A Language-based Approach for Interoperability of IoT Platforms

Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Stefano Pio Zingaro
Università di Bologna / INRIA
maurizio.gabbrielli@unibo.it, saverio.giallorenzo@gmail.com,
ivan.lanese@gmail.com, stefanopio.zingaro@unibo.it

Abstract—*The Internet of Things (IoT) promotes the communication among heterogeneous entities, from small sensors to Cloud systems. However, this is realized using a wide range of communication media and data protocols, usually incompatible with each other. Thus, IoT systems tend to grow as homogeneous isolated platforms, which hardly interact. To achieve a higher degree of interoperability among disparate IoT platforms, we propose a language-based approach for communication technology integration. We build on the Jolie programming language, which allows programmers to easily make the same logic work over disparate communication stacks in a declarative, dynamic way. 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. As technical result, we integrate in Jolie the two most adopted protocols for IoT communication, i.e., CoAP and MQTT. In this paper, we report our experience 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

IoT systems are being developed for a wide range of applications and target areas [1], [2], using a number of different technology stacks [3]. Nevertheless, as reported, e.g., in [4], [5], IoT platforms frequently take the shape of vertical solutions (usually dubbed "IoT islands") that focus on a specific application domain and rely on a single communication technology stack. Such platforms provide little support for collaboration and integration. How to overcome this limitation is currently a hot topic, tackled also by ongoing EU projects, e.g., symbIoTe [5] and bIoTope [6].

The problem of integration involves many layers, spanning from link-layer communication technologies, such as ZigBee and WiFi, to application-layer protocols like HTTP, CoAP [7], [8], and MQTT [9], [10], reaching the top-most layers of data-format integration [11].

In this work, we tackle the main issues of intercommunication among IoT islands, by focusing on integration at both the transport (TCP or UDP) and application level. The approach we propose is language based, that is, we aim at devising a programming language where different communication protocols can seamlessly coexist and interoperate. Thanks to proper abstractions provided by the language we propose, programmers can easily change the transport and application protocols used for a given communication, even at runtime. 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, the language we propose can automatically marshal and un-marshal data as required.

Following our approach, most of the complexity of guaranteeing interoperability among protocols is managed by the programming language, and hidden from the programmer. This hidden complexity is particularly high when many technologies are involved. The problem is exacerbated when one has to replace the technology 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. These are scenarios where our language-based approach can make the difference with respect to other approaches.

As an illustrative example, 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. We want to program a collector that receives temperature measurements from both islands and uses them for further elaboration.

In the language we propose, the description of such a collector is divided into two parts: a *behavior* specifying the logic of the elaboration and a *deployment* describing in a declarative way how communication is performed.

The behavior could be of the shape:

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

URI: http://hdl.handle.net/10125/50603
ISBN: 978-0-9981331-1-9

where Line 3 is a reception statement, expecting a temperature measurement on *operation* receiveTemperature (an operation is an abstraction for technology-specific concepts such as channels, resources, URLs, . . .) and storing it in variable data.

Once we defined the logic of the collector, we need to specify on which technologies the communication happens; in the example above, how the collector accepts communications from devices. In our language this information is defined within *ports*. For instance, the port for receiving (denoted with keyword **inputPort**) HTTP measurements can be defined as follows:

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

inputPort CollectorPort1 {
 Location: "socket://collector.net:8000"
 Protocol: http
 Interfaces: TemperatureInterface
}
```

Port CollectorPort1 specifies that the collector expects inbound communications via **Protocol** http using a TCP/IP socket receiving at URL "collector.net" and on TCP port 8000. The port and the operation are linked by the definition of **interface** TemperatureInterface. 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.

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 at the application layer. Hence, the whole code of the collector becomes:

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

inputPort CollectorPort1 {
  Location: "socket://collector.net:8000"
  Protocol: http
  Interfaces: TemperatureInterface
}

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

main {
  receiveTemperature( data );
}
```

Listing 1. 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 disparate technology stacks. Clearly, one can define different operations for different technologies, if the required elaborations differ. Our language supports both inbound and outbound communications, the latter declared with **outputPorts**, whose structure follows that of **inputPorts**. In addition, we let programs change the **Location** and **Protocol** of its **outputPorts** at runtime, enabling the selection of the appropriate technologies for each context.

1.1 Our contribution

To enable the programming of IoT integration in the above style, we do not start from scratch, but we leverage the work done in the area of Service-Oriented Architectures (SOAs) [12] and, in particular, we build on the Jolie programming language [13], [14], [15]. Indeed, the example above uses the Jolie syntax and abstraction mechanisms.

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 choice).

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 [16]). Dually, inbound **OneWay** communications wait for a message, while **RequestResponse** communications wait for a message and send back a reply.

A main feature of the Jolie language, and the reason why we base our approach on it, is that it allows one to switch among many communication media (via keyword **Location**) and data protocols (via keyword **Protocol**) in a simple, uniform way. Being born in the field of SOAs, Jolie supports the main communication media (TCP/IP sockets, Bluetooth L2CAP, Java RMI, and Unix local sockets) and data protocols (HTTP, JSON-RPC, XML-RPC, SOAP and their respective SSL versions) from this area.

We think that the ability to use different communication modalities in a uniform way and to easily switch between them is very useful in the area of IoT. However, to make this approach practical, we also need to support the main communication stacks used in the IoT setting. Indeed, the main technical contribution of the present paper is the introduction in Jolie of the support for two application protocols relevant in the IoT scenario, namely CoAP [8], [7] and MQTT [10], [9].

Even if Jolie provides support for easy integration of new protocols, the task is non trivial. Indeed, all the protocols currently integrated in Jolie support 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, both these assumptions fall 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 battery and bandwidth preservation is important.
- MQTT communications are based on the publish-subscribe paradigm, which contrasts with Jolie point-to-point communication primitives. Hence, we need to define a mapping of the general abstractions of the Jolie language into the publish-subscribe paradigm, balancing two factors: *i*) preserving the simplicity of the point-to-point communication style and *ii*) capturing the typical flow of communications as programmed in a publish-subscribe style. An evident example of the challenges of our mapping is the implementation of request-response communications on top of publish-subscribe interactions. Remarkably, the mapping that we present in this work is general and could be used also in other contexts.

The flexibility provided by Jolie can be used to support and interconnect multiple IoT islands, as discussed above. Jolie supports also more advanced scenarios where the selection of the protocol to use changes according to internal or environmental conditions, such as available energy or quality of communication, but of course this requires some capability of switching the protocol also from the side of the Things, which may not be the case in many practical situations.

Indeed, in the rest of the paper, we omit to model IoT devices — like Arduino and other microcontrollers — that are at the edge of the network, since they are normally programmed by using low-level languages. In principle, these devices could be programmed by using Jolie-like languages, possibly extending them to provide those low-level abstractions needed by programmers to access the in-board sensors and actuators. However, the constraints on the hardware and the usually limited amount of energy available to edge devices require a low-footprint, lightweight execution environment. Although these requirements could be achievable also for a language like Jolie, this would require a strong engineering effort, which is not considered in this paper. We argue that this direction of work is not 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 other devices (gateways, aggregators, servers). These other devices have more powerful hardware and less constraints on energy consumption, and can then implement the logic for the data processing. Hence, here we neglect the programming of edge devices and we focus on those devices that can both host the Jolie runtime and whose topological context can benefit from the flexibility offered by the language.

2 Jolie for IoT

As mentioned above, Jolie currently supports some of the main technologies used in SOAs. However, only a limited amount of IoT devices uses the media and protocols already supported by Jolie. Indeed, protocols such as CoAP/REST [8], [7] and MQTT [10], [9], which are widely used in IoT scenarios, are not yet implemented in Jolie. Implementing these protocols is essential in order to allow Jolie programs to directly interact with the majority of IoT devices. However there are some challenges linked to the implementation of these technologies within Jolie:

- *lossless vs. lossy protocols* — In SOAs, machine-to-machine communication relies on lossless protocols, as there are no strict constraints on energy consumption or bandwidth, hence the number of message exchanges at the transport level needed for ensuring a message delivery is not critical. On the contrary, in IoT networks these constraints exist and are important, and the choice of the protocol needs to take them into account. Many protocols, and the CoAP application protocol in particular, 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 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* — In order to provide language constructs that do not depend on the chosen protocol, we need to find a uniform setting covering both point-to-point communications, such as the ones of HTTP and CoAP, and publish-subscribe communications typical of MQTT. 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, such that typical programming patterns produce similar effects in both settings. In this way, one can program a unique behavior valid for both point-to-point and publish-subscribe scenarios.

We detail how we integrate CoAP/UDP and MQTT in the Jolie language respectively in sections 3 and 4. The Jolie language interpreter, including our extensions at version 1.0, is available at [17]. The integration of our extension into an official Jolie release is ongoing work.

3 Supporting CoAP in Jolie

The *Constrained Application Protocol* (CoAP) [7], [8] 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 [18] into an IoT setting, that is, optimizing it for Machine-to-Machine applications. In particular, CoAP makes use of GET, PUT, POST, and DELETE methods like HTTP. Following the RFC [8], CoAP is implemented on top of the UDP transport protocol [19], with optional reliability. Indeed, CoAP provides two communication modalities: a reliable one, obtained by marking the message as confirmable, and an unreliable one, obtained by marking the message as non confirmable.

As an example, we consider a scenario with a controller, programmed in Jolie, that communicates with one of many thermostats in a home automation scenario. Thermostats are accessible at the generic address `"coap://thermostat/##"` where `"##"` is a two-digit number representing the identifier of a specific device. Thermostats accept two interactions: a GET request on URI `"coap://thermostat/##/getTemperature"`, that returns the current temperature, and a POST request on URI `"coap://thermostat/##/setTemperature"`, that sets the temperature of the HVAC system. We report and comment below the code of a possible Jolie controller.

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

Listing 2. Jolie controller communicating over CoAP/UDP.

Our scenario uses two CoAP resources: `"/getTemperature"` and `"/setTemperature"`. We model them in Jolie at lines 3–6 of Listing 2, 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, as described below, one could override the default mapping, defining the coupling of protocol-specific concepts (here CoAP resources) and operations inside ports. In this way, programmers can define interactions at a high level with interfaces, while the grounding to the specific case is done in the deployment.

At lines 8–24 we define an **`outputPort`** to interact with the `Thermostat`. At line 9 we specify the **`Location`** of the thermostat. Recalling that the scheme of the resources of the thermostats is `"coap://thermostat/##/..."`, we define the **`Location`** of the port using the UDP `"datagram://"` protocol, followed by the first part of the resource schema `"thermostat"` 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://thermostat: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 10 we define `coap` to be the protocol used by the **`outputPort`**. At lines 11–21 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 [20] for the implementation of the HTTP protocol in Jolie — indeed CoAP adopts HTTP naming schema and resource interaction methods. In particular, we draw from [20] 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`. In particular, we define at line 12 that operation `getTmp` uses the non confirmable modality of CoAP. At line 15 we specify that the CoAP method used is `GET`. At line 14 we define, using the `.format` parameter, that the encoding of the payload of the message in a binary format (`"raw"`). Other accepted values for the `.format` 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 easily be translated into representations based on that shape. At line 15, following the practice introduced in [20], we specify that `getTemp` `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

27[1]. Hence, the interpretation of the declaration at line 15 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 20) is similar to that of `getTmp`, while the operation is set as confirmable and to use method `POST`. Since here the `.format` parameter is omitted, the default `"raw"` is used.

To conclude, we briefly comment the runtime execution of the example, described in the behavior at lines 26–33. At line 27 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 15, at line 27 we define the value of element `id = 42`, thus the URI of the resource invoked at runtime is `"coap://thermostat/42/getTemperature"`. Notably, in the example we hard-coded the `id` of the device, however in a realistic setting the value of `id` would be retrieved from a variable. Once received, the response from thermostat `42` is assigned to variable `temp`. The example concludes with a conditional in which, if the temperature is above `27` (line 28), 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 `confirmable` 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.

3.1 Implementation of CoAP/UDP in Jolie

We end this section reporting the most relevant issues met during the implementation in Jolie of the CoAP/UDP stack. 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. In particular, the implementation of UDP that we provide can also be used to support other protocols relying on UDP like MQTT-SN [21]. For this reason, we separately present the integration in Jolie of UDP and of CoAP.

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 module which is loaded only if the protocol is used. This expedites the integration of new protocols in the language.

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`.

The implementation of UDP consists in a listener and a sender class, both based on the Netty framework [22]. Since the structure expected by Jolie and the one provided by Netty are similar, the integration of UDP is smooth.

The implementation of CoAP consists in a unique class, taking care of both encoding and decoding messages, and is based on nCoAP [23].

We notice that 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) [24], which enables to rigorously reason on the behavior of Jolie programs. The semantics in [24] 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.

4 Supporting MQTT in Jolie

Message Queue Telemetry Transport (MQTT) [9], [10] 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.

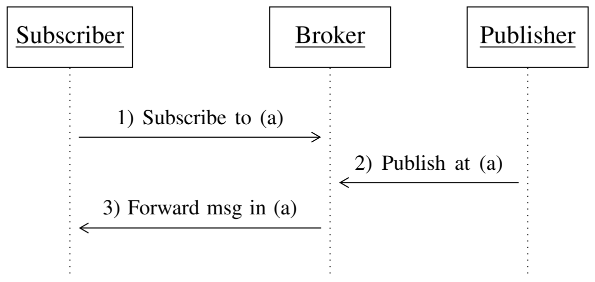

Fig. 1. Typical publish/subscribe interaction pattern.

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

To present how we model the MQTT protocol in Jolie, we first detail the simpler case of **OneWay** communications in section 4.1. Then, we address the more complex case of **RequestResponse**s in section 4.2. Both mappings are general, i.e., in principle they can be applied to implement one-way and request-response communication patterns on top of any publish/subscribe protocol. Remarks on the implementation are provided in section 4.3.

4.1 One-Ways in MQTT

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

To exemplify **OneWay** inbound communications, we use the example in Listing 3, which is a revision of the example in Listing 1 by omitting the HTTP and CoAP ports and by adding an MQTT **inputPort**.

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

Listing 3. Code of the Collector Example, revised for MQTT.

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, specifies the address of the Broker.

While the syntax and the effect of the communication primitive from the point of view of the programmer are the same as the ones in Listing 1, the actual message exchanges performed to obtain such an effect are different.

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 Jolie 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. In Jolie, 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 we default to mapping Jolie operations to topics, otherwise we use the `.osc` parameter `.alias` to enable loose coupling between operations and topics. We remark that `.alias` 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 correct 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`. Instead, in MQTT, the `.alias` parameter is used to identify the topic for subscription. For example, in Listing 3, 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 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 4. Outgoing **OneWay** operations simply cause the publication of the value passed as the parameter of the invocation (line 17) 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`.

```
 1 interface ThermostatInterface {
 2  OneWay: setTmp( TmpType )
 3 }
 4
 5 outputPort Broker {
 6  Location: "socket://iot.eclipse.org: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 main {
17  setTmp@Broker( 24 { .id = "42" } )
18 }
```

Listing 4. Example of outgoing MQTT **OneWay** communication.

4.2 Request-Responses in MQTT

To discuss **RequestResponse** communications, let us consider the example in Listing 2, revised in Listing 5 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**.

```
 1 interface ThermostatInterface {
 2   RequestResponse: getTmp( TmpType )( int )
 3 }
 4
 5 outputPort Broker {
 6  Location: "socket://iot.eclipse.org:1883"
 7  Protocol: mqtt {
 8   .osc.getTmp << {
 9    .format = "raw",
10    .QoS = 2, // exactly once QoS
11    .alias = "%!{id}/getTemperature",
12    .aliasResponse = "%!{id}/getTempReply",
13   }
14  }
15  Interfaces: ThermostatInterface
16 }
17
18 main {
19  getTmp@Broker( { .id = "42" } )( temp )
20 }
```

Listing 5. Jolie controller communicating over MQTT.

Syntactically, the main novelty with respect to the **outputPort** in Listing 4 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, hence we exploit the content and the topic of messages to this end. 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 5, we use `.id` in line 19 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 19 in Listing 5. As a reference, the pattern of interactions is depicted in the left part of Fig. 2. We will describe the right part later on, after having introduced inbound request-response communications.

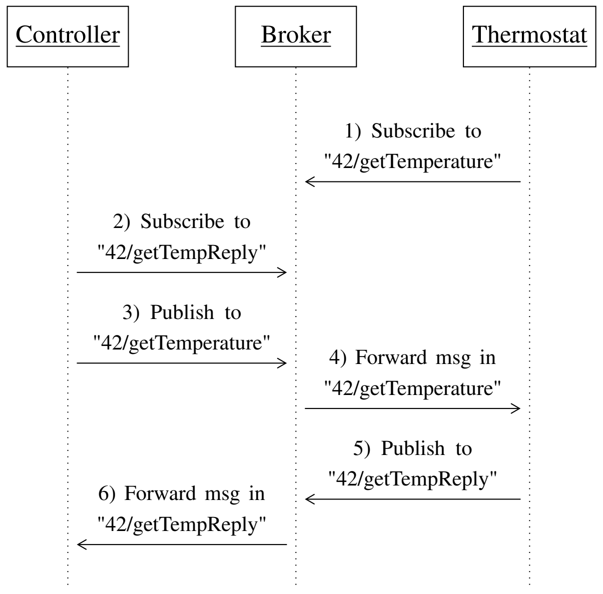
Controller
Broker
Thermostat
1) Subscribe to "42/getTemperature"
2) Subscribe to "42/getTempReply"
3) Publish to "42/getTemperature"
4) Forward msg in "42/getTemperature"
5) Publish to "42/getTempReply"
6) Forward msg in "42/getTempReply"

Fig. 2. Interaction in the home automation example in MQTT.

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 payload of the message contains the reply topic `"42/getTempReply"`. 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, by anticipating the subscription, it would complicate the usage of runtime information in templates.

To exemplify inbound **RequestResponse** communications, we assume that the thermostat in our example is programmed in Jolie. We report its code in Listing 6.

```
 1 interface ThermostatInterface {
 2   RequestResponse: getTmp( TmpType )( TmpType )
 3 }
 4
 5 inputPort Thermostat {
 6  Location: "socket://localhost:9000"
 7  Protocol: mqtt {
 8   .broker = "socket://iot.eclipse.org:1883";
 9   .osc.getTmp << {
10    .format = "raw",
11    .alias = "42/getTemperature"
12   }
13  }
14  Interfaces: ThermostatInterface
15 }
16
17 main {
18  getTmp( temp )( temp ){
19   // retrieves temperature and stores
20   // it within the root of variable temp
21  }
22 }
```

Listing 6. Jolie thermostat communicating over MQTT.

At line 11 in Listing 6, the `.alias` parameter `"42/getTemperature"` is static, as usual for **inputPort**s.

When the thermostat program is launched, it subscribes to topic `"42/getTemperature"`. When a message on this topic arrives, the reply topic, e.g., `"42/getTempReply"`, is extracted and the rest of the payload (empty in this case) is passed to the behavior. The body of the **RequestResponse** (lines 19–20) is executed to compute the return value. Finally, the return value is published on the reply topic `"42/getTempReply"`.

We now summarize the exchange between the controller and the thermostat (left part of Fig. 2):

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 computes the response and publishes it at topic `"42/getTempReply"`;
6) the Broker forwards the message in topic `"42/getTempReply"` to the controller.

We remark that **RequestResponse** operations are meant to be one-to-one 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.

4.3 Implementation of MQTT in Jolie.

The implementation of MQTT in Jolie required the creation of two main classes: the actual MQTT protocol and a generic publish/subscribe meta-channel that bridges between the end-to-end style of Jolie communications and publish/subscribe interactions. Furthermore, we had to update the Jolie class for channel creation so that it could choose between the standard end-to-end media and the new publish/subscribe meta-channel.

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 [22]. The main difficulty in the implementation of the protocol is the definition of the message patterns needed to implement **OneWay** and **RequestResponse** communications, which have been described above. 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).

In addition to the MQTT protocol, we also implemented a generic publish/subscribe meta-channel. Indeed, since Jolie is based on an end-to-end communication pattern, it assumes that the caller requires the creation of a connection to the server, which waits for 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 sender class.

This pattern, separating listeners from senders, does not apply to publish/subscribe protocols, where both the subscriber and the publisher need to establish a connection with the broker. For this reason we use the publish/subscribe meta-channel to bridge between the two styles. On the one hand, it implements the interfaces of listeners and senders as required by Jolie. On the other hand, it relies on the pre-existing Jolie sender classes (TCP socket in the case of MQTT) to create the connection to the broker.

5 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 our current research.

Recently the W3C started the Web of Things (WoT) Working Group [25]. 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 [26]. 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 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 [27] and [28] 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., [29], [30], [31], 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 [32], [33], [34] and to integrate both CoAP and MQTT by means of specific middlewares [35]. Eclipse IoT [36] 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 [37] and symbIoTe [5], that address the issue of interoperability in IoT and have produced several concrete proposals. Finally, a work close to ours is [38], 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 [39], [40]. 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 most related to ours is SensorML [41]. 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.

6 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.

Another interesting direction comes from the inclusion of publish/subscribe protocols in Jolie. Indeed, the publish/subscribe pattern is renowned for enabling high scalability of networks, as well as supporting flexible and highly dynamic network topologies [42]. Our intuition is that, besides efficiency and scalability, publish/subscribe architectures can achieve a higher degree of reliability if programmed using the Jolie language.

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 light-weight 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] 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.
[5] 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.
[6] "The bIoTope project." http://www.biotope-project.eu/, 2017.
[7] C. Bormann, "CoAP website." http://coap.technology/, 2016.
[8] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," RFC 7252, IETF, 2014.
[9] MQTT community, "MQTT website." http://mqtt.org, 2014.
[10] 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/.
[11] 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.
[12] T. Erl, *Soa: principles of service design*. Prentice Hall Press, 2007.
[13] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, "JOLIE: a Java Orchestration Language Interpreter Engine," *ENTCS*, vol. 181, pp. 19 – 33, 2007.
[14] F. Montesi, C. Guidi, and G. Zavattaro, "Composing services with JOLIE," in *ECOWS*, pp. 13–22, IEEE, 2007.
[15] "Jolie website." http://jolie-lang.org, 2017.
[16] 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.
[17] M. Gabbrielli, S. Giallorenzo, I. Lanese, and S. P. Zingaro, "Jolie for IoT website." http://www.cs.unibo.it/projects/jolie/jiot.html, 2017.
[18] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
[19] J. Postel, "User datagram protocol," RFC 768, IETF, 1980.
[20] F. Montesi, "Process-aware web programming with Jolie," *SCP*, vol. 130, pp. 69–96, 2016.
[21] 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.
[22] N. Maurer and M. Wolfthal, *Netty in Action*. Manning Publications, 2016.
[23] O. Kleine, "nCoAP." https://github.com/okleine/nCoAP.
[24] 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.
[25] "Web of things." https://www.w3.org/WoT/, 2017.
[26] "Web of things architecture." https://w3c.github.io/wot/architecture/wot-architecture.html, 2017.
[27] G. Dominique, "A web of things application architecture-integrating the real-world into the web," *Zurich, Diss. ETH*, no. 19891, pp. 10–12, 2011.
[28] 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.
[29] "Smartthings." http://www.smartthings.com/, 2016.
[30] "Meshlium." http://www.libelium.com/products/meshlium/, 2016.
[31] "Thinking things." http://www.thinkingthings.telefonica.com/, 2016.
[32] 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.
[33] 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.
[34] E. Mingozzi, G. Tanganelli, and C. Vallati, "CoAP proxy virtualization for the Web of Things," in *CloudCom*, pp. 577–582, IEEE Computer Society, 2014.
[35] 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.
[36] "The Eclipse for IoT Project." https://iot.eclipse.org/, 2017.
[37] 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.
[38] W. Zhiliang, Y. Yi, W. Lu, and W. Wei, "A SOA based IoT communication middleware," in *MEC*, pp. 2555–2558, IEEE, 2011.
[39] 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.
[40] 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.
[41] "The sensorML project." http://www.opengeospatial.org, 2017.
[42] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.