UNIVERSITÀ DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXVIII Ciclo

Efficient Data Management with Applications to IoT

Coordinatore:

Chiar.mo Prof. Marco Locatelli

Tutor:

Chiar.mo Prof. Gianluigi Ferrari

Dottorando: *Laura Belli*

Gennaio 2016

To My Family

Contents

List of Figures

List of Tables

Introduction

The Internet of Things (IoT) consists of a global worldwide "network of networks," composed by billions of interconnected and communicating heterogeneous devices denoted as "Smart Objects" (SOs). The IoT includes billions of nodes, thus enabling new forms of interaction between things and people. The actors involved in IoT scenarios will have extremely heterogeneous characteristics, in terms of processing and communication capabilities, energy supply and consumption, availability, and mobility, spanning from constrained devices equipped with sensors or actuators, to smartphones and other personal devices, Internet hosts, and the Cloud.

Shared and interoperable communication mechanisms and protocols are currently being defined and standardized, allowing heterogeneous nodes to efficiently communicate with each other and with existing Internet actors. Significant research efforts have been dedicated in past years to port the experience gained in the design of the Internet to the IoT, while taking into account the very nature of smart objects. In particular, with the goal of maximizing interoperability among devices, the use of the Internet Protocol (IP) [1] has been widely accepted as the driver for the effective evolution of the IoT and its integration with the Internet. Many research projects and organizations, such as the IPSO Alliance and the Internet Engineering Task Force (IETF), focused their work on the design of an IoT IP-based protocol stack which can match that of the Internet.

At the application layer, an important result of this standardization effort has been the design of IoT-specific protocols, like the Constrained Application Protocol (CoAP) [2], which is set to become the standard communication protocol between

and with smart objects, similarly to what HTTP is for the Internet.

As this first wave of standardization can be considered successfully concluded, we can assume that communication with and between smart objects is no longer an issue. At this time, in order to favor the widespread adoption of the IoT by users, it is crucial to provide mechanisms that can facilitate IoT data management and the development of new services and applications enabling a real interaction with the surrounding smart objects.

In order to bridge the gap between things and humans, IoT applications can provide useful services to final users as a consequence of the processing work on the huge amount of data collected by SOs. Moreover several reference IoT scenarios, such as industrial automation, transportation, networks of sensors and actuators, require real-time/predictable latency and could even change their requirements (e.g., in terms of data sources) dynamically and abruptly. In this context, with billions of nodes capable of gathering data and generating information, Big Data techniques address the need to process extremely large amounts of heterogeneous data for multiple purposes. These techniques have been designed mainly to deal with huge volumes (focusing on the data itself), rather than to provide real-time processing and dispatching. Thus, the IoT scenario, can be mistakenly considered only as a Big Data scenario, but is important to note that Smart-X services significantly differ from traditional Internet services, in terms of: i) number of data sources; ii) rate of information exchange; and iii) need for real-time processing.

The requirements listed above create a new need for architectures specifically designed to handle this kind of scenario and to guarantee minimal processing latency. Such systems are denoted as “Big Stream” systems. Another important step, needed to foster IoT development and diffusion, is to build more applications around the well-known Web model, bringing about the so-called Web of Things (WoT). The Web-based approach helped to greatly expand the Internet and will likely have the same effect on the IoT.

WoT applications rely on specific Web-oriented application-layer protocols similar to HTTP, such as CoAP and, more generally, protocols complying with the Representational State Transfer (REST) architectural style [3]. While simulation tools

typically focus on evaluating lower-layer communication protocols, in recent years several IoT testbeds have been deployed to evaluate IoT solutions in realistic smart environments under real-world conditions, with the aim of testing protocols at the link and network layers and to collect performance results such as energy consumption or packet delivery ratio. Although these lower layer protocols have been widely investigated, additional efforts are needed to create new innovative services, to easily build applications for the IoT environment. To make IoT real, developers need the ability to work in a new kind of testbed, with a high level of abstraction and without worrying about low-level details.

Another relevant aspect, related to innovative applications which can favor the widespread diffusion of the IoT by common people, is to provide mechanisms and paradigms that can facilitate the discovery and the interaction with surrounding smart objects. The extreme popularity of mobile devices, combined with their great capabilities, in terms of processing power and communication, makes them a perfect fit to bridge the gap between things and humans. Mobile devices like smartphones or tablet, in fact, can allow seamless interaction between users and smart objects, in particular with new mechanisms for the dynamic generation of User Interfaces (UIs) on mobile devices driven by smart objects. This kind of mechanism should be REST-compliant and follow the "Hypermedia As The Engine Of Application State" (HATEOAS) constraint of the REST paradigm [4].

Thesis Structure

Chapter 1 presents the state of the art of data management related to the IoT scenario. First, the main protocols and communication models for IoT are presented, with reference to the most relevant existing IoT Testbeds. Then, the main data management solutions, in literature and inside companies, are analyzed, with particular attention to the real-time and low latency scenario. Finally, some works related to the dynamic generation of UIs in IoT are presented.

Chapter 2 introduces the concept of "Big Stream Paradigm" and describes in detail the proposed Big Stream Graph Architecture and its modules. It also presents the architecture implementation with open source technologies and provides some

results related to the system performance.

In Chapter 3, possible security issues and technologies, which can be adopted in the proposed Big Stream Graph architecture, are described. More in detail, the chapter presents how the entire system module can be modified in order to take into account security problems. Finally a performance evaluations of the secured architecture is proposed.

Chapter 4 is dedicated to IoT Applications. In the first part, the Web of Things Testbed (WoTT) structure and its benefits for developers are presented in detail. The last part of the chapter describes a novel paradigm to dynamically generate UIs on mobile devices in order to make common people able to easily interact with smart objects in an IoT network.

Chapter 1

Literature and Software Analysis

1.1 Internet of Things

The huge number of heterogeneous smart objects deployed in an IoT system allows the development of ubiquitous sensing in most areas of modern living. The outcome of this trend is the generation of a huge amount of data that should be treated, aggregated, processed, transformed, stored, and delivered to the final users of the system, in an effective and efficient way, by means of traditional commodity services.

Several architectures for IoT scenarios have been proposed in the literature. The European Union (EU), under its 7th Framework Program (FP7, 2007-2013), has supported a significant number of IoT-related projects, converging in the "IoT European Research Cluster" (IERC) and addressing relevant challenges, particularly from a Wireless Sensor Network (WSN) perspective. Among them, we recall "IoT European Research Cluster" (IERC) described in "Internet of Things-Architecture" (IoT-A) [5] and SENSEI [6].

In the IoT-A project, developers focused their work on the design of an IoT architecture, aiming at connecting vertically closed applications, systems and architectures, in order to create integrated and open-interoperable environments and platforms. The main goal of the SENSEI project was to create a business-driven platform that addresses the scalability problems for an amount of globally distributed devices

in Wireless Sensor and Actuator (WS&A) networks, enabling an accurate and reliable interaction with the physical environment, and providing network and information management services.

"Connect All IP-based Smart Objects!" (CAL*IP*SO) was another EU FP7 project with relevant implications on the design of IoT platforms [7]. The main goal of CAL*IP*SO was to enable IPv6-based [8] connectivity of smart objects to IoT networks with very low power consumption, thus providing long lifetime and high interoperability, also integrating radio duty cycling and data-centric mechanisms with IPv6.

1.1.1 Protocols and Communication Models for IoT

It is a common assumption that, in IoT, the most prominent driver to provide interoperability is IPv6. Referring to the application layer of the IP stack, developers can find a variety of possible protocols applicable to different IoT scenarios, according to specific application requirements. Among many options, the following are relevant.

- HyperText Transfer Protocol (HTTP) [9], is mainly used for the communication with the consumer's devices.
- Constrained Application Protocol (CoAP), defined in [2], is built on the top of User Datagram Protocol (UDP), follows a request/response paradigm, and is explicitly designed to work with a large number of constrained devices operating in Low power and Lossy Networks (LLNs).
- Extensible Messaging and Presence Protocol (XMPP) [10], is based on XML, supports decentralization, security (e.g., TLS), and flexibility.
- MQ Telemetry Transport (MQTT) [11] is a lightweight publish/subscribe protocol running on top of TCP/IP. It is an attractive choice when a small code footprint is required and when remote sensors and control devices have to communicate through low bandwidth and unreliable/intermittent channels. It is characterized by an optimized information distribution to one or more receivers, following a multicast approach. Being based on a publish/subscribe

communication paradigm, it acts through a "message broker" element, responsible for dispatching messages to all topic-linked subscribers.

- Constrained Session Initiation Protocol (CoSIP), described in [12, 13], is a lightweight version of the Session Initiation Protocol (SIP) [14] aiming at allowing constrained devices to instantiate communication sessions in a standard fashion, optionally including a negotiation phase of some parameters, which will be used for all subsequent communications.

1.2 Stream and Real Time Management

The term Big Stream was first introduced in [15]. Since the concept is related to platforms abiding by low-latency and real-time requirements, it can be compared with other solutions relying on such constraints. Possible examples of such solutions are Apache Storm [16] and Apache S4 [17].

Apache Storm is a free and open-source distributed real-time computation system, oriented to reliably process unbounded streams of data. It can be integrated with different queueing and database technologies, providing mechanisms to define topologies in which nodes consume and process data streams in arbitrarily, complex ways. Apache S4 is a general purpose, near real-time, distributed, decentralized, scalable, event-driven, and modular platform that allows programmers to implement applications for data streams processing. Multiple application nodes can be deployed and interconnected on S4 clusters, creating more sophisticated systems.

Marganiec et al., in their works [18, 19], address the problem to process, procure and provide information related to the IoT scenario with almost zero latency. The authors consider, as a use case, a taxi fleet management system, which has to identify the most relevant taxi in terms of availability and proximity to the customer's location. The core of the publish/subscribe architecture proposed in [18] is the *Mediator*, which encapsulates the processing of the incoming requests from the consumer side and the incoming events from the services side. Services are publishers (taxis in the proposed example) which are responsible to inform the Mediator if there is some change in the provided service (e.g., the taxi location or the number of current

passengers). Thus, instead of pulling data at consumer's request time, the Mediator knows at any time the status of all services, being able to join user requests with the event stream coming from the taxis, using temporal join statements expressed through SQL-like expressions.

1.3 IoT Testbeds and Applications

1.3.1 IoT Testbeds

Due to recent development and innovation in both hardware and software, the global network of networks, or IoT, is finally becoming a reality. The IoT's diverse billions of communicating devices, or smart objects, enable a new paradigm of interactivity among all manner of things and people. One of the IoT's biggest hurdles is the monolithic nature and fragmentation of existing vertical closed systems, architectures, and application areas. To overcome this, researchers are defining and standardizing interoperability in communication protocols and device mechanisms to allow for more efficient interaction among all IoT components, and Internet Protocol version 6 (IPv6) [8] is emerging as the base network protocol for all IoT applications [20].

To foster IoT development and diffusion, applications are increasingly built around the well-known Web model, bringing about the so-called Web of Things (WoT). The Web-based approach helped to greatly expand the Internet, and will likely have the same effect on the IoT [21]. WoT applications rely on specific Web-oriented application-layer protocols similar to HTTP, such as CoAP and, more generally, protocols complying with the Representational State Transfer (REST) architectural style [3].

Whereas simulation tools typically focus on evaluating lower-layer communication protocols, in recent years several IoT testbeds have been deployed to evaluate IoT solutions in realistic smart environments under real-world conditions.

The IoT-Lab [22] is an example of this kind of testbed environment: it provides a very large-scale infrastructure with more than 2,700 wireless sensor nodes spread across six different sites in France, and is used to test protocols at the link and network layers and to collect performance results such as energy consumption or packet

delivery ratio. Although these lower-layer protocols have been widely investigated, additional efforts are needed to create new innovative services, promote long-term evolution of systems, and ensure the robustness of applications against changes that might occur over time, thereby furthering WoT development. To easily build applications for this environment, developers need the ability to work at a high level of abstraction and without worrying about low-level details.

Another relevant large-scale IoT testbed is SmartSantander [23], consisting of approximately 20,000 nodes deployed in different cities across Europe. With its orientation toward smart-city services and applications, however, this testbed focuses on environmental data collection and is thus not set up to allow experimentation on a fully addressable and resource-oriented WoT. In particular, SmartSantander does not consider direct and bi-directional interactions between humans and objects.

1.3.2 User Interfaces for IoT

UI Description Languages (UIDLs) have been investigated for a long time, initially with the goal of relieving developers from the task of manual GUI creation. In [24], some early classifications of model-based UI description frameworks are presented. In particular, the problem of automatic UI generation with the purpose of transforming hand-held computers into universal control devices (in e.g Televisions, VCRs or photocopiers) and improving the generation process introducing the concept of "smart templates" is addressed in [25, 26].

Research concerning UIDLs targeting IoT scenarios has not been fully explored and has started rather recently. Mayer et al. present a model-based interface description scheme aiming at generating UI [27]. The proposed UIDL, a high-level scheme for smart things, captures the semantics of the interaction between the user and the device, rather than modeling concrete interface elements. In this approach, each smart object should embed a JavaScript Object Notation (JSON) based UIDL description which can be used to provide an appropriate concrete interface to the user. Although this approach presents some similarities with the one proposed in [28], there are significant differences between the two. While the former focuses exclusively on the definition of a suitable UIDL format, the work described in [28] also consider re-

source discovery, which is considered as an integral part of the proposed mechanism. Moreover, this work provides support for IoT-oriented protocols, such as CoAP, to exchange data with smart objects, as well as HTTP. The aim of our approach is to enable users and consumers to easily interact with resources in an IoT network, without any a-priori knowledge of the system. Moreover, the proposed dynamic UI management is fully integrated and consistent with REST principles: UI descriptors are rich hypermedia and are considered as resources within the system's domain.

1.4 Data Management in Companies

Even if not directly related to the Internet of Things scenario, also many companies and industries face on one hand the problem of a high rate information incoming flow, and on the other hand, the real-time and low latency requirements of final consumers.

A remarkable example is the management of data flow related to waste and recyclable materials tracking. The company Multiraccia S.C. (Reggio Emilia, Italy) developed the "CGO - Cobat Gestionale Online" software, a web application with the aim to efficiently handle all request coming from the different kind of users which need to interact system to track a garbage movement or treatment. The software is thus used by a large number of users (about 2,800), with different roles:

- Materials Producers / Manufacturers: that should declare the quantity of produced item which in future will become dangerous or recyclable waste like batteries, tires, solar panels or components containing lead;
- Garbage Gatherer: which are requested to collect and eventually temporarily store used materials in one of their warehouse and track the movement;
- Garbage Carriers: that carry out the transportation of collected waste loads from a source (i.e. a Producer) to a destination (i.e. a Gatherer);
- Smelters: which finally dispose of waste, with different processing activities depending on the kind of material;

- Administrative Users: which are required to check and verify all operations made by other operative users. Moreover, administrative accounts can use the software to generate reports and manage all monetary aspects related to these activities;

The totality of CGO users' work generate a flow of incoming requests ranging from 3,000 to 5,000 per day and require the storage of about 10 Gb of data per year.

The application has been developed in using different technologies, in particular PHP, with the support of the CodeIgniter [29], an open source web Framework based on the Model View Controller (MVC) pattern, which enables developers to easily create fully-featured applications. Databases in CGO are implemented with MySQL [30] and Filemaker [31], a cross-platform relational database application from FileMaker Inc., integrating both a database engine and a Graphical User Interface (GUI), allowing users to modify the database with graphical design tools.

Chapter 2

The Big Stream Architecture

2.1 Big Stream Paradigm

IoT application scenarios are characterized by a huge number of data sources sending small amounts of information to a collector service at a typically limited rate. Many services can be built upon these data, such as: environmental monitoring, building automation, and smart cities applications. These applications typically have real-time or low-latency requirements in order to provide efficient reactive/proactive behaviors, which could effectively be implemented especially in an IP-based IoT, where smart objects can be directly addressed.

Applying a traditional Big Data approach for IoT application scenarios might bring to higher or even unpredictable latencies between data generation and its availability to a consumer, since this was not among the main objectives behind the design of Big Data systems. As stated in [15], a major difference between Big Data and Big Stream approaches resides in the real-time/low-latency requirements of consumers. The gigantic amount of data sources in IoT applications has mistakenly made developers and implementors believe that re-using Big Data-driven architectures would be the right solution for all applications, rather than designing new paradigms specific for IoT scenarios.

Figure 2.1 illustrates the time contributions introduced when data pushed by

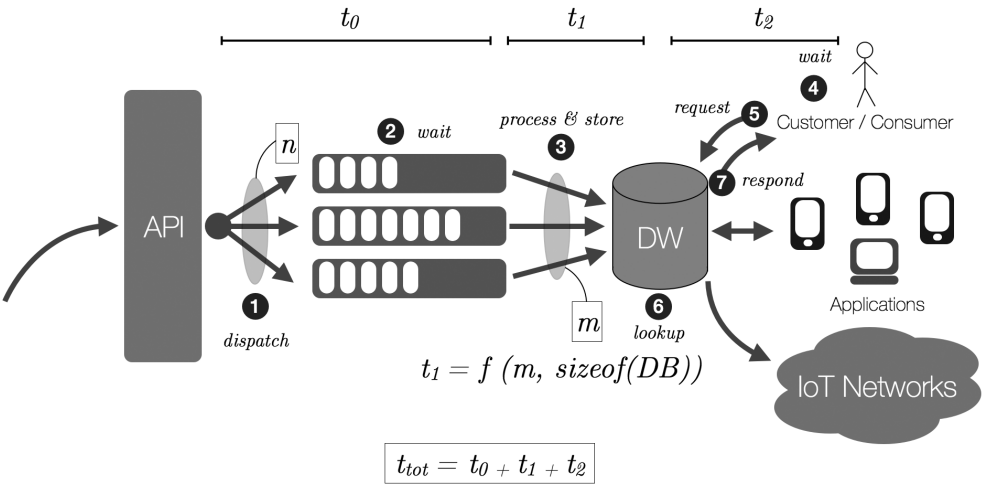


Figure 2.1: Traditional Big Data architecture for IoT and delay contributions from data generation to applications information delivery.

smart objects need to be processed, stored, and then polled by consumers. The total time required by any data to be delivered to a consumer can be expressed as

$$T = t_0 + t_1 + t_2, \text{ where:}$$

- t_0 is the time elapsed from the moment a data source sends information, through an available API, to the Cloud service (1) and the service dispatches the data to an appropriate queue, where it can wait for an unpredictable time (2), in order to decouple data acquisition from processing;

- t_1 is the time needed for data, extracted by the queue, to be pre-processed and stored into a Data Warehouse (DW) (3); this time depends on the number of concurrent processes that need to be executed and get access the common DW and the current size of the DW;

- t_2 is the data consumption time, which depends on the remaining time that a polling consumer needs to wait before performing the next fetch (4), the time for a request to be sent to the Cloud service (5), the time required for lookup in

the DW and post-process the fetched data (6), and the time for the response to be delivered back to the consumer (7).

It can be observed that the architecture described is not optimized to minimize the latency and, therefore, to feed (possibly a large number of) real-time applications, but, rather, to perform data collection and batch processing. Moreover, it is important to understand that data significant for Big Stream applications might be short-lived, since they are to be consumed immediately, while Big Data applications tend to collect and store massive amounts of data for an unpredictable time.

In [15] a novel architecture explicitly designed for the management of Big Stream applications targeting IoT scenarios is firstly presented. The main design criteria of the proposed architecture are: (i) the minimization of the latency in data dispatching to consumers and (ii) the optimization of resource allocation.

The main novelty in the proposed architecture is that the data flow is "consumer-oriented", rather than being based on the knowledge of collection points (repositories) where data can be retrieved. The data being generated by a deployed smart object might be of interest for some consumer application, denoted as listener. A listener registers its interest in receiving updates (either in the form of raw or processed data) coming from a streaming endpoint (i.e., Cloud service). On the basis of application-specific needs, each listener defines a set of rules, which specify what type of data should be selected and the associated filtering operations. For example, in a smart parking application, a mobile app might be interested in receiving contents related only to specific events (e.g., parking sensors status updates, the positions of other cars, weather conditions, etc.) that occur in a given geographical area, in order to accomplish relevant tasks (e.g., find a covered free parking spot).

The pseudo-code that can be used to express the set of rules for the smart parking application is shown in the following listing:

```
when
$temperatureEvent = {@type:http://schema.org/Weather#temperature}
$humidityEvent = {@type:http://schema.org/Weather#humidity}
$carPositionEvent = {@type:http://schema.org/SmartCar#travelPosition}
```

```
$parkingStatusEvent = {@type:http://schema.org/SmartParking#status}

@filter: {
location: { @type:"http://schema.org/GeoShape#polygon",
coordinates: [ [
[41.3983, 2.1729], [41.3986, 2.1729], [41.3986, 2.1734],
[41.3983, 2.1734], [41.3983, 2.1729]
] ]
}
then
<application logic>
```

[Pseudo-code to express the set of rules for a smart parking application.]

The set of rules specifies i) which kinds of events are of interest for the application and ii) a geography-based filter to apply in order to receive only events related to a specific area. Besides the final listener (end-user), at the same time, the Cloud service might act as a listener and process the same event data stream, but with different rules, in order to provide a new stream (e.g., providing real-time traffic information), which can be consumed by other listeners. An illustrative pseudo-code for a real-time traffic information application is presented in the following listing:

```
when
$cityZone = {@type:http://schema.org/SmartCity#zone}
$carPositionEvents = collect({
@type: http://schema.org/SmartCar#travelPosition,
@filter: {
location: cityZone.coordinates
}
}) over window:time(30s)
then
emit {
@type: http://schema.org/SmartCity#trafficDensity,
city_zone: $cityZone,
density: $carPositionEvents.size,
}
```

[Pseudo-code to express the set of rules for a real-time traffic information application.]

The proposed Big Stream architecture guarantees that, as soon as data are available, they will be dispatched to the listener, which is thus no longer responsible to poll data, thus minimizing latencies and possibly avoiding network traffic. The information flow in a listener-based Cloud architecture is shown in Figure 2.2 With the new paradigm, the total time required by any data to be delivered to a consumer can be expressed as:

$$T = t_0 + t_1 \text{ where:}$$

- t_0 is the time elapsed from the moment a data source sends information, through an available API, to the Cloud service (1) and the service dispatches the data to an appropriate queue, where it can wait for an unpredictable time (2), in order to decouple the data acquisition from processing;

- t_1 is the time needed to process data extracted from the queue and be processed (according to the needs of the listener, e.g., to perform format translation) and then deliver it to registered listeners.

It is clear that the inverse of perspective introduced by a listener-oriented communication is optimal in terms of minimization of the time that a listener must wait before it receives data of interest. In order to highlight the benefits brought by the Big Stream approach, with respect to a Big Data approach, consider an alerting application, where an event should be notified to one or more consumers in the shortest possible time. The traditional Big Data approach would require an unnecessary preprocessing/storage/post-processing cycle to be executed before the event could be made available to consumers, which would be responsible to retrieve data by polling. The listener-oriented approach, instead, guarantees that only the needed processing will be performed before data are being delivered directly to the listener, thus providing an effective real-time solution. This general discussion proves that a consumer-oriented paradigm may be better suited to real-time Big Stream applications, rather

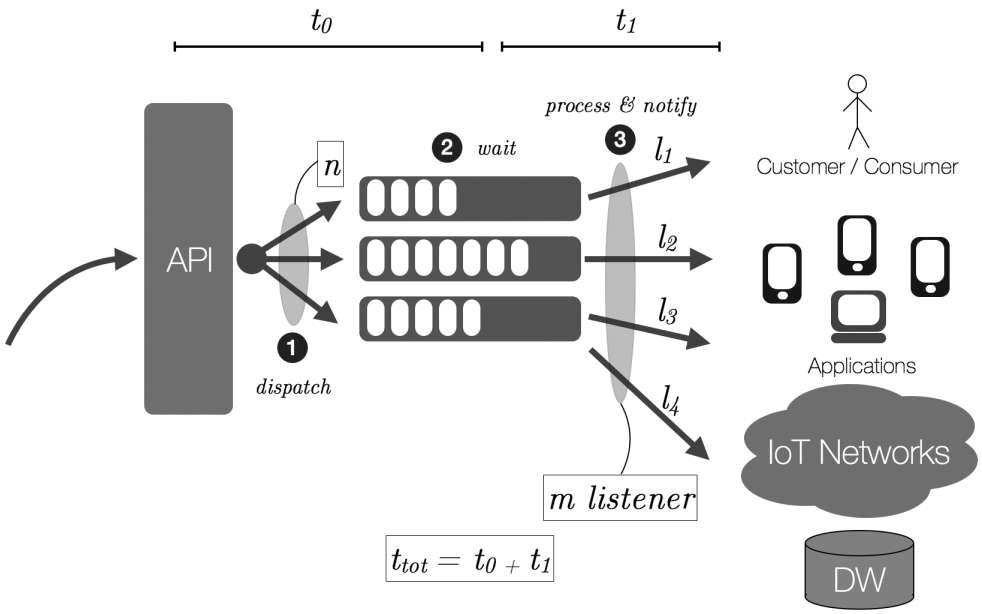


Figure 2.2: The proposed listener-based architecture for IoT delay contributions from data generation to consumers information delivery are explicitly indicated.

than simply reusing existing Big Data architectures, which fit best applications that do not have critical real-time requirements.

Finally, in order to clarify all aspect related to the new paradigm, in the following are summarized the differences between the Big Stream and the Big Data paradigm, while both deal with massive amounts of data.

- **The Meaning of the Term "Big:"** In Big Data it refers to "volume of data," while in Big Stream it refers to "data generation rate."
- **Real-Time or Low-Latency Requirements of Different Consumers:** They are typically not taken in account by Big Data.
- **Nature of Data Sources:** Big Data deals with heterogeneous data sources in a wide range of different areas (e.g., health, economy, natural phenomena), not necessarily related to IoT. Instead, Big Stream data sources are strictly related to the IoT, where heterogeneous devices send small amounts of data generating, as a whole, a continuous and massive information stream.
- **Objective:** Big Data focuses on information management, storage and analysis, following the data-information-knowledge model [32]; Big Stream, instead, focuses on the management of data flows, being specifically designed to perform real-time and ad-hoc processing, in order to forward incoming data streams to consumers.

2.2 The Graph Architecture

In order to overcome the limitations of the "process-oriented" approach described in the previous section, a new Cloud Graph-based architecture has been envisioned and designed. The architecture is built on top of basic building blocks that are self-consistent and perform "atomic" processing on data, but that are not directly linked to a specific task. In such systems, the data flows are based on dynamic graph-routing rules determined only by the nature of the data itself and not by a centralized coordi-

nation unit. This new approach allows the platform to be "consumer-oriented" and to implement an optimal resource allocation.

Without the need of a coordination process, the data streams can be dynamically routed in the network by following the edges of the graph and allowing the possibility to automatically switch-off nodes when some processing units are not required at a certain point and transparently replicate nodes if some processing entities is consumed by a significant amount of concurrent consumers.

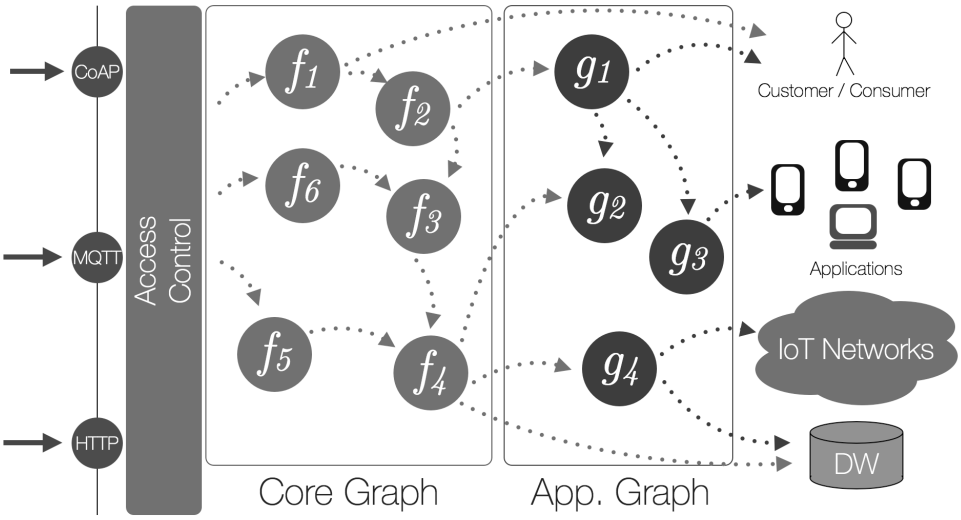


Figure 2.3: The proposed listener-based Graph architecture: the nodes of the graph are listeners; the edges refer to the dynamic flow of information data streams.

In order to minimize the delay between the instant of raw data generation (e.g., at sensors) and the instant at which information is notified to the final consumer, the proposed platform, illustrated in Figure 2.3, adheres to the publish/subscribe model and is based on the concept of "listener."

A listener is an entity (e.g., a processing unit in the graph or an external consumer) interested in the raw data stream or in the output provided by a different node in the graph. Each listener represents a node in the topology and the presence and

combination of multiple listeners, across all processing units, defines the routing of data streams from producers to consumers More in detail, the fundamental components of the Graph-based Cloud architecture are the following.

- **Nodes:** processing units which process incoming data, not directly linked to a specific task. A Graph node can be, at the same time, a listener of one or more streams, coming from other nodes, and a publisher of a new stream, that could be of interest to some other nodes.

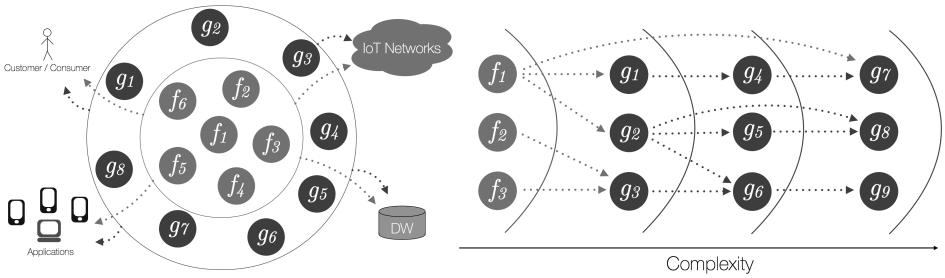
- **Edges:** flows of informations (streams) linking together various nodes: this allows complex processing operations.

In order to provide a set of commonly available functionalities, while allowing to dynamically extend the capabilities of the system, the graph is organized on concentric levels, composed by two different kind of nodes:

- **Core Graph Nodes:** implement basic processing operations, provided by the architecture, and are deployed into inner layers of the Graph.

- **Application Graph Nodes:** represent the building blocks of outer layers, listening and accepting flows coming from nodes belonging to inner Graph layers. This type of nodes always receive already processed streams.

Layers in the Graph are characterized by an increasing degree of complexity. This means that, as shown in Figure 2.4, data streams generated from nodes in a generic layer of the Graph, can be used by nodes in higher-degree layers, to perform more complex processing operations, generating new data streams which can be used in higher layers, and so on. As a general rule, to avoid loops, nodes at inner Graph layers cannot be listeners of nodes of outer Graph layers. In other words, there can be no link from an outer Graph node to an inner Graph node, but only vice versa. Same-layer Graph nodes can be linked together if there is a need to do so. Figure 2.5 illustrates incoming and outgoing listener flows between Core and Application graphs units In particular, a Core Graph node can be a listener only for other nodes of the



(a) (b)

Figure 2.4: (a) The concentric linked Core and Application Graphs. (b) Basic processing nodes build the Core Graph: the outer nodes have increasing complexity.

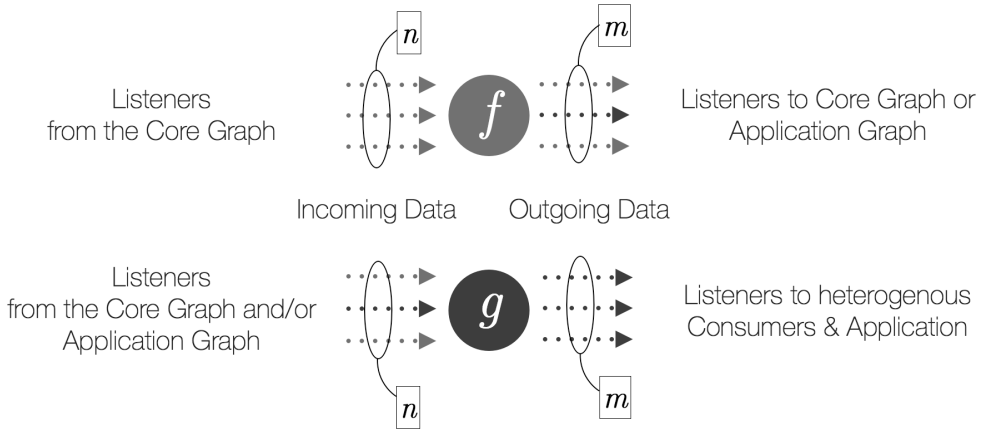


Figure 2.5: Allowed input and output flows for Core Graph nodes and Application Graph nodes.

same layer and a source both for other Core or Application Graph nodes. In other words, we consider only acyclic graphs.

The proposed architecture has been explicitly designed for the management of Big Stream applications targeting IoT scenarios. It aims at decreasing the latency in data dispatching to consumers and optimizing resource allocation.

Figure 2.6 illustrates the architectural components defining the proposed system and the relationships between each element. The next subsections describe in detail all the building blocks. The complete architecture is shown in Figure 2.7.

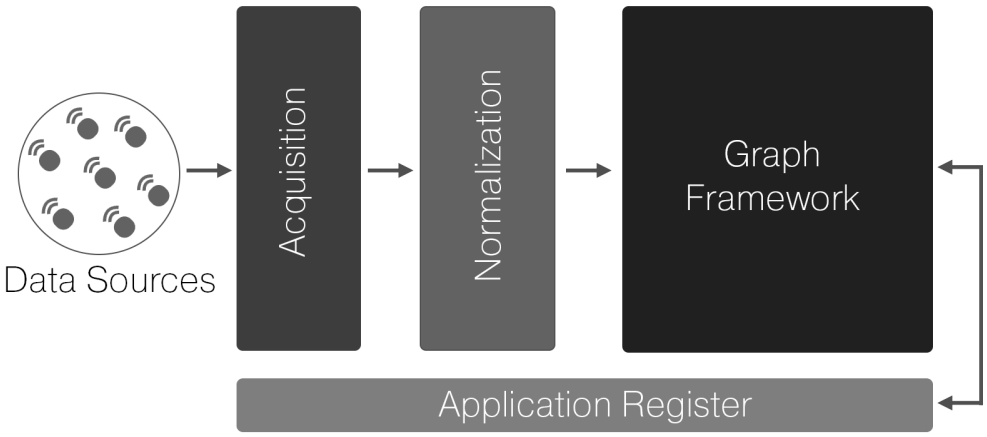


Figure 2.6: Components of the proposed Graph Cloud architecture and relations between each element.

2.2.1 The Acquisition Module

The Acquisition Module represents the entry point for external IoT networks of smart objects to the Cloud architecture. Its purpose is to collect raw data from different and heterogeneous data sources and make them available to the other functional blocks. It is important to underline that several application-layer protocols can be implemented by smart objects. For this reason, the Acquisition Module includes a set of different connectors in order to properly handle each protocol-specific incoming data stream.

2.2.2 The Normalization Module

Raw data are generally application-dependent, thus a Normalization Module has been designed in order to normalize all the collected information and generate a representation suitable for processing. The normalization procedure is made by fundamental and atomic operation on data such as: i) the suppression of useless information (e.g., unnecessary headers or meta-data); ii) the annotation with additional information; and iii) the translation of the payload to a suitable format. In order to handle the huge amount of incoming data efficiently, the normalization step is organized with protocol-specific queues and Exchanges. An Exchange works as a router in the system and dispatches incoming data to one or more output queues depending on dynamic routing rules. As shown in the normalization section of Figure 2.7, the information flow originating from the Acquisition Module is handled as follows:

- all data streams relative to a specific protocol are routed to a dedicated protocol-specific exchange, which forwards them to a protocol-dedicated queue;
- a normalization process handles the input data currently available on the queue and performs all necessary normalization operations in order to obtain a stream of information units that can be processed by next modules;
- the normalized stream is forwarded to an output exchange; The output of the Normalization block represents the entry-point of the first Graph Module Exchange, that pass it to all the interested listeners of the next levels. The main advantage of using Exchanges is that queues and normalization processes can be dynamically adapted to the current workload; for instance, normalization queues and processes could be easily replicated to avoid system congestion.

2.2.3 The Graph Framework

The Graph Framework is composed of listeners. Each listener represents a node in the topology. The connection of multiple listeners across all processing units define the routing of data streams from producers to consumers. The nodes are processing units performing some kind of computation on incoming data and edges represent the

flow of information linking together processing units, which implement some complex behavior as a whole. As shown in Figure 2.3, the graph is divided in two stages: i) Core Graph Nodes provide basic processing (e.g., format translation, normalization, aggregation, data correlation, and other transformations); ii) Application Graph Nodes require data coming from the core or an inner graph level to perform custom processing on already processed data are defined.

In the "Graph Framework", each level is accessible from a level-dedicated Exchange that forwards all data streams to nodes in its level. Each graph node *i* in a specific layer *n* can listen for incoming data stream on a dedicated queue managed by the Exchange of level *n*. If the node *i*, as well as being a consumer, it acts also as a publisher, then its computation results are delivered to the Exchange of level *n*, which is bounded with the Exchange of layer *n+1*. Therefore the Exchange in level *n+1* can forward streams coming from level *n* to all nodes of level *n+1* interested in this kind of data.

2.2.4 The Application Register Module

The Application Register Module has the fundamental responsibility to maintain the information about the current state of all graph nodes in the system, and to route data across the graph. In more detail, the application register module performs the following operations:

- attach new nodes or consumer applications interested in some of the streams provided by the system;
- detach nodes of the graph that are no more interested in streaming flows and eventually re-attach them;
- handle nodes that are publishers of new streams;
- maintain information regarding topics of data, in order to correctly generate the routing-keys and to compose data flows between nodes in different graph levels.

In order to accomplish all these functionalities, the Application Register Module is composed by two main components, as shown in Figure 2.7. The first one is the Graph State Database, which is dedicated to store all the information about active graph nodes, such as their state, level, and whether they are publishers. The second one is the Node Registration and Queue Manager (NRQM), which handles requests from graph nodes or external process, and handles the management of queues and the routing in the system. When a new process joins the graph as a listener, it sends an attach request to the Application Register Module, specifying the kind of data to which it is interested. The NRQM module stores the information about a new process in the Graph State Database and creates a new dedicated input queue for the process, according to its preferences. Finally, the NRQM sends a reference of the queue to the process, which becomes a new listener of the graph and can read the incoming stream from the input queue. After this registration phase, the node can perform new requests (e.g., publish, detach, get status), which are detailed next.

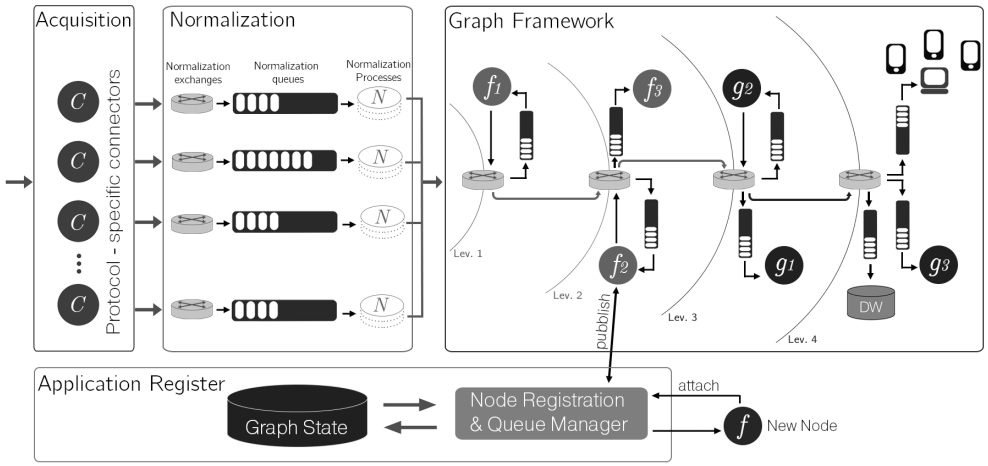


Figure 2.7: The complete Graph Cloud Architecture with reference to the data stream flow between all building blocks, from IoT data sources to final consumers.

The designed graph-based architecture allows to optimize resource allocation in terms of efficiency, by switching off processing units that have no listeners registered to them (enabling cost-effectiveness), and scalability, by replicating those processing

units which have a large number of registered listeners. The combination of these two functionalities and the concept of listener allow the platform and the overall system to adapt itself to dynamic and heterogeneous scenarios by properly routing data streams to the consumers and add new processing unit and functionalities on demand. In Figure 2.7, all the architecture modules with the complete flow of information through all steps described above are presented in detail.

2.3 The Graph Architecture Implementation

In [33], a first implementation of the Big Stream architecture with open-source technologies has been presented. Three main modules concur in forming the entire system: (i) the Acquisition and Normalization modules; (ii) the Graph Framework module; (iii) the Application Register module.

The implementation has been carried out by deploying an Oracle VirtualBox R VM, equipped with Linux Ubuntu 12.04 64-bit, 2GB RAM, 2 CPU, 10GB HDD.

Since the architecture is based on a queue-communication paradigm, an instance of RabbitMQ [34], an open-source queue server implementing the standard Advanced Message Queuing Protocol (AMQP) [35], was used. RabbitMQ provides a multi-language (Java, PHP, Python, C, ...) and platform indipendent API.

In the following the implementation of each fundamental building block is described in detail.

2.3.1 Acquisition Nodes and Normalization Nodes implementation

The system needs an input block capable to handle external incoming raw data, through different application-layer protocols. Data must then be processed and structured, in order to be managed by the graph processes.

Acquisition Nodes

Considering the main and most widespread IoT application layer protocols, the current implementation supports: i) HTTP, ii) CoAP, and iii) MQTT. For the sake of

scalability and efficiency, an instance of Nginx [36] has been adopted as HTTP acquisition node reachable via the default HTTP port. As a processing module, a dedicated PHP page has been configured to forward incoming data to the inner queue server. Nginx has been selected instead of the prevailing and well-known open source Apache HTTP Server Project [37] because it uses an event-driven asynchronous architecture to improve scalability and specifically aims to reach high-performances even in case of critical number of request. CoAP acquisition node has been implemented using a Java process, based on a mjCoAP [38] server instance connected to the RabbitMQ queue server. MQTT acquisition node is realized by implementing an ActiveMQ [39] server through a Java process, listening for incoming data over a specific input topic (*mqtt.input*). This solution has been preferred over other existing solution (e.g., the C-based server Mosquitto [40]), because it provides a dedicated API that allows a custom development of the component. The MQTT acquisition node is also connected to the architecture's queue server. In order to avoid potential bottlenecks and collision points, each acquisition protocol has a dedicated Exchange and a dedicated queue (managed by RabbitMQ), linked together with a protocol related routing key, ensuring the efficient management of incoming streams and their availability to the normalization nodes. The details of this module, are shown in Figure 2.8.

Normalization Nodes

Incoming raw data from the acquisition nodes may require a first optimization process, aiming at structuring them into a common and easily manageable format. As shown in Figure 2.8, Normalization processes extract raw data from dedicated incoming queues, leaving the routing key, which identifies the originator smart object protocol, unchanged. Each normalization node is implemented as a Java process, which processes incoming raw data extracted from a queue identified through a protocol-like routing key (e.g., <*protocol*>.*event.in*). Received data are fragmented and encapsulated in a new JSON structure, which provides an easy-to-manage format. At the end of the processing chain, each normalization node forwards the new data chunk to its next Exchange, which connects the normalization block to the first

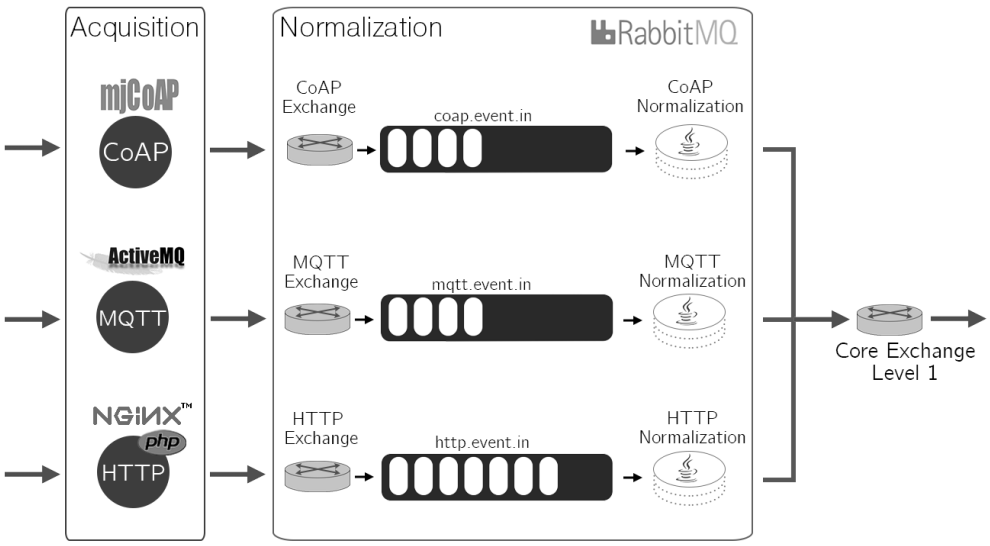


Figure 2.8: Acquisition and Normalization blocks.

Graph layer Exchange.

2.3.2 Graph Management Implementation

Incoming messages are stored into active queues, connected to each Graph Layer's Exchange. Queues can be placed into the Core Graph layers, for basic computation, or into Application Graph layers, for enhanced data treatment.

Layers are connected with one-way links with their own successor Exchange by using the binding rules allowed by queue manager, ensuring proper propagation of data flows and avoiding loops. Each graph layer is composed by Java-based Graph Nodes dedicated to process data provided by the Graph layer's Exchange. Such processes can either be Core Nodes, if they are dedicated to simple and primitive data processing, or Application Nodes, if they are oriented to a more complex and specific data management. Messages, identified by a routing key, are first retrieved from the layer's Exchange, then processed, and finally sent to the target Exchange, with a new work-related routing key, as depicted in Figure 2.9.

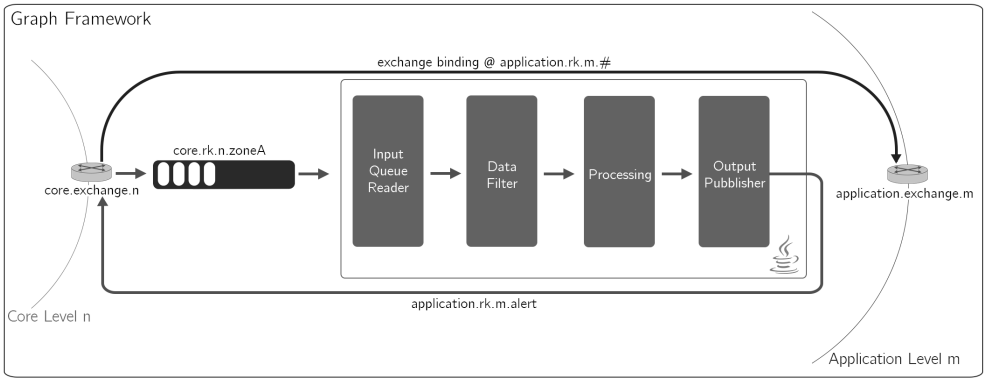


Figure 2.9: Interaction between Core and Application layers with binding rule.

If the outgoing routing key belongs to the same incoming graph layer, data object or stream stay within the same Exchange and becomes available for other local processes. If the outgoing routing key belongs to an outer graph layer, then data are forwarded to the corresponding Exchange, and finally forwarded by following a binding rule and assuring data flow.

Each graph node, upon becoming part of the system, can specify if it acts as a data publisher, capable of handling and forwarding data to its layer's Exchange, or if it acts as data consumer only. A data flow continues until it reaches the last layer's Exchange, responsible to manage the notification to the external entities that are interested in final processed data (ex. Data Warehouse, browsers, smart entities, other cloud graph processes, ...).

2.3.3 Application Register Implementation

The overall state of the architecture is managed by the Application Register, a Java process, that has the role to coordinates the interactions between graph nodes and external services, like the RabbitMQ queue server and the MySQL database. It maintains and updates all information and parameters related to processing unit queues.

As a first step, the Application Register starts up all the external connections, then it activates each layer's Exchange, binding them with their successors. At the end,

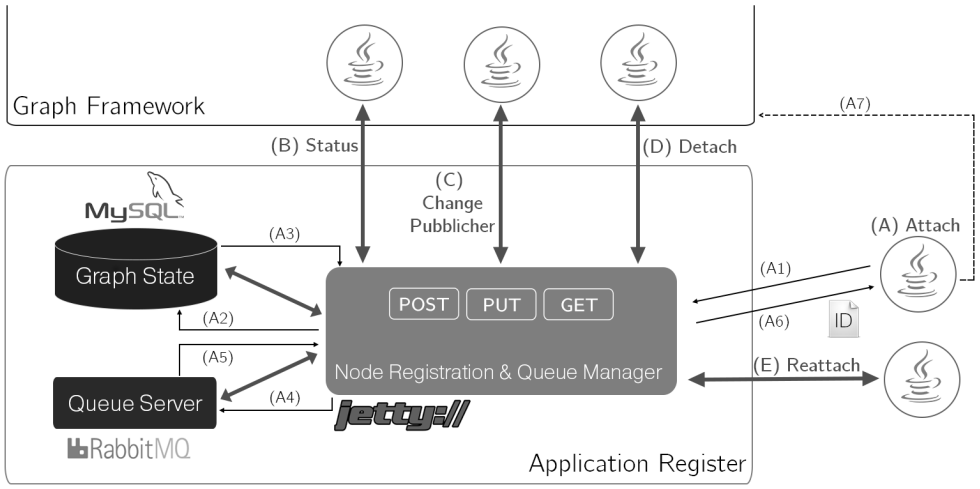


Figure 2.10: Detail of Application Register module, with possibles actions required by graph nodes, deepening steps for ATTACH request.

it proceeds with the activation of a Jetty HTTP server, responsible for listening and handling all Core and Application nodes requests. More in particular, the Application Register module menages the following requests (as depicted in Figure 2.10):

- (A) Attach new nodes or consumers, interested in some of the streams provided by the platform;
- Menage status requests from nodes (B);
- Detach (D) nodes from the Graph, when they are no longer interested in receiving flows, and, possibly, re-attach them (E);
- Handle nodes that are publishers of new streams (C);
- Maintain information regarding topics of data, in order to correctly generate the routing keys, and to compose data flows between nodes in different Graph layers.

All requests sent by nodes in the Graph are HTTP requests following the REST paradigm.

The proposed architecture can be deployed on a single server. However, the exploitation on the Cloud is preferable, in order to better scale the system and manage the workload in presence of a huge number of data sources and processing nodes. Another important benefit, related to exploiting Cloud Computing, is that the Cloud provides a common platform in which data streams can be shared among several actors (e.g., IoT data-sources, developers or consumers), in order to build useful services for Smart Cities, composing and aggregating different data streams.

Although the proposed architecture has not been deployed on the Cloud yet, the proposed architecture is oriented to all developers interested in creating new useful services on top of IoT-generated data. For this reason, the platform will provide user-friendly tools to upload to the Cloud custom processing units (namely nodes) using currently available streams (already processed or not) as inputs, generating a new edge of the Graph which can be potentially employed from other developers. In this scenario, the paths are thus automatically generated on the basis of the needs and interests of each node. Therefore, the proposed architecture can be applied to all scenarios in which scalability is required. IoT Big Stream applications have been targeted because they represent an innovative field of research, which has not been fully explored.

2.4 Test and Results

The implemented architecture has been evaluated in [33] through the definition of a real use case, represented by a smart parking scenario. The data traces used for the evaluation of the proposed architecture have been provided by Worldsensing [41] from one of the company's deployments in a real-life scenario, used to control parking spots on streets. The traces are a subset of an entire deployment (more than 10,000 sensors) with information from 400 sensors over a 3-month period.

2.4.1 Experimental Setup

Each of 604k parking spot's data has been used in the cloud infrastructure using a Java-based data generator, which periodically selects an available protocol (HTTP,

CoAP or MQTT) on a random basis and sends raw data to the corresponding acquisition node interface.

Once the raw data have been received by the acquisition layer, they are forwarded to the dedicated normalization Exchange, where corresponding nodes enrich incoming data with parking zone's details, retrieved from an external database. Once the normalization module has completed its process, it sends the structured data to the Graph Framework, allowing the processing of the enriched data.

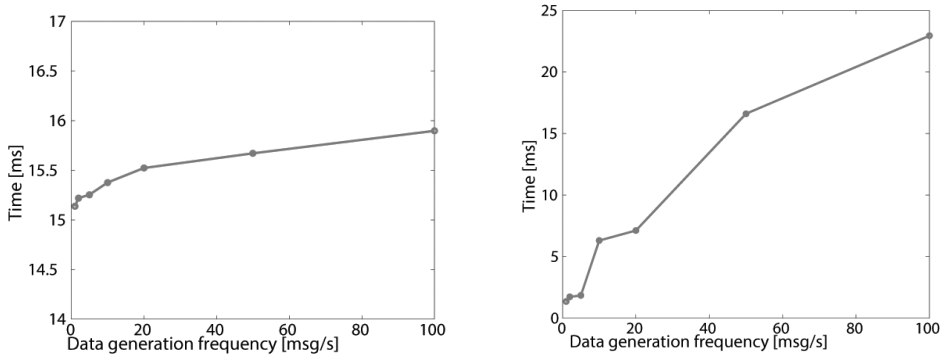
This Graph Framework is composed by 7 Core Nodes and 7 Application Nodes. The processed data follows a path based on routing keys, until reaching the architecture's final external listener. Each Application node is interested in detecting changes of parking spot data, related to specific parking zones. Upon a change of the status, the Graph node generates a new aggregated descriptor, which is forwarded to the responsible layer's Exchange, which has the role to notify the change event to external entities interested in the update ($free \rightarrow busy, busy \rightarrow free$).

2.4.2 Results

The proposed architecture has been tested, using the testbed described above, by varying the inter-arrival time of each incoming raw data from 1 message per second, up to 100 messages per second. The evaluation consists in assessing the performance of i) the acquisition stage and ii) computation stage.

First, performance evaluation has been made measuring the time period between points in time when data objects are sent from a data generator to the corresponding acquisition interface, and a point in time when the object is enriched by normalization nodes, thus becoming available for the first processing Core Node. The results are shown in Figure 2.11(a). The acquisition time is slightly increasing but it is around 15 ms at all considered rates.

The second performance evaluation has been carried out by measuring the time (dimension: [ms]) between the instant in which enriched data become ready for processing activities, and the time when the message ends its Graph Framework routes, becoming available for external consumers/customers. The results, shown Figure 2.11(b), have been calculated using the following expression:



(a) (b)

Figure 2.11: (a) Average times (dimension: [ms]) related to the acquisition block. (b) Average times (dimension: [ms]) related to Graph Framework processing block.

$$T_{processing_{freq}} = \frac{T_{out} - T_{in} - \sum_{i=1}^{N} graph\_process_i}{N}$$

Performance results were calculated by subtracting the processing time of all Core and Application Nodes, in order to consider only the effective overhead introduced by the architecture, and without considering implementation-specific times. Finally, these times have been normalized over the number of computational nodes, in order to obtain the per-node overhead introduced by the architecture, in a way that is independent of the specific routing that was implemented.

2.5 Solutions and Practical Use

The described architecture is designed with reference to a specific IoT scenario with strict latency and realtime requirements, namely a smart city-related Smart Parking scenario. There are several possible use cases and applications fitting this scenario, alerting or real time monitoring applications.

Vilajosana et al., in [42], have shown how Smart Cities are having difficulties in real deployment, even though obvious factors justify the necessity and the usefulness

of making cities smarter. The authors of [42] analyze in detail the causes and factors which act as barriers in the process of institutionalization of smart cities, and propose an approach to make smart cities become a reality. More in detail, three different stages are advocated in order to deploy smart cities technologies and services.

- **The bootstrap phase:** This phase is dedicated to offer services and technologies that are not only of great use and really improve urban living, but also offer a return on investments. The important objective of this first step is, thus, to set technological basis of the infrastructure and guarantee the system long life by generating cash flows for future investments.

- **The growth phase:** In this phase, the finances generated in the previous phase are used to ramp up technologies and services which require large investments and not necessarily produce financial gains but are only of great use for consumers.

- **The wide adoption phase:** In this third phase, collected data are made available through standardized APIs and offered by all different stakeholders to third party developers in order to create new services. At the end of this step, the system becomes self-sustainable and might produce a new tertiary sector specifically related to services and applications generated using the underlying infrastructure.

With reference to the third phase, three main different business models are proposed to handle the delivery of informations to third parties.

- The App store-like model: developers can build their apps using a set of verified APIs after a subscription procedure which might involve some subscription fee. IoT operators can hold a small percentage of gains of Apps published in Apple and/or Android market.

- The Google Maps-like model: the percentage fee on apps sales price is scaled according to the number and granularity of the queries to deployed APIs.

- The Open data model: this model grants access to APIs in a classical open data vision, without charging any fee to developers.

The architecture described in the previous sections is compatible with the steps described in the work of Vilajosana and, more specifically, it can adopt the "Google-maps-like" where infrastructure APIs make available different information streams with different complexity layers. The graph architecture, moreover, gives another opportunity to extend the business model, as developers can use available streams to generate a new node of the graph, and publish a new stream for the system.

In the previous sections, the implementation of the Graph-based Cloud architecture for a Big Stream IoT scenario has been detailed. Now, some aspects regarding practical use of the proposed architecture, taking into account its deployment on a Cloud platform are presented. The proposed architecture is mainly intended for developers, interested in building applications based on data generated by IoT networks, with real-time constraints, low-overhead, customizing paths and informations flows, in order to generate new streams, through the addition of newly developed and deployed Graph nodes. Analyzing the Cloud components of the platform, the preferred service model seems to be the Software-as-aService (SaaS) model, providing the following useful services for developers:

- node upload/deletion: to change the Graph Framework topology, loading or removing newly custom processing node;
- stream status: to get the list of all available streams generated by the graph;
- data source upload/deletion: to load or remove a new external data source before the Acquisition module of the Graph-based system.

It is important to observe that each developer, accessing the architecture, could operate on data streams coming from IoT networks (already processed or not) which he/she does not own. The interactions between IoT developers and the proposed Cloud architecture are similar to those provided by Node-RED, a WEB-based application, running on Node.js engine, which allows developers to create IoT graphs, wiring together hardware devices, APIs, and online services.

Chapter 3

Applying Security to the Graph Architecture

3.1 Analysis of Security Issues and Technologies

Addressing the security problem in the above Graph-based system entails a wide approach, owing to different needs of each specific component involved. In Figure 3.1, the main building blocks listed above are shown. The main components, in correspondence to which security mechanisms are required, are explicitly indicated. The enhanced Graph architecture, presented in [43] provides security by means of the following two modules.

- **Outdoor Front-end Security (OFS) Module:** Carries out security operations which could be applied a-priori, before receiving data from a generic external source, as well as before a final consumer can start interacting with the Graph-based Cloud platform.

- **In-Graph Security (IGS) Module:** Adopts security filters that could be applied inside the heart of the Graph-based Cloud platform, in order to make processing nodes able to control accesses to the streams generated by internal computational modules.

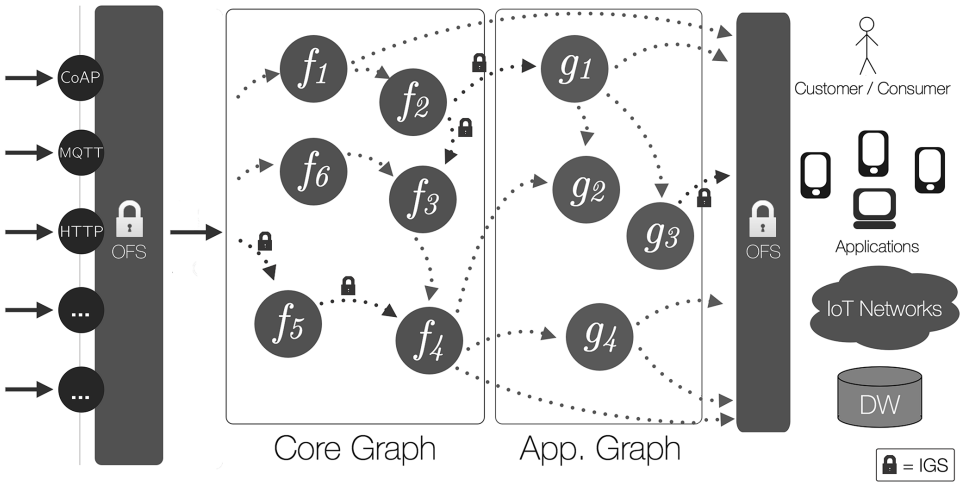


Figure 3.1: Main building blocks of the secured listener-based Graph architecture. Edges can be “open” or “secured”.

The OFS module is crucial for infrastructure safety: its role includes monitoring the access to the platform and authorizing the information flows coming from or directed to external entities. On one side, OFS must verify and authorize only desired external input data sources, allowing them to publish raw streams inside the IoT system. On the other hand OFS is required to secure outgoing streams, generated by different layers of the Graph platform itself, authorizing external consumers to use, if needed, the processed streams.

Consider, as an example, the case of the company “C” that owns a set of particular sensors, and wishes to become an IoT stream source for the Graph-based Cloud platform. However, this company requires (i) to sell sensed data only to a specific subset of customers, in order to protect its commercial interests, and (ii) to reach a profit from these sales. Therefore, the OFS module is strictly related to sensors and devices, at the input side, and to customers SOs, at the output stage, so that it becomes protocol-dependent and can be adapted to the specific technologies supported by the target devices.

The IGS module is not related to the OFS module, as it acts exclusively at the

heart of the IoT Graph architecture coordinating and managing inner inter-node interactions. The IGS module must be implemented inside single processing nodes enabling them to define a set of rules which describe what entities may become listeners of a generated stream. Referring to the Graph architecture, shown in Figure 3.1, edges in the Graph can be classified as follows.

- "Open" Edges: Data streams, generated by both Core or Application nodes in the Graph platform, that can be forwarded to all interested listeners without the need of isolation or access control.
- "Secured" Edges: Data streams that should comply with some specified rules or restrictions, which describe all possible consumers of generated data.

As an example, consider company "C," which provides its sensors as data sources and has notified the architecture that the streams produced by its sensors should be secured. The integration of security modules in the IoT architecture entails modifications in the structure and into the modules of the first (not secured) architecture described in [33].

In the nest sections, an analysis of each module composing the Graph architecture is presented, in order to explain how security mechanisms can be embedded and managed. In particular, is introduced the OFS module, which supports the Acquisition and Normalization modules on authorization of external entities. Moreover, an enhanced version of the Application Register is described, in order to underline the management of secure interaction with processing nodes. Finally, an overview inside Graph nodes, analyzing how security has been applied in processing stages, is presented.

3.2 Normalization after a Secure Stream Acquisition with OFS Module

The Acquisition and Normalization modules, shown in Figure 2.8, represent the entry point, for external sources (e.g., SOs deployed in different IoT networks), to the

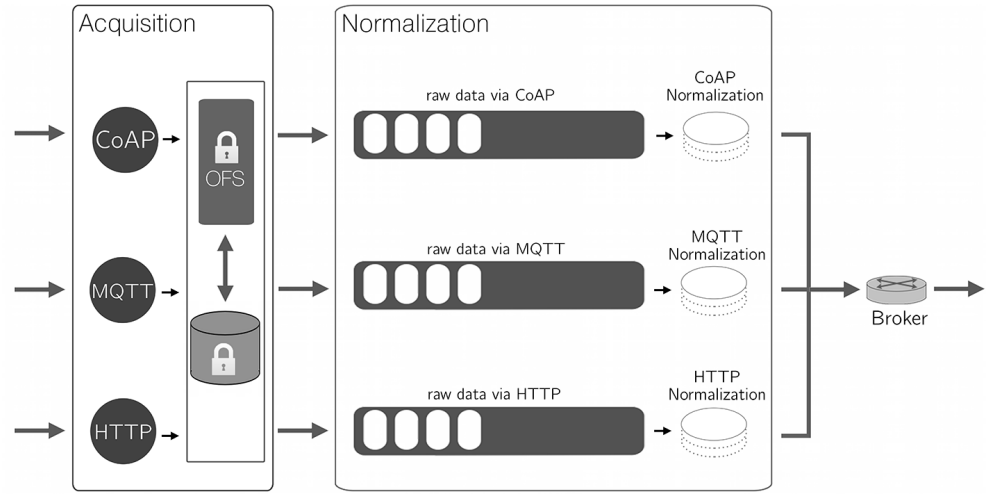


Figure 3.2: The OFS module manages security in the Acquisition and Normalization blocks, interacting with an authentication storage entity containing data sources identities.

proposed architecture.

The purpose of the Acquisition block is to receive incoming raw data from heterogeneous sources, making them available to all subsequent functional blocks. Since raw data are generally application- and subject-dependent, the Normalization block has to "normalize" incoming data, generating a common adapted representation, suitable for further processing (e.g., suppression of unnecessary data, information addiction, format translation). After the Normalization process, data are sent to the first Core layer. Within each Graph layer, streams are routed by dedicated components, denoted as brokers, which are layer-specific and, in the current implementation of the architecture, are represented by instances of RabbitMQ Exchanges. As stated before, SOs can communicate using different protocols. For this reason, the Acquisition block has to include a set of different connectors, one for each supported protocol, in order to properly handle each protocol-specific incoming data stream. Figure 3.2 shows how the acquisition and normalization modules have been modified in order to introduce security. These modules must cooperate with the OFS module, which

has to be activated before an external source becomes able to operate with the Graph platform. At the Acquisition stage, in order for the proposed IoT platform to support both "open" and "secured" communications, communication protocol-specific security mechanisms have to be implemented at proper layers (e.g., at the network layer through IPSec, at the transport layer through TLS/ DTLS, at the application layer through S/MIME or OAuth [44]). As stated before, the current implementation supports different application protocols at the Acquisition stage, namely: MQTT, HTTP, and CoAP. In order to secure all communications with these protocols, different protocol-specific policies must be introduced. In the work [45], an OAuth-based secure version of the MQTT protocol is proposed, showing that MQTT complies also with n-Legged OAuth protocol.

This means that the proposed IoT platform can provide a good way to authenticate external data providers, adopting open-source and well-known solutions. Therefore the OFS module can be secured by OAuth, being employed according to specific communication protocols supported by heterogeneous IoT SOs.

A suitable solution to provide authorization in IoT scenarios is IoT-OAS [46], which represents an authorization framework to secure HTTP/ CoAP services. The IoT-OAS approach can be applied by invoking an external OAuth-based Authorization Service (OAS). This approach is meant to be flexible, highly configurable, and easy to integrate with existing services, guaranteeing: (i) lower processing load with respect to solutions with access control implemented in the SO; (ii) fine-grained (remote) customization of access policies; and (iii) scalability, without the need to operate directly on the device. Although the OFS module has not been implemented yet, provided references show that several options are available.

Following the previous example, company "C", to became a secured IoT data source, selects one of the supported protocols (HTTP, CoAP or MQTT) to send raw data stream in the secured version.

3.3 Securing Application Register with IGS Module

One of the main motivations to secure a system internally is the need to secure some of its operations, as well as to isolate some processing steps of the entire stream management. The security features should be coordinated by the Application Register module, which maintains and manages interactions between inner Graph nodes of the IoT platform using different communication protocols, as requested by the architecture itself.

In order to accomplish the operational functionalities listed previously, the Application Register module has two main components, as shown in Figure 3.3.

- The Graph State database, responsible to maintain all information about the current Graph status. Since this component is not critical from a performance viewpoint, it has been implemented through a simple relational SQL database.
- The Node Registration and Queue Manager (NRQM) module, which is responsible to manage communications with existing Graph nodes, as well as with external entities that ask to join the Big Stream architecture.

To add security features to these modules, the Application Register defines entities and modules specifically related to security management and coordination. As shown in Figure 3.3, the Application Register is composed by the following additional modules:

- The **Policy Manager and Storage Validator (PMSV)**, responsible for managing and verifying authorization rules, interacts with a storage element, which persistently keeps updated authorization policies;
- The **Graph Rule Authorization Notifier (GRAN)** interacts with publisher Graph nodes and verifies if listener nodes are authorized to receive streams associated with specific topics;
- A **Persistent Storage Entity** (e.g., a non-relational database) maintains authorization rules specified by publisher Graph nodes.

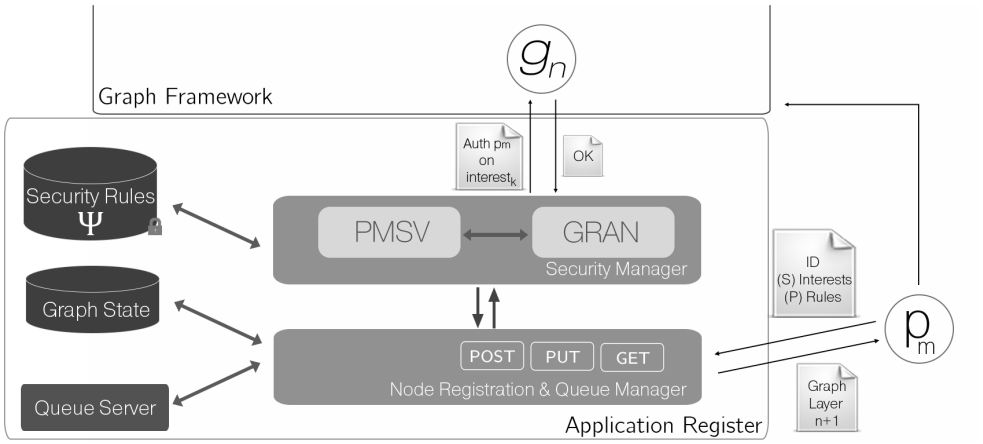


Figure 3.3: The Application Register module structure with security elements. PMSV and GRAN modules interact with a storage entity to manage authorization in the Graph.

While a processing node p_m asks to register to the IoT platform, requiring to join the Graph, after authentication (e.g., using a username/password pair, cryptographic certificates, ACLs, OAuth), there are two cases that require the use of security mechanisms and involve the defined modules:

- A registration request coming from a node that is willing to become a publisher node for a secured stream (e.g, an Application node created by developers of company “C”);
- A registration request sent by a node which asks to be attached as a listener for some streams.

In the first case, when an external process p_m requests to register to the Graph architecture, in order to secure one or more of its own streams, it updates the PMSV module. After indicating its published topics, it specifies some policies and rules, to be stored, together with the assigned operative Graph layer, into the persistent security storage, by PMSV itself. These rules will be checked in case of future subscription requests for the node.

In the second case, an external process p_m , upon issuing a request to attach to the Graph platform and to become a node, provides information related to its identity and also specifications about its interests, on which p_m asks to subscribe.

The Application Register, after having identified the Graph layer into which the new node could be placed, takes charge of these interests specifications, passing then to PMSV, that acts as follows.

For each provided $interest_k$, the PMSV module interacts with the persistent storage entity, making a lookup for a matching between interest and stored publishing policies, and refining this lookup with layer matching:

$$Match = \{layer = x\ OR\ layer(x+1)\} AND \{interest_k \in W\}$$

where: x stands for identified listener Graph layer; $interest_k$ indicates each single specified interest, extracted from attaching request; W represents the persistent storage element; and $Match$ contains a list of publisher nodes that have to authorize subscriptions.

If $Match$ contains some results (e.g., g_m node), these are forwarded to the GRAN module, which interacts with discovered publisher nodes, sending them the identity of the requesting listener node and asking them to allow or deny the subscription to the requested topics. This response is forwarded back to the GRAN, that analyzes it and, in compliance with the Application Register, authorizes or rejects the listener Graph node subscription.

In order to better explain the behavior of the Application Register module, in relation to the join operation of an external entity that asks to become a Graph listener, in the following Figure 3.4 the interactions between this external entity and the Application Register have been detailed, through a pseudocode representation.

The processing nodes in the Graph architecture can be, at same time, listeners as well as publishers, so that the previously detailed mechanisms could be applied together, without any constraint on the execution order. The flows shown in Figure 3.3 represent the interactions related to this mixed case. The rule on node authority, restricted to the same layer and to next one, decreases lookup times in rules matching execution.

```
NodeIdentityCertificate = {
  NodeInformations = {....};
  INTERESTS = {interest_1, interest_2, ...., interest_N};
}

MAIN() {
  Pm = receive_join_request(NodeIdentityCertificate);
  authenticated = authenticate_node(Pm);
  if (authenticated is TRUE) {
    x = identify_graph_layer_for_node(Pm);
    foreach (interest interest_k in INTERESTS) {
      Match = send_request_to_PMSV(layer = (x OR x+1), interest = interest_k);
      if (Match is EMPTY) {
        REPLY_DENY(topic = interest_k, destination_node = Pm);
      }
      else {
        foreach (GraphNode Gm in Match) {
          allow = request_grant_to_owner_via_GRAN(topic = interest_k, owner = Gm, listener = Pm);
          if (allow is FALSE) {
            REPLY_DENY(topic = interest_k, destination_node = Pm);
          }
          else {
            REPLY_SUCCESS(topic = interest_k, destination_node = Pm);
          }
        }
      }
    }
  }
}

REPLY_DENY(topic, destination_node) {
  send_DENY_response_to_destination(required_topic = topic);
}

REPLY_SUCCESS(topic, destination_node) {
  send_SUCCESS_response_to_destination(required_topic = topic, security_parameters = {....});
}
```

Figure 3.4: Pseudocode representation of the operations done by the Application Register, when an external process p_m asks to become a Graph listener in the Big Stream architecture.

Moreover, external SOs producers could also request a totally "secured" path, from source to final consumer. These constraints have a higher priority than policies defined by publisher Graph nodes, being forced by the stream generators. In this way, these external priority rules are stored into the persistent storage elements as well, and when a new Graph node registers to the proposed IoT platform, its Graph-related policies are left out, forcing these new nodes to comply with these external rules.

3.4 Securing Stream inside Graph Nodes

The Graph Framework is composed by several processing entities, which perform some kind of computation on incoming data, representing a single node in the Graph topology.

The connection of multiple listeners across all processing units defines the routing path of data streams, from producers to consumers. All nodes in the Graph can be input listeners for incoming data and output producers for other successor Graph nodes.

Since each Graph node is owner of the stream generated by its processing activity, it is reasonable to assume that it could decide to maintain its generated stream "open" and accessible to all its interested listeners, instead of applying securing policies, isolating its own information and defining a set of rules that restrict the amount of authorized listeners nodes. In this latter case, a "secured" stream should be created and encrypted using the algorithms selected by the owner.

Each listener is thus required to decrypt incoming data before performing any processing. These encryption/decryption operations could be avoided if listeners adopt homomorphic encryption [47], that allows to carry out computations on ciphertext, instead of on plaintext, generating an encrypted result that matches with one performed on the plaintext, without exposing the data to each of different steps chained together in the workflow. Homomorphic encryption allows to execute computation in the Encrypted Domain, providing end-to-end security, avoiding encryption/decryption hop-by-hop needs.

In Figure 3.5, the modules inside a Graph node are shown: the broker of the Core

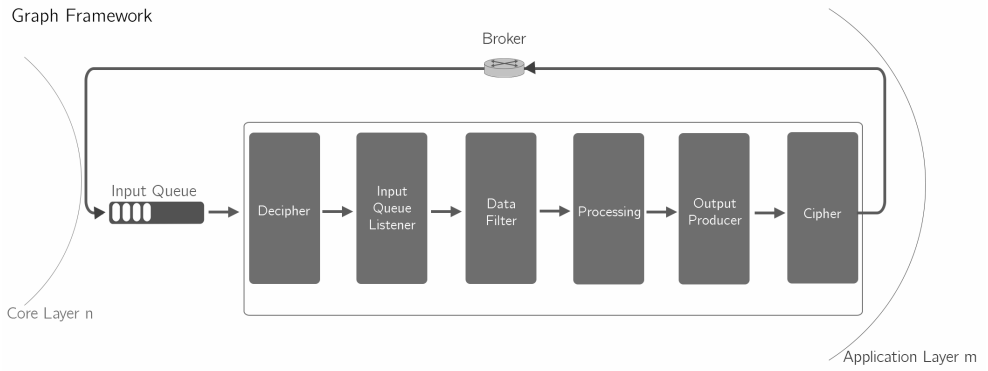


Figure 3.5: Pseudocode representation of the operations done by the Application Register, when an external process p_m asks to become a Graph listener in the Big Stream architecture.

layer n forwards streams to interested Graph nodes, forwarding these data into the input queue of the single node. The output stream, generated by the processing of this node, will be sent to the same broker of the Core layer n, which is linked to the broker of the next Graph layer, that "spreads" generated streams to all interested nodes. Some of these modules will be activated only in specific situations. In particular, the illustrated node acts as listener of a "secured" data stream, so it has to decrypt an incoming message, activating the decryption module. Moreover, this Graph node acts also as a producer of a "secured" stream and, then, it has to encrypt its processed streams with the encryption module before forwarding it, thus hiding the stream from other unauthorized listener Graph nodes. Referring to the previously described example, this is the case in which a Graph node, that is already a listener of the secured stream owned by company "C," wants to secure the stream generated by its processing. It is important to point out that each Graph node controls its generated flow with a visibility of only one step. This means that a listener of a "secured" flow can publish an "open" stream, and vice versa, thus producing "hybrid" path combinations, that across a flow from IoT source to final consumer produce a combination of "secured" and "open" steps. Referring to the example of company "C," which generates

Table 3.1: Comparison between Graph Framework actors and OAuth roles.

Graph Framework Actor	OAuth Role
Publisher Graph node, owner of the outgoing data stream	Resource Owner
Listener Graph node, willing to subscribe to interested topics	Consumer
Infrastructure routing element (Broker in a pub/sub paradigm)	Provider

a "secured" stream with data coming from its sensors, an IoT developer can decide to create a new Graph node listening from both the secured stream of company "C" and the stream of another company "D." The processing unit of the new Graph node, the developer can aggregate and transform input streams, generating new and different output streams, which can be published in an "open" mode, since the developer is the owner of this new produced stream. According to the inner organization of the IoT architecture, there could exist a parallelism between actors enrolled in Graph Framework and OAuth roles, as illustrated in Table 3.1. More precisely, OAuth roles could be detailed as follows.

- **Resource Owner:** The entity which owns the required resource and has to authorize an application to access it, according to authorization granted (e.g., read/write permission).
- **Consumer:** The entity that wants to access and use the required resource, operating in compliance with granted policies related to this resource.
- **Provider:** The entity that hosts the protected resource and verifies the identity of the Consumer that issues an access request to this resource.

As previously stated, each Graph node can apply cryptography to its streams, using encryption and decryption modules. The security mechanisms leave a few degrees of freedom to developers, who can adopt self-made solutions (e.g., using OAuth tokens) to secure streams, as well as rely on already secured protocols, thus adopting well-known and verified solutions. An overall view of the envisioned IoT architecture is shown in Figure 3.6, showing all component modules and their interactions.

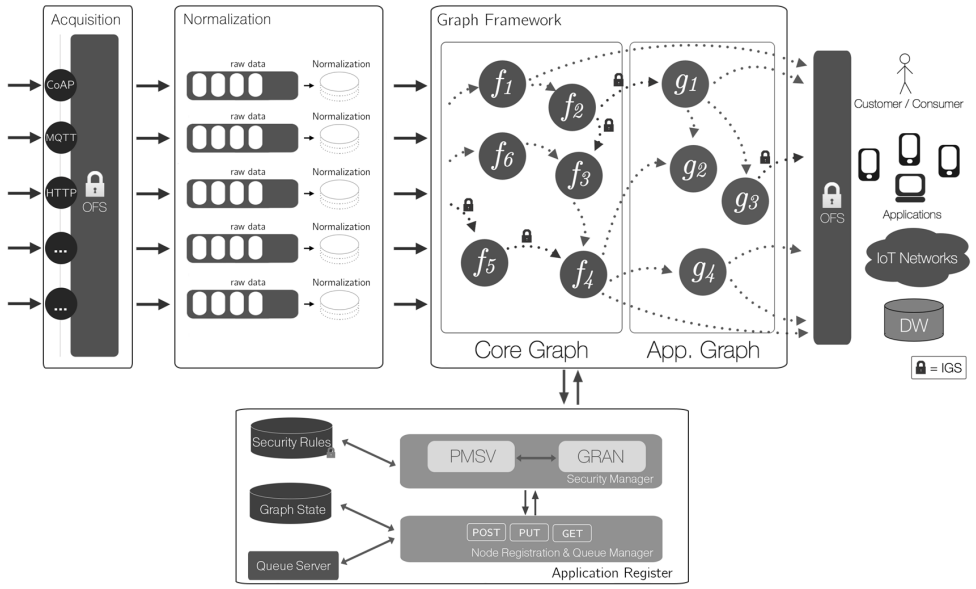


Figure 3.6: Complete IoT Cloud architecture, including proposed security modules and showing different interactions, from incoming stage to final consumer notification.

3.5 Evaluation of the Secured Architecture

3.5.1 Experimental Setup

In [33], a first implementation of the architecture, without security modules, has been presented, deploying it on a Virtual Machine and through the definition of a real use case, represented by a Smart Parking scenario. The dataset used for the evaluation contains more than 600k parking events, related to the spots status (free/busy), but lacking of geographical information (namely, geographic coordinates of the spots). Thus, in order to create a realistic scenario, parking spots have been located into 7 clusters.

In order to stress enough the proposed Big Stream platform, the evaluation has been conducted by varying the data generation rate in a proper range, forcing a specific frequency for incoming events, without taking into account real parking spots timestamps gathered from the dataset.

The evaluation involves a Java-based data generator, which: simulates events arrivals from IoT sensors networks; randomly selects an available protocol (HTTP, CoAP, MQTT); and periodically sends streams to the corresponding interface in the Acquisition Module. Once received, data are forwarded to the dedicated stage into the Normalization module, which enriches them with parking membership information, retrieving this association from an external SQL database, thus structuring the stream into a JSON schema compatible with the architecture. Once the Normalization module has completed its processing, it sends the structured data to the Graph Framework, that forwards the stream following paths based on routing keys, until final external listener is reached. The Graph considered in our experimental set-up is composed by 8 Core layers and 7 Application layers, within which different graph topologies (from 20 to 50 nodes) are built and evaluated.

3.5.2 Results

The proposed architecture has been evaluated by varying the incoming data stream generation rate between 10 msg/s and 100 msg/s. The first evaluation, which repre-

sents a benchmark for our performance analysis, has been made using the platform without security mechanisms. Then, security mechanisms have been introduced in the Graph Framework Module, in order to assess the impact of a security stage on the overall architecture.

The first performance evaluation has been conducted by measuring the delay (dimension: [ms]) between the instant at which normalized data are injected into the Graph Framework and the instant at which the message reaches the end of its routes, becoming available for external consumers/customers. In order to consider only the effective overhead introduced by the architecture and without taking into account implementation-specific contributions, performance results were obtained by subtracting the processing time of all Core and Application Nodes. Finally, these times have been normalized over the number of computational nodes, in order to obtain the per-node overhead introduced by the architecture, in a way that is independent of the implemented routing and topology configurations.

The second performance evaluation has been conducted adopting the same structure of the unsecured implementation, but introducing security mechanisms inside Graph nodes, through the adoption of symmetric encryption to encrypt/decrypt operations mentioned in the previous section. In order to guarantee a trade-off between security-level and reliability, Advanced Encryption Standard (AES) which is a symmetric cryptosystem, in its 256-bit key version [48], has been selected. AES is a block cipher based on a substitution and permutation (SP) combination, working on 128-bits blocks. The chosen key size specifies the number of repetitions of transformation rounds that convert the input, applying several processing stages and depending on the encryption key. The strength of AES256 is derived by its key space, with 1077 possible 256-bit keys, which affects the time needed to made a successful brute-force attack to a system implementing this cipher.

Moreover, in order to perform the second evaluation, a new version of the processing Core and Application nodes has been implemented, applying security at both input and output stages of the single Graph node, implementing the following behavior:

- If the processing node has received an AES 256-bit decryption key from the

Application Register, it then uses this key to decrypt incoming messages, returning a plaintext useful for processing operations;

- If an AES 256-bit encryption key was provided by the Application Register to the Graph node, it then encrypt the processed stream before forwarding it to its proper Exchange, using this symmetric key as encryption secret.

This security model is applicable also to the previously described example, in which the company "C" would like to secure its paths into the Graph Framework. The second evaluation has been made providing encryption and decryption keys to all Graph nodes, in order to secure all the intermediate routes followed by streams owned by that company. The results of the performance evaluations outlined above, carried out using different topologies obtained by varying the subset of deployed nodes, from 20 to 50, and the data generation rate R_{gen} , from 10 msg/s to 100 msg/s, are shown in Figure 3.7. The stream delay can be given using the following expression:

$$T_{processing_{freq}} = \frac{T_{out} - T_{in} - \sum_{k=1}^{N} GP_k}{N}$$

where: T_{out} is the instant (dimension: [ms]) at which parking data reach the last Application processing node; T_{in} indicates the instant (dimension: [ms]) in which normalized data comes to the first Core layer; and GP_k is the processing time (dimension: [ms]) of a Graph node $k \in \{1, 2, ...N\}$.

Moreover, in order to investigate the benefits and drawbacks of the adoption of other security solutions, different from symmetric encryption, an asymmetric-cryptography version of the Graph processing nodes has been implemented, adopting RSA [49] with a 512-bit key, which represents a private-public key cryptosystem. In Table 3.2, the performance results, in terms of stream delay, retrieved from a third evaluation scenario, obtained by replacing symmetric cryptosystem adoption with private/public RSA certificates provided to the Graph nodes by the Application Register module, are shown. The obtained results shown in Table 3.2 highlight how the adoption of an asymmetric cryptosystem represents a bad choice for the Graph inter-node security. Asymmetric solutions might be adopted outside of the Graph nodes, when an external node is willing to become an operating entity of the Graph Framework, challenging an authentication transaction with its signed certificate, that allows

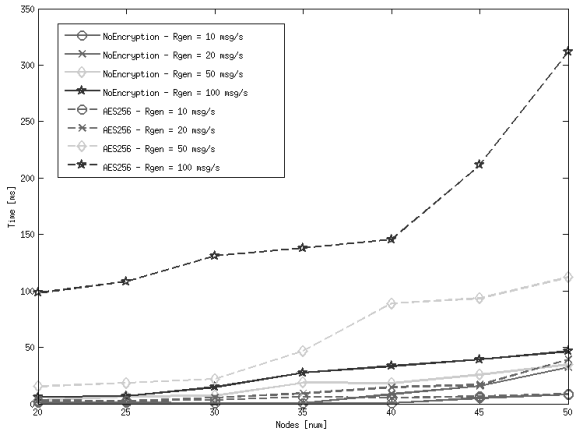


Figure 3.7: Average stream delay (dimension: [ms]) related to Graph Framework processing block, showing per-node time, in the case of unsecured communication as well as the case of adoption of symmetric encryption.

to verify its identity by the Application Register. Therefore, in the joining phase the asymmetric solutions could also be motivated by the evidence that time consumption is not the main constraint of this step.

In order to better highlight this final analysis of the evaluation results, in Figure 3.8 a logarithmic-scaled version of previously carried results is shown, evaluating the logarithm of the stream delay as a function of the number of nodes in the Graph, considering the following cases: (i) no encryption, (ii) symmetric encryption (AES256), and (iii) asymmetric encryption (RSA512). For comparison purpose, in all cases, two values of data generation rate R_{gen} are considered: 50 msg/s and 100 msg/sec.

The obtained results allow us to highlight that it seems possible to identify 3 main performance regions, in which the proposed Big Stream platform could work:

- **"Working" region**, tentatively identified around the "NoEncryption" curves, on which the system has the benchmark results, and where processing do not introduce heavy delays;

Table 3.2: Average stream delay related to the adoption of asymmetric encryption solution (RSA) into the Graph Framework processing block.

Number of Nodes	Stream Delay (R_{gen}= 50 msg/s)	Stream Delay (R_{gen}= 100 msg/s)
20	128.7453 ms	10890.7263 ms
25	156.909 ms	12962.2934 ms
30	2783.8599 ms	13841.4744 ms
35	10104.7272 ms	14048.8625 ms
40	11283.9916 ms	14515.0021 ms

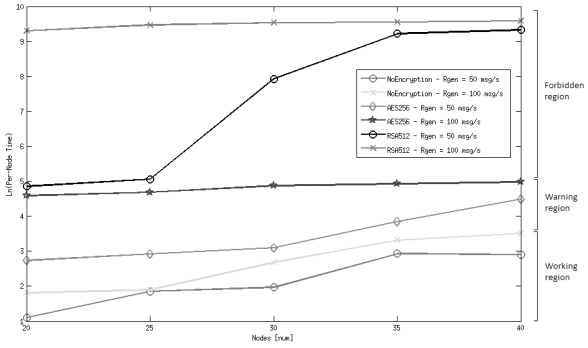


Figure 3.8: Logarithmic representation of the stream delay as a function of the number of nodes of the Graph, evaluated both without security mechanisms, as well as with cryptographic features.

- **"Warning" region**, limited around the "AES256" curves, in which delays introduced by security adoptions degrade performances in a little way, while maintaining good Quality of Service (QoS);
- **"Forbidden" region**, around the "RSA512" curves, in which the system incurs high delays, that may cause crashes and drop services, invalidating QoS and any Service Level Agreements (SLAs) signed with data stream producers and consumers.

Chapter 4

IoT Applications

4.1 The Web of Things Testbed

The Web of Things Testbed (WoTT) is a heterogeneous and innovative Web of Things (WoT) based testbed, described in [50], that enables developers to easily design and evaluate new services and applications in a real IoT environment and to effectively test human-object interaction mechanisms, which will play a fundamental role in broadening IoT use.

WoTT is particularly suited for this purpose because its architecture is completely based on standard protocols and network interfaces, without custom or proprietary solutions that would jeopardize interoperability among nodes. WoTT's main goals are:

- to hide low-level implementation details;
- to enhance network self-configuration by minimizing human intervention;
- to transparently and simultaneously manage multiple protocols and platforms;
- to provide a platform for the design and testing of human-object interaction patterns.

Table 4.1: Constrained Internet of Things (IoT) nodes in the Web of Things Testbed.

Constrained IoT nodes				
No.	**Node**	**Hardware**	**OS**	**Network Interface**
6	TelosB	MCU: TI MSP430F1611, RAM: 10 Kbytes, ROM: 48 Kbytes	Contiki	IEEE 802.15.4
20	Zolertia Z1	MCU: TI MSP430F2617, RAM: 8 Kbytes, ROM: 92 Kbytes	Contiki	IEEE 802.15.4
10	OpenMote	MCU: ARM Cortex-M3, RAM: 32 Kbytes, ROM: 512 Kbytes	Contiki	IEEE 802.15.4
35	SimpleLink SensorTag	MCU: ARM Cortex-M3, RAM: 20 Kbytes, ROM: 128 Kbytes	TI-RTOS	IEEE 802.15.4 / BLE

To effectively test new WoT-related applications, WoTT consists of several types of nodes that differ in terms of both computational capabilities and radio-access interfaces. Nonetheless, these nodes can be grouped into two main classes: constrained IoT (CIoT) nodes and single-board computer (SBC) nodes.

CIoT nodes are mainly based on the open source Contiki OS [51] and correspond to Class 1 devices – those that cannot easily talk to other Internet nodes that have a full protocol stack, but that can use a protocol stack designed for constrained nodes – as defined in the Internet Engineering Task Force (IETF) memo "Terminology for Constrained Node Networks" [52]. SBC nodes are more powerful, typically running a Linux OS and having multiple network interfaces. These nodes correspond to Class 2 devices – those that use the same protocols as notebooks or servers [52].

Regardless of what the actual nodes are, the standard communication protocols and mechanisms used in WoTT enable the testbed to manage node diversity seamlessly, making it possible to treat each node simply as an IP-addressable host. Table 4.1 shows the list of currently available CIoT nodes, while Table 4.2 contains the details about currently available SBC nodes in WoTT.

Table 4.2: Single-Board Computer (SBC) nodes in the Web of Things Testbed

SBC nodes				
No.	**Node**	**Hardware**	**OS**	**Network Interface**
20	Intel Galileo	CPU: SoC X Intel Quark X1000, RAM:256 Mbytes, Memory (SD): 8 Gbytes	[Linux] Debian	IEEE 802.3
5	Raspberry Pi B	CPU: Broadcom BCM2835 ARM11, RAM: 512Mbytes, Memory (SD): 8 Gbytes	[Linux] Raspbian	IEEE 802.3/802.11
5	Arduino Yún	Linux environment, CPU: Atheros AR9331,RAM: 64 Mbytes, ROM: 16 Mbytes	[Linux] OpenWRT	IEEE 802.3/802.11
		Arduino environment, MCU: ATmega32u4, RAM: 2.5 Kbytes, ROM: 32 Kbytes	Arduino	
4	UDOO	Linux environment, CPU: Freescale i.MX 6 ARM Cortex-A9, RAM: 1Gbyte, Memory (SD): 8 Gbytes	[Linux] UDOObuntu	IEEE 802.3/802.11
		Arduino-like environment, MCU: Atmel SAM3X8E ARM Cortex-M3, RAM: 100 Kbytes, ROM: 512 Kbytes	Arduino	

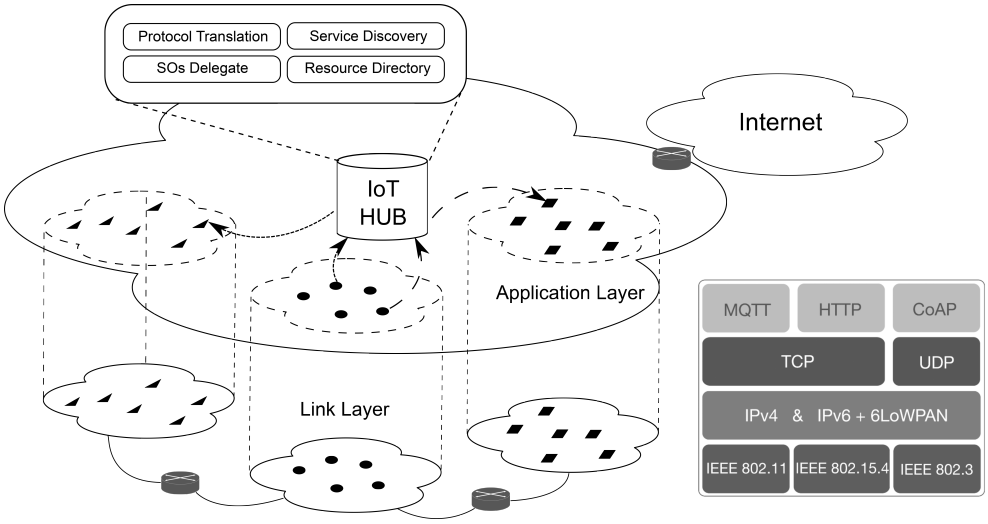


Figure 4.1: WoTT architecture and protocol stack. The central component is the IoT Hub, which interacts with the various layers and manages the testbed's heterogeneous network.

4.1.1 An IP-Based Infrastructure for Smart Objects

CIoT nodes are connected at the physical layer by IEEE 802.15.4 wireless links, whereas IPv6 is used at the network layer in combination with 6LoWPAN (IPv6 low-power wireless personal area network [53]) and RPL, the routing protocol for low-power and lossy networks, or LLNs [54].

Sensor-equipped CIoT nodes can act as CoAP servers or clients running Erbium, a lightweight CoAP implementation [55]. As with CIoT nodes, SBC nodes can act as CoAP clients or servers, with fewer implementation constraints. For example, on Arduino Yún nodes, a JavaScript application initializes a CoAP server through an instance of Node.js. Other SBC nodes, such as the Intel Galileo boards and Raspberry Pi, can support different types of languages ranging from Python to Java.

As Figure 4.1 shows, WoTT is heterogeneous by design to enable seamless communication among SOs and between the WoTT and external Internet elements such as the cloud, ISPs, and consumers.

The IP protocol adoption, in particular IPv6 or IPv6+6LoWPAN, is universally considered a key communication enabler for the future IoT. Thus, WoTT adopts IP as a common network substrate, thereby allowing simple integration into the existing Internet. Note that all WoTT nodes use standard protocols at all layers of the protocol stack; this includes several physical (PHY) and media access control (MAC) standards – for example, IEEE 802.11, IEEE 802.15.4, and IEEE 802.3 – as well as application-layer protocols.

The architecture also contains an innovative network element, the IoT Hub [56, 57], which operates at different layers of the protocol stack to further enhance interoperability among communicating devices by integrating several networks into a single IP-based substrate and implementing important functions at the application layer. Due to greater capabilities in computational power and networking, SBC nodes can effectively implement all IoT Hub functions.

WoTT's Wi-Fi networking infrastructure is based on Cisco Connected Mobile Experiences (CMX) access points and can be used to track devices for indoor localization purposes. In particular, the CMX platform provides Mobility Services Engine RESTful APIs, which allow developers to integrate service customization with location information into mobile applications, such as location-aware equipment tracking, guest access, and device-based services. This feature is currently used to build user location-ware IoT applications that can continuously monitor users, enabling specific and augmented interaction with the surrounding environment. Focusing on the application layer, WoTT currently supports CoAP, MQTT, and HTTP. HTTP is mainly used for communication between WoTT and external Internet actors or consumers, such as cloud storage services or IoT-unaware clients.

Among WoTT components, the IoT Hub is the key IoT enabler because it manages the different access technologies and supports full IP connectivity among all objects. The implementation of several functionalities at the application layer enables all the protocols listed in Figure 4.1 to coexist in the same environment. The software on smart objects has been developed with different programming languages, reinforcing the idea that – because of various features provided at the application layer, together with strict compliance to IoT standards – developers can create new IoT applications

easily and without additional constraints.

4.1.2 IoT Hub-enabled Smart Object Interactions

WoTT does not simply enable communications between IoT actors: it constitutes a "uniform" super-entity able to provide enhanced functionalities that go beyond the mere union of its components' features. To achieve this super-entity status, WoTT uses the various communication technologies in the IoT Hub to bridge and merge together several networks into a single IP network.

The IoT Hub also implements several functions at the application layer: it manages the services and resources available in the overall infrastructure, thereby playing a key role at the application layer. IoT Hub use is expedient for several reasons. The extreme heterogeneity of IoT devices requires mechanisms to support the management of and seamless interactions among SOs as well as humans. Moreover, because of SOs' limited data-collection capabilities could preclude them from handling large numbers of concurrent requests, limiting direct access to SOs is preferable. In other cases, extremely limited devices could act as clients to implement data-collection behavior.

Standardization efforts for IoT Hub design are in progress. A relevant example standard is HyperCat [58], which introduces standard specifications to allow servers to expose JSON-based hypermedia catalogues as collections of URLs. Thus, IoT clients can discover data available on servers using RESTful methods on HTTPS and JSON formats. Unlike the IoT Hub, however, HyperCat works at a higher level of abstraction because it is intended to allow IoT concept-based reasoning and service composition, and it does not take into account direct interactions with constrained devices in which specific protocols (such as CoAP) should be adopted to minimize energy and memory consumption.

From a networking standpoint, the IoT Hub is a Fog node [59] placed at the edge of multiple physical networks with the goal of creating an IP-based IoT network. The IoT Hub plays a fundamental role by implementing the following functions at the link and application layers of the protocol stack.

- Border Router: the IoT Hub bridges one or more networks (such as several IEEE 802.15.4 networks).
- Service and Resource Discovery: the IoT Hub is able to discover which SOs are available in the network and their hosted resources.
- Resource Directory (RD): the IoT Hub complies with the specifications provided in the IETF technical report "CoRE Resource Directory" [60] and maintains a list of all resources available in the bridged networks, thereby creating a centralized entry point for applications that need to perform resource lookup.
- Origin Server (OS): the IoT Hub provides a CoAP server that hosts the SOs' resources.
- CoAP-to-CoAP (C2C) proxy: the IoT Hub provides proxying capabilities for CoAP requests coming from external clients targeting constrained nodes.
- HTTP-to-CoAP (H2C) proxy: the IoT Hub provides HTTP-to-CoAP cross-proxying (protocol translation) to allow HTTP clients to access CoAP resources.
- Cache: the IoT Hub keeps a cache with the representation of most recently accessed resources to act as an SO delegate, thus minimizing latencies and unloading constrained devices.

Because the IoT Hub relies on standard interaction mechanisms and communication protocols, SOs do not depend on it for their operation. Instead, the IoT Hub mitigates the presence of nonstandard components that are interoperable with standard-compliant devices. The IoT Hub is not required for interaction and interoperability, but its presence extends the IoT network and increases its capabilities by simplifying and hiding complex and important tasks such as service discovery and routing.

4.1.3 Integration Challenges

The main challenges encountered in WoTT deployment have been design related: how to define different elements and their functionalities, represent different resources

and their relationships through suitable hypermedia, and maintain compatibility with standards.

The efforts devoted to WoTT design, the IoT Hub, and the use of standards have simplified the deployment process, making the integration of all different elements straightforward, notwithstanding their heterogeneity. In particular, WoTT hides critical implementation issues encountered at lower layers. For example, in a constrained network, such as IEEE 802.15.4, the resource advertisement feature is a critical point, requiring strict assumptions about energy and memory consumption on each constrained device. It is possible to tackle this issue by adopting an efficient multicast-based solution that can reduce the energy consumption [61]. WoTT's application-oriented end users are not concerned with these implementation details.

Another implementation challenge hidden from end users is configuring network elements such as routers and access points to create a unique IP-addressable network. Original firmware provided with common network elements typically does not allow such unusual configurations. To overcome this limitation, WoTT network components have been flashed with other more customizable and advanced firmware, like Tomato.

4.1.4 Building WoT applications

To validate WoTT's benefits and demonstrate the ease of integrating a newly deployed application within the testbed, an application for wearable and mobile-oriented applications have been implemented. Thanks to their portability, wearable and mobile devices are obvious solutions for tracking people's activities. To accomplish this, we implemented indoor localization features into WoTT using CMX access points to triangulate the locations of people and objects. The availability of localization APIs in the CMX system enables us to create applications that can follow users throughout an IoT environment and possibly even anticipate their movements.

Together with access-point based localization, the use of on-board inertial measurement units can further improve the tracking experience. Mobile devices play an important role in WoTT's architecture. Aside from interacting with SOs, they can also act as SOs, providing data generated by their on-board sensors. Mobile devices

Figure 4.2: A WoTT-based application for mobile and wearable devices. Resources and interactions are revealed gradually, according to the REST paradigm, so that the application can adapt itself dynamically.

can thus be considered as WoTT nodes, making WoTT highly dynamic.

As a uniform, application-oriented platform, WoTT can be used by developers to easily create and test real-world IoT applications in a short period of time, thus making it more attractive than other currently available platforms. This is due to ready-to-use capabilities and the direct/active interactions that a deployed application can have with the resources available in WoTT. From an operational point of view, developers simply run their applications on testbed resources without needing to add virtualized environments or services; thus the application becomes part of a WoT scenario in which consumers are not only "readers" but active participants.

Based on the WoTT's capabilities, Figure 4.2 shows some possible developed application. This application has been tested on the Android Wear platform using LG G Watches and Android 5.0.1 smartphones. In the near future, as more SOs are deployed, vendor-provided apps are less likely to be the usual means by which we in-

teract with things, and a more standard approach will be required to do so effectively.

The application performs the following steps. First, the mobile device discovers nearby SOs proactively and reactively, by means of standard service discovery and resource directory mechanisms. Then, it forwards the collected information to its connected wearable device interface. Through wearable interfaces, a user can see and browse a list of all the resources that have been discovered and select one to interact with. Interactions are thereby performed according to the function set specified by the selected SO, for example, a light bulb might provide an on/off switch, or a temperature sensor might provide a way to read its value. Resources and interactions are revealed gradually, according to the REST paradigm, so that the application can adapt itself dynamically.

WoTT resources can be deployed on different platforms, such as different SOs, and by using heterogeneous protocols. However, this is completely transparent to a developer who is able to access all these resources; the only constraint is to use standard protocols. This is possible through the IoT Hub's abstraction ability, it is not provided by other existing testbeds, such as SmartSantander [23].

The SOs' own sensors and actuators, users' devices, and other SOs can each act as clients and interact with one another through the following approaches.

- Polling allows clients to retrieve the value associated with the queried resource by performing a CoAP GET request.
- Observing can be used by clients to receive asynchronous updates when the value of the specified resource changes, which is a more efficient mechanism because it avoids periodic data polling.
- Acting is used by clients to set up the value of a specified resource, such as activating an actuator, depending on the function set provided by the resource.

The observing approach, which has not been defined in HTTP, is a lightweight CoAP-oriented interaction mechanism [62]. SO resource observation is achieved by performing a particular CoAP GET request, which contains an "Observe" option. This option instructs the target SO to add a new subscriber that will receive subse-

quent resource updates in push mode. Subscribers can also stop observing a resource at any time and unsubscribe from updates.

The use of standard communication protocols and network interfaces, well-known Web-based design approaches, and widely varied hardware platforms are changing the IoT and presenting new opportunities to developers, businesses, and users. In this dynamic and evolving scenario, the availability of real and accessible resource-oriented testbeds that allow active and direct interaction among users and devices with low-level nodes and services are a key enabler for widespread future IoT adoption and, indeed, are driving the transition from the IoT to the WoT.

WoTT exemplifies a novel architectural and networking approach to important IoT challenges. Designing and implementing the testbed has confirmed the need for open evaluation platforms to explore and integrate innovative IoT applications and to bridge the gap between users and things. WoTT's hardware/software heterogeneity confirms that proper use of standard protocols such as HTTP, CoAP, and MQTT and of interaction paradigms such as REST and service/resource discovery are fundamental to enabling transparent and dynamic interactions among multiple SOs and personal mobile and wearable devices. Moreover, this heterogeneity can be extended by improving the IoT Hub features, for example, by adding support to Bluetooth devices. This would open WoTT to an emerging category of IoT-enabled devices, namely those using Bluetooth low-energy (BLE), and BLE service discovery mechanisms such as UriBeacon.

Nevertheless, important issues must be addressed to make the IoT part of daily life. Improved security, greater device and network interoperability, faster data processing, and easier development and deployment will be core focus areas for academic and industrial Rf&D during the next few years.

Testbeds like WoTT are the perfect experimental playgrounds to boost and support IoT application development by creating a common space that designers, hardware manufacturers, and companies can exploit to make the IoT accessible and easy to use for everyone, just as the Internet is right now.

4.2 The Dynamic User Interface Paradigm

In [63], a UI-generation approach, which allows client mobile applications to dynamically render UIs and perform interactions driven by surrounding smart objects, is proposed. More in detail, the system architecture is composed by an IoT network of smart objects hosting one or more *resources*, such as sensors data, actuators, or UI descriptors. In particular, the latter are suited for the communication between smart objects and humans. A mobile application shows discovered resources and, according to user's selection, automatically renders the correct UIs on the base of UI descriptors.

Smart objects operate in a LLN, typical of IoT scenarios, where the low capabilities of smart objects, in terms of computational power and battery life, requires the usage of low-power radio protocols, such as IEEE 802.15.4, and unreliable transport protocols, such as UDP, in order to minimize energy consumption. In such LLN scenario, nodes are not able to transfer rich interfaces like those that standard web servers might provide (e.g., HTML/CSS/JS pages). In fact, delivering rich interfaces, implies large amount of data traffic which results in delays and undesired energy consumption mainly due to fragmentation of data packets and packet loss. However, even in such constrained scenario, the limitations of these nodes should not impact the experience of those who are willing to consume the resources they are providing. In order to minimize this problem, in the considered system, constrained nodes send lightweight UI descriptors (better described in the next subsection) which can be efficiently transmitted and used by the mobile application in order to generate an appropriate and rich user interface and to pro-actively interact with the smart objects.

The complete procedure for interacting with smart objects is shown in Figure 4.3: Smart objects provide resources through CoAP servers; available UI descriptors resources are first discovered by the mobile application through suitable service and resource discovery mechanisms, such as the ZeroConf [64] protocol suite. Alternatively, the representation of a resource might be a hypermedia that contains link(s) to suitable UI Descriptor resources to be used for subsequent interactions.

Once discovered, the mobile application displays the list of available resources.

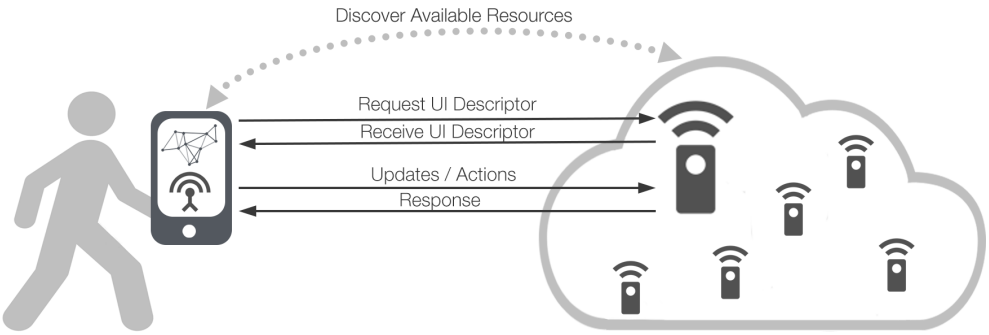


Figure 4.3: Efficient User Interface generation in an IoT scenario with multiple Smart objects.

After the user selects a particular resource, the application sends a request to the corresponding CoAP server to retrieve the UI descriptor needed to generate the appropriate UI. It is important to note that, in the implementation proposed in [63], the UI Descriptor is a resource itself and can be accessed in the same way as other resources. When the UI descriptor is received, the application adapts its UI accordingly. In this way, the smart object, which hosts the resource, can publish the correct UI descriptor to guide the client to properly interact with it. This approach makes it very efficient and lightweight to let constrained devices drive the interplay with client applications.

4.2.1 User Interface Descriptor (UID)

In order to represent a complete UI, a simple but expressive data structure, denoted as "User Interface Descriptor" (UID), has been defined.

The UID does not simply describe which graphical elements should be presented on the UI, but it also allows to handle actions and requests, according to REST applications principles. Each UID is characterized by:

- a *name*;
- an optional *description*;

- a list of UISections.

UISections represent self-consistent modules of the UI and can be populated by UIElement objects.

A UIElement is a generic UI component characterized by; i) a unique *name*; ii) a textual *description*; and iii) a *type* property. Starting from this generic element, the following types of graphical UIElement are available.

- UIInputField: to collect input from users. This component is characterized by different specific properties as: i) *mandatory* – if the input field is required or not; ii) *dataDescription* – a text label to inform the user about what kind of information must be entered in the input field; and iii) *dataType* – to specify the data type (i.e Text, Numeric, or Boolean).

- UILabel: non-editable text that should be displayed in the UI.

- UIButton: interaction point with the user. This component is characterized by the following properties: i) *text*: containing the text that is shown to the user on the button component; and ii) *action*: the behavior triggered when the user clicks on the button.

- UIImage: an image element in the UI. The *url* property defines the source to retrieve the image. Other properties are *width* and *height*.

- UIMap: a map in the UI containing one or more markers. The main properties are: i) *type* to define the kind of visualization (i.e. Satellite, Hybrid or Standard); ii) the initial zoom level (*startingZoom*); and iii) the initial center point (*centerLat* and *centerLon*). The *markerList* property contains a list of *UIMarkers* that should be shown inside the map.

- UIMarker: it represents a geographical point on a *UIMap* element. Each marker is characterized by *latitude*, *longitude*, and *markerImage* properties. An *action* element can be used to describe the UI behavior when a user clicks on a specific marker.

Following the Hypermedia as the Engine of Application State (HATEOAS) paradigm [4], in order to make the UIs flexible and interactive, the *action* and *datasource* elements have been defined. The *action* component is related to a *UIElement* which can trigger events (e.g. *UIButton* or *UIMarker*) which are mapped into corresponding REST methods (GET, POST, PUT, DELETE). The properties of the *action* component are:

- *protocol*, *port*, *host*, *path*, *query*, and *fragment* to construct the complete resource URI;
- *method* containing the REST method to be used;
- *payload* containing additional data to be included in the REST request;
- *dataFormat* containing data format of the payload.

The concept of *datasource* specifies a (remote or local) source of information enabling updates in a dynamically generated UI. Specifically, each generic UIElement can be enriched with a *datasource* component, which describes the URI of a resource to be used for updates, by means of the following properties:

- *protocol*, *port*, *host*, *path*, *query*, and *fragment* to construct the complete URI;
- *dataFormat* contains the data format of received updates, using Internet media types;
- *id* uniquely identifies the UIElement to refresh;
- *payload* contains updated information;
- *type* defines whether the data source is remote or local;
- *refreshInterval* contains a suggested interval value, in milliseconds, between successive updates;
- *observable* is a boolean property to define whether the resource related with the UIElement is observable.

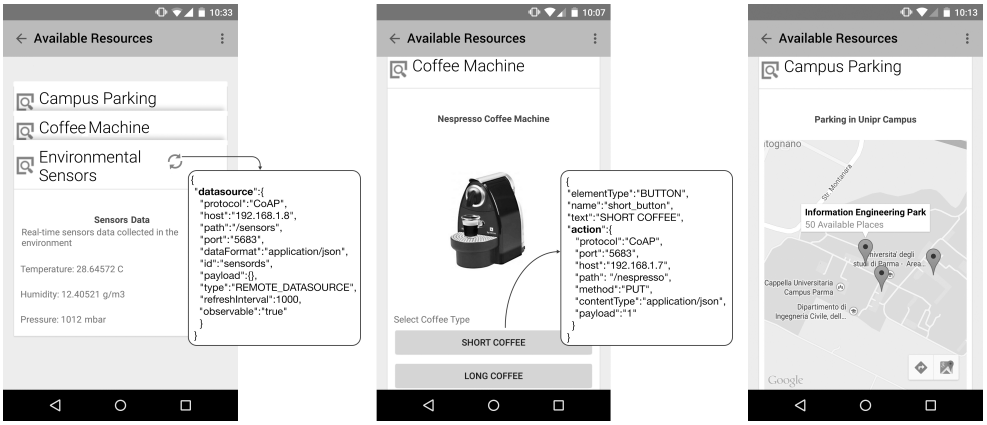
A resource denoted as "observable" is a resource that can change its status over time (e.g., a temperature sensor) and sends updates about its changes to subscribed clients (denoted as "observers"), thus implementing the Observer design pattern. As users are generally interested in updated values, from an implementation point of view, an observable resource supports the publish/subscribe paradigm, which is more efficient than the request/response model with periodic polling for data. Periodic polling is, however, supported.

It is important to note that in the proposed paradigm, it has been designed a new hypermedia specific for the UID, rather than reusing existing ones (e.g., ATOM [65]) because, typically, they are not designed for representing UI elements and IoT interactions between smart objects. UID resources can be formatted using different data-formats (e.g., XML, JSON) thus generating different Internet media types (e.g., `application/uid+json` and `application/uid+xml`). In [63], the JSON data format has been selected because it is widely supported and easy to generate and parse. Moreover, it is more lightweight than typical markup languages (e.g., HTML and XML) and this is especially important for IoT nodes operating in LLNs, which should avoid transmitting large amounts of data, in order to minimize retransmissions that might occur using low-power and unreliable communication protocols. Although these aspects have lead to the choice of JSON, nothing prevents to represent the UID with other formats (e.g., `uid+xml`, `uid+html`).

4.2.2 Android Implementation

After the definition of the components and properties of an UID, it has been implemented *VoiláLib*, an Android library project which contains the engine for parsing uid+json and dynamically building all elements to be presented to the user. Each component of a UID has a corresponding Java class responsible for rendering the corresponding Android graphical view. The library can be easily integrated in all Android applications which require interaction with an IoT network providing self-modifying UI functionalities.

Typically, RESTful IoT applications require the use of protocols specifically designed for smart objects; for this reason the VoiláLib implementation supports both



(a) (b) (c)

Figure 4.4: Representative IoT use cases: (a) Environmental Monitoring; (b) Coffee Machine; (c) Smart Parking.

HTTP and CoAP, used in requests related to *datasources* and *actions* UID components.

More in detail, when a UI element is enriched with datasource information, the VoiláLib engine is required to send one or more GET requests to the specified endpoint to retrieve updates. Referring to a CoAP endpoint, two kinds of interactions are possible. If the resource is marked as *not observable*, the engine performs periodic GET requests to the smart object, according with the provided *refreshInterval* (polling approach). If, instead, the resource is marked as *observable*, according with specifications described in [62], the engine, with a single GET request, notifies the CoAP endpoint its interest in receiving updates (push approach). For protocols which, like HTTP, do not contemplate the Observe option the polling approach is preferred: push-based updates are feasible only with endpoint providing a Push Notification Service (PNS). In order to send CoAP requests, VoiláLib uses mjCoAP [38], a Java-based CoAP library selected because of its simplicity and lightness, which are particularly suited to mobile devices.

To test the functionalities provided by the library, has been created the Voilá An-

droid application, which can be used to easily interact with smart objects deployed in an IoT network. The operational steps of the application are the following.

First, a service discovery procedure is performed, in order to present a list of available resources to the user in a CardView layout. When the user selects a resource in the list, Voilá transparently requests the uid+json document to the corresponding smart object, and shows the proper UI to the user. The Voilá application is compliant with the Android Material Design guidelines and uses jmDNS, an open source Java implementation of ZeroConf, for the service discovery module.

4.2.3 Experimental Analysis and Testbed Integration

The experimental analysis and testing of the system has been conducted in the Web of Things Testbed (WoTT) [50]. The WoTT has been deployed at the Wireless Ad-hoc and Sensor Network Sensor Networks (WASN) Laboratory of the Department of Information Engineering of the University of Parma in order to create an innovative IoT experimental environment.

In order to install the CoAP servers on the IoT and SBC nodes in WoTT have used the Erbium (Er) [55] and Californium (Cf) [66] libraries respectively. In particular, Erbium is the official low-power REST engine for the Contiki OS and it obviously includes a comprehensive embedded CoAP implementation. Californium is a powerful CoAP framework written in Java which targets stronger IoT devices.

In order provide representative and useful IoT applicative examples, three main use-cases have been considered: (i) Environment Monitoring, (ii) Coffee Machine, and (iii) Smart Parking. The UIs for each use-case are depicted in Figure 4.4 and have been generated with a LG Nexus 5 (Android 5.0.1).

In the first use-case, the user discovers the “Environmental Sensors” resources and taps the corresponding card. This operation causes the request for a uid+json of this resource and the corresponding UI generation. As shown in Figure 4.4 (a), the UI contains three different UILabels reporting the current values for temperature, humidity and pressure. In this case, the uid+json resource includes also a *datasource* element, thus the UI contains a “refresh” button which can be pressed by the user in order to request updates for data.

The second use-case enables the user to actively interact with the IoT system, performing some actions such as making a coffee with a connected coffee machine. The uid+json contains two buttons with actions related to the kind of coffee that the machine can dispense. Figure 4.4 (b) shows the resulting UI and, in particular, the *action* component related to one of the UIButtons is shown.

The last use-case is related to a smart parking application used by people in the campus of the University of Parma to monitor the status of parking spots through their smartphones. The UID document of this resource is composed by a UIMap component, with a list of markers indicating the parking location. By tapping on a marker, a pop-up shows the current number of available parking spots. The rendering of the "Campus Parking" resource is shown in Figure 4.4 (c).

Finally, a performance evaluation about the required time to load and render the entire UI with the proposed approach has been considered. With this aim, the developed Android application has been installed on three devices, with different hardware profiles and releases of the operating system:

- Samsung Galaxy S (Android 2.3.3);
- Samsung Galaxy S3 (Android 4.1.2);
- Samsung Galaxy Nexus (Android 4.2.1).

The considered devices and their corresponding operating systems have been selected in order to consider a worst case scenario, without using high-end, cutting edge devices.

In order to effectively test the performance of the dynamic UI generation, six uid+json have been configured with different numbers of UIElements (UILabel, UIButton, UIInputField and UIImage) and each shown value has been averaged on ten runs for each device. The graph in Figure 4.5 shows the total loading time in relation to the total number of UI elements in the descriptor. It can be noted that the total required time is comparable to that of standard mobile applications, also for descriptors with a high number of UI components. The value of 200 has been chosen as a worst-case upper bound of the number of UI elements concurrently displayed by a single user

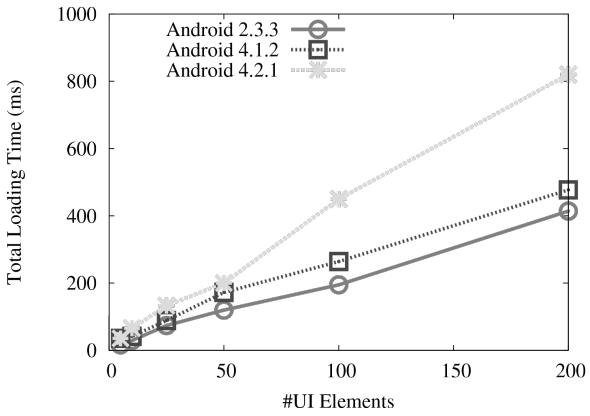


Figure 4.5: Required time (dimension: [ms]) to load and render the entire dynamic UI as a function of the number of UI elements.

interface. In fact, typical mobile application UIs consist of much fewer elements because of screen size and user experience guidelines.

These results are encouraging and demonstrate that, in fact, any UI can be rendered in reasonable times and with a linearly increasing (with the number of UI elements) memory allocation. Additional evaluation will be performed by means of a user survey targeting both end users and service providers, in order to retrieve feedback regarding the usability and the simplicity of developing services for pervasive scenarios.

Conslusions

This thesis has been focused on efficient data management in the IoT scenario, with particular attention to mechanisms and applications which can favor the widespread of IoT technologies and their adoption by common people.

Several reference IoT scenarios (i.e., alerting, industrial automation, transportation, networks of sensors and actuators) have real-time or predictable latency requirements, and deals with billions of heterogeneous connected devices collecting and sending an enormous quantity of data. These particular features create a new need for architectures specifically designed to handle this kind of scenario and to guarantee minimal processing latency.

First, the selected scenario requirements and the new paradigm, denoted as "Big Stream Paradigm," have been described, highlighting the main differences with the well known "Big Data Paradigm," which is not the correct solution in the IoT scenario. The designed listener-based architecture and its components have been detailed: Acquisition Module, Normalization Module, Graph Framework, and Application Register. The implementation of the overall system and its evaluation on a real-world Smart Parking dataset has been presented.

The listener-oriented approach has several benefits, such as: i) decreased latency: the push-based approach guarantees that no delays due to polling and batch processing are introduced; ii) fine-grained self-configuration: listeners can dynamically "plug" to those that output data of interest; iii) optimal resource allocation: processing units that have no listeners can be switched off, while those with many listeners can be replicated.

Security issues have also been addressed. The implementation of each module of the platform has been detailed taking particularly into account security enhancements. The OAuth protocol, adapted to the specific needs of each module, complies with proposed publish/subscribe platform and is expedient to carry out all security tasks.

Later, the availability of real and accessible resource-oriented testbeds that allow active and direct interaction among users and devices has been analyzed driving the transition from the IoT to the WoT. More in detail, the WoTT testbed has been described, exemplifying a novel architectural and networking approach to important IoT challenges. Designing and implementing the testbed has confirmed the need for open evaluation platforms to explore and integrate innovative IoT applications and to bridge the gap between users and things.

Finally, an innovative smart object-driven UI generation pattern for mobile applications in heterogeneous IoT networks has been presented. This paradigm aims at simplifying interactions between users and smart objects. The benefits introduced by the proposed approach are many. In particular, end-users can interact with smart objects without any a-priori knowledge — for instance, without the need to use of custom mobile vendor-provided apps. The proposed approach makes use of a lightweight UI descriptor, denoted as UID, which includes details on both UI components and "interaction" components, such as actions and datasources. An evaluation of the the cost of the generation of dynamic UIs on Android platforms has also been performed and shows how the approach is absolutely feasible even with a large number of graphical components.

List of Publications

International Journal

- L. Belli, S. Cirani, L. Davoli, G. Ferrari, L. Melegari, and M. Picone, "Applying security to a big stream cloud architecture for the Internet of Things," *International Journal of Distributed Systems and Technologies* (IJDST), Special Issue on *Advances in Cloud for Smart Cities*, vol. 7, no. 1, January 2016, pp. 7-58. DOI: 10.4018/IJDST.2016010103

- L. Belli, S. Cirani, L. Davoli, A. Gorrieri, M. Mancin, M. Picone, and G. Ferrari, "Design and deployment of an IoT application-oriented testbed", *IEEE Computer*, Special Issue *Activating the Internet of things*, vol. 48, no. 8, September 2015, pp. 32-40. DOI:10.1109/MC.2015.253.

- L. Belli, S. Cirani, L. Davoli, G. Ferrari, L. Melegari, M. Montón, M. Picone, "A scalable big stream cloud architecture for the Internet of Things," *International Journal of Systems and Service-Oriented Engineering* (IJSSOE), 5(4), pp. 26-53, January 2015.

International Conference

- L. Belli, S. Cirani, A. Gorrieri, M. Picone "A novel smart object-driven UI generation approach for mobile devices in the Internet of Things", *Proc. 1st International Workshop on Experiences with the Design and Implementation of Smart Objects* (SmartObjects'15), Paris, France, September 2015, pp. 1-6. DOI: 10.1145/2797044.2797046

- L. Belli "Big stream cloud architecture for the Internet of Things", in *Proc. 2015 MobiSys PhD Forum*, Florence, Italy, May 2015, pp. 1-2. DOI: 10.1145/2752746.2752789

Book Chapters

- L. Belli, S. Cirani, L. Davoli, L. Melegari, M. Montón, M. Picone, "An Open-Source Cloud Architecture for Big Stream IoT Applications," chapter in *Interoperability and Open-Source Solutions for the Internet of Thing*, edited by I. Podnar Zarko, K. Pripuzic, M Serrano, pp.73-88, Springer: 2015. Proc. of IEEE 22st International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Split (Croatia), September 2014. DOI: 10.1007/978-3-319-16546-2_7.

- L. Belli, S. Cirani, G. Ferrari, L. Melegari, M. Picone, "A Graph-based Cloud Architecture for Big Stream Real-time Applications in the Internet of Things," in *Advances in Service-Oriented and Cloud Computing*, edited by G. Ortiz and C. Tran, Springer, Germany: 2015. Proc. of Workshops of ESOCC 2014 (2nd International Workshop on CLoud for IoT, CLIoT 2014), Manchester, UK, September 2014, Revised Selected Papers Series: Communications in Computer and Information Science, Vol. 508. ISBN: ISBN 978-3-319-14885-4

Bibliography

[1] J. Postel. Internet protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864. URL: `http://www.ietf.org/rfc/rfc791.txt`.

[2] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252 (Proposed Standard), June 2014. URL: `http://www.ietf.org/rfc/rfc7252.txt`.

[3] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[4] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010.

[5] European lighthouse integrated project - 7th framework. Internet of Things - Architecture (IoT - A). URL `http://www.iot-a.eu/`, 2012. URL: `http://www.iot-a.eu/`.

[6] Mirko Presser, Payam M Barnaghi, Markus Eurich, and Claudia Villalonga. The sensei project: integrating the physical world with the digital world of the network of the future. *Communications Magazine, IEEE*, 47(4):1–4, 2009.

[7] Paolo Medagliani, Jérémie Leguay, Andrzej Duda, Franck Rousseau, Marc Domingo, Mischa Dohler, Ignasi Vilajosana, and Olivier Dupont. Bringing IP to Low-power Smart Objects: the Smart Parking Case in the CALIPSO Project. 2014.

[8] S Deering and R Hinden. Rfc 2460-internet protocol, version 6 (ipv6) specification, 1998, 2006.

[9] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0. RFC 1945 (Informational), May 1996. URL: `http://www.ietf.org/rfc/rfc1945.txt`.

[10] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. RFC 3920 (Proposed Standard), October 2004. Obsoleted by RFC 6120, updated by RFC 6122. URL: `http://www.ietf.org/rfc/rfc3920.txt`.

[11] Dave Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html*, 2010.

[12] Simone Cirani, Marco Picone, and Luca Veltri. Cosip: a constrained session initiation protocol for the internet of things. In *Advances in Service-Oriented and Cloud Computing*, pages 13–24. Springer, 2013.

[13] Simone Cirani, Luca Davoli, Marco Picone, and Luca Veltri. Performance evaluation of a sip-based constrained peer-to-peer overlay. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 432–435. IEEE, 2014.

[14] Andrew Edney and Rui Maximo. Session initiation protocol. *Pro LCS: Live Communications Server Administration*, pages 15–25, 2007.

[15] Laura Belli, Simone Cirani, Gianluigi Ferrari, Lorenzo Melegari, and Marco Picone. A graph-based cloud architecture for big stream real-time applications in the internet of things. In *Advances in Service-Oriented and Cloud Computing*, pages 91–105. Springer, 2014.

[16] David Mera, Michal Batko, and Pavel Zezula. Towards fast multimedia feature extraction: Hadoop or storm. In *Multimedia (ISM), 2014 IEEE International Symposium on*, pages 106–109. IEEE, 2014.

[17] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[18] Stephan Reiff Marganiec, Marcel Tilly, and Helge Janicke. Low-latency service data aggregation using policy obligations. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 526–533. IEEE, 2014.

[19] Marcel Tilly and Stephan Reiff-Marganiec. Matching customer requests to service offerings in real-time. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 456–461. ACM, 2011.

[20] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting smart objects with ip: The next internet*. Morgan Kaufmann, 2010.

[21] Roy Want, Bill N Schilit, and Scott Jenson. Enabling the internet of things. *Computer*, (1):28–35, 2015.

[22] Georgios Z Papadopoulos, Julien Beaudaux, Antoine Gallais, Thomas Noël, and Guillaume Schreiner. Adding value to wsn simulation using the iot-lab experimental platform. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pages 485–490. IEEE, 2013.

[23] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, et al. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 61:217–238, 2014.

[24] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):18, 2009.

[25] J. Nichols and B. A. Myers. Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):17, 2009.

[26] J. Nichols, B. A Myers, and K. Litwack. Improving automatic interface generation with smart templates. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 286–288. ACM, 2004.

[27] S. Mayer, A. Tschofen, A. K. Dey, and F. Mattern. User interfaces for smart things a generative approach with semantic interaction descriptions. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(2):12, 2014.

[28] Laura Belli, Simone Cirani, Andrea Gorrieri, and Marco Picone. A novel smart object-driven ui generation approach for mobile devices in the internet of things. In *Proceedings of the 1st International Workshop on Experiences with the Design and Implementation of Smart Objects*, pages 1–6. ACM, 2015.

[29] David Upton. *CodeIgniter for Rapid PHP Application Development*. Packt Publishing Ltd, 2007.

[30] Paul Du Bois. *MySQL*. Pearson Italia Spa, 2004.

[31] Maria L Langer and Corbin Collins. *Database publis hing with FileMaker pro on the Web*. Peachpit Press, 1998.

[32] Agnar Aamodt and Mads Nygård. Different roles and mutual dependencies of data, information, and knowledgeâĂŤan ai perspective on their integration. *Data & Knowledge Engineering*, 16(3):191–222, 1995.

[33] Laura Belli, Simone Cirani, Luca Davoli, Lorenzo Melegari, Màrius Mónton, and Marco Picone. An open-source cloud architecture for big stream iot applications. In *Interoperability and Open-Source Solutions for the Internet of Things*, pages 73–88. Springer, 2015.

[34] RabbitMQ. URL `http://www.rabbitmq.com/`.

[35] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, November 2006. URL: http://dx.doi.org/10.1109/MIC.2006.116, doi:10.1109/MIC.2006.116.

[36] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[37] Roy T Fielding and Gail Kaiser. The apache http server project. *Internet Computing, IEEE*, 1(4):88–90, 1997.

[38] Simone Cirani, Marco Picone, and Luca Veltri. mjcoap: an open-source lightweight java coap library for internet of things applications. In *Interoperability and Open-Source Solutions for the Internet of Things*, pages 118–133. Springer, 2015.

[39] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in action*. Manning, 2011.

[40] Mosquitto. An Open Source MQTT Broker. URL http://mosquitto.org/.

[41] Worldsensing. URL http://www.worldsensing.com/.

[42] Ignasi Vilajosana, Jordi Llosa, Brais Martinez, Marc Domingo-Prieto, Albert Angles, and Xavier Vilajosana. Bootstrapping smart cities through a self-sustainable model based on big data flows. *Communications Magazine, IEEE*, 51(6):128–134, 2013.

[43] Laura Belli, Simone Cirani, Luca Davoli, Gianluigi Ferrari, Lorenzo Melegari, and Marco Picone. Applying security to a big stream cloud architecture for the internet of things. *International Journal of Distributed Systems and Technologies (IJDST)*, 7(1):37–58, 2016. doi:10.4018/IJDST.2016010103.

[44] D. Hardt. The oauth 2.0 authorization framework. RFC 6749 (Proposed Standard), October 2012. URL: http://www.ietf.org/rfc/rfc6749.txt.

[45] Paul Fremantle, Benjamin Aziz, Jacek Kopecky, and Philip Scott. Federated identity and access management for the internet of things. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 10–17. IEEE, 2014.

[46] Simone Cirani, Marco Picone, Pietro Gonizzi, Luca Veltri, and Giorgio Ferrari. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. *Sensors Journal, IEEE*, 15(2):1224–1234, 2015.

[47] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[48] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[49] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[50] Laura Belli, Simone Cirani, Luca Davoli, Andrea Gorrieri, Mirko Mancin, Marco Picone, and Gianluigi Ferrari. Design and deployment of an iot application-oriented testbed. *Computer*, (9):32–40, 2015.

[51] Adam Dunkels, Oliver Schmidt, Niclas Finne, Joakim Eriksson, Fredrik Österlind, Nicolas Tsiftes, and Mathilde Durvy. The contiki os: The operating system for the internet of things, 2011.

[52] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228 (Informational), May 2014. URL: `http://www.ietf.org/rfc/rfc7228.txt`.

[53] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82. ACM, 2007.

[54] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, and R. Alexander. Rpl: Ipv6 routing protocol for low-power and

lossy networks. RFC 6550 (Proposed Standard), March 2012. URL: `http://www.ietf.org/rfc/rfc6550.txt`.

[55] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. Erbium (er) rest engine and coap implementation for contiki.

[56] Simone Cirani, Gianluigi Ferrari, Nicola Iotti, and Marco Picone. The iot hub: a fog node for seamless management of heterogeneous connected smart objects. In *Sensing, Communication, and Networking-Workshops (SECON Workshops), 2015 12th Annual IEEE International Conference on*, pages 1–6. IEEE, 2015.

[57] Simone Cirani, Luca Davoli, Giorgio Ferrari, Rémy Léone, Paolo Medagliani, Marco Picone, and Luca Veltri. A scalable and self-configuring architecture for service discovery in the internet of things. *Internet of Things Journal, IEEE*, 1(5):508–521, 2014.

[58] Rodger Lea. Hypercat: an iot interoperability specification. 2013.

[59] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[60] Zach Shelby and Carsten Bormann. Core resource directory. 2014.

[61] Mattia Antonini, Simone Cirani, Gianluigi Ferrari, Paolo Medagliani, Marco Picone, and Luca Veltri. Lightweight multicast forwarding for service discovery in low-power iot networks. In *Software, Telecommunications and Computer Networks (SoftCOM), 2014 22nd International Conference on*, pages 133–138. IEEE, 2014.

[62] K. Hartke. Observing Resources in CoAP. Technical report, IETF Internet Draft draft-ietf-core-observe-16, December 2014. URL: `https://tools.ietf.org/html/draft-ietf-core-observe-16`.

[63] Laura Belli, Simone Cirani, Andrea Gorrieri, and Marco Picone. A novel smart object-driven ui generation approach for mobile devices in the internet of things.

In *Proceedings of the 1st International Workshop on Experiences with the Design and Implementation of Smart Objects*, pages 1–6. ACM, 2015.

[64] Stuart Cheshire. Zero configuration networking (zeroconf). 2004. URL: `http://www.zeroconf.org`.

[65] M. Nottingham and R. Sayre. The atom syndication format. RFC 4287 (Proposed Standard), December 2005. Updated by RFC 5988. URL: `http://www.ietf.org/rfc/rfc4287.txt`.

[66] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable cloud services for the internet of things with CoAP. In *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

Ringraziamenti

Se sono arrivata a questo traguardo, lo devo a me ma anche a tante altre persone, che ringrazio qui perchè senza di loro non sarei mai riuscita a raggiungere l'obiettivo.

Prima di tutti ringrazio il mio relatore Gianluigi Ferrari, che mi ha sostenuta e guidata nel corso degli ultimi anni, con competenza e pazienza.

Grazie a tutti i componenti dell'azienda Multitraccia: Francesco Gregori, Roberto D'Autilio, Fabio D'Autilio e Victor, per avermi dato la possibilità di intraprendere questo percorso, e di confrontarmi con sfide sempre nuove. Grazie anche per gli innumerevoli caffè!

Un grazie davvero speciale a Marco Picone e Simone Cirani: il vostro supporto (che non è stato solo tecnico!) e i consigli che mi avete dato durante il dottorato sono stati davvero essenziali, e mi hanno aiutata a migliorare. Grazie per la vostra grande disponibilità.

Grazie ad Andrea Gorrieri, Luca Davoli e Federico Parisi, con cui ho condiviso l'ufficio e gran parte del dottorato. Grazie per i consigli, la disponibilità, le chiacchiere, l'amicizia e naturalmente per la pazienza.

Devo dire grazie anche tutti i ragazzi che, mese dopo mese e a vario titolo, si sono avvicendati nel WasnLab: Marco Martalò, Matteo Giuberti, Giovanni Spigoni, Pietro Gonizzi, Stefania Monica, Muhammad Asim, Mirko Mancin, e Nicolò Strozzi, Gabriele Ferrari e Mattia Antonini. Lavorare in un ambiente sereno è davvero importante, e non scontato. Ognuno di voi mi ha insegnato qualcosa e mi ha regalato un sorriso, quindi grazie.

Grazie anche ai componenti del MultimediaLab: Luca Cattani, Carlo Tripodi, e

Davide Alinovi per i consigli, la compagnia e per aver reso speciale e divertente ogni pausa pranzo!

Grazie a tutta la mia famiglia e a Stefano, per avermi sostenuta e incoraggiata, e per credere in me sempre.