Intelligent Checkpointing Strategies for IoT System Management

François Aïssaoui, Gene Cooperman, Thierry Monteil, Saïd Tazi

Intelligent Checkpointing Strategies for IoT System Management

François Aïssaoui[1], Gene Cooperman[1,2], Thierry Monteil[1], Saïd Tazi[1]
[1]LAAS-CNRS, Université de Toulouse, CNRS, INSA, UT1 Capitole, Toulouse, France
[2]College of Computer and Information Science Northeastern University, Boston, MA / USA
Emails: aissaoui@laas.fr, gene@ccs.neu.edu, {monteil, tazi}@laas.fr

***Abstract*—The Internet of Things (IoT) continues to expand in terms of the number of connected devices. To handle the data produced by those devices, gateways are deployed to collect data, possibly to analyze it, and finally to send it to the cloud or to the end-user to support new services. This process involves complex software that is deployed on those gateways. Moreover, the dynamicity due to new services, mobility, etc., could be corrupted by new events that then require the deployment of software components on additional equipment. Those new events arise in at least two fundamental ways: devices that may change their geographical location; and limitations due to hardware resources and energy consumption. We propose to use autonomic monitoring and control in response to a changing environment in order to manage deployed software with little or no human intervention. A new generic approach is described, based on a semantic model of the system being monitored. Much of the power of the proposed approach is accomplished through a novel use of checkpointing in order to control the software deployed on the gateway.**

***Index Terms*—Semantics, Ontology, Internet of Things, Checkpointing, System Management, Self-healing**

I. Introduction

The Internet of Things (IoT) is a growing field with a prediction by Gartner of 20 billion connected devices in 2020[1]. Those devices use gateways to communicate with the cloud, the web services and the users. Usually, the gateway consists of cheap and low-powered machines with a reduced energy consumption. These under-powered machines are not able to fully handle exceptional events arising from the large, complex software. Examples of such exceptional events include failure recovery and event reporting. Since an IoT application is often large (for example, transport, logistics, e-health, automotive, etc. [1]), the problem is best tackled through a generic approach.

The problem of management of such a system becomes more difficult in the case of multiple gateways to handle all the devices. There is a need to: monitor this kind of system; discover the symptoms that impede the correct system operation; determine their cause; and then decide on actions to perform — all while requiring only limited human interaction.

We propose a novel, intelligent framework based on semantic reasoning, which makes use of a checkpointing mechanism to more flexibly manage such an IoT system. Traditional monitoring systems manage such complex systems while continuing to tightly bind software processes and their associated devices to the original gateway on which they run. By using a checkpointing mechanism, software can be migrated from one gateway to another. *Checkpointing* is the ability to save the state of a running process to a file in stable storage, where the process may later be restarted and continued using that file. Since files can be copied between gateways, this immediately enables *software migration* between gateways.

The ability to migrate software is useful in at least two cases for IoT. First, the ability to migrate software is valuable in cases such as logistics, transport, and other domains where the devices may change their geographical location. Second, migration may be required when a gateway has limited RAM, CPU cores, battery-based energy, etc. In both of these cases, even the mapping of devices to IoT gateways may change over time. Rather than terminate a process and start a new one when a device moves, one simply migrates an associated process, complete with all the device data and history contained in that process. There is no need to follow the error-prone process of exporting and re-importing device data in a standardized manner.

To manage the possibilities offered by a checkpointing mechanism, we need to represent the knowledge of the IoT system in a formal way. For this purpose, we choose semantic technologies which use standards provided by the World Wide Web Consortium (W3C) to express vocabularies. Those vocabularies provide a formal description of concepts from a specific field. With the use of a common vocabulary, different entities can exchange information in a formal way. Moreover, the model defined by ontologies can include an embedded set of rules. Those rules and the description allow a reasoner to infer new knowledge related to the provided data. This is used in our contribution in order to determine issues in the managed system.

Our framework uses semantic description to represent a complex IoT system while respecting its constraints. Thus, by providing information on the current system state, any symptoms and requests for change can be inferred by the semantic reasoner, based on the rules and the constraints of the system. Then, using those symptoms and a request for change, the framework generates a plan of action to perform on the system in order to restore itself to a correct state.

As a proof of concept, we demonstrate the semantic model through a logistics application. In this application, the transport of a physical package may have specific requirements. To

[1]http://www.gartner.com/newsroom/id/3598917

satisfy those requirements, sensors can be placed on the physical package in order to monitor the environment. The data produced by the sensors must be sent to the nearest compliant gateway, where it can be forwarded to the specific high-level application for processing. But if the package moves, which occurs frequently in logistics applications, the software that ensures communication between the package and the high-level application must then be migrated. Our framework, based on semantic description, is able to represent this kind of scenario and also perform the needed software migration through a checkpointing mechanism.

In Section V, we will model a scenario involving a *box of vaccines*. The transport of a box of vaccines requires that the box remain within a certain range of temperature and humidity in order to ensure the quality of the vaccines. As trucks arrive at a transport center, boxes of vaccines are migrated to an appropriate warehouse according to vaccine type.

The paper is organized as follows. Section II presents the background and related work on checkpointing strategies and the "standard" usage of semantics in the IoT. Section III presents our contribution, describing the ontology used to manage the system and how the interaction is managed. Section IV describes two instances of the proposed model as applied to logistics. Section V provides an experimental analysis of the scalability of the semantic manager. We then present conclusions and future work in Section VI.

II. Background and Related Work

An overview of the related literature is provided next. First, an overview of a checkpointing mechanism using Distributed Multi-Threaded CheckPointing (DMTCP) is provided. Then, the usage of the semantic technologies in the domain of IoT is discussed. Finally, we compare our approach to the paradigm of autonomic computing, and provide the background for our use of some well-known IoT standards.

A. Checkpointing Mechanism

Checkpointing enables the creation of images (snapshots) of a running process. The checkpoint image allows one to recreate the process, and even to migrate it to another computer if needed.

In this work, the semantic framework manages both the software processes and their migration as required by the constraints of the software. To perform this migration, a checkpointing mechanism is used, implemented by DMTCP [2]. DMTCP provides a *transparent* checkpointing mechanism that provides for checkpoint/restart without any modification of the original application code or operating system. DMTCP also provides a plugin facility to adapt the transparent checkpointing capability of the target application to external subsystems, such as the handling of a network connection [3].

It has been demonstrated that the checkpointing mechanism using DMTCP can be adapted to fit the IoT domain in term of performances in [4]. This current work does not focus on the performance of the checkpointing mechanism itself but on how to use this mechanism to provide an intelligent manager framework for IoT.

B. Semantics usage in the IoT

Semantic technologies and linked data are used to provide shared vocabularies that enable the interaction between different components. They follow W3C standards for these technologies and their implementation. Serrano et al. [5] recommend the use of semantic technologies for IoT to provide interoperability in the case of heterogeneous data.

Different types of data can be formalized by the semantic models. Sheth et al. [6] provide a fundamental approach to sensor data interoperability through semantic modeling. This formalization facilitates the development of generic applications that require data for a sensor network. Barnaghi et al. [7] also provide data interoperability for sensors through semantics to facilitate data integration and service discovery in the IoT system.

A different point of view is taken by Desai et al. [8]. The authors directly model the description of the nodes in their ontology, i.e., for the sensors and the gateway. This allows the representation of the capabilities of the nodes and facilitate the creation of new services. They also describe the gateways as the main interface between the devices and high-level business applications. The role of the gateways is to translate the fuzziness of the sensor networks into well-known and standardized protocols. This shows the importance of gateways in the IoT architecture and the software that supports this interface. Our approach provides for the execution of the target software in the framework of their own requirements.

C. Autonomic Computing and Device Management

In a manifesto of IBM from 2001 [9], Paul Horn describes the growing complexity of the software ecosystem and industry. More and more, the development of software requires increasing care to ensure the smooth functioning of such systems. This vision has been discussed in a work by Kephart et al. [10] that provides four features needed for autonomic computing: 1) self-configuration; 2) self-optimization; 3) self-healing; 4) self-protection.

In our approach, we are using the *self-healing* feature to implement our manager framework. To represent the data of the managed system, we will use autonomic computing paradigm vocabulary for the "Event", "Symptom", "Request for Change" and "Plan" that will be included in our ontology. Then, to interact with the system, our framework uses a *device management protocol* such as Lightweight M2M (LWM2M)[2], a standard that has been proposed by the Open Mobile Alliance (OMA). This enables us to retrieve such information from the gateways as the levels of RAM usage, CPU usage and disk space. This allows the system to use the DMTCP checkpointing component to propose the checkpoint and restart operations for the processes under consideration.

III. Framework Description

Our framework uses multiple components and technologies to achieve the management of an IoT system.

[2]http://openmobilealliance.org/iot/lightweight-m2m-lwm2m/

The first part is the ontology containing the vocabulary and the rules of the framework. It is described in the following, in sub-section III-A.

The second part corresponds to the strategies used to operate on the system being managed. This second part uses the inferences of the reasoner operating on the ontology and the data of the system in order to determine the actions required to perform. This component is described later, in sub-section III-B.

A. System Representation

The first part of the system is in charge of finding the *symptoms* and *potential actions* to perform on the system to resolve those symptoms. To determine that information, the component uses an ontology, the semantically enriched data of the system, and a reasoner to infer new knowledge based on the rules.

A *symptom* corresponds to an entity or parameter of the system that has an issue. This can be seen as a *symptom* within the paradigm of autonomic computing.

1) ***Ontology classes****:* The first part of the ontology describes the deployed *System* in terms of *Machines* and *Devices*.

The *System* class corresponds to an abstract concept of the target system being managed. A System instance is linked to the other instances to be managed within the current scenario and is also linked to the policies that need to be applied.

A *Machine* corresponds to a physical (or virtual) entity that runs an operating system capable of hosting and executing some software. Some example of sub-classes of the Machine class are "Gateway" and "VirtualMachine" (see Figure 1).

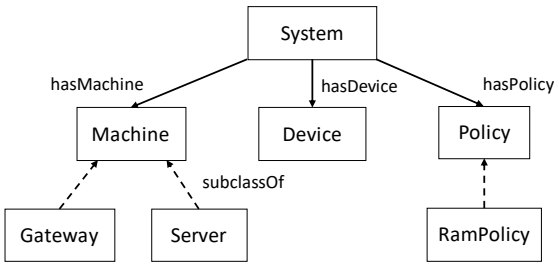

Fig. 1. Main System components

A *Device* is a physical connected entity which is able to sense or act on the environment. A Device can be connected to a Gateway to send its information to an application. It may use a communication protocol, such as Bluetooth or Wi-Fi, in order to send or receive information from a gateway. This concept is aligned with the definition of Sensor from the SSN ontology[3].

A *Network* corresponds to a communication Network. It allows one to determine which entity is reachable through the Network.

A *Policy* correspond to a general requirement for the managed system. If the policy is not satisfied after a modification of the system or a fluctuating parameter, a symptom has to be issued and changed to be applied to the system. An example of a sub-class of Policy is RamPolicy which defines the minimum available RAM required on the gateway of the system.

On the Machines, software is executed and needs to be represented. An abstract class *SoftwareEntity* is defined to represent an entity that is executed on a Machine. "Application" and "Process" are sub-classes of SoftwareEntity. The *Process* class represent a process in term of an "Operation System" that is executed on a Machine. An *Application* is an abstract entity that provides a set of features or services. An Application represents complex software that can be split into multiple processes.

A characteristic of all software being considered here is that it is checkpointable and migratable by DMTCP. This characteristic is described by a *CkptableEntity* (checkpointable entity) class that represent a checkpointable Process or Application. *CkptableProcess* and *CkptableApplication* represents the corresponding checkpointable entities in term of classes in the ontology. Figure 2 illustrates this hierarchy. It also provides a higher-level abstraction, the *MigrationEnabledEntity* class, which represents an entity that can be migrated through a specific mechanism.

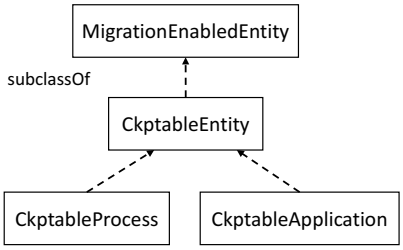

Fig. 2. Migration Enabled Entity classes

2) ***Ontology properties****:* The previous classes are linked by a set of Object Properties or Data Properties defined in the ontology.

In this section, we consider two types of properties: "static properties" and "dynamic information properties". The *static properties* correspond to **descriptive information** provided by the system that will not change, such as the machine specifications, the devices to consider, and the policies. The *dynamic information properties* correspond to the information concerning the **current state** of the system. That information changes over time. But at any given moment, we will consider a specific state of the system, and we will use the dynamic information to infer the potential symptoms.

The following properties correspond to the **static properties** of the system.

The *System* class is the domain of the object property *hasSystemComponent*. It links the virtual concept of system to the physical components that it is composed of. This object property has two sub-properties: *hasMachine* and *hasDevice*. They represent the corresponding machines of the system and the devices of the system.

[3]https://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Device

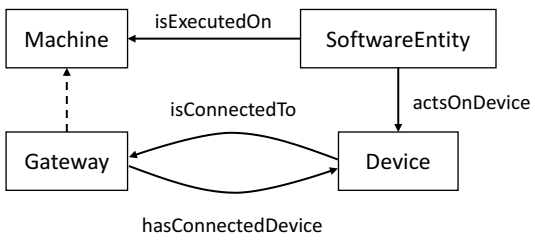

Fig. 3. Relation illustration between Machine, Device and Software classes. Also represents the possible constraint between a SoftwareEntity and a Device by the *actsOnDevice* object property.

The *System* class also has a relation with the *Policy* class. The *hasPolicy* class defines the policies to apply to the target system.

The RAM usage of the gateway can impact its behavior in the case of a RAM constraint. To account for this, we need to represent the maximum amount of RAM available on a machine. The *hasMaxRam* data property defines from a Machine this amount.

Next, to represent some application constraints, the ontology allows us to express some requirements between two concepts. For example, the requirement represented by the object property *actsOnDevice* between a SoftwareEntity and a Device means that the SoftwareEntity has to be executed on the Gateway where the device is located. If this requirement is not satisfied, the framework considers that there is a Symptom in the system that needs to be fixed. Figure 1 illustrates the System class and its links to the system components.

The next properties correspond to the **dynamic information properties** that change over time and that allow us to represent the state of the system.

The Devices are moving and are sending information to the Gateway. The *isConnectedTo* relation links a Device to the Gateway by which it sends and receives information. The inverse property is defined as *hasConnectedDevice* and it lists the Devices connected to a Gateway.

A relation is defined to represent the location of the software, i.e., where it is executed. This relation is *isExecutedOn*, and it links a SoftwareEntity to a Machine.

To represent the current state of the gateway, the *hasCurrentRamUsage* data property provides the amount of RAM used. Coupling this information with the current RAM policy and the maximum RAM usage, we can determine if a gateway is in a critical state.

B. Inferences and Application of Checkpointing Strategies

From the system description, the semantic reasoner is going to infer new knowledge. For this purpose, we have defined a set of classes representing the MAPE-K loop from autonomic computing [10], which receives and reacts to events from the monitored system. Based on the events received, symptoms of problems, and requests for change, the actions needed by the system for its correct operation are inferred and performed.

The *Symptom* class is an abstract representation of something outside the normal operation of the system. By itself, this class is not enough to define the symptom of the system. For this, we define sub-classes such as *LackOfRam* or *LackOfMemory*, which represent a lack of RAM and a lack of Memory in a Machine, respectively. The *hasSymptom* object property allows one to link a machine to a specific instance of a symptom. The inference of those symptoms is done using SWRL rules. The rule for the inference of the symptom LackOfRam is provided in Listing 1.

```
RamPolicy(?policy) ^
hasPolicy(?system, ?policy) ^
hasMachine(?system, ?machine) ^
hasMinAllowedRamLeft(?policy, ?minRam) ^
Gateway(?gw) ^
hasRamLeftPercent(?gw, ?ramLeft) ^
swrlb:lessThan(?ramLeft, ?minRam) ^
swrlx:createOWLThing(?symptom, ?gw, ?policy)

-> LackOfRam(?symptom) ^
hasSymptom(?gw, ?symptom)
```

Listing 1. Inference rule for LackOfRam symptom

For the symptom of the Software part, we defined a subclass named *WrongSoftwareLocation*, which represents a problem in which some Software does not satisfy its constraints. For instance, if a SoftwareEntity "requires" a Device and the Device is connected to a different gateway than the gateway where the software is executed, then this symptom has to be inferred. An object property is linked to this symptom, describing the potential location where the software *should* be. This property is named *potentiallyMigratesTo*. We cannot assert the migration request at this time because we have to check the resources of the target gateway. Listing 2 shows the SWRL rules that enable this inference.

```
hasMachine(?system, ?deviceGw) ^
hasMachine(?system, ?softwareGw) ^
actsOnDevice(?software, ?device) ^
isConnectedTo(?device, ?deviceGw) ^
isExecutedOn(?software, ?softwareGw) ^
differentFrom(?deviceGw, ?softwareGw) ^
swrlx:createOWLThing(?symptom, ?software,
                                  ?deviceGw)

-> WrongSoftwareLocation(?symptom) ^
potentiallyMigratesTo(?symptom, ?deviceGw) ^
concernsSoftware(?symptom, ?software)
```

Listing 2. Inference rule for WrongSoftwareLocation symptom. The abreviation "Gw" in the rule variables stands for "gateway".

Next, we define a *RequestForChange* (RFC). This concept represents a modification required for the correct operation of the system. It does not contain *how* to perform this change.

For the scenario presented in Section IV-A, we defined two RFCs. The first RFC is named *MigrateEntity*. It represents the required migration of the software to a new location. Two object properties can be linked to a MigrateEntity RFC. The first one concerns the software to migrate and it is named *hasMigrationRequest*. The second object property is *hasMigrationTarget*, and it determines the target machine.

The second RFC is *LightenMachine*. The request represents the need to "lighten" a machine by removing some software. This RFC has an object property that shows the target of the request: *targetsMachine*. This RFC is inferred when a WrongSoftwareLocation symptom is emitted to a gateway and this gateway has a critical state to address. Two examples are lack of RAM and lack of memory.

When the Symptoms and RFCs are inferred by the reasoner, our framework retrieves that information from knowledge base. Then, based on the inferences, the framework creates a set of migration plans to fix the system state.

At first, we need to consider the gateway to lighten of software, in order to create some space for the incoming software. For this, the framework looks at the currently running software on the gateway to be lightened and tries to find which software is not *strongly constrained* to be on the gateway; i.e., which software has no explicit constraint to remain on the current gateway. An example of software with no explicit constraint is software that does not require any device that is connected to the current gateway. When extracting this software from the initial gateway, we need to ensure that the new target gateway is not also a gateway that needs to be lightened. Another element to check is if there is a migration request on the new target gateway. We need to be sure there are enough resources for all the software to be migrated there.

After the lightened gateways are handled, we can next create the migration plan for the MigrateEntity RFCs. Since the management of resources was already been done in the previous part, it is not required to check again if there are enough resources for the migrations under consideration.

Require: Cs is $CheckpointableEntity$
Ensure: Cs is migrated to Tg
  $CurrentLocation \leftarrow$ currentLocation(Cs)
  $CsCkptImg \leftarrow$ createCkptImage(Cs, $CurrentLocation$)
  migrateCkptImage($CsCkptImg$, $CurrentLocation$, Tg)
  {migrating an image correspond to a file transfer}
  $Result \leftarrow$ restart(Cs, $CsCkptImg$, Tg)
  if $Result == SUCCESS$ **then**
    updateLocation(Cs, Tg)
  else {operation failed, report failure}
    reportMigrationError(Cs, Tg)
  end if

Fig. 4. Algorithm for the establishment of Migration Plan using DMTCP

The algorithm presented in Figure 4 shows the process to create a migration plan. This process is applied to each software required to be migrated. At first, we retrieve the knowledge base where the software is currently being executed. Then, we can proceed to the checkpoint of this software using the *checkpoint* operation of DMTCP. When the checkpoint image has been created, the software can be stopped on the current gateway. The checkpoint image file is then copied to the new target gateway. Finally, using DMTCP, the restart operation is performed on the image and the process is restarted on the new target gateway.

IV. The Semantic Model in Operation

This section presents the scenario used to instantiate the model presented in Section III. It begins by describing a scenario from the logistics domain. Then, two instances of the model to detect incoherences in the proposed scenario are presented. They are motivated by the two rationales for process migration described in the introduction. The first model considers issues of geographical location: placing incoming boxes of vaccines in a specific location along with their associated monitoring software. The second model considers the issue of limited resources, such as hardware limitations or energy consumption. the second model demonstrates the usage of the model rules by simulating a lack of RAM in the system.

A. Model instance: Box of Vaccines

For this scenario, the logistics of transport of goods is considered. This is an interesting domain because a transportation company has to handle many different goods and it is difficult to provide an associated traceability mechanism.

More precisely, this scenario considers the transportation of critical goods. In particular, the package of goods must be kept in a specific state for its safety. This is applied to the transport of a *box of vaccines* that must be kept at a specific range of temperature and humidity for its conservation.

For this purpose, the temperature, humidity and GPS sensors are attached to a box of vaccines that senses the environment of the box. Then, the data are sent via a low-powered and short-range wireless communication protocol such as Bluetooth Low Energy (BLE).

The software is required to be executed on a gateway connected to the box of vaccines sensors in order to receive the data. Then, to ensure the security of the communication, the data is encrypted and sent to the global business orchestrator. Finally, this orchestrator will check the values of the data and, depending on the business rules, will require intervention concerning the associated software process for the package of vaccines.

Moreover, the boxes of vaccines are separated by types. Each type of vaccine is stored in a specific warehouse for that type, and each box of vaccines contains only one type of vaccine.

The need to migrate the software along with the box of vaccines then arises. This can be a complex task when considering many boxes of vaccine and many gateways. To evaluate our approach, we consider two specific cases: 1) the box of vaccines moves to the warehouse of the same type, the gateway has enough *resources* to accept the software, and the migration is planned; and 2) the box of vaccines moves but the target gateway does not have enough RAM to accept the software, and so the system must find another plan to satisfy this constraint. Recall that we are considering gateways to be of low capacity and devices to be low-powered. Hence, any process swapping mechanism by the operating system is disabled, so as not to allow the gateways to become overloaded.

Figure 5 shows the architecture deployed in the first scenario. The box of vaccines with its sensors is displayed along with the wireless connection to the gateway. The software of the box is represented by the diamond labeled "Monitoring Software" in the gateway of the truck. The connection between the software and the devices is not established. So the software is not able to pursue its normal operation. The goal of our framework is to detect this type of issue by providing a semantic description of the software and then to use the checkpointing mechanism to fix it.

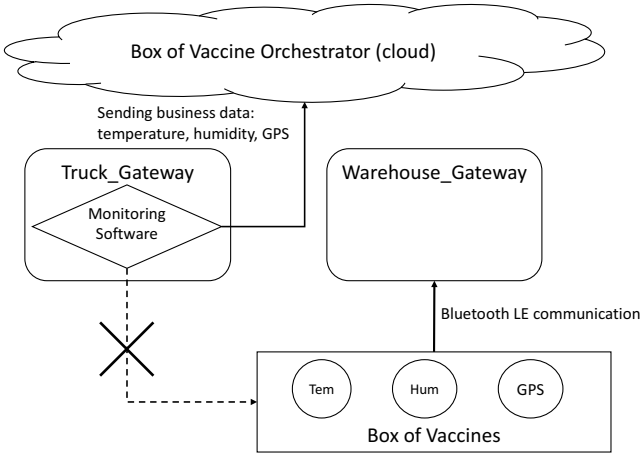

Fig. 5. Architecture of the Scenario presented in Section IV-A. It displays the Box of Vaccines connected to the Warehouse_Gateway after it has been moved to the warehouse. The Software monitoring the Box is still executed on the Truck, where it cannot receive information from the sensors.

In this scenario, we define another semantic class in the ontology that represents a box of vaccines. The class is called *BoxOfVaccines*. This class has an object property *hasSensor*, which links the box to its sensors. The sensors are instances of the Device class.

a) First case: For the first case, we consider a set of five warehouse gateways spread within a transport site for logistics. Each warehouse handled a specific type of vaccine. A varying number of trucks, containing a random number of boxes of vaccines, with each box chosen of random type, will arrive at the site. In this situation, one needs to dispatch the box of vaccines depending on its type and requires the software executing on the truck gateway to be migrated to the correct warehouse gateway.

Figure 6 shows the instances created in the knowledge base. Each different shape represents a different type of instance in the knowledge base. The labeled circles are instances of the Device class. The rectangular box is an instance of the BoxOfVaccines class. The diamond-shaped box is an instance of SoftwareEntity. The rectangles with rounded corners are instances of the Gateway class. The hexagon represents an instance of the VaccineType class.

In this scenario, we assume there is no RAM constraint and the $Warehouse\_Gateway$ has enough resources to accept the incoming software. For the purpose of this scenario, a new symptom is defined that represents the box of vaccines

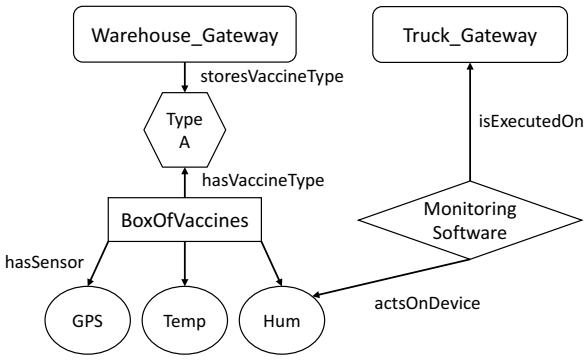

Fig. 6. Representation of the ontology instances used for the first scenario. Snapshot after the truck arrives at the site. The box of vaccines is linked to its type, which is the same as the Warehouse_Gateway. The Monitoring_Software is still connected to the Truck_Gateway and must be migrated to the warehouse. This inconsistency triggers a rule in the semantic reasoner.

that requires a placement into a warehouse. The SWRL rule shown in Listing 3 creates this symptom when a box of vaccines is still linked to a TruckGateway. The TruckGateway class is a subclass of the Gateway class (in semantic terms). Another rule, similar to the MigrateEntity RFC inference rule presented in Section III is also defined for the scenario. This last inference will then trigger the creation of a migration plan in the framework, followed by the actions performed using DMTCP. This demonstrates the use and the extensibility capability of our framework model in a basic example of migration.

```
TruckGateway(?truck) ^
transportsBoxOfVaccines(?truck, ?box) ^
hasVaccineType(?box, ?vaccineType) ^
storesVaccineType(?warehouse, ?vaccineType) ^
swrlx:makeOWLThing(?symptom, ?box, ?warehouse)

-> BoxRequiresPlacement(?symptom) ^
concernsBoxOfVaccines(?symptom, ?box) ^
targetsLocation(?symptom, ?warehouse)
```

Listing 3. Inference rule for BoxRequiresPlacement symptom

b) Second case: For this case, a third gateway is taken into account, called $Warehouse\_Gateway\_Bis$. Moreover, a RamPolicy is defined, which has a threshold of 80% of the max RAM Usage. The intended migration is the same: we want the software to be migrated onto the $Warehouse\_Gatewaay$. The same symptom as before, showing the wrong software location, is also created for this case. Let's consider 2048 MB to be the maximum RAM available on the gateway of the warehouse and 1750 MB to be the current RAM usage. This parameter triggers the RamPolicy rules and will create a *LackOfRam* symptom linked to the target gateway. Those symptoms, both targeting the same gateway, create the *LightenGateway* RFC.

The framework receiving the information will begin by finding software running on the target gateway that is not strongly constrained on this machine. It will then create a plan to migrate this software to the second gateway, which

corresponds to the warehouse. Now that the second gateway has sufficient resources to accept the software, a migration plan is created for the software corresponding to the box of vaccines.

V. Experimental Evaluation

This section presents a scalability study performed on this first scenario, geographical migration, as described in Section IV. First, the experimental environment is specified, followed by the scalability study.

A. Experimental Environment

To evaluate our work, a study of the scalability of the model has been carried out and is described in this section. Since the goal of this framework is its use in the IoT domain in general, we need to evaluate the size of the system that can be manage in a reasonable time.

To evaluate the scalability, we consider the first scenario, which was presented previously in Section IV-A. Multiple instances of this scenario are injected into the knowledge base to generate a complex instance of the model. Then the reasoner is executed on this knowledge base and it performs the inferences required for the analysis. The time taken to perform this analysis and determine the required migration is evaluated and help us determine the scalability of our method.

The experiment was carried out using an Ubuntu server (version 14.04) with an Intel(R) Xeon(R) CPU E5-2623 v3 (3.00 GHz) and 32 GB of RAM. The Java Virtual Machine (JVM) used is OpenJDK JVM version 1.8.0_111. To manipulate the RDF and OWL files, serialized as XML, representing the model and the data, OWLAPI version 4.2.7 has been used. The SWRLAPI version 2.0.0 has been used to create and manipulate the SWRL rules. Then, the Drools engine (version 6.5.0) is linked to apply the SWRL rules to the ontology via the SWRLAPI Drools bridge (version 2.0.0). The JFact reasoner (version 4.0.4) is then used to ensure the consistency of the ontology. Protégé version 5.1.0 has been used to create the model, but it is not used in the experiment.

B. Scalability study

Figure 7 shows the results of the experiment with the detailed values in table I. The x axis represents the number of box of vaccines in the knowledge base. Each truck gateway is linked to a random set of box of vaccines, between 10 and 20, with a random type. Each random uses a uniform distribution. The y axis shows the execution time to perform the inferences on the knowledge. The time is expressed in seconds and is displayed on a logarithmic scale. For each x value displayed, the experiment has been run 30 times, and the chart shows the average execution time with the solid black line, and the minimum and maximum of the series are displayed.

Table I shows that the execution time of our process depends greatly on the number of instances in the model. Starting with an average of 3.71 seconds for 1 instance in the model to about 63 seconds for 1430 instances. The first three numbers (11, 67 and 143 boxes of vaccines) show an execution time of about 4 seconds. The next number, with 289 is just 1 second longer than the 10 instances.

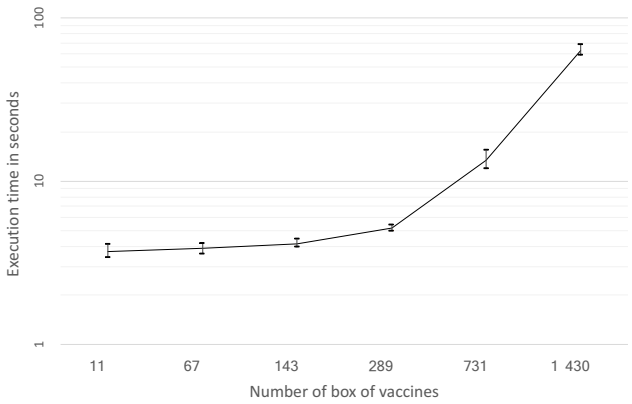

Fig. 7. Average-min-max chart displaying the dependence of execution time (in seconds, logarithmic scale) of the semantic reasoner on the number of box of vaccines in the Semantic Knowledge Base. For each number of boxes of vaccines, the experiment is performed 30 times. The high value represents the maximum of the series, the middle value represented by the line is the average, and the lowest value is the minimum. The exponentially increasing execution time here would be improved by coupling the semantic reasoner to a combinatory computation tool in future work.

TABLE I
This table displays the data shown in the figure 7. It represents the execution time of the semantic reasoner depending on the number of truck gateways present in the knowledge base. The values are expressed in seconds.

Truck gateways	Box of vaccines	Average	Minimum	Maximum
1	11	3.71	3.42	4.10
5	67	3.88	3.59	4.16
10	143	4.15	3.93	4.41
20	289	5.18	4.95	5.42
50	731	13.36	11.86	15.48
100	1 430	63.10	59.04	69.03

We note that creating the framework has a static cost of 3 seconds, due to the choice of technology. Specifically, the Java platform has some virtualization costs and the semantics technologies are not efficient in terms of execution time. In particular, the creation of the rule engine with the SWRLAPI and Drools takes about 3 seconds to be instantiated, independently of the number of box of vaccines.

The execution time increases greatly with the number of instances and it is especially high when considering more than 1000 boxes of vaccines. The time of 63 seconds for 1430 boxes of vaccines is excessive because our framework has the goal of providing a quick analysis of the system in order to perform changes in reasonable time. If the analysis takes too much time, the system may have changed again and the result of the analysis may be wrong. For this reason, future work will couple a semantic reasoner to a combinatory computation tool.

To conclude, the experiments have shown that the model is well suited for IoT applications in general. We have instantiated this model for a transportation logistics scenario and have

shown that the response time is sufficient for fewer than 1000 boxes of vaccines. In fact, the number of devices to manage in a real system is closer to hundreds of nodes rather than 1000, and there are several devices per gateways. So our approach provides reasonable performance in term of execution time for this scale. Moreover, a hundred truck will not arrive at the same time at a logistic site. Then, the reasoning can be performed several times when some trucks arrives with fewer instances and so, fewer execution time.

For a greater number of instances, another approach or distribution of the problem may have to be considered. Another kind of approach would be to couple the semantic reasoner to a combinatory engine that would help the computation of the rules.

C. Cost of Checkpointing in Process Migration

While the previous section showed that the semantic reasoner operates in reasonable time for typical systems of today, there remains the question of the additional overhead time in checkpointing and restarting a process in IoT. This question was answered in a previous paper [4]. Specifically, [4, Figure 3] shows that IoT processes with a memory footprint of up to 20 MB can generally be checkpointed in about 2 seconds and restarted in a similar time using the DMTCP [2] checkpointing package. Similarly, by using DMTCP-specific optimizations, the time could be reduced to 0.2 seconds. In future work, the overhead of the network will also be considered.

VI. Conclusion and Future Work

This paper proposes a novel framework based on semantic technology that represents current IoT systems, while providing distributed management of the running software elements through a checkpointing mechanism. This approach enables the management of large IoT system software without human interaction and with only a limited number of issues that must be passed on directly to the business software. The semantic approach provides for interoperability of our model with other models and also provides for extensibility of the ontology. Also, this work demonstrates that the execution time is reasonable for most IoT systems, but would need some improvement through a combinatory engine integrated with the semantic reasoner in order to handle a large numbers of elements beyond what is common in IoT systems today.

Another possible improvement to the current work is the consideration of policies. Sometimes the system may have to make a choice between several possibilities. This choice can be impacted by a policy. For instance, we can choose between several communication protocols, but we need to consume less energy, and so this energy policy will affect this choice.

In future work, we aim to provide a different architecture for this framework that enables scalability for a larger number of elements by creating a distributed environment with collaboration between the instances of the framework. In the context of IoT, a framework with a large deployment of gateways is envisioned, and the collaboration among them will be managed through a single gateway that connects all devices. This will provide a practical and scalable approach toward a distributed system.

Acknowledgments

This work has been supported by a “Chaire d’Attractivité” of the IDEX Program of the Université Fédérale de Toulouse Midi-Pyrénées under Grant 2014-345, and by the National Science Foundation under Grant ACI-1440788.

References

[1] R. Van Kranenburg, E. Anzelmo, A. Bassi, D. Caprio, S. Dodson, and M. Ratto, “The Internet of Things,” *A critique of ambient technology and the all-seeing network of RFID, Network Notebooks*, vol. 2, 2011.

[2] J. Ansel, K. Aryay, and G. Cooperman, “DMTCP transparent checkpointing for cluster computations and the desktop,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, may 2009, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5161063

[3] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, “Design and implementation for checkpointing of distributed resources using process-level virtualization,” in *IEEE Int. Conf. on Cluster Computing (Cluster’16)*. IEEE Press, 2016, pp. 402–412.

[4] F. Aïssaoui, G. Cooperman, T. Monteil, and S. Tazi, “Smart scene management for IoT-based constrained devices using checkpointing,” in *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*. IEEE, 2016, pp. 170–174.

[5] M. Serrano, P. Barnaghi, F. Carrez, P. Cousin, O. Vermesan, and P. Friess, “Internet of Things (IoT) semantic interoperability: Research challenges, best practices, recommendations and next steps,” IERC: European Research Cluster on the Internet of Things, Tech. Rep., 2015, http://www.internet-of-things-research.eu/pdf/IERC_Position_Paper_IoT_Semantic_Interoperability_Final.pdf.

[6] A. Sheth, C. Henson, and S. S. Sahoo, “Semantic sensor web,” *IEEE Internet Computing*, vol. 12, no. 4, pp. 78–83, jul 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4557983/

[7] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, “Semantics for the Internet of Things: early progress and back to the future,” *International Journal on Semantic Web and Information Systems*, vol. 8, no. 1, pp. 1–21, 2012.

[8] P. Desai, A. Sheth, and P. Anantharam, “Semantic gateway as a service architecture for IoT interoperability,” in *2015 IEEE International Conference on Mobile Services*. IEEE, jun 2015, pp. 313–319. [Online]. Available: http://ieeexplore.ieee.org/document/7226706/

[9] P. Horn, “Autonomic Computing: IBM’s perspective on the state of information technology,” IBM, Tech. Rep., 2001.

[10] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.