Developing IoT Artifacts in a MAS Platform

Subjects: Computer Science, Artificial Intelligence

Contributor: Vicente Julian

The Internet of Things (IoT) is a computational paradigm where a massive number (perhaps billions) of ordinary objects are endowed with interconnection capabilities, making them able to communicate and cooperate with other (surrounding) devices, generally via the Internet.. The Internet of Things (IoT) is a growing computational paradigm where all kinds of everyday objects are interconnected, forming a vast cyberphysical environment at the edge between the virtual and the real world. Since the emergence of the IoT, Multi-Agent Systems (MAS) technology has been successfully applied in this area, proving itself to be an appropriate paradigm for developing distributed, intelligent systems containing sets of IoT devices. However, this technology still lacks effective mechanisms to integrate the enormous diversity of existing IoT devices systematically.

multi-agent systems IoT agent platforms artifacts

1. Introduction

The Internet of Things (IoT) is a computational paradigm where a massive number (perhaps billions) of ordinary objects are endowed with interconnection capabilities, making them able to communicate and cooperate with other (surrounding) devices, generally via the Internet. The growing effect of this paradigm is the appearance of a vast, decentralized, heterogeneous, and dynamic ecosystem where everyday objects (sensors, gadgets, tags, wearables, etc.) become active participants in processes of all kinds, such as industrial, logistic, domotics, social, health care, etc.

In this paradigm, the “things” that become interconnected, generally via the Internet, are sometimes called “smart objects.” However, many objects used currently still lack actual intelligence mainly due to their limited hardware and software resources. This lack has hindered the development of intelligent end-to-end solutions in the IoT arena, which can effectively integrate different AI techniques in a simple, transparent, and distributed way. In this sense, since the emergence of the IoT in 1999 [1], Multi-Agent-Systems (MAS)-based technology has fostered the connection of small, commonly used devices to open distributed intelligent systems, enabling these devices to exchange and transmit knowledge in real time [2]. Furthermore, there is remarkable parallelism between the Agent-Based Computing (ABC) and Multi-Agent Systems (MAS) paradigms and the smart object and IoT ecosystem concepts, respectively. Hence, many researchers have extensively used such paradigms methodologically in the IoT domain, as well as to model, program, or simulate IoT systems [3].

One class of MAS that fits the requirements of the IoT is open multi-agent systems, which has received significant interest from the scientific community in recent years. Open multi-agent systems are defined as open systems

consisting of heterogeneous entities with a separation between form and function that explains their behavior [4], and they are particularly suitable for the implementation of virtual organizations. Recent research conducted on the modeling and implementation of open MAS in complex scenarios includes the works published in [5][6][7]. Numerous proposals have worked on improving the intelligence of IoT systems (e.g., a MAS equipped with swarm intelligence [8]). However, in most cases, current IoT networks are still incapable of generating cooperative strategies that make these networks act as ubiquitous and intelligent systems [9].

In particular, one critical problem of the IoT is its intrinsic heterogeneity. According to [10], the heterogeneity of devices and the high technological diversity in the IoT impose an enormous modeling effort for large-scale systems, where thousands of different devices may coexist. At this moment, multi-agent systems lack mechanisms to deal with this diversity effectively.

2. The SPADE Platform

SPADE [11] is a multi-agent system platform whose primary purpose is to provide a flexible, simple, and open agent execution framework. The cornerstone of this platform is the employment of a communication mechanism based on the XMPP standard [12] for instant messaging, which is the same one typically used in a chat program. Therefore, humans can interact with software agents as they would with other humans by connecting to XMPP servers and exchanging “chat messages”.

The two main characteristics of the SPADE platform are the extensive and strategic usage of the XMPP standard and its proposed agent model, which are now presented in the two following subsections.

2.1. XMPP

The XMPP protocol provides the necessary elements for real-time conversations. In addition to exchanging messages, which can be used between agents, between humans, and even between agents and humans, XMPP has a presence notification system, which lets contacts know if their contact list or roster is online or unavailable. Since the IETF formalized XMPP as the standard for instant messaging and presence notification, it is now an open standard that offers several compelling features:

- Decentralized: XMPP is based on an architecture similar to email. In particular, it features a client–server architecture in which the clients connect to a private server or a public one. Servers exchange messages between them (as mail servers do) to deliver each message to its recipient;
- Secure: XMPP has a robust security system including a secure transport layer and a secure authentication system that allows for establishing ciphered communications between entities. In addition, an XMPP server may be isolated from the Internet if required;
- Extensible: XMPP is based on XML, allowing it to easily include new features in the protocol to extend its capabilities. A set of extensions to the protocol (called XEPs) is continuously improved, but it is also open to everyone to build their private extensions to fit any particular need;

- Flexible: Besides instant messaging, there are numerous applications for which XMPP can be used. Agent communication is just one application, but XMPP is also used for many other purposes, such as network management, collaboration tools, gaming, file sharing, content syndication, web services, or remote system monitoring;
- Proven: XMPP was initially proposed in 1998 by Jeremie Miller, and currently, it is a very stable and well-tested standard, with hundreds of developers and tens of thousands of XMPP servers deployed around the world. Some big companies use XMPP (or a protocol modification) as the core of their services (e.g., WhatsApp, Google Talk, Facebook Messenger);
- Open: The XMPP protocol is free, open, public, and easy to understand. There are no limits for the implementations and the collaboration in the standard development.

This protocol is the core element of the SPADE platform because agents need an adequate and efficient transport layer that can be extended to foster new types of interactions (computer-to-human, computer-to-computer, human-to-human) and tackle new requirements or domains successfully. In this sense, it is worth mentioning that a working group inside the XMPP Foundation is devoted to studying the application of XMPP to the IoT domain. The support defined by the XMPP standard perfectly fits the main requirements of the IoT, such as the need for communication protocols and standards, the usage of communication patterns (publish/subscribe, event subscription, delayed delivery, etc.), scalability, security, and interoperability, among others. The model of the IoT artifact presented later in this paper takes advantage of these features to provide appropriate support to artifacts in this domain, including new functionalities such as presence notification.

2.2. The Agent Model

Agents in SPADE are autonomous entities with a transport layer based on the XMPP protocol. By design, the activities that agents perform are encapsulated into components called behaviors. Every agent may define one or more behaviors, and the platform executes them independently. In addition, the agent has a connection mechanism called the message dispatcher to deliver the agent's incoming messages to each of its behaviors. This proposal is similar to those available on other platforms, such as JADE.

The main characteristic of a behavior is its life cycle, which depends on how the behavior runs. SPADE offers different behaviors, in particular: `CyClicBehavior`, which runs forever in an infinite loop until the agent is stopped; `OneShotBehavior`, which runs just one time and then is destroyed; `PeriodicBehavior`, which runs every pre-defined *period* of time; `TimeoutBehavior`, which is a subtype of `OneShotBehavior`, which runs after a timeout. Finally, a more complex type of behavior allows the agent developer to create finite-state machines, which gives the developer a more powerful control over the design of the agent. As shown in **Figure 1**, `CyclicBehavior` is the base of all the other behaviors.

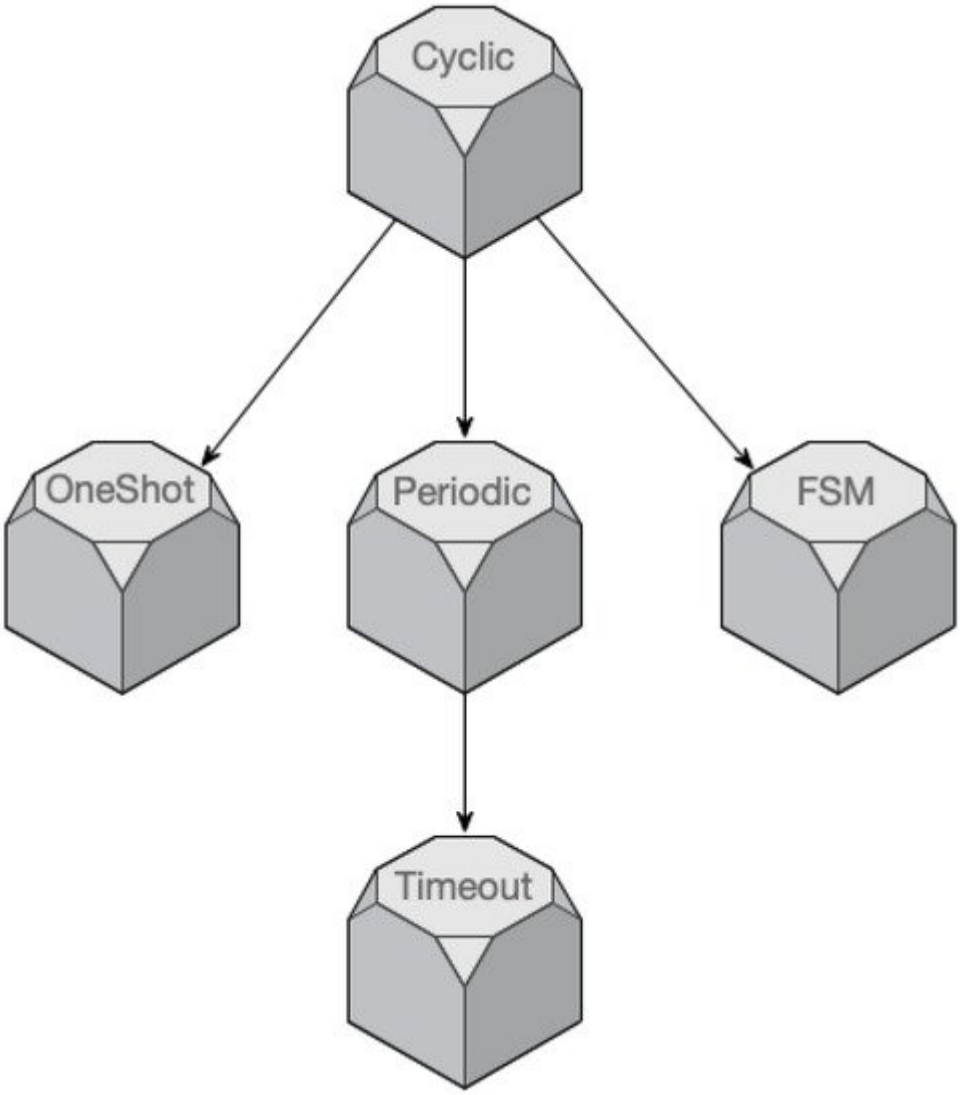

Figure 1. SPADE's behavior hierarchy.

In addition, SPADE has recently incorporated BDI behaviors [13]. This new class of behavior allows for the development of agents that operate on desires and intentions, coded in the AgentSpeak language [14].

Apart from the agent model, SPADE offers agent developers many functionalities and much flexibility to build their multi-agent system applications. The main ones are now highlighted. First, designers can easily integrate complex perception behaviors (such as artificial vision or natural language processing) by using the advantages offered by a language such as Python. Second, SPADE has been developed by following an asynchronous programming model to increase the developed applications' performance and responsiveness. This programming model improves the scalability of MAS implementations by optimizing the send and receive operations (as well as any other I/O operation), which is a crucial aspect in IoT environments, where the system may need to interact with hundreds or thousands of devices. Third, although there is a complete reference implementation of SPADE in Python, the SPADE framework is, in fact, language-agnostic. As long as the implementation follows the communication protocols defined by the platform (based on the XMPP standard), agents may be implemented in any language. For example, implementing SPADE agents in the C language may be appropriate in embedded systems with

scarce hardware resources. Fourth, SPADE favors the incorporation of new functionalities as plugins, which makes it easy for the community to extend the support. Examples of recent plugins are the *spade-bdi* and *spade-pubsub* plugins, which have incorporated the BDI behaviors and the publish–subscribe protocol, respectively.

Regarding the design and implementation of multi-agent systems, SPADE provides the developer with the agent concept to model the system's intelligent behavior. However, a much simpler and lighter abstraction was needed to adequately model the environment, especially in cyberphysical and IoT scenarios. In such scenarios, the system typically accesses the environment through a series of small devices with scarce computational resources. To this end, the following section incorporates the concept of the IoT artifact into SPADE to facilitate the development of SPADE-based MAS in the context of IoT environments.

3. The IoT Artifact

This section presents a specialization of the *A&A* meta-model described in the previous section, which is called the *IoT artifact*. This specialization allows for the modeling of IoT devices and their implementation in the SPADE platform. The *IoT artifact* model attempts to maintain the expressiveness of the previously presented theoretical model while also considering the specific aspects of IoT devices, as their limited computational resources, and integrating all these characteristics into the programming model of the SPADE platform.

In summary, the *IoT artifact* specialization model proposes a correspondence between each of the elements and an entity in the SPADE platform:

- IoT artifact. An IoT artifact is a new SPADE computational element that can communicate with agents (through an XMPP server). IoT artifacts associate with a workspace by registering to the corresponding XMPP server and present a well-known interface by which SPADE agents may use them, as described below. This interface includes all the characteristics of the theoretical model, except the so-called *linked interface*, which SPADE does not support due to the distinct shortage of the computational resources of IoT devices.
- Compared to the theoretical model, an IoT artifact always includes a particular observable property called *presence*, which maintains the current state of the associated IoT device. By using this property, agents interested in a given IoT device may know its availability and any other application-specific status information that the artifact can express;
- Workspace. The theoretical concept of workspace here corresponds to an XMPP server, which is the component in the SPADE platform that supports the communication among all the SPADE communicating parties (agents and artifacts). In this model, any IoT artifact must register to an XMPP server before being accessible to agents. IoT artifacts register (and therefore belong) to a single XMPP server;
- Environment. Following the workspace definition above, this concept would be equivalent to the group of all the XMPP servers involved in a particular multi-agent system;
- Agent. This entity corresponds to a SPADE agent. SPADE agents can communicate with other agents and artifacts, among other features.

Table 1 compares the basic features and properties that are essential for artifacts independent of the implementation model, according to [15]. The table also includes some relevant implementation considerations, in each case comparing its availability in the CArtAgO platform and in the *IoT artifact* framework. The main novelty of the IoT artifact's proposal is the consideration of the typical characteristics of IoT devices, to which the model has been targeted. In particular, the strict limitation of computational resources that is common in such devices has been especially taken into account. As a result, a *minimal* artifact model has been proposed, by which artifacts can be implemented in languages such as Python or C, and be directly executed in small, embedded devices. On the contrary, the CArtAgO approach requires a Java virtual machine to execute the artifact's code. However, it is important to point out that, despite being *minimal*, the IoT artifact model incorporates all the features of the abstract model, except the linked interfaces, as they can produce too much computational cost for small devices.

Table 1. Comparison of features between CArtAgO and IoT artifacts.

Features	CArtAgO	IoT Artifacts
Identity	Full name (including Workspace)	JID (Jabber ID)
Usage interface and events	Set of operations and observable events	Op. interface: Jabber-RPC, default observable property (presence)
Function description and operating instructions	Yes	Not available in the current version
Observable state	Yes	Presence
Programming language	Java	Python 3, Python 2.7, C
Virtual machine needed	Yes	No
Linked interface	Yes	No

Regarding communication aspects, SPADE agents may communicate with any IoT artifact registered to any workspace (XMPP servers) known to the agent. To do so, a SPADE agent needs first to send a *focus* request to the XMPP server, expressing an interest in that particular IoT artifact. Once under its focus, the agent will be able to interact with the IoT artifact by using its interface.

The interface of an IoT artifact defines two types of interactions. The first type permits accessing the artifact's observable properties (perceptions), including the presence property mentioned above. The second one allows for the artifact's operation, which typically will modify its internal state or make the artifact actuate over the environment or both. The following subsections explain these interface features in further detail, which are related to the functionalities of the XMPP protocol adopted by SPADE.

3.1. Perception of Observable Properties

IoT artifacts generally perceive their environment by using physical sensors attached to the corresponding IoT device and change their internal variables accordingly. Such variables correspond to the observable properties in the meta-model above. As a result, every time one of these observable properties changes its value, the IoT artifact should generate the corresponding observable events to communicate the change to the interested agents.

As explained above, agents must focus on an artifact before interacting with it. In order to focus and also to observe the artifact's properties, SPADE proposes to use an extension of the XMPP protocol called Publish–Subscribe (PubSub) https://xmpp.org/extensions/xep-0060.html. This extension enables any individual connected to an XMPP server to subscribe to the information that any other connected entity may want to share. Once subscribed, the interested individual automatically receives updates any time the entity publishes new information.

The mechanism works in two steps. First, an agent sends a message to the workspace (the XMPP server) to subscribe to an IoT artifact. Then, whenever the artifact generates a new observable property value or event, it publishes the event with the updated information, which all subscribed agents receive. This way, agents may keep track of the information they are interested in by focusing on the corresponding IoT artifacts. **Figure 2** illustrates these interactions by the dashed lines, where Agent 1 focuses (subscribes) on Artifact 1, which perceives the environment temperature, and then, it automatically receives the published events corresponding to temperature changes perceived by the artifact.

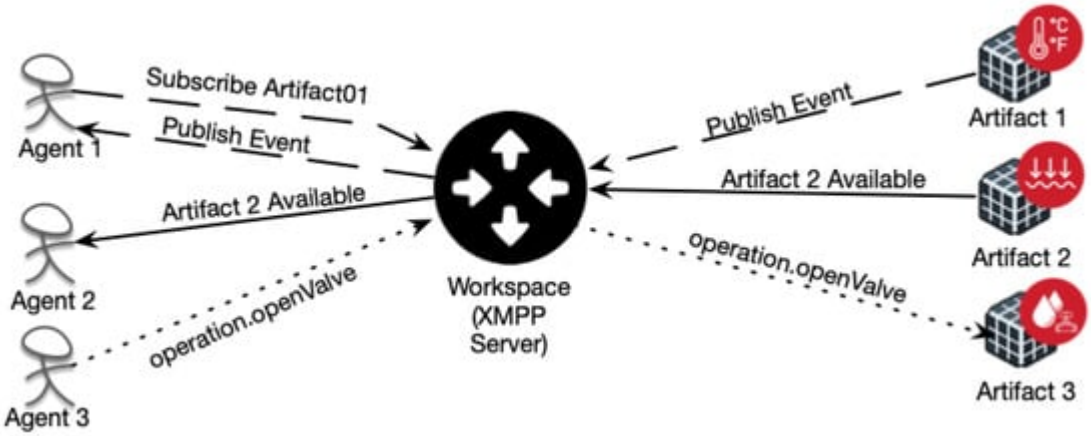

Figure 2. Examples of interactions between IoT artifacts and agents in SPADE.

3.2. Presence Notification for Artifacts

Presence notification is a typical feature of SPADE agents that has also been incorporated into IoT artifacts since it is considered an advantageous property for artifacts in IoT scenarios.

In essence, presence notification enables any entity connected to an XMPP server to know the availability status of other connected entities (customarily called the former entity's *contacts*) and also to notify its own availability status to these contacts. This simple yet powerful mechanism can be used for many different purposes (e.g., as a coordination protocol in distributed systems), and it is helpful in many scenarios. The presence notification mechanism of XMPP offers the possibility of including custom messages related to each entity's availability (such

as free, busy, or waiting), but it also sets the status as *unavailable* if the entity's connection suddenly drops out. Thus, IoT artifacts can notify their availability (and any other status) to the interested agents in real time through this handy feature, allowing them to know if the artifacts are ready to communicate or if they are having some issue. This way, for example, an agent could decide whether or not to request an operation on the artifact or ascertain why it is not receiving updates from the artifact's observable properties recently. In the latter case, the presence notification system could inform the agent of the artifact's situation: it has been disconnected; it is experiencing some technical problems; it needs maintenance; it is simply busy performing other tasks. A simple interaction of this type is shown by the solid lines in **Figure 2**, where Artifact 2, representing a pressure sensor, becomes available, and this is automatically published to any interested agents, as Agent 2 in the figure.

3.3. Operation of IoT Artifacts

SPADE employs another standard extension from the XMPP protocol to implement the operation interface over IoT artifacts. This XMPP Extension Protocol (XEP) is called Jabber-RPC https://xmpp.org/extensions/xep-0009.html, and it allows any entity connected to an XMPP server to make available its operations to other entities by using a well-known Remote Procedure Call (RPC) standard: XML-RPC. By incorporating this standard into SPADE, agents can send a message with the required operation to an artifact and receive a response, both in a structured form defined by the standard.

In Listing 1, a typical request message is shown. This example illustrates how to request an artifact to open Valve Number 4 to 50%, which is also graphically represented as dotted lines in **Figure 3**, where Agent 3 operates the valve actuator of Artifact 3.

Listing 1. An example of a request message to an artifact.

```
<iq type='set' id='rpc1'
 from='agent@workspace.com/jrpc-client'
 to='artifact@workspace.com/jrpc-server'>
 <query xmlns='jabber:iq:rpc'>
 <methodCall>
 <methodName>operation.openValve</methodName>
 <params>
 <param><value><i4>4</i4></value></param>
 <param><value><i4>50</i4></value></param>
 </params>
 </methodCall>
 </query>
</iq>
```

3.4. Creating an IoT Artifact

A specific library has been developed to allow for the implementation of IoT artifacts in SPADE in a simple way. Its installation is performed by including the *spade_artifact* package. Once this package is installed, the developer can create instances of the *artifact* class, which is an extension of an abstract class that provides the *PubSub* protocol as shown in **Figure 3**. According to this figure, the main methods offered by the class are the following: the *start* method, which is invoked to start the artifact execution; the *setup* method, which allows an initialization adjusted to the domain; the *run* method, which is the method that includes the code to be executed by the artifact. Other auxiliary methods are the *send* and *receive* methods used for sending and receiving messages and the *publish* method for publishing information according to the *PubSub protocol*.

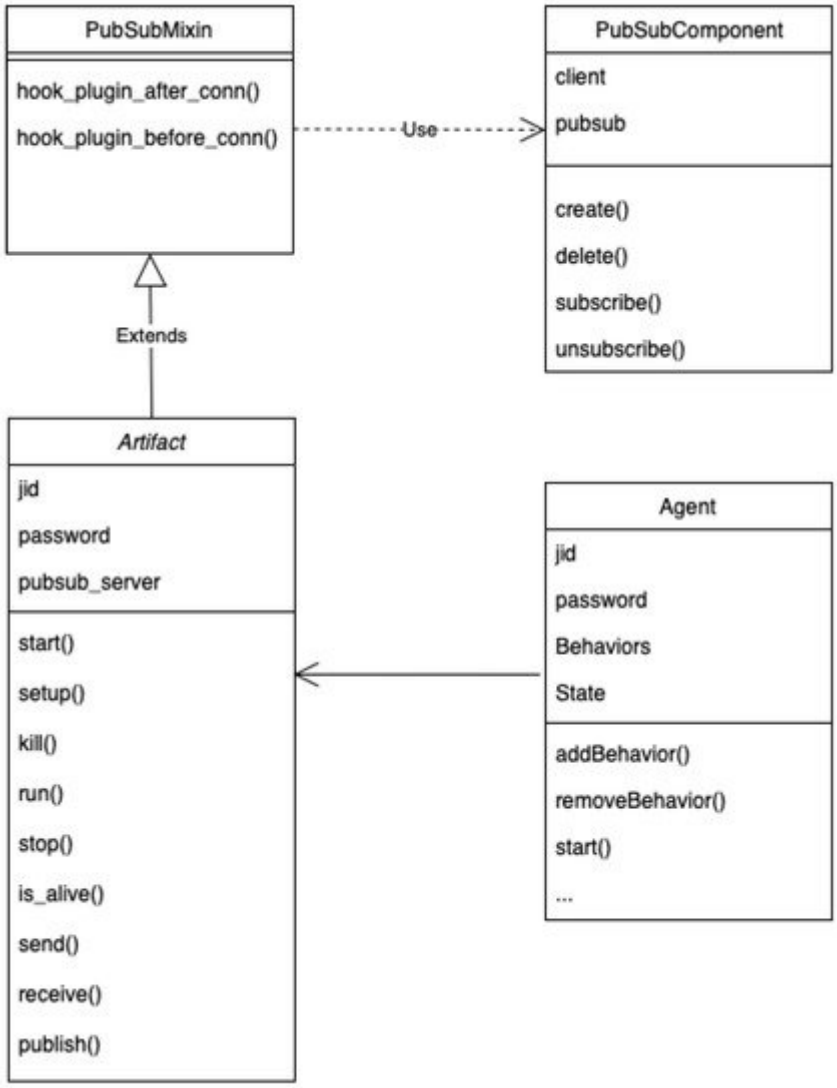

Figure 3. Class diagram of an IoT artifact in SPADE.

According to this structure, Listing 2 shows a simple example of an artifact devoted to the publication of the temperature of a particular sensor to all the interested (subscribed) agents. The shown code is incomplete, as it focuses on the overwriting of two methods only. In the *setup* method, the artifact, using the presence functionality,

makes itself visible and then accepts by default all the agents that may request the subscription. Subsequently, the *run* method enters in an infinite loop that first detects if there are any agents in its contact list and then reads the current temperature value and publishes it. This simple example illustrates how easy integrating artifacts in the multi-agent system is.

Listing 2. An example of an implementation of an artifact in SPADE.

```
class TemperatureSensorArtifact(spade_artifact.Artifact):

 async def setup(self):
       """
 Setup artifact before startup.
 """
 self.presence.set_available()

 async def run(self):
 while True:
  # Publish only if my friends are online
  if len(self.presence.get_contacts()) >= 1:
   temperature = read_temperature()
   await self.publish(f"{temperature}")
   logger.info(f"Publishing {temperature}")
  await asyncio.sleep(1)
```

The *spade_artifact* package described above includes the Python implementation of the IoT artifact now included in the reference version of the SPADE middleware. However, a Python implementation may not be appropriate for many IoT devices. For example, devices based on the ESP32 or the ESP8266 micro-controllers do not support this language due to their limited architecture. Thanks to the language-agnostic trait of SPADE, it is possible to implement the IoT artifact model in different programming languages.

The most obvious choice for IoT devices would be the C language since it is still the most widely used language for programming embedded systems. For this reason, a C implementation of the IoT artifact has also been developed. As an example, Listing 3 shows a C implementation of an artifact that is equivalent to the one presented in Listing 2.

In this case, the definition of an artifact starts with the connection to the WiFi network (method *wifi_connection()*), which requires the configuration of the *WiFi SSID* and *WiFi password*. The second step is the connection to the *XMPP* server, which the artifact performs by calling the *init_communication_with_xmpp_server()* method. The third step is to determine if this artifact is visible to the agents, for which the *presence_show(true)* method is used. Then, the artifact enters its main loop, where it uses the *get_num_available_contacts()* to obtain the number of available

contacts (agents) that have subscribed to its presence and are currently online. If this value is at least one, the artifact reads the temperature value and then publishes it to all the contacts subscribed to that observable property.

Listing 3. An example of an implementation of an artifact in C.

```
int temperature_data = 0;
char temperature_str[10];

void main(){
 wifi_connection();
 init_communication_with_xmpp_server();

 presence_show(true);

 while(1){
 // Publish only if my friends are online
 if(get_num_available_contacts() >= 1) {
  temperature_data = read_temperature_data();
  itoa(temperature_data, temperature_str, 10);
  publish(temperature_str);
 }
 delay(1000);
 }
 }
```

This example illustrates the versatility of SPADE in communicating with very-low-powered systems, allowing direct communication between the agent and the IoT artifact (running in the device), even if they are implemented in different languages.

References

1. Rose, K.; Eldridge, S.; Chapin, L. The internet of things: An overview. Internet Soc. (ISOC) 2015, 80, 1–50.
2. Atzori, L.; Iera, A.; Morabito, G. The internet of things: A survey. Comput. Netw. 2010, 54, 2787–2805.
3. Savaglio, C.; Ganzha, M.; Paprzycki, M.; Bădică, C.; Ivanović, M.; Fortino, G. Agent-based Internet of Things: State-of-the-art and research challenges. Future Gener. Comput. Syst. 2020, 102, 1038–1053.

4. Foster, I.; Kesselman, C.; Tuecke, S. The anatomy of the grid: Enabling scalable virtual organizations. High Perform. Comp. Appl. 2001, 15, 200–222.
5. Bajo, J.; Julian, V.; Corchado, J.; Carrascosa, C.; de Paz, Y.; Botti, V.; de Paz, J. An execution time planner for the ARTIS agent architecture. Eng. Appl. Artif. Intell. 2008, 21, 769–784.
6. Leitao, P.; Karnouskos, S.; Ribeiro, L.; Lee, J.; Strasser, T.; Colombo, A.W. Smart agents in industrial cyber–physical systems. Proc. IEEE 2016, 104, 1086–1101.
7. Wang, S.; Wan, J.; Zhang, D.; Li, D.; Zhang, C. Towards smart factory for industry 4.0: A self-organized multi-agent system with big data based feedback and coordination. Comput. Netw. 2016, 101, 158–168.
8. Giordano, A.; Spezzano, G.; Vinci, A. Smart agents and fog computing for smart city applications. In Smart Cities, Proceedings of the First International Conference, Smart-CT 2016, Málaga, Spain, 15–17 June 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 137–146.
9. Wu, Q.; Ding, G.; Xu, Y.; Feng, S.; Du, Z.; Wang, J.; Long, K. Cognitive internet of things: A new paradigm beyond connection. IEEE Internet Things J. 2014, 1, 129–143.
10. Ayala, I.; Amor, M.; Horcas, J.M.; Fuentes, L. A goal-driven software product line approach for evolving multi-agent systems in the Internet of Things. Knowl.-Based Syst. 2019, 184, 104883.
11. Palanca, J.; Terrasa, A.; Julian, V.; Carrascosa, C. SPADE 3: Supporting the New Generation of Multi-Agent Systems. IEEE Access 2020, 8, 182537–182549.
12. Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, RFC Editor. 2011. Available online: https://xmpp.org/rfcs/rfc3920.html (accessed on 8 February 2022).
13. Rao, A.S.; Georgeff, M.P. BDI agents: From theory to practice. In Proceedings of the ICMAS, San Francisco, CA, USA, 12–14 June 1995; Volume 95, pp. 312–319.
14. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. Programming Multi-Agent Systems in AgentSpeak Using JASON; John Wiley & Sons: Hoboken, NJ, USA, 2007; Volume 8.
15. Ricci, A.; Viroli, M.; Omicini, A. Construenda est CArtAgO: Toward an Infrastructure for Artifacts in MAS. Cybern. Syst. 2006, 2, 569–574.