Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT

Georgios Bouloukakis, Nikolaos Georgantas, Patient Ntumba, Valérie Issarny

▶ **To cite this version:**

Georgios Bouloukakis, Nikolaos Georgantas, Patient Ntumba, Valérie Issarny. Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT. Future Generation Computer Systems, 2019, 101, pp.1271-1294. 10.1016/j.future.2019.05.064 . hal-02304074

HAL Id: hal-02304074

https://inria.hal.science/hal-02304074v1

Submitted on 14 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated Synthesis of Mediators for Middleware-layer Protocol Interoperability in the IoT

Georgios Bouloukakis[a,b,*], Nikolaos Georgantas[b], Patient Ntumba[b], Valérie Issarny[b]

[a] *University of California, Irvine, USA*
[b] *MiMove Team, Inria Paris, France*

Abstract

To enable direct Internet connectivity of Things, complete protocol stacks need to be deployed on resource-constrained devices. Such protocol stacks typically build on lightweight IPv6 adaptations and may even include a middleware layer supporting high-level application development. However, the profusion of IoT middleware-layer interaction protocols has introduced technology diversity and high fragmentation in the IoT systems landscape with siloed vertical solutions. To enable the interconnection of heterogeneous Things across these barriers, advanced interoperability solutions at the middleware layer are required. In this paper, we introduce a solution for the automated synthesis of protocol mediators that support the interconnection of heterogeneous Things. Our systematic approach relies on the *Data eXchange (DeX)* connector model, which comprehensively abstracts and represents existing and potentially future IoT middleware protocols. Thanks to DeX, Things seamlessly interconnect through lightweight mediators. We validate our solution with respect to: (i) the support to developers when developing heterogeneous IoT applications; (ii) the runtime performance of the synthesized mediators.

Keywords: Internet of Things, Interoperability, Protocol Mediators, Middleware, Software Composition

1. Introduction

The IoT comprises sensors and actuators that are heterogeneous with different operating (e.g., operating platforms) and hardware (e.g., sensor chip types) characteristics, hosted on diverse Things (e.g., mobile phones, vehicles, clothing, etc.). To support the deployment of such devices, major tech industry actors have introduced their own middleware APIs and protocols, which deal with: *(i)* the limited hardware (e.g., energy, memory) and network resources (e.g., low bandwidth); and *(ii)* loosely coupled interactions in terms of time and space. The resulting APIs and protocols are highly heterogeneous. In particular, protocols differ significantly in terms of interaction paradigms and data formats. For instance, protocols such as CoAP [1] relying on *Client/Server (CS)* interactions, MQTT [2] and WebSocket [3] based on the *Publish/Subscribe (PS)* and *Data Streaming (DS)* interaction paradigms, respectively, or SemiSpace [4] offering a lightweight shared memory (*Tuple Space, TS*) are among the most widely employed ones in IoT applications.

As an illustration, consider the current popular traffic management systems [5]. We can classify them into three categories, leveraging *fixed-sensors* (traffic cameras, doppler radars, etc.), *vehicle-devices* (on-board, GPS-based [6]), and *smartphones* with embedded sensors (accelerometer, gyroscope [7, 8]). The combination of these sensors/devices provide an overall *Traffic Information Management* (TIM) system (see Fig. 1). To enable the effective exploitation of the underlying communication infrastructure [9, 10], each sensor/device possibly employs a different IoT middleware protocol for data exchange, as depicted in Fig. 1. In particular, fixed city-deployed sensors employ the WebSocket [3] protocol to stream their data to a *traffic estimation service* that applies the REST [11] architectural style. Vehicle-devices and smartphone users employ the MQTT [2] and SemiSpace [4] protocols, respectively, to periodically push data to the estimation service. Finally, the REST estimation service processes the collected data and provides back estimated traffic information to the end-users (smartphones, vehicle end-users).

Each one of the above protocols implements different APIs and primitives for sending/receiving data of different formats [12, 9, 13]. Hence, to enable such a scenario, the heterogeneity between the involved peers (e.g., WebSocket → REST) must be tackled. Solving the interoperability problem is challenging, especially due to the fast development of protocols and APIs aiming to support IoT applications. Existing cross-protocol interoperability efforts are based on: *(i)* bridging communication protocols [14, 15, 16]; *(ii)* wrapping systems behind standard technology interfaces [17, 18]; and *(iii)* providing common API abstractions [19, 20]. In particular, such techniques have been applied mainly for client/server protocols through the service oriented architecture (SOA) and enterprise service bus (ESB) technologies [21].

In this paper, we introduce the *Data eXchange (DeX)* API & connector model, which supports the abstraction

*Corresponding author at: Donald Bren School of Information and Computer Sciences University of California, Irvine, USA.
Email address: boulouk@gmail.com (Georgios Bouloukakis)

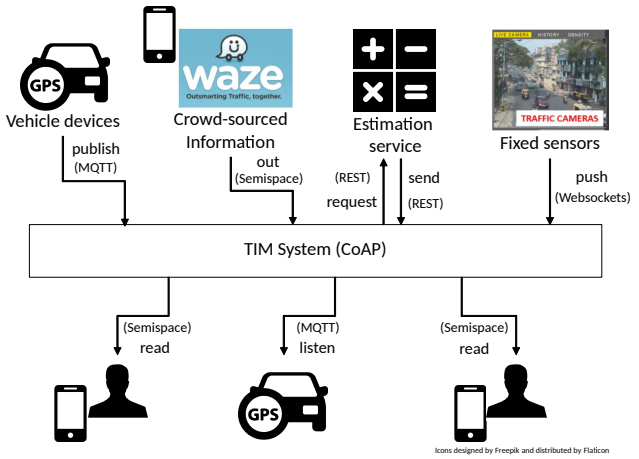

Figure 1: Transport Information Management (TIM) system.

of the functional semantics of middleware IoT protocols (e.g., CoAP, MQTT, WebSocket, etc.). Our approach involves two steps. First, we introduce an API & connector model for each one of the core interaction paradigms (i.e., CS, PS, DS and TS), which implements the most common functional semantics of existing middleware IoT protocols that follow the specific paradigm. Subsequently, we devise the DeX API & connector model, which comprehensively abstracts and represents the semantics of various middleware protocols that follow any of the four core paradigm models. By relying on the DeX API & connector model, we introduce our middleware protocol interoperability solution, implemented by the *DeX Mediator Synthesizer (DeXMS)* framework [22, 23]. Inside DeXMS, we provide support for an extensible set of middleware protocols, abstracted as DeX connectors. Additionally, we elicit the *DeX Interface Description Language (DeXIDL)*, which can be used to describe the application interfaces of Things in DeX terms, i.e., abstracting the heterogeneous middleware protocols employed by the Things. Then, by relying on the previous and applying *model-driven development* techniques, we are able to synthesize *mediators* (i.e., connector wrappers or mediating adaptors [24]) in an automated manner for connecting heterogeneous Things with each other.

The rest of this paper is structured as follows. In section 2, we provide an overview of existing middleware-layer protocols found in the IoT. The majority of these protocols can be abstracted by using the models of the core interaction paradigms (CS, PS, DS and TS) introduced in section 3. Section 4 presents our DeX connector model that abstracts and represents the semantics and primitives of the core models in a unified way. In section 5, we present our middleware protocol interoperability solution provided by the DeXMS framework. Then, in section 6, we discuss the results of the DeXMS evaluation. This is followed by related work and conclusions in section 7 and section 8, respectively.

2. Middleware-Layer IoT Protocols

IoT applications are typically deployed on resource-constrained devices (i.e., *Things*). Depending on their available computational resources, Things (i) may or (ii) may not host a complete protocol stack (including a middleware protocol) that enables their direct connection to the Internet. In the case of (ii), access to the Things is performed through a proxy/gateway. With the technological evolution of sensor nodes, case (i) is now attracting much attention, as it enables autonomous Things. The authors in [31] undertake this approach by deploying SOAP-based Web services directly on the nodes without using gateways. DPWS [25] was introduced in 2004 as an open standard by OASIS. It is suitable for supporting large-scale deployments and mobile devices. However the introduced protocol overhead is noticeable and it requires a large amount of memory. Hence, at the same time, deploying the middleware component directly on the device might cause several issues, such as message delays, limited supported interactions, limited computational capacity, high energy consumption, etc.

Several other middleware protocols have been developed to address the above issues, along with standardization efforts that aim to guarantee interoperability. Table 1 summarizes existing middleware protocols along with their supported interactions, strengths and weaknesses with regard to IoT applications. More specifically, OPC UA [26] was designed in 2008 by the OPC foundation targeting resource constrained devices. Similarly to DPWS, it introduces a large payload unsuitable for IoT applications. Due to the complexity and the limitations of the above protocols, IoT developers turned to simpler protocols. Among them, REST [11], is not really a protocol but an architectural style. It is ideal for mobile development but is not suitable for resource constrained devices. Hence, IETF designed CoAP [1], a lightweight protocol which supports highly resource-constrained devices and the delivery of small message payloads. Despite the fact that CoAP supports extremely low-resource interactions, it is more suitable for request/response interactions. On the other hand, the performance of CoAP decreases significantly when transmitting large message payloads and the request/response interaction style affects the battery usage. Finally, XMPP [27], despite the fact that it was standardized by the IETF over a decade ago, re-gained a lot of attention as a suitable protocol for IoT real-time communications. However, it uses XML data formats that create considerable computational overhead.

To provide alternatives to the request/response style and offer time decoupled communication, middleware developers introduced several middleware protocols that follow the publish/subscribe interaction paradigm. JMS [28], a standard by Sun Microsystems, has been one of the most successful asynchronous messaging technologies available; it defines an API for building messaging systems. It is not a messaging protocol, hence, it is possible to build on top of several messaging protocols. DDS [29] is a messaging protocol designed for brokerless architectures and real-time applications. AMQP [30] is

Protocol	Interactions	Weaknesses	Strengths	Other Characteristics
DPWS [25]	request/response; streaming;	noticeable protocol overhead; amount of memory used;	SOA; large-scale deployments; for resource constrained devices;	introduced in 2004; OASIS open standard;
OPC UA [26]	request/response; streaming;	not suitable for IoT;	SOA; highly resource constrained devices;	designed in 2008 by the OPC foundation;
CoAP [1]	request/response; streaming;	high latency for large payloads; request/response affects battery usage;	highly resource constrained devices; suitable for small payloads;	designed by IETF;
REST [11]	request/response;	not suitable for resource constrained devices;	mobile development;	supported by multiple IoT platforms;
XMPP [27]	request/response; streaming;	additional overhead due to XML data formats;	suitable for real-time applications;	standardized by the IETF a decade ago;
JMS [28]	streaming;	focused on Java-centric systems;	support for underlying messaging protocols; widely used;	standard by Sun Microsystems;
DDS [29]	request/response; streaming;	development and configuration complexity;	real-time applications;	brokerless messaging protocol;
MQTT [2]	streaming;	not suitable for large payloads;	highly resource constrained devices;	centralized architecture; OASIS standard;
AMQP [30]	streaming;	not suitable for resource constrained devices;	supports high traffic load;	ISO/IEC standard;
SemiSpace [4]	request/response;	not widely used;	distributed architecture of shared spaces;	based on JavaSpaces;
WebSockets [3]	streaming;	not suitable for resource-constrained devices;	real-time full duplex interactions; only 2 bytes overhead;	part of HTML 5 initiative;

Table 1: Comparison of IoT protocols at the middleware-layer.

another messaging protocol designed to support applications with high message traffic rates. However, it is not suitable for resource-constrained devices. To support highly resource-constrained devices, MQTT [2] offers a publish/subscribe centralized architecture. However, MQTT's performance decreases significantly when sending large message payloads. Leveraging the grouped reception of messages in response to a request addressed to a shared memory, developers of Semispace [4] developed a lightweight middleware by relying on JavaSpaces [32]. Such a middleware reduces energy consumption since it receives grouped messages and avoids HTTP long polling notifications which affect battery usage. Finally, as part of the HTML 5 initiative, WebSockets [3] was introduced to support real-time full duplex (streaming) interactions, using only two bytes of overhead in message payloads.

The authors in [12, 9, 13] compare the most promising IoT middleware protocols: DPWS, CoAP, MQTT, Websockets, XMPP, REST and AMQP. They recommend combining one or more protocols in an IoT application to better exploit the underlying communication infrastructure. However, this comes with increased heterogeneity at the middleware layer. Solving the interoperability problem is challenging, especially due to the fast development of protocols and APIs aiming to support IoT applications. Section 7 provides the most recent efforts in academia and in industry coping with Things interoperability at the middleware layer.

In the next section, we analyze the semantics of common interactions found in the IoT which are part of well known *interaction paradigms* (i.e., CS, PS, DS and TS) in distributed systems. We then introduce the DeX API, which abstracts common interactions of these interaction paradigms.

3. Models for Core Interaction Paradigms

This section identifies the four core interaction paradigms and defines their corresponding models. The proposed models are the outcome of an extensive survey of these paradigms as well as of related middleware platforms in the literature of distributed systems [33, 34, 20]. Typically, middleware protocols provide an API to application developers. Each protocol provides several characteristics (supported interactions, QoS guarantees, etc.) and can be classified under an interaction paradigm. In particular, for each interaction paradigm we provide its model by specifying: (i) its *semantics*, which expresses the different dimensions of coupling among communicating peers and the supported interaction types; (ii) its API (*Application Programming Interface*), which is a set of *primitives* expressed as functions supported by the middleware; and (iii) *sequence diagrams* that show the detailed interactions between the peers.

The semantics of interest includes *space coupling*, *time coupling*, *concurrency* and *synchronization coupling*. Space coupling determines how peers identify with each other and, consequently, how interaction elements (such as messages) are routed from one peer to another. Time coupling essentially determines if peers need to be present and available at the same time for an interaction or if, alternatively, the interaction can take place in phases occurring at different times. Concurrency characterizes the exclusive or shared access semantics of the virtual channel established between interacting peers. Finally, synchronization coupling determines whether the initiator of an end-to-end interaction blocks or not until the interaction is complete; in the former case, the interaction is executed in a synchronous way between the interacting peers. To express synchronization semantics, but also other semantics

of end-to-end interactions, we define four *interaction types*:

- one-way: each peer can take either the *sender* or the *receiver* role. The sender sends a piece of data without waiting for a response; the receiver will receive this element after having set up an appropriate reception mechanism.
- two-way asynchronous (async): each peer can take either the *client* or the *server* role. Clients initiate a request to a server and then continue their processing (non-blocking). The server receives the client's request. It handles it and returns the response, possibly at a later time. The client then receives the response asynchronously and proceeds with its processing.
- two-way synchronous (sync): each peer can take the *client* or *server* role. A synchronous interaction is blocking for the client and requires a prompt response from the server. Clients invoke a request on the server and then suspend their processing while they are waiting for a response for a specific `timeout` period.
- two-way stream: each peer can take either the *consumer* or the *producer* role. The consumer requests to establish a dedicated session with the producer. Once this is established, the producer sends multiple pieces of data that will asynchronously be received by the consumer. Depending on the middleware protocol, both peers or just the consumer can suspend, resume and terminate the session using the corresponding interaction elements.

The above interaction types identify common patterns of interaction that are encountered across the four core interaction paradigms. When two Things employ middleware protocols that follow different interaction paradigms, these common patterns are sought in the two protocols. If a common interaction type is successfully identified, then the two Things can be interconnected and perform this interaction type despite their interaction paradigm heterogeneity (certainly they should also match in terms of application semantics).

To specify the APIs of the core models, we use a pseudo C syntax with the following conventions: (i) functions have no return value; they only have `I` and `O` parameters; (ii) we identify only the parameter names but not their types; and (iii) the pointer (`*`) represents a callback function or an output parameter. The objective for each one of these APIs is to be able to represent the supported interaction types of a wide-range of middleware IoT protocols that follow the corresponding interaction paradigm. Finally, the provided sequence diagrams show the peers' interactions and their specific order for each interaction type.

3.1. Client/Server Model

Middleware-layer IoT protocols such as CoAP [1], XMPP [27] and OPC UA [26], follow the *Client/Server (CS)* interaction paradigm. A client communicates directly with a server either by direct messaging or with a remote procedure call (RPC). In the first case, a single `item` (which encloses data) is sent from the sending entity to the receiving entity. This can be a one-way operation

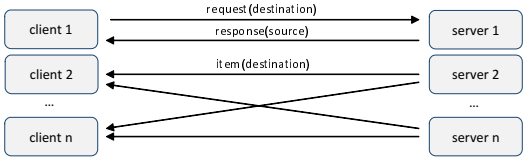

Figure 2: CS semantics.

Interaction	Role	CS Primitives
one-way	Server	**send**(destination, operation, item, lifetime)
	Client	**receive**(operation, *on_receive()) **on_receive**(source, item)
two-way async	Client	**request**(destination, operation, req_item, lifetime, *on_receive()) **on_receive**(resp_item)
	Server	**receive**(operation, *on_receive()) **on_receive**(source, req_item) **send**(source, operation, resp_item, lifetime)
two-way sync	Client	**request**(destination, operation, req_item, *resp_item, timeout)
	Server	**receive**(operation, *on_receive()) **on_receive**(source, req_item, thr_id) { **send**(resp_item, thr_id) }

Table 2: CS model API.

from client to server or a push notification from server to client [25]. The two directions can be modeled in the same way. For simplicity of presentation and because push notifications are common in the IoT, we describe in the following only the server-to-client direction. In the second case, an exchange takes place between the two entities with a `request` message followed by a `response`. In both cases, the communication is identified with an `operation` name. We depict the two cases in Fig. 2.

CS semantics. In terms of space coupling semantics between the two interacting entities, CS requires that the sending entity (`source`) must know the receiving entity (`destination`) and hold a reference of it. Thus, CS represents tight space coupling. With respect to time coupling semantics, both entities must be connected at the same time of the interaction for immediate data transmission. With respect to concurrency semantics, a dedicated virtual channel is used between a sender and a receiver. Items sent by different servers will be received (or not) by the designated clients, based on the offered QoS guarantees of the underlying infrastructure [35]. Regarding synchronization semantics, CS supports one-way, two-way asynchronous and two-way synchronous interactions.

CS API. The above semantics is supported by the CS API primitives and their parameters listed in Table 2. The `lifetime` parameter characterizes the item/request validity in time for asynchronous interactions. This parameter is optional; it applies, for example, in cases where IoT data become obsolete after some time and thus need to be delivered before expiration. The `timeout` parameter characterizes the maximum time interval in which the two-way synchronous interaction must be completed. We detail next the CS API primitives:

send: executes the emission of an `item`. As for its

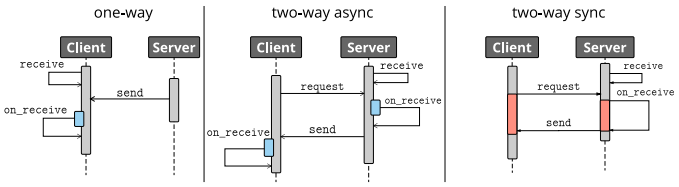

Figure 3: CS sequence diagram.

parameters, it embeds the `destination/source` address, the corresponding `operation` name and the related `item`.

request: executes the emission of a `request` to implement two-way interactions. For asynchronous interactions, it sets the `*on_receive()` callback for receiving the response. For synchronous interactions, it gets blocked until it receives the response; this should be done within a `timeout` period.

receive: sets the reception of one-way items or two-way requests using the `*on_receive()` callback.

on_receive: it is executed upon the reception of one-way items or two-way requests or two-way asynchronous responses. A received two-way request may be part of a synchronous or asynchronous interaction. In the synchronous case, the response is typically sent through the same underlying transport-layer connection as the request. This enables associating the response to the request. We model this with a thread identifier (`thr_id`) that is passed by the middleware in `on_receive` and is used in the subsequent `send` primitive. In the asynchronous case, the `source` and `operation` parameters used in the `send` primitive enable the request–response association, unless there are multiple parallel operations of the same type between the client and the server. In this case, an application-level unique identifier should be included in the parameters `req_item` and `resp_item` of the request and the response, respectively.

CS sequence diagrams. In Fig. 3, we provide the CS sequence diagrams that represent a more detailed view of the supported interaction types by using the above primitives. Particularly, each supported interaction type is specified as follows:

one-way: the client executes the `receive` primitive to set the `*on_receive()` callback for receiving items from any server. Independently, the server executes the `send` primitive for the transmission of an item. Each item is valid for a `lifetime` period and it will be received in an asynchronous way (through the `on_receive` primitive).

two-way async: the server executes the `receive` primitive to set the `*on_receive()` callback for receiving requests from any client. Independently, the client executes the `request` primitive to transmit the request to the server and at the same time set the `*on_receive()` callback in order to receive the response from the specific server. After the `request` primitive is emitted, the client continues its processing. Each request is valid for a `lifetime` period. On the server side, the `on_receive` primitive is executed, and, possibly at a later time depending on the server's priorities, the `send` primitive is executed with the response (assigned a `lifetime` period). Finally, at the client's side, the response is received via the `on_receive` primitive.

two-way sync: similarly to async, the server initiates a `receive` primitive to set the `*on_receive()` callback for receiving requests from any client. After the client has executed the `request` primitive, it blocks its processing until either the reception of the response from the specific server or the expiration of the `timeout` period. On the server side, upon the reception of the request via the `on_receive` primitive, the server must process it and provide a prompt response to the client with the `send` primitive.

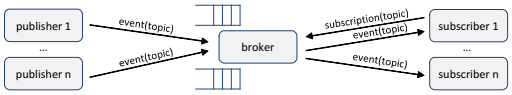

Figure 4: PS semantics.

3.2. Publish/Subscribe Model

The *Publish/Subscribe (PS)* interaction paradigm is commonly used for content broadcasting/feeds. Middleware IoT protocols such as MQTT [2] and AMQP [30], as well as tools and technologies such as RabbitMQ [36], Kafka [37] and JMS [28] follow the PS paradigm. In PS, multiple peers interact via an intermediate `broker` entity. Publishers produce `events` characterized by a specific `filter` to the broker. Subscribers `subscribe` their interest for specific filters to the broker, who maintains an up-to-date list of subscriptions. The broker matches received events with subscriptions and delivers a copy of each event to each interested subscriber. There are different types of subscription schemes, such as *queue-based*, *topic-based*, *content-based* and *type-based* [33]. In queue-based PS, subscribers are associated to queues, individually or in groups, and publishers publish to queues. In topic-based PS, events are characterized with a topic, and subscribers subscribe to specific topics. In content-based PS, subscribers provide content filters, and receive only the events that satisfy the specific filters. Such filters may define constraints in the form of *name–value* pairs of properties and basic comparison operators ($=$, $<$, $\leq$, $>$, $\geq$), which identify valid events. Constraints can be logically combined (and, or, etc.) to form complex filters [33]. Finally, in type-based PS, the event structure is abstracted based on specific types and subscribers receive events based on their type. Regardless of the subscription scheme, we use the generic term `filter`, which represents the subset of events that each peer is interested in to publish/receive.

PS semantics. In terms of space coupling semantics between interacting peers, in the PS style, peers do not need to know each other or how many they are. For instance, in the case of topic-based systems, events are diffused to subscribers only based on the topic (see Fig. 4). With respect to time coupling semantics, peers do not need to be present at the same time. Subscribers may be disconnected at the time when the events are published to the broker. Upon their re-connection to the broker they will receive the pending events. With respect to concurrency semantics, the broker maintains a dedicated buffer

Interaction	Role	PS Primitives
one-way	Publisher	**publish**(broker, filter, event, lifetime)
	Subscriber	**listen**(broker, filter, *on_listen()) **on_listen**(event) **end_listen**(broker, filter)
two-way stream	Subscriber	**subscribe**(broker, filter, lifetime) **listen**(broker, filter, *on_listen()) **on_listen**(event) **end_listen**(broker, filter) **unsubscribe**(broker, filter)
	Broker	**listen**(filter, *on_listen()) **on_listen**(filter) { **publish**(filter, event, lifetime) }

Table 3: PS model API.

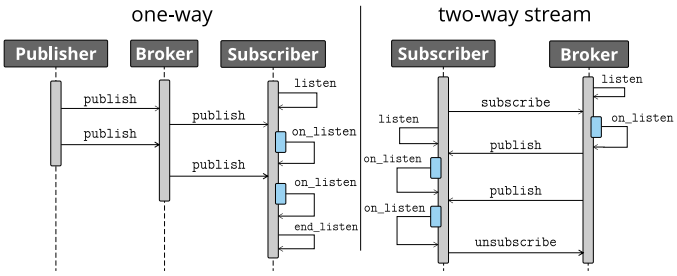

Figure 5: PS sequence diagram.

for each subscriber. Hence, unless an event expires, or the PS QoS guarantees [35] do not support event persistence, all events sent by different publishers will be eventually received by interested subscribers. Furthermore, existing PS middleware protocols support several synchronization semantics between subscribers and the broker. Subscribers may choose to check for pending events synchronously themselves (just check instantly or wait as long as it takes or with a timeout) or set up a callback function that will be triggered asynchronously by the broker when an event arrives. We focus on the latter case that constitutes the most common practice used in the PS style.

PS API. The above semantics is supported by the PS API primitives and their parameters listed in Table 3. We represent the notions of queue, topic, content and type with the generic `filter` parameter, which can be a value or an expression. In addition, the `lifetime` parameter stands for the availability of the `event` in time. We detail next the PS API primitives:

subscribe: executes the subscription of a peer to a `broker` for receiving events that are qualified by `filter`.

publish: at the publisher's side, it publishes an `event` (to a `broker`) that is semantically qualified by `filter`. At the broker's side, it forwards the already published event to the corresponding subscribers (subscribed to `filter`). In both cases, the event is available for the corresponding `lifetime` period.

listen: it is executed at the subscriber's side to enable the asynchronous reception of multiple events related to the `filter` applied. To this end, it specifies the associated `*on_listen()` callback to handle each event received. At the broker's side, it enables the asynchronous reception of subscriptions using the `*on_listen()` callback.

on_listen: it is executed upon the reception of an event at the subscriber's side. Additionally, it is used at the broker's side to receive a subscription (characterized by a `filter`), update the broker's subscriptions list and enable the execution of multiple `publish` primitives which correspond to a flow of events.

end_listen: closes a session of asynchronous event reception.

unsubscribe: ends a subscription for the specific `filter`.

PS sequence diagrams. In Fig. 5 we provide the PS sequence diagrams that represent a more detailed view of *one-way* and *two-way stream* interaction types using the above primitives. Particularly, each interaction type is specified as follows:

one-way: to represent such an interaction, we assume that a subscriber is already subscribed to receive events using a specific `filter`. Similarly, a publisher publishes events on the same `filter`. Thus, there is an end-to-end interaction between the publisher and the subscriber through the `broker`. Since the subscriber is already subscribed, the publisher is able to `publish` events at any point in time. As soon as the subscriber executes the `listen` primitive, it connects and asynchronously receives events via the `on_listen` primitive. The subscriber is able to disconnect with the `end_listen` primitive. As pointed out, to identify this interaction type we abstract the subscription/unsubscription actions of the subscriber. In this way, the delivery of an event being published by a publisher to one of the interested subscribers can be seen as an interaction that is basically the same as, e.g., the transmission of an item from a server to a client in a CS one-way interaction. This can enable, for example, a Thing acting as a PS publisher to connect to another Thing acting as a CS client (by certainly deploying proper mediation functionality, which, among others, should compensate for the missing subscription/unsubscription actions).

two-way stream: for such an interaction, initially the subscriber executes a `subscribe` primitive and afterwards a `listen` primitive, which enables its connection to the `broker`. At the broker's side, a `listen` primitive is executed to receive subscriptions. In particular, each subscription is received through the `on_listen` primitive, which then enables the forwarding of multiple events (coming from multiple publishers) to the corresponding subscriber using the `publish` primitive. Finally, at the subscriber's side, each event is received via the `on_listen` primitive until a disconnection (`end_listen` primitive) or a termination (`unsubscribe` primitive). We note here that the one-way and two-way stream interaction types represent two different ways of seeing PS interactions. For two-way stream, we make abstraction of the potentially multiple publishers that feed the broker. The subscriber–broker remaining part of the interaction has the characteristics of an on-demand streaming interaction, similarly to one of the interaction types identified in the next section.

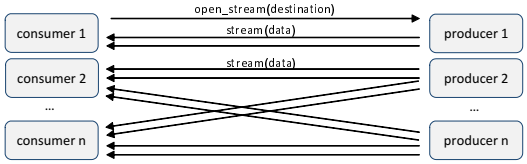

Figure 6: DS semantics.

3.3. Data Streaming Model

The *Data Streaming (DS)* interaction paradigm is commonly used for continuous interactions. Middleware protocols such as WebSocket [3] and Dioptase [38] are based on the DS style. IoT applications (e.g., traffic management, warehouse logistics etc.) produce data coming from the physical world. Such information is produced as a flow of structured data (stream) and thus requires continuous handling.

In DS, a consumer (typically) establishes a dedicated session with an `open_stream` request (see Fig. 6) sent to a producer. Upon the session's establishment, a continuous flow of `data` is pushed from the `producer` to the `consumer`. A `stream` is identified by the pair `<producer, stream_id>`, i.e., the name or address of the producer and a qualifier of the stream that is unique for the specific producer. Finally, each peer (but most commonly the consumer) is able to `suspend`, `resume` and `close` the stream. Our DS model represents only the related interaction semantics of streaming protocols and middleware platforms. Other features found in data streaming, such as continuous queries, compression and windowing mechanisms, can be added on top of the stream interaction semantics of the DS model.

DS semantics. Similarly to CS, DS represents tight space coupling semantics, with the consumer and producer knowing each other. There is also tight time coupling, with peers' availability being crucial for immediate `data` transmission. In terms of concurrency semantics, multiple consumers can receive streams of data from multiple producers over dedicated virtual channels. Hence, depending on the QoS guarantees [35] of the underlying communication infrastructure, all streamed data can be received successfully (or not) by the designated consumers. Regarding synchronization semantics, consumers receive asynchronously each arriving piece of data.

DS API. Our DS model abstracts common semantics widely found in data streaming protocols and related middleware platforms. This semantics is supported by the DS primitives and their parameters listed in Table 4. As already pointed out, the pair `<producer, stream_id>` is unique for each stream. The parameters `producer` and `consumer` are the identifiers of the corresponding peers. Finally, the `lifetime` parameter stands for the validity of each piece of pushed data in time. We detail next the DS API primitives:

open_stream: it is executed by the consumer to request the establishment of a session with the producer.

open: it is executed at the producer's side to handle the incoming `open_stream` requests (characterized by the producer's address and the `stream_id`). For each request, the *on_open() callback is triggered.

Interaction	Role	DS Primitives
one-way	Producer	**push**(consumer, stream_id, data, lifetime)
	Consumer	**accept**(producer, stream_id, *on_accept()) **on_accept**(data)
two-way stream	Consumer	**open_stream**(producer, stream_id) **accept**(producer, stream_id, *on_accept()) **on_accept**(data) **suspend_stream**(producer, stream_id) **resume_stream**(producer, stream_id) **close_stream**(producer, stream_id)
	Producer	**open**(producer, stream_id, *on_open()) **on_open**(producer, stream_id) { **push**(consumer, stream_id, data, lifetime) } **suspend_stream**(stream_id) **resume_stream**(stream_id) **close_stream**(stream_id)

Table 4: DS model API.

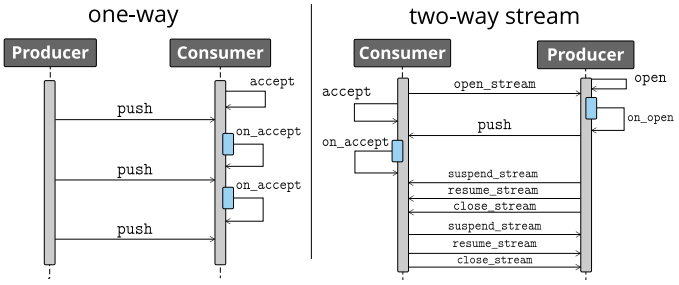

Figure 7: DS sequence diagram.

on_open: it is executed at the producer's side to establish a requested dedicated session with a consumer and start pushing the related `data` flow.

push: it is executed at the producer's side for the transmission of a `data` piece semantically qualified by the `stream_id`. This data piece is valid for the `lifetime` period.

accept: initiates the asynchronous reception of a `data` flow at the consumer's side related to the pair `<producer, stream_id>`. To this end, it specifies the associated `*on_accept()` callback.

on_accept: it is executed upon each data reception at the consumer's side.

suspend_stream: suspends the data flow. It can be executed at both the consumer's and producer's side.

resume_stream: resumes the previously suspended data flow. It can be executed at both the consumer's and producer's side.

close_stream: terminates the data flow. It can be executed at both the consumer's and producer's side.

DS sequence diagrams. In Fig. 7, we provide the DS sequence diagrams that represent a more detailed view of the supported *one-way* and *two-way stream* interactions using the above primitives. Particularly, each interaction type is specified as follows:

one-way: this interaction assumes that the dedicated

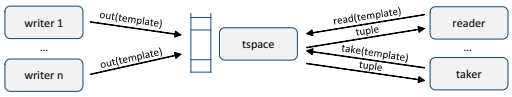

Figure 8: TS semantics.

session between the consumer and producer is already established. Thus, the producer starts transmitting the `data` flow associated to the corresponding `stream_id` using multiple `push` primitives. At the consumer's side, the `accept` primitive initiates the data flow reception by setting up the `on_accept` callback. Similarly to what was said about the PS one-way interaction type, we focus here on the transmission of a single piece of data from the producer to the consumer and classify this under the same interaction type. This can enable, for example, a DS consumer to receive data in the form of items sent by a CS server or of events published by a PS publisher. In the same way, a DS producer can push data to a CS client or a PS subscriber.

two-way stream: in this interaction, initially the consumer executes an `open_stream` primitive to request a stream of data from the producer. Once the request is accepted, the `accept` primitive is executed to set up the `*on_accept` callback for receiving the requested stream of data. At the producer's side, the `open` primitive is executed to enable receiving requests for the establishment of dedicated stream sessions. Once a dedicated session is established via the `on_open` primitive, the producer transmits the data flow using multiple `push` primitives. Finally, both sides are able to `suspend`, `resume` and terminate (`close`) the session. The two-way stream interaction type represents the complete interaction between a consumer and a producer. As the same interaction type was identified also in PS interactions in the previous section, DS producers can potentially mix and interact with PS subscribers, and DS consumers with PS brokers.

3.4. Tuple Space Model

The *Tuple Space (TS)* interaction paradigm is commonly used for data sharing among multiple read/write peers. Tuple space middleware protocols, such as SemiSpace [4], GigaSpaces [39], JavaSpaces [32] etc., are based on the TS paradigm. By relying on TS protocols, system designers are able to build IoT applications that exploit effectively the Things' energy resources [10]. The definition of our TS model is based on the classic tuple space semantics as introduced by the Linda coordination language [40]. In TS, multiple peers interact via an intermediate entity with a tuple space (`tspace`, see Fig. 8). Peers can write (`out`) data into the `tspace` and can also synchronously retrieve data from it, either by reading (`read`) a copy or removing (`take`) the data. Data take the form of tuples; a `tuple` is an ordered list of typed elements. Data are retrieved by matching based on a tuple `template`, which may define values or expressions for some of the elements.

TS semantics. Similarly to PS space coupling semantics, in TS, interacting peers write and read/take data from the tuple space independently and with no knowledge of each other. As for time coupling semantics, TS peers can act without any synchronization. In comparison to PS, peers do not need to subscribe for data, they can retrieve data spontaneously and at any time. Nevertheless, the tuple space maintains a tuple until it is removed by some peer or until the tuple expires. With respect to concurrency, peers have access to a single, commonly shared copy of a tuple. Additionally, concurrent access semantics of the tuple space is non-deterministic: among a number of peers trying to access the tuples concurrently, the order is determined arbitrarily. Hence, if a peer that intends to take specific tuples is given access to the space before other peers that are interested in the same tuples, the latter will never access those tuples. This means that not all tuples added to the space by different writers eventually reach all interested readers. In addition to the above semantics and as already pointed out, readers/takers access the tuple space by issuing synchronous queries with a `timeout`.

Interaction	Role	TS Primitives
one-way	Writer	**out**(tspace, template, tuple, lifetime)
	Tspace	**save**(*on_save()) **on_save**(template, tuple)
two-way sync	Reader	**read**(tspace, template, *tuple, timeout)
	Taker	**take**(tspace, template, *tuple, timeout)
	Tspace	**return**(*on_return()) **on_return**(reader, template, thr_id) { **out**(tuple, thr_id) }
	Tspace	**delete**(*on_delete()) **on_delete**(taker, template, thr_id) { **out**(tuple, thr_id) }

Table 5: TS model API.

TS API. We model the TS model semantics by using the primitives and their parameters listed in Table 5. The `lifetime` parameter characterizes the `tuple` availability in time. We detail next the TS API primitives:

out: executes the emission of a `tuple`, semantically qualified by a `template`, to the `tspace` (by a writer) or to a reader/taker (by the `tspace`).

on_save: it is executed when a new `tuple` is inserted into the `tspace`. To enable the acceptance of tuples, the `save` primitive must have been previously executed by the `tspace`.

read/take: execute a synchronous request for retrieving tuples that match a provided template. `take` additionally removes the tuples from the `tspace`.

return/delete: they are executed at the `tspace`'s side to initiate the handling of the incoming read/take requests for tuples matching a given `template`. For each read/take request, the corresponding `*on_return/*on_delete` callback is triggered for providing in reply the corresponding tuples (using the `out` primitive).

TS sequence diagrams. In Fig. 9, we provide the TS sequence diagrams that represent a more detailed view of the supported *one-way* and *two-way sync* interactions using the above primitives. In particular, each interaction type is specified as follows:

one-way: differently from the one-way interaction type identified for PS, here the flow of tuples from a writer to a

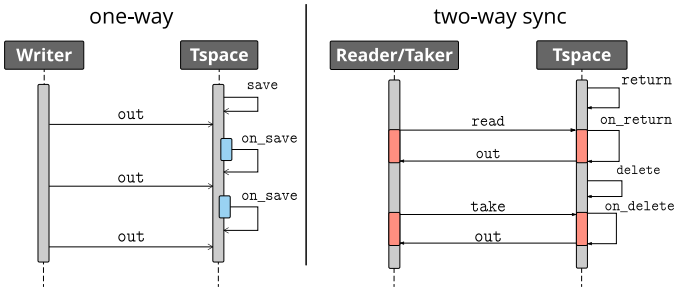

Figure 9: TS sequence diagram.

reader/taker is not one-way end-to-end, as readers/takers have to issue a query each time they wish to retrieve data from the `tspace`. Hence, we limit the one-way interaction type to interactions between writers and the `tspace`. Thus, a writer inserts tuples that match a specific template using the `out` primitive. At the `tspace`'s side, the `save` primitive enables the insertion of tuples and sets up the `*on_save()` callback in order to store the incoming tuples. Given the identification of TS one-way interactions, TS writers and `tspace`s can potentially interconnect with corresponding entities of CS, PS and DS one-way interactions.

two-way sync: for such an interaction, a reader/taker executes the corresponding primitive (`read/take`) for requesting tuples matching a specific `template`. At the `tspace`'s side, the retrieval (plus removal in the case of `take`) of tuples is enabled via the `return` or `delete` primitives. Then, every `read/take` request is received via the corresponding `on_return` or `on_delete` callback which provides back the requested tuples with the `out` primitive, by employing the thread identifier that associates the response to the request in the same way as for CS. For the two-way sync interaction type, we make abstraction of the potentially multiple writers that feed the `tspace`. Besides TS, this interaction type was also identified inside CS interactions. Hence, TS readers and takers can mix and interact with CS servers, and TS `tspace`s with CS clients.

4. Data eXchange (DeX) Connector Model

Given the above four core models (CS, PS, DS and TS), we now introduce the *Data eXchange (DeX) connector model*. As already pointed out, the above models represent the semantics of the majority of existing middleware protocols. Our objective is to devise a generic connector that comprehensively abstracts and represents the semantics of the various middleware protocols that follow the four core models. Based on the DeX abstraction, we will later introduce our middleware protocol interoperability solution.

To define the behavioral semantics of our DeX connector, we identify two main high-level API primitives:

- (i) `post` employed by a peer for sending data to one or more other peers.
- (ii) `get` employed by a peer for receiving data.

We note here that `post` and `get` are not primitives of a new concrete protocol. They are abstract primitives that can represent corresponding concrete primitives of the various existing or possibly to-come protocols. For example, a PS `publish` primitive can be abstracted by a `post`. We then create a number of variations of these primitives in order to satisfy the various interaction type semantics of our CS, PS, DS and TS models. We identify space coupling semantics for the DeX connector by appropriately mapping among the space coupling semantics of the core models. For instance, we define the essential interaction element for DeX to be `message`, which can represent any one of CS `item`, PS `event`, DS `data` or TS `tuple`.

Interaction	Role	DeX Primitives
one-way	Sender	**post**(destination, scope, post_message, lifetime)
	Receiver	**mget**(scope, *on_get()) **on_get**(source, get_message) **end_mget**(scope)
		xmget(source, scope, *on_xget()) **on_xget**(get_message) **end_xmget**(source, scope)

Table 6: DeX one-way interaction.

Below, we introduce the complete API for DeX. Conceptually, the DeX API primitives are (abstractly) employed by application-level Things that run on top of diverse middleware protocols abstracted by the DeX connector.

4.1. Data eXchange (DeX) API

Similarly to section 3, our DeX API is defined using a C-like syntax. For each one of the interaction types: one-way, two-way async, two-way sync and two-way stream, the corresponding API is provided in Tables 6,7,8 and 9. It also distinguishes between the two roles involved in an interaction type, such as: *sender* and *receiver*, *client* and *server*, *consumer* and *producer* as described in section 3. To demonstrate how DeX can represent any middleware protocol that follows one of the core interaction paradigms (i.e., CS, PS, DS and TS), we map the API of our core models to the DeX API.

DeX One-Way. The DeX API that supports one-way interactions is listed in Table 6. These represent CS, PS, DS and TS one-way interactions. In particular, peers that play the *sender* role, i.e., CS server, PS publisher, DS producer and TS writer, transmit messages using the primitive `post`. This is mapped to the CS `send`, PS `publish`, DS `push` and TS `out` primitives. The `destination` parameter corresponds to the logical identifier (name or address, e.g., URL) of the *receiver* (i.e., client, broker, consumer and tspace) as supported by its concrete middleware protocol. The `scope` parameter is used to unify identification for the specific CS `operation`, PS `filter`, DS `stream_id` and TS `template`. The `post_message` parameter embeds the corresponding `item`, `event`, `data` or `tuple`. Finally, the `lifetime` parameter is similar to the same parameter of any core model.

At the receiver's side, messages can be received using the following primitives:

mget: initiates the reception of multiple messages from possibly multiple peers. In CS, this is mapped to, e.g., a

client's `receive` primitive for multiple messages that come asynchronously from multiple servers for a specific `operation`. In PS, it corresponds to, e.g., a subscriber's `listen` primitive that receives events from multiple publishers. Finally in TS, it corresponds, e.g., to the `save` primitive which stores tuples coming from multiple writers to the tuple space.

xmget: initiates the reception of multiple messages from an exclusive source. In DS, this is mapped to, e.g., a consumer's `accept` primitive that accepts multiple data asynchronously from a specific `<producer, stream_id>`.

The above `mget` and `xmget` primitives set the `*on_get()` and `*on_xget()` callback functions, respectively, for performing the asynchronous reception of a single message. Finally, the `end_mget` and `end_xmget` primitives are used, respectively, to terminate the reception of messages.

DeX Two-Way Async. The DeX API that supports two-way asynchronous interactions is listed in Table 7. In CS, these interactions are executed using the `request`, `receive`, `on_receive` and `send` primitives (see Fig. 3). In DeX, we map these primitives as follows: (i) the client executes a request using the `post` primitive; (ii) upon the emission of the `post` primitive, the `*on_xget()` callback is set for receiving the response. `on_xget` handles the reception of a single message from an exclusive source (server); (iii) at the server's side, `mget` enables the reception of multiple requests from multiple clients via the `*on_get()` callback; (iv) finally, the server receives the request and sends back the response using the `post` primitive. Similarly to CS, the association between the request and the response is achieved via the `<source,scope>` parameter pair or, in the case of multiple parallel interactions of the same `scope` between the client and the server, via an application-level unique identifier passed in the parameters `post_message` and `get_message`. Finally, it is worth noting that the client's workflow is not blocked after the emission of the `post` primitive.

DeX Two-Way Sync. DeX two-way synchronous interactions are supported using the API listed in Table 8. Unlike two-way async interactions, the client's processing is blocked until the interaction is complete. By employing the primitive `post_xtget`, the client sends a request to a server and receives a reply from the same server within a `timeout` period. The server employs the middleware-level thread identifier that associates the response to the request (in the same way as already discussed for CS and TS).

With regard to our core models, the presented API supports CS and TS two-way sync interactions. In CS, the `request`, `receive`, `on_receive` and `operation` primitives and parameters are mapped to the `post_xtget`, `mget`, `on_get` and `scope` primitives and parameters in DeX. In TS, based on the API of Table 5, each reader/taker takes the client's role and the tspace takes the server's role. At the reader/taker's side, the `read` primitive corresponds to the `post_xtget` primitive. At the server's side, the `return/delete` and `on_return/on_delete` primitives correspond to the `mget` and `on_get` primitives.

DeX Two-Way Stream. DeX two-way stream interactions are supported using the API listed in Table 9. This API can be mapped to PS and DS stream interactions (Table 3 and 4). Accordingly, at the consumer's side, the `post` primitive includes the `OPEN_FLOW` and `flow_qualifier` parameters for representing the PS `subscribe` and DS `open_stream` primitives. In particular, the `flow_qualifier` parameter corresponds to the PS `filter` and DS `stream_id` parameters. To initiate the callback for receiving the requested stream (or flow) of messages, the `xmget` primitive is executed, which corresponds to PS `listen` or DS `accept`. Messages are received using the primitive `on_xget`, which corresponds to PS `on_listen` or DS `on_accept`.

At the producer's side, requests for a new stream are handled via the `mget` primitive (and the `on_get` callback) qualified with the `OPEN_FLOW` parameter. Once a stream session has been established, multiple messages are sent to the consumer with the `post` primitive, which corresponds to PS `publish` and DS `push`. It is worth noting that usually both peers are able to `suspend`, `resume` and `close` the flow via the corresponding primitives. While this is the case for the majority of DS protocols, in PS, only the subscriber can handle the stream via `listen`, `end_listen`

Interaction	Role	DeX Primitives
two-way async	Client	**post**(destination, scope, post_message, lifetime, *on_xget()) **on_xget**(get_message)
	Server	**mget**(scope, *on_get()) **on_get**(source, get_message) **post**(source, scope, post_message, lifetime)

Table 7: DeX two-way asynchronous interaction.

Interaction	Role	DeX Primitives
two-way sync	Client	**post_xtget**(destination, scope, post_message, *get_message, timeout)
	Server	**mget**(scope, *on_get()) **on_get**(source, get_message, thr_id) { **post**(post_message, thr_id) }

Table 8: DeX two-way synchronous interaction.

Interaction	Role	DeX Primitives
two-way stream	Consumer	**post**(destination, flow_qualifier, OPEN_FLOW, 0) **xmget**(destination, flow_qualifier, *on_xget()) { **on_xget**(get_message) } **end_xmget**(destination, flow_qualifier) **suspend_flow**(destination, flow_qualifier) **resume_flow**(destination, flow_qualifier) **close_flow**(destination, flow_qualifier)
	Producer	**mget**(OPEN_FLOW, *on_get()) **on_get**(source, flow_qualifier) { {...**post**(source, flow_qualifier, post_message, lifetime)...} **end_mget**(flow_qualifier) } **suspend_flow**(flow_qualifier) **resume_flow**(flow_qualifier) **close_flow**(flow_qualifier) }

Table 9: DeX two-way stream interaction.

DeX	CS	PS	DS	TS
post	send	publish	push	out
get	receive	listen	accept/open	save/ret./del.
scope	operation	filter	stream_id	template
message	item	event	data	tuple

Table 10: Primitives of core models mapped to DeX primitives.

and `unsubscribe`.

Table 10 summarizes the mapping between DeX and CS, PS, DS, TS with respect to the main primitives and their parameters.

5. DeXMS: Data eXchange Mediator Synthesizer

In this section, we introduce the *Data eXchange Mediator Synthesizer* (DeXMS) framework. DeXMS supports the synthesis of *mediators*, also called connector wrappers or mediating adaptors [24], that seamlessly interconnect Things employing different interaction protocols at the middleware level, e.g., REST, CoAP, MQTT, WebSocket, etc. The architecture of the proposed mediators is inspired by the ESB paradigm [21]. In this paradigm, a common intermediate bus protocol is used to facilitate the interconnection between applications employing diverse middleware protocols: instead of implementing all possible conversions between the protocols, a developer needs only to implement the conversion of each protocol to the common bus protocol, thus considerably reducing the development effort. This conversion is done by a component associated to each application and its middleware, called a *Binding Component*, as it binds the application to the service bus. Likewise, a mediator binds a Thing to the common protocol of an IoT application. We call this mediation approach *indirect mediation*.

As depicted in Fig. 1, a common protocol is used in the TIM system to facilitate the exchange of data between the participating Things. Each Thing whose middleware protocol is different from TIM's common protocol requires a mediator for taking part in TIM. A more detailed view that includes the mediators is depicted in Fig. 10a, showing a case of interconnection in the TIM system. In this scenario, a `vehicle-device` publishes messages through the MQTT middleware protocol, while an `estimation-service` receives messages through the REST protocol. Mediator 1 is associated to `vehicle-device`, while mediator 2 is associated to `estimation-service`. We select CoAP to be the common protocol of the IoT application. Accordingly, mediators 1 & 2 perform bridging between MQTT and REST, respectively, through CoAP.

To enable such bridging, a mediator employs the same (or symmetric, e.g., client vs. server) middleware protocol as its associated Thing (MQTT/REST), and all mediators use a library implementing the common protocol (CoAP), as shown in Fig. 10a. Furthermore, a mediator contains the *mediator logic*, which maps between the primitives of the bridged protocols. To enable such mapping, we rely on the DeX connector model. More specifically, each end-to-end interaction using the same middleware-layer protocol (in our example, REST following the CS interaction paradigm, MQTT following PS, and CoAP following CS) is modeled and abstracted by a DeX connector. This facilitates the mapping, which is thus performed at the DeX abstraction level. Note that our mediators perform middleware-layer data type (e.g., XML to JSON) and primitive (e.g., CS `send` to PS `publish`) conversions. Application-layer heterogeneity issues (e.g., differences in application data syntax and semantics) are beyond the scope of this paper. In our solution, we deal in a simple way with such issues by performing one-to-one mapping between application data through configuration files. More advanced solutions from the literature can be easily integrated.

Based on the above architecture, any heterogeneous Thing that employs a middleware protocol applying one of the CS, PS, DS and TS interaction paradigms, abstracted by DeX, can be connected to the common protocol. Furthermore, since the common protocol is abstracted by a DeX connector in the same way as a Thing's protocol, potentially any protocol can be introduced as the common protocol. Let's take for example the case of an IoT application that leverages a distributed message broker (e.g., RabbitMQ [36]) for data collection and routing. Things connect to push/pull data to/from a network of brokers; brokers interact with each other by relying on a specific protocol (e.g., AMQP). Such a protocol can be taken as the common protocol. Then, based on DeX and by deploying approriate mediators, any heterogeneous Thing (possibly employing a middleware protocol other than AMQP) can be connected to the network of message brokers.

In a similar way to indirect mediation, we also support *direct mediation* between heterogeneous Things. As depicted in Fig. 10b, a mediator is associated to both `vehicle-device` and `estimation-service`. This mediator performs direct bridging between MQTT and REST, again by relying on the DeX connector abstraction. We note here that the DeX connector & API model (as well as the CS, PS, DS, TS core paradigm models represented by DeX) abstract the most common characteristics of middleware protocols related to interaction semantics (e.g., synchronous, asynchronous, etc. interactions). Hence, the resulting mediators solve data exchange interoperability issues. Additional protocol features related to security, reliability, etc., can be manually included as extensions to the mediators.

In what follows, we introduce further the DeXMS framework. By relying on the principles discussed in this section, DeXMS provides a systematic solution for the synthesis of mediators. This enables application developers to integrate heterogeneous Things inside IoT applications in a automated manner. A key element for interconnecting heterogeneous Things is to be able to describe their application interfaces in a unifying way, similarly to what the service oriented computing paradigm has achieved for service oriented applications. In the next section, we elicit an interface description language that

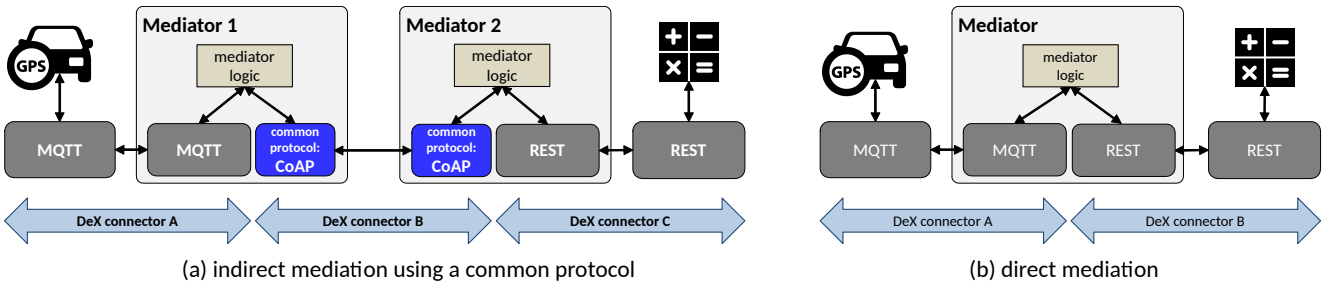

Figure 10: DeXMS mediators end-to-end runtime architecture.

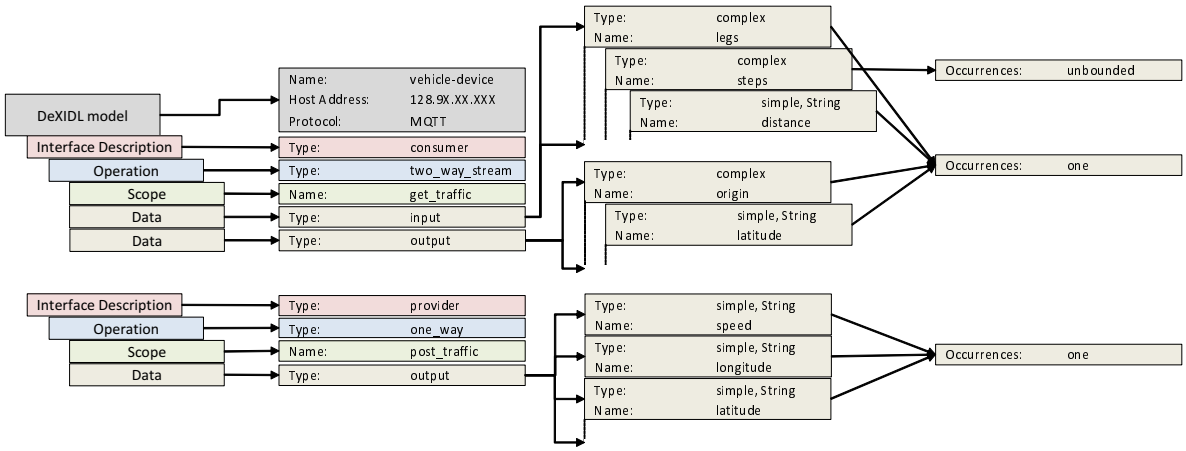

Figure 11: Extract of the DeXIDL description for vehicle-devices of the TIM system.

can be used to describe a Thing's concrete interactions by relying on the DeX abstraction.

5.1. DeX Interface Description Language (DeXIDL)

Service Oriented Architecture (SOA) enables the interaction of software components in standard ways. Interactions are realized using well known protocols, such as SOAP and REST; and each service exposes its functionalities (operations, messages, etc.) by relying on XML-based standard interface descriptions (WSDL/WADL). The existence of standard interface descriptions facilitates development of frameworks and wrapping of systems for interoperability. However, with the advent of the IoT, major tech industry actors have introduced their own APIs and protocols to support the deployment of Things. Accordingly, there are very few efforts to specify standard interface descriptions that represent physical objects in the real world [41]. This lack hampers interconnection between heterogeneous Things in IoT applications.

To facilitate the automated synthesis of mediators for interconnecting heterogeneous Things, we propose a generic interface description for Things, which we call *Data eXchange Interface Description Language* (DeXIDL). A DeXIDL interface description corresponds to a Thing that employs any middleware protocol which can be abstracted as a DeX connector. DeXIDL enables the definition of operations provided or required by a Thing that follow the interaction types and roles identified in sec:core-paradigms. Besides an operation's type, the names and data types of its parameters are also specified. The description is complemented by the logical identifier (e.g., URL) of the Thing. An example extract of such an interface description is depicted in fig:vehicle-dexidl, which defines the interaction semantics of a vehicle-device of the TIM system. In particular, a vehicle-device can take the role of both a *consumer* and a *provider*. When acting as provider, the interaction type is *one_way* and the device pushes data using the *post_traffic* scope. This scope corresponds to the abstracted PS topic qualifying the data, as the device employs the MQTT protocol to push data. As shown for the provider and consumer roles, the input and output parameters of operations can be simple Java types or complex types defined using Java classes. Moreover, the occurrences of simple or complex data are defined – for example an arbitrary list of values can be defined as unbounded.

To specify DeXIDL, we have created a metamodel using the *Eclipse Modeling Framework* (EMF) [42]. This metamodel enables us to generate a set of data structures related to a Thing from its DeXIDL description, which express the Thing's operations, input/output data, etc. in JSON format. These data structures are then leveraged to build a mediator for this Thing. The DeXIDL metamodel can be found in the appendix:gidl. To facilitate the definition of a DeXIDL model (i.e., the DeXIDL description of a specific Thing), we have developed an Eclipse plugin[1] using EMF tools. Application developers can install this

[1] `https://gitlab.inria.fr/zefxis/dex-idl`

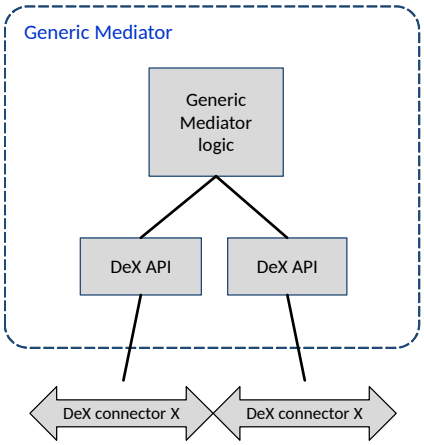

Figure 12: Generic Mediator architecture.

plugin into their favorite Eclipse package and use it to create a DeXIDL model via the included graphical editor. The complete DeXIDL model for vehicle-devices specified using our Eclipse plugin can be found in the Appendix A.

Towards the systematic synthesis of mediators, besides DeXIDL, we also elaborate a generic architecture for mediators as well as a pool of protocol-related components for the customization of generic mediators. We introduce those in the next section.

5.2. Generic Mediator and Protocol Pool

By relying on the DeX abstraction of the protocols bridged by a mediator, we design and build the architecture of a mediator at an abstract level, which we call a *Generic Mediator*, as shown in fig:generic-architecture. A Generic Mediator performs bridging between two instances of the DeX connector (X and Y in the figure), to each of which it connects through the DeX API. The bridging functionality is implemented by the Generic Mediator's logic, which is a set of DeX primitive-rules of the type:

if *get_primitive* *received_on* *DeX connector X*(*Y*),

then *execute_symmetric_post_primitive_on* *DeX connector Y*(*X*)

The association between `get` and symmetric `post` primitives is based on the DeX API and the DeX interaction types.

To enable customization of the Generic Mediator, we have developed the *Protocol Pool*. This contains implementations of the DeX API for concrete middleware protocols. Each such implementation realizes one interaction type (e.g., one-way) supported by a concrete protocol, with the DeX API, in a programmatically efficient way. This is done by mapping the specific protocol's primitives and semantics for this interaction type to primitives and semantics of the DeX API. Additionally, it implements conversion between the protocol's data types (e.g., a message in JSON format) and the DeX data types. We develop these DeX API implementations as generic code excerpts in Java.

To enable DeXMS support for an IoT protocol, DeX API implementations for this protocol's interaction types need to be added to the Protocol Pool. As an example, we show how a developer can introduce support for MQTT into the Protocol Pool by following the steps below:

1. Classify MQTT under one of the core interaction paradigms. MQTT follows the PS interaction paradigm (§3.2).
2. Identify the interaction types supported by the corresponding paradigm. PS supports *one-way* and *two-way stream* interactions (§3.2).
3. Identify the DeX primitives that implement the interaction types found in the previous step. These are listed in Tables 6 and 9 for one-way and two-way stream interaction types, respectively.
4. Implement the identified DeX primitives by using MQTT's primitives. In Listing 1, we show how to implement the DeX one-way interaction type by using the MQTT Eclipse Paho Java library.
5. Implement conversion between MQTT and DeX data types. In Listing 1, we use the `buildMqqtEvent()` method to convert a DeX message to an MQTT event and the `buildDexMessage()` method to do the inverse conversion.

```
/* MQTT publisher and subscriber are already instantiated */

/* Sender's one-way post DeX primitive */
public void
      postOneway(String dest, Scope sc, DexMessage dexMsg) {
  MqttMessage mqttEvent
        = new MqttMessage(); // MQTT event instantiation
  /* the DeX message must
        be first converted to an MQTT compatible data type */
  mqttEvent.setPayload(msgBuilder.buildMqttEvent(dexMsg));
  mqttSender.publish(sc
      .getName(), mqttEvent); // event published to a topic
}
/* Receiver's one-way mget DeX callback */
public void mgetOneway(Scope sc) {
  mqttReceiver
      .connect(); // MQTT receiver connects to the broker
  mqttReceiver.subscribe(sc
      .getName()); // subscribes to the specific topic name
  mqttReceiver.setCallback
      (new MqttCallback() { // new callback to handle events
  @Override
  public void
       messageArrived(String topic, MqttMessage mqttEvent) {
    dexMsg = msgBuilder.buildDexMessage
        (mqttEvent); // MQTT event to DeX message
    onGet(dexMsg); // new onGet DeX callback
  }});
}
/* Receiver's one-way onget DeX primitive */
public void onGet(DexMessage dexMsg) {
/* DeX message
     received to be delivered to the mediator logic */
}
```

Listing 1: Abstracting MQTT one-way interactions using the DeX API.

Finally, in the next section, we leverage the Generic Mediator, the Protocol Pool and the DeXIDL metamodel to support automated synthesis of mediators.

5.3. DeX mediator synthesis

Development of mediators is a tedious and error-prone process, which can highly benefit from automated systematic support. Moreover, an automated mediator synthesis process is essential for IoT applications that require dynamic, runtime composition of heterogeneous Things without human intervention. Our DeXMS framework[2] (initially developed under the name of *VSB*) [22, 43] can be leveraged by application developers to support the automated execution of mediator synthesis.

To present the mediator synthesis process implemented by DeXMS, we rely on the example of end-to-end interaction between a vehicle-device and the estimation-service

[2] `https://gitlab.inria.fr/zefxis/dexms`.

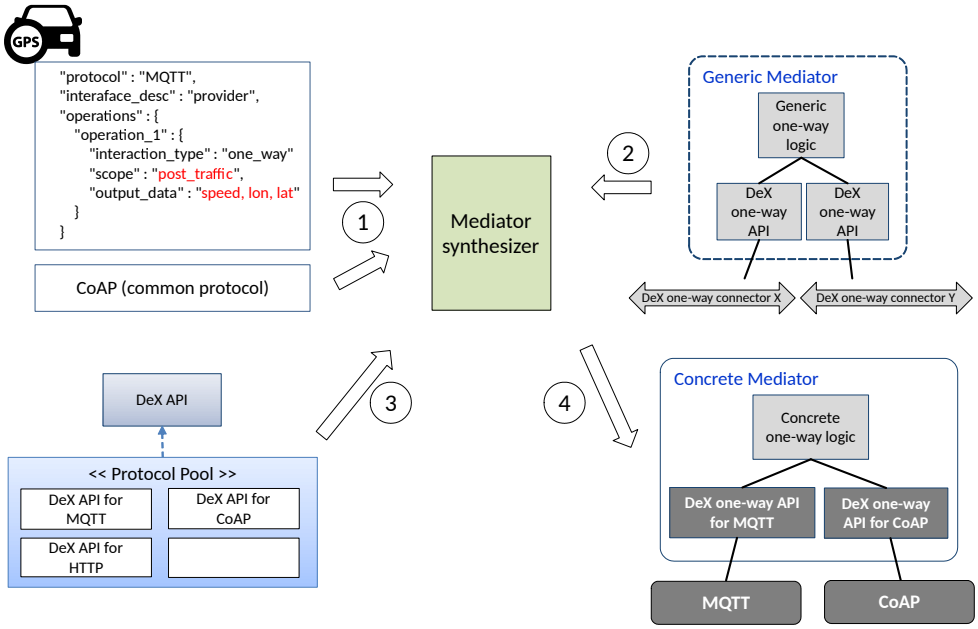

Figure 13: DeX mediator synthesis process.

of the TIM system via indirect mediation, as depicted in Fig. 10a. In particular, we consider one-way interactions between the two entities. The steps of the process concerning the mediator associated to the vehicle-device are shown in Fig. 13.

In Step 1, the DeXIDL model of the vehicle-device (the part concerning one-way interactions) and the identification information of the selected common protocol (CoAP) are provided as input to the *Mediator Synthesizer*. In particular, the DeXIDL model in Fig. 13 states that the vehicle-device employs MQTT to push data (speed, longitude, latitude) qualified by the `post_traffic` scope (corresponding to PS topic).

In Step 2, the Mediator Synthesizer receives as input the Generic Mediator elements that correspond to the interaction type identified in the DeXIDL model of the previous step, i.e., *DeX one-way API* and *Generic Mediator one-way logic*. This enables building a minimal (and hence as lightweight as possible) mediator comprising only the necessary elements. If more interaction types need to be supported (e.g., a two-way stream interaction between vehicle-device and estimation-service), this can be included in the same way.

To complete the synthesis process, the Generic Mediator needs to be customized for the concrete middleware protocols identified in Step 1, i.e., MQTT and CoAP. Hence, in Step 3, the Mediator Synthesizer receives as input the DeX one-way API implementations for MQTT and CoAP from the Protocol Pool. The Generic Mediator needs also to be refined with the concrete application data types defined in the DeXIDL model of vehicle-device, i.e., the data structure `<speed,lon,lat>`. This refinement concerns the conversions between MQTT and DeX data types (cf. methods `buildDexMessage()` and `buildMqqtEvent()` of Listing 1) as well as the relaying of data executed by the Mediator logic (cf. DeX primitive-rules of sec:chap3:vsb:gbc). In this way, the customized Mediator is able to convert protocol primitives and data types (for the concrete data types of vehicle-device) from MQTT to CoAP by relying on the intermediate DeX primitives and data types. The resulting Concrete Mediator is thus produced in Step 4, complemented by external third-party libraries implementing the MQTT and CoAP protocols.

Besides the synthesis of the mediator associated to the vehicle-device, the mediator associated to the estimation-service has to be synthesized as well. By relying on the DeXIDL model (the part concerning one-way interactions) of the estimation-service and following the same steps as above, a customized CoAP to REST one-way mediator is produced.

Applying the approach detailed above enables end-to-end one-way interactions between the vehicle-device that publishes messages with the MQTT protocol and the estimation-service that receives messages via the REST protocol. In particular, our approach ensures proper execution of these interactions with respect to middleware-level semantics. Furthermore, application data types of both sides are properly converted to a common denominator (data types of the selected common protocol CoAP).

Nevertheless, our approach assumes that application-level semantics have already been matched between the vehicle-device and the estimation-service. As already mentioned, resolving complex differences in application data syntax and semantics is beyond the scope of this paper. We assume that there is one-to-one correspondence between application-level operations as well as their data, and we enable application developers to perform manual mapping between differing parameter names and parameter order via a configuration file. Differences in data types can also be easily tackled by deploying appropriate

data converters. We integrate these mappings into the mediator logic of one of the two mediators. As already pointed out, more sophisticated methods for dealing with application-layer heterogeneity can be sought in the literature and easily integrated into our solution.

Finally, DeXMS supports direct mediation as well, where there is no intermediate common protocol and a single mediator is deployed between two heterogeneous Things (see Fig. 10b). Such direct mediator (MQTT to REST in our example) is synthesized in a similar way as for indirect mediation. The synthesis process takes as input at once the vehicle-device and estimation-service DeXIDL models and customizes a Generic Mediator with appropriate implementations from the Protocol Pool. Application-level mapping relies again on a configuration file.

6. Assessment of DeXMS

DeXMS was originally developed as core component (under the name of VSB) of the *H2020 CHOReVOLUTION* European project [44] to support heterogeneous interactions in choreographies of services and Things. Currently, DeXMS supports the following middleware protocols: REST, CoAP, MQTT, WebSockets and DPWS (additional implementation details are provided in [22, 43]). We evaluate the DeXMS framework (development and runtime environment) with respect to two criteria: *(i)* the support offered to developers when developing a new IoT application that may contain several heterogeneous Things; and *(ii)* the runtime performance of the synthesized mediators in terms of latency as well as resource consumption on the hosting node, under both low traffic and stress conditions. We present our evaluation results in the following.

6.1. Support to Developers

In the highly diverse IoT landscape, building middleware mediators requires from a developer to be knowledgeable about the numerous APIs and protocols. Furthermore, such a development process can be particularly error prone, due to the required fine-grained, both semantic and syntactic, mapping between the specific primitives and data types of the interconected protocols.

In comparison, applying the DeX mediator synthesis process detailed in section 5.3, a developer only needs to: *(i)* ensure application-level semantic matching between the Things participating in the target IoT application and build a set of configuration files to resolve application-level heterogeneity; *(ii)* build the DeXIDL model of each participating Thing; and *(iii)* select an appropriate common protocol in the case of indirect mediation. The rest of the synthesis process is executed automatically, resulting in the generation of finalized mediators ready to be deployed. This certainly has as prerequisite the understanding of the DeX abstraction and its relation with the CS, PS, DS, TS core paradigms, which is reflected in the DeXIDL model. Once this is acquired, the developer can use the graphical editor of our Eclipse plugin to easily and rapidly fill up the information defining the semantics of a Thing in DeX terms, as depicted for the vehicle-device of the TIM system in Fig. 11 (model extract) and Fig. A.28 (complete model).

We evaluate the effectiveness of the development process provided by DeXMS by applying it to the TIM system. TIM includes several heterogeneous Things employing protocols classified under all four core interaction paradigms: fixed-sensors (WebSockets, DS), estimation-service (REST, CS), vehicle-devices (MQTT, PS) and smartphones (SemiSpace, TS). To ensure interconnections among these Things, a developer needs to build a set of mediators, one for each of the above Thing types, where we assume indirect mediation with CoAP used as the common protocol. For our evaluation, we compare the manual effort in *lines of code* (LoC) when developing mediators for the TIM system with and without DeXMS. In the case of employing DeXMS, LoC corresponds to the lines of DeXIDL description for the various Things, expressed in JSON format. In particular, we measure the rate of achieved LoC reduction when automatically synthesizing a mediator with DeXMS. This is a good indicator of the productivity gain [45] obtained with DeXMS, where we assume developers that have already some experience with DeXMS. On the other hand, we only measure code development effort without DeXMS, not including testing and debugging tasks that are essential to manual development. We use the Metrics 1.3.6 Eclipse plugin[3] to measure LoC without DeXMS.

As shown in Table 11, the DeXMS framework considerably reduces the application development effort. In particular, an application developer is able to save between 94.3% and 98.3% of manual LoC writing when building mediators for the TIM system. The estimation-service is the most complex application component for building its DeXIDL model, since there are multiple operations as well as input and output parameters to be defined. Accordingly, it requires the most LoC for manually building its mediator.

6.2. Performance Evaluation

As already presented, our interoperability solution between two Things relies on either a single mediator (direct mediation) or a pair of mediators connected through a common protocol (indirect mediation). Indirect mediation employs three native protocol connections, comprising the middleware protocols of the Things and the common protocol. So roughly it is at least three times more costly in latency than a homogeneous interaction between two Things. But, this is proper to the common protocol based approach. Therefore, the interest is in evaluating the performance of mediators (in either direct or indirect mediation) and how much overhead they add to the end-to-end latency.

On the other hand, mediators are software components that are external to the Things that we wish to interconnect. Mediators can be deployed on the same physical nodes as the Things. Alternatively, deploying them on a separate node enables seamless (as much as possible)

[3] `http://metrics.sourceforge.net`

heterogeneous Thing	LoC with DeXMS	LoC without DeXMS	LoC reduction using DeXMS (%)
`fixed-sensors`	13	765	98.3
`vehicle-devices`	44	1189	96.3
`smartphones`	45	1184	96.2
`estimation-service`	91	1597	94.3

Table 11: Development effort of the application developer.

interconnection between the Things and does not require any resources from the typically resource-constrained Things. Still, this deployment may be on an edge device which can also be resource-constrained (e.g., a mobile phone, a portable gateway device).

Based on the above, we evaluate in this section the performance of our mediation solution with two experimental scenarios:

1. We evaluate an indirect mediation scenario, where the mediators run on resource-rich nodes and communicate through two alternative common protocols. We measure latencies inside the mediators and end-to-end, under both low traffic and stress conditions. Besides the performance of the mediators, we also analyze the impact of the selected common protocol on the end-to-end mediation.
2. We evaluate a direct mediation scenario, where the mediator runs on a Raspberry Pi. Besides latencies for both low and high traffic, we also measure CPU utilization, memory footprint and energy consumption on the resource-constrained node.

Test Scenarios

For our experimental scenarios, we develop heterogeneous mock Things, that is, sender and receiver components engaging in one-way interactions, and we synthesize corresponding mediators. We utilize the supported middleware protocols of the DeXMS framework: WebSocket, REST, CoAP, MQTT and DPWS. These are protocols that are commonly used in IoT systems, while some of them have especially been introduced for the IoT [9]. To create stress conditions for the mediators, we follow the method applied in [46] and [47]. In particular, we need to create bottlenecks in the mediators for testing their maximum performance, while at the same time making sure that the sender and receiver components are below their maximum load. To be able to generate high input traffic for the mediators, the sender component is multi-threaded. By creating an appropriate number of threads, each of which produces messages at a constant rate (one every second), we can precisely control the input traffic load. At the same time, the receiver component must be able to receive thousands of messages per second.

In the first scenario, we connect a WebSocket DS producer to an MQTT PS subscriber. Data items streamed by the former are received as events of interest by the latter. In particular, the MQTT subscriber executes in "at-most-once-delivery" mode [2], which enables fast event delivery without acknowledgments. The two generated mediators communicate via the DPWS or REST CS protocol in turn as common protocol. To convey the streaming interaction

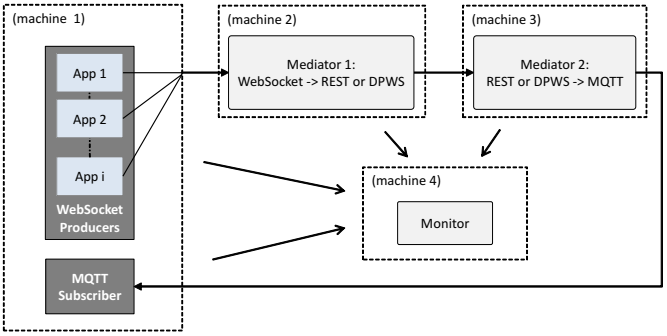

Figure 14: Test setup for the indirect mediation scenario.

traffic, we rely on one-way notifications of DPWS, while with REST we have to use the HTTP 1.1 request–response pattern for each one-way message. Nevertheless, DPWS also uses HTTP 1.1 as underlying binding, which means that the same request–response pattern is used, transparently for the application developer.

In the second scenario, we connect a WebSocket DS producer to a CoAP CS client. In particular, the latter executes in "observer" [48] and "non-confirmable" mode [1], which enable multiple responses to a single request without acknowledgments. Hence, data items streamed by the WebSocket producer are received as messages by the CoAP client.

Test Setups

The test setup for the first scenario, shown in Fig. 14, consists of four machines, connected via a local switch (GS900/8, Allied Telesis) creating a private 1000 Mb/s Ethernet local network. By relying on this network setup, we assume that there are no message losses and retransmissions at the lower transport/network layers; moreover, that these layers create no bottleneck. The first machine (M1) has an Intel Xeon CPU W3540 2.93 GHz x 4 (4 GB RAM), the second (M2) an Intel Xeon CPU W3550 3.07 GHz x 4 (8 GB RAM), the third (M3) an Intel Core i7-4600U CPU 2.10 GHz x 2 (8 GB RAM), and the last machine (M4) has an Intel Core i7-4790 CPU 3.60 GHz x 4 (8GB RAM).

In our experiments, we had to deal with the clock synchronization problem between machines. We tested the Network Time Protocol (NTP) as possible solution, which however did not produce satisfactory results for having a precision in the range of sub-milliseconds, similar to what is reported in [49]. An alternative solution would be to use network emulators like mininet or ns3; however, we insisted on evaluating our solution on a real experimental setup. Therefore, we had to take two countermeasures:

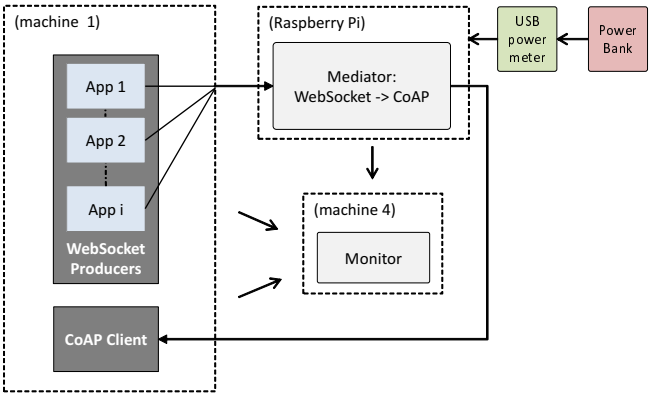

Figure 15: Test setup for the direct mediation scenario.

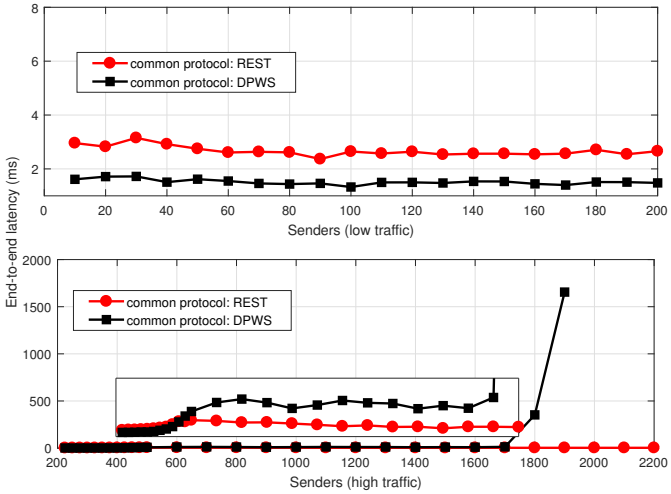

Figure 16: End-to-end latency through DPWS and REST common protocols (indirect mediation scenario).

(i) deploy the sender and receiver components on the same machine, so as to be able to measure end-to-end latency with accuracy; (ii) employ alternative solutions for identifying communication bottlenecks (measure local resource consumption on machines, as we will see next), since we could not precisely measure latency between machines. Based on the above, we deployed the sender and receiver components on M1, the first mediator on M2 and the second mediator on M3 as depicted in Fig. 14. M4 is used as monitor that collects measurement data via lightweight probes deployed on all three machines M1, M2, M3. This architecture enables minimizing the effect of monitoring on the monitored system.

In the test setup for the second scenario, shown in Fig. 15, we employ again machines M1 and M4 for hosting the sender/receiver components and the monitor, respectively. We now deploy a single mediator on a Raspberry Pi ARM Cortex A53 CPU 1.2 GHz x 4 (1 GB RAM). We use an external battery Mophie pwrstion-4k-alm 4000 mAh to power the Pi, and a USB power meter to measure energy consumption on the Pi.

Finally, we used the following third-party implementations of the middleware protocols involved in the experiments: jWebSocket, RESTlet framework, Californium project for CoAP, Eclipse Paho for the MQTT client, Apache ActiveMQ for the MQTT server, and JMEDS for DPWS.

Results

In the following, we present our experimental results for the indirect and direct mediation scenarios. For each topology and protocol setup, we run multiple experiments starting with low input traffic and increasing each time the input message rate. Things generally do not exchange very large quantities of data, so messages usually tend to be of small to average size. Hence, we set the size of messages to 284 bytes. For each experiment, we create and send a sufficient number of messages, so that measurements to be considered are taken after the end-to-end system reaches a steady state. For steady state detection, since dynamic throughput over a time window can be oscillating, we check that the incremental average throughput in [0,t] (total number of messages that have arrived at the receiver divided by t) stabilizes (as expected, to the level of the input message rate). Hence, we run each experiment for at least half an hour. While we reach pretty high application message rates in our experiments, we assume that there is no message loss. In particular, there is no loss over a communication link, as transmission is reliable: WebSockets, DPWS, REST, MQTT run on top of TCP, while CoAP runs on top of UDP but the local network link quality is good, as pointed out in the description of our Test Setups. Moreover, there is no buffer overflow in any of the sender, receiver or middleware components; we ensure this by setting the JVM heap size to a sufficient level.

Based on the previous, we measure for the indirect mediation scenario the average end-to-end message latency for the interaction between the two Things, in turn for the two selected common protocols, i.e., DPWS and REST. The results are plotted in Fig. 16. As we see in the figure, end-to-end latencies are at similar levels for the two cases (around 2 ms for DPWS and 4 ms for REST) and remain stable for a great range of input message rates, from 10 to 1700 msg/s. Then, in the case of DPWS, there is a very steep increase after 1700 msg/s, where latency reaches 1700 ms at 1900 msg/s. While in the case of REST, latency remains unchanged up to 2200 msg/s. These results clearly show a bottleneck in the end-to-end mediation in the case where the common protocol is DPWS. Since we reach a steady state in each experiment, we detect in the DPWS case a situation that is close to saturation but not saturation itself (which would produce an uncontrollable increase in the latency).

As a next step, we try to locate the identified bottleneck along the end-to-end interaction in the DPWS case. In Figs. 17 and 18, we depict the average message latencies measured inside the two mediators, i.e., combining the queueing and processing times of messages served by the mediator logic, for both the DPWS and REST cases. As we can see, both mediators behave in a perfectly scalable

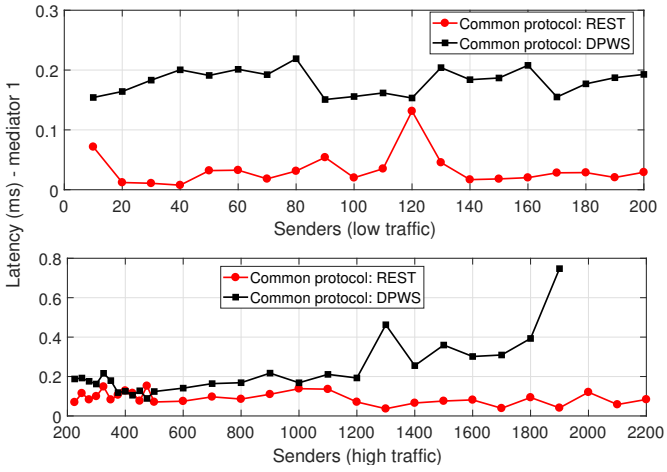

Figure 17: Mediator 1 latency (indirect mediation scenario).

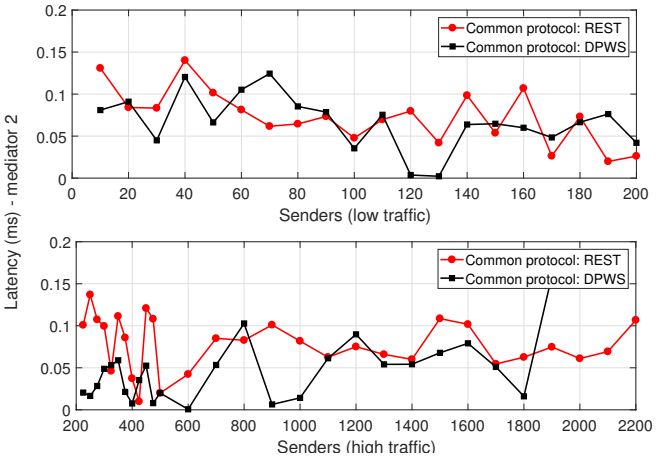

Figure 18: Mediator 2 latency (indirect mediation scenario).

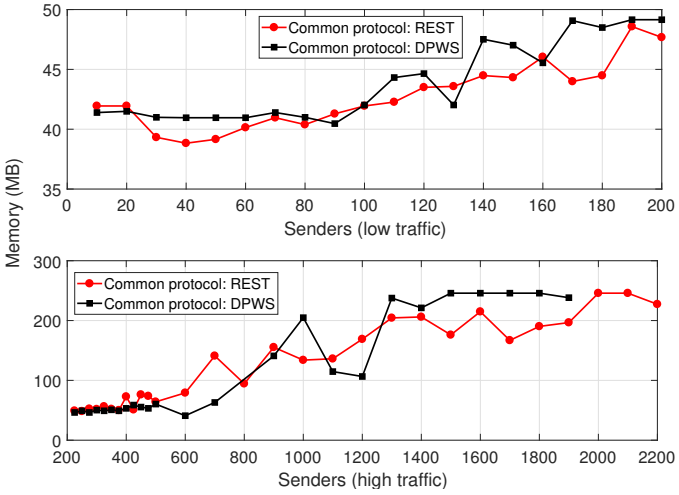

Figure 19: Memory consumption on Machine 1 hosting sender and receiver components (indirect mediation scenario).

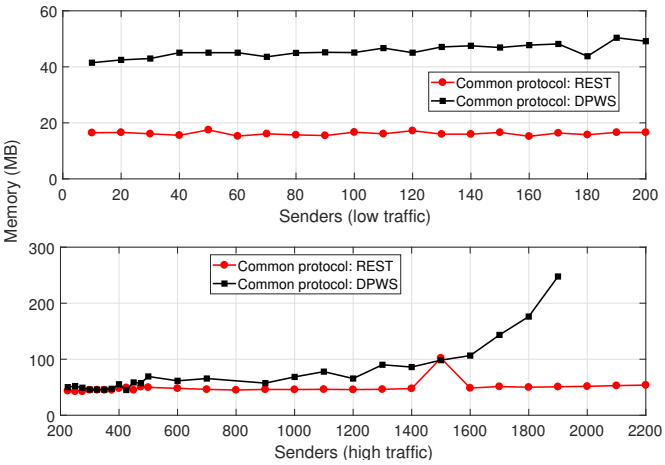

Figure 20: Memory consumption on Machine 2 hosting Mediator 1 (indirect mediation scenario).

way when the input message rate increases. This leads us to infer that the bottleneck in the DPWS case occurs when generating, sending and receiving messages over the three native protocol connections, i.e., over WebSocket, DPWS and MQTT.

As already pointed out, we cannot accurately measure individual latencies over these protocols due to the clock synchronization problem between the different machines of our experimental setup. Alternatively, we check resource consumption by the software components of our experiment on the machines. CPU consumption of our components stays at levels below 20% at any one of the machines and for all the experiments, for both DPWS and REST. This means that CPUs are not solicited more to deal with the bottleneck problem. We then check memory consumption of our components; results are depicted in Figs. 19, 20 and 21.

On Machine M3 hosting Mediator 2, memory consumption remains pretty stable, for both DPWS and REST. On M1 hosting the sender and receiver components, memory consumption is stable for low message rates and then increases starting from 700 msg/s, in the same way for the DPWS (bottleneck) and REST (no bottleneck) cases. So, this is not related to the bottleneck. It is due to the creation of sender threads at the sender component. Finally, on M2 hosting Mediator 1, memory consumption is stable for REST. For DPWS, it is stable for low message rates, then it increases 2–3 times at 1700–1900 msg/s; this is related to the bottleneck.

Hence, we locate the bottleneck at the common protocol DPWS: because of this, messages accumulate at the sender entity of Mediator 1 and increase memory consumption. We interpret this bottleneck in the following. As already mentioned, DPWS one-way notifications employ underlying HTTP requests–responses. This mandatory two-way handshake for each DPWS message creates a bottleneck at high message rates (the two sides of the handshake have to wait for one another, even if there is more available CPU). At the same time the sender component employs WebSocket streaming, which does not have the same limitation. REST also relies on HTTP requests–responses. However, the REST implementation that we employed allows up to 100 (parameter controlled by the application)

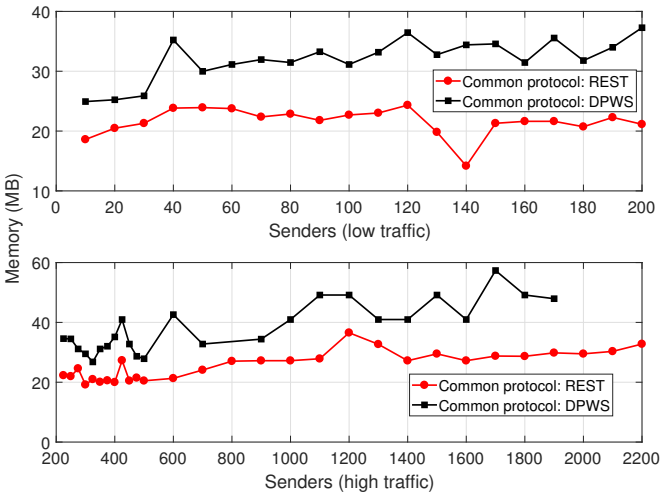

Figure 21: Memory consumption on Machine 3 hosting Mediator 2 (indirect mediation scenario).

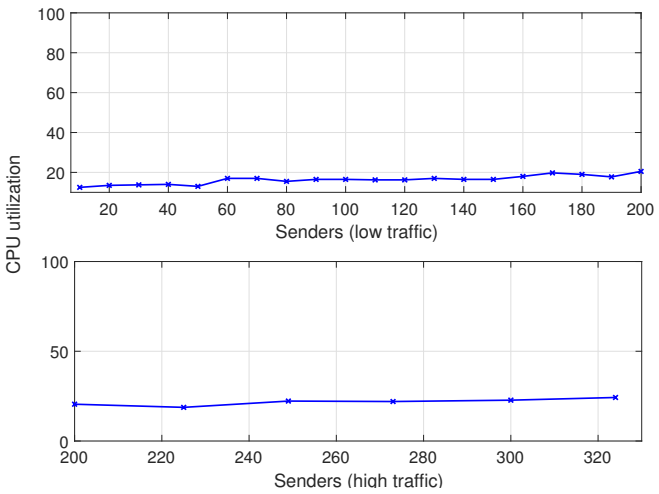

Figure 22: CPU utilization (%) on Raspberry Pi (direct mediation scenario).

parallel HTTP connections, which improves considerably the scalability of the protocol. While in the DPWS implementation that we employed, the underlying HTTP parallelism is limited to 10 connections and is not controlled by the application. Finally, same as WebSocket, MQTT (running in at-most-once-delivery mode) does not have the two-way handshake limitation of DPWS and REST.

In conclusion of our analysis, our experiments showed that, when stressing the indirect mediation setting, the mediators demonstrate a perfectly scalable behavior, whereas at some point the common protocol may become the performance bottleneck. Then, each common protocol reveals different stress-level properties in terms of communication latency. This certainly points out the importance in the choice of the common protocol, depending on the interconnected Things and their middleware protocols. Additionally, we observed that the latency inside the mediator logic was in the order of 1/30 to 1/10 of the end-to-end latency (for both common protocols), for input message rates causing moderate message queueing effects. Our evaluation of mediators shows that they introduce limited performance overhead into end-to-end interactions.

Similarly to the previous, we measure, also for the direct mediation scenario, the average end-to-end message latency for the interaction between the two Things, as well as the average message latency inside the mediator, for both low and high message rates. The results are plotted in Figs. 23 and 25, respectively. As we see, the end-to-end latency presents a very steep increase after 260 msg/s and reaches 3700 ms at 325 msg/s. At the same time, the mediator latency remains low for the same range of input rates, between 0.15 and 0.18 ms.

Similarly to the indirect mediation case, in order to interpret the observed bottleneck, we measure resource consumption on the Raspberry Pi node hosting the mediator. Our measurements of CPU and memory utilization are depicted in Figs. 22 and 26, respectively. CPU values present an expected gradual increase with the message rate, but remain below 25%. On the other hand, memory values are quite stable, around 50 KB, until 270 msg/s, but then they show a clear increase up to 60 KB for 325 msg/s. This increase in memory consumption is due to a communication bottleneck over the CoAP connection and subsequent message accumulation at the sender entity of the mediator. More specifically, CoAP has an internal throughput limitation: Each CoAP message contains a Message ID used to detect duplicates. The Message ID is compact; its 16-bit size enables up to about 250 msg/s from one endpoint to another [1].

The results of our direct mediation experiments confirm the scalability of our mediators, at least up to the point where one of the middleware protocols of the involved Things becomes the performance bottleneck. Moreover, same as in the indirect mediation case, the latency inside the mediator logic represents a small part of the end-to-end latency: less than 1/10 of the latter, for input message rates causing no saturation effects.

Besides the scalability and latency overhead of the mediator, we also evaluate next its resource impact on a resource-constrained hosting device as the Raspberry Pi. Checking again Figs. 22 and 26, we see that the mediator (including the mediator logic and the two third-party middleware protocol libraries) utilizes between 12.5 and 25% of the CPU as well as between 50 and 60 MB of memory (out of 1 GB RAM of the Pi), depending on the input message rate. Furthermore, as we show in Fig. 24, the overall energy utilized for the functioning of the mediator on the Pi ranges between 2 and 28 mAh for the 30 min duration of each experiment, again depending on the input message rate. This corresponds to between 0.05 and 0.7% of the charge capacity of the external 4000 mAh battery that we used to power the Pi or between 0.09 and 1.2% of the 2300 mAh battery of an average smartphone. Over a whole day, if the mediator serves without stopping an average input message rate of 150 msg/s, energy consumption will be around 720 mAh or 18% of the 4000 mAh battery.

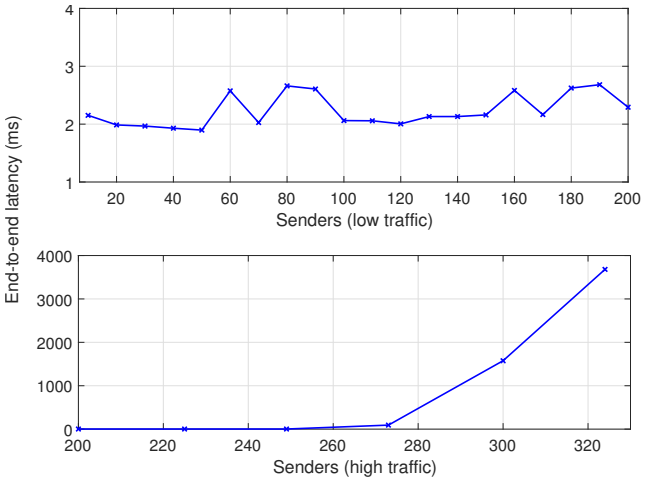

Figure 23: End-to-end latency (direct mediation scenario).

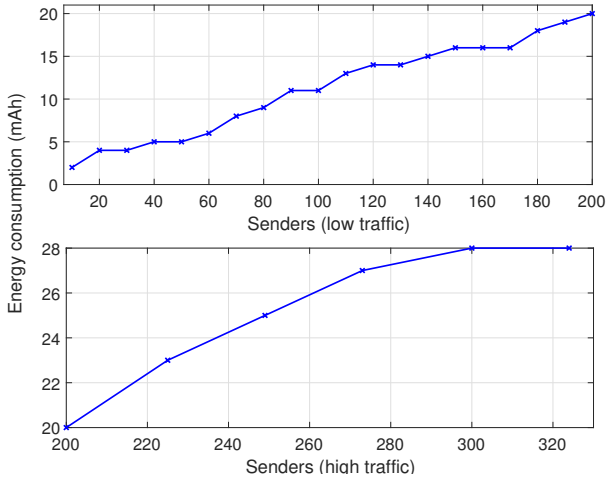

Figure 24: Energy consumption on Raspberry Pi (direct mediation scenario).

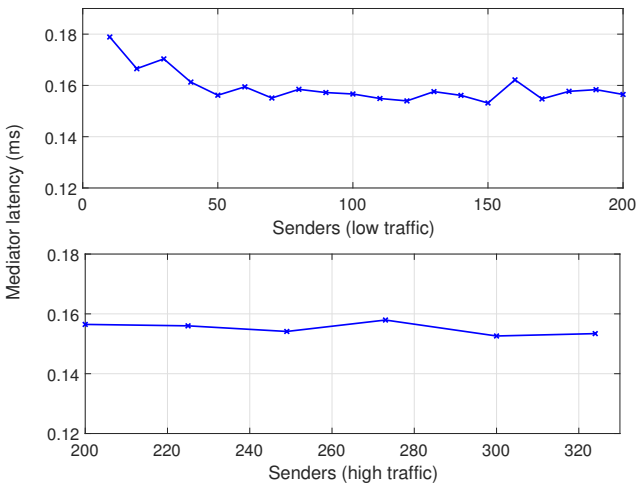

Figure 25: Mediator latency (direct mediation scenario).

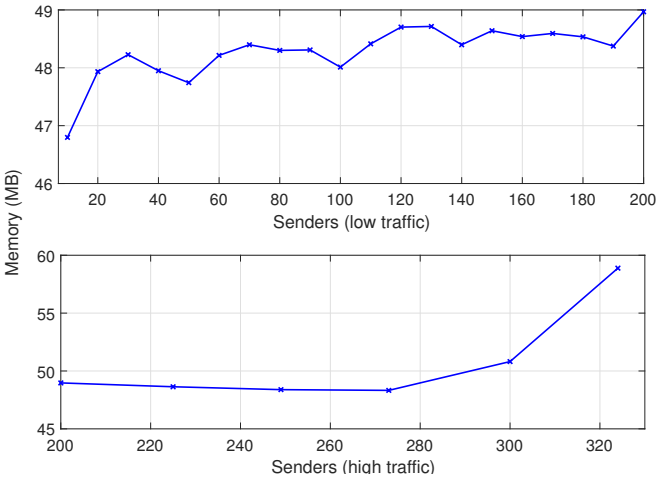

Figure 26: Memory consumption on Raspberry Pi (direct mediation scenario).

7. Related Work

Enabling interoperability between heterogeneous distributed systems (different hardware, OS, programming languages) has been a long-standing problem. One of the early efforts to establish a common system specification framework towards interoperability was the Reference Model of Open Distributed Processing (RM-ODP) [50], result of the combined effort of the ISO and ITU-T standards organizations. RM-ODP introduces a set of abstract viewpoints for specifying a distributed system, without identifying any concrete system architecture or technology. Some of its viewpoints establish the essential distinction between computational objects and underlying protocol bindings, found in all later middleware-based efforts for distribution-transparent system development and integration. Furthermore, RM-ODP introduces a canonical model of interaction between computational objects to represent all patterns of communication in distributed systems. This canonical model identifies three interaction styles between a pair of computational objects: *signal*, corresponding to one-way communication; *operation*, mostly concerning a two-way invocation; and *stream*, representing a sequence of interactions or information flow. The above abstractions certainly resemble some of the interaction types that we identified across the four core interaction paradigms and which constitute the basis of the DeX connector model.

As brought up in the previous paragraph, interoperability has been one of the main goals of the middleware research community. Conformance to *standards-based* approaches that introduce common protocols and interface description languages, such as CORBA [51], DCOM [52] and Web Services [53], has been the first attempt of the community. This resulted in distributed systems employing multiple standards and, hence, a layer of common protocol and data format heterogeneity was introduced. Therefore, to resolve heterogeneity of middleware solutions, various approaches were applied: *(i) software bridges*, such as OrbixCOMet [54] and SOAP2CORBA [55], to achieve one-to-one mapping between different protocols; *(ii) intermediary-based* solu-

		Supported protocols	Direct bridging	Common protocol	Software abstractions	Constrained devices	Mediator synthesis
SOA	Swarm [18, 61]	1	✗	✓	✗	✓	✗
	LISA [17, 62, 63]	1	✗	✓	✗	✓	✗
	Cheng et al. [64]	1	✗	✓	✗	✗	✗
	Tao et al. [65]	3	✗	✓	✗	✓	✗
	Ismail et al. [66]	1	✗	✓	✗	✓	✗
	Derhamy et al. [67]	3	✓	✗	✓	✓	✗
	XWARE [68]	2	✓	✗	✓	✓	✗
Gateways	QEST [69]	2	✓	✗	✗	✓	✗
	Han et al. [70]	2	✓	✗	✗	✓	✗
	Ponte [14]	3	✓	✗	✗	✓	✗
	Macêdo et al. [15]	4	✓	✗	✓	✓	✗
	Desai et al. [71]	3	✓	✗	✓	✓	✗
	Al-Fuqaha et al. [16]	3	✓	✗	✓	✓	✗
CC	symbIoTe [72]	3	✗	✓	✓	✓	✗
	BIG IoT [73]	1	✗	✓	✓	✓	✗
MDE	Vorto [74]	0	✗	✗	✓	✓	✓
	Ciccozzi et al. [75]	0	✗	✗	✓	✓	✓
	DeXMS	5	✓	✓	✓	✓	✓

Table 12: Comparison of DeXMS with related work.

tions, such as ESB [21], INDISS [56], uMiddle [57] and SeDIM [58], to achieve N-one-M mapping, by using an intermediary protocol, between N and M systems that employ various protocols; and *(iii) common abstractions*, such as ReMMoC [59] and WSIF [60], which enable the interoperation of legacy systems by abstracting their behavior.

While inheriting from the long-investigated but still open question of distributed system interoperability, solving the heterogeneity problem between IoT applications is particularly challenging, especially due to the fast development of protocols and APIs in the IoT. Approaches stemming from distributed systems have been also applied in the IoT. In this section, we discuss the most recent efforts of the academic and industrial communities coping with Things interoperability at the middleware layer. In particular, paradigms and settings such as *Service Oriented Architecture (SOA)*, *Gateways*, *Cloud Computing (CC)* and *Model Driven Engineering (MDE)* have been used to provide middleware interoperability solutions in the IoT. We summarize the principal solutions that we survey in the next subsections in Table 12 with regard to their support for: *(i)* several middleware protocols (how many); *(ii)* direct bridging and/or *(iii)* bridging via a common protocol; *(iv)* extensibility by using software abstractions; *(v)* constrained devices; and *(vi)* automated mediator synthesis. At the bottom of Table 12, we provide in comparison the same information for DeXMS.

7.1. Service-oriented approaches

Among the software architecture paradigms envisioned for IoT-based systems, the literature suggests that service orientation is particularly promising due to its inherent support for interoperability and composability [76]. To build an IoT platform that supports SOA functionalities such as *Discovery*, *Composition* of, and *Access* to services, the starting point is to abstract Things or their data as services [77, 78, 79]. While many attempts in the literature to build service-oriented IoT platforms support Discovery and Composition, we focus in this section on efforts regarding the Access functionality.

Compared to classic *Business services*, *Thing-based services* encompass highly heterogeneous software entities, among which resource-constrained ones [80]. To deal with heterogeneous service access (i.e., interconnect service providers and consumers that employ different middleware protocols), the *Enterprise Service Bus (ESB)* paradigm [21] is the predominant solution in SOA. Hence, ESB-based solutions have further been applied in the IoT.

In [17, 62, 63], the authors introduce the *Lightweight Internet of Things Service Bus (LISA)* for tackling IoT heterogeneity. LISA facilitates the task of developers by providing an API for resource-constrained devices that supports access, discovery, registration and authentication. Devices deployed based on different standards interact via a common communication protocol. An ESB is also used in [64] as the core infrastructure for an event-driven IoT service coordination platform. It enables interconnecting heterogeneous components such as devices acting as event publishers and/or subscribers, users issuing HTTP requests, a Complex Event Processing (CEP) engine, and an Event Condition Action (ECA) rule engine.

To support local wireless sensor networks, authors in [65] provide an implementation of a *Home Service Bus* for solving interoperability problems among embedded electronic devices and resource-constrained sensors in smart home environments. Similarly, an ESB-based industrial middleware is proposed in [66], where multiple sensor gateways (enabling sensors that only communicate at the MAC layer to connect to the Internet) make part of a service oriented setting.

The last two efforts of this section that we survey next are the most relevant to our solution. The authors in [67] introduce a protocol translator that utilizes an intermediate format (not a protocol as in ESBs) to capture all protocol specific information. Translators can be placed in local clouds and be used in a transparent and on-demand way. While the authors claim that many different protocols can be mapped with their protocol translator, this work lacks sufficiently general abstractions for enabling its wide application. Finally, XWARE [68] is an event-based framework for solving interoperability across different middleware platforms by using plugins and mediators. Plugins communicate with legacy services to send/receive messages and convert them to events; mediators translate events between different plugins by relying on an intermediate format. To deal with protocols classified under different interaction paradigms, the authors rely on our previous work [20] that is also the base for DeXMS. However, they do not provide support for automated synthesis of mediators.

7.2. Gateway-based approaches

While in the present work we focus on middleware-layer interoperability, another source of heterogeneity in IoT

applications comes from the lower MAC layer protocols. In particular, resource-constrained connected devices may host a lightweight protocol stack that goes only up to the MAC layer. Then, a common approach to connect a set of sensors and actuators interacting using MAC layer protocols (e.g., Bluetooth, ZigBee, etc.) to the Internet is through *Sensor Gateways*. Furthermore, to integrate Things that employ MAC and/or middleware-layer protocols, *IoT Gateways* [81, 16] have been developed, which may also provide advanced functionalities such as sensor data aggregation and sending to a cloud.

Identifying MQTT and REST as the state-of-the-art protocols of the IoT and the Web, respectively, the authors in [69] aim to bridge the gap between the two worlds. In particular, QEST broker is a gateway that enables interoperability between these two protocols. By extending QEST, Ponte [14], which is developed as part of the Eclipse IoT open source community [82], provides APIs to application developers that enable automatic conversion between REST, CoAP and MQTT protocols. Similar gateways exist in the literature providing a cross-protocol proxy, such as in [83] for HTTP-CoAP interoperability and in [70] for DPWS-REST interoperability. Other Eclipse IoT projects proposing IoT gateways are Kura [84], NeoSCADA [85], and SmartHome [86]. However, they focus on MAC layer protocol interoperability.

Focusing on middleware-layer protocols, the work in [15] introduces a gateway that provides interoperability between REST, CoAP, MQTT and XMPP protocols. Request/response and publish/subscribe messaging patterns are supported. Furthermore, the architecture of the gateway enables addition of new protocols via plugins. By using semantic technologies, the authors in [71] take a step beyond just protocol interoperability by providing a gateway that: (i) bridges XMPP, CoAP and MQTT; and (ii) annotates exchanged messages with a sensor data description via the W3C's Semantic Sensor Network (SSN) ontology. While the latter can enable automating application-level interoperability (as an extension to our manual mapping based on configuration files), the authors do not develop this aspect any further.

Finally, the work in [16] introduces a so-called *intelligent IoT gateway*, which embeds a protocol translator that can be programmed via an XML-based rule language. While this resembles our mediator logic, the authors do not provide any detail about this language nor about the related protocol abstractions. Still, what is interesting in this work is the proposed direct mediation between resource-constrained and resource-rich devices. Although the former are assumed to host a lightweight IP (uIP or lwIP) protocol implementation, functionalities such as data transport and security are delegated to the gateway. In our future work, we envision extending our mediators to include support for resource-constrained devices with incomplete protocol stacks, as well as for advanced gateway functionalities such as sensor data aggregation.

7.3. Cloud-based approaches

Cloud Computing (CC) has been closely associated to the IoT since its origins: Huge amounts of data coming from multiple IoT ecosystems, such as environment, agriculture, transportation, etc., require mechanisms to offload, store, process, analyze and retrieve them; CC provides such resources remotely, reliably and at a low cost. Later, the need was identified to bring resources closer to the producers and consumers of data with solutions at the edge of the network and fog computing. Since cloud and fog computing make integral part of IoT platforms, they also make part of the high fragmentation of the IoT landscape with numerous vertical, closed solutions. We survey next two representative efforts towards IoT interoperability involving cloud architectures.

The solution developed by the H2020 *symbIoTe* [72] European project provides an interoperability framework across vertical IoT platforms for enabling cross-platform application development. Each such platform involves a hierarchical IoT stack connecting numerous Things into smart spaces through IoT gateways, which further connect these smart spaces to a cloud. The symbIoTe high-level architecture foresees horizontal interoperability mechanisms at multiple levels of the hierarchical IoT platforms, including the Thing domain level, the smart space domain level (across gateways), and the cloud domain level (across clouds). Moreover, the top – application domain – level supports common, cross-platform APIs for application development. symbIoTe's envisioned multi-level mechanisms address interoperability for both IoT communication protocols and semantic representation of sensor data. This shows the potential of our DeXMS mediator solution, which can be applied at many different levels of an IoT ecosystem.

Having similar objectives to symbIoTe, the H2020 *BIG IoT* [73] European project introduces an approach for enabling IoT ecosystems by establishing interoperability across IoT platforms that may also belong to different vertical markets or application domains. BIG IoT focuses on interoperability among IoT data (their syntax and semantics) hosted by the different IoT platforms. For this, it requires each platform to be enriched with a common API for interoperability, the *BIG IoT API*, besides its own interfaces. It is particularly interesting that a platform may operate at cloud level, fog level (e.g., a gateway) or even device level (e.g., a Raspberry Pi or smartphone): the BIG IoT API can be used independently of a platform's scale. While BIG IoT does not foresee any mediation between protocols, complementing its data-oriented interoperability approach with a solution like DeXMS would enable relaxing its pretty restrictive requirement for a universal API and associated protocol to be implemented by all IoT platforms.

7.4. Model driven approaches

In the Model-Driven Engineering (MDE) paradigm, models are considered as first-class artefacts that can be used throughout the software development process, enabling the creation and/or automatic execution of software systems starting from these models [87]. Model-driven development approaches define modeling languages for specifying a system at different levels of abstraction.

They further provide *(i) model-to-model* transformations that translate models into another set of models, typically closer to the final system, and *(ii) model-to-text* transformations that generate software artefacts, e.g., source code or XML code, from models.

Vorto [74] is an Eclipse IoT project that aims to establish standardized abstractions of IoT devices. To this end, it specifies a metamodel using the Eclipse Modeling Framework (EMF) [42]. Using this metamodel, a developer is able to build the *information model* of a device, which describes its capabilities and exposes its properties, operations and events. This information model is then stored in a global repository. Other developers wishing to integrate the specific device in their applications can access the information model. They can use specific code generators for creating implementation skeletons of the device for their specific IoT technology environments. Vorto's abstractions and related code generation address IoT device capabilities but not their communication protocols. Nevertheless, this approach has some similarities with our DeXIDL and Protocol Pool abstractions and their use in the automated synthesis of mediators.

The work in [75] presents a comprehensive discussion of the specificities and challenges of mission-critical (but also more general) IoT systems and the solutions that model-driven engineering can provide to the development and runtime management of such systems. Among the identified challenges, we point out: heterogeneity and associated complexity, lack of reusability leading to multiple similar but incompatible solutions, and runtime context uncertainty resulting in emergent system properties. As discussed by the authors, MDE can help tackling this issues by relying on: *(i)* high-level abstractions enabling platform and technology neutrality; *(ii)* models and methods for systematic, automated system development, and hence sustainable regarding time, cost, and effort; and *(iii)* runtime models for system self-adaptation to dynamic environments. DeXMS applies the MDE paradigm to deal with the heterogeneity and complexity of IoT middleware protocols, and to provide a reusable process as well as artefacts for automated building of protocol mediators. Furthermore, it is in our future goals to support runtime synthesis, deployment and adaptation of mediators.

8. Conclusion

Integrating Things that employ heterogeneous middleware protocols is challenging due to the differences of protocols in semantics and implementation exacerbated by the high technology diversity of the IoT solutions landscape. In this paper, we introduced a systematic solution to the IoT interoperability problem at the middleware layer. Our approach relies on identifying common abstract interaction types across the core interaction paradigms (Client/Server, Publish/Subscribe, Data Streaming and Tuple Space) encountered in the IoT. These paradigms represent the vast majority of the existing and possibly future IoT middleware protocols. We model the abstract interaction types into the DeX API & connector model, which is thus able to abstract in a unifying way the multitude of heterogeneous IoT protocols. We further elicit the DeXIDL language, which can be used to describe the application interfaces of Things in a common abstract way independently of the underlying middleware protocols. Based on DeX and DeXIDL, we introduce an architecture for mediators that can bridge heterogeneous Things and their protocols in a direct or indirect (via an intermediate common protocol) way. The outcome of our overall effort is the DeXMS development & runtime framework, which supports the automated synthesis, deployment and execution of mediators. Use of DeXMS results in 94-98% manual code saving for the IoT application developer, which further does not include the debugging and testing effort involved in the manual development of mediators. Furthermore, DeXMS is flexible, enabling selection of any common protocol for indirect mediation and easy integration of support for new IoT protocols. Finally, we have favorably tested DeXMS mediators for both performance and resource consumption under high serving loads and on a resource-constrained hosting node.

Acknowledgment

This work has been partially supported by the European Union's Horizon 2020 project CHOReVOLUTION under grant agreement No. 644178.

Appendix A. DeX-IDL metamodel

We provide in this appendix the DeXIDL metamodel as we created it inside the Eclipse Modeling Framework (EMF). The resulting UML class diagram is depicted in Fig. A.27. We further provide the complete DeXIDL model for vehicle-devices that we specified using our Eclipse plugin. It is shown in Fig. A.28.

References

[1] Z. Shelby, et al., The constrained application protocol (CoAP), Tech. rep. (2014).

[2] A. Banks, R. Gupta, MQTT Version 3.1.1, OASIS standard (2014).

[3] I. Fette, A. Melnikov, The websocket protocol, Tech. rep. (2011).

[4] SemiSpace. Light weight Open Source interpretation of Tuple Space / Object Space, `http://www.semispace.org/`.

[5] D. Schrank, et al., TTI's 2012 urban mobility report, Texas A&M Transportation Institute. The Texas A&M University System (2012).

[6] J. Yoon, et al., Surface street traffic estimation, in: ACM Mobisys, PR, USA, June 2007.

[7] Waze, `https://www.waze.com/en`.

[8] P. Mohan, et al., Nericell: rich monitoring of road and traffic conditions using mobile smartphones, in: ACM SenSys, Raleigh, NC, USA, November 2008.

[9] K. Fysarakis, I. Askoxylakis, O. Soultatos, I. Papaefstathiou, C. Manifavas, V. Katos, Which IoT Protocol? Comparing Standardized Approaches over a Common M2M Application, in: IEEE Global Communications Conference (GLOBECOM), 2016, pp. 1–7. `doi:10.1109/GLOCOM.2016.7842383`.

[10] G. Bajaj, G. Bouloukakis, A. Pathak, P. Singh, N. Georgantas, V. Issarny, Toward Enabling Convenient Urban Transit through Mobile Crowdsensing, in: IEEE ITSC, Las Palmas, Gran Canaria, 2015.

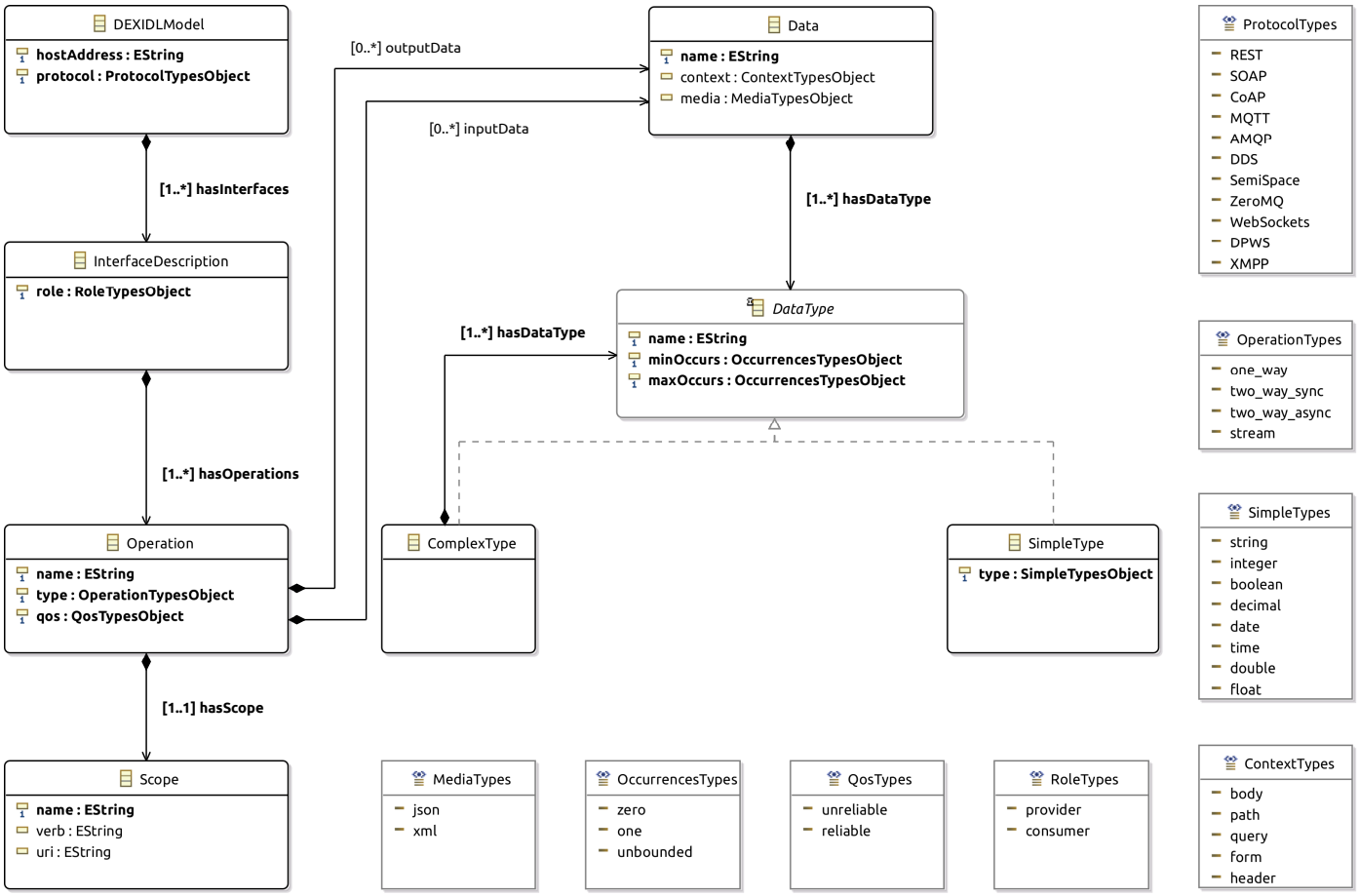

Figure A.27: The DeX-IDL metamodel.

[11] T. F. Roy, The REpresentational State Transfer (REST), Department of Information and Computer Science, UCI (2000).

[12] V. Karagiannis, et al., A survey on application layer protocols for the internet of things, Transaction on IoT and Cloud Computing (2015).

[13] A. Talaminos-Barroso, et al., A Machine-to-Machine protocol benchmark for eHealth applications–Use case: Respiratory rehabilitation, Computer methods and programs in biomedicine (2016).

[14] Ponte, `http://www.eclipse.org/proposals/technology.ponte/`.

[15] W. Macêdo, et al., GoThings-An Application-layer Gateway Architecture for the Internet of Things, in: WEBIST, Lisbon, Portugal, May 2015.

[16] A. Al-Fuqaha, et al., Toward better horizontal integration among iot services, IEEE Communications Magazine (2015).

[17] B. Negash, et al., LISA: lightweight Internet of Things service bus architecture, Procedia Computer Science (2015).

[18] L. Alboaie, et al., Swarm Communication-A Messaging Pattern Proposal for Dynamic Scalability in Cloud, in: IEEE HPCC EUC, Zhangjiajie, China, November 2013.

[19] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, G. Roussel, D-LITe: Building Internet of Things Choreographies, arXiv (2016).

[20] N. Georgantas, et al., Service-oriented distributed applications in the future internet: The case for interaction paradigm interoperability, in: ESOCC, Managa, Spain, September 2013.

[21] D. Chappell, Enterprise service bus, O'Reilly Media, Inc., 2004.

[22] G. Bouloukakis, et al., Integration of heterogeneous services and things into choreographies, in: ICSOC, Banff, Canada, 2016.

[23] V. Issarny, et al., Revisiting service-oriented architecture for the iot: A middleware perspective, in: ICSOC, Banff, Canada, 2016.

[24] A. Bennaceur, V. Issarny, Automated synthesis of mediators to support component interoperability, IEEE Transactions on Software Engineering 41 (3) (2015) 221–240.

[25] E. Zeeb, et al., Service-oriented architectures for embedded systems using devices profile for web services, in: AINA Workshops, Niagara Falls, Canada, May 2007.

[26] W. Mahnke, et al., OPC unified architecture, OPC Unified Architecture. Springer-Verlag Berlin Heidelberg, 2009.

[27] P. Saint-Andre, Extensible messaging and presence protocol (XMPP), 2011.

[28] Sun Microsystems. JMS Specifications and Reference Implementation, `http://www.oracle.com/technetwork/java/jms/index.html`.

[29] DDS. Data Distribution Service, `http://portals.omg.org/dds/`.

[30] O. Standard, Oasis advanced message queuing protocol (amqp) version 1.0., 2012.

[31] R. Kyusakov, J. Eliasson, J. Delsing, J. van Deventer, J. Gustafsson, Integration of wireless sensor and actuator nodes with it infrastructure using service-oriented architecture, IEEE Transactions on industrial informatics (2013).

[32] Javaspaces. beyond conventional distributed programming paradigms, `http://www.oracle.com/technetwork/articles/java/javaspaces-140665.html`.

[33] P. Eugster, et al., The many faces of publish/subscribe, ACM Computing Surveys (2003).

[34] L. Aldred, et al., On the notion of coupling in communication middleware, in: OTM, Springer, Agia Napa, Cyprus, October 2005.

[35] G. Bouloukakis, et al., Performance Modeling of the Middleware Overlay Infrastructure of Mobile Things, in: IEEE ICC, 2017.

[36] Pivotal, "RabbitMQ", `https://www.rabbitmq.com/`.

[37] Apache Kafka, `http://kafka.apache.org/`.

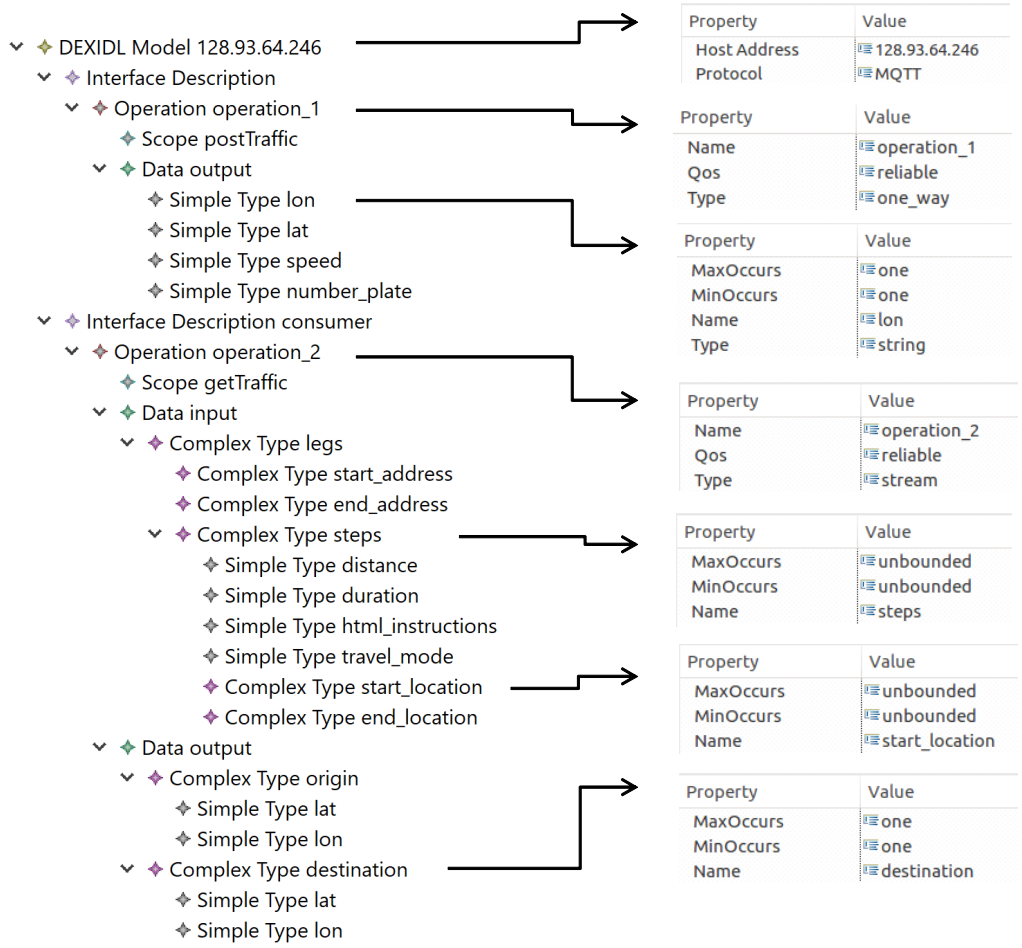

Figure A.28: Designing the DeXIDL model for vehicles devices using our Eclipse plugin.

[38] B. Billet, V. Issarny, dioptase: data streaming middleware for the internet of things, ERCIM News (2015).

[39] GigaSpaces. Empowering next Generation Web Scale Applications, http://www.gigaspaces.com/.

[40] Lime., http://lime.sourceforge.net/Lime/index.html.

[41] Oma. fi-ware ngsi context management specifications, http://www.openmobilealliance.org/wp.

[42] Eclipse modeling framework, https://eclipse.org/modeling/emf.

[43] P. Ntumba, G. Bouloukakis, N. Georgantas, Interconnecting and monitoring heterogeneous things in iot applications, in: International Conference on Web Engineering, Springer, Cham, 2018, pp. 477–481.

[44] H2020 chorevolution project, www.chorevolution.eu.

[45] R. Sugihara, R. K. Gupta, Programming models for sensor networks: A survey, ACM Transactions on Sensor Networks (TOSN) 4 (2) (2008) 8.

[46] K. Ueno, et al., Early capacity testing of an enterprise service bus, in: IEEE ICWS, Chicago, USA, September 2006.

[47] S. Desmet, B. Volckaert, S. Van Assche, D. Van Der Weken, B. Dhoedt, F. De Turck, Throughput evaluation of different enterprise service bus approaches, in: Proceedings of the 2007 International Conference on Software Engineering Research & Practice, 2007, pp. 378–384.

[48] K. Hartke, Observing Resources in the Constrained Application Protocol (CoAP), RFC 7641 (Sep. 2015). doi:10.17487/RFC7641.
URL https://rfc-editor.org/rfc/rfc7641.txt

[49] Z. B. Babovic, J. Protic, V. Milutinovic, Web performance evaluation for internet of things applications, IEEE Access 4 (2016) 6974–6992. doi:10.1109/ACCESS.2016.2615181.

[50] H. Kilov, P. F. Linington, J. R. Romero, A. Tanaka, A. Vallecillo, The reference model of open distributed processing: Foundations, experience and applications, Computer Standards Interfaces 35 (3) (2013) 247 – 256. doi:https://doi.org/10.1016/j.csi.2012.05.003.
URL http://www.sciencedirect.com/science/article/pii/S0920548912000621

[51] Object Management Group, The common object request broker: architecture and specification, Tech. rep., version 2.0 (1995).

[52] Microsoft Corporation, DCOM (2000), https://docs.microsoft.com/el-gr/openspecs/windows_protocols/ms-dcom.

[53] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web services description language (wsdl) 1.1 w3c note 15 march 2001, World Wide Web Consortium, Mar.

[54] Object Management Group, COMCORBA Interworking Spec. Part A and B, Tech. rep. (1997).

[55] SOAP2CORBA, http://soap2corba.sourceforge.net/.

[56] Y.-D. Bromberg, V. Issarny, Indiss: Interoperable discovery system for networked services, in: Proceedings of the ACM/IFIP/USENIX 2005 international Conference on Middleware, 2005, pp. 164–183.

[57] J. Nakazawa, H. Tokuda, W. K. Edwards, U. Ramachandran, A bridging framework for universal interoperability in pervasive systems, in: 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), IEEE, 2006, pp. 3–3.

[58] C. Flores, P. Grace, G. S. Blair, Sedim: A middleware framework for interoperable service discovery in heterogeneous networks, ACM Transactions on Autonomous and Adaptive Systems

(TAAS) 6 (1) (2011) 6.
[59] P. Grace, G. S. Blair, S. Samuel, A reflective framework for discovery and interaction in heterogeneous mobile environments, ACM SIGMOBILE Mobile Computing and Communications Review 9 (1) (2005) 2–14.
[60] M. J. Duftler, N. K. Mukhi, A. Slominski, S. Weerawarana, Web services invocation framework (wsif), in: OOPSLA Workshop on Object Oriented Web Services, Vol. 194, 2001, p. 49.
[61] L. Alboaie, S. Alboaie, T. Barbu, Extending swarm communication to unify choreography and long-lived processes, in: The 23rd International Conference on Information Systems Development, Varazdin, Croatia, 2014.
[62] B. Negash, A. Rahmani, T. Westerlund, P. Liljeberg, H. Tenhunen, Lisa 2.0: lightweight internet of things service bus architecture using node centric networking, Journal of Ambient Intelligence and Humanized Computing, Springer (2016).
[63] B. Negash, A. Rahmani, T. Westerlund, P. Liljeberg, H. Tenhunen, Enabling Layered Interoperability for Internet of Things Through LISA, in: Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 2014.
[64] B. Cheng, D. Zhu, S. Zhao, J. Chen, Situation-aware IoT service coordination using the event-driven SOA paradigm, IEEE Transactions on Network and Service Management (2016).
[65] Y. Tao, X. Xu, X. Wang, Service-Based Interactive Proxy for Sensor Networks in Smart Home: An Implementation of Home Service Bus, in: IEEE ICDH, Guangzhou , China, November 2014.
[66] A. Ismail, W. Kastner, A middleware architecture for vertical integration, in: 1st International Workshop on Cyber-Physical Production Systems (CPPS), IEEE, Vienna, Austria, April 2016, pp. 1–4.
[67] H. Derhamy, J. Eliasson, J. Delsing, Iot interoperabilityon-demand and low latency transparent multiprotocol translator, IEEE Internet of Things Journal (2017) 1754–1763.
[68] F. M. Roth, C. Becker, G. Vega, P. Lalanda, Xwarea customizable interoperability framework for pervasive computing systems, Pervasive and Mobile Computing (2018) 13–30.
[69] M. Collina, G. E. Corazza, A. Vanelli-Coralli, Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST, in: IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC), Sydney, Australia, September 2012, pp. 36–41.
[70] S. Han, S. Park, G. Lee, N. Crespi, Extending the devices profile for web services standard using a rest proxy, IEEE Internet Computing (2015).
[71] P. Desai, A. Sheth, P. Anantharam, Semantic gateway as a service architecture for iot interoperability, in: IEEE MS, New York, USA, June 2015.
[72] S. Soursos, I. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, G. Carrozzo, Towards the cross-domain interoperability of iot platforms, in: IEEE EuCNC, Athens, Greece, June 2016.
[73] A. Bröring, et al., Enabling iot ecosystems through platform interoperability, IEEE Software, forthcoming (2017).
[74] Vorto, `https://projects.eclipse.org/proposals/vorto`.
[75] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, R. Spalazzese, Model-driven engineering for mission-critical iot systems, IEEE Software (2017).
[76] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio, Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services, IEEE transactions on Services Computing (2010).
[77] C. Fok, G. Roman, C. Lu, Servilla: a flexible service provisioning middleware for heterogeneous sensor networks, Science of Computer Programming (2012).
[78] I. Corredor, J. F. Martínez, M. Familiar, L. López, Knowledge-aware and service-oriented middleware for deploying pervasive services, Journal of Network and Computer Applications (2012).
[79] C. Perera, P. Jayaraman, A. Zaslavsky, D. Georgakopoulos, P. Christen, Mosden: An internet of things middleware for resource constrained mobile devices, in: IEEE HICSS, Waikoloa, HI, USA, January 2014.
[80] D. Athanasopoulos, M. Autili, N. Georgantas, V. Issarny, M. Tivoli, A. Zarras, An architectural style for the development of choreographies in the future internet, Global Journal of Advanced Software Engineering 1 (1) (2014) 14–28.
[81] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: A survey on enabling technologies, protocols, and applications, IEEE Communications Surveys & Tutorials (2015).
[82] Eclipse iot - open source for iot, `https://iot.eclipse.org/`.
[83] A. Castellani, T. Fossati, S. Loreto, Http-coap cross protocol proxy: an implementation viewpoint, in: IEEE MASS, Las Vegas, Nevada, USA, October 2012.
[84] Kura - osgi-based application framework for m2m service gateways, `http://www.eclipse.org/proposals/technology.kura`.
[85] Eclipse neoscada, `http://projects.eclipse.org/projects/technology.eclipsescada`.
[86] Eclipse smarthome, `https://eclipse.org/proposals/technology.smarthome/`.
[87] A. R. da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems and Structures 43 (2015) 139 – 155. `doi:https://doi.org/10.1016/j.cl.2015.06.001`.

Georgios Bouloukakis is a postdoctoral researcher at the University of California, Irvine in the Distributed Systems Middleware group. His research mainly focuses on the design of extensible and efficient IoT systems by leveraging fundamental mathematical models and state-of-the-art technologies. Before joining UC Irvine, Georgios received a postdoctoral scholarship from the Inria@SiliconValley research program. He obtained his Ph.D. from UPMC/Sorbonne, conducting his thesis at the research center of Inria Paris in the MiMove team in France.

Nikolaos Georgantas is a researcher at Inria Paris and Head of the MiMove research team on mobile distributed systems and supporting middleware. Nikolaos received a Ph.D. in electrical and computer engineering from the National Technical University of Athens (NTUA), Greece and a habilitation degree in computer science from UPMC/Sorbonne University, France. His research interests relate to mobile computing, service-oriented computing, and self-adaptive systems. Nikolaos is currently working on interoperability and QoS analysis in the IoT as well as on dynamic resource management at the network edge for IoT data streaming. He regularly serves as member in several international conference program committees. He has been involved in a large number of European research projects and undertaken work package leader and PI roles.

Patient Ntumba is a Ph.D. student at Inria (Paris research center) in the MiMove team since August 2018. Before starting his Ph.D., he was a Development Engineer in the same institute, working in the H2020 CHOReVOLUTION European Project. His cur-

rent research focuses on middleware, self-adaptive systems, mobile distributed systems and software engineering. He holds a M.Sc. degree in Distributed Systems and Networks from the University of Franche Comté and a B.Sc. degree in Software Engineering from the University of Kinshasa.

Valérie Issarny holds a "Director of research" position at Inria, in the research center of Inria Paris. Her research lies in the study of middleware solutions easing the development of distributed collaborative services, including mobile services deployed over smartphones and interacting with sensors and actuators. From 2002 to 2013, Valérie led the ARLES research team in which they investigated distributed software systems leveraging wirelessly networked devices, with a special emphasis on service-oriented systems. Valérie regularly serves in the program committees of major conferences of the software engineering and middleware domains. She is currently associate editor of ACM TAAS, ACM TIOT, IEEE TSE and IEEEE TSC.