A Scalable Big Stream Cloud Architecture for the Internet of Things

Laura Belli, University of Parma, Parma, Italy

Simone Cirani, University of Parma, Parma, Italy

Luca Davoli, University of Parma, Parma, Italy

Gianluigi Ferrari, University of Parma, Parma, Italy

Lorenzo Melegari, University of Parma, Parma, Italy

Màrius Montón, WorldSensing, Barcelona, Spain

Marco Picone, University of Parma, Parma, Italy

ABSTRACT

The Internet of Things (IoT) will consist of billions (50 billions by 2020) of interconnected heterogeneous devices denoted as "Smart Objects:" tiny, constrained devices which are going to be pervasively deployed in several contexts. To meet low-latency requirements, IoT applications must rely on specific architectures designed to handle the gigantic stream of data coming from Smart Objects. This paper propose a novel Cloud architecture for Big Stream applications that can efficiently handle data coming from Smart Objects through a Graph-based processing platform and deliver processed data to consumer applications with low latency. The authors reverse the traditional "Big Data" paradigm, where real-time constraints are not considered, and introduce the new "Big Stream" paradigm, which better fits IoT scenarios. The paper provides a performance evaluation of a practical open-source implementation of the proposed architecture. Other practical aspects, such as security considerations, and possible business oriented exploitation plans are presented.

Keywords: *Big Stream, Cloud, Internet of Things (IoT), Low Latency, Open-Source Applications, Processing Graph, Real-Time Applications, Smart-X Applications*

INTRODUCTION

The actors involved in IoT scenarios have extremely heterogeneous characteristics (in terms of processing and communication capabilities, energy supply and consumption, availability, and mobility), spanning from constrained devices, also denoted as "Smart Objects (SOs)," to smartphones and other personal devices, Internet hosts, and the Cloud. Smart Objects are typi-

DOI: 10.4018/IJSSOE.2015100102

cally equipped with sensors and/or actuators and are thus capable to perceive and act on the environment where they are deployed. By 2020, 50 billions of Smart Objects are expected to be deployed in urban, home, industrial, and rural scenarios (Evans, 2011), in order to collect relevant information, which may be used to build new useful applications.

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 common Internet-based hosts or general-purpose Internet-ready devices. The most prominent driver for interoperability in the IoT is the adoption of the Internet Protocol (IP), namely IPv6 (Postel, 1981; Deering & Hinden, 1998). An IP-based IoT will be able to extend and interoperate seamlessly with the existing Internet.

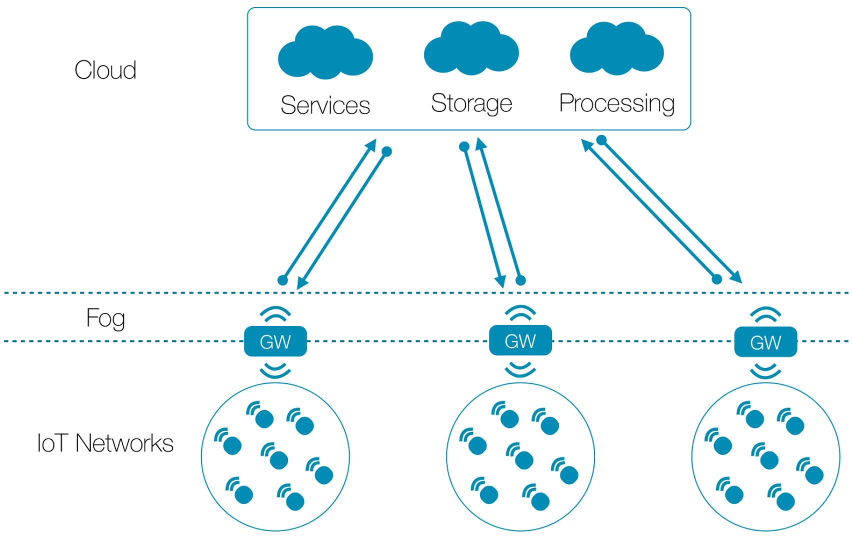
In a typical IoT scenario, sensed data are collected by SOs, deployed in and populating the IoT network, and sent uplink to collection entities (servers or the Cloud). In some cases, an intermediate element may support the Cloud, carrying out storage, communication, or computation operations in local networks (e.g., data aggregation or protocol translation). This approach is the basis of the Fog Computing (Bonomi, Milito, Zhu, & Addepalli, 2012) and will be better explained in the "Background" section.

Figure 1 shows the hierarchical structure of layers involved in data collection, processing, and distribution in IoT scenarios.

With billions of nodes capable of gathering data and generating information, the availability of efficient and scalable mechanisms for collecting, processing, and storing data is crucial.

Big Data techniques, which were developed in the last few years and became popular due to the evolution of online and social/crowd services, 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 of information (focusing on storage, aggregation, analysis, and provisioning of data), rather than to provide real-time processing and dispatching (Zaslavsky,

Figure 1. The hierarchy of layers involved in IoT scenarios: the Fog works as an extension of the Cloud to the network edge to support data collection, processing, and distribution



Perera, & Georgakopoulos, 2013; Leavitt, 2013). Cloud Computing has found a direct application with Big Data analysis due to its scalability, robustness, and cost-effectiveness.

One of the distinctive features of IoT systems is the deployment of a huge amount of heterogeneous data sources collecting data from the environment and sending information through the internet to collectors. The work of all data sources generate, as a whole, streams with a very high frequency. Moreover, several relevant IoT scenarios (such as industrial automation, transportation, networks of sensors and actuators) need real-time or predictable latency.

The number of data sources, on one side, and the subsequent frequency of incoming data, on the other side, create a new need for Cloud architectures to handle such massive information flows.

Big Data approaches typically have an intrinsic inertia because they are based on batch processing. For this reason, they are not suitable to the dynamicity of IoT scenarios with real-time requirements.

To better fit these requirements, the Big Data paradigm is shifted to a new paradigm, which has been denoted as "Big Stream" (Belli, Cirani, Ferrari, Melegari, & Picone, 2014). Big Stream-oriented systems should react effectively to changes and provide smart behavior for allocating resources, thus implementing scalable and cost-effective Cloud services. The Big Stream paradigm is specifically designed to perform real-time and ad-hoc processing in order to link incoming streams of data to consumers. This new paradigm should: have a high degree of scalability and fine-grained/dynamic configurability; and efficiently manage heterogeneous data formats which are not a priori known.

The main differences between Big Data and Big Stream paradigms can be summarized as follows.

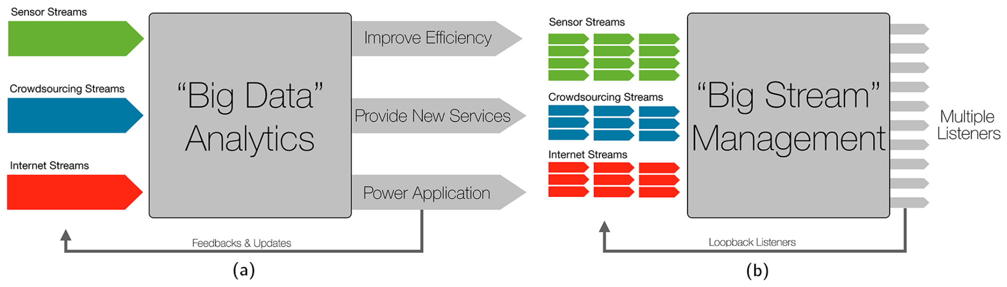
- The nature of data sources: Big Stream refers to scenarios with a huge number of data sources sending small amounts of information.
- The real-time or low-latency requirements of consumers: information in Big Stream IoT scenarios is usually short-lived and should be provided to consumers before it becomes outdated (and useless).
- The meaning of the adjective "Big:" for Big Data, it refers to the data volume, whereas for Big Stream it refers to the global aggregate information generation rate of data sources. Moreover, this has an impact on the data that are considered relevant to consumer applications. While for Big Data applications it is important to keep all sensed data, in order to be able to perform any required computation, Big Stream applications might decide to perform local data aggregation/pruning, in order to minimize the latency in conveying the final processing results to consumers, without persistence needs.

In conclusion, although both Big Data and Big Stream deal with massive amounts of data, they have different purposes (as shown in Figure 2): the former focuses on storage, analysis and interpretation of data, while the latter focuses on data flow management in order to provide informations to interested customers with minimum latency.

From a more general perspective, Big Data applications might be consumers of Big Stream data flows.

For the above observations, the objective of this paper is to propose an architecture targeting Cloud-based applications with real-time constraints, i.e., Big Stream applications, for IoT scenarios. The proposed architecture relies on the concepts of a data listener and data-oriented processing graph in order to implement a scalable, highly configurable, and dynamic chain of computations on incoming Big Streams and to dispatch data with a push-based approach, thus providing the lowest delay between the generation of information and its consumption.

Figure 2. (a) Data sources in Big Data systems. (b) The multiple data sources and listeners management in Big Stream system



BACKGROUND

The IoT paradigm refers to a huge number of different and heterogeneous SOs connected in a worldwide "Network of Networks." These nodes are envisioned as collectors of information from the environment in order to provide useful services to users. This ubiquitous sensing, enabled by the IoT in most areas of modern living, has led to information and communication systems invisibly embedded in the environment, thus making the technology disappear from the consciousness of the users. The outcome of this trend is the generation of a huge amount of data that, depending on the specific application scenario, should be processed, aggregated, stored, transformed, and delivered to the final users of the system, in an efficient and effective way, with traditional commodity services.

In the next sections, related works regarding Cloud, Big Data, IoT architectures and models are first presented; then, some suitable open-source technologies and protocols are listed.

IoT Architectures

A large number of architectures for IoT scenarios have been proposed in the literature. For instance, most of the ongoing projects on IoT architectures address relevant challenges, particularly from a Wireless Sensor Networks (WSN) perspective. Some examples are given by a few European Union projects, such as SENSEI project (European Community's 7th Framework Programme, 2008-2010) and Internet of Things-Architecture (IoT-A) project (European Community's 7th Framework Programme, 2012-2015).

The purpose of the SENSEI project is to create an open and business-driven architecture that addresses the scalability problems for a large number of globally distributed Wireless Sensor and Actuator (WS&A) network devices, enabling interactions with physical environment.

The IoT-A project consortium has focused on the definition of an initial set of key building blocks, aiming at creating open interoperable platforms, connecting vertically closed architectures.

"Connect All IP-based Smart Objects!" (CALIPSO) (European Community's 7th Framework Programme, 2011-2014) is another European project whose main purpose is to build IoT systems with IPv6-connected and low-power consumption SOs, thus providing both high interoperability and long lifetime, entailing three communication protocol stack layers (network, routing, and application).

In (Gubbi, Buyya, Marusic, & Palaniswami, 2013) is proposed an IoT architecture which is not based on WSNs and is focused instead on the user and the Cloud. The consumer is the "center" and drives the use of data and infrastructure to develop new applications. The rest of the work discusses the key enabling technologies and the different future applications domains, describes a Cloud-centric architecture for IoT, and presents a real implementation.

Another Cloud-based IoT architecture is proposed in the FI-WARE project (European Community's 7th Framework Programme, 2011), an open infrastructure with public, royalty-free, and OCCI-compliant API, providing to developers a platform to build innovative products.

As previously stated, the most prominent driver to provide interoperability in the IoT, referring to IP stack, is IPv6. At the application layer, the IoT scenario brings a variety of possible protocols that can be employed according to the specific applications requirements. Relevant options are: (i) HyperText Transfer Protocol (HTTP) (R. Fielding et al., 1999); (ii) Constrained Application Protocol (CoAP) (Shelby, Hartke, Bormann, & Frank, 2014); (iii) Extensible Messaging and Presence Protocol (XMPP) (Saint-Andre, 2004); (iv) MQ Telemetry Transport (MQTT) protocol (Locke, 2010); (v) Constrained Session Initiation Protocol (CoSIP) (Cirani, Picone, & Veltri, 2013, 2014; Cirani, Davoli, Picone, & Veltri, 2014).

Regardless of the selected application-layer protocol, most IoT/M2M applications follow the REpresentational State Transfer Protocol (REST) architectural model presented in (R. T. Fielding, 2000), as this provides simple and uniform interfaces and is designed to build long-lasting, robust, and resilient to changes applications.

Big Data Processing Pattern

From a business perspective, managing and gaining insights from data is a challenge and a key to competitive advantage. Analytical solutions that mine structured and unstructured data are important, as they can help companies to gain cross-related information not only from their privately acquired data, but also from large amounts of data publicly available on the Web, social networks, and Blogs. Big Data opens a wide range of possibilities for organizations to understand the needs of their customers, predict their demands, and optimize the use of evaluable resources.

The work of (McAfee & Brynjolfsson, 2012) illustrates that the Big Data notion is different and more powerful with respect to traditional analytics tools used by companies. As analytics tools, Big Data can find patterns and glean intelligence from data translating that into business advantage. However, Big Data is powered by what is often referred as a multi V model, in which V stands for:

- **Variety:** To represent the data types;
- **Velocity:** To represent the rate at which the data is produced and processed and stored according with further analysis;
- **Volume:** To define the amount of data;
- **Veracity:** Refers to how much the data can be trusted given the reliability of its sources.

Big Data architectures generally use traditional processing patterns with a pipeline approach (Hohpe & Woolf, 2003). These architectures are typically based on a processing perspective where the data flow goes downstream from input to output, to perform specific tasks or reach the target goal.

Typically, the information follows a pipeline where data are sequentially handled with tightly coupled pre-defined processing sub-units (static data routing). The described paradigm can be defined as "process-oriented:" a central coordination point manages the execution of subunits in a certain order and each sub-unit provides a specific processing output, which is created to be used only within the scope of its own process without the possibility to be shared among different processes. This approach represents a major deviation from traditional Service Oriented

Architectures (SOAs), where the sub-units are external web services invoked by a coordinator process rather than internal services (Isaacson, 2009). Big Data applications generally interact with Cloud Computing architectures which can handle resources and provide services to consumers.

In (Assunção, Calheiros, Bianchi, Netto, & Buyya, 2014), the authors presents a survey on approaches, environments, and technologies on key-areas for Big Data analytics capabilities, investigating how they can contribute to build analytics solutions for Clouds. A set of gaps and recommendations, for the research community, on future directions on Cloud-supported Big Data computing are also described.

Fog Computing

In the area of user-driven and Cloud IoT architectures, (Bonomi, Milito, Zhu, & Addepalli, 2012) propose Fog Computing as a novel and appropriate paradigm for a variety of IoT services and applications that require mobility support, low latency, and location awareness.

The Fog can be described as a highly virtualized platform that provides computing, storage, and networking services between end-devices and the Cloud. In other words, the Fog is meant to act as an extension of the Cloud, operating at the edge of the network to support endpoints by providing rich services that can fulfill real-time and low-latency consumer's requirements. The Fog paradigm has specific characteristics, which can be summarized as follows:

- Geographical distribution, in contrast with the centralization envisioned by the Cloud;
- Subscriber model employed by the players in the Fog;
- Support for mobility.

The architecture described by (Bonomi, Milito, Zhu, & Addepalli, 2012) is based on the Fog and Cloud interplay: the former provides localization, low latency, and context awareness to endpoints; the latter provides global centralization functionalities. In the presented IoT Fog scenario, collectors at the edge of the network manage the data generated by sensors and devices: the portion of these data that require real-time processing (from milliseconds to tenths of seconds) are consumed locally by the first tier of the Fog. The rest is sent to the higher tiers for operations with less stringent time constraints (from seconds to minutes). The higher is the tier, the wider is the geographical coverage and the longer the time scale. As a result, the Fog must support several types of storage: from ephemeral, at the lowest tier, to semi-permanent, at the highest tier. The ultimate and global coverage is provided by the Cloud, which is used as repository for data with a potential duration of months or years.

Stream and Real-Time Management

The architecture proposed in the current paper is specifically designed for scenarios with low latency and real-time requirements. Other projects related to real-time and stream management are Apache Storm (Apache, n.d-a.) and Apache S4 (Neumeyer, Robbins, Nair, & Kesari, 2010).

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

Although there are several similarities between these systems and the architecture proposed here, such as modularity, scalability, latency minimization and the graph topology, there are some notable differences. The most relevant use cases for Storm and S4 are stream processing and continuous computations on data stored in databases (e.g., message processing for database update). The proposed architecture, on the other hand, is specifically designed to work in dynamic IoT scenarios comprising heterogeneous data sources and making no assumption on the repositories (if needed) where data can be retrieved or stored.

Another major difference is related to the nature of the topology of the processing units. While Storm stream management is based on an operator-defined and static graph topology, the architecture proposed in the remainder of this paper is extremely dynamic, as the number of nodes and edges in the Graph Framework can change according to the workload and listener's requirements.

The works described in (Marganiec et al., 2014; Tilly & Reiff-Marganiec, 2011) address the problem to process, procure, and provide information related to the IoT scenario with almost zero latency. The authors consider, as a motivating example, 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 (Marganiec et al., 2014) 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.

Cloud Computing

Cloud Computing represents the increasing trend moving to the external deployment of Information Technology (IT) resources, obtaining them as services (Stanoevska-Slabeva, Wozniak, & Ristol, 2009). Cloud Computing enables convenient and on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage elements, applications, and services) that could be rapidly provisioned and released with minimal management effort or service provider interaction (Mell & Grance, 2011).

At hardware level, a number of physical devices, including processors, hard drives, and network devices, fulfill processing and storage needs. Above this, the combination of (i) software layer, (ii) virtualization layer, and (iii) management layer, allows effective management of servers. In Cloud Computing, available service models are the following.

- **Infrastructure as a Service (IaaS):** Provides processing, storage, networks, and other computing resources, allowing the consumer to deploy and run arbitrary software, including OSs and applications. The consumer has control over OSs, storage, deployed applications and, possibly, limited control of select networking components.
- **Platform as a Service (PaaS):** Provides the capability to deploy infrastructure, consumer-created, or acquired applications. The consumer has no control on the underlying infrastructure (e.g., network, servers, OSs, or storage) but only manages deployed applications.
- **Software as a Service (SaaS):** Provides the capability to use the provider's applications, running on the Cloud infrastructure, by accessing from various client devices through proper client interfaces. The consumer does not manage or control the underlying Cloud

infrastructure or individual application capabilities, with the possible exception of limited user-specific application configuration settings.

Cloud Computing is generally complementary to the IoT scenario, as it acts (i) as collector of real-time sensed data and (ii) as provider of services built on the basis of collected informations. The main need is to be extremely scalable, allowing the support to large-scale IoT applications.

There are several open source frameworks and technologies which can be used for Cloud IoT systems, such as OpenStack (Rackspace, NASA, n.d.) and OpenNebula (Milojičić, Llorente, & Montero, 2011). The former is an open Cloud OS that controls large pools of computing, storage, and networking resources, while OpenStack can be seen as a framework with a vendor-driven model, the second is an open-source project aiming at delivering a simple, feature-rich, and flexible solution to build and manage enterprise Clouds and virtualized data centers.

ARCHITECTURE

As previously stated, a major difference between Big Data and Big Stream resides in the real-time and low-latency requirements of consumers. The gigantic amount of data sources in IoT applications has mistakenly made Cloud services implementers believe that re-using Big Data-driven architectures would be the right solution for all applications, rather than designing specific paradigms for those scenarios.

IoT application scenarios are characterized by a huge number of data sources, sending small amounts of information to a collector service, typically at a limited data rate. Many services can be built on top of these data, such as environmental monitoring, building automation, and smart cities applications. These applications are typically characterized by low-latency or real-time requirements, in order to provide efficient reactive/proactive behaviors.

Big Stream Oriented Architecture

Applying a traditional Big Data approach for IoT application scenarios might lead to high - 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.

Figure 3 illustrates the main delay contributions introduced when data, generated by SOs in IoT networks, need to be processed, stored, and then polled by consumers. Clients interested in processed data are extremely heterogeneous, spanning from mobile or desktop applications to Data Warehouse (DW) applications and till other IoT Smart Objects networks.

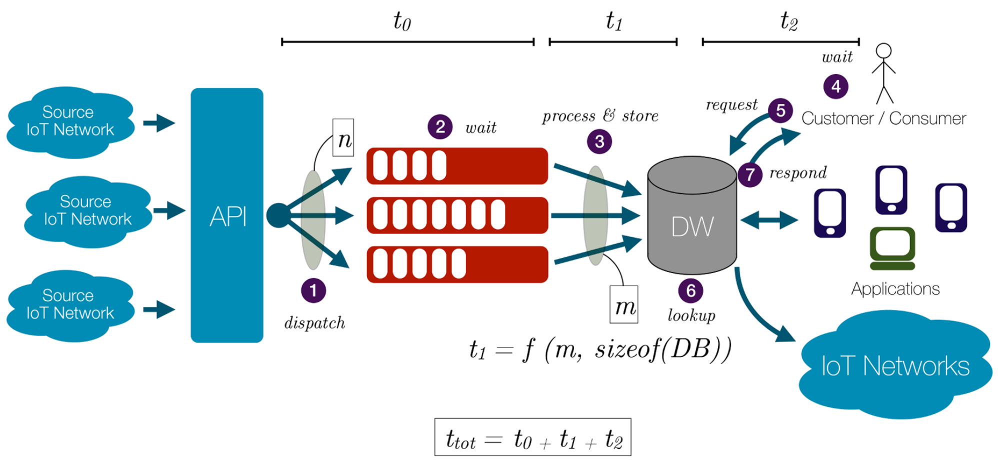
The total delay required by any data to be delivered to a consumer can be expressed as:

$$T = t_0 + t_1 + t_2$$

where:

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

Figure 3. Delay contributions in a traditional Big Data architecture for IoT, from data generation to applications information delivery



- t_1 is the time needed for data, extracted by the queue, to be pre-processed and stored into a DW (3): this time contribution 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: (i) the remaining time that a polling consumer needs to wait before performing the next fetch (4); (ii) the time for a request to be sent to the Cloud service (5); (iii) the time required for lookup in the DW and post-process the fetched data (6); and (iv) the time for the response to be delivered back to the consumer (7).

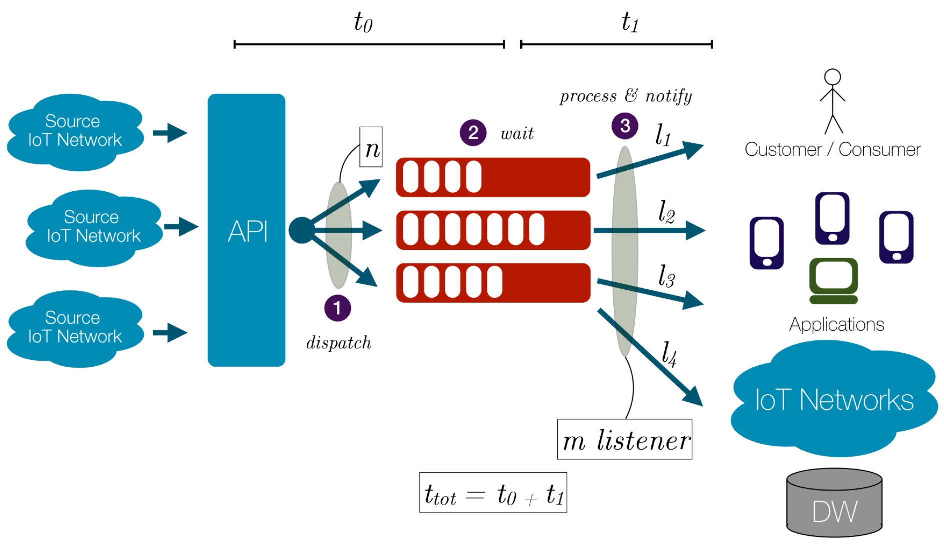
It can be observed that the architecture described is not optimized to minimize the latency and, therefore, to feed (possibly a large number of) real-time applications but, rather, to perform data collection and batch processing. Moreover, it is important to underline and understand that significant data 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.

The main design criteria of the architecture proposed in this paper are:

- The minimization of the latency in data dispatching to consumers;
- The optimization of resource allocation.

The main novelty in the presented architecture lies in the concepts of "consumer-oriented" data flows and "listeners." The former denotes a different approach in retrieving incoming data, rather than being based on the knowledge of collection points (repositories) to which request data. The latter relies on final consumers: data generated by a deployed Smart Object, might be of interest for some consumer application, denoted as listener, which can register itself in order to receive updates (either in the form of raw or processed data) coming from a particular streaming endpoint (i.e., Cloud service). On the basis of application-specific needs, each listener

Figure 4. The delay contributions from data generation to consumers information delivery following the listener-based Big Stream approach



defines a set of rules, which specify what type of data should be selected and the associated filtering operations. For instance, referring to a smart parking scenario, a mobile application might be interested in receiving contents related only to specific events that occur within a given geographical area, in order to accomplish relevant tasks. Specifically, the application can listen for parking sensor status updates, the positions of other cars, or weather conditions, in order to find available parking spots.

The proposed Big Stream architecture guarantees that, as soon as they are available, data 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 4.

With the Big Stream paradigm, the total time required by any data to be delivered to a consumer can be expressed as:

$$T = t_0 + t_1$$

where:

- t_0 is the same time delay contribution defined for Figure 3;
- 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 perspective inversion 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 Big Data, an alerting application (where an event should be notified to one or more consumers in the shortest possible time) can be considered. The traditional Big Data approach would require an unnecessary pre-processing/storage/post-processing cycle to be executed before the event can 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 than simply reusing existing Big Data architectures, which better fit applications that do not have critical real-time requirements.

Graph-Based Processing

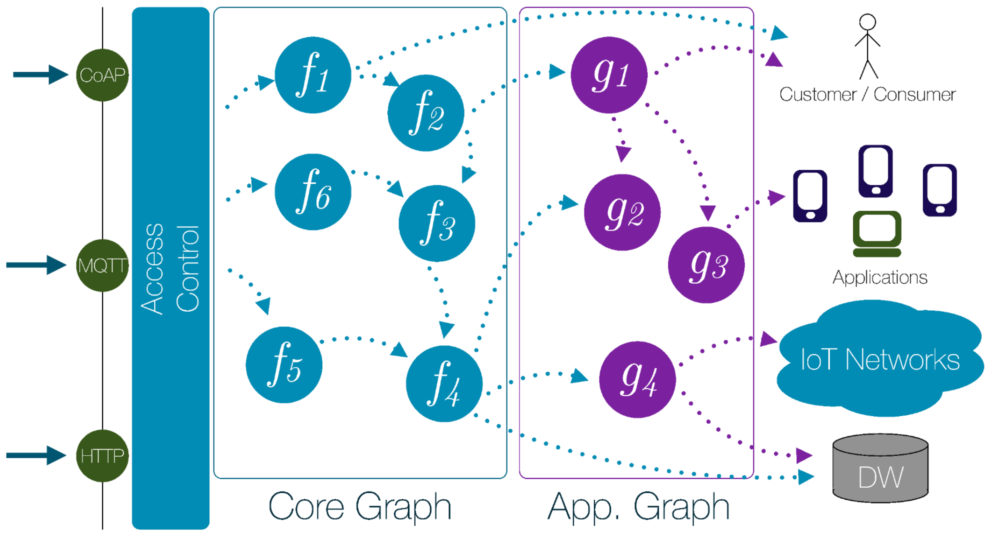
In order to overcome the limitations of the "process-oriented" approach described in the previous section and fit with the proposed Big Stream paradigm, the proposed Cloud architecture is based on a Graph Framework. More precisely, we consider a graph composed by basic building blocks that are self-consistent and perform "atomic" processing on data, but that are not directly linked to a specific task. In such a system, the data flows are based on dynamic graph-routing rules determined only by the nature of the data itself and not by a centralized coordination unit. This new approach allows the platform to be "consumer-oriented" and to implement 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 (if some processing units are not required at a certain point) and transparently replicate nodes (if some processing entities are consumed by a significant amount of concurrent consumers).

Figure 5 illustrates the proposed directed Graph-based processing architecture and 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, in this architectural approach:

- Nodes are processing units (processes), performing some kind of computation on incoming data;
- Edges represent flows of information linking together various processing unit, which are thus able to implement some complex behavior as a whole;
- Nodes of the graph are listeners for incoming data or outputs of other nodes of the graph.

The designed Graph-based approach 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) and 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 to add new processing units and functionalities on demand.

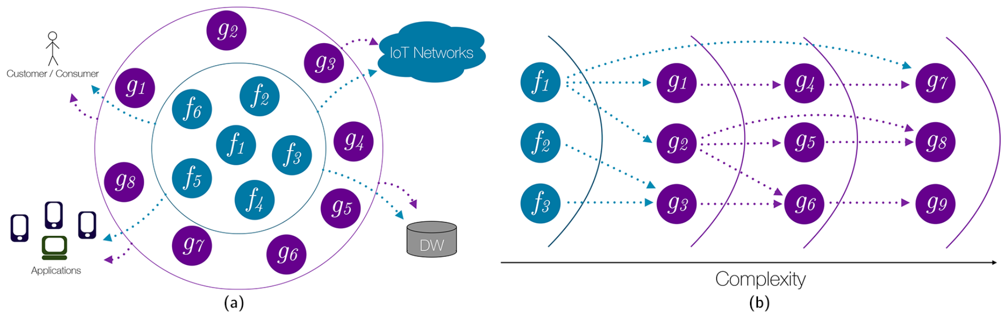
Figure 5. The proposed listener-based Graph architecture: the nodes of the graph are listeners; the edges refer to the dynamic flow of information data streams



In order to provide a set of commonly available functionalities, while allowing to dynamically extend the capabilities of the system, the graph is composed by concentric layers. Each layer contains two types of nodes, as shown in Figure 6 (a):

- **Core Graph Nodes:** Listeners which perform basic processing operations provided by the architecture (e.g., format translation, normalization, aggregation, data correlation, and other transformations);
- **Application Graph Nodes:** Listeners that require data coming from an inner graph layer in order to perform custom processing on already processed data.

Figure 6. (a) The concentric linked Core and Application Layers. (b) Basic processing nodes build the Core Graph Layer, the outer nodes have increasing complexity



The architecture thus consists of a single Core Layer including many core nodes, and several Application Layers containing application nodes. The complexity of processing is directly proportional to the number of layers crossed by the data. This also means that data at an outer graph layer must not be processed again at an inner layer, which also guarantees that processing loops, due to misconfigurations, are avoided by design.

From an architectural viewpoint, as shown in Figure 6 (b), 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 may be linked together if there is a need to do so.

In particular, a processing unit of the Core Graph layer can be a listener only for other nodes of the same layer (*n* incoming streams) and a source for other Core and Application graph nodes (*m* outgoing streams). A node of an Application Graph layer can be, at the same time:

- A listener of *n* incoming flows from Core and/or Application graph layers;
- A data source only for other *m* nodes of the application graph layers or heterogeneous external consumers.

The overall behavior of a task is generated by following a complete path in the Graph from a data source to a final consumer. Processing units perform operations that can be reused, thus data produced by a node can belong to several different paths and can be forwarded to all interested listeners. For this reason, in order to optimize the workload nodes with a large number of listeners can be replicated and nodes with no listeners can be shut down.

IMPLEMENTATION

In this section, the details of the functionalities and implementation of the proposed architecture by using standard protocols and open-source components are presented (Belli, & al., 2015).

Three main modules concur in forming the entire system:

- Acquisition and normalization of the incoming raw data;
- Graph management;
- Application register entity.

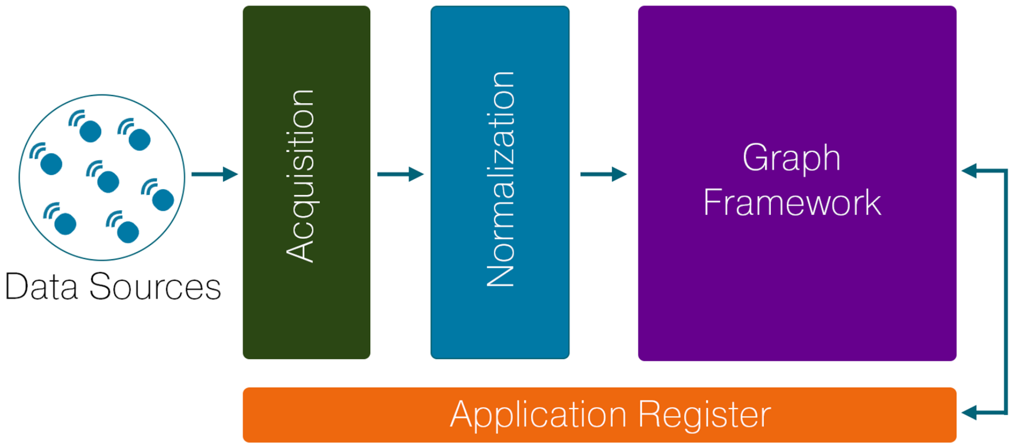
All modules and their relationships are shown in Figure 7. A detailed explanation is given in the following sections.

Acquisition Module

The Acquisition Module represents the entry point, for external IoT networks of SOs, to the Cloud architecture. Its purpose is to receive incoming raw data from heterogeneous sources, making them available to all subsequent functional blocks. As mentioned before, about IoT models, several application-layer protocols can be implemented by SOs; adhering to this idea, the Acquisition Module has been modeled to include a set of different connectors, in order to properly handle each protocol-specific incoming data stream.

Considering the main and most widespread IoT application-layer protocols, the current implementation of the Acquisition Module supports: HTTP, CoAP and MQTT.

Figure 7. Components of the proposed Graph Cloud architecture and relations between each element



In order to increase scalability and efficiency, in the module implementation an instance of NGINX (Reese, 2008) has been adopted as an HTTP acquisition server node. The server is reachable via the default HTTP port, working with a dedicated PHP page, as processing module, which has been configured to forward incoming data to the inner queue server. We have chosen NGINX, instead of the prevailing and well-known open source Apache HTTPD Server (R. T. Fielding & Kaiser, 1997), because it uses an event-driven asynchronous architecture to improve scalability and, specifically, aims to guarantee a high performance even in the presence of a critical number of requests.

The CoAP acquisition interface has been implemented using a Java process, based on a mjCoAP server (Cirani, Picone, & Veltri, 2014) instance, waiting for incoming raw messages, and connected to the RabbitMQ queue server (RabbitMQ, n.d.), passing it injected elements. Indeed, since the proposed architecture is Big Stream-oriented, a well-fitting messaging paradigm is given by queue communication; therefore, in the developed platform an instance of RabbitMQ queue broker was adopted.

The MQTT acquisition node is built by implementing an ActiveMQ (Apache, n.d.) server through a Java process which listens for incoming data over a specific input topic (*mqtt.input*).

This solution has been preferred over other existing solutions (e.g., the C-based server Mosquitto) 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 module has dedicated Exchange module and queue (managed by RabbitMQ), linked together with a protocol-related routing key, ensuring the efficient management of incoming streams and their availability to the subsequent nodes.

In the described implementation, an Exchange is a RabbitMQ component which acts as a router in the system and dispatches incoming messages to one or more output queues, following dynamic routing rules.

Normalization Module

Since incoming raw data are generally application- and theme-dependent, 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 operations on data such as:

- Suppression of useless information (e.g., unnecessary headers or meta-data);
- Annotation with additional information;
- Translation of the payload to a suitable format.

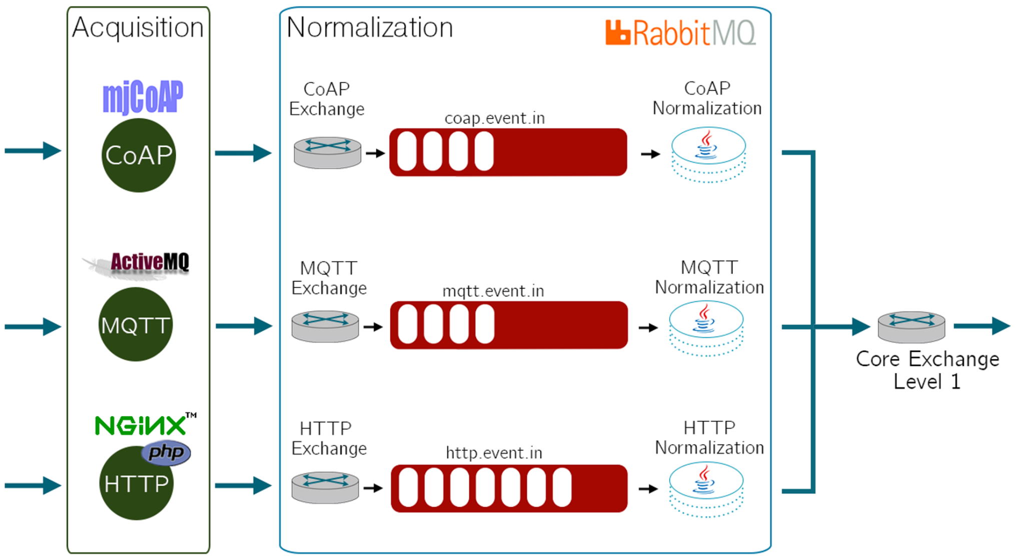
In order to handle the huge amount of incoming data efficiently, the normalization step is organized with protocol-specific queues and Exchanges.

As shown in the normalization section of Figure 8, the information flow originated by the Acquisition Module is handled as follows.

- All protocol-specific data streams are routed to a dedicated protocol-dependent Exchange, which forwards them to a specific queue.
- A normalization process handles the input data currently available on that queue and performs all necessary normalization operations in order to obtain a stream of information units that can be processed by subsequent modules.
- The normalized stream is forwarded to an output Exchange.

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.

Figure 8. Detailed representation of Acquisition and Normalization blocks



Each normalization node has been implemented as a Java process, which analyzes incoming raw data extracted from a queue identified through a protocol-like routing key (e.g., *<protocol>.event.in*), leaving unaltered the associated routing key, which identifies the originator SO protocol. The received data are fragmented and encapsulated into a JSON-formatted document, which provides an easy-to-manage format.

At the end of the normalization chain, each processor node forwards its new output chunk to its next Exchange that represents the entry-point of the Graph Module, promoting data flows to next layers of the proposed architecture.

Graph Framework

The Graph Framework is composed by an amount of different computational processes representing a single node in the topology; layers are linked together with frontier Exchanges, forwarding data streams to their internal nodes.

Each Graph node *i* of a specific layer *n* is a listener, waiting for input data stream on a dedicated layer *n* Exchange-connected queue. If this node also acts as publisher, after performing its processing on input data, it can deliver computation results to the its layer *n* Exchange. In order to forward streams, informations generated by node *i* become available for layer *n* and layer *n+1* listeners, interested for this kind of data, thanks to the binding between layer *n* and layer *n+1* Exchanges.

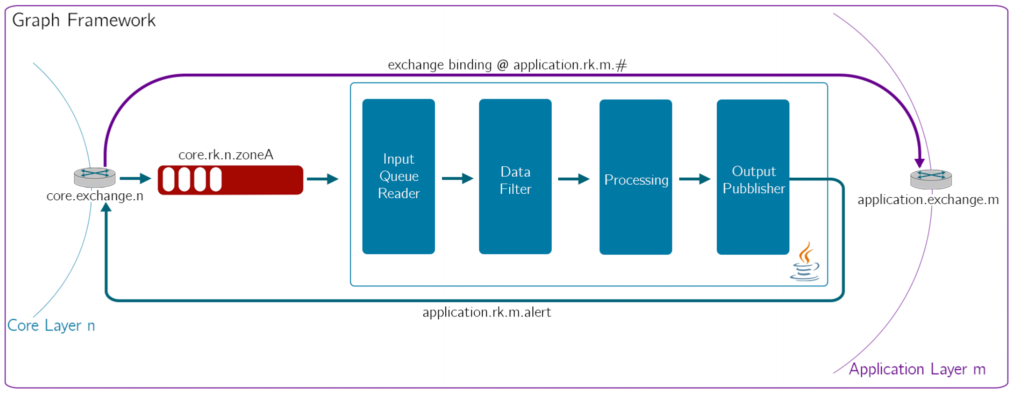
Incoming messages are stored into active queues, connected to each Graph Layer's Exchange. Queues can be placed into the Core Graph layers, for basic computation, or into Application Graph Layers, for enhanced computation. Layers are connected, through one-way links, with their successor Exchange by using the binding rules allowed by the 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 nodes can either be Core, if they are dedicated to simple and primitive data processing, or Application, if they are oriented to a more complex and specific data management. Messages, identified with 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 shown in Figure 9. If the outgoing routing key belongs to the same incoming graph layer, data remain into same Exchange and become 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 adhering to binding rules. 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 only as data consumer. 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 (e.g., Data Warehouse, browsers, Smart entities, other Cloud Graph processes).

Application Register Module

The Application Register Module has the fundamental responsibilities (i) to manage the processing graph by maintaining all the information about the current statuses of all graph nodes in the system and (ii) 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;

Figure 9. Interaction between Core and Application layers with binding rule

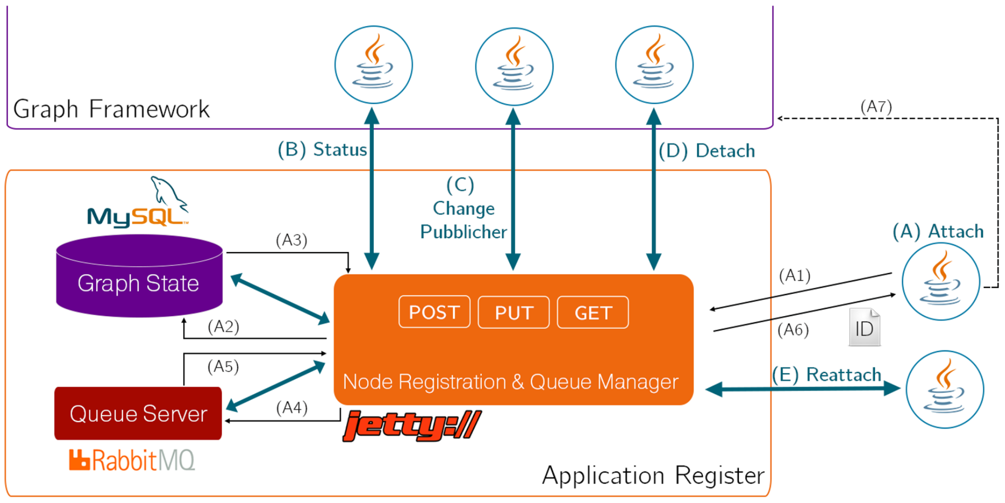


- Detach nodes of the graph that are no more interested in streaming flows and eventually re-attach them;
- Handle nodes that are publishers of new streams;
- Maintain information regarding topics of data, in order to correctly generate the routing keys and to compose data flow between nodes in different graph layers.

In order to accomplish all these functionalities, the Application Register Module is composed by two main components, as shown in Figure 10.

The first module is the Graph State Database, which is dedicated to store all the information about active graph nodes, such as: their states, layers, and whether they are publishers. The second one is the Node Registration and Queue Manager (NRQM), which handles requests from

Figure 10. Detailed representation of the Application Register module, with possible actions that may be performed by Graph nodes, highlighting ATTACH request steps needed to include an external node in the Graph



graph nodes or external processes, and handles queue management and 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 which it is interested to. The NQRM module stores the information of the 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, and get status).

The overall architecture is managed by a Java process (Application Register), which has the role to coordinate 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, and then it activates each layer's Exchange, binding them with their successors. At the end, it proceeds with the activation of a Jetty HTTP server, responsible for listening and handling all Core and Application nodes requests, as shown in Figure 10: (A) attach, (B) status request, (C) change publishing policy, (D) detach, and (E) re-attach request, using a RESTful HTTP paradigm.

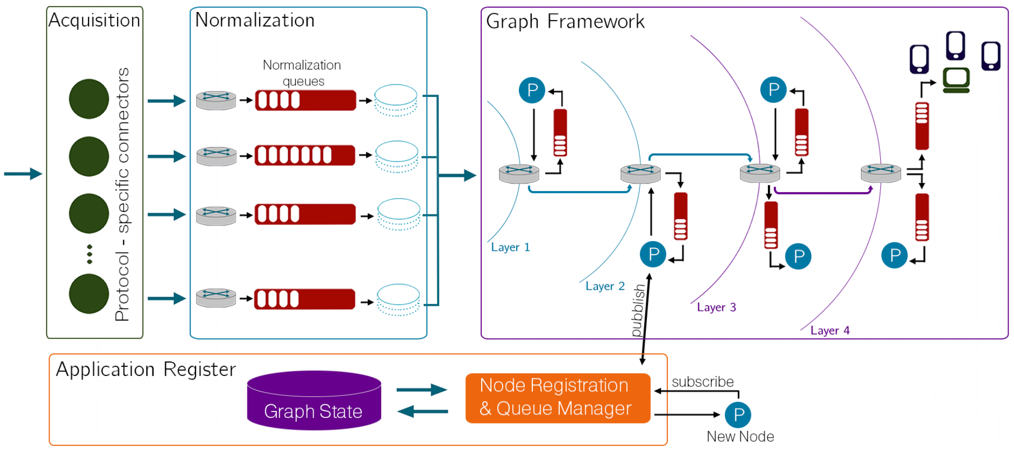
In Figure 11, all the proposed architecture modules described above, with a detailed indication of the information flows, are shown.

PERFORMANCE EVALUATION

The implementation of the proposed Graph Framework for Big Stream management has been carried out by deploying an Oracle VirtualBox VM, equipped with Linux Ubuntu 12.04 64-bit, 4GB RAM, 2 CPUs and 10GB HDD.

The implemented architecture has been evaluated 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 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

Figure 11. The complete Graph Cloud Architecture, with reference to the data stream flows between all building blocks, from IoT data sources to final consumers



deployment (more than 10,000 sensors) with information from 400 sensors over a 3 month period, forming a dataset with more than 604k parking events.

Each dataset item is represented by: (i) sensor ID; (ii) event sequence number, relative to the specific sensor; (iii) event timestamp; and (iv) parking spot status (free/busy). No additional informations about parking zone are provided. Therefore, thus, in order to create a realistic scenario, parking spot sensors are divided into 7 groups, representing different parking zones of a city. This parking spot-city zone association is stored into an external database.

Experimental Setup

The parking dataset has been used in the Cloud infrastructure using a Java-based data generator, which simulates the IoT sensors network. The generator randomly selects an available protocol (HTTP, CoAP, or MQTT) and periodically sends streams to the corresponding acquisition node interface. Once the data has been received by the acquisition layer, they are forwarded to the dedicated normalization Exchange, where corresponding nodes enrich incoming data with platform-specific details. With reference to the selected scenario, the normalization stage adds parking zone details to input data, retrieving the association from an external database. Once the normalization module has completed its processing, it sends the structured data to the Graph Framework, allowing to further process the enriched data stream.

The Graph Framework considered in our experimental set-up is composed by 8 Core layers and 7 Application layers, within which different node topologies are built and evaluated.

Processed data follow a path based on routing keys, until the final external listener is reached. 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 → busy, busy → free).

The rate of these events, coming from a real deployment in a European city, respects some rules imposed by the company, and for our purposes might seems low. Thus, in order to stress enough the proposed Big Stream Cloud system, the performance is evaluated by varying the data generation rate in a proper range. In other words, we force a specific rate for incoming events, without taking into account real parking spots timestamps gathered from the dataset.

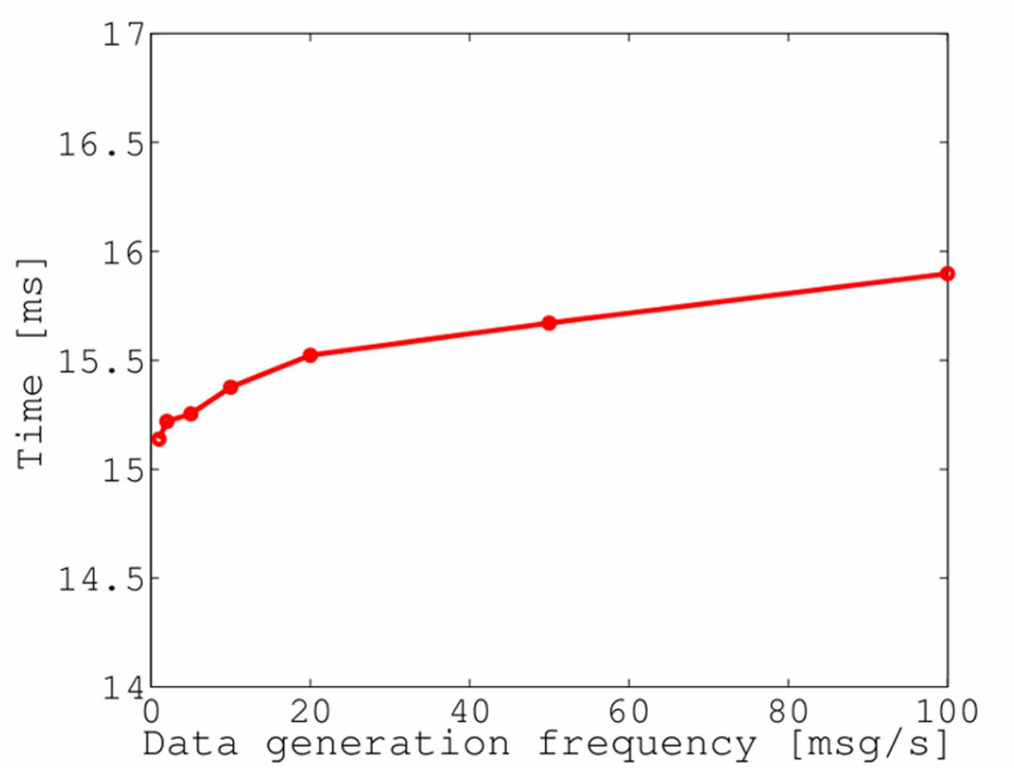
Results

The proposed architecture has been evaluated, using the testbed described in the previous subsection, by varying the incoming raw data from 1 msg/s to 100 msg/s. The evaluation consists in assessing the performance of the acquisition stage and the computation stage.

First, performance is evaluated by measuring the time difference (dimension: [ms]) between the instant at which data are sent from a data generator to the corresponding acquisition interface and the instant at which the data are enriched by normalization nodes, thus becoming available for the first processing Core Node. The results are shown in Figure 12. 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 at which enriched data become ready for processing activities and the time instant at which the message reaches the end of its Graph Framework routes, becoming available for external consumers/customers. In order to consider only the effective overhead introduced by the architecture, and without considering 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

Figure 12. Average time (dimension: [ms]) related to the acquisition block



order to obtain the per-node overhead introduced by the architecture, in a way that is independent of the specific routing and topology that were implemented. The results, shown in Figure 13 and Figure 14, have thus been calculated using the following expression:

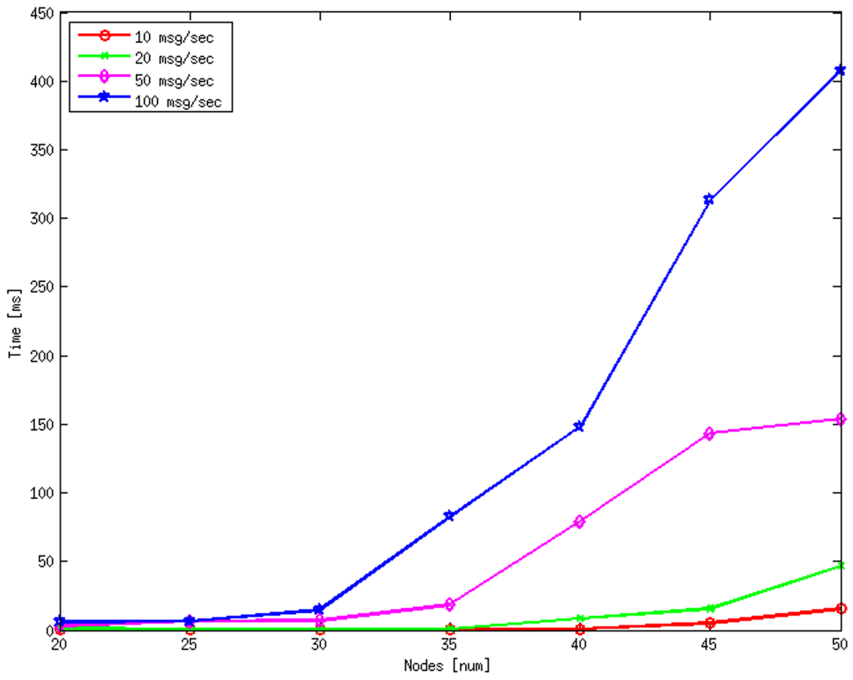
$$T_{processing_{freq}} = \frac{T_{out} - T_{in} - \sum_{k=1}^{N} GP_k}{N}$$

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

Figure 13 shows how $T_{processing}$ values grow increasing the data generation frequency (from 10 msg/s to 100 msg/s). Each curve is related to a different Graph topology.

Figure 14 shows how $T_{processing}$ values grow increasing the number of nodes composing the Graph topology (from 20 to 50 nodes). Each curve in Figure 14 is related to a different value of frequency rate.

Figure 13. Average times (dimension: [ms]) related to Graph Framework processing block, showing per-node time, varying data generation rate, for each subset of nodes deployed into the Graph topology



DISCUSSIONS

Solutions and Security Considerations

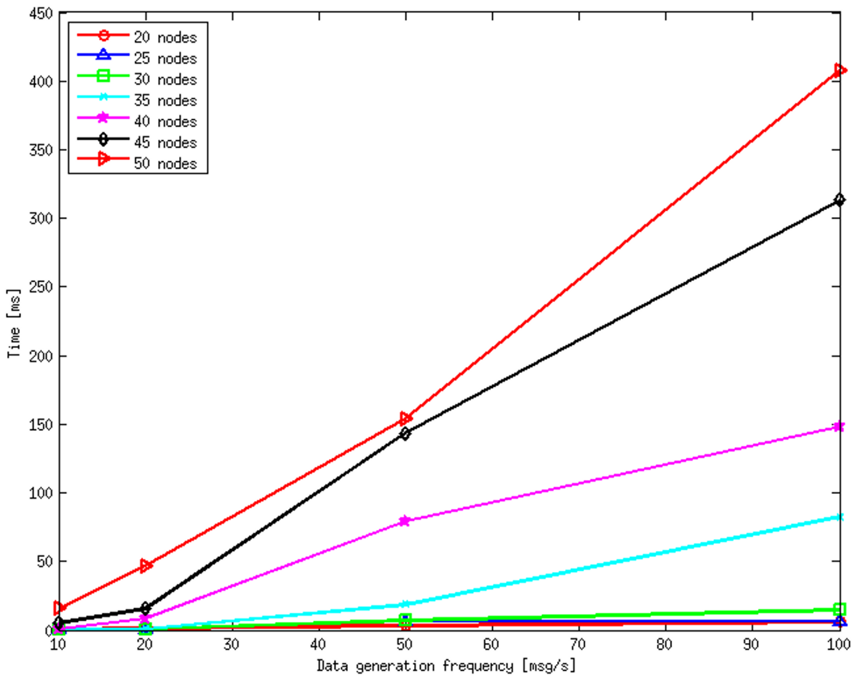
The presented architecture is designed with reference to a specific IoT scenario with strict latency and real-time requirements, namely a smart city-related Smart Parking scenario. There are several possible use cases and applications fitting this scenario, alerting or real time monitoring applications.

The work of (Vilajosana et al., 2013) shows 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 (Vilajosana et al., 2013) 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.

The authors advocate three different stages 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.

Figure 14. Average times (dimension: [ms]) related to Graph Framework processing block, showing per-node time, varying the subset of nodes deployed into the Graph topology, for each evaluated data generation frequency



- **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, (Vilajosana et al., 2013) propose three main different business models 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 this paper is compatible with the steps described in the work of (Vilajosana et al., 2013) 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.

Another aspect, with a relevant impact on the business model, is security. This entails both processing module and interaction with external entities. It is possible to adopt different policies related to authentication and/or authorization on data sources, e.g., based on well-known and standard solutions such as OAuth (Hammer-Lahav, 2010; Hardt, 2012), avoiding data stream malicious alterations and following negative consequences, that could affect both processing results and platform reliability. At a final stage, security could be applied for consumer accounting and authentication, ensuring appropriate platform access only by authenticated/authorized entities, and providing security transactions, with authorized entities, via secured communications.

Security features, including authorization, authentications and confidentiality, should be integrated into the architecture, in order to make the implementation complete and usable. Details about integration of security features in the proposed Big Stream platform and its further impact on the system performance are not included in this paper. They represent interesting research topics for future work.

Practical Use

In the previous sections, we have detailed the implementation of the Graph-based Cloud architecture for a Big Stream IoT scenario. This section addresses some aspects regarding practical use of the proposed architecture, taking into account its deployment on a Cloud platform.

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-a-Service (SaaS) model, providing 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 (IBM Emerging Technology, 2013), a WEB-based application, running on Node.js engine, which allows developers to create IoT graphs, wiring together hardware devices, APIs, and online services.

FUTURE RESEARCH DIRECTIONS

The proposed architecture is oriented to large amounts of incoming raw data, providing to interested consumers an enhanced version of them: this could be useful in scenarios in which consumers are final entities, interested only in retrieving aggregated data. The proposed architecture could also be seen as a first-step processing platform, in which final data could represent an incoming set for other processing entities. This flow could be applied to many scientific fields: for example, since the proposed architecture is not a simulation or emulation platforms, could serve as data provider for instances of those processors types. In medical environments, the proposed platform could be seen as a platform trying to work on an enhanced dataset, looking for some diagnosis. Other possible applications fields, are related to mobility and vehicular simulation and emulation, where simulations platforms (e.g., ns-2, ns-3) could apply their functionalities over enhanced datasets, being able to work properly, looking for a good performance, in terms of processing time and result reliability.

As stated before, security is a central aspect to be taken into account, in order to enhance the architecture reliability and the processing control. To provide guarantees at input stages, an optimal solution could be represented by the introduction of an authorization module, which tokenizes incoming data adopting an asymmetric security paradigm, to sure that raw data providers are authorized to provide information.

Looking for a reliable behavior at the output stage, a good solution could be reached by introducing an Accounting/Authentication/Authorization (AAA) module, which manages and controls the acceptance of consumers, providing some cryptographic functionalities, to check security-level of each entity.

CONCLUSION

In this paper, the authors presented a novel Cloud Graph-based architecture for efficient management of Big Stream Real-time applications in IoT scenarios. After describing the main requirements, in terms of reduced latency between the data creation instant and the instant at which processed data can be delivered to a consumer, the new Big Stream paradigm has been introduced highlighting its differences with respect to the Big Data paradigm. The main components of the designed listener-based architecture are the following: the Acquisition Module, the Normalization Module, the Graph Framework, and the 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 generates several benefits, such as:

- **Decreased Latency:** The push-based approach guarantees that no delays due to polling and batch processing are introduced;
- **Fine-Grained Self-Configuration:** Listeners can dynamically "plug" to streams interest data;
- **Optimal Resource Allocation:** Processing units that have no listeners can be switched off, while those with many listeners can be replicated, thus leading to cost-effectiveness from the Cloud service perspective.

KEY TERMS AND DEFINITIONS

Big Data: Paradigms and technologies to handle massive volume of structured and unstructured data which is so large that it's impossible to process using traditional database and software techniques.

Big Stream: Paradigms and technologies to handle with real-time and low-latency requirements, the massive volume of data generated with very high frequency by a huge number of different data sources.

Exchange: In a generic network or graph topology is a component that receives messages from producers and dispatches them to one or more output queue depending on specific routing rules.

Graph: A mathematical model to represent a set of entities connected to each other. The complete topology of a graph is identified by a list of vertices (or nodes) and a list of edges (or links) between two vertices.

Internet of Things: The interconnection of billions of heterogeneous devices called "Smart Objects" through the Internet infrastructure. Smart Objects are typically constrained devices like sensors or actuator and are deployed to collect data and to build useful services to consumers.

Listener: In an event driven system, a process or a component which is able to listen for and to handle it a particular event.

Real-Time System: System required to guarantee responses with hard and strict time constraints.

Smart Object: A device with communication capabilities deployed in IoT systems. Generally has constrained capabilities and is equipped with sensor or actuator to collect data or act in the environment.

REFERENCES

Apache. (n.d.-a). *Storm*. Retrieved from https://storm.incubator.apache.org/

Apache. (n.d.-b). *ActiveMQ*. Retrieved from http://activemq.apache.org/

Assunção, M. D., Calheiros, R. N., Bianchi, S., Netto, M. A., & Buyya, R. (2014). Big data computing and Clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*.

Belli, L., Cirani, S., Davoli, L., Melegari, L., Mónton, M., & Picone, M. (2015). An Open-Source Cloud Architecture for Big Stream IoT Applications. In I. Podnar Žarko, K. Pripužić, & M. Serrano (Eds.), Interoperability and Open-Source Solutions for the Internet of Things (Vol. 9001, pp. 73-88). Lecture Notes in Computer Science (LNCS). Springer International Publishing. Retrieved from DOI: doi:10.1007/978-3-319-16546-2_7

Belli, L., Cirani, S., Ferrari, G., Melegari, L., & Picone, M. (2014). *A Graph-based Cloud architecture for Big Stream real-time applications in the Internet of Things*. In *2nd International Workshop on Cloud for IoT (CLIoT 2014)*, Manchester, United Kingdom, September 2014.

Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012). *Fog Computing and its role in the internet of things.* In *Proceedings of the First Edition of the ACM Workshop on Mobile Cloud Computing* (p. 13-16). New York, NY, USA. Retrieved from http://doi.acm.org/10.1145/2342509.2342513

Cirani, S., Davoli, L., Picone, M., & Veltri, L. (2014, July). *Performance Evaluation of a SIP-based Constrained Peer-to-Peer Overlay*. In 2014 International Conference on High Performance Computing Simulation (HPCS), (p. 432-435). Retrieved from doi:10.1109/HPCSim.2014.6903717

Cirani, S., Picone, M., & Veltri, L. (2013). CoSIP: A Constrained Session Initiation Protocol for the Internet of Things. In C. Canal & M. Villari (Eds.), Advances in Service-Oriented and Cloud Computing (Vol. 393, pp. 13–24). Springer Berlin Heidelberg. Retrieved from doi:10.1007/978-3-642-45364-9_2

Cirani, S., Picone, M., & Veltri, L. (2014). *A Session Initiation Protocol for the Internet of Things.* Scalable Computing: Practice and Experience, 14 (4), 249-263. Retrieved from doi:10.12694/scpe.v14i4.931

Cirani, S., Picone, M., & Veltri, L. (2015). mjCoAP: An Open-Source Lightweight Java CoAP Library for Internet of Things Applications. In: Interoperability and Open-Source Solutions for the Internet of Things. LNCS, vol. 9001, Retrieved from DOI:, Springer International Publishing Switzerland. doi:10.1007/978-3-319-16546-2_10

Deering, S., & Hinden, R. (1998, December). *Internet Protocol, version 6 (IPv6) Specification (No. 2460)*. RFC 2460 (Draft Standard). IETF. Retrieved from http://www.ietf.org/rfc/rfc2460.txt. (Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112)

Dunkels, A., Gronvall, B., & Voigt, T. (2004). *Contiki - A Lightweight and flexible Operating System for tiny networked sensors.* Local Computer Networks, 2004. 29th Annual IEEE International Conference on (pp. 455-462). IEEE.

Emerging Technology, I. B. M. (2013). *Node-RED*. Retrieved from http://nodered.org/

European Community's 7th Framework Programme. (2007). *OpenIoT - Open Source Cloud solution for the Internet of Things.* Retrieved from http://openiot.eu/. Retrieved from http://openiot.eu/

European Community's 7th Framework Programme. (2008-2010). *SENSEI Project*. Retrieved from http://www.ict-sensei.org/

European Community's 7th Framework Programme. (2011-2014). *CALIPSO - Connect All IP-based Smart Objects.* Retrieved from http://www.ict-calipso.eu/

European Community's 7th Framework Programme. (2011). *FI-Ware Project*. Retrieved from http://www.fi-ware.org/

European Community's 7th Framework Programme. (2012-2015). *Internet of Things - Architecture (IoT - A)*. Retrieved from http://www.iot-a.eu/

Evans, D. (2011). *The Internet of Things: How the next evolution of the internet is changing everything.* CISCO white paper, 1.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol – http/1.1.* United States: RFC Editor.

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Unpublished doctoral dissertation).

Fielding, R. T., & Kaiser, G. (1997). The Apache HTTP server project. *IEEE Internet Computing*, *1*(4), 88–90. Retrieved from doi:10.1109/4236.612229

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, *29*(7), 1645–1660. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167739X13000241 doi:10.1016/j.future.2013.01.010

Hammer-Lahav, E. (2010). *RFC 5849: The OAuth 1.0 protocol. Internet Engineering Task Force*. IETF.

Hardt, D. (2012). RFC 6749: *The OAuth 2.0 authorization framework-revision*.

Hohpe, G., & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Isaacson, C. (2009). *Software pipelines and SOA: Releasing the power of multi-core processing* (1st ed.). Addison-Wesley Professional.

Leavitt, N. (2013). Storage challenge: Where will all that big data go? *Computer*, *46*(9), 22–25. Retrieved from doi:10.1109/MC.2013.326

Locke, D. (2010). *MQ Telemetry Transport (MQTT) v3. 1 protocol specification.* IBM developer Works Technical Library], Retrieved from https://www.ibm.com/developerworks/webservices/library/ws-mqtt/

Marganiec, S. R., Tilly, M., & Janicke, H. (2014, June). *Low-Latency Service Data Aggregation Using Policy Obligations. In Web Services (ICWS)*, 2014 IEEE International Conference on (pp. 526-533). IEEE.

McAfee, A., & Brynjolfsson, E. (2012). Big data: The management revolution. *Harvard Business Review*, (90): 60–66. PMID:23074865

Mell, P., & Grance, T. (2011). The NIST definition of Cloud Computing. *National Institute of Standards and Technology*, *53*(6), 50.

Milojičić, D., Llorente, I. M., & Montero, R. S. (2011). *OpenNebula: A Cloud management tool*. IEEE Internet Computing, 15(2), 0011-14.

Mosquitto. (n.d.). *An Open Source MQTT Broker*. Retrieved from http://mosquitto.org/

MySQL. (n.d.). Retrieved from http://www.mysql.com/

Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010). S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data mining workshops (ICDMW)* (pp. 170–177). doi:10.1109/ICDMW.2010.172

Postel, J. (Ed.). (1981, September). RFC 791 Internet Protocol - DARPA Internet program, protocol specification [Computer software manual]. Retrieved from http://tools.ietf.org/html/rfc791

Rabbit, M. Q. (n.d.). Retrieved from http://www.rabbitmq.com/

Rackspace, N. A. S. A. (n.d.). *OpenStack Cloud Software - Open source software for building private and public Clouds.* Retrieved from https://www.openstack.org/

Reese, W. (2008). *NGINX: the high-performance web server and reverse proxy*. Linux Journal, 2008 (173), 2.

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., & Schooler, E. (2003). *RFC 3261: SIP: Session Initiation Protocol.* IETF, Tech. Rep., 2002. Retrieved from http://www.ietf.org/rfc/rfc3261.txt

Saint-Andre, P. (2004, October). *Extensible messaging and presence protocol (XMPP): Instant messaging and presence*. Internet RFC 3921.

Shelby, Z., Hartke, K., Bormann, C., & Frank, B. (2014). *RFC 7252: The Constrained Application Protocol (CoAP)*. Internet Engineering Task Force.

Stanoevska-Slabeva, K., Wozniak, T., & Ristol, S. (2009). *Grid and Cloud Computing: a business perspective on technology and applications*. Springer Science & Business Media.

Tilly, M., & Reiff-Marganiec, S. (2011, March). *Matching customer requests to service offerings in real-time.* In Proceedings of the 2011 ACM Symposium on Applied Computing (pp. 456-461). ACM. Retrieved from doi:10.1145/1982185.1982285

Vilajosana, I., Llosa, J., Martinez, B., Domingo-Prieto, M., Angles, A., & Vilajosana, X. (2013). Bootstrapping smart cities through a self-sustainable model based on big data flows. Communications Magazine, IEEE, 51(6).

Vinoski, S. (2006, November). *Advanced message queuing protocol*. IEEE Internet Computing, 10 (6), 87–89. Retrieved from .10.1109/MIC.2006.116

Zaslavsky, A., Perera, C., & Georgakopoulos, D. (2013). *Sensing as a service and big data.* Retrieved from http://arxiv.org/abs/1301.0159

Laura Belli received the Dr. Ing. degree in Computer Engineering from the University of Parma, Parma, Italy in 2011. Currently she is a PhD student in Information Technologies at the same University.

Simone Cirani is a Postdoctoral Research Associate at the Department of Information Engineering of the University of Parma, Italy. He received his Dr. Ing. (Laurea) degree in Computer Science "cum laude" from the University of Parma, Italy, in 2007. In 2011, he received his PhD in Information Technologies at the Department of Information Engineering of the same university. His research interests are Internet of Things, Peer-to-peer networks, Network Security, Pervasive Computing, and Mobile Application Development.

Luca Davoli received the BSc and MSc degrees in Computer Science Engineering from the University of Parma, Parma, Italy, in 2011 and 2013, respectively, and is currently working toward the PhD degree in Information Engineering at the University of Parma. His research interests include Peer-to-Peer networks, security and protocols for the Internet of Things, and Pervasive Computing.

*Gianluigi Ferrari received his Laurea and PhD degrees from the University of Parma, Parma, Italy, in 1998 and 2002, respectively. Since 2002, he has been with the University of Parma, where he currently is an Associate Professor of Telecommunications. He was a Visiting Researcher at USC (Los Angeles, CA, USA, 2000-2001), CMU (Pittsburgh, PA, USA, 2002-2004), KMITL (Bangkok, Thailand, 2007), and ULB (Brussels, Belgium, 2010). Since 2006, he has been the Coordinator of the Wireless Ad-hoc and Sensor Networks Lab (*http://wasnlab.tlc.unipr.it/*) at the Department of Information Engineering. As of today, he has published/accepted more than 200 papers in leading international journals and conferences, 22 book chapters, 9 patents, and 8 books. He edited the book Sensor Networks: Where Theory Meets Practice (Springer: 2010). His research interests include wireless ad hoc and sensor networking, adaptive digital signal processing, and communication theory. He participates in several research projects funded by public and private bodies (recovered funds over 1.9 M€). Prof. Ferrari is corecipient of a best student paper award at IWWAN'06; a best paper award at EMERGING'10; the first Body Sensor Network (BSN) contest winner award (as member of the WASNLab team) held in conjunction with BSN 2011; an award for the outstanding technical contributions at ITST-2011; the best paper award at SENSORNETS 2012; the best paper award at EvoCOMNET 2013; the Best Runner-up Paper Award at BSN 2014; the Best Conference Paper Award at SoftCOM'14 (Symposium on "RFID Technologies and Internet of Things"). He acts as a frequent reviewer for many international journals and conferences, as well as TPC for many international conferences. He currently serves on the editorial boards of several international journals. He was a Guest Editor of the 2010 EURASIP JWCN Special Issue on "Dynamic Spectrum Access: From the Concept to the Implementation" and of the 2014 Hindawi IJDSN Special Issue on "Advanced Applications of Wireless Sensor Network Using Sensor Cloud Infrastructure." He is a Guest Editor of the 2015 Hindawi IJDSN Special Issue on "Wireless Sensor Networks for Structural Health Monitoring." He is an IEEE Senior Member.*

Màrius Montón obtained a PhD in Computer Science from the Universitat Autonoma de Barcelona (UAB) and a Masters in Microelectronics and Electronic Systems and Computer Engineer degree from the same university. Currently he is working as Head of Firmware group in WorldSensing. He was working several years as engineer at Cephis-UAB. He also worked as associate professor at the university. In addition,he is working as ca onsultor for GreenSocs developing TLM-2.0 based solutions for ESL businesses.

Marco Picone currently is a Postdoctoral Research Associate at the University of Parma. He received the Dr. Ing. degree (Master) in Computer Engineering "cum laude"in 2008 and the PhD degree in Information Technologies in 2012, both from the same university. Between January 2011 and June 2011, he was a research visitor in the Network and Operating Systems group at the Computer Laboratory, University of Cambridge. His supervisor in Cambridge was Dr. Cecilia Mascolo and the research activities were focused on mobile based sensing systems and sensor networks interaction. His research activity focuses on Distributed and Peer-to-Peer Systems, with particular interest for solutions that involve mobile devices. Application fields include: Neighbor position discovery in peer-to-peer networks, Vehicle-to-Vehicle and Vehicle-to-Infrastructure communications, P2P approach for Inter-vehicular networks, Vehicular networks simulation and mobility model, Mobile based sensing system and Vertical Handover Algorithms & Applications. He is a lecturer for the class of Mobile Application Development (Programmazione di Sistemi Mobili) at the University of Parma, for the 2013/2014 school year.