimal and sustainable infrastructures for city occupants, existing smart city platforms such as MARGOT [26], Fogflow [11], Trustyfeer [21], ALMANAC [10], CPaaS.io [12], and WiseIoT [16] provide a method for cross-domain communication, interoperability, and IoT resource discovery. Cloud federation, context broker federations, and service orchestration are some of the techniques/technologies used to support such interoperable and wide-scale deployments. Most of these platforms either lack service programming models or specify a programming model solely based on their own private data model and interfaces, limiting their openness and interoperability as smart community platforms. *ComDEX* tackles this lack of openness by using property graphs combined with Semantic Web domain models.

The NGSI-LD standard [1] is used as the basis of the information model of *ComDEX* and the API for publishing, querying, and subscribing to context data. Its purpose is to make the open exchange and sharing of structured data amongst various parties easier. It is utilized in multiple smart city domains. It has also been discussed in many recent research publications, such as the the CityDataHub [17] project, which is an online platform that is under trial in several South Korean cities that provides a communication window for sharing various policies promoted by the public and the Demeter [23] project, which is a large-scale deployment of farmer-driven, interoperable smart farming-IoT based platforms. On the other hand, NGSI-LD has its limitations such as limited federation and forwarding support. Unlike other NGSI-LD solutions, *ComDEX* handles these limitations by implementing a novel forwarding scheme.

Regarding smart IoT platforms, FogFlow introduces a fog computing based framework for designing and implementing city-scale IoT applications. Tricomi et al. [31], introduce a software-defined platform that uses a city infrastructure to develop applications on top while dealing with numerous administrative domains through federated architectures. High availability disaster recovery (HADR) is supported in the MARGOT [26] platform by exploiting caching capabilities in federated nodes. HADR scenarios are also supported in city-scale deployments by the Pradhan federated platform [24], which leverages the lightweight MQTT protocol for its support of different QoS message delivery modes. Finally, Trustyfeer [21] relies on federated cloud environments to improve the number of services exchanged by cloud providers that conform to SLAs. *ComDEX* deals with QoS in the aspect of end-to-end delivery guarantees, taking advantage of the MQTT [22] communication protocol. Fault tolerance is also considered and enabled in *ComDEX* with the use of backup MQTT bridges that connect the federation.

The Space Broker [3] defines a smart space based on contextual data, spatial attributes, and user-specified application requirements. Interoperability capabilities, reusability (smart apps reused in many settings), user privacy and data sovereignty are all crucial characteristics of IoT smart spaces that SemIoTic [35] addresses. However, both of these works are focused on a single smart space and do not offer a solution for all of these important qualities in a federation of smart communities/smart spaces, contrary to *ComDEX*.

While existing technologies can be utilized to cover certain requirements specifically defined for the creation of an optimal context-aware IoT platform for smart communities as well as the solutions proposed for singular smart spaces, a solution in a federation of brokers that covers all the requirements of a smart IoT community has not been properly defined yet. Overall, *ComDEX* can be described as a distributed context-aware federation architecture platform for enabling widespread IoT applications that implements novel federation topology and information propagation techniques, facilitating data sovereignty while still enabling open collaboration between smart communities. A small overview of the use of federation and the capabilities of the systems of the various works referenced in this section can be found in the table 9.1.

	Methodology	*Security Mentioned*	*Interoperability*	*Resilience*	*Scalability*	*Discovery*	*Prototype*	*Federation Usage*
Carvajal et al. 2015 (ALMANAC)	Adapts the context of cloud federation to create a custom smart city platform.	SAML Authentication	✓	✓	**X**	**X**	✓	-Interoperability
Morelli et al. 2020 (MARGOT)	Uses federation services and ABE for the development of a smart city platform.	Attribute Based Encryption (ABE)	✓	✓	**X**	✓	✓	-Interoperability -Resilience/Fault Tolerance. -Security/Isolation
Cirillo et al. 2019 (Cipaas.io)	Smart City platform, using Fiware based components.	Key-Rock, Pep-proxy	✓	**X**	**X**	✓	**X**	- Interoperability - Discovery
Cheng et al. 2018 (Fogflow)	IoT edge computing framework using an intent-based programming model and context-driven service orchestration	Fiware Iot Stack's Security Mechanisms	✓	**X**	✓	✓	✓	- Interoperability - Scalability - Discovery
Jaeyoung Hwang et al. 2017 (WiseIoT)	Leveraging semantics and federation to build interoperabillity between smart city platforms.	**X**	✓	**X**	**X**	✓	**X**	- Interoperability - Discovery
Heba Kurdi et al. 2018 (Trustyfeer)	Usage of subjective logic equations based on SLAs and CSP's global reputation to create a trust management system for P2P federated clouds	Trustyfeer's Security	✓	**X**	✓	✓	✓	- Interoperability - Scalability - Discovery
Tricomi et al. 2019	Extends the vision of the SOC paradigm with the concept of federation	**X**	✓	**X**	**X**	✓	✓	-Interoperability -Discovery
Pradhan Manas 2021	Communication between agencies in HADR scenarios using federation of MQTT brokers.	X (unencrypted mqtt communication)	✓	✓	**X**	✓	✓	-Interoperability -Discovery -Resilience
ComDEX	Context aware Intercommunication between different smart communities using advertisements and MQTT bridges to create federations.	**X**	✓	✓	✓	✓	✓	-Interoperability -Scalability -Discovery -Resilience

Table 9.1: Smart-city platforms with federated designs

Chapter 10

Conclusions and future work

10.1 Conclusions

While smart communities can be established using existing platforms, a standard IoT platform architecture does not exist. This thesis, introduces *ComDEX*, an approach for creating federated context-aware IoT platforms for smart communities. A federation architecture, is proposed, that represents data as property graphs for smart community entities. Context-aware messages are subsequently created, using a topic/type-based pub/sub subscription scheme. While some might argue that it should not be considered a true federation as it requires participants to have similar technologies, it is important to point out that NGSI-LD provides various ”gateways” such as IoT-agents to facilitate easier intercommunication/cooperation between different existing platforms. Such messages are exchanged between smart communities via the *ComDEX* hybrid and hierarchical federation topology. *ComDEX*'s architecture is evaluated by creating and using a prototype that utilizes technologies such as NGSI-LD and MQTT, qualitative and quantitative comparisons are performed with other existing solutions.

10.2 Future Work

We believe that *ComDEX* as it is in its current form, can be the start of a much more ambitious platform. Firstly, as NGSI-LD is a technology that is still evolving with new features being added at a pace that is difficult to keep up with, it would take a lot of resources to end up with a fairly compliant platform to the latest NGSI-LD specification, but we believe that it is something that a future version of *ComDEX* could take more seriously into consideration. Secondly, while QoS guarantees are briefly mentioned and implemented by our decision to utilize MQTT as the main messaging protocol, it only concerns the MQTT delivery guarantees and not the various different QoS that might be required by different data recipients. For example, applications operating under normal circumstances or emergency response cases. Optimally, there should be ways to enable the dynamic configuration of the platform, using different routes or message handling mechanisms, based on the requirements of data recipients (e.g., emergency responders) or the deployed IoT applications of each community. Cross-layer optimization mechanisms can be leveraged to tune such a federated system. The policy manager component mentioned in section 3.2 while not implemented, was added to the overall design with the aforementioned prospective dynamic configuration capabilities in mind. Additionally, as of the time of writing this thesis, the prototype implementation has no real concern about secure communication between the various nodes. A version of *ComDEX* with a component that handles certifications is something to be seriously examined in future work. In the future, we could also examine how well and with how much difficulty *ComDEX* can be integrated with existing smart city platforms, so that we can prove that *ComDEX* can be considered a federated platform with the whole meaning of the word. Lastly, more experiments could be performed with real sensors and over a longer period of time to get a proper evaluation of the system under realistic scenarios.

Bibliography

[1] Context information management (cim) ngsi-ld api v1.4.2, 04 2021.

[2] Abowd. Towards a better understanding of context and context-awareness. 01 1999.

[3] Hamim Md Adal, Colin Milhaupt, Jie Hua, Christine Julien, and Gruia-Catalin Roman. The space broker: A middleware for mediating interactions in smart iot spaces. In *Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '21, page 101–110, New York, NY, USA, 2021. Association for Computing Machinery.

[4] João Almeida, Jorge Silva, and Thaís Batista. A linked data-based service for integrating heterogeneous data sources in smart cities. pages 205–212, 01 2020.

[5] Renzo Angles. The property graph database model. In *AMW*, 2018.

[6] R Baldoni, Leonardo Querzoni, Sasu Tarkoma, and Antonino Virgillito. *Distributed Event Routing in Publish/Subscribe Communication Systems.* 02 2009.

[7] Nazmiye Balta-Ozkan, Rosemary Davidson, Martha Bicket, and Lorraine Whitmarsh. The development of smart homes market in the uk. *Energy*, 60:361–372, 10 2013.

[8] Eddas Bertrand[U+2010]Martinez, Phelipe Feio, Vagner Nascimento, Fabio Kon, and Antônio Abelém. Classification and evaluation of iot brokers: A methodology. *International Journal of Network Management*, 31, 06 2020.

[9] Andre Borrmann, Jakob Beetz, Christian Koch, T. Liebich, and Sergej Muhic. *Industry Foundation Classes: A Standardized Data Model for the Vendor-Neutral Exchange of Digital Building Models*, pages 81–126. 09 2018.

[10] José Carvajal Soto, Otilia Werner-Kytölä, Marco Jahn, Pullman J., Dario Bonino, Claudio Pastrone, and Maurizio Spirito. Towards a federation of smart city services. 11 2015.

[11] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018.

[12] Flavio Cirillo, Gurkan Solmaz, Everton Luis Berz, Martin Bauer, Bin Cheng, and Erno Kovacs. A standard-based open source iot platform: Fiware. *IEEE Internet of Things Magazine*, 2(3):12–18, Sep 2019.

[13] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, 06 2003.

[14] The Apache Software Foundation. Apache qpid, messaging built on amq, 2015.

[15] Alireza Hassani, Alexey Medvedev, Arkady Zaslavsky, Pari Delir Haghighi, Prem Prakash Jayaraman, and Sea Ling. Efficient execution of complex context queries to enable near real-time smart iot applications. *Sensors*, 19(24), 2019.

[16] Jaeyoung Hwang, JongGwan An, Hotaek Joo, ChanHyung Lee, and Jaeseung Song. Development and application of interoperability techniques with semantics for global internet of things (giots). *The Journal of Korean Institute of Communications and Information Sciences*, 42:2208–2216, 11 2017.

[17] Seungmyeong Jeong, Seongyun Kim, and Jaeho Kim. City data hub: Implementation of standard-based smart city data platform for interoperability. *Sensors*, 20:7000, 12 2020.

[18] Zhuangwei Kang, Robert Canady, Abhishek Dubey, Aniruddha Gokhale, Shashank Shekhar, and Matous Sedlacek. A study of publish/subscribe middleware under different iot traffic conditions. M4IoT'20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Saad Liaquat Kiani, Ashiq Anjum, Michael Knappmeyer, Nik Bessis, and Nikolaos Antonopoulos. Federated broker system for pervasive context provisioning. *Journal of Systems and Software*, 86(4):1107–1123, 2013. SI : Software Engineering in Brazil: Retrospective and Prospective Views.

[20] Rakesh Kumar, Mayank Swarnkar, Gaurav Singal, and Neeraj Kumar. Iot network traffic classification using machine learning algorithms: An experimental analysis. *IEEE Internet of Things Journal*, 9(2):989–1008, 2022.

[21] Heba Kurdi, Bushra Alshayban, Lina Altoaimy, and Shada Alsalamah. Trustyfeer: A subjective logic trust model for smart city peer-to-peer federated clouds. *Wireless Communications and Mobile Computing*, 2018:1–13, 02 2018.

[22] Shinho Lee, Hyeonwoo Kim, Dong-Kweon Hong, and Hongtaek Ju. Correlation analysis of mqtt loss and delay according to qos level. pages 714–717, 01 2013.

[23] Juan A. López-Morales, Juan A. Martínez, and Antonio F. Skarmeta. Improving energy efficiency of irrigation wells by using an iot-based platform. *Electronics*, 10(3), 2021.

[24] Pradhan Manas. Federation based on mqtt for urban humanitarian assistance and disaster recovery operations. *IEEE Communications Magazine*, 59(2):43–49, 2021.

[25] Anahita Molavi, Gino Lim, and Bruce Race. A framework for building a smart port and smart port index. *International Journal of Sustainable Transportation*, 04 2019.

[26] Alessandro Morelli, Lorenzo Campioni, Niccolò Fontana, Niranjan Suri, and Mauro Tortonesi. A federated platform to support iot discovery in smart cities and hadr scenarios. pages 511–519, 09 2020.

[27] Gilles Privat. Guidelines for modelling with ngsi-ld (etsi white paper), 03 2021.

[28] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[29] Bessid Sahbia, Alaeddine Zouari, Frikha Ahmed, and Benabdelhafid Abdellatif. Smart ports design features analysis: A systematic literature review. 11 2020.

[30] Manu Sporny, Gregg Kellogg, and Markus Lanthaler. Json-ld 1.0 - a json-based serialization for linked data. *W3C Recommendation*, 01 2014.

[31] Giuseppe Tricomi, Giovanni Merlino, Francesco Longo, Distefano Salvatore, and Antonio Puliafito. Software-defined city infrastructure: A control plane for rewireable smart cities. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 180–185, 2019.

[32] UN/ECE. Regulation no 107 of the economic commission for europe of the united nations (unece) — uniform provisions concerning the approval of category m2 or m3 vehicles with regard to their general construction. *Official Journal of the European Union*, pages 1–115, 2015.

[33] Bharati Wukkadada, Kirti Wankhede, Ramith Nambiar, and Amala Nair. Comparison with http and mqtt in internet of things (iot). In *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, pages 249–253, 2018.

[34] Tetsuya Yokotani and Yuya Sasaki. Comparison with http and mqtt on required network resources for iot. pages 1–6, 09 2016.

[35] Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, and Nalini Venkatasubramanian. The semiotic ecosystem: A semantic bridge between iot devices and smart spaces. *ACM Transactions on Internet Technology – TOIT*, 2022.