Accepted Manuscript

Interoperable communication framework for bridging RESTful and topic-based communication in IoT

Ahmed E. Khaled, Sumi Helal

PII: S0167-739X(17)31738-7
DOI: https://doi.org/10.1016/j.future.2017.12.042
Reference: FUTURE 3880

To appear in: *Future Generation Computer Systems*

Received date : 31 July 2017
Revised date : 13 November 2017
Accepted date : 24 December 2017

Please cite this article as: A.E. Khaled, S. Helal, Interoperable communication framework for bridging RESTful and topic-based communication in IoT, *Future Generation Computer Systems* (2018), https://doi.org/10.1016/j.future.2017.12.042

Interoperable Communication Framework for Bridging RESTful and Topic-based Communication in IoT

Ahmed E. Khaled
Mobile & Pervasive Computing Lab
University of Florida
Gainesville, FL 32611, USA
aekhaled@cise.ufl.edu

Sumi Helal
School of Computing and Communication
Lancaster University
Lancaster, LA1 4WA, UK
s.helal@lancaster.ac.uk

ABSTRACT

The promise of the Internet of Things (IoT) and the many visions of unprecedented and transforming IoT applications are challenged by the realities of a highly fragmented ecosystem of devices, standards and industries. Systems research in IoT is shifting priorities to explore explicit "thing architectures" that promote and enable the friction-free interactions of things despite such fragmentations. In this paper, we focus on overcoming light-weight communication protocol fragmentation. We introduce the Atlas IoT communication framework which enables interactions among things that speak similar or different communication protocols. The framework tools up Atlas things with protocol translator "attachments" that could be either hosted on board the Atlas thing platform, or in the cloud. The translator enables the seamless communication between heterogeneous things through a set of well-defined interfaces. The proposed framework supports seamless communication among the widely adopted Constrained Application Protocol (CoAP), Representational State Transfer (REST) over Hypertext Transfer protocol HTTP, and the Message Queue Telemetry Transport protocol (MQTT). Our framework is carefully designed to facilitate interoperability among heterogeneously communicating things without taxing the performance of things that are homogenously communicating. The framework itself utilizes the topic concept and uses a meta-topic hierarchy to map out and guide the translations. We present the details of the Atlas IoT communication framework and give a detailed benchmarking study to measure the energy consumption and code footprint characteristics of the different aspects of the framework on real hardware platforms. In addition to basic characterizations, we compare our framework to the Eclipse Ponte framework and show how our framework is advantageous in energy consumption and how it is unique in that it does not tangibly penalize the homogeneous communication case.

Keywords: Internet of Things interoperability, Translator, Topic, Atlas thing architecture, IoT-DDL, CoAP, MQTT

1. INTRODUCTION

Over the last decade, the Internet of Things (IoT) has attracted tremendous community and industry interest for its potential to bring more informative and interactive flavors to our lives [16][17]. Current advancements in IoT have shifted the primary focus of the research from the pervasive presence of smart objects and cyber elements in a smart space and the ad hoc integration of objects in smart systems, to IoT infrastructure and architecture, security, sensing technologies, communication protocols, and many other aspects of IoT. Things—the basic building blocks of IoT—empowered with evolving communication technologies and computing capabilities, utilize their communication capabilities to establish a wide range of interactions and interconnections with other things in the smart space [15][16]. To enable such interactions, several competitive IoT application-layer communication protocols (e.g., eXtensible Messaging and Presence Protocol (XMPP) [14][15], HTTP REST [27][28][29], MQTT [25][26], and CoAP [11][12][13]) have been developed to satisfy the properties of constrained ecosystems such as IoT. Each protocol is designed for a specific set of application requirements and aspects of IoT communications, such as low-power operation, lightweight headers, semantic intent-oriented

messaging, service discovery and orientation, bootstrapping, and statelessness. Such communication protocols blur the line between messaging mechanisms and semantic-based computation.

However, the wide heterogeneity in types and communication capabilities makes the participation of things in a smart space a significant challenge that requires considerable effort and human intervention, and limits programming opportunities that may involve things that utilize different communication languages to cooperate and interact. Such a highly fragmented market between many communication protocols that share no horizontal connectivity endangers the adoption of programming opportunities and restricts service and application development that can benefit from heterogeneous things. Such vertical slices create a set of isolated islands that oxymoron IoT vision, where a smart thing can smoothly communicate and interact with any other smart thing in the ecosystem. Interoperability—among current IoT research interests—has the potential to homogenize things that speak different communication languages so that they can interact and cooperate.

A few proposed solutions that target IoT interoperability [3][6][7] require all communications to be routed through a centralized server. Such a server speaks all possible IoT communication protocols and is assumed to hold the resources of the smart space. However, such an assumption ignores the distributed and dynamic nature of IoT, where the things themselves are the resource holders. This approach also imposes inefficiencies in communication in some cases where only homogenous things (e.g., CoAP- or REST-speaking things) exist in the smart space with no requirement for a centralized server to handle the communication channels and resources. In this paper, we propose the Atlas communication framework for an interoperable distributed IoT ecosystem. The proposed framework enables seamless interaction between homogenous things that speak similar languages as well as heterogeneous things that speak different communication languages in a smart space with minimal human intervention. In addition, the framework does not additionally tax performance of the homogenous communicating things.

We base our approach on the Atlas thing architecture and the IoT Device Description Language (IoT-DDL) proposed in [30]. The architecture takes advantage of the thing's OS services to provide new layers and functionalities that introduce the novel capabilities a thing needs to engage in ad hoc interactions and interconnections, as well as IoT scenarios and applications. The architecture fully utilizes the specifications of IoT-DDL, which is a machine- and human-readable XML-based descriptive language that describes a thing's identity, components, and services. The thing can self-discover its identity and components, and then formulate APIs of the services it wishes to offer and get involved in information-based and action-based interactions with other things in the smart space. Information-based interactions enable the thing to announce its identity, capabilities, and APIs, while action-based interactions include the applications that target the thing's capabilities and services in terms of API calls. Such new services focus on descriptive and semantic aspects of things to better enable thing engagement, interaction, and programmability into an IoT. The Atlas IoT platform layer that provides these new services represents the logical layer of the architecture and is composed of three sublayers: 1) the DDL sublayer, which is responsible for configuring the Atlas architecture modules using the specifications of the IoT-DDL configuration file; 2) the tweeting sublayer, which tools the thing to uniquely define itself in the smart space, in addition to discovering thing mates; and 3) the interface sublayer, which holds the different communication protocols (e.g., MQTT, CoAP, HTTP REST) that allow the thing to engage and interact. As currently designed and implemented, the interface sublayer—the focus of this paper—supports only homogeneous communication.

Our proposed Atlas communication framework is composed of three parts: 1) Atlas topics, 2) Atlas protocol translator, and 3) an extended version of the interface sublayer of the Atlas architecture to enable the seamless and automatic interoperability between the things that utilize similar and different communication protocols. We use the idea of the MQTT topic (as a channel where the publisher can post data without prior knowledge of the clients subscribed to the same topic) to create a set of common "Atlas topics" through which things can communicate and interact. Such communication channels enable both multicast communication for the thing to discover other

things and distributed services in the smart space as well as peer-to-peer communication for unicast interactions. CoAP and REST also utilize the MQTT topic through Uniform Resource Locators (URLs) as a path to the resource [12][27]. Our proposed Atlas communication framework utilizes the minimum delimiter among the various communication protocols to provide homogenous communication links for the things in a smart space. The framework promotes topics as a suitable mechanism to link topic-oriented communication protocols (e.g., MQTT) with URL-oriented protocols (e.g., CoAP and HTTP REST). It uses CoAP, HTTP REST, and MQTT IoT communication standards for application-layer network protocol interoperability.

The framework also utilizes the idea of expanding the thing's capabilities through attachments [30]. A thing attachment is linked to the thing and tooled with additional functionalities that provide further representations (e.g., thing virtualization) and services (e.g., log server, database, or dashboard) that may require additional resources (e.g., memory storage or processing power). The Atlas protocol translator, as the third part of the framework, is a thing attachment that enables seamless communication and interaction with heterogeneous things through a set of well-defined interfaces.

The paper is organized as follows. Section 2 highlights related work, followed by a description of the proposed Atlas heterogeneous communication framework in section 3. In section 4, we present our implementation and a benchmarking study in which we measure and assess code footprint and energy characteristics of the framework on real hardware platforms. Finally, a discussion of the result and conclusions are presented in section 5.

2. BACKGROUND AND RELATED WORK

In this section, we provide a quick background on the emerging IoT lightweight communication protocols utilized in our proposed communication framework. We then highlight proposed frameworks and approaches in the literature that target IoT interoperability on application-layer communication protocols.

2.1 Overview of CoAP, MQTT, and HTTP REST

The Constrained Application Protocol (CoAP) [11][12][13] is a specialized web transfer protocol for use with constrained nodes and constrained networks in a wireless sensor network (WSN) and IoT. CoAP is a client/server computing model that provides a request/report paradigm model over UDP. To compensate for UDP's unreliability, CoAP defines a retransmission mechanism and provides a resource discovery mechanism with a resource description. CoAP provides URI and REST methods such as GET, POST, PUT, and DELETE to access the various resources. The Atlas architecture uses CoAP protocol support for multicasting, which allows things to broadcast tweets for all listening things. We extend the imported library [23] with Unix multicast sockets to enable the CoAP multicast feature.

The Hypertext Transfer Protocol (HTTP) [27][28] is a reliable TCP application protocol for distributed and collaborative ecosystems, and is believed to be the foundation of data communication on the World Wide Web. HTTP is also a client/server computing model that provides URI and REST methods such as GET, POST, PUT, and DELETE to access the different resources hosted by the server.

The Message Queue Telemetry Transport (MQTT) [25][26] uses a publish/subscribe architecture on top of the TCP/IP protocol in contrast to the HTTP request/response paradigm. An MQTT broker is the central communication point in charge of dispatching all messages between senders and receivers. MQTT's publish/subscribe is an event-driven paradigm that allows clients to publish messages with topics and to subscribe to topics. The topic is routing information for the broker; a client subscribes to a topic and the broker delivers all messages with the matching topic to that client. Topics can be organized into a name space of any hierarchical structure. MQTT enables highly scalable and flexible solutions, where clients only communicate over the topic of interest without having to know each other. Each MQTT client has a permanently open TCP connection to the broker. If this connection is interrupted, the MQTT broker can buffer all messages and send them to the client when it is back online.

2.2 State of the Art

C. Lee et al. [2] proposed a hybrid IoT communication framework based on a software-defined network (SDN) that intercepts all packets from CoAP and MQTT, and vice versa. The proposed framework defines URL rules to specify the topic and differentiate homogenous (e.g., from CoAP client to CoAP client) from heterogeneous packets (e.g., from MQTT client to CoAP client). If packets belong to the same protocols, they operate as the original communication scenarios and the SDN just ignores this traffic. When the traffic of heterogeneous protocols is intercepted, the SDN switch delivers these packets to the SDN controller, and the SDN controller is responsible for redirecting the packets to the cross proxy for protocol translation.

P. Desai et al. [3] proposed a gateway and semantic web-enabled IoT that provide translation between messaging protocols such as XMPP, CoAP, and MQTT using a multiprotocol proxy with a separate interface for each protocol. The proposed gateway, which is located between the physical-level sensors and the cloud-based services, holds a centralized topic router that maps information from the different communication protocols. However, the authors did not provide an implementation for the proposed gateway to show its feasibility.

A. AL-Fuqaha et al. [4] developed a generic IoT protocol by enhancing the baseline MQTT protocol and allowing it to support rich quality-of-service (QoS) features by exploiting a mix of IP multicasting, intelligent broker queuing management, and traffic analytics techniques. The hybrid architecture allows the protocol to seamlessly utilize direct and brokerless multicast communication while utilizing the broker for machine-to-server (M2S) communications. This hybrid architecture would allow MQTT to extend its role in the IoT to handle M2M communications and would allow for multiple MQTT brokers to cooperate in delivering better QoS and reliability capabilities. The proposed new MQTT lets multiple brokers receive multicast communications from IoT devices, enabling failure recovery in the event of broker failures and allowing brokers to move subscribers to other brokers in support of QoS that is beyond the naïve QoS features currently offered by MQTT. Although the authors do not explicitly address the heterogeneity between the different communication protocols, the idea of brokerless multicast opens the door for other protocols to engage with MQTT.

P. Bellavista et al. [1] proposed a gateway-oriented architecture where gateways jointly exploit MQTT and CoAP to achieve highly scalable IoT device management through dynamic hierarchical tree organizations. The proposed gateway extends the Kura Eclipse framework, which is based on the interworking of MQTT that is already integrated into the Kura framework [1], with CoAP coordination functionality as an added protocol. The extended Kura framework offers improved scalability and reduced latency for communication/coordination among wide-scale sets of geographically distributed IoT devices interworking via gateways for efficient resource lookup. MQTT plays a central role and is strongly exploited by the Kura framework, but exhibits nonnegligible limitations in terms of scalability, e.g., inefficient usage of TCP connections toward the broker when growing the number of IoT devices in a gateway locality.

H. Derhamy et al. [5] investigated error-handling challenges for a multiprotocol SOA-based translator, taking MQTT and CoAP as a proof-of-concept implementation. In a single protocol system, errors are propagated according to protocol specification. In the case of multiprotocol systems, error handling becomes more complex. In designing an SOA-based translator, error handling and considerations become critical to robust communication. An error in one protocol must be translated to be understood by other protocols. While an SOA-based translator must also address other aspects such as QoS, control messaging, security, and semantic translation, these issues are considered future work so are not addressed in this paper. Experimental results show that multiprotocol error handling is possible and, furthermore, a number of areas needing more investigation have been identified.

Ponte [6] is an Eclipse IoT project that offers uniform open APIs to let developers create applications supporting CoAP, MQTT, and HTTP REST communication protocols through an independent module for each protocol. The Ponte gateway can reside in a server or edge where clients with different communication protocols can communicate. Ponte provides a centralized solution for interoperability, where the smart space resources

reside in the cloud and can be accessed from different clients. Data collected from the three modules is stored in the SQL or NoSQL database; therefore, no matter which protocol clients utilize for communication, they can access the same resources.

EMQTT [7] is a massively scalable MQTT broker for IoT and mobile applications licensed under Apache. An EMQTT server implements the MQTT protocol and supports a set of plugins that allows other communication protocols to coexist in parallel (e.g. MQTT-SN, CoAP, and Web sockets).

The aforementioned approaches introduce a centralized server that resides on either a cloud platform or an edge, with all communications routed through it. The server implements all possible communication protocols and is assumed to hold the smart space resources. However, such an assumption ignores the distributed nature of IoT, where resources are distributed on the things. At the same time, such a centralized approach reduces flexibility and imposes inefficiencies in both communication links and bandwidth usage in cases where only homogenous things existed in the smart space with no requirement for a centralized server to handle the communication channels (e.g., CoAP- or HTTP REST-speaking things). Our proposed Atlas communication framework enables seamless interaction between things that speak different communication languages and does not tax performance when homogenous things are communicating. In the next section, we introduce our proposed Atlas communication framework and describe the different scenarios where the framework handles communication of smart spaces with homogenous and heterogeneous things.

3. ATLAS IoT COMMUNICATION FRAMEWORK

As noted earlier, we base our approach on the Atlas thing architecture and the IoT Device Description Language (IoT-DDL) proposed in [30]. The Atlas architecture takes advantage of a thing's OS services to provide the new capabilities a thing needs to engage in ad hoc interactions and interconnections, as well as IoT applications. The architecture fully utilizes the specifications of IoT-DDL, which is a machine- and human-readable XML-based descriptive language that describes a thing's identity, components, and services.

The Atlas IoT platform layer of the architecture (which is built on top of the thing's OS) provides new services that focus on descriptive and semantic aspects of things to better enable thing engagement, interaction, and IoT programmability. The platform tools the thing to self-discover its identity, components, and services, and to formulate APIs of its services. The thing also announces its identity, capabilities, and APIs to other things in the smart space through information-based interactions, while the action-based interactions include the applications that target a thing's capabilities and services in terms of API calls. The Atlas IoT platform layer is further divided into the DDL sublayer, which configures the architecture modules according to the specifications of the uploaded IoT-DDL configuration file; the tweeting sublayer, which tools the thing to uniquely define itself in the smart space and to discover thing mates; and the interface sublayer, which holds the communication protocols the thing uses to engage and interact. Atlas IoT-DDL builder is a web service tool that allows a thing's creator (e.g., the original equipment manufacturer (OEM)) or owner to create, update, or upload an IoT-DDL to a thing. The OEM of a thing could be the source of the IoT-DDL; a developer who utilizes space things' services and resources might also be the source. Such flexibility facilitates further adoption of IoT-DDL with changes, and supports thing innovation, in which makers or hobbyists may be assembling new things not established by OEM. We developed an initial version of the web tool [34] that enables space users to develop an IoT-DDL that reflects, the thing's metadata, attachments as well as the resources and services. The IoT-DDL manager of the DDL sublayer parses the different parameters of the uploaded IoT-DDL's different sections required for the operation of the different modules of the Atlas architecture. The current implantation of the Atlas thing architecture [30] utilizes the lightweight OMA-LwM2M device management standard to manage the different aspects of IoT-DDL. The architecture translates the different sections and subsections of IoT-DDL into a set of dynamic objects to represent the different entities, services, resources, and attachments of an Atlas thing. The object creation and management occur on demand when IoT-DDL is uploaded to the thing during lifetime updates. At the same time, authorized

lifetime management and updates from the management server trigger the device manager's object engine module to maintain the corresponding objects and enables the authorized dynamic updates to the IoT-DDL parameters and attributes during the lifetime of the thing.

The interface sublayer, the focus of this paper, currently supports only homogenous communication. The framework also utilizes the idea of expanding thing capabilities through attachments [30]. A thing attachment is an optional and additional accessory linked to the thing to tool it with further functionalities. Such functionalities can extend thing to provide additional services (e.g., log server, data store, software adaptor, specialized user interface, or dashboards) that may require additional resources and assets (e.g., software, manifests, memory storage or processing power). Such thing attachments could either reside on a more powerful platform (e.g., local cloud or edge) on the same smart space network with the things, or on a remotely accessible network. The owner of an attachment (e.g., a consumer electronics thing vendor) provides a well-defined interface to the offered function, where the function is either: 1) an online resource that the thing can access remotely (e.g., off-thing translator – as will be shown in section 3.2) or 2) a programmable object that the thing can download and dynamically integrate and bind with the thing (outside the scope of this paper).

Our proposed Atlas communication framework is composed of three parts: 1) the interface sublayer of the Atlas thing architecture extended to enable seamless interoperability between things that use both similar and different communication protocols; 2) Atlas topics empowered by the idea of MQTT topics to create a set of common channels for things to discover other things and distributed services in the smart space, as well as enable peer-to-peer interactions; and 3) the Atlas protocol translator as a thing attachment, which can be either off-thing, residing on a cloud platform or edge, or an on-thing service running within the Atlas thing architecture. The translator provides protocol translation service to the thing through a set of well-defined interfaces, allowing the thing to communicate and interact with heterogeneous things. The proposed Atlas communication framework utilizes the widely adopted CoAP, HTTP REST, and MQTT IoT communication standards for application-layer network protocol interoperability. Each of these parts are described in detail in the following sections.

An example application, which we implemented, that can benefit from our proposed framework is an infotainment application consisting of a Sports mobile app as a thing in the IoT [35], and a DVR recorder thing that can record games. The DVR and the mobile app, under our architecture, tweet and learn they can form an IoT application where the DVR recorder may offer recording service to the Sports app (details in [35]). This application scenario will benefit from our framework here where the vendor of the DVR may decide to choose one communication protocol or another, yet the interaction will be enabled under our IoT communication framework.

3.1 Atlas IoT Topic Hierarchy

The framework features the idea of MQTT topics to create a set of common channels ("Atlas topics") that things can use to communicate and interact. Such communication channels enable both the multicast communication paradigm that tools the thing to discover other things and distributed services in the smart space as well as the peer-to-peer communication paradigm for unicast interactions. In this paper, we use "Atlas topic" to refer to the channels on which HTTP REST-, CoAP- and MQTT-speaking Atlas things can communicate, advertise, and interact.

The framework extends the Atlas thing IoT-DDL specifications proposed in [30], which describe the different aspects and dimensions of the thing, to include the Smart Space ID (SSID), Atlas Thing ID (ATID), and the Smart Space Broker (SSB) attributes. SSID is an identifier for the smart space defined by the space owner (e.g., owner name, cellphone number, and zip code of the space) that ensures the uniqueness of the identity between the different smart spaces. ATID is an identifier for the thing defined by the owner or IoT-DDL developer that ensures the uniqueness of the thing for the same smart space. SSB defines the access interface to a broker provided by the space owner or offered by a third party (e.g., MQTT-HiveHQ [9] or Eclipse MQTT sandbox [8]). Learning the SSB is an essential step to establishing an initial shared communication medium through which

smart space things speaking different communication languages can sense their mutual presence and communicate. On powering up an Atlas thing, as illustrated in Fig. 1, the thing connects to the default broker that is indicated by the SSB of IoT-DDL and subscribes to the Atlas topics. MQTT-speaking things connect directly to the broker, while HTTP REST- and CoAP-speaking thing utilize their own Atlas protocol translator, as will be illustrated in section 3.2, to connect and subscribe to this broker. The non-MQTT-speaking things in the scope of this paper are things that speak CoAP or HTTP REST.

To keep the communication channels secure and private, or to utilize a required broker, the framework lets the space owner dynamically assign a new broker to the smart space through one of the shared Atlas topics. The new broker can be an Atlas thing with high capabilities (e.g., memory or power source) that offers broker service to the smart space. The Atlas things in the same smart space must have the same SSID and SSB values to ensure that they can communicate and interact with the same broker using the same channels, as will be discussed later.

To ensure uniqueness, the framework uses the Atlas_(SSID) topic as the root of the hierarchy under which all Atlas things communication and interactions take place. The root topic (illustrated in Table 1) is further divided into network, multicast, and unicast subtopics. Such subtopics enable things in a smart space to sense the network for the presence of MQTT clients and any required broker that replaces the default broker. The subtopics also allow things to multicast packets and perform peer-to-peer interactions. The three subtopics are described next.

Table 1: Atlas IoT topic hierarchy

/Atlas_(SSID)	/Network	/MQTT_Client
		/Private_Broker
	/Multicast	/Thing_Identity
		/Thing_API
	/Unicast	/Interaction_(ATID)

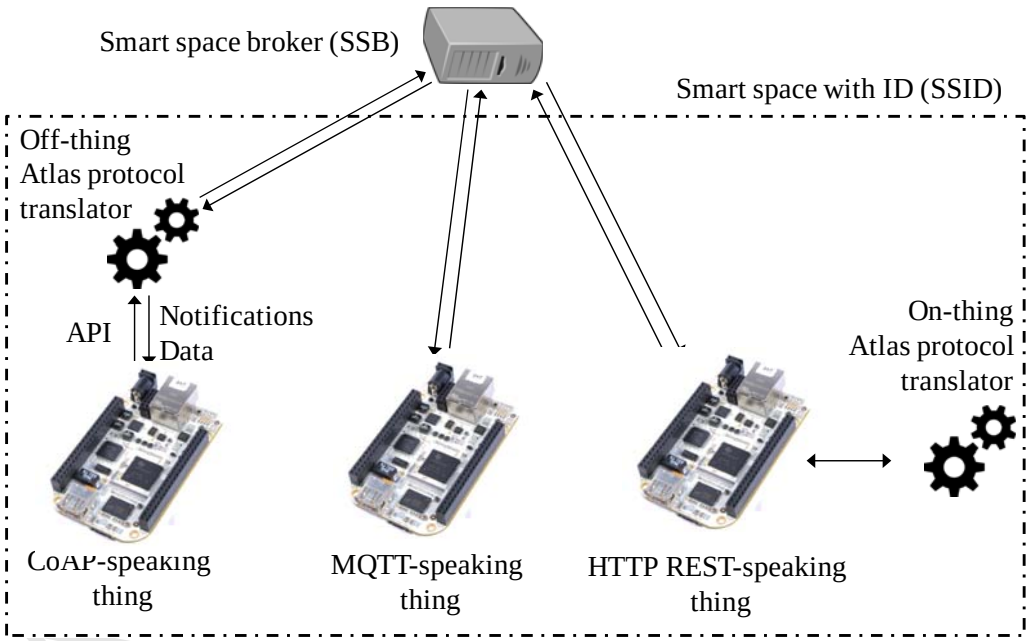


Figure 1: Smart space broker and HTTP REST-, CoAP- and MQTT-speaking Atlas things.

3.1.1 Network Topic

The network topic of the Atlas topic hierarchy keeps the thing's attention on network updates through two subtopics: MQTT_Client and Private_Broker.

- The MQTT_Client topic indicates the existence of at least one MQTT-speaking Atlas thing in the smart space. After powering up, an MQTT-speaking thing publishes "true" to the MQTT_Client to mark the existence of an MQTT thing in the smart space. At the same time, non-MQTT-speaking things subscribed to the topic hierarchy detect the presence of MQTT client(s) in their smart space and route all future unicasting with intended MQTT peers through the broker, which requires an Atlas protocol translator, as will be shown in section 3.1.3. Similarly, multicasting of non-MQTT-speaking things is also routed through the broker, as will be detailed in the next section. The content of the MQTT_Client topic is empty by default to indicate no MQTT-speaking things exist in the smart space initially.
- The Private_Broker topic holds the access information (e.g., URL, IP, and port) for an optional new broker assigned to the current smart space by the space owner. The Private_Broker topic enables the space owner to specify an alternative broker for the space things to use, other than the default broker defined by SSB of the IoT-DDL. If the content of the topic is null by default, no private broker exists and Atlas things will use the default broker. Such a powerful Atlas thing (which offers broker service) publishes the access information of the offered broker service to the Private_Broker topic. All other subscribed Atlas things are notified of the new broker, disconnect from the default broker defined by the SSB identified in IoT-DDL, and connect and subscribe to the same topic hierarchy on the new broker.

3.1.2 Multicast Topic

In the Atlas topic hierarchy, the multicast topic is the topic channel where Atlas things can propagate information-based interactions to all other Atlas things in the smart space [30]. An Atlas thing announces its presence, identity, and internal entities on the Thing_Identity topic, and APIs for its offered resources to other things in the smart space on the Thing_API topic. The multicast topic, which all Atlas things are subscribed to, not only allows an Atlas thing to announce its presence and capabilities, but also enables an Atlas thing to discover other things and the distributed services offered by them. However, as will be mentioned in section 3.1.3, an Atlas thing can unicast an API call for a service offered by another thing on a separate dedicated channel linked to the peer's ID, which guarantees that no other things are listening to the conversation.

The updates on the topics listed under "Network Topic" shape two cases for multicasting. The existence of MQTT-speaking things in the smart space indicated by the MQTT_Client topic (as the first case), requires the MQTT-speaking things to affect a multicast through the broker and the non-MQTT-speaking things to utilize the translator to multicast through the broker. The absence of MQTT-speaking things, on the other hand (as in the second case), triggers CoAP-speaking things to utilize the UDP multicasting capability of the CoAP standard on the same multicast topics, as illustrated in Fig. 2.

3.1.3 Unicast Topic

The unicast topic of the Atlas topic hierarchy is the topic channel where an Atlas thing can initiate peer-to-peer, action-based interactions in the form of API calls. The action-based interactions enable an Atlas thing to request a specific service or call for an API offered by another Atlas thing in the smart space. With the presence of MQTT-speaking things in the smart space, indicated by the MQTT_Client topic (as in the first case), each non-MQTT-speaking thing through its translator as well as MQTT-speaking thing subscribes to its Interaction_(ATID) topic on the broker. Including ATID as part of the topic name ensures that each thing has a unique channel through which other things can initiate a private unicast. An Atlas thing that initiates a unicast interaction publishes to the unicast topic of the corresponding peer using the peer's ATID to ensure that no other things are listening to the conversation.

The absence of MQTT-speaking things (as the second case) triggers non-MQTT-speaking things to utilize the UDP unicasting capability of the CoAP standard on the same topic of the peer (in case of CoAP-speaking things)

and the TCP unicasting capability of HTTP (in the case of HTTP-speaking things), as illustrated in Figs. 2 and 3, respectively. The payload of the unicast interaction contains the API call for the offered service in addition to the API's required arguments for the other thing to handle the execution, as described in [30].

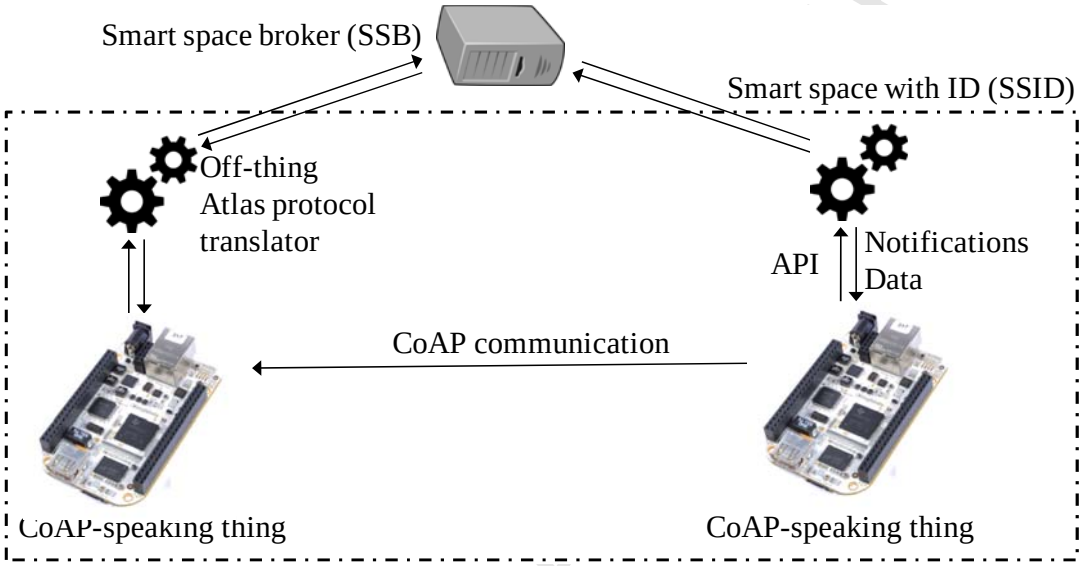


Figure 2: CoAP standard multicast and unicast communication.

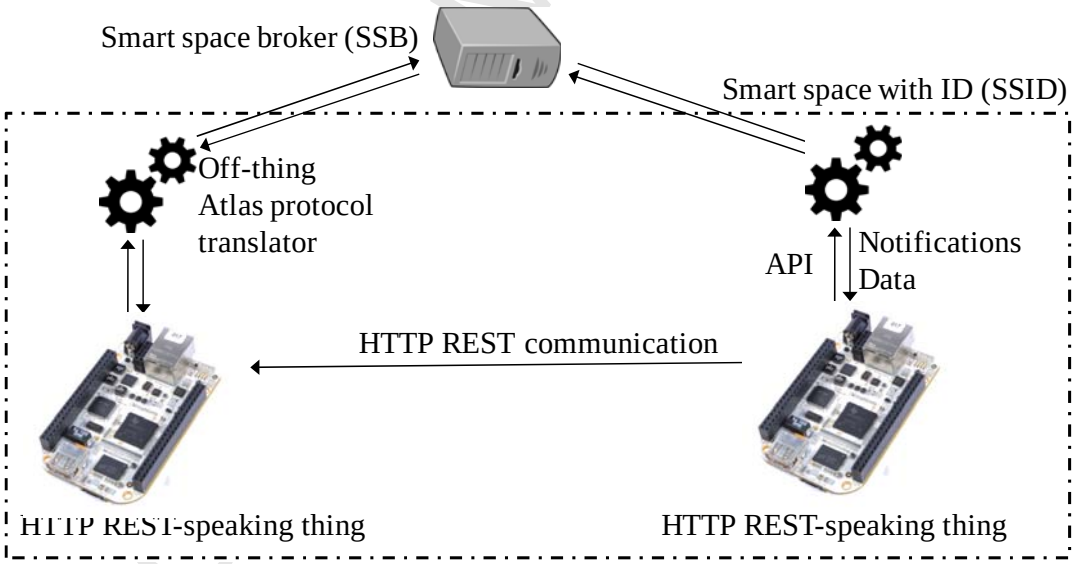


Figure 3. HTTP REST standard unicast communication.

3.2 Atlas IoT Protocol Translator

The Atlas protocol translator provides MQTT translation service for a non-MQTT-speaking thing through a set of well-defined programmed interfaces for the seamless interaction with MQTT things. The proposed translator is a thing attachment [30] that can either be an off-thing accessory residing on a remote platform (e.g., cloud platform

or edge device) or an on-thing service that is executed by the thing itself, as will be explained in section 3.3. In either case, the Atlas thing interfaces with its own protocol translator and asks the translator to (a) connect to the broker identified by SSB and subscribe to the Atlas topic hierarchy; or (b) multicast or unicast data on specific topics through the interface functions depicted in Table 2. The translator, in return, notifies the thing of updates published to Atlas topics and interactions destined to the thing's unicast topic through the appropriate interface function.

Table 2: Translator interface functions

Method *EstablishConnection* **Input** [translator IP and Port]
Method *BrokerSettings* **Input** [*SSB* and Atlas Topic Hierarchy]
Method MulticastPublish **Input** [Multicast Topic]
Method UnicastPublish **Input** [Unicast Topic]

The Atlas protocol translator, as illustrated in Fig. 4, is composed of the following modules:

- The *translator gateway* accepts translation requests received from the connected non-MQTT-speaking thing through the interface function *EstablishConnection* and hands them to the *packet dispatcher and response builder* module. The gateway module also opens a communication channel with the broker indicated by the parameters of the *EstablishConnection* request and hands the responses from the packet dispatcher and response builder module back to the connected thing.
- The *packet analyzer and response builder* performs two functions. First, it interprets the received translation request on the interface functions. The request can be to (a) establish a connection with a broker and subscribe to the topic hierarchy (the *BrokerSettings* interface function), (b) publish data to a specific multicast topic (the *MulticastPublish* interface function), or (c) unicast an interaction topic (the *UnicastPublish* interface function). The packet analyzer submodule forwards the request to the connection manager, multicast handler, or the unicast handler, respectively. The module's second function is to capture MQTT responses on these requests from the broker and map them to the connected thing through the interface functions. It should be noted that each interface function request is processed on a separate thread, in which the thread is blocked until the packet dispatcher and response builder returns the response.
- The *connection manager* handles the requests from non-MQTT-speaking things to establish a connection with a broker and to subscribe to topic hierarchy (specified from the inputs to the *BrokerSettings* interface function). The manager also handles new connections to new private brokers that publish their existence to the Private_Broker topic. MQTT acknowledges on broker connection and subscriptions as well as when data notifications are handed back to the response builder submodule to notify the connected thing.
- The *multicast handler* translates data posting requests to a multicast topic (the *MulticastPublish* interface function) into MQTT publish requests to the *proxy client* and hands the acknowledgment back to the response builder submodule.
- The *unicast handler* translates the interaction requests to the unicast topic (the *UnicastPublish* interface function) into MQTT publish requests to the *proxy client* and hands the acknowledgment back to the response builder submodule.
- The *proxy client* is a lightweight client process that handles the connection, subscription, and publishing of MQTT packets to the broker specified by the connected thing through the gateway module. The proxy client also hands acknowledgments and data notifications back from the broker to the connection manager, multicast handler, and unicast handler modules according to the content.

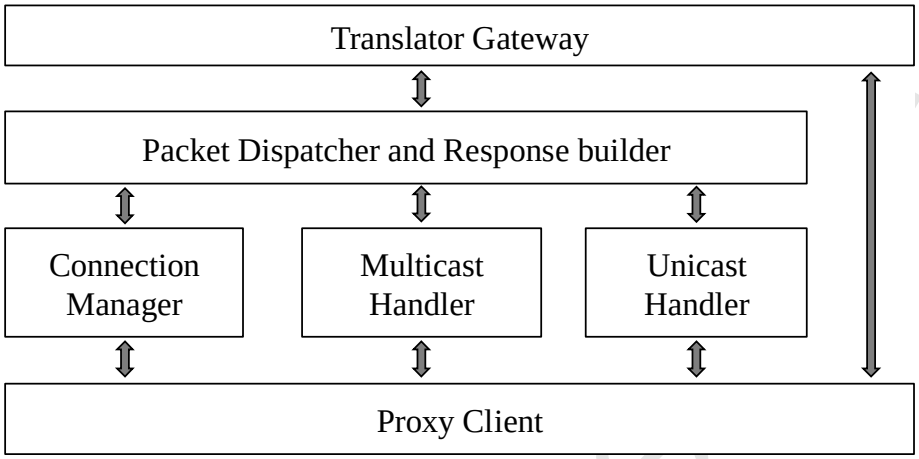


Figure 4: Atlas protocol translator.

For a precise and clear illustration of the operation of the proposed Atlas on- and off-thing protocol translator, we provide an algorithmic representation of the translator below. We also provide, in section 3.3, an algorithmic representation of operation of the proposed framework (Atlas IoT topic hierarchy, the Atlas IoT protocol translator, and the extended interface sublayer of the Atlas thing architecture). The algorithm creates a set of event call-back functions that are triggered on specific events (e.g., connection request from an Atlas thing, request to publish to topic, received updates on a subscribed topic) and the different interface functions defined in table 2.

Atlas protocol translator

Algorithm:

**SSID* //Smart space id, default empty
**ATID* //Atlas thing id, default empty
**MQTT_ Exist* //MQTT client exist, default false
**Current_Broker* //Broker access info, default empty

1. ***Set an event listener*** (Connection request from Atlas thing)
2. Set *SSID* to the smart space id of the connected thing
3. Set *ATID* to the atlas thing id of the connected thing
4. ***Set an event listener*** (Broker settings information from the connected Atlas thing)
5. Set *Current_Broker* according to the broker information
6. Connect to the *Current_Broker*
7. Subscribe to the Atlas topic hierarchy
8. ***Set an event listener*** (updates on *MQTT_Client* topic)
9. Set *MQTT_Exist* to true
10. ***Set an event listener*** (updates on *Private_Broker* topic)
11. Disconnect from the *Current_Broker*
12. Set the *Current_Broker* is the new broker
13. Connect to the *Current_Broker*
14. Subscribe to the Atlas topic hierarchy
15. Set *MQTT_Exist* to false
16. ***Set an event listener*** (publish request on multicast or unicast topic)
17. **If** *MQTT_Exist* is true **then**

```
        Build an MQTT packet with the input payload
        Publish the MQTT packet to the input topic
    end if
Set an event listener (updates on Multicast or Unicast topic)
    Parse the received MQTT response
    Reply on the connected Atlas thing with the received payload and the corresponding topic
Set an event listener (Broker is down)
    Set Current_Broker to the default SSB broker
    Connect to the Current_Broker
    Subscribe to the Atlas topic hierarchy
    Set MQTT_Exist to false
Set an event listener (Atlas Thing connection is down)
    Disconnect from the Current_Broker
```

3.3 Atlas Architecture Interface Sublayer

We extended the current interface sublayer of the Atlas thing architecture with further capabilities to enable the seamless interaction between the things that speak similar languages as well as things that speak different languages. The Atlas architecture [30] enables a thing in a smart space to engage in ad hoc interactions and interconnections, as well as IoT applications through a set of new services. Such services focus on descriptive and semantic aspects of things to better enable thing engagement, interaction, and IoT programmability. The platform tools the thing to self-discover its identity, components, and services, and to formulate APIs of its services. The thing also announces its identity, capabilities, and APIs to other things in the smart space through information-based interactions, while the action-based interactions include the applications that target the thing's capabilities and services in terms of API calls. The original interface sublayer of the architecture (the focus of this subsection), through which the communication protocols the thing speaks to engage and interact, currently supports only homogenous communication. Fig. 5 illustrates the Atlas thing architecture with a focus on the interface sublayer with the extended capabilities.

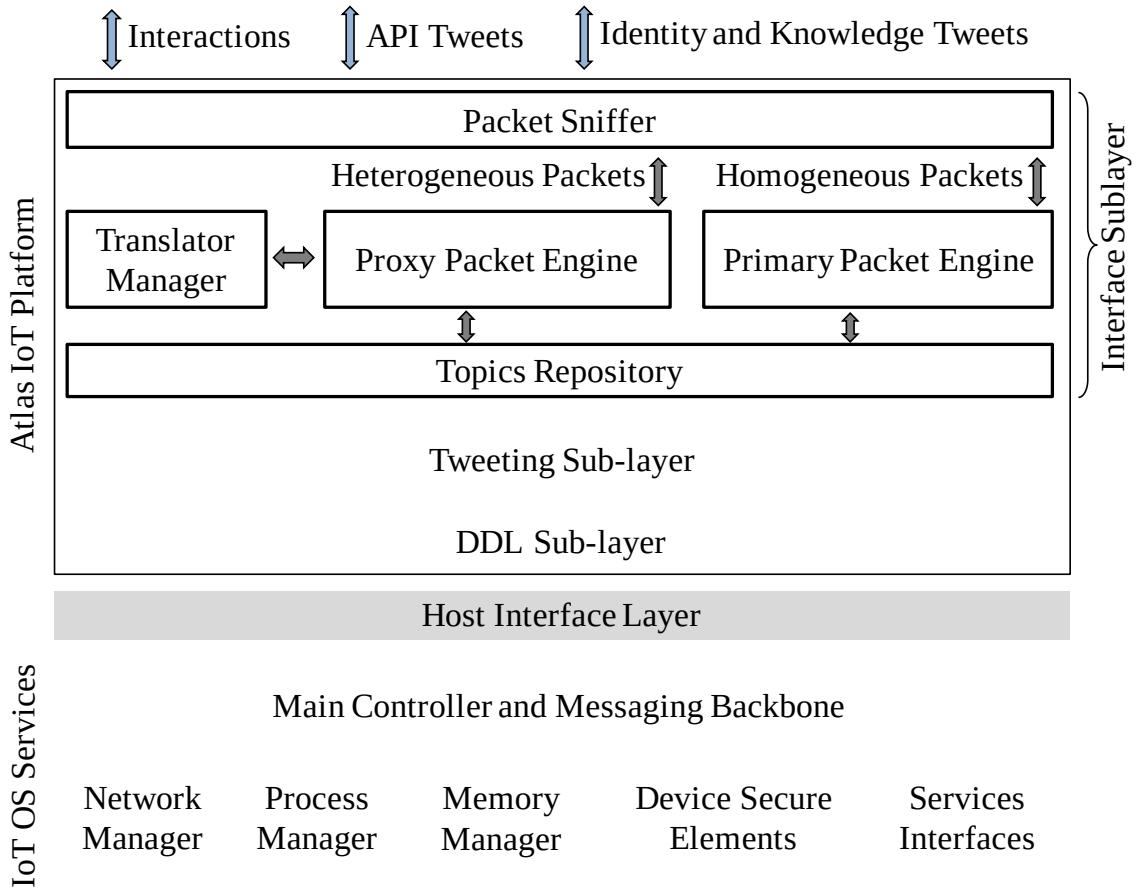


Figure 5: Atlas architecture with focus on the interface sublayer.

The extended interface sublayer of the Atlas thing architecture is composed of the following modules:

- The *packet sniffer* senses communication packets and classifies them as either homogenous packets of the same language that the thing speaks or heterogeneous packets received from the connected Atlas protocol translator.
- The *primary packet engine* handles the homogenous requests and interactions received, and builds the corresponding responses to the requesters.
- The *proxy packet engine* handles the heterogeneous requests and interactions destined for the translator manager and captures the responses to the requesters.
- The *translator manager* is either an on-thing attachment of the Atlas protocol translator or an off-thing attachment of the same. In the latter case, the translator manager opens and manages a communication channel with the protocol translator through the well-defined interfaces depicted in table 2.
- The *topic repository* keeps track of the latest published values to the various Atlas topics and interacts with the tweeting sublayer of the Atlas thing architecture to access both information-based and action-based tweets and interactions.

In this section, we described our Atlas IoT heterogeneous communication framework by presenting the Atlas IoT topic hierarchy, the Atlas IoT protocol translator, and the extended interface sublayer of the Atlas thing architecture. Seeking a precise and clear definition of our proposal, we provide an algorithmic representation of the framework below.

Atlas thing communication framework

Algorithm:

**SSID*	//Smart space id, default empty
**ATID*	//Atlas thing id, default empty
**MQTT_ Exist*	//MQTT client exist, default false
**Current_Broker*	//Broker access info, default empty
**Native_Language*	//Communication protocol, default empty
**Topic_Map*	//record for each topic, default empty
**Translator_settings*	//Atlas translator info, default empty

1. Powering Atlas thing
2. Call *Initialize* method
3. Call *Event_Handler* method
4. **while** *true* **do**
5. Call *Interaction_Handler* method
6. **end while**

The supporting algorithm *Initialize*, which configures the different modules of the interface sublayer of the Atlas thing architecture according to the uploaded IoT-DDL, is specified as follows.

Method *Initialize*: **Initialize interface sub-layer, communication channels and event listeners**.

1. Read *SSID, ATID, Current_Broker, Native_Language and Translator_settings* parameters from the uploaded IoT-DDL
2. ***If*** the Atlas thing provides private broker service **then**
3. Connect to the *Current_Broker*
4. Publish broker access information to Private_Broker
5. Disconnect from the *Current_Broker*
6. Change *Current_Broker* to the new broker
7. **end if**

The supporting algorithm Event_Handler, which initializes communication channels and creates event call-back functions that are triggered on specific events (e.g., a packet is received, the broker connection is down, or a MQTT client is in the smart space), is specified below. The event listener is a call-back mechanism used and developed by the communication standards (MQTT, CoAP, and HTTP REST) to track specific events.

Method *Event_Handler*: **Initialize communication channels and listeners for events**.

31. ***If*** *Native_Language* is MQTT **then**
32. Connect to the *Current_Broker*
33. Set *MQTT_Exist* to true
34. Publish true to MQTT_Client topic
35. Subscribe to Atlas topic hierarchy
36. ***else if*** *Native_Language* is CoAP or HTTP-REST **then**
37. Connect to translator using *Translator_settings*

```
38.       Subscribe to Private_Broker and MQTT_Client topics
39.       Set an event listener (MQTT_Client topic update)
40.       Set MQTT_Exist to true
41.       Subscribe to multicast topics and the unicast topic
42. end if
43. //the above if-else statements handle both MQTT and non MQTT speakers differently
44. //the next events' call back functions handle both similarly
45. Set an event listener (Packet received)
46.    Call Interaction_Handler method
47. Set an event listener (Private_Broker topic update)
48.    Disconnect from the current broker
49.    Change Current_Broker to the new broker
50.    Call Event_Handler method
51. Set an event listener (Broker is down)
52.    Set Current_Broker to the default SSB broker of IoT-DDL
53.    Call Event_Handler method
```

The supporting algorithm Interaction_Handler, which builds multicast and unicast interactions as well as interactions' responses to other things in the smart space, is specified as follows.

Method *Interaction_Handler*: **Creates outgoing interactions and processes incoming interactions**.

```
1.  If new interaction then     //Creating outgoing interaction
2.    Build up the interaction
3.    If Multicast interaction then
4.      If MQTT_Exist then
5.        If Native_Language is MQTT then
6.            Publish interaction to the corresponding multicast topic
7.        Else if Native_Language is CoAP or HTTP then
8.            Publish interaction through translator to the corresponding multicast topic
9.        end if
10.     else if Native_Language is CoAP then
11.       CoAP UDP-multicast the interaction to the corresponding multicast topic
12.     end if
13.   else                                  // Unicast interaction
14.     If Native_Language is CoAP & peer is CoAP then
15.       CoAP UDP-unicast the interaction to the corresponding unicast topic
16.     else if Native_Language is HTTP & peer is HTTP then
17.       HTTP-REST TCP-unicast the interaction to the corresponding unicast topic
18.     else if Native_Language is CoAP & peer is non CoAP then
19.       Post the interaction through the Atlas protocol translator to the corresponding unicast topic
20.     else if Native_Language is HTTP & peer is non HTTP-REST then
21.       Post the interaction through the Atlas protocol translator to the corresponding unicast topic
22.     else                                  //MQTT speaking thing
23.       Publish interaction to the corresponding unicast topic
```

```
24.    end if
25.  end if
26. else                          //Processing an incoming interaction
27.  Parse request
28.  If (Multicast interaction) then
29.     Update the Topic_Map
30.  else       //unicast interaction received on Interaction_(ATID)
31.     Build up the corresponding response interaction
32.     Jump to line 14
33.   end if
34. end if
```

The proposed framework imposes some requirements on things to be part of such an interoperable multiprotocol ecosystem and to enable a smooth interaction with thing mates that utilize similar or different communication protocols. First, Atlas things are assumed to have an Internet connection (e.g., Wi-Fi, Ethernet, or cellular network) if the default SSB specified in the IoT-DDL or the private broker required by the space owner is not in the things' local network. Second, Atlas things are assumed to have an operating system (e.g., Android, embedded Linux, ARM Mbed-OS [33], or Google Brillo [32]) that supports multithreading, where the thing can create a bidirectional communication channel that can interact and listen for interactions simultaneously.
Seeking a clear illustration of the operation of the Atlas protocol translation, we provide the transition state diagram in figure 6 below.

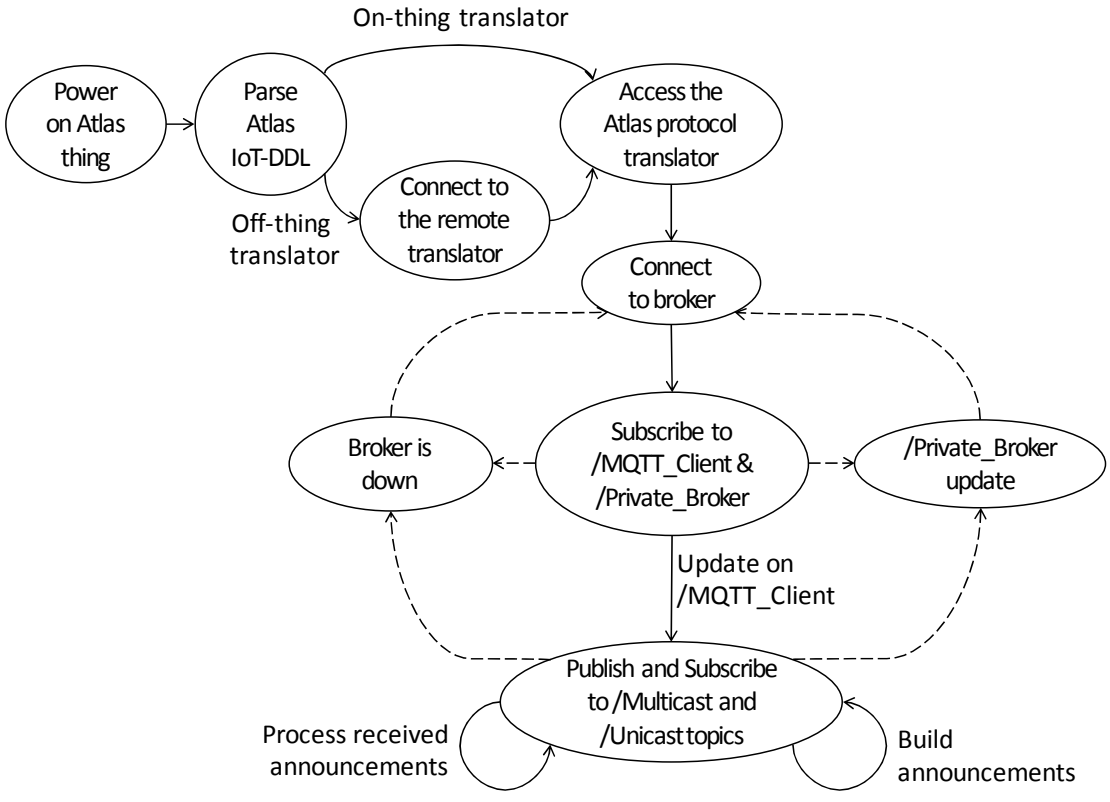


Figure 6: State transition diagram for the operation of the on-thing and off-thing Atlas protocol translators.

4. IMPLEMENTATION AND BENCHMARKING

In this section, we give an overview of the communication protocols and how the IoT-DDL of an Atlas thing reflects the proposed framework attributes for an interoperable ecosystem. We then provide a benchmarking study to measure energy consumption as well as the code footprint of the different aspects of each protocol. The aspects benchmarked are the engagement of the thing with other things through multicasting its presence, unicasting interactions (e.g., issuing API calls) and listening to similar interactions sent by thing mates in the smart space. This section also describes the implementation details of engaging the Atlas communication framework in various homogenous and heterogeneous network scenarios. The framework adopts the open source C++ implementation of CoAP developed by Noisy Atom [23], C/C++ implementations of MQTT developed by the Paho Eclipse project [22], and HTTP REST developed by the C Curl library [24]. The framework also utilizes a connection with the cloud-based MQTT broker HiveMQ dashboard [9] as the default broker defined by SSB in the Atlas thing IoT-DDL. The things used in this study are Raspberry Pi Model B and Beaglebone black sensor platforms with the specifications listed in Table 3. The benchmarking and implementation demonstrate the feasibility of the framework on these real platforms.

Table 3: Sensor platform specifications

Specifications	Raspberry Pi Model B	Beaglebone Black
OS	Raspbian	Angstorm
Processor	900 MHz Quadcore ARM Cortex A7	1 G-Hz Sitara AM3359 ARM Cortex A8
Network Module	Onboard Ethernet	Onboard Ethernet
RAM	1 GB	512 MB

4.1 IoT-DDL Specifications of the Atlas IoT heterogeneous communication framework

This section shows how the Atlas thing IoT-DDL configuration file is extended to enable the proposed framework and engage the communication language (detailed in section 2.1) the thing speaks. IoT-DDL specifications [30] describe the different parts and accessories of a thing in a smart space through a set of attributes, parameters, and properties. The structure of IoT-DDL is divided into (a) the Atlas thing section, which characterizes the thing as a whole, describing the different resources and components on board; (b) the thing entities section, which describes the different types of entities (hardware, software, or hybrid) that can be embedded, built in, or connected to the thing; and (c) the thing attachments section, which describes the different cloud-based and on-thing accessories that expand the thing's functionalities and resources.

The descriptive metadata subsection of the Atlas thing section of IoT-DDL specifications, as illustrated in Fig. 7, is the subsection that is used to uniquely identity the thing in the smart space, by providing its ATID (line 8), the smart space SSID (line 9), and the thing's vendor, owner, model, and type (line 2 to 7). The IoT-DDL manager module of the DDL sublayer of the architecture parses the attributes of the descriptive metadata subsection and configure the parameters of the Identity parser module of the DDL sublayer to uniquely identify the thing [30]. Another IoT-DDL subsection of the Atlas thing section is the network manager. The network manager describes the network connection capabilities in terms of the mounted network module (e.g., Wi-Fi or Bluetooth), network access information, and preferences (lines 3–5). The network manager lists the required attributes and properties of the communication protocol that the thing speaks (lines 6–10). The network manager section also, as illustrated in Fig. 8, includes configuration of the SSB broker for the thing to initially connect to as well as the Atlas topics (lines 11–17) for the thing to subscribe to either directly (for the MQTT-speaking things) or through the Atlas protocol translator (for the non-MQTT-speaking things, as will be declared in the thing attachment section of IoT-DDL). The IoT-DDL manager module of the DDL sublayer of the architecture parses the attributes of the network manager subsection and configures the parameters of the Interface sublayer of the architecture for the thing to start interact with other things in the smart space.

```
1. <Descriptive_Metadata>
2.    <Thing_Owner>CISE-UF</Thing_Owner>
3.    <Thing_Name>Raspberry Pi</Thing_Name>
4.    <Thing_Id>RPi1</Thing_Id>
5.    <Thing_Model>B+</Thing_Model>
6.    <Thing_ShortDescription>sensor-platform </Thing_ShortDescription>
7.    <Thing_Type>ThingOfThings</Thing_Type>
8.    <Thing_ATID>AtlasThing128</Thing_ATID>
9.    <Thing_SSID>SmartSpace326012</Thing_SSID>
10.         ….
11. </Descriptive_Metadata>
```

Figure 7: The descriptive metadata section in the IoT-DDL for an Atlas thing.

```
1. <Administrative_Metadata>
2.      <Network_Manager>
3.        <Module>Wifi</Module>
4.        <Type>External USB</Type>
5.        <Network_Name>ATLAS</Network_Name>
6.        <Transport_Protocol>TCP</Transport_Protocol>
7.        <Communication_Protocol>CoAP </Communication_Protocol>
8.        <Mulicast_Address>266.1.1.1</Multicast_Address>
9.        <Listen_Port>5755</Listen_Port>
10.       <Interaction_Port>4322</Interaction_Port>
11.       <SSB>broker.hivemq.com:1883</SSB>
12.       <Topic_Root>Atlas</Topic_Root>
13.       <Network_MQTTClient>/Network /MQTT_Client </Network_MQTTClient>
14.      <Network_PrivateBroker>/Network /Private_Broker </Network_PrivateBroker>
15.      <Multicast_ThingIdentity>/Multicast/Thing_Identity </Multicast_ThingIdentity>
16.      <Multicast_API>/Multicast/Thing_API </Multicast_API>
17.       <Unicast>/Unicast/Interaction</Unicast>
18.           ….
19.    </Network_Manager>
20.         ….
21. </Administrative_Metadata>
```

Figure 8: The network manager subsection in the IoT-DDL for a CoAP-speaking Atlas thing.

The thing attachment section of the IoT-DDL specifications includes the protocol translator for CoAP- and REST-speaking things. The protocol translator subsection includes the different attributes and parameters to configure the translator (in case of an on-thing translator) or to create a communication channel with the translator (in case of an off-thing translator as shown in Fig. 9).

```
1. <Thing_Attachment>
2.    <Protocol_Translator>
3.       <Type>off-thing</Type>
4.       <Description>Atlas protocol translator</Description>
5.       <Translator_URL>192.168.0.4</Translator_URL>
6.       <Translator_PORT>57577</Translator_PORT>
7.       <Transport_Protocol>TCP</Transport_Protocol>
8.       <Availability>Enabled</Availability>
```

```
9.      ....
10.  </Protocol_Translator>
11.         ....
12. </Thing_Attachment>
```

Figure 9: An off-thing Atlas protocol translator in the thing attachment section in the IoT-DDL.

4.2 Benchmarking Code Footprint and Energy Consumption of Basic Communication Operations

To characterize our proposed framework in terms of concrete metrics we measured its various code footprints and the energy cost of all its basic operations. We measured the code footprint of the different features of the proposed Atlas communication framework. The code footprint depicted in table 4, in kilobytes, represents the compressed version of the code files and libraries of the proposed framework components and features.

Table 4: Code footprint (in kilobytes) for the different features of Atlas communication framework

Feature	Code footprint
Atlas thing architecture	85
MQTT client library [22]	525
CoAP library [23]	76
REST library [24]	12
On-thing Atlas protocol translator	540
Off-thing Atlas protocol translator	565

Our communication framework loaded with IoT-DDL, in addition to the specific lightweight communication library (MQTT, CoAP or HTTP-REST) is evidently small in size to fit many of the available IoT platforms. Our framework footprint also compares favorably when compared to existing related work. In fact the Atlas thing architecture with a CoAP library and an on-thing translator (which is about 700KB – the highest footprint among all features in our framework) records a slightly smaller footprint when compared to the 710KB of Eclipse Ponte framework (which we described in the related work section 2.2) in addition to the CoAP client library for communication.

To benchmark energy consumption characteristics of the homogenous basic communication functionalities of our framework (where things that speak similar languages can interact), we measured the energy consumed under CoAP, HTTP and MQTT protocols on the two different IoT platforms described in table 3. Basic functionalities, as depicted in table 5,6 and 7, enable the thing to:

a. Announce its presence (64 bytes) and own APIs (64 bytes) to the Atlas multicast topics, through publishing to the broker (in case of MQTT speaking thing), UDP GET on a multicast IP address (in case of CoAP speaking thing) or TCP POST on the thing mate's IP address (in case of HTTP-REST speaking thing). It should be noted as mentioned before in section 3.1.2 that MQTT speaking things affect a multicast through the broker, whereas the CoAP standard protocol is equipped with UDP multicast feature in which the homogenous CoAP-speaking things can join the same multicast group to receive such announcements. However, HTTP-REST utilizes TCP for communication that does not enable a multicast feature. Thus, an HTTP-REST speaking thing can only utilize a unicast POST on other peers' IP addresses.
b. Interact with a thing mate in a peer-to-peer fashion (e.g., API call) on the Atlas unicast topic through publishing, UDP GET unicasted on the mate's IP address and TCP GET on the mate's IP address in cases of MQTT, CoAP and HTTP-REST speaking things, respectively.

Discover presence and APIs of thing mates by receiving other thing announcements on the Atlas multicast topics as well as processing API's on an Atlas unicast topic.
Energy consumption is measured using PowerJive [31] – a USB-based device that measures voltage and capacity. For each operation in tables 5-7, we measured the total energy consumed in executing a loop consisting of only this operation for 10 minutes. To factor out energy consumed by background OS processes, we measured their total energy consumption over the same 10 min duration, which we then subtracted from that of the operation to obtain the adjusted total energy consumed by the operation. A first average was taken by dividing the adjusted total energy by the number of times this operation was completed within the 10 minutes duration. The whole process was repeated three times for each operation and a second average was calculated over the three measurements. Tables 5-7 show the measurements of energy consumption (watt-seconds) on the two hardware platforms with respect to the basic communication functions under MQTT, CoAP and HTTP-REST protocols, respectively.

Table 5: Energy consumption (in watt seconds) measurements for MQTT functionalities.

	Connect to SSB broker	Publish	Receive and process message
Raspberry Pi	0.2985	0.03947	0.0399
Beaglebone Black	1.2739	0.16689	0.1507

Table 6: Energy consumption (in watt seconds) measurements for CoAP functionalities.

	Multicast (POST)	Unicast (GET)	Receive and process message
Raspberry Pi	0.00025	0.00247	0.00053
Beaglebone Black	0.000790	0.00340	0.00160

Table 7: Energy consumption (in watt seconds) measurements for REST functionalities.

	Unicast (POST)	Unicast (GET)	Receive and process message
Raspberry Pi	0.01813	0.0165	0.0078
Beaglebone Black	0.02885	0.0209	0.0327

It should be noted that basic measurements obtained in tables 5-7 could vary if the if we change the type of network used, the processor clock frequency or the amount of RAM used. However, these measurements are important and critical to designing a battery-operated thing in a given use scenario with a targeted thing operational lifetime.
It should also be noted that the more powerful platform – the Raspberry Pi with 1GB of RAM and quad-core processor – consumes less energy for all basic operations compared to the Beaglebone Black platform. CoAP that utilizes UDP is also shown to be a lightweight communication protocol compared to both MQTT and REST that utilize TCP on both Raspberry Pi and BeagleboneBlack.

4.3 Benchmarking Energy Consumption under Different Scenarios

Unlike section 4.2, in which we measured the energy consumption of each individual operation in our communication framework, in this section we measure the energy consumed in full-interaction scenarios from the point of view of one thing in a network of four or more things. We examine two main scenarios: homogeneously communicating things (section 4.3.1) and heterogeneously communicating things (section 4.3.2). We also compare energy consumption of our communication framework with Eclipse's Ponte system (section 4.3.3) and

finally examine the scalability of our proposed approach as the number of communicating Atlas things in a smart space increases (section 4.3.4).

For the above scenarios, we set up a network of four Atlas communicating things (two Raspberry-Pi's and two Beaglebone black). Each Atlas thing interacts with thing mates by generating random interactions that follow a uniform distribution process with a mean interval of 5 seconds, for a period of 10 minutes. Interactions are limited to only identity announcements or unicast interactions (e.g., API calls) with other thing. Each thing also listens to other things' announcements and API calls.

If configured to speak HTTP REST or CoAP, an Atlas thing can be further configured with either on-thing or off-thing Atlas protocol translator attachments. No attachments are necessary if an Atlas thing is configured to speak MQTT. In case of off-thing attachment, the Atlas protocol translator resides on a Linux Ubuntu machine on the same local network as the Atlas things.

Given space limitation we report only for the Raspberry Pi even though we have conducted experiments for the Beaglebone black as well.

4.3.1 Experiment-1: Homogenous things scenario

In this experiment we measure the energy consumed by an Atlas thing when it engages within a homogeneous network of four things. In this experiment an arbitrary thing is chosen at random to take the measurements. MQTT-speaking things are found to consume more energy, as shown in figure 10 as an MQTT client requires a long-term TCP connection to be maintained with the broker. In the case of CoAP or HTTP REST, an Atlas thing can talk in discontinued sessions with other things over UDP and TCP channels respectively. Figure 9 shows that an Atlas MQTT-speaking thing consumes around 8% more energy than an HTTP REST-speaking thing and around 20% more than a CoAP-speaking thing for a homogenous things scenario.

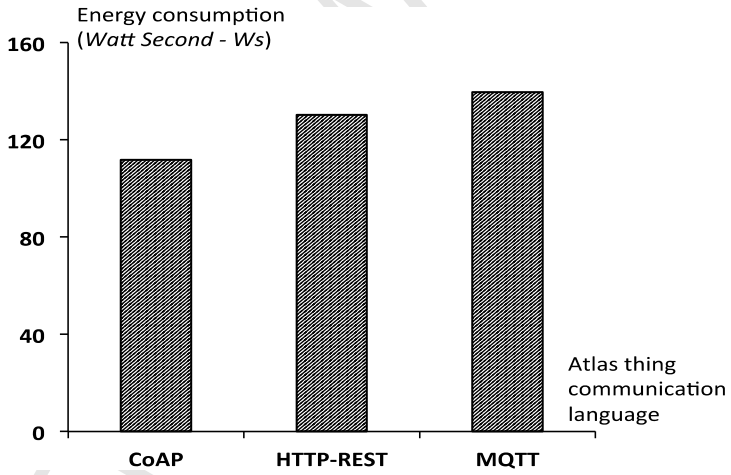


Figure 10: Atlas communication framework for homogenous speaking things.

4.3.2 Experiment-2: Heterogeneous things scenario

In this experiment we measure the energy consumed by an Atlas thing when it engages within a heterogeneous network of four things (two MQTT-speaking, one CoAP-speaking and one-HTTP REST speaking). The experiment shows the energy difference between utilizing an on-thing and off-thing Atlas protocol translator for the non MQTT-speaking things to engage with other things through the SSB broker. As it uses no translators, MQTT thing's energy consumption is not measured in this experiment. The on-thing Atlas protocol translator –as listed in table 4– taxes an extra code footprint on the Atlas node, however it saves energy that would be consumed by the thing to create a connection and to listen to notifications from an off-thing Atlas protocol translator, as shown in figure 11. It is worth mentioning that with a 540 more kilobytes of code, the on-thing Atlas protocol

translator saves around 7% in energy. Such little energy saving could add up in a real life scenario over a much longer duration.

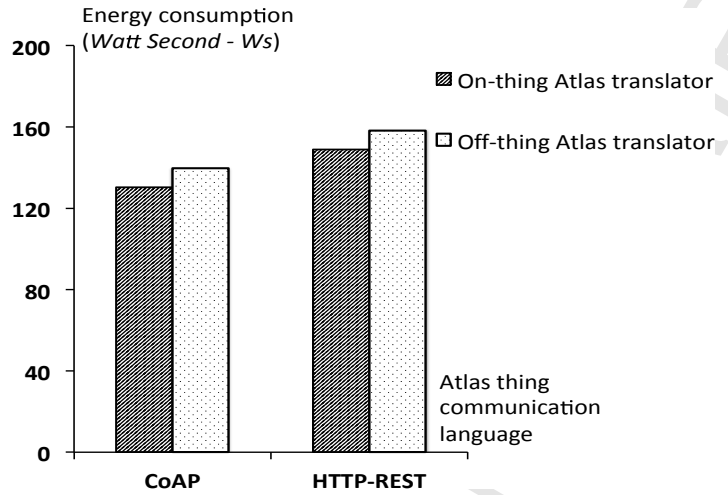


Figure 11: Atlas communication framework for heterogeneous speaking things.

In this experiment we also measure the time required for the non-MQTT speaking thing to utilize the Atlas protocol translator (on-thing and off-thing) to multicast an announcement. Time is measured using the Unix-Chrono library for a high-resolution clock cast to microseconds. The off-thing translator takes more time, as would be expected, shown in figure 12, as it receives request from the connected thing to send an announcement to the smart space. The time difference between off-thing and on-thing is not significant though. The off-thing translator takes more time because, upon receiving a request, it must build the MQTT packet and announce it using the Atlas IoT topic hierarchy (Table 1). It should be noted that this set of experiments shows a slight increase in latency mainly because the four Atlas things and the off-thing Atlas protocol translator (on a Linux Ubuntu machine) reside in the local network with low traffic load. However, the time difference between utilizing the on-thing and off-thing translator to multicast announcement would increase proportionally to the network traffic when more things are added into the smart space. Supporting both on- and off-thing translator in our approach would therefore allow our framework to be configured appropriately based on the scale of the smart space deployment.

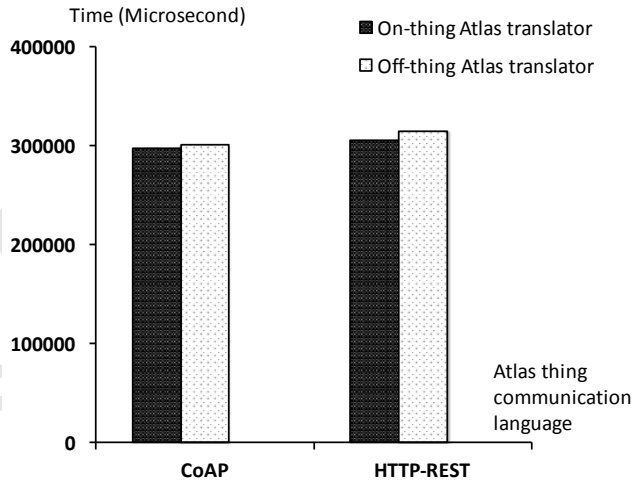


Figure 12: Time comparison for the on-/off-thing Atlas translator for non-MQTT speaking things.

4.3.3 Experiment-3: Atlas framework vs Eclipse Ponte

In this experiment we compare energy consumption by our Atlas communication framework with Eclipse's Ponte framework [22] in both homogenous and heterogeneous scenarios. Eclipse Ponte –described in section 2.2– offers a set of APIs that enable MQTT, REST-HTTP and CoAP speaking things to communicate through a centralized gateway that speaks all these languages. Eclipse Ponte requires the space owner to dedicate a powerful edge device (e.g., a server) that hosts the gateway with a continuous source of power that guarantee the gateway availability. Our framework, on the other hand, utilizes the cloud-based MQTT broker HiveMQ dashboard [9] as the default broker defined by SSB in the Atlas thing IoT-DDL. Such cloud-based broker alleviates the requirement on the space owner to set up a centralized gateway dedicated to the smart space and reduces the effort of creating a shared channel for things to communicate.

For this experiment, we installed Eclipse Ponte on a Linux-Ubuntu workstation connected to the local network and which utilized MQTT client [22], CoAP client [23] and HTTP-REST client [24] on things listed in table 3. We also utilized a network of 4 things (2 MQTT things, one CoAP thing and one HTTP REST thing). For homogenous speaking things, as shown in figure 13, MQTT-speaking things that utilize Atlas framework consume more energy as the SSB broker is an online cloud broker compared to the Ponte workstation that exists on the local network. The homogenous things that speak CoAP or HTTP within Ponte framework consume more energy, as such things require a continuous connection to the centralized gateway for notifications and updates from other things that announce their presence or those that initiate unicast interactions (e.g., API calls). The advantage of our framework in the homogeneous case is however shown clearly in the CoAP and HTTP REST cases. Our framework consumes 22% less energy for CoAP things, and 17% less energy for HTTP-REST things. This homogeneous advantage is critical in planned spaces in which it is more likely to find homogeneously communicating things than heterogeneous.

For heterogeneous speaking things, as shown in figure 14 (notice MQTT is not presented as it does not use attachments), our framework and Ponte consume almost the same energy in the case of on-thing attachments (actually our framework is slightly better in the CoAP case, and slightly over in the HTTP-REST case). In the case of off-thing attachments, our framework consumes only 5% and 8% more energy than Ponte in the CoAP and HTTP REST cases, respectively.

It should be noted that our framework does not impose the use of a centralized gateway dedicated to the smart space to host Eclipse Ponte. Such is a heavy requirement imposed on the space owner. In contrast, our framework configures the Atlas things –through the IoT-DDL– to utilize any readily available standardized broker, which in this article we used the HiveHQ broker to enable a seamless interoperable smart space with the least human intervention and effort on the part of the space owner.

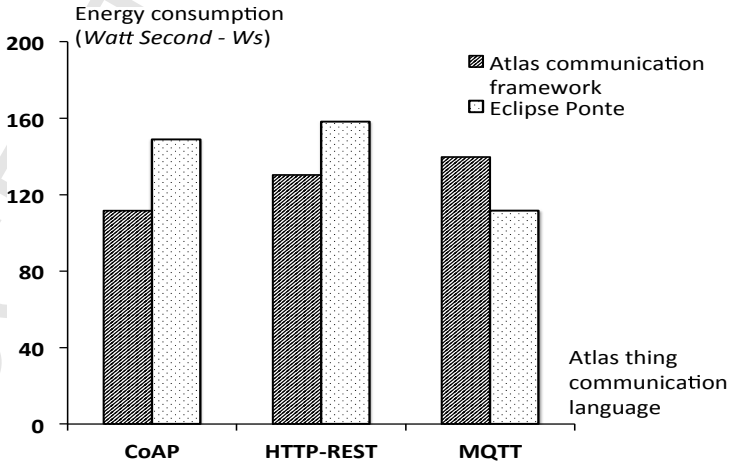


Figure 13: Comparing Atlas communication framework and Eclipse Ponte for homogenous communicating things.

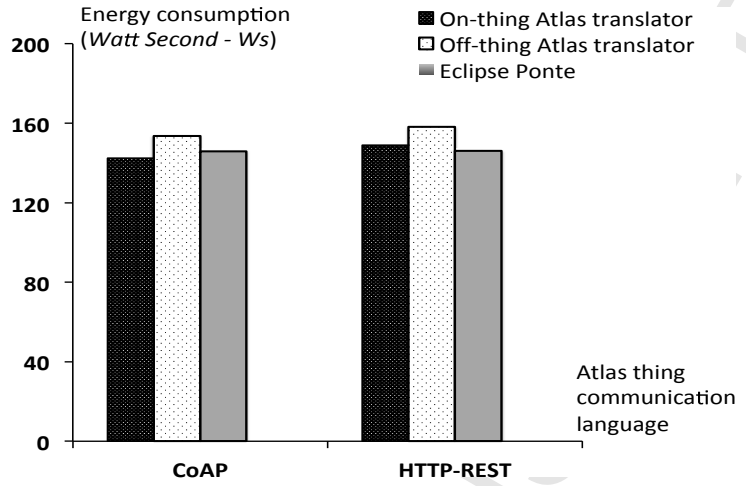


Figure 14: Comparing Atlas communication framework and Eclipse Ponte for heterogeneous communicating things.

4.3.4 Experiment-4: Atlas framework scalability

In this experiment we examine the scalability of our proposed framework by increasing the number of Atlas things that engage with each other in the smart space. However, in order to manage the experiment, we ran 5, 10 and 15 concurrent Atlas thing processes (virtual things) on a shuttle Linux-Ubuntu machine (a portable server machine). One Raspberry Pi Atlas thing that speaks CoAP and utilizes an on-thing Atlas protocol translator was used as point of measurements, and connected to a network of 6, 11, and 16 things running the same interaction scenario described in the beginning of section 4.3. Figure 13 shows the energy consumed by the Raspberry Pi Atlas thing. As shown in figure 15, the Atlas thing can interact and engage with a large number of thing mates that speak similar or different languages exhibiting only a small increase in the consumed energy. In fact, such small increase in the consumed energy seems to decrease gradually with the number of communicating Atlas things in the smart space. Doubling the number of things from 5 to 10 increased energy consumption by only 19%. Adding 50% more nodes (from 10 to 15) increased energy consumption by only 8%. The scalable increase in energy consumption reflects the energy spent in listening to announcements and learning APIs of the IoT as it grows and expands.

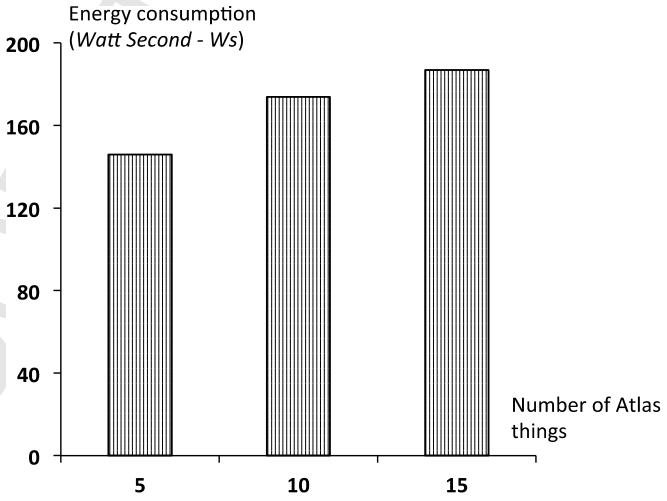


Figure 15: testing scalability on a CoAP-speaking Atlas thing with on-thing Atlas protocol translator.

5. CONCLUSION

In this paper, we propose the Atlas communication framework that enables distributed interaction between things that speak similar or different IoT communication languages. The proposed framework does not tax the performance of the homogenously communication things. The Atlas communication framework offers lightweight IoT interoperability through a protocol translator that resides either on the cloud platform or on the thing itself. The protocol translator enables the seamless communication between heterogeneous things through a set of well-defined interfaces. We demonstrate the feasibility of implementing and deploying the Atlas communication framework on real hardware platforms. We also present a benchmarking study to validate the energy consumption feasibility of our approach. The study measures both code footprint and energy consumption rate for the different aspects of the framework in homogeneous and heterogeneous settings. The results show the feasibility of enabling seamless heterogeneous communication between things with an acceptable energy cost. The results also show the advantage of not taxing the energy profile in the homogenous communication cases. We also compare our framework to the Eclipse Ponte framework and show the energy consumption advantage of our framework in the different communication scenarios. The scalability of the proposed framework is also tested and reflected the slight increase in energy consumption with adding more things in the smart space. The proposed framework and the experimentation results pave the way for other IoT communication protocols to engage in such interoperable ecosystem (e.g., Advanced Message Queuing Protocol (AMQP) and XMPP that utilize publish-subscribe architecture). The proposed Atlas communication framework enables the real participation of heterogeneous things in a smart space with least human intervention and the seamless integration of new things. Our proposed framework also opens the door for programming opportunities that may involve things that utilize different communication languages to cooperate and interact.

REFERENCES

[1] Bellavista, P. and Zanni, A. 2016, September. Towards better scalability for IoT-cloud interactions via combined exploitation of MQTT and CoAP. In Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), 2016 IEEE 2nd International Forum on (pp. 1-6). IEEE.

[2] Lee, C.H., Chang, Y.W., Chuang, C.C. and Lai, Y.H. 2016, October. Interoperability enhancement for Internet of Things protocols based on the software-defined network. In Consumer Electronics, 2016 IEEE 5th Global Conference on (pp. 1-2). IEEE.

[3] Desai, P., Sheth, A. and Anantharam, P. 2015, June. Semantic gateway as service architecture for IoT interoperability. In Mobile Services (MS), 2015 IEEE International Conference on (pp. 313-319). IEEE.

[4] Al-Fuqaha, A., Khreishah, A., Guizani, M., Rayes, A. and Mohammadi, M. 2015. Toward better horizontal integration among IoT services. *IEEE Communications Magazine*, 53(9), pp.72-79.

[5] Derhamy, H., Eliasson, J., Delsing, J., Pereira, P.P. and Varga, P. 2015, September. Translation error handling for multi-protocol SOA systems. In Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on (pp. 1-8). IEEE.

[6] Eclipse Ponte. http://www.eclipse.org/ponte/

[7] Massive scalable MQTT broker. http://emqtt.io/

[8] Eclipse MQTT sandbox. https://iot.eclipse.org/

[9] HiveHQ Enterprise MQTT broker. http://www.mqtt-dashboard.com/

[10] Mosquitto MQTT broker. https://mosquitto.org/

[11] Constrained Application protocol. http://coap.technology/

[12] CoAP RFC. https://tools.ietf.org/html/rfc7252

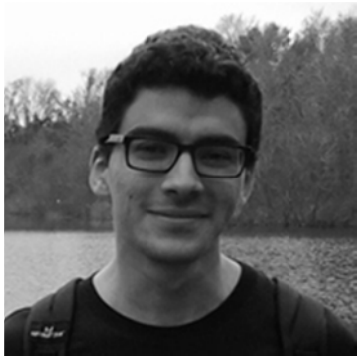
[13] Publish-Subscribe Broker for CoAP draft IETF. https://tools.ietf.org/html/draft-ietf-core-coap-pubsub-0

[14] XMPP open standard for messaging and presence. https://xmpp.org/

[15] Extensible Messaging and Presence Protocol (XMPP): Instant messaging and presence. https://xmpp.org/rfcs/rfc3921.html

[16] Want, R. and Dustar, S. 2015. Activating the Internet of Things [Guest Editor Introduction]. *Computer*, 48(9), pp. 16-20, (Sept. 2015).

[17] Gubbia, J., Buyyab, R., Marusic, S. and Palaniswami, M. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7), pp. 1645-1660.
[18] Miorandi, D., Sicari, S. De Pellegrini, F. and Chlamtac, I. 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7), pp. 1497-1516.
[19] Atzori, L., Iera, A. and Morabito, G. 2010. The Internet of Things: A survey. *Computer Networks*, 54(15), pp. 2787-2805.
[20] Coetzee, L. and Eksteen, J. 2011, May. The Internet of Things—Promise for the future? An introduction. IEEE IST-Africa Conference Proceeding, (pp. 1-9).
[21] Tan, L. and Wang, N. 2010, August. Future Internet: The Internet of Things. Advanced Computer Theory and Engineering (ICACTE), 3rd International Conference on, Vol. 5 (p. 376).
[22] Eclipse Paho open-source implementation of MQTT project. https://eclipse.org/paho/.
[23] CoAP implementation by Noisy Atom http://www.noisyatom.
[24] Libcurl—The multiprotocol file transfer library. https://curl.haxx.se/libcurl/
[25] MQ Telemetry Transport connectivity protocol. http://mqtt.org/
[26] IBM MQTT v3.1 protocol specification. public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html.
[27] Hypertext Transfer Protocol HTTP v1.1 rfc. https://tools.ietf.org/html/rfc2616
[28] Xinyang, F., Shen, J. and Fan, Y. 2009. REST: An alternative to RPC for Web services architecture. Future Information Networks (ICFIN), First International Conference on. IEEE.
[29] Karagiannis, V. et al. 2015. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1), pp. 11-17.
[30] Khaled, A.E. and Helal, S. 2017. IoT-DDL Device Description Language for the "T" in IoT. Submitted. For reviewing purposes only, the manuscript can be accessed at http://www.cise.ufl.edu/~aekhaled/IoTDDL_Device_Description_Language_for_the_T_in_IoT.pdf
[31] Reference the PowerUnit module PowerJive USB Voltage/Amps power meter. http://www.measuringsupply.com/artifact/1402679/.
[32] Google Brillo 2016. http://developers.google.com/brillo/.
[33] ARM Mbed OS 2016. https://www.mbed.com/en/development/mbed-os/
[34] Atlas IoT-DDL builder web tool. https://cise.ufl.edu/~aekhaled/AtlasIoTDDL_Builder.html
[35] Sumi Helal, Ahmed E. Khaled, and Venkata Gutta. 2017. Demo: Atlas Thing Architecture: Enabling Mobile Apps as Things in the IoT. In Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17). Utah, USA, October, 2017, pp. 480-482.



Ahmed E. Khaled is currently pursuing the Ph.D. degree in computer engineering, at the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA. He received the B.Sc. and M.Sc. degrees in computer engineering from Cairo University, Egypt in 2011 and 2013, respectively. His current research interests include Internet of Things, smart spaces, and ubiquitous computing.

Abdelsalam (Sumi) Helal (F'15) received the Ph.D. degree in computer sciences from Purdue University, West Lafayette, IN, USA. He is currently professor and the Chair in Digital Health, School of Computing and Communications, and the Division of Health Research, Lancaster University, UK. Before joining Lancaster University, he was professor in the department of Computer and Information Science and Engineering, University of Florida, USA, where he directed the Mobile and Pervasive Computing Laboratory and the Gator Tech Smart House. His research interests span pervasive systems, the Internet of Things, smart spaces, with applications to digital health and and assistive technologies for successful aging and independence.

- In this paper, we focus on overcoming light-weight communication protocol fragmentation and introduce the Atlas IoT communication framework which enables interactions among things that speak similar or different communication protocols.

- The framework tools up Atlas things with protocol translator “attachments” that enables the seamless communication between heterogeneous things through a set of well-defined interfaces.

- The framework supports seamless communication among the widely adopted Constrained Application Protocol (CoAP), Representational State Transfer (REST) over Hypertext Transfer protocol HTTP, and the Message Queue Telemetry Transport protocol (MQTT).

- We give a detailed benchmarking study to measure the energy consumption and code footprint characteristics of the different aspects of the framework on real hardware platforms.

- We compare our framework to the Eclipse Ponte framework and show how our framework is advantageous in energy consumption and how it is unique in that it does not tangibly penalize the homogeneous communication case.