Linguistic Abstractions for Interoperability of IoT Platforms

Maurizio Gabbrielli[1], Saverio Giallorenzo[2], Ivan Lanese[1], and Stefano Pio Zingaro[1]

[1] Università di Bologna / INRIA
[2] University of Southern Denmark

Abstract. The Internet of Things (IoT) advocates for multi-layered platforms—from edge devices to Cloud nodes—where each layer adopts its own communication standards (media and data formats). While this freedom is optimal for in-layer communication, it puzzles cross-layer integration due to incompatibilities among standards. Also enforcing a unique communication stack within the same IoT platform is not a solution, as it leads to the current phenomenon of "IoT islands", where disparate platforms hardly interact with each other. In this paper we tackle the problem of IoT cross-layer and cross-platform integration following a language-based approach. We build on the Jolie programming language, which provides uniform linguistic abstractions to exploit heterogeneous communication stacks, allowing the programmer to specify in a declarative way the desired stack, and to easily change it, even at runtime. Jolie currently supports the main technologies from Service-Oriented Computing, such as TCP/IP, Bluetooth, and RMI at transport level, and HTTP and SOAP at application level. We integrate in Jolie the two most adopted protocols for IoT communication, i.e., CoAP and MQTT. We report our experience on a case study on Cloud-based home automation, and we present high-level concepts valuable both for the general implementation of interoperable systems and for the development of other language-based solutions.

1 Introduction

The Internet of Things (IoT) advocates for multi-layered software platforms, each adopting its own media protocols and data formats [1, 2, 3]. The problem of integrating layers of the same IoT platform, as well as different IoT vertical solutions, involves many levels of the communication stack, spanning from link-layer communication technologies, such as ZigBee and WiFi, to application-layer protocols like HTTP, CoAP [4, 5], and MQTT [6, 7], reaching the top-most layers of data-format integration [8].

Technology-wise, architects of IoT platforms can choose between two approaches at odds. The first approach favors optimal in-layer communications, i.e., choosing media protocols and data formats best suited for the interactions happening among homogeneous elements, e.g., edge devices (connectionless protocols and binary data formats [3]), mid-tier controllers (gateways and aggre-

gators on the RESTful stack [9]), or Cloud nodes (scalable publish-subscribe message queues [10]). Following this first approach is optimal for in-layer communication. However, at the cross-layer level, the heterogeneity and possible incompatibility of the chosen standards make enforcing integrity within the IoT system complex and the resulting integration fragile. The second architectural approach favors cross-layer consistency, enforcing a unique communication stack over a single IoT platform. Here cross-layer integration is simpler thanks to the adoption of a single medium and data format. However such enforced uniformity is the main cause of the phenomenon known as "IoT island" [11, 12], where IoT platforms take the shape of vertical solutions that provide little support for collaboration and integration with each other. How to overcome this limitation is currently a hot topic, tackled also by ongoing EU projects, e.g., symbIoTe [12] and bIoTope [13].

In this paper we tackle the problem of IoT integration (both cross-layer and cross-platform) following a language-based approach focused on integration at both the transport (TCP or UDP) and application layer. To reach our goal we do not start from scratch, but we leverage the work done in the area of Service-Oriented Architectures (SOAs) [14] and we build on the Jolie programming language [15, 16, 17, 18]. In particular, we rely on those abstractions provided by Jolie that *i*) let different communication protocols seamlessly coexist and interoperate within the same program and *ii*) let programmers dynamically choose which communication stack should be used for any given communication. Concretely, we fork the Jolie interpreter—written in Java—into a prototype called JIoT [19], standing for "Jolie for IoT". JIoT supports all the protocols already supported by the Jolie interpreter (TCP at the transport level, and protocols such as SOAP, RMI and HTTP at the application level), while adding the application-level protocols for IoT, namely CoAP (and, as a consequence, UDP at the transport level) and MQTT. Notably, when the application protocol supports different representation formats (such as JSON, XML, etc.) of the message payload, as in the case of HTTP and CoAP, JIoT, like Jolie, can automatically marshal and un-marshal data as required.

We structure our presentation as follows. We overview in Section 2 our approach and summarize our contribution in Section 3. Then, we discuss the main challenges we faced in our development in Section 4, we present how a programmer can use CoAP/UDP (Section 5) and MQTT (Section 6) in JIoT, and we detail our implementation in Section 7. We describe, in Section 8, a scenario on Cloud-based home automation where a JIoT architecture coordinates heterogeneous edge devices. Finally, we position our contribution with respect to related work in Section 9 and we draw final remarks in Section 10.

JIoT is available at [19], released under the LGPL v2.1 license. The code snippets reported in this paper are based on version 1.2 of JIoT. The integration of JIoT into the official code-base of the Jolie language is ongoing work.

2 Approach Overview

Without proper language abstractions, guaranteeing interoperability among protocols belonging to different technology stacks is highly complex. The problem is further exacerbated when one has to modify the technology stack used for some specific interaction. The replacement may be either static, e.g., because of the deployment of new, heterogeneous devices in a pre-existing system, or dynamic, e.g., to support a changing topology of disparate mobile devices. Contrarily, with JIoT most of the complexity of guaranteeing interoperability is managed by the language interpreter and hidden from the programmer.

As an illustrative example of the proposed approach, let us consider a scenario where we want to integrate two islands of IoT devices, both collecting temperature data, but relying on different communication stacks, namely HTTP over TCP and CoAP over UDP. The end goal is to program a collector which receives and aggregates temperature measurements from both islands.

Following the structure of Jolie programs, the collector programmed in JIoT is composed of two parts: a *behavior*, specifying the logic of the elaboration, and a *deployment*, describing in a declarative way how communication is performed. This separation of concerns is fundamental to let programmers easily change which communication stack to use, preserving the same logic for the elaboration.

As an example of program behavior, let us consider the code below, where `main` is the entry point of execution of Jolie programs.

```
1 main {
2   ...
3   receiveTemperature( data );
4   ...
5 }
```

Above, line 3 contains a reception statement. Receptions in Jolie indicate a point where the program waits to receive a message. In this case, the collector waits to receive a temperature measurement on *operation* `receiveTemperature` (an operation in Jolie is an abstraction for technology-specific concepts such as channels, resources, URLs, ...). Upon reception, it stores the retrieved value in variable `data`. Besides the logic of computation of the collector, we also need to specify the deployment, i.e., on which technologies the communication happens; in the example above, how the collector receives messages from other devices. In Jolie this information is defined within *ports*. For example, the port to receive (denoted with keyword `inputPort`) HTTP measurements can be defined as in Listing 1. Port `CollectorPort1` specifies that the collector expects inbound communications via `Protocol http` using a TCP/IP socket receiving at URL `"localhost"` on TCP port `8000`. A port exposes a set of operations, collected within a set of `Interfaces`. In the example, the input port `CollectorPort1` declares to expose interface `TemperatureInterface`, which is defined at lines 1–3 of Listing 1. The interface declares the operation `receiveTemperature`, including the type of expected data (`string`), as a `OneWay` operation, namely an asynchronous communication that does not require any reply from the collector (except the acknowledgment automatically provided by the TCP implementation).

```
interface TemperatureInterface {
 OneWay: receiveTemperature( string )
}

inputPort CollectorPort1 {
 Location: "socket://localhost:8000"
 Protocol: http
 Interfaces: TemperatureInterface
}
```

Listing 1. Example of interface and input port in Jolie.

Thanks to port `CollectorPort1`, the collector can receive data from the HTTP island. To integrate the second island, we just need to define an additional port, similar to `CollectorPort1`, except for using UDP/IP datagrams at the transport layer and CoAP [5, 4] at the application layer. Hence, the whole code of the collector becomes:

```
interface TemperatureInterface {
 OneWay: receiveTemperature( string )
}

inputPort CollectorPort1 {
 Location: "socket://localhost:8000"
 Protocol: http
 Interfaces: TemperatureInterface
}

inputPort CollectorPort2 {
 Location: "datagram://localhost:5683"
 Protocol: coap
 Interfaces: TemperatureInterface
}

main {
 ...
 receiveTemperature( data );
 ...
}
```

Listing 2. Code of the Collector Example.

The example above highlights how, using the proposed language abstractions, the programmer can write a unique behavior and exploit it to receive data sent over heterogeneous technology stacks. Indeed, the `receiveTemperature` operation takes measurements from both the **`inputPort`**s. For instance, if communication over `CollectorPort2` fails, port `CollectorPort1` can still receive data. Programmers can also specify elaborations that depend on the used technologies, by using dif-

ferent operations in different ports. Jolie supports both inbound and outbound communications, the latter declared with `outputPort`s, whose structure follows that of `inputPort`s. Furthermore, the `Location` and `Protocol` of `outputPort`s can be changed at runtime, enabling the dynamic selection of the appropriate technologies for each context.

As mentioned, Jolie enforces a strict separation of concerns between behavior, describing the logic of the application, and deployment, describing the communication capabilities. The behavior is defined using the typical constructs of structured sequential programming, communication primitives, and operators to deal with concurrency (parallel composition and input choices [17]).

Jolie communication primitives comprise two modalities of interaction.

Outbound `OneWay` communications send a message asynchronously, while `RequestResponse` communications send a message and wait for a reply (they capture the well-known pattern of request-response interactions [20]). Dually, inbound `OneWay` communications wait to receive a message, without sending a reply, while inbound `RequestResponse`s wait for a message and send back a reply.

Jolie supports many communication media (via keyword `Location`) and data protocols (via keyword `Protocol`) in a simple, uniform way. This is one of the main features of the Jolie language, and the reason why we base our approach on it. Each communication port declares the medium and data protocol used to communicate, hence, to switch to a different technology stack, one just needs to change the declaration of `Location` and `Protocol` of a given port. As expected, the behavior (i.e., the actual logic of computation) of any Jolie program is unaffected by any change to its ports. Hence, a Jolie program can provide the same service (i.e., the same behavior) through different media and protocols just by specifying different deployments. Being born in the field of SOAs, Jolie supports the main technologies from that area: e.g., communication media like TCP/IP sockets, Bluetooth L2CAP, Java RMI, and Unix local sockets; and data protocols like HTTP, JSON-RPC, XML-RPC, SOAP and their respective SSL versions.

3 Contribution

To substantiate the effectiveness of our language-based approach to IoT integration, we add to Jolie support for the main communication stacks used in the IoT setting. Concretely, the added contribution of JIoT with respect to Jolie is the integration of two application protocols relevant in the IoT scenario, namely CoAP [5, 4] and MQTT [7, 6]. Notably, in JIoT the usage of such protocols is supported by the same linguistic abstractions that Jolie uses for SOA protocols such as HTTP and SOAP.

Even if Jolie provides support for the integration of new protocols, when set in the context of IoT technology, the task is non trivial. Indeed, all the protocols previously supported by Jolie exploit the same internal interface, based on two assumptions: *i*) the usage of underlying technologies that ensure reliable communications and *ii*) a point-to-point communication pattern.

However, those assumptions do not hold when considering the two IoT technologies we integrate:

- CoAP communications can be unreliable since they are based on UDP connectionless datagrams. CoAP provides options for reliable communications, however these are usually disabled in an IoT setting, since it is important to preserve battery and bandwidth;
- MQTT communications are based on the publish-subscribe paradigm, which contrasts with the point-to-point paradigm underlying the Jolie communication primitives. Hence, we need to define a mapping to express publish-subscribe operations in terms of Jolie communication abstractions. In doing so, we need to balance two factors: *i*) preserving the simplicity of use of the point-to-point communication style and *ii*) capturing the typical publish-subscribe flow of communications. Such a mapping is particularly challenging in the case of request-response communications. Remarkably, the mapping that we present in this work is general and could be used also in other contexts.

This paper integrates, revises, and extends material from [21], where we presented, discussed, and provided basic technical details on the proposed language-based approach to IoT integration. Main extensions comprise:

- advanced technical details on our implementation (Section 7) including:
  - a general account on how media and protocols are separated from the Jolie interpreter and how they can be developed as independent modules;
  - extensive details on the implementation of UDP, CoAP, and MQTT protocols;
- a comprehensive case study on a home automation scenario (Section 8) where we consider:
  - local, cross-layer communication among things and mid-tier controllers (edge devices and fog nodes);
  - remote, cross-layer interactions among Cloud nodes and mid-tier controllers.

We conclude this section briefly discussing the current limitations of JIoT related to its usage in the programming of low-level edge devices—like Arduinos and other microcontrollers. JIoT supports dynamic scenarios where the nodes in the network can switch among many technology stacks according to internal or environmental conditions, such as available energy or quality of communication. From preliminary discussions with collaborators and IoT practitioners, we collected positive opinions on the idea of using JIoT for programming low-level edge devices. Given these positive remarks, we investigated the feasibility of running JIoT programs over edge devices, possibly including additional language abstractions to provide low-level access to in-board sensors and actuators. However, our survey revealed a market of devices fragmented over incompatible hardware architectures and characterized by strong constraints over both computational power and energy consumption. Considering these limitations, we concluded that supporting the execution of JIoT-like programs over edge devices would require a strong engineering effort. While this research direction is

promising, we deem it non-urgent, since currently developers tend to program very simple behaviors for edge devices [3], which usually capture some data (e.g., through one of their sensors) and then send them to mid-to-top-tier devices. The latter usually process and coordinate the flow of data: they have powerful hardware, they communicate over reliable channels, and they have fewer (if any) constraints with respect to battery/energy consumption.

Considered the discussion above, in this work we omit the low-level programming of edge devices and we focus on mid-to-top-tier ones, which can host the JIoT runtime and which, given their topological context, directly benefit from the flexibility of the approach.

4 JIoT: Jolie for IoT

Jolie currently supports some of the main technologies used in SOAs (e.g., HTTP, SOAP). However, only a limited amount of IoT devices uses the media and protocols already supported by Jolie. Indeed, protocols such as CoAP [5, 4] and MQTT [7, 6], which are widely used in IoT scenarios, are not implemented in Jolie. Integrating these protocols, as we have done, is essential to allow Jolie programs to directly interact with the majority of IoT devices. We note that emerging frameworks for interoperability, such as the Web of Things [22], rely on the same protocols we mentioned for IoT, thus JIoT is also compliant with them. However there are some challenges linked to the integration of these technologies within Jolie:

- *lossless vs. lossy protocols* — In SOAs, machine-to-machine communication relies on lossless protocols: there are no strict constraints on energy consumption or bandwidth and it is not critical how many transport-layer messages are needed to ensure reliable delivery. That is not true in IoT networks, where communication is constrained by energy consumption, which defines what technology stack can be used. Indeed, many IoT communication technologies, among which the mostly renowned CoAP application protocol, rely on the UDP transport protocol — a connectionless protocol that gives no guarantee on the delivery of messages, but allows one to limit message exchanges and, by extension, energy and bandwidth consumption. Since Jolie assumes lossless communications, the inclusion of connectionless protocols in the language requires careful handling to prevent misbehaviors;
- *point-to-point vs. publish-subscribe* — The premise of the Jolie language is to provide communication constructs that do not depend on a specific technology. To do so, the language assumes a point-to-point communication abstraction, which is common to many protocols like HTTP and CoAP. However, to integrate the MQTT protocol in Jolie, we need to model Jolie point-to-point semantics as MQTT publish-subscribe operations. Indeed, Jolie already provides language constructs usable with many communication protocols, hence the less disruptive approach is to use the same constructs, which are designed for a

point-to-point setting, also for MQTT. This requires to find for each point-to-point construct a corresponding effect in the publish-subscribe paradigm. The final result is that the execution of a given Jolie behavior is similar under both point-to-point and publish-subscribe technologies.

5 Supporting Constrained Application Protocol in Jolie

The *Constrained Application Protocol* (CoAP) [4, 5] is a specialized web transfer protocol for constrained scenarios where nodes have low power and networks are lossy. The goal of CoAP is to import the widely adopted model of REST architectures [23] into an IoT setting, that is, optimizing it for machine-to-machine applications. In particular, like HTTP, CoAP makes use of GET, PUT, POST, and DELETE methods. Following the RFC [5], CoAP is implemented on top of the UDP transport protocol [24], with optional reliability. Indeed, CoAP provides two communication modalities: a reliable one, obtained by marking the message type as confirmable (CON), and an unreliable one, obtained by marking the message type as non confirmable (NON).

As an example, we consider a scenario with a controller, programmed in JIoT, that communicates with one of many thermostats in a home automation scenario. Thermostats are accessible at the generic address `"coap://localhost/##"` where `"##"` is a two-digit number representing the identifier of a specific device. Each thermostat accepts two kinds of interactions: a GET request on URI `"coap://localhost/##/getTemperature"`, that returns the current temperature, and a POST request on URI `"coap://localhost/##/setTemperature"`, that sets the temperature of the HVAC (heating, ventilation, and air conditioning) system.

We comment below Listing 3, where we report the code of a possible JIoT controller that interacts with a specific thermostat.

Our scenario includes two CoAP resources, referred to as `"/getTemperature"` and `"/setTemperature"`. We model them in JIoT at lines 4–7 of Listing 3, by defining the **`interface`** `ThermostatInterface`, which includes a **`RequestResponse`** operation `getTmp`, representing resource `"/getTemperature"`, and a **`OneWay`** operation `setTmp`, representing resource `"/setTemperature"`. By default, we map operation names to resource names, hence in our example we would need resources named `"/getTmp"` and `"/setTmp"`, respectively. However one can override this default by defining the coupling of resource names and operations as desired. This allows programmers to use interfaces as high level abstractions for interactions, while the grounding to the specific case is done in the deployment. Here we purposefully choose to use operation names that differ from resource names to underline that the two concepts are related but loosely coupled. On the one hand the coupling between the name of the resource and the operation can be seen as a way of quickly binding actions exposed by the CoAP server with operations. On the other hand decoupling resource names and operations permits to handle more complex deployments where, for instance, a single operation responds for different resources. At lines 9–25 we define an **`outputPort`** to interact with the

```
1 type getTmpType: void { .id: string }
2 type setTmpType: int { .id: string }
3
4 interface ThermostatInterface {
5  RequestResponse: getTmp( getTmpType )( int )
6  OneWay: setTmp( setTmpType )
7 }
8
9 outputPort Thermostat {
10 Location: "datagram://localhost:5683"
11 Protocol: coap {
12  .osc.getTmp << {
13   .messageCode = "GET",
14   .contentFormat = "text/plain",
15   .messageType = "CON",
16   .alias = "/%!{id}/getTemperature"
17  };
18  .osc.setTmp << {
19   .messageCode = "POST",
20   .messageType = "CON",
21   .alias = "/%!{id}/setTemperature"
22  }
23 }
24 Interfaces: ThermostatInterface
25 }
26
27 main {
28 getTmp@Thermostat( { .id = "42" } )( temp );
29 if ( temp > 27 ) {
30  setTmp@Thermostat( 24 { .id = "42" } )
31 } else if ( temp < 15 ) {
32  setTmp@Thermostat( 22 { .id = "42" } )
33 }
34 }
```

Listing 3. JIoT controller communicating over CoAP/UDP.

`Thermostat`. At line 10 we specify the **`Location`** of the thermostat. Recalling that the scheme of the resources of the thermostats is `"coap://localhost/##/..."`, we define the **`Location`** of the port using the UDP `"datagram://"` protocol, followed by the first part of the resource schema `"localhost"` and the UDP port on which it accepts requests. Here we assume thermostats to use CoAP standard UDP port, which is `"5683"`. Note that, in the **`Location`**, we do not define the address of a specific thermostat, e.g., `"datagram://localhost:5683/42"`. On the contrary, we just specify the generic address to access thermostats in the system, while the specific binding will be done at runtime, thanks to the `.alias` parameter of the `coap` protocol, described later on.

At line 11 we define `coap` to be the protocol used by the **`outputPort`**. At lines 12–22 we specify some parameters of the `coap` protocol — this matches the standard way in which Jolie defines parameters for **`Protocol`**s in ports. Here, we follow the methodology presented in [25] for the implementation of the HTTP protocol in Jolie — indeed CoAP adopts HTTP naming schema and resource interaction methods. In particular, we draw from [25] the parameter prefix `.osc`, whose name is the acronym of "operation-specific configuration" and which is used for configuration parameters related to a specific operation.

In the example, we define `.osc` parameters for both operations `getTmp` and `setTmp`. At line 13 we specify that the CoAP verb used for operation `getTmp` is `"GET"`. At line 14 we define, using the `.contentFormat` parameter, that the encoding of the payload of the message is in text format. Other accepted values for the `.contentFormat` parameter are `"json"` and `"xml"`. Marshalling and un-marshalling is automatic and transparent to the programmer. This feature is enabled by the structure of Jolie variables, which are always tree-shaped, hence they can be easily translated into representations based on that shape. At line 15 we set the `.messageType` parameter to `"CON"`, that stands for confirmable. Accepted values for the `.messageType` parameter are confirmable and not confirmable (`"NON"`), the latter being the deafault value. In the first case the sender will receive an acknowledgment message from the receiver, in the second case it will not. At line 16, following the practice introduced in [25], we specify that `getTmp` `aliases` a resource whose path concatenates a static part, given by the **`Location`**, and the instantiation of the template `"/%!{id}/getTemperature"` provided by protocol parameter `.alias`. The template is instantiated using values from the parameter of the operation invocation in the behavior, e.g., value `42` at line 28[1]. Hence, the interpretation of the declaration at line 16 is that, when invoking operation `getTmp` at runtime, the element `id` of the invocation will be removed from the payload and used to form the address of the requested resource. The aliasing for operation `setTmp` (line 21) is similar to that of `getTmp`, while the operation uses verb `POST`. Since here the `.contentFormat` parameter is omitted, the default `"text/plain"` is used.

To conclude, we briefly comment the runtime execution of the example, described in the behavior at lines 28–33. At line 28 the controller invokes operation `getTmp`. Being an outgoing **`RequestResponse`**, the invocation defines on which port to perform the request (`Thermostat`) and presents two pairs of round brackets: the first contains the data for the request, the second points to the variable that will store the received response. Recalling the aliasing defined at line 16, at line 28 we define the value of element `id = 42`, thus the URI of the resource invoked at runtime is `"coap://localhost/42/getTemperature"`. Notably, in the example we hard-coded the `id` of the device, however in a more realistic setting the value of `id` would be retrieved dynamically, e.g., as an execution parameter, from a configuration file or from a database. Once received, the response from thermostat `42` is assigned to variable `temp`. The example concludes with a conditional

[1] In Jolie the dot `.` defines path traversals inside trees. Hence, the notation `{.id = 42}` indicates a tree with an empty root and a subnode called `id`, whose value is `42`.

in which, if the temperature is above `27` degrees (line 29), the thermostat is set to lower room temperature to `24` degrees, while, if the temperature lies below `15` degrees, the thermostat is set to raise the temperature to `22` degrees.

Dually to **`outputPort`**s, **`inputPort`**s allow the programmer to specify inbound communications. The parameters described above are valid also for **`inputPort`**s, with the only difference that `messageType` works only for **`RequestResponse`**s, and specifies whether the communication of the reply is reliable or not. Note that, concerning the `.alias` parameter, the template is instantiated using the address of the incoming communication and the values are inserted among the elements of the payload.

6 Supporting Message Queue Telemetry Transport in Jolie

Message Queue Telemetry Transport (MQTT) [6, 7] is a publish/subscribe messaging application protocol built on top of the TCP transport protocol.

A typical publish/subscribe interaction pattern can be diagrammatically represented as in Fig. 1 where:

1. a Subscriber subscribes to topic (a) at some Broker;
2. a Publisher publishes a message to topic (a) at the same Broker;
3. the Broker forwards the message to topic (a) to the Subscriber.

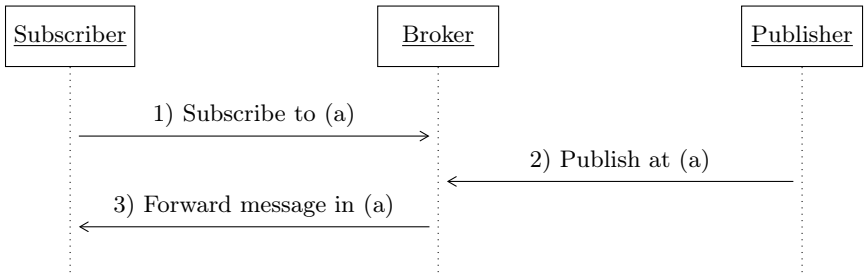

Fig. 1. Typical publish/subscribe interaction pattern.

More generally, messages published on a topic are forwarded to all current subscribers for the topic.

On top of the basic mechanism of publish/subscribe, MQTT defines three levels of quality of service (QoS) for the delivery of each message published by a publisher. QoS levels determine whether messages can be lost and/or duplicated. Concretely, QoS levels are as follows:

– *At most once* — the message can be lost, no duplication can occur.
– *At least once* — delivery of the message is guaranteed, but duplication may occur.

```
1  interface TemperatureInterface {
2   OneWay: receiveTemperature( string )
3  }
4
5  inputPort CollectorPort3 {
6   Location: "socket://localhost:8050"
7   Protocol: mqtt {
8    .broker = "socket://localhost:1883"
9   }
10  Interfaces: TemperatureInterface
11 }
12
13 main {
14  ...
15  receiveTemperature( data );
16  ...
17 }
```

Listing 4. Code of the Collector Example, revised for MQTT.

– *Exactly once* — delivery of the message is guaranteed and duplication cannot occur.

To present how we model the MQTT protocol in JIoT, we first detail the simpler case of **OneWay** communications in Section 6.1. Then, we address the more complex case of **RequestResponse** communications in Section 6.2. Notably, our modeling of end-to-end communications over a publish/subscribe channel is independent from JIoT, i.e., it is a general reference on how to implement one-way and request-response communications on top of any publish/subscribe channel.

6.1 One-Way Communications in MQTT

We first consider the case of inbound communications and then the case of outbound communications.

We exemplify **OneWay** inbound communications using the example in Listing 4, which is a revision of the example in Listing 2 by omitting the ports `CollectorPort1` and `CollectorPort2` and by adding an MQTT **inputPort** named `CollectorPort3`.

As expected, the program behavior and the structure of the **inputPort** are unchanged. Main novelties are:

– the used **Location** (line 6) has the prefix `"socket://"` (as seen in the HTTP port) since MQTT relies on TCP transport protocol;
– the used **Protocol** (line 7) is `mqtt`;
– the `.broker` protocol parameter (line 8), which is compulsory when the `mqtt` protocol is used in **inputPort**s, specifies the address of the Broker.

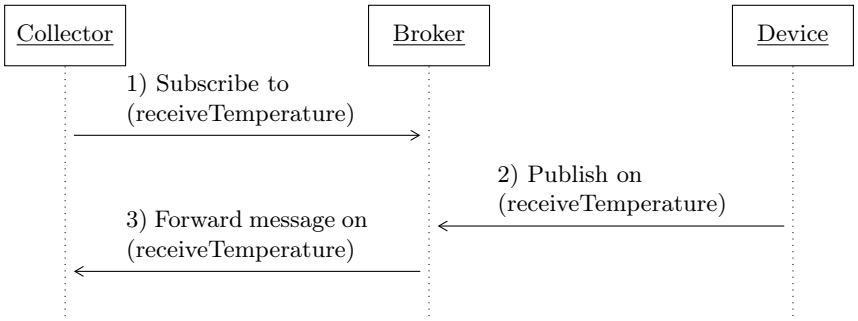

Fig. 2. Representation of the example in Listing 4.

From the perspective of the programmer, the syntax and the effect of the communication primitive are the same as in Listing 2. However, we actually exchange several messages to capture that effect in MQTT, as shown in Fig. 2.

Beyond defining such message exchanges, we also need to decide how to identify the topic on which the message exchange is performed.

Regarding the message exchanges, from the point of view of the programmer, an inbound `OneWay` communication receives a datum from the communication partner. To obtain the same effect using the publish/subscribe paradigm, one has first to subscribe at the Broker to the chosen topic and then wait to receive a message on that topic, forwarded by the Broker. How topics are selected will be detailed later on. The execution of a reception on a `OneWay` operation comprises two actual communications: a subscription from the program to the Broker and a message delivery in the opposite direction. However, subscription to topics and the execution of a message reception are logically separated and can be done at different moments. Indeed, the subscription is performed when the JIoT program is launched for all operations present in MQTT `inputPort`s. This choice is more in line with the expected behavior of Jolie programs — and of Service-Oriented programs in general — where messages to operations whose reception statements are not yet enabled are stored until the actual execution of the reception. Here, if the subscription is performed along with the execution of the `OneWay` operation, previous messages could be no more available. In JIoT, the compulsory parameter `.broker` is needed precisely to know the address at which the subscription needs to be performed. The address for the delivery of the actual message is the usual `Location` of the `inputPort`.

Regarding the selection of topics, similarly to what done for CoAP resources, in MQTT by default we map JIoT operations to topics, otherwise we use the `.osc` parameter `.alias` to loose the coupling between operations and topics. We remark that `.alias` parameters in `inputPort`s have a different behavior in MQTT with respect to HTTP and CoAP. In CoAP the name of the resource extracted from the received message is used to derive the instantiation of the `.alias` template. The values resulting from the match are then inserted among the elements of the payload before storing it in the target variable `data`. In-

```
1  interface ThermostatInterface {
2   OneWay: setTmp( TmpType )
3  }
4
5  outputPort Broker {
6   Location: "socket://localhost:1883"
7   Protocol: mqtt {
8    .osc.setTmp << {
9     .format = "raw",
10    .QoS = 2, // exactly once QoS
11    .alias = "%!{id}/setTemperature"
12   }
13  }
14  Interfaces: ThermostatInterface
15 }
16
17 main {
18  ...
19  setTmp@Broker( 24 { .id = "42" } );
20  ...
21 }
```

Listing 5. Example of outgoing MQTT `OneWay` communication.

stead, in MQTT, the `.alias` parameter is used to identify the topic for subscription. For example, in Listing 4, one could add the `Protocol` parameter `.osc.receiveTemperature.alias = "temperature"` to specify that the selected topic for operation `receiveTemperature` is `"temperature"`. Note that, since there is no outgoing data, templates in MQTT `inputPort`s, such as `"temperature"` in the example, are constants (we require all such constants defined within the same `inputPort` to be distinct). Having only constant aliases is not a relevant limitation in the context of IoT, where topics are mostly statically fixed. Addressing this limitation without disrupting the uniformity of the Jolie programming model is not trivial and it is left as future work.

To conclude the mapping of `OneWay` operations in MQTT, we consider here the case of outbound operations, exemplified in Listing 5. Outgoing `OneWay` operations simply cause the publication of the value passed as the parameter of the invocation (line 19) at the Broker. The address of the Broker is defined by the `Location` (line 6) of the `outputPort` `Broker`. The topic is derived from the name of the `operation` and the parameter of the invocation, using protocol parameter `.alias` as usual. Being an MQTT publication, we specify the `.QoS` protocol parameter (line 10), which selects the QoS level "Exactly once" for the operation `setTmp`. Similarly to what we have done in CoAP with the `contentFormat` protocol parameter, we define in `.format` the encoding of the message payload, in this case a "raw" stream of bytes.

```
interface ThermostatInterface {
 RequestResponse: getTmp( TmpType )( int )
}

outputPort Broker {
 Location: "socket://localhost:1883"
 Protocol: mqtt {
  .osc.getTmp << {
   .format = "raw",
   .QoS = 2, // exactly once QoS
   .alias = "%!{id}/getTemperature",
   .aliasResponse = "%!{id}/getTempReply"
  }
 }
 Interfaces: ThermostatInterface
}

main {
 ...
 getTmp@Broker( { .id = "42" } )( temp );
 ...
}
```

Listing 6. JIoT controller communicating over MQTT.

6.2 Request-Response Communications in MQTT

To discuss `RequestResponse` communications, let us consider the example in Listing 3, revised in Listing 6 by replacing the CoAP protocol with MQTT. We omit `OneWay` communications and concentrate on the outbound `RequestResponse`. Afterwards, we will also discuss the dual inbound `RequestResponse`.

Syntactically, the main novelty with respect to the `outputPort` in Listing 5 is the addition of `Protocol` parameter `.aliasResponse`. This parameter specifies the name of the topic where the receiver will publish its response.

From the point of view of the programmer, an outbound `RequestResponse` is composed of an outgoing communication followed by an inbound reply. The outgoing communication is implemented using the approach already seen for `OneWay` communications, i.e., using the `.alias` `Protocol` parameter to identify the topic. Then, one has the issue of relating the outgoing request with its reply. Many standard point-to-point communication technologies, such as HTTP/TCP and the already discussed CoAP/UDP, support request-response communications by defining means to link a given outgoing request to its reply. MQTT does not provide dedicated means to do such a linking. Thus we specify topics for both the request and the response, but it is responsibility of the programmer to ensure that corresponding topics are used in the client and in the server. A possibility for the programmer is to send the topic for the response inside the

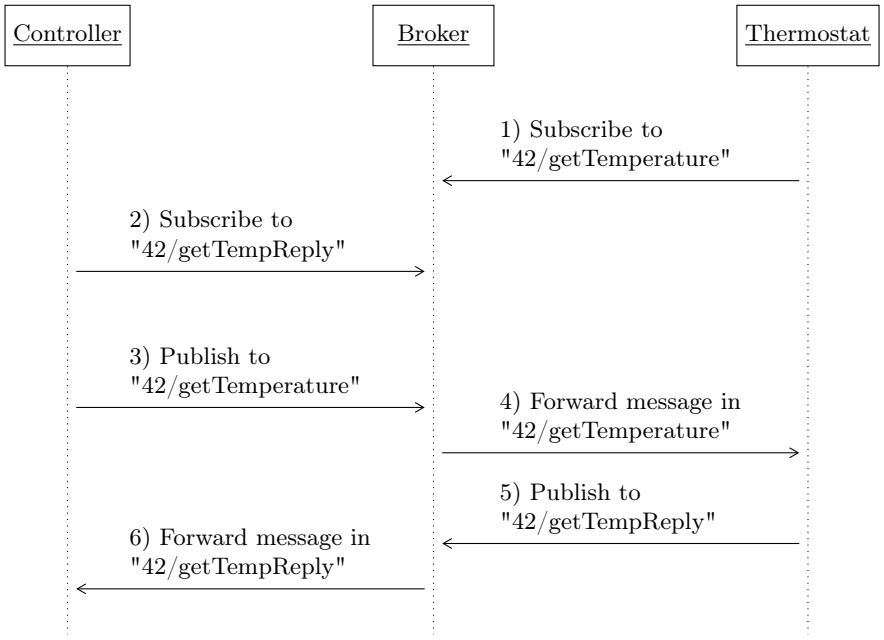

Fig. 3. Interaction in the temperature automation example in MQTT.

payload of the request message. We identify the topic for the reply with the `.aliasResponse` **`Protocol`** parameter. Like for `.alias` parameters, the template of the `.aliasResponse` parameter is instantiated using the content of the message sent in the behavior. For example, in Listing 6, we use `.id` in line 20 to obtain `"42/getTemperature"` and `"42/getTempReply"`, respectively the publication and reply topics.

We can now describe the pattern of interactions that we use to implement the outgoing **`RequestResponse`** communication at line 20 in Listing 6. As a reference, the pattern of interactions is depicted in the left part of Fig. 3. We will describe the right part later on, after having introduced inbound request-response communications.

First, the controller subscribes to the reply topic `"42/getTempReply"` at the Broker. Then, the controller sends to the Broker the request message on topic `"42/getTemperature"`. The execution of the **`RequestResponse`** terminates when the Broker forwards the reply received on topic `"42/getTempReply"` to the controller.

Differently from inbound **`OneWay`** communications, here we do not subscribe to the reply topic when the program is launched. Indeed, it would be useless since no relevant message can arrive on this topic before the controller sends its message to the Broker, and anticipating the subscription would complicate the usage of runtime information in templates.

```
1  interface ThermostatInterface {
2   RequestResponse: getTmp( TmpType )( TmpType )
3  }
4
5  inputPort Thermostat {
6   Location: "socket://localhost:9000"
7   Protocol: mqtt {
8    .broker = "socket://localhost:1883";
9    .osc.getTmp << {
10    .format = "raw",
11    .alias = "42/getTemperature",
12    .aliasResponse = "42/getTempReply"
13   }
14  }
15  Interfaces: ThermostatInterface
16 }
17
18 main {
19  //        ↓ receive the temperature and store it under the root of temp
20  getTmp( temp )( temp ){
21  //                ↑ update the content of temp and send it back as response
22  }
23 }
```

Listing 7. JIoT thermostat communicating over MQTT.

To exemplify inbound **RequestResponse** communications, we assume that the thermostat in our example is programmed in JIoT. We report its code in Listing 7.

At line 11 in Listing 7, the `.alias` parameter `"42/getTemperature"` must be defined statically, as required for **inputPort**s. When the thermostat program is launched, it subscribes to topic `"42/getTemperature"`. When a message on this topic arrives, the payload (empty in this case) is passed to the behavior. The body of the **RequestResponse** (line 20) is executed to compute the return value. Finally, the return value is published on the reply topic `"42/getTempReply"`, as specified by `osc` parameter `.aliasResponse`. While in this example the parameter `.aliasResponse` is statically defined, our implementation supports the definition of dynamic `.aliasResponses` as in **outputPort**s (e.g., as seen in Listing 6).

We now summarize the exchange between the controller and the thermostat (left part of Fig. 3):

1. when the thermostat is started, it subscribes to topic `"42/getTemperature"` at the Broker;
2. when the outgoing **RequestResponse** is executed, the controller subscribes to topic `"42/getTempReply"` at the Broker;
3. the controller publishes the request message to topic `"42/getTemperature"`;

4. the Broker forwards the message in topic `"42/getTemperature"` to the thermostat;
5. the thermostat publishes the computed response at topic `"42/getTempReply"`;
6. the Broker forwards the message on topic `"42/getTempReply"` to the controller.

We remark that `RequestResponse` operations in Jolie are meant to be end-to-end communications. To ensure this in a publish/subscribe setting while using the approach above, one has to ensure that no other participant subscribes to the selected topics, which essentially act as namespaces.

7 Implementation

To illustrate the structure of our implementation, in Section 7.1 we discuss how media and protocols are separated from the Jolie interpreter and available as independent libraries. Then we describe the highlights of the implementation of UDP and CoAP in Section 7.2 and of MQTT in Section 7.3.

7.1 Programming a Jolie Extension

In Jolie the implementations of the supported application and transport protocols are independent. This enables the composition of any transport protocol with any application protocol. Concretely, the Jolie language is written in Java and provides proper abstract classes that represent application and transport protocols. Each protocol is obtained as an implementation of the corresponding abstract classes. Each implementation is a separated library which is loaded only if the protocol is used. This expedites the integration of new protocols in the language.

To better illustrate this structure, we report in Fig. 4 a conceptual representation of the call flow that originates from the execution logic of the language and interacts with the external libraries present in a given installation. The flow starts from the Execution Engine, which interprets Jolie commands, and which is the originator of the communication flows. This is represented by arrow (0) from the Execution Engine. From there, the call reaches the Communication Core, which implements the generic logic of channel creation, in turn relying on the pairing of a medium and a protocol. In the interpreter, this division is generalized with abstract factories for media and protocols. At runtime, the Communication Core proceeds (arrows (1)) to load the medium factory requested in the call from the Execution Engine — in the figure we assume this is Socket — and, from that, it obtains an implementation of the actual logic of TCP/IP channels, split between a channel class, to handle outbound communications, and a listener class, for inbound communications. Finally, the Communication Core associates (arrows (2)) a protocol to the obtained medium. The flow is similar to that of media: the Communication Core loads the protocol factory requested in the call from the Execution Engine — in the figure we assume this is HTTP — and, from that, it obtains an object that implements the logic of the HTTP protocol.

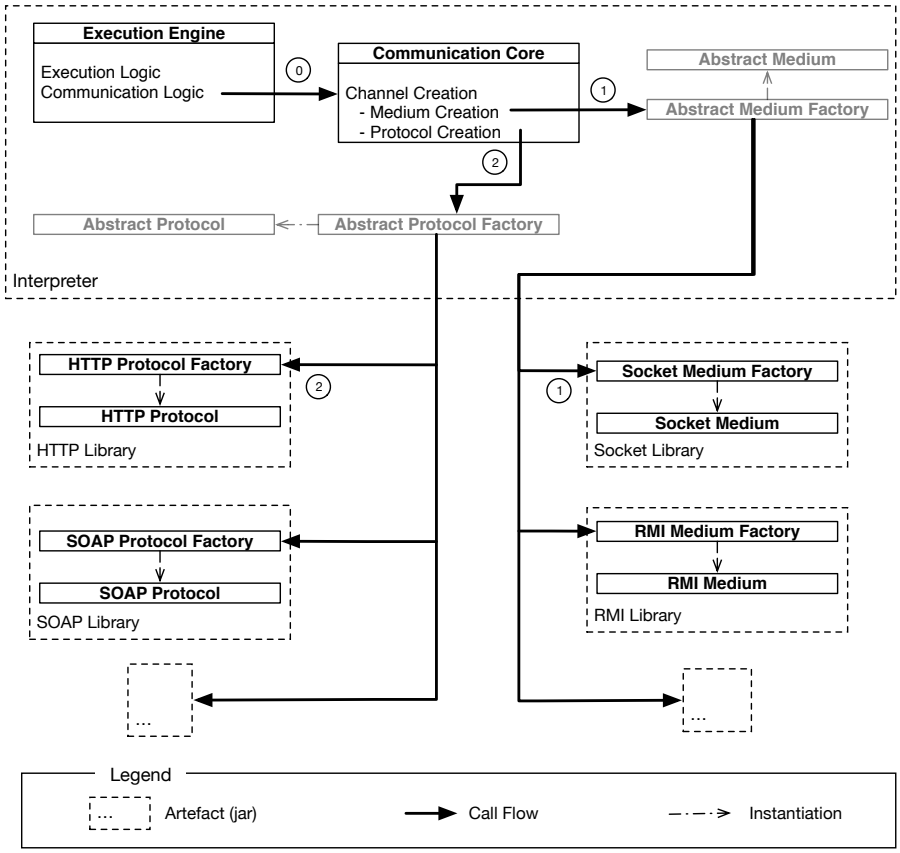

Fig. 4. Conceptual representation of the call flow among the Jolie interpreter and its communication libraries.

7.2 Implementation of CoAP/UDP in Jolie

Since by specification the CoAP protocol relies on the UDP medium protocol, in order to integrate CoAP in Jolie we also had to integrate the UDP medium. As described in Section 7.1, this entailed the creation of two new libraries for the Jolie interpreter: a medium library for UDP and a protocol library for CoAP.

We remark that, since UDP and CoAP are independent libraries, our implementation of UDP can also be used to support other protocols relying on UDP, such as MQTT-SN [26]. The implementation of UDP consists in a listener and a channel class, both based on the Netty framework [27]. Since the structure expected by Jolie and the one provided by Netty are similar, the integration of UDP is smooth. An interesting point is that exceptions raised by Netty are captured and transformed into Jolie exceptions. These exceptions are notified to the application protocol, which can either manage them or raise them at the level of the behavior of the Jolie program.

The implementation of the CoAP library consists in a class taking care of encoding and decoding the message abstraction of Jolie, namely the CommunicationMessage, into a CoAP formatted one. A second class, handling the encoding and decoding of a CoAP message into a buffer of bytes, is based on the work done in nCoAP [28], an open source project providing a CoAP implementation for Java, based itself on Netty.

CoAP supports request-response communications and, in particular, CoAP messages include fields *i*) to specify at which address the reply is expected and *ii*) to match a reply with a previous request. Hence, the implementation of `RequestResponse` communications in CoAP is sound also with a transport protocol which is not connection-oriented, such as UDP. This would be a problem for protocols that do not provide such a facility, such as HTTP, which is indeed not commonly used over UDP.

Notably, Jolie comes with a formal semantics (in terms of a process calculus) [29], which enables to rigorously reason on the behavior of Jolie programs. This has been instrumental in the evolution of the language, e.g., to specify and prove properties on the fault handling mechanisms of the language [30] or to correctly implement sessions [31] based on correlation mechanisms [32]. The semantics in [29] only considers reliable communications and needs to be extended to also cover the unreliable case. We do not report here on this topic, since it is not central for the purpose of this paper.

7.3 Implementation of MQTT in Jolie.

By specification, MQTT relies on the TCP/IP protocol, already implemented in Jolie. This means that, theoretically, the implementation of MQTT would have only entailed the creation of a dedicated MQTT protocol library. However, as detailed in Section 7.1, Jolie assumes an end-to-end communication pattern where the caller initiates the creation of a communication channel with a server, which in turn expects such inbound requests. For this reason, given a certain medium, `inputPort`s and `outputPort`s use a medium-specific implementation of, respectively, a listener class and a channel class. This pattern, separating listeners from channels, does not apply to publish/subscribe protocols, where both the subscriber and the publisher need to establish a connection with the broker. In our implementation, we mediated between the two approaches with a Publish-Subscribe medium, which is essentially a wrapper implementing the logic of Publish-Subscribe message handling on any other point-to-point medium available (TCP socket in the case of MQTT) to the interpreter. Although we strove to separate the concerns between the Jolie interpreter and this new Public-Subscribe channel, we had to introduce a minimal update into the Jolie Communication Core so that it could choose between the standard end-to-end media and the new wrapper.

The MQTT protocol class both encodes and decodes messages and implements the QoS policies of the MQTT standard. Concretely, as for CoAP, we based the implementation of MQTT on Netty [27]. The main difficulty in the implementation of the protocol is the definition of the message patterns needed

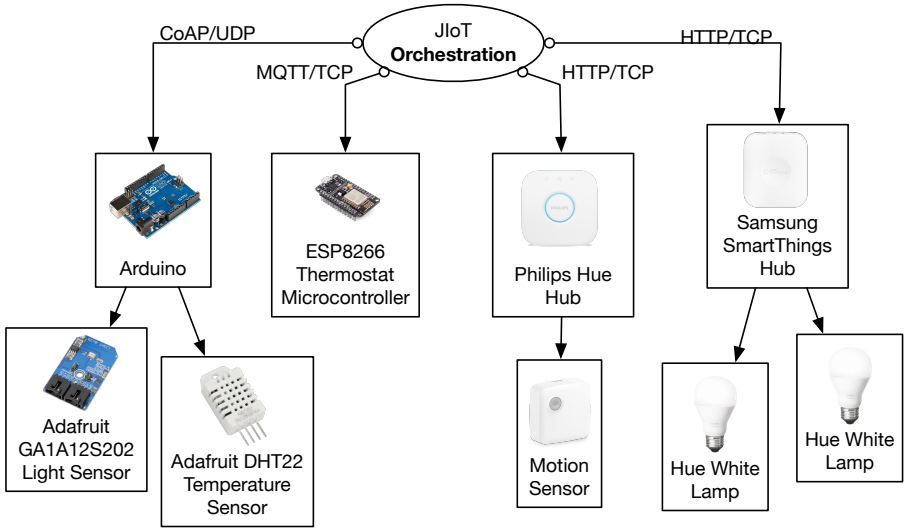

Fig. 5. Conceptual overview of the home automation case study.

to implement `OneWay` and `RequestResponse` communications, which have been described in Section 6. Beyond being invoked when operations are executed, the MQTT class is also invoked when the program is started, to perform port initialization. In particular, this is when subscriptions to topics identified in `inputPort`s are performed (along with the related connections to the brokers).

8 Case Study

In this section, we detail the programming of a home automation case study with JIoT. We remark that the techniques presented in this case study are not specific to home automation and can be used in any setting where heterogeneous IoT technology stacks need interact. The use case is peculiar as new edge devices can be included in the system at runtime. The source code of the system is released under the GPL v.3 license and available at [19]. We report in Fig. 5 a schematic overview of the case study, where Cloud nodes and mid-tier controllers (represented by the element labeled "JIoT Orchestration" in Fig. 5) are programmed in JIoT and orchestrate the behavior of a number of heterogeneous edge devices (whose low-level programming is omitted here):

- *Philips Hue Hub*: a hub to control the Philips Hue smart home devices;
- *Philips Hue White Lamps*: connected to the hub above;
- *Samsung SmartThings Hub*: a hub to control devices following the SmartThings specification [33];
- *Samsung SmartThings Motion Sensor*: connected to the hub above and used as a presence sensor;

– *Arduino Uno*: a general-purpose microcontroller;
– *Adafruit GA1A12S202 Analog Light Sensor*: connected to the Arduino above;
– *Adafruit DHT22 Temperature Sensor*: also connected to the Arduino above;
– *ESP8266*: a microcontroller to manage a pre-existing thermostat.

The case study combines commercial solutions — e.g., the Philips Hue Hub and the Hue White Lamps system where the Lamps are controlled by the Hub — with custom ones — these span from sensors directly connected to a board, as it happens for the Adafruit DHT22 temperature sensor, to solutions that integrate a pre-existing hardware, like the ESP8266 that manages a pre-existing thermostat. As illustrated in Fig. 5, this heterogeneity of devices provides for a comprehensive scenario where we need JIoT programs that use different application and transport protocols. In particular, Philips and Samsung Hubs communicate with the orchestrator over HTTP/TCP, the Arduino over MQTT/TCP, and the ESP8266 over CoAP/UDP.

In the use case we build a simple logic providing two functionalities: lighting and temperature system control. The lighting system turns on the lights when the motion sensor detects someone at home and the outdoor luminosity is below some threshold. The temperature control checks the temperature and turns on the heating system when the temperature is below some threshold. The threshold has different values depending on whether someone is at home or not.

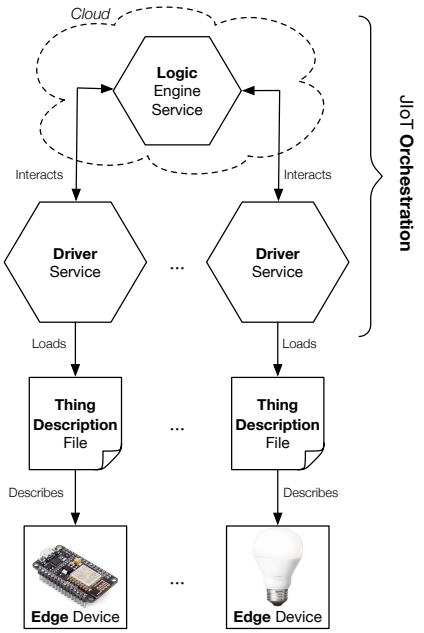

Fig. 6. Scheme of orchestration in the case study.

8.1 Structure of the Orchestration

We now describe the structure of the orchestration in the case study, which is illustrated in Fig. 6. The orchestration is composed of multiple JIoT programs. From top to bottom of Fig. 6, the `LogicEngine` contains the general logic of system control (i.e., the one that collects the data from sensors and coordinates the execution of the actuators in the system). Since the `LogicEngine` interacts with a multitude of mid-tier devices, its natural deployment is in the Cloud, where it is possible to scale it according to the number of managed devices and the load of computation. At the mid-tier level we have JIoT `Drivers`. Each `Driver` interacts with a specific edge device and it is deployed in a mid-tier machine in the proximity of the controlled edge device.

8.2 Thing Descriptors

In the case study, the `Drivers` are statically configured to manage a single fixed device using a JSON-LD 1.1 (that stands for JSON Linkage Data) configuration file [34]. The choice of JSON-LD is not mandatory, but it has the benefit of following the standard W3C Web of Things [22] definition of Thing Description (TD). This makes our `Drivers` already compliant with other WoT frameworks, simplifying future integrations with other WoT systems.

While discussing the full structure of TD is out of the scope of this paper, we present in Figs. 7 and 8 examples of TDs used in our case study. In Fig. 7 we report the TD for the DHT22 temperature sensor. For each device the JSON-LD file specifies whether it is a sensor or an actuator (key `"type"`) and provides a textual description (key `"description"`) and its name (key `"name"`). Each TD provides a list of properties (key `"properties"`) that can be read. Each property is described by the property identifier, `"temperature"` in our example. The property identifier has various sub-elements describing it. In our example we use just key `"label"` to describe the unit of measure.

```
{
  "type": "sensor",
  "description": "Thing uses JSON-LD 1.1 serialization",
  "name": "Adafruit DHT22 Temperature Sensor",
  "properties": [
    {
      "temperature": {
        "label": "Celsius"
      }
    }
  ]
}
```

Fig. 7. Adafruit DHT22 Thing Descriptor.

```
{
  "type": "actuator",
  "description": "Thing uses JSON-LD 1.1 serialization",
  "name": "Philips Hue White Lamp",
  "actions": {
    "toggleLight": {
      "description": "Turn on or off the lamp."
    }
  }
}
```

Fig. 8. Philips Hue White Lamp Thing Descriptor.

JSON-LD configuration files for MQTT and HTTP devices are similar. Also configuration files for sensors and actuators are similar. As an example, we report in Fig. 8 the configuration file for Philips Hue White Lamps. There the main differences with respect to the previous TD (Fig. 7) are:

- the `"type"` is now `"actuator"`;
- the key `"actions"` replaces the key `"properties"`;
- the key `"description"` is used also to describe the single action.

In principle a TD can describe multiple properties belonging to a group of one or more edge devices controlled by the same `Driver`. For simplicity, here we have one TD for each edge device and, correspondingly, one `Driver` that controls one edge device. We also assume that each sensor provides one property.

8.3 System Deployment

Deployment-wise, JIoT provides a vast choice regarding what technology stack to use between the `LogicEngine` and the `Drivers`. Moreover, since both programs are developed in JIoT, it is easy to change their deployment, switching to the technology stack that best suites a given scenario (e.g., HTTP, to exploit caching, or binary formats like SODEP [17], to limit bandwidth usage). Here, we choose to use the HTTP/TCP stack to make our system compatible with the majority of existing third-party solutions [9]. However, different technology stacks fit different purposes. The benefit of JIoT is that programmers can re-use the same software components adapting their deployment to the desired communication stacks. For example, if our goal was to be natively compatible with other JavaScript IoT frameworks, we could have used the JSON-RPC binary protocol; if we wanted to deploy our system as part of a Service-Oriented Architecture [14], we could have used the SOAP protocol.

While JIoT-to-JIoT deployment is flexible, the deployment towards edge devices is defined by the technology supported by the edge device. Concretely, in our case study each `Driver` communicates with its edge device using (one of) the protocol(s) supported by the latter.

8.4 Components Behavior

When started, a `Driver` loads the TD configuration file of its edge device. Then it registers itself to the `LogicEngine`. In the registration, it sends the information retrieved from the TD, enriched with two additional pieces of information: the address where the edge device can be contacted — i.e., the `Driver` location — and the identifier of the user to which the edge device belongs. Once registered, the `Driver` acts as a forwarder between the `LogicEngine` and the edge device.

The `LogicEngine` runs on the Cloud and manages a number of sensors and actuators. More precisely the `LogicEngine` has one running session for each user (distinguished according to the user identifier), managing all her sensors and actuators. Each session is associated with an array of devices that can be scanned to find the location of `devices` with specific properties and interact with them; e.g., at lines 10–26 of Listing 8 the procedure `getTemperature` of the `LogicEngine`, computing the average temperature recorded by the sensors of one user.

```
1  interface driverInterface {
2   RequestResponse: engineRequest( request )( response )
3  }
4
5  outputPort Driver {
6   Protocol: http
7   Interfaces: driverInterface
8  }
9
10 define getTemperature {
11  sum = 0 ;
12  n = 0 ;
13  for ( device in devices ) {
14   if( device.type == "sensor" &&
15     is_defined( device.properties.temperature ) ) {
16    Driver.location = device.driverLocation ;
17    request.operationName = "getTemperature" ;
18    engineRequest@Driver( request )( response ) ;
19    sum = sum + response.deviceResponse ;
20    n++
21   }
22  } ;
23  if( n!=0 ) {
24   temperature = sum / n
25  }
26 }
```

Listing 8. `LogicEngine` `Driver` **`outputPort`** and `getTemperature` procedure.

Briefly, procedure `getTemperature`:

- scans the `devices` structure (line 13) containing all registered `Drivers`;
- selects those whose `type` is `"sensor"` and have a property (under the sub-structure `properties`) named `temperature`. Note how Jolie tree-shaped variables ease the exploration of structured data; in this case the one sent by the `Drivers` at registration time (and read from their associated JSON-LD file);
- it dynamically sets (line 16) the location of **`outputPort`** `Driver` (lines 5-8) to contact the selected `Driver`;
- it sets the request operation to `getTemperature` (line 17);
- it retrieves the temperature sensed by the edge device controlled by the selected `Driver`, invoking it through operation `engineRequest`;
- it aggregates the sensed temperature in variable `sum` and keeps track of the number of requests in variable `n` (lines 19–20);
- it computes the mean `temperature` (lines 23–25).

The procedures that calculate the mean of the sensed external luminosity and the one to check the presence of people at home are similar to the one in Listing 8, except that the searched properties are `light` in the first case, and `motion` in the second.

We report in Listing 9 one of the procedures managing the actuators, specifically the one used to set the temperature. The main difference with respect to the logic in Listing 8 is that procedure `setTemperature`:

- selects the `devices` whose `type` is `"actuator"` (line 3);
- sets the request operation to `"setTemperature"` and passes the value in variable `comfortTemperature` as parameter of the request (lines 6-7).

Note that the operation called on the `Driver` is `engineRequest` both in Listing 8 and Listing 9. This support the extension of the `LogicEngine` with new procedure definitions that implement a given goal without requiring to change the interface between the `LogicEngine` and the `Drivers`. In turn, a request with

```
1  define setTemperature {
2   for ( device in devices ) {
3    if( device.type == "actuator" &&
4      is_defined( device.properties.temperature ) ) {
5     Driver.location = device.location ;
6     request.operationName = "setTemperature" ;
7     request.deviceRequest = comfortTemperature ;
8     engineRequest@Driver( request )( response )
9    }
10  }
11 }
```

Listing 9. `LogicEngine` setTemperature procedure.

the same `operationName` (e.g., `"setTemperature"`) triggers different behaviors in different `Drivers`, as each implements the specific logic of interaction with its associated edge device.

8.5 Cloud Deployment

We conclude this section by describing the Cloud deployment of the `LogicEngine`, which is containerized using Docker [35]. The container is deployed automatically into an Amazon Web Service cluster via the Docker Swarm manager [36]. The `LogicEngine` is deployed in the worker cluster, allowing the manager to balance the load of requests. We report in Listing 10 the content of the Dockerfile used to deploy the `LogicEngine`.

At line 1 we declare the starting image for the container, which is the lightweight Linux Alpine distribution with OpenJDK 8 pre-installed. At lines 3–4 we install the JIoT fork interpreter and we set the environmental variable `JOLIE_HOME` to point to the location of the installed interpreter. At lines 6–7 we add the source code of the `LogicEngine` in the home directory of the image. Finally, at line 8 we start the execution of the `LogicEngine`.

9 Related Work

In the literature there are many proposals for platforms, middlewares, smart gateways, and general systems, all aimed at solving the interoperability problem arising from the current "babel" of IoT technologies (protocols, formats, and languages). Without any claim of being complete, here we mention a few notable examples which are somehow related to the topic of the current paper.

Recently the W3C started the Web of Things (WoT) Working Group [22]. The aim of WoT is to define a standard stack of layered technologies, as well as software architectural styles and programming patterns, to uniform and simplify the creation of IoT applications. In this context, the W3C is working on a WoT Architecture [37]. The main concept of the architecture is the notion of "servient", a virtual entity that represents a physical IoT device. Servients provide technology-independent, standard APIs that developers can use to transparently

```
1 FROM openjdk:alpine
2
3 RUN java -jar jiot.jar -jh /usr/lib/jolie/ -jl /usr/bin/
4 ENV JOLIE_HOME /usr/local/lib/jolie
5
6 ADD logic_engine.ol /home/.
7 WORKDIR /home
8 RUN jolie logic_engine.ol
```

Listing 10. The Dockerfile used to deploy the `LogicEngine`.

operate in heterogeneous environments. Remarkably, both the WoT proposal and ours concern high-level abstractions for low-level access to devices provided via, e.g., HTTP, CoAP, and MQTT. However, while we propose a dedicated language, they provide API specifications. More in general, there are many proposals for the integration of WoT and IoT. For example [38] and [39] define general platforms covering different layers of IoT, including an accessibility layer which integrates concepts like smart gateways and proxies to facilitate the connection of (smart) Things into the Internet infrastructure, using architectural principles based on REST. Smart gateways and proxies are used in several industrial proposals to facilitate the development of applications. Common denominator of some of these proposals, e.g., [33, 40, 41], is the abstraction of low-level functionalities provided by embedded devices (e.g., connectivity and communication over low-level protocols like ZigBee, Z-Wave, Wi/IP/UPnP, etc.). Smart gateways are used also to translate (or integrate) CoAP into HTTP [42, 43, 44] and to integrate both CoAP and MQTT by means of specific middlewares [45]. Eclipse IoT [46] is an IoT integration framework proposed by the Eclipse IoT Working Group. Aim of Eclipse IoT is to build an open IoT stack for Java, including the support for device-to-device and device-to-server protocols, as well as the provision of protocols, frameworks, and services for device management. There exist several European projects, notably INTER-IoT [47] and symbIoTe [12], that address the issue of interoperability in IoT and have produced several concrete proposals. Finally, a work close to ours is [48], where a middleware converts IoT heterogeneous networks into a single homogeneous network.

Although related to our aim in this paper, the cited proposals tackle the problem of IoT integration from a framework perspective: they provide chains of tools, each addressing a specific level of the integration stack. Differently, we extend a language specifically tailored for system integration and advanced flow manipulation, Jolie, to support integration of IoT devices. This offers a single linguistic domain to seamlessly integrate disparate low-level IoT devices and intermediate nodes (collectors, aggregators, gateways). Moreover, Jolie is already successfully used for building Cloud-based, microservice solutions [49, 50]. This makes the language useful also for assembling advanced architectures for IoT, e.g., to handle real-time streaming and processing of data from many devices. The benefit, here, is that, while solutions based on frameworks require dedicated proficiencies on each of the included tools, Jolie programmers can directly work at any level of the IoT stack, without the need to acquire specific knowledge on the tools in a given framework.

To conclude our revision of related work, we narrow our focus on language-based integration solutions for IoT. The work mostly related to ours is SensorML [51]. SensorML, abbreviation of Sensor Model Language, is a modeling language for the description of sensors and, more in general, of measurement processes. Some features modeled by the language are: discovery and geolocalization of sensors, processing of sensor observations, and functionalities to program sensors and to subscribe to sensor events. While some traits of SensorML are common to our proposal, the scopes of the two languages sensibly differ.

Indeed, while Jolie is a high-level language for programming generic architectures (spanning from Cloud-based microservices to low-level IoT integrators), SensorML just models IoT devices, their discovery, and the processing of sensor observations.

10 Discussion and Conclusion

In this paper, we proposed a language-based approach for the integration of disparate IoT platforms. We built our treatment on the Jolie programming language. This first result is an initial step towards a more comprehensive solution for IoT ecosystem integration and management. Concretely, we included in Jolie the support for two of the most widely used IoT protocols. The inclusion enables Jolie programmers to interact with the majority of present IoT devices. Summarizing our results: *i*) we included in Jolie the CoAP application protocol, also extending the Jolie language to support the UDP transport protocol, *ii*) we added the support for the MQTT protocol and, in doing so, *iii*) we tackled the challenging problem of mapping the renowned pattern of request-responses (typical of HTTP and other widely used protocols) into the publish/subscribe message pattern of MQTT. The mapping abstracts from peculiarities of MQTT and is applicable to any publish/subscribe protocol.

Regarding future work, we are currently investigating the integration in Jolie of more IoT protocols [3], in order to extend the usability of the language in the IoT setting.

It would also be interesting to extend not only the Jolie interpreter, as we have done, but also the formal model behind it [29, 31, 52]. To this end, we can take ideas from the formal model of IoT systems presented in [53].

Another interesting direction for future developments is studying how Jolie can support the testing of IoT technologies, e.g., to test how different protocol stacks perform over a given IoT topology. Thanks to the simplicity of changing the combination of the used protocols (application and transport), experimenters can quickly test many configurations, also enjoying a more reliable platform to compare them. Indeed, usually even changing one of the protocols in the configured stack would require an almost complete rewrite of the logic of network components. Contrarily, in Jolie, this change just requires an update of the deployment part of programs, leaving the logic unaffected. Moreover, such an update could even be done programmatically, making the practice of repeated experimenting on IoT networks easier and more standardized.

Finally, as future work, we also consider the possibility of developing a lightweight version of the language, to be used on low-power IoT devices. Indeed, in this paper, we assumed that these devices are programmed with low-level languages, since they can support only a very constrained execution environment. Clearly, letting programmers develop all the components of an IoT network in the same language would not only ease its implementation but also testability, deployment, and maintenance. However, achieving such a result would require a very challenging engineering endeavor.

Acknowledgments We thank Marco Di Felice, Luca Bedogni, and Federico Montori for useful suggestions and comments.

References

1. J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Comp. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.
2. L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.
3. A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.
4. C. Bormann, "CoAP website." http://coap.technology/, 2016.
5. Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, IETF, 2014.
6. MQTT community, "MQTT website." http://mqtt.org, 2014.
7. A. Banks and R. Gupta, "MQTT Version 3.1.1," Oasis standard, Oasis, 2014. Available at http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/.
8. M. Milenkovic, "A case for interoperable IoT sensor data and meta-data formats: The Internet of Things (Ubiquity symposium)," *Ubiquity*, pp. 2:1–2:7, 2015.
9. L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2008.
10. N. Garg, *Apache Kafka*. Packt Publishing Ltd, 2013.
11. S. Soursos, I. P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, and G. Carrozzo, "Towards the cross-domain interoperability of IoT platforms," in *EuCNC*, pp. 398–402, IEEE, 2016.
12. I. Gojmerac, P. Reichl, I. Podnar Žarko, and S. Soursos, "Bridging IoT islands: the symbIoTe project," *Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 315–318, 2016.
13. "The bIoTope project." http://www.biotope-project.eu/, 2017.
14. T. Erl, *Soa: principles of service design*. Prentice Hall Press, 2007.
15. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, "JOLIE: a Java Orchestration Language Interpreter Engine," *ENTCS*, vol. 181, pp. 19 – 33, 2007.
16. F. Montesi, C. Guidi, and G. Zavattaro, "Composing services with JOLIE," in *ECOWS*, pp. 13–22, IEEE, 2007.
17. F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with Jolie," in *Web Services Foundations*, pp. 81–107, Springer, 2014.
18. "Jolie website." http://jolie-lang.org, 2017.
19. M. Gabbrielli, S. Giallorenzo, I. Lanese, and S. P. Zingaro, "Jolie for IoT website." http://www.cs.unibo.it/projects/jolie/jiot.html, 2017.
20. W3C, "Transport message exchange pattern: Single-request-response." https://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport_MEP, 2001.
21. M. Gabbrielli, S. Giallorenzo, I. Lanese, and S. P. Zingaro, "A language-based approach for interoperability of IoT platforms," in *HICSS*, AIS Electronic Library (AISeL), 2018.
22. "Web of Things." https://www.w3.org/WoT/, 2017.
23. R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

24. J. Postel, "User datagram protocol," RFC 768, IETF, 1980.
25. F. Montesi, "Process-aware web programming with Jolie," *SCP*, vol. 130, pp. 69–96, 2016.
26. U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S—a publish/subscribe protocol for wireless sensor networks," in *COMSWARE*, pp. 791–798, IEEE, 2008.
27. N. Maurer and M. Wolfthal, *Netty in Action*. Manning Publications, 2016.
28. O. Kleine, "nCoAP." https://github.com/okleine/nCoAP.
29. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "SOCK: a calculus for service oriented computing," in *ICSOC*, pp. 327–338, Springer, 2006.
30. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro, "Dynamic error handling in service oriented applications," *Fundam. Inform.*, vol. 95, no. 1, pp. 73–102, 2009.
31. F. Montesi and M. Carbone, "Programming services with correlation sets," in *ICSOC*, pp. 125–141, Springer, 2011.
32. OASIS, "Web Services Business Process Execution Language." http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
33. "SmartThings." http://www.smartthings.com/, 2016.
34. M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "JSON-LD 1.1." https://json-ld.org/spec/latest/json-ld/.
35. D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.
36. F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Packt Publishing, 2017.
37. "Web of Things architecture." https://w3c.github.io/wot/architecture/wot-architecture.html, 2017.
38. G. Dominique, "A web of things application architecture-integrating the real-world into the web," *Zurich, Diss. ETH*, no. 19891, pp. 10–12, 2011.
39. I. Corredor, E. Metola, A. M. Bernardos, P. Tarrío, and J. R. Casar, "A lightweight web of things open platform to facilitate context data management and personalized healthcare services creation," *IJERPH*, vol. 11, no. 5, pp. 4676–4713, 2014.
40. "Meshlium." http://www.libelium.com/products/meshlium/, 2016.
41. "Thinking things." http://www.thinkingthings.telefonica.com/, 2016.
42. A. B. Sulaeman, F. A. Ekadiyanto, and R. F. Sari, "Performance evaluation of HTTP-CoAP proxy for wireless sensor and actuator networks," in *APWiMob*, pp. 68–73, IEEE, 2016.
43. A. Ludovici and A. Calveras, "A proxy design to leverage the interconnection of CoAP wireless sensor networks with web applications," *Sensors*, vol. 15, no. 1, pp. 1217–1244, 2015.
44. E. Mingozzi, G. Tanganelli, and C. Vallati, "CoAP proxy virtualization for the Web of Things," in *CloudCom*, pp. 577–582, IEEE Computer Society, 2014.
45. D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. K. Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," in *ISSNIP*, pp. 1–6, IEEE, 2014.
46. "The Eclipse for IoT Project." https://iot.eclipse.org/, 2017.
47. M. Ganzha, M. Paprzycki, W. Pawlowski, P. Szmeja, and K. Wasielewska, "Semantic technologies for the IoT - an inter-IoT perspective," in *IoTDI*, pp. 271–276, IEEE, 2016.
48. W. Zhiliang, Y. Yi, W. Lu, and W. Wei, "A SOA based IoT communication middleware," in *MEC*, pp. 2555–2558, IEEE, 2011.
49. M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi, "Self-reconfiguring microservices," in *TPFM*, vol. 9660 of *LNCS*, pp. 194–210, Springer, 2016.

50. F. Callegati, S. Giallorenzo, A. Melis, and M. Prandini, "Insider threats in emerging mobility-as-a-service scenarios," in *HICSS*, AIS Electronic Library (AISeL), 2017.
51. "The sensorML project." http://www.opengeospatial.org, 2017.
52. S. Giallorenzo, F. Montesi, and M. Gabbrielli, "Applied choreographies," in *FORTE*, pp. 21–40, Springer, 2018.
53. I. Lanese, L. Bedogni, and M. Di Felice, "Internet of Things: a process calculus approach," in *SAC*, pp. 1339–1346, ACM, 2013.