Logic Reasoning Based IoT Middleware - Extended Version

Abstract

Many IoT applications, although often called smart, lack any form of intelligence. At best, they integrate basic automation. Integrating automatic device configuration, connection management, and user support for solving connectivity issues, is a real endeavor for application developers. Likewise, integrating and enforcing policies in an IoT application is complex. Moreover, the dynamic nature of IoT systems makes it even more difficult to develop applications that properly handle changes in the environment.

This paper presents a platform-independent middleware that simplifies the development of smart applications. The middleware hosts a modular, event-based logic reasoner, developed in Prolog, communicating with the underlying IoT framework. It not only supports advanced automation, but also holds functionality for automatic device and connection management, access control and, an abstraction module that decouples the applications from the underlying infrastructure. Moreover, the middleware leverages the actual benefits of Prolog through complex querying and inference capabilities. As a demonstrator, the middleware and modules are integrated in both a server and mobile application.

1 Introduction

Several frameworks have been developed [1, 2] that assist developers in building Internet of Things (IoT) applications. However, these frameworks mainly focus on the discovery and communication with IoT devices. But even the most basic IoT application must cope with several issues typically encountered in such environments. Examples are devices coming in range or a loss of device connectivity, enforcing policies such as latency requirements or energy usage concerns, and handling the replacement of devices by similar devices. The diversity of platforms on which IoT applications run isn't making things easier, going from edge devices, gateways and mobile devices to servers and cloud platforms. Building applications that are able to combine these platforms is even more complex. But maybe even more important, although many of these applications are called smart, traces of intelligence are often hard to find.

All these issues make companies struggle with building reusable and sustainable smart IoT applications. Porting an application to a similar but different setting often results in a lot of labour and is expensive: devices may have different properties requiring application logic to be rewritten; information is derived based on different technologies (e.g., instead of GPS to determine a user's presence, infrared sensors are used).

This paper aims at decreasing the burden on IoT application developers, by providing a platform-independent middleware that addresses some of these typical concerns. It takes care of *contextual changes* as a result of devices getting in or out of range, and supports application developers to build applications based on more intuitive concepts called *assets*. Assets were introduced in [3] as a virtualization of the real environment. For instance, a *kitchen* may be an asset with a certain room temperature, which can simply indicate whether it is cold or warm, instead of working with a `temperature_sensor_1` with a value of 25°C. Besides decoupling the application logic from the device layer, and, hence, minimize the required changes when a device is replaced, it makes defining the applications more intuitive. Moreover, the middleware can be installed on a gateway that only exposes the assets. An application connecting to the gateway will retrieve assets and its properties instead of devices.

Another concern the middleware addresses, is a solution to properly manage and enforce policies such as *access control and Quality-of-Service (QoS) policies*. In multi-user environments, IoT systems often require strict access control measures. For instance, a gateway may control that the devices are accessed only during office hours, when the user is on premise, or when certain sensors have a specific state. Likewise, to control energy consumption, QoS policies may define constraints on data processing in case the battery of a device is below a certain threshold.

Finally, to cope with the growing need for more advanced intelligence in IoT applications, the middleware includes a *logic reasoner* (i.e., Prolog [4]) as a first step towards building smart applications. It allows applications to automatically infer knowledge and provides users with querying capabilities to gain insights in the IoT system. Computing a query to request all machines in alarm follows naturally from the logic reasoner, without any additional coding. Moreover, the inference allows us to retrieve a common potential root cause. For instance, they are all in the same manufacturing hall and that hall has a power outage.

Contributions. This paper proposes an **IoT middleware** that supports application developers in building and maintaining IoT applications. The middleware can be used in many locations of the IoT ecosystem such as client applications, gateway applications, and even inside IoT devices requiring some form of intelligence. The middleware builds on a flexible event-based architecture running a logic reasoner in the background. It hosts a number of IoT modules specially devised to handle the issues discussed above, such as the handling of contextual changes, managing and enforcing access control and QoS policies, and supporting a full featured automation engine. The logic reasoner is also the basis to further add more advanced intelligence to the IoT applications.

To demonstrate its feasibility, the middleware has been integrated in a server and mobile application, a basic integration of the IoT modules is provided. Although not a requirement, for this demonstrator, the middleware was built on top of a redesign of an existing framework [3] providing generic access to IoT devices. Both have been developed in JavaScript making it portable to different environments such as iOS, Android, and server applications.

The paper is structured as follows. Section 2 points to related work. Section 3 explains the proposed middleware itself, together with the main building blocks used. More details about the implementation can be found in Section 4. Section 5, discusses the proposed work. The paper ends with conclusions.

2 Related Work

To aid application developers, a number of frameworks have been developed that simplify the integration of devices, vendor and protocol independent. Many of them are cloud based [1, 2], commercial examples are Siemens MindSphere[5], GE Predix[6], Zetta[7], ThingSpeak[8] and DeviceHive[9]. Previous mentioned commercial examples have their own servers running, which can be used by their users. Others run on a dedicated personal server, for instance Home Assistant[10], OpenHAB[11], Macchina.io[12] and Blynk[13]. The latter may provide cloud based integrations allowing communication with other applications. These frameworks mainly focus on integrating IoT devices, vendor and protocol independent. Some only support monitoring and visualising data (e.g., ThingSpeak), others focus on a specific domain (e.g., OpenHAB and Home Assistant in the domain of smart homes). An open framework is proposed in [14]. By using microservices, their solution is made scalable, extendable and maintainable. Functionalities can be added such as automation, artificial intelligence and big data.

Although many frameworks make the applications smart by adding automation functionality, almost none of them take care of the dynamic nature of IoT ecosystems, such as the changing connection states and replacement of devices. Rules are static and updating them is often difficult. Handling multiple types of connections for the same devices is even worse. Consider, for example, connecting via a cloud server when a direct connection with the device is lost . Moreover, while IoT systems offer means to maximize reliability, control latency and hence, increase QoS [15], this functionality is usually left unused. Our middleware solution makes it possible to manage rules dynamically and support applications in coping with the dynamic nature of IoT systems. The middleware is designed with device availability, reliability and QoS in mind. It allows to specify QoS policies based on for instance connectivity and reliability, and enforces them dynamically.

Automation in IoT systems is widely available. Reasoning in these systems, however, is often static, limited to very basic event-condition-action rule handling and tightly coupled to the framework (e.g., Home Assistant). To provide more advanced reasoning capabilities, recent work [16, 17, 18, 19, 20] suggests the use of more complex symbolic reasoners. [16] propose LPAAS, a logic programming REST-based service for IoT Systems. For example, a hybrid reasoning based on compositional rules selecting the reasoner based on the context is proposed in [17]. A semantic model and an IoT middleware service for data stream reasoning is presented in [18]. The stream of events is enriched with semantic meaning. Bonte et. al. tackles the problem that occurs when incoming data frequency is higher than the reasoning time, especially in time-critical systems [19]. They present an ontology-based approximation technique that allows to extract a subset of data to speed-up the reasoning process, resulting in faster decision making. A comparison of IoT system architectures focusing on semantic ontology is discussed in detail in [20]. A framework for real-time and dynamic ontology modeling is proposed using the Pellet reasoner.

Most of the reasoners above start from the use of ontologies to provide higher level reasoning. Although our middleware may be extended to take advantage of these ontologies, for many applications it would be too extensive and computationally heavy to reason on an entire ontology. The logic reasoner is not only involved in the extraction of knowledge. It is also the backbone of our middleware, controlling event flow, dynamic processing and reasoning. In other words, while most solutions are imperative programs that interact with a reasoner for symbolic reasoning, our middleware is mainly a logic program, that interfaces with standard (imperative) libraries when appropriate.

[21] proposes an ontological reasoning approach that focuses on the cloud application orchestration. Interactions among cloud applications are detected and resolved. Our solution is a middleware that is platform independent, and while our primary focus is on IoT devices, gateways or mobile applications, integrating it in

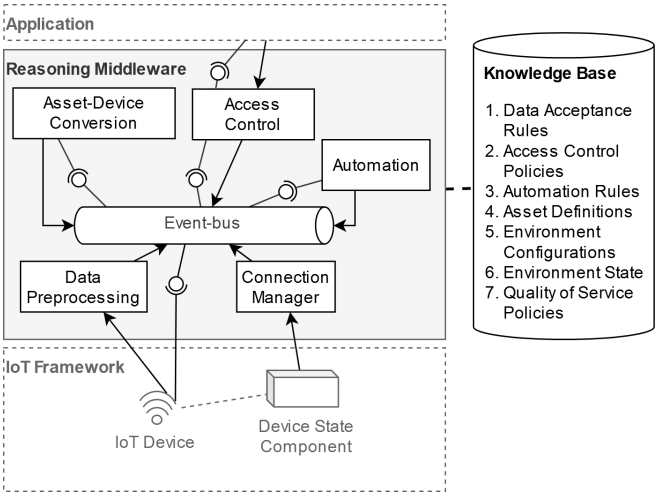

Figure 1: Structure of the middleware for a typical IoT application

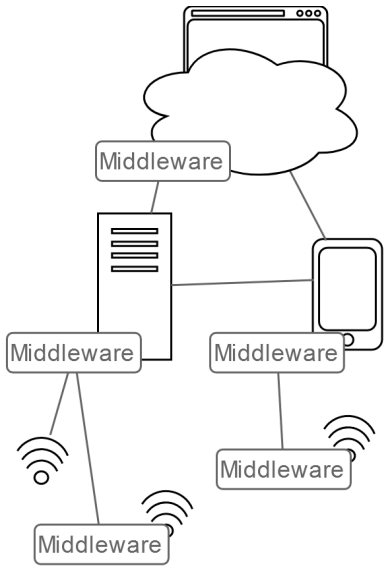

Figure 2: An IoT middleware for device, mobile and cloud platforms

a cloud context is straightforward.

More importantly, taking advantage of the reasoning engine, our middleware provides a number of prepared building blocks, called IoT modules, that greatly simplify IoT application development: an advanced automation module, a connection manager, a module for access control and a module to manage assets.

3 Middleware

The middleware consists of an *event-based logic reasoning engine*, hosting multiple IoT modules. Figure 1 presents the architecture of the reasoning engine for a typical IoT application. Adhering to the principle of separation of concerns, each *module* implements a particular functionality. Immutable *event* messages are used as a messaging mechanism between modules and for communication with the external environment. Hence, each module subscribes for specific incoming events, processes them and may create new events as a result of its processing. The core of the middleware, such as the routing of messages between modules is developed in Prolog and the IoT modules are implemented as a actual Prolog modules with some basic predicates as an interface. Modules have access to a knowledge base containing general information such as the current configuration and state for the middleware, and module specific rules and policies.

The configuration of the middleware defines the modules to be loaded and the way they are connected with each other. To communicate with the application or the underlying IoT Framework, a device and an app module are developed. By default, new events are submitted to the event-bus, and IoT modules subscribe for new events on that bus. The event-bus allows multiple modules to listen for the same events. For instance, both the automation module and asset-device conversion module may be interested in update events containing data from an IoT device. While subscribing, each module passes a filter to the bus specifying the events they handle.

To provide more control and flexibility, the application designer may choose to configure a more hybrid architecture, combing the event-based with a flow-based approach. For instance, to limit the events on the event-bus (i.e., access control module) or to split related functionality into reusable sub-modules. Through this mechanism, IoT modules are easily added, replaced or adapted by modifying the configuration of the engine.

As shown in Figure 2, the middleware distinguishes itself from other IoT frameworks in that it can be used on a multitude of platforms, going from IoT devices, mobile platforms up to even cloud platforms. Although each will have its own requirements, the framework provides features that are useful for all of them. But more importantly, communication between them is handled by the framework in a unified way.

The main building blocks (i.e., events and IoT modules), will now be presented in more detail.

3.1 Events

Events denote changes in the system. In the event-driven architecture, event messages (henceforth called *events*) are used to notify these changes. In the middleware, subscriptions are set up to register for event streams. For instance, a module may subscribe to retrieve events asynchronously from a specific IoT device, such as measurements or changes in the connectivity of the device. Likewise, an application on top of the middleware creates a subscription to the middleware to receive changes related to a certain asset or device managed by the mid-

dleware. Note that the events associated with an asset (e.g., clarity change in a room) are most-likely triggered based on events from its underlying devices (e.g., value change in a light sensor). To control the application, next to these change based events, the application also uses events to trigger certain functionality in the framework (i.e., action and query events).

Events are defined by their properties and consist at least of the following:

- the `type`, specifies the type of event,
- the `id`, uniquely identifies the event,
- the `creation-time`, specifies the time the event was created,
- the `creator` of the event. This can, for instance, be the device that generated the event or the user that sent the request. Other application components or the middleware itself can also be the creator of an event.

Optionally, the `origin-event` property specifies the id of the event that triggered it. This may refer to the *id* of the action request in case of an action response event, or to the id of the subscription request in case of an update event. Similarly, the `subject` property describes the entity to which the event information is related. For instance, when new sensor data is received from a sensor, the subject identifies the sensor (e.g., `temp_sensor_ID1`). In the case of an action-event, the subject indicates the entity on which the action must be executed.

Resources or modules are free to augment the event with additional properties. Below, a number of events are discussed in more detail. These are general messages that are used by the modules presented in this paper. These events are **update-events**, **action-events** and **query-events**.

For update-events a subscription is made to obtain a notification whenever there is a change in its content.

Update-events. As IoT-environments are highly dynamic, it is important to inform the application of any change in the environment. Not only the monitoring of sensor data is of interest, also the connectivity and reachability status of a device is useful information, as well as configuration-changes in the system. The type of information being updated is specified in the 'update-property' field. This simplifies filtering on these particular type of update events. *Parameter, connectivity and reachability* related updates are discussed below:

parameter: Upon a change of a sensor value, a parameter-update-event is generated. A single specific sensor input or actuator output is called a parameter. An IoT device may have a single parameter, for instance, the `temperature` in the case of a temperature sensor, or multiple parameters in the case of more complex IoT devices. The event is triggered when an IoT device presents a new value for one or more of its parameters. For instance, after a subscription to sensor `temp_sensor_ID1`, the sensor submits a temperature value of 24.8°C and humidity value of 45.4%. A similar event is created by the asset-device conversion module discussed in Section 3.2.4.

```
{
  type: 'update',
  id: '2cd3ae3f-100d-4cbd-...',
  creation-time: 2020/01/01 16:40:00,
  origin-event: 'f198ae08-be4d-4...',
  creator: 'temp_sensor_ID1',
  subject: 'temp_sensor_ID1',
  update-property: 'parameter',
  data:
  [
   {
    parameter: 'temp_1',
    value: 24.8
   },
   {
    parameter: 'hum_1',
    value: 45.4
   }
  ]
}
```

connectivity: In many IoT environments, the connectivity of IoT devices changes frequently. Connectivity typed update-events allow the application to properly address connectivity changes. The IoT framework is responsible for monitoring the IoT devices and submits connectivity-update events to the middleware. A device can either be connected or disconnected from the application. When the `temp_sensor_ID1` is connected, the middleware will receive the following event:

```
{
  type: 'update',
  event-id: 'e444269c-76df-4d5f-...',
  creation-time: 2020/01/01 16:40:00,
  creator: 'framework',
  subject: 'temp_sensor_ID1',
  origin-event: 'f198ae08-be4d-4...',
  update-property: 'connectivity',
  connectivity-state: 'connected'
}
```

reachability: Similar to the connectivity-update events, the device reachability state may change often. These events are triggered when devices come in-range or go out-of-range. Based on these events decisions are made to manage device connections. As an example, when the `temp_sensor_ID1` comes in range, the middleware will receive the following event:

```
{
  type: 'update',
```

```
    ...
    creator: 'framework',
    subject: 'temp_sensor_ID1',
    update-property: 'reachability',
    reachability-state: 'in-range'
  }
```

Action events. Action events are events created to trigger specific functionality. They can be created by the application as well as modules defined in the middleware. For best-practices, actions should be performed on assets instead of devices. For instance, when a user clicks a button the application creates an action event to turn the light on in the dining room, and submits it to the middleware. The asset-device conversion module (see Section 3.2.4) listens to events for the dining room asset and transforms them into new action events for the specific lamp configured for this asset. The new action event will be submitted to the IoT framework.

```
{
  type: 'action',
  event-id: '0a80f883-556a-4489-...',
  creation-time: 2020/01/01 16:40:00,
  creator: 'john',
  subject: 'dining_room',
  data:
    {
     parameter: 'table_light',
     value: 1
    }
}
```

```
{
  type: 'action',
  event-id: '5ddb9c75-c674-43bf-...',
  origin-id: '0a80f883-556a-4489...',
  creation-time: 2020/01/01 16:40:01,
  creator: 'engine',
  subject: 'lamp_ID2',
  data:
    {
     parameter: 'on-off-status',
     value: 1
    }

}
```

Query events. The middleware uses a Prolog reasoner. This reasoner takes care of the channeling of messages, filtering of messages and rule handling. A strong property of Prolog, however, is its inductive reasoning. It allows us to query the knowledge base (including the configuration, policies and the current state of the engine) and infer new conclusions, without code to be written specifically to handle the query. An example query could be to request in which manufacturing halls there are machines in alarm. In Prolog the query would be defined as follows:

```
?- asset(Hall_Asset, manufacturing_hall),
  location(Machine_Asset, Hall_Asset),
  event(Alarm, Machine_Asset, update),
  alarm_event(Alarm, Level), Level>=major.
```

Integrated into a query event we get the following:

```
{
  type: 'query',
  event-id: '5ddb9c75-c674-43bf-...',
  creation-time: 2020/01/01 16:40:01,
  creator: 'app',
  query: 'asset(Hall, manufacturing_hall),
   ↪  location(Machine, Hall),
   ↪  event(Alarm, Machine, update),
   ↪  alarm_event(Alarm, Level),
   ↪  Level>=major.',
  return:['Hall', 'Machine']
}
```

The *return* property defines which information should be returned. In this case, the *Hall* and *Machine* is requested. Since multiple solutions are possible, they are collected and returned in a single query-result event. The following query-result event is a possible response:

```
{
  type: 'query-result',
  event-id: '54a451b5-0f79-41bc-...'
  origin-event: '5ddb9c75-c674-43bf-...',
  creation-time: 2020/01/01 16:40:02,
  creator: 'middleware',
  query-result: [ ['hall_1','machine_1'],
   ↪  ['hall_1','machine_2'],
   ↪  ['hall_3','machine_7']]
}
```

3.2 IoT Modules

IoT modules are the workhorses of the middleware. A module provides a specific functionality by processing incoming events and producing new events. It either filters existing events, or creates new events as a result of it's functionality. Filtering happens, for instance, by the access control module which only forwards user requests if they are allowed according to their policies. The automation module, on the other hand, may create an action event to switch on the heating.

Events are asynchronous, and except for setting up a subscription, come in unadvertised. On the other hand, some events require a certain response to be returned to the client application. In that case, special care is required as the handling modules are now responsible to create response events. To prevent undefined behaviour, the middleware contains logic to detect cases where no response is generated (e.g., a timeout-event is raised if a query event does not result in a response event within a reasonable amount of time).

In the middleware, the core is implemented in Prolog. It provides the routing of events to the correct modules,

implements the main IoT modules and keeps a knowledge base on which reasoning can be done. The IoT modules are defined as Prolog modules with a common interface. The interface with the core is specified through a set of predefined rules that each module must provide, namely `init/0` to initialize the module and `handle/1` to allow the middleware to send events to the module for processing. For several operations such as automation and policies, a logic rule-based approach is preferable. Nevertheless, the middleware easily adapts to support script based modules. These modules are better suited for more computationally intensive operations, such as aggregation of data or analytic processing.

To help IoT application developers in building applications, the middleware provides a number of basic modules, each responsible for tackling specific challenges:

- connection manager
- access control
- data preprocessing
- asset-device conversion
- automation

Each of these modules will now be discussed.

3.2.1 Connection Manager

The connection manager is responsible for the creation and teardown of communication channels with IoT devices. QoS policies define rules on the type of channels that are set up with devices, potentially depending on other environmental factors. For instance, the interval of polling based subscriptions may be increased in case of a low battery level of the device.

When a subscription for parameter updates is requested, the connection manager takes care of the communication channel but also makes a subscription for state change events. When QoS policies state to automatically recover a connection, appropriate actions are taken as soon as the device comes in range. On the other hand, QoS policies may specify to only search for devices when certain constraints are satisfied (e.g., only try to connect to certain devices when there is a WiFi connection with a specific network). The connection manager module monitors device connectivity through the reachability and connectivity update events.

To determine the most suitable action to take, rules specify the constraints and requirements to manage connections. This module executes these rules and enforces related QoS policies. The rules and policies are stored in the knowledge base, and consulted by this module.

In addition to the more general connection control, this module also offers a functionality called *automated source selection*. Several IoT solutions provide multiple channels to collect data from a device. Sometimes, data can be collected both directly from the device, and via a cloud platform. Applications may benefit when both channels can be used. If a device is in range, the data is retrieved directly, and when no direct connection is available, the application connects to the cloud, despite an increase in latency or lower accuracy. This module allows to switch flawlessly between both, without intervention of the application developer or the user.

Since the data available via a direct or remote connection (via the cloud) may have different characteristics (e.g., aggregated, lower precision, higher latency), this module notifies the application in case of a switch.

3.2.2 Access Control

The Access Control module integrates a policy based access control mechanism (PBAC [22]). Policies define who has access to what information in the knowledge base and to which assets, and under which conditions. The module supports both positive (`allow`) and negative (`deny`) assertions. The following rule shows how this is enforced. By default, access is denied. In order to allow an event to be processed, access must be granted explicitly. Moreover, it should not be denied access explicitly. This allows us more freedom in writing access rules. Hence, an event will be forwarded only if there is at least one allow rule that succeeds, and there is no deny rule that succeeds. Of course, in production systems access control requires a more elaborate treatment to , for instance, verify the authenticity of the user. This is, however, outside the scope of this paper.

```
handle(Event) :- allow(Event),
  \+deny(Event),! -> forward(Event);
  throw(access_error('not_allowed',Event)).
```

The access control module is responsible for handling action events originating from the application. The module consults the access control policies to determine if an action may be executed. In case access is granted, it is forwarded to the event-bus. Otherwise, the module blocks the event.
An example of an access policy is the following:

> *An employee can only unlock his own office, but can do this at all times. The cleaning staff can unlock all offices, but only during certain time slots. In order to lock or unlock a room, a person must be in the neighbourhood of the room. John, however, is denied access permanently.*

In Prolog the above policy looks as follows:

```
1  allow(Event) :-
2    event_creator(Event,Creator),
3    event_subject(Event,Subject),
4    allow(Creator,Subject).
5  allow(Employee,Office) :-
6    asset(Employee,employee),
```

```
7     asset(Office,office),
8     owner(Employee,Office),
9     location(Employee,Office).
10
11  allow(CleaningStaff,Office) :-
12    asset(CleaningStaff,cleaning_staff),
13    time_between(time(10,0,0),time(15,0,0)),
14    location(CleaningStaff,Office).
15
16  deny(Event) :- event_creator(Event,john).
```

This approach allows for a fine grained access control. Access can be limited to specific parameter actions (e.g., monitor, read, write) on a device or asset, based on contextual information, or even depending the current value of other device parameters. Besides action events, also queries and subscription requests can be filtered.

3.2.3 Data Preprocessing

Parameter update events enter the system frequently. The pace at which these updates come in, is not always under control of the middleware. When the size of the stream is too large, it could lead to unnecessary or excessive processing. Depending on QoS policies (e.g., to limit processing in case of a low battery) and the requirements of the applications, events are filtered to limit the number of events exposed to the middleware. For instance, firing an update event may only be worthwhile when the value of the data changed sufficiently or when it crossed a certain value.
The following types of filters are available:

1. *pass*: all events are forwarded.
2. *value change*: the value must change.
3. *absolute difference*: the value change must be more than a fixed value.
4. *relative difference*: the value change must be more than a value relative to the range of the parameter.
5. *time difference*: the time difference between two subsequent events must be more than a fixed value.
6. *time delay*: an update is delayed for a predefined amount of time.

The latter is a special type of filter, requiring a timer to wait for a predefined time interval. If no new value arrives in the meanwhile, the value is accepted.

By default, the *value change* filter is used. Filters can be created to support more complex situations. For instance, the relative difference filter can be combined with the delay filter, to prevent short bursts of changes to be presented to the other modules or the application. In Table 1 the conditions are formalized.

Filter-type	Condition
pass	$true$
value change	$x_{new} \neq x_{old}$
absolute difference	$\lvert x_{new} - x_{old} \rvert \geq x_{\Delta}$
relative difference	$\lvert x_{new} - x_{old} \rvert \geq p * \lvert x_{max} - x_{min} \rvert$
time difference	$\lvert t_{new} - t_{old} \rvert \geq t_{\Delta}$
delay trigger	$no\, new\, value\, for\, t_{\Delta}$

Table 1: Filter definitions

3.2.4 Asset-Device Conversion

This module is in charge of the conversion of device related events into asset related events, and vice versa. By defining domain, device and state definitions and the binding between them, incoming events are easily translated. The most basic integration of an asset (e.g., living room) simply converts events coming from its underlying devices (e.g., light, temperature sensor, ...) into new events for the new resource and corresponding parameter definitions. Action events performed on the asset parameters are converted into action events on specific parameters of a device and forwarded to the device.

More advanced cases may consider to transform sensor data into a different representation, e.g., from a temperature of 40°C to a categorical value 'hot', or include specialized functionality that may be provided by the asset (e.g., to create a certain mood in the room).

It is advised for other modules to define their functionality based on these assets. This ensures that it is not necessary to update the modules that depend on the underlying devices in case a device is replaced. In fact, every device in the system can be converted into a simple asset, and higher level assets are then defined based on those simple assets. Replacing a device only requires to remap the new device with the existing asset.

This module and its implementation is discussed in more detail in Section 4.

3.2.5 Automation

As one of the most basic functionalities in IoT applications, this module supports automation. Basic event-condition-action (ECA) rules are easily defined using simple Prolog logic.

Although in general, automation rules are triggered by events coming from the underlying IoT framework (i.e., update, connectivity, reachability), this module takes advantage of the event-based architecture, and can listen for any event exposed to the module, including user actions and events created by internal modules. Logging user clicks is simply adding a rule to listen for user actions, and create a log statement.

In automation, some actions should be triggered at specific moments in time. Therefore, the module includes functions to register for timer events (at absolute dates/-times or by interval).

Automation rules sometimes imply that a device is monitored. For example, switching on the light in the room when it gets dark implies that you need to monitor the clarity status of the room. In this case (based on the asset state definition) the light sensor must be monitored. The automation module takes care of this and will request a subscription for the asset or device parameter to be sensed.

Usually, this module outputs action events to perform specific actions either in the middleware or on the peripherals.

4 Implementation

The proposed middleware with basic integrations of the IoT modules is implemented as an open source demonstrator[1].

For portability reasons, Javascript was used to integrate the reasoning engine. Combined with a custom Javascript based IoT framework to connect with the physical devices, it allows for an easy integration in both server (e.g., using Node.js) and mobile applications. The middleware's logic reasoning capabilities are provided using Tau-Prolog[23], a lightweight, open source Prolog interpreter also developed in JavaScript.

The main Prolog program in the middleware *(main.pl)* is responsible for the routing of events between the different modules in the reasoning engine. Connections define how the main program needs to route messages.

A custom interface *(connector.js)* is developed to support communication between the Prolog and outside environment. This allows events to be sent asynchronously from the application or device layer to the reasoning engine and vice versa. Each IoT module is defined as an individual Prolog module.

Two IoT modules (i.e.,*asset_ device_ conversion.pl, query.pl*) are described in detail in the following sections.

4.1 Asset-Device Conversion Module

Previously, in Section 3.2.4, the goal of the *asset-device conversion* module was presented. In the following, we shed some light on how this is achieved using Prolog.

This module registers for both update and action events. Depending on the type of event, it is handled in a different way:

```
handle(E) :- event_type(E, EType),
 event_subject(E, Subject),
 event_data(E, Data),
 data_parameter(Data, Param),
 data_value(Data, Value),
 (EType==update->(
  forall(map(Asset, AssetParam, Subject,
   ↪ Param),
    handle_update(Asset, AssetParam)));
  (EventType==action->(
    set_value(Subject, Param, Value)))).
```

First, event information is derived (i.e., the type of event, the event subject and the parameter to be modified). Next, for an ***update event*** (`EType==update`), all affected assets and corresponding parameters are retrieved using the `map` predicate. For each of these asset-parameter pairs, the `handle_update` rule is triggered:

```
handle_update(Asset,AssetParam):-
 get_value(Asset,AssetParam,Value),
 create_update_event(NewEvent, Asset,
  ↪ AssetParam, Value),
 forward(NewEvent,
  ↪ asset_device_conversion).
```

It computes the parameter value for that asset parameter (using `get_value`), and sends an new asset-specific update event back to the main application with the `forward` term. As a result, other modules are informed about the asset parameter update. Note that the module may receive the newly created event as input again, to update assets that include a parameter that depends on the parameter in this new event (e.g., the presence parameter of a residence asset will be updated when the presence parameter of one of the room assets in the residence is updated, which in turn is updated when the motion sensor in that room is updated).

The `get_value` predicate, is used to implement predefined parameter conversion rules to compute the value of an asset parameter. The following rule defines the `heat` parameter for a room asset. It converts a sensor temperature of 17°C or lower into the discrete value `low` for the heat parameter of a room asset.

```
get_value(R,heat,low) :-
  asset(R, room),
  property(D,location,R),
  get_value(D,temperature,Value),
  Value=<17.
```

It takes the value of a temperature parameter from an asset or device located in the room and checks its value. Extending the case for the discrete values `normal` and `high` is trivial. The above rule assumes that only a single device or asset with a temperature parameter is located in the room. However, sometimes several assets or devices with a temperature parameter are located in the same room. It is then possible to extend the rule and calculate the result based on all these temperature parameters. To derive the result based on the average of all temperature sensors, the rule is defined as follows:

```
get_value(R,heat,low) :-
  asset(R, room),
  findall(V, (property(D,location,R),
    get_value(D,temperature,V)), VList),
  average(Vlist, Average), Average=<17.
```

[1]The source code of the middleware may be found at *removed for blind review*

All temperature values are retrieved and put together in a list, `VList`. The average value of all the elements in the list is then calculated using the `average` predicate. The calculated average, `Average`, is then compared against the predefined value.

When no asset specific rule is found, which is the case for devices, the general `get_value` definition is used. As shown below, the current parameter value is then retrieved from the knowledge base.

```
get_value(D, ParamName, V):-
  asset(D, device),
  parameter(D,ParamName,P),
  parameter_value(P,V).
```

An ***action event*** is handled differently. After parsing the event information, the `set_value` predicate is invoked. The module only registers for action events on assets. Hence, its rules define how actions on assets are to be converted into actions in the real world. For example, the `heat` in a room can be set to `low` by setting the temperature of all heaters in the room to a value of 15.

```
set_value(R, heat, low) :
    asset(R, room),
    forall((property(Dev, location, R),
       parameter(Dev, temperature,P),
       action_type(P, write)),
       set_value(Dev, temperature, 15)).
```

These rules easily extend to more complex rules where multiple devices or assets are controlled. As an example, the `climate` of a `room` is set to `normal`, by setting both the `humidity` and `heat` to `normal`:

```
set_value(R, climate, normal) :-
    asset(R, room),
    set_value(R, heat, normal),
    set_value(R, humidity, normal).
```

The above examples show how the `set_value` rules are defined. When modifying an asset parameter means modifying another asset parameter, the rule will call the rule for the underlying asset parameter. If, however, it means that a device parameter is to be set, the following rule for changing a device parameter is invoked:

```
set_value(Device, ParamName, Value) :-
  asset(Device, device),
  parameter(Device, ParamName, Param),
  action_type(Param, write),
  create_action_event(Event, Device,
  ↪ ParamName, Value),
  forward(Event, asset_device_conversion).
```

It checks whether the parameter supports modification (i.e., write). Upon success, an action event is created and forwarded. Eventually, the action is invoked on the physical device.

This approach allows to reuse the rules defined for certain asset or device parameters when defining rules for other assets, without requiring additional logic.

4.2 Query Module

A particularly interesting consequence of using Prolog as the backbone of our middleware, is inference and reasoning. The *Query* module is responsible for handling events to query the knowledge base. As explained in Section 3.1, a query event consists of the query itself and the variable terms of interest.

The simplest solution would be to directly execute the query on the Prolog interpreter and return the query results from within the Javascript environment. However, to keep maximum flexibility (e.g., to support access control), the event is handled inside the Prolog environment. This makes things a bit more complex.

The main program forwards the received query event to the query module which processes the event. The query in the event is formatted as a string. Therefore, it is first parsed into Prolog terms. Executing the parsed query is straightforward, but retrieving its results is more complicated.The following code sample shows how to parse, execute and post-process the query execution.

```
handle_query(Query, TermList, Result):-
    parse_query(Query, ParsedQuery),
    argument_indices(ParsedQuery,
    ↪ ArgList),
    lookup(IList,ArgList,TermList),
    findall(Answer, (ParsedQuery,
    ↪ argument_indices(ParsedQuery,
    ↪ ArgListResult), lookup(IList,
    ↪ ArgListResult, Answer)), Result).
```

After parsing the query, a lookup list `ArgList` is created that lists the arguments in the query, along with their position in the query. As an example, below is a query with its corresponding lookup list:

```
?-asset_type(Hall, manufacturing_hall),
 relation(located_in, Machine, Hall).

[((1,1), Hall), ((1,2),manufacturing_hall),
 ((2,1),located_in), ((2,2),Machine),
 ((2,3),Hall)]
```

In other words, the first argument of the first predicate is `Hall`, the second argument of the first predicate is `manufacturing_hall` and so on. Based on the lookup list, and the list of requested terms, the indices of the requested terms are determined. In the example where the `Hall` and `Machine` must be returned, `IList` equals `[(1,1),(2,2)]`.

Subsequently, the query is executed multiple times to retrieve all answers.After each query execution, the variable terms are now instantiated and the answer is formed by retrieving their values and selecting only the ones mentioned in `IList`. The final list of answers, `Result`, contains the values of the requested terms for each possible answer. `Result` is then returned in a query-result event.

5 Discussion & Future Work

The main goal of the middleware introduced in this paper, is to support application developers in building responsive, easy to maintain, and smart IoT applications. Several modules have been proposed and specified, each taking care of certain functionality useful in such environments. Depending on the platform and type of application, the modules to be used may vary. In addition, adhering to some basic rules, it is easy to add new modules with new functionality and there is no need to recompile the application.

As discussed in the related work, many reasoning applications for IoT try to tackle the use of ontologies to enhance reasoning capabilities. Although ontologies may indeed have interesting applications, this paper demonstrates that even without adding these additional complexities, including a logic reasoner in the middleware already adds important benefits to IoT applications.

While integrating the data preprocessing module in an imperative programming language is an option, modules such as automation and access control greatly benefit from a rule based approach. For such cases, defining complex logic reasoning, including contextual information, is achieved with intuitive basic rules. It not only allows defining generic rules, but provides the ability to include more specific rules with limited effort. For more advanced functionality, involving reasoning, more knowledge on Prolog programming may, of course, help.

Prolog's reasoning capabilities allow for an easy integration of a query module. Providing similar capabilities using an imperative language would be difficult or nearly infeasible. Nevertheless, an imperative language may be better suited for more analytical tasks. Therefore, support is available to define script based IoT modules. These scripts handle incoming events and return new events whenever appropriate.

The demonstrator has been developed in Javascript using Tau-Prolog. This made it easy to create a cross-platform demo for both a server and mobile application. For performance or support reasons, more established reasoners such as SWI or XSB-Prolog can be preferred. Even with faster reasoners, for high throughput or large-scale IoT systems, the concurrency limitations and performance of the proposed middleware may be unacceptable. Nevertheless, it is a viable solution for many IoT applications, especially with the current drive towards edge intelligence.

Last but not least, a reactive and modular event based reasoning system was integrated using Prolog as a backbone. While often Prolog is used only for very specific reasoning tasks, this papers shows that integrating a full featured IoT middleware with complex reasoning in Prolog is a valuable alternative.

Future work. The use of Prolog also provides a number of other advantages. When defining rules and policies, it is important that they are not in conflict with one another. Although conflicts can be avoided by carefully drawing them up, a *conflict detection module* as discussed in [24] is still lacking. Furthermore, despite being very flexible, integrating fine-grained access control on query events remains a complex endeavor. Extending the access control module with automated verification of access control rules, and support for fine-grained filtering in the case of query events is left for future work.

Currently, the query module, and policies are written entirely in Prolog. Application users usually have no experience with programming languages, let alone Prolog. Providing a basic, graphical interface for creating queries and updating policies is desirable.

Last but not least, although modules are currently loaded at initialization, the dynamic loading of modules at runtime is one of the next steps to make the middleware even more dynamic. New modules could then be loaded without the need to interrupt the application.

6 Conclusion

In this paper a logic reasoning based IoT middleware is presented. It is available for integration in either cloud, mobile and gateway applications, and even inside IoT devices. A flexible event-based architecture, hosting different modules, supports application developers in building and maintaining smart and reactive IoT applications. Including contextual information in rules and policies is smooth. The integrated reasoner brings advanced intelligence to IoT applications and enforces access control and QoS policies. A demonstrator of the middleware is integrated using a Javascript based back-end. Android, iOS and NodeJS applications can be built in no time.

References

[1] X. Wang and S. Cai, "An efficient named-data-networking-based iot cloud framework," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3453–3461, 2020.

[2] A. Bhawiyuga, D. Primanita, K. Amron, O. Pratama, and M. Habibi, "Architectural design of iot-cloud computing integration platform," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 17, p. 1399, 06 2019.

[3] reference removed for blind review.

[4] "Information technology — programming languages — prolog — part 1: General core," International Organization for Standardization, Geneva, CH, Tech. Rep., 1995.

[5] Siemens. (2020) Mindsphere - connecting the things that run the world. [Online]. Available: https://siemens.mindsphere.io/

[6] General Electric Digital. (2018) Predix platform | industrial cloud based platform. [Online]. Available: https://www.ge.com/digital/iiot-platform

[7] Zetta. (2016) Zetta - an api-first internet of things (iot) platform. (accessed: 30.11.2020). [Online]. Available: https://www.zettajs.org/

[8] The MathWorks, Inc. (2017) Iot analytics - thingspeak internet of things. (accessed: 30.11.2020). [Online]. Available: https://thingspeak.com/

[9] DeviceHive. (2013) Devicehive open source iot data platform. (accessed: 30.11.2020). [Online]. Available: http://boem.com

[10] Home Assistant. (2017) Home assistant - open source home automation that puts local control and privacy first. (accessed: 30.11.2020). [Online]. Available: https://www.home-assistant.io/

[11] openHAB. (2015) openhab - a vendor and technology agnostic open source automation software for your home. (accessed: 30.11.2020). [Online]. Available: https://www.openhab.org/

[12] Applied Informatics Software Engineering GmbH. (2015) macchina.io - control and manage your devices with a secure, private connection. (accessed: 30.11.2020). [Online]. Available: https://macchina.io

[13] Blynk Inc. (2015) Blynk iot platform: for businesses and developers. (accessed: 30.11.2020). [Online]. Available: https://blynk.io/

[14] L. Sun, Y. Li, and R. A. Memon, "An open iot framework based on microservices architecture," *China Communications*, vol. 14, no. 2, pp. 154–162, 2017.

[15] H. D. S. Araújo, J. J. Rodrigues, R. D. A. Rabelo, N. D. C. Sousa, J. V. Sobral *et al.*, "A proposal for iot dynamic routes selection based on contextual information," *Sensors*, vol. 18, no. 2, p. 353, 2018.

[16] R. Calegari, E. Denti, S. Mariani, and A. Omicini, "Logic programming as a service (lpaas): Intelligence for the iot." in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, G. Fortino, M. Zhou, Z. Lukszo, A. V. Vasilakos, F. Basile, C. E. Palau, A. Liotta, M. P. Fanti, A. Guerrieri, and A. Vinci, Eds. IEEE, 2017, pp. 72–77. [Online]. Available: http://dblp.uni-trier.de/db/conf/icnsc/icnsc2017.html#CalegariD0O17

[17] R. Machado, R. Almeida, R. Albandes, A. M. Pernas, and A. C. Yamin, "An iot architecture to provide hybrid context reasoning," in *Internet of Things. A Confluence of Many Disciplines*, ser. IFIP Advances in Information and Communication Technology. Springer International Publishing, 2020, pp. 86–102.

[18] R. dos Reis, M. Endler, V. P. de Almeida, and E. H. Haeusler, "A soft real-time stream reasoning service for the internet of things," in *13th IEEE International Conference on Semantic Computing, ICSC 2019*, IEEE. IEEE, 2019, pp. 166–169. [Online]. Available: https://doi.org/10.1109/ICOSC.2019.8665668

[19] P. Bonte, F. Ongenae, and F. De Turck, "Subset reasoning for event-based systems," *IEEE Access*, vol. 7, pp. 107 533–107 549, 2019.

[20] G. Chen, T. Jiang, M. Wang, X. Tang, and W. Ji, "Modeling and reasoning of iot architecture in semantic ontology dimension," *Computer Communications*, vol. 153, pp. 580–594, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419318912

[21] E. Pencheva, I. Atanasov, A. Nikolov, and R. Dimova, "Ontology and reasoning on device connectivity," in *Proceedings of International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST), Nish, Serbia.* Publishing House, Technical University of Sofia, june 2017, pp. 157–160.

[22] P. Samarati and S. C. de Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 137–196.

[23] J. A. R. Valverde. (2020) Tau prolog - an open source prolog interpreter in javascript. (accessed: 30.11.2020). [Online]. Available: http://tau-prolog.org/

[24] A. Al Farooq, E. Al-Shaer, T. Moyer, and K. Kant, "Iotc 2: A formal method approach for detecting conflicts in large scale iot systems," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM).* IEEE, 2019, pp. 442–447.