AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur : ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite de ce travail expose à des poursuites pénales.

Contact : portail-publi@ut-capitole.fr

LIENS

Code la Propriété Intellectuelle – Articles L. 122-4 et L. 335-1 à L. 335-10
Loi n° 92-597 du 1er juillet 1992, publiée au *Journal Officiel* du 2 juillet 1992
http://www.cfcopies.com/V2/leg/leg-droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Université Toulouse 1 Capitole (UT1 Capitole)

Présentée et soutenue le *28/11/2018* **par :**

François Aïssaoui

Autonomic Approach based on Semantics and Checkpointing for IoT System Management

---

JURY

J.-C. LAPAYRE	Professeur d'Université	Rapporteur
Didier DONSEZ	Professeur d'Université	Rapporteur
Dalila CHIADMI	Professeur d'Université	Examinatrice
Gene Cooperman	Professeur d'Université	Examinateur
Thierry MONTEIL	Professeur d'Université	Directeur de thèse
Saïd Tazi	Maître de Conférences	Co-directeur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Thierry MONTEIL et *Saïd TAZI*
Rapporteurs :
Didier DONSEZ et *Jean-Christophe LAPAYRE*

Thesis written by:
François Aïssaoui

Autonomic Approach based on Semantics and Checkpointing for IoT System Management

Thesis supervised by:

Thierry Monteil	INSA Toulouse, LAAS-CNRS, Toulouse France
Saïd Tazi	UT1 Capitole, LAAS-CNRS, Toulouse France
Gene Cooperman	College of Computer Science Northeaster University, Boston USA

This work has been funded by a "Chaire d'Attractivité" of the IDEX Program of the Université Féderale de Toulouse Midi-Pyrénées, Grant 2014-345.

Acknowledgments

Enfin de m'avoir permis de faire cette thèse, je tiens à remercier mes directeurs de thèse Thierry Monteil, Saïd Tazi ainsi que Gene Cooperman. Vos expertises, visions et conseils que vous m'avez donnés pendant cette thèse m'ont grandement aidé à venir à bout de ce projet.

Je remercie les rapporteurs Didier Donsez et Jean-Christophe Lapayre, ainsi que Dalila Chiadmi pour avoir accepté d'être examinatrice et présidente de mon jury.

Je tiens aussi à remercier mes collègues de bureau, Nicolas et Guillaume, qui avons vécu cette expérience de thèse ensemble. Leur soutien tout au long de cette thèse ainsi que le partage de nos différentes expériences m'a beaucoup aidé. Je tiens à remercier Chloé, aussi présente dans notre bureau, qui m'a donné envie de continuer l'escalade. Je remercie Karima et Santi pour avoir supporté nos différentes conversations avec Nicolas et Guillaume quand nous parlions de nos projets communs.

Je remercie aussi mes amis proches, avec qui j'ai pu partager mes inquiétudes durant cette période et qui ont toujours été là pour me motiver à continuer.

Enfin, je remercie ma famille et surtout mes parents pour avoir été là tout au long de cette aventure. Leur soutien sans égal m'a permis de continuer d'aller de l'avant et de donner le meilleur de moi-même.

Contents

Introduction

Context

Machine-to-Machine (M2M) is a large domain discussing the communication capabilities between machines in general. Decades ago, this communication was non-trivial and not as standardized as it is nowadays. It went through several steps of innovation to be adapted to the needs of the users and so it led to the creation of Internet, a large computer network.

The Internet itself has evolved to integrate more types of computer like entities over the years, such as the smart phones, connected cameras, etc. More and more devices have become connected to this network and have taken part in the communications. We now talk about the **Internet of Things (IoT)**, where a *thing* corresponds to any object capable of sending or receiving data over the Internet.

However, the applications of IoT are not only for home appliance and connected objects [Atzori 2010]. It is a large domain that comprises factories, transportation, agriculture or e-health. The goal is to connect the *things* that enable the interaction with the users present at different locations and to provide an **ambient intelligence**. For this reason, those domains have names using the "smart" prefix, such as smart cities or smart factories.

The devices enable the gathering of data from the environment of the user, or the interaction with the environment or directly with the user. The data is collected by application software entities present on the Internet. Depending on the environment state and the user policies, they are able to react by using actuator devices or by sending information to the users.

However, not all devices are able to directly handle the connection to the Internet to send their data. To perform such an operation, the device would need to implement the standard protocols to connect to the network. This requires a minimum of processing power to do so. This is not possible in all cases, since we want autonomous devices that run on a battery for months or years. To counterbalance this, specific device protocols are created by industrial organizations that enable the creation of low-cost devices with a specific communication channel. This approach requires the usage of *gateways* that receive the data from the devices. A gateway corresponds to a machine closely located to the devices. Those machines are usually low-powered and do not have a lot of processing power to reduce their initial and maintenance costs. The gateways provide interface between the devices (using their specific communication protocol) and the Internet (where the high-level applications using the data of the device are located).

For this purpose, software processes are executed on the gateways. They enable the communication with the devices and, depending on the application, may have other features. For instance, such a piece of software may be in charge of the security of the data handled.

The main concern with such software is that they may experience a failure for different reasons. For instance, the gateway may be run on top of solar panel-powered batteries and may lack energy during some days, or the dynamicity and mobility of the devices may interfere with the correct operation of the software entities.

With the growing number of connected devices, Gartner envisioned 20 billion devices by 2020[1]. The number of gateways deployed will also grow. Furthermore, the amount of software to handle this will increase, and with the diversity of possible applications, the task to manage those entities become difficult.

We distinguish the need to provide a **management** system for this software with several considerations. The **distributed** aspect of the gateway has to be taken into account. In fact, the gateways are spread over different locations and may not be on the same network. The **heterogeneity** of the systems and the applications has to be represented in the management. Depending on the application needs, the management operations have to be different. This heterogeneity also brings another issues with is the connectivity of the devices and gateways. The **interoperability** of the approach is necessary in term of connectivity and representation of the data.

Moreover, a mechanism is required to change the state of the running software. For this purpose, we propose the use of a migration mechanism such as checkpointing. It enables the creation of checkpoint image files of a running program that may be restarted on another machine.

Contributions and outline of the thesis

The chapters of this thesis are structured as follow:

Chapter 1: Scientific Context and Background

In this chapter, several domains covered by this thesis are presented. General concepts and protocols of IoT are studied. Secondly, some technologies and techniques that enables process migration between machines are presented, with an emphasis on the checkpointing mechanism used in the thesis. Then, semantic web technologies are described and an overview of their usage in IoT is given. Finally, the autonomic computing approach, used to structure the work of this thesis, is described.

Chapter 2: The IoT System: Monitoring and Migration mechanism

In Chapter 2, the first contribution of this thesis is presented, discussing the required elements to interact with an IoT software infrastructure. For this purpose, the operation of two autonomic components are described. The monitoring component, implemented with device management technologies is presented.

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

Then, the execution of the actions on the system is described with the execution component. It uses migration techniques based on checkpointing in order to manage the software entities of the gateways. The usage of checkpointing mechanism is evaluated in this section.

The proposed Scene mechanism is described and evaluated in this chapter. This mechanism provides a novel approach to design softwares using large data on memory-constrained gateways using the checkpointing mechanism.

Chapter 3: Knowledge Base for IoT Infrastructure Management

Chapter 3 illustrates the knowledge base of the autonomic manager. It presents the semantic model proposed in this thesis that aims at representing the IoT software infrastructure. An emphasis on the representation of the software entities is given in the ontology.

Moreover, for the integration of the autonomic approach, the description of the autonomic data such as the symptoms and requests for change is discussed.

Chapter 4: Semantic Analyzer for Symptom and RFC inference

Chapter 4 presents the semantic inference system in place to infer issues from the system. It corresponds to the autonomic analyzer component. Based on SWRL rules, this chapter demonstrates how the symptoms and the requests for change are inferred.

This inference system is evaluated with an application to a logistics scenario.

Chapter 5: Software processes optimization in an IoT system

Chapter 5 describes the planner component in charge of creating a plan of action when an issue is detected. This planning is performed by using a genetic algorithm in order to find an optimized solution in finite time. In fact, this approach leads to a combinatorial set of possibilities when we consider the placement of the software entities on the gateways. This contribution is evaluated and compared through a brute-force approach.

Finally, a conclusion to this thesis is provided, along with a discussion of future work.

Chapter 1

Scientific Context and Background

Contents

The core contribution of this thesis is the high-level management of IoT software processes. This chapter describes the context of the different domains handled in the document.

First, we discuss IoT diversity and current challenges. Then, in order to handle the software processes, we describe the existing runtime software platforms that enables the migration of the processes. Moreover, definitions of technologies associated with web semantic technologies are provided. Its general usage for IoT is discussed along with the set of known ontologies published for the IoT. Finally, the autonomic computing paradigm is described and will be used to structure the thesis work.

1.1 The Internet of Things

The IoT is a large domain with many possible applications [Atzori 2010]. This section provides an overview of some aspects of IoT in term of the protocols used, the device technologies and the literature challenges. Moreover, a discussion around the device management technologies is provided.

1.1.1 IoT Technologies

Many technologies and protocols must be considered in IoT [Al-Fuqaha 2015]. A description of the main protocols used among the machines and with some devices are given. A brief overview of the device technologies is also provided.

1.1.1.1 Protocols used in the IoT

Some well-known and standardized protocols are at stake for IoT. Depending on the application context and its needs, the choice of protocol may differ. Moreover, several protocols can be used at different layers of the applications.

Hypertext Transfer Protocol (HTTP)[1] : is a well-known standard protocol used in the Web. HTTP is a connection-oriented protocol, built on top of Transmission Control Protocol (TCP), which enables the exchange of information between a client and a server. However, it is a verbose protocol, including a lot of meta-information in the packets. Moreover, the expressiveness of the HTTP operations allows the implementation of Representational State Transfer (REST) architecture. The protocol provides a set of operations such as GET, POST, PUT or DELETE to perform, respectively, a retrieval of resource, a creation, an update and deletion.

Constrained Application Protocol (CoAP)[2] : is a protocol built on top of the User Datagram Protocol (UDP) and is not connection-oriented. It is based on a subset of HTTP operations. In comparison to the previous protocol, it uses an optimized set of headers, with a binary representation, to lighten the communications. The application payload is also limited in size and pushes the users to use optimized data formats. It uses a client-server architecture.

Message Queuing Telemetry Transport (MQTT)[3] : uses a different approach than the other protocols. It is a *publish/subscribe* mechanism. MQTT *clients* send messages to a *broker*. The messages are sent to a named topic by the client. Other clients are able to subscribe to topics on a broker and will receive the messages sent to the requested topics. A comparison of the performances between HTTP and MQTT in an IoT scenario has been done by [Yokotani 2016] and is in favor of MQTT. The authors also propose an enhancement of the protocol to reduce the network consumption.

1.1.1.2 Device technologies and constraints

There is a large diversity of devices and machines deployed for IoT. This large deployment leads to the usage of constrained devices in order to reduce the cost of this large deployment. Those constraints come from different aspects such as the energy management of the entities, or the restricted processing power of the machines.

In term of energy, a lot of devices rely on internal battery or external power sources. *Energy harvesting* is an emerging approach [Kamalinejad 2015] that uses the power received from an electromagnetic signal to perform its own communication. Mechanical power is also used to harvest enough energy to perform a radio communication [Gorlatova 2014]. *EnOcean*[4] technology is based on this mechanical approach and proposes devices that do not have any battery but also send their data using radio communications.

There are also more standard IoT devices using other kind of wireless communications. *Z-Wave*[5] is another short-range communication technology and aims at providing devices for home applications.

Bluetooth Low Energy (BLE) is an extension of the Bluetooth technology with a reduced functionality in order to reduce the power consumption of the communication.

Some long-range technologies are being developed. They are comparable to cellular networks, but with lower energy consumption and bandwidth. *SigFox*[6] is a company providing the eponymous long-range network. It provides a radio technology that enables the communication of some devices via the Internet. The messages sent over this communication are stored in a SigFox cloud and can be retrieved by other applications through Web Services.

LoRa is another long-range communication technology with a specification handled by the LoRa Alliance[7]. This is a more open eco-system compared to SigFox, where everyone can deploy their own LoRa network. This architecture is the same as SigFox in that the devices can send messages to gateways that are connected to the Internet. The final destination of the messages is not directly a cloud provider and may depend on the application.

1.1.2 Architectures and Challenges

Several contributions from the literature discuss the commonly deployed and architectures and the challenges that are present in this context.

The classic IoT architecture is viewed in three layers as described in the literature:

- The top layer is composed of powerful nodes, named as “weakly constrained nodes” in [Zanella 2014b]. This layer is also assimilated into the Cloud by

[4] `https://www.enocean.com`
[5] `https://www.z-wave.com`
[6] `https://www.sigfox.com`
[7] `https://www.lora-alliance.org/`

others [Liu 2015, Szilagyi 2016]. It corresponds to a set of machine that have no constraints based on the energy consumption and have large computation capabilities. Those nodes are the hosts for large programs concerning data analytics and often serve as an interface between the IoT sub-systems and the high-level applications. Usually, the storage capabilities of this layer are extremely large.

- The middle layer corresponds to a mix of several kind of entities. We find the machines, called gateways in the literature by [Compton 2009, Desai 2015], whose resources are constrained. In terms of costs, their computation capabilities are quite low and so do not use a lot of energy. However, they are deployed close to the devices and implement a particular device protocol that enables their communication with the higher layer.

- The bottom layer corresponds to the end devices that have a limited power source and almost no computation capabilities. These constrained nodes are the sensors (as presented in [Compton 2009]) or actuators, which will enable interaction with the real world. They often implement device-specific technologies, described in the previous section, and interact with the gateway in order to send their data to high-level applications.

Other contributions points out specific cases. [Yashiro 2013] proposes an IoT architecture integrating the already deployed embedded systems in the device networks. This is done through a specific framework developed by the author and is not standardized. An interesting aspect of this work is the usage of protocols with low network impact, such as CoAP.

[Ma 2011] provides objectives and challenges for the IoT. The author highlights challenges such as large data exchange among heterogeneous elements, and the integration of uncertain information for the decision or the adaption of the dynamic system environment.

[Zanella 2014a] discusses the usage and challenges of IoT in smart city scenarios. The authors show the diversity of domains present in this context, such as waste management, traffic congestion, city energy consumption, etc. A standardized approach is suggested with the proposal of web services technologies supported by IETF, ETSI or W3C. This standard approach is also supported on a larger scale by [Gyrard 2014].

The vision also includes virtual objects able to interact with and affect the real world. This creates a significant number of challenges [Whitmore 2015]. In most cases, these objects have strong constraints in terms of energy, communication and/or processing [Chen 2014].

1.1.3 Device Management

Device Management corresponds to the management of the physical entities in an information infrastructure. The management of deployed devices is a common issue in the IoT ([Perumal 2015], [Zhu 2010] or [Kim 2015]).

The Open Mobile Alliance (OMA)[8] organization proposed several standard protocols for Device Management. A first standard, OMA DM[9], has been proposed and the first version has been finalized in June 2008.

Afterwards, they proposed another device management protocol Lightweight M2M (LWM2M) more oriented for embedded equipment due to its "lightweight" nature. The first standardized version of the protocol was released in February 2017.

This new protocol is resource-oriented with a REST architecture and based on CoAP to lighten the networks. Several security mechanisms can be applied in order to secure the communication channels and authenticate the entities.

This device management protocol enables the representation of the state of the devices in a resource format. It allows a management entity to retrieve different information of the deployed devices such as the memory usage, the battery level, etc. It also contains some descriptive information of the nature of the device handled. Moreover, it is possible to perform actions on the managed devices. For instance, standard operations are *reboot* or *firmware update*.

Additionally, it is possible to define its own model in the LWM2M resource tree. This enables the extension of the protocol to other types of data that can transit by the protocol.

Several implementations of the standard exist in the open-source ecosystem. In the Eclipse foundation, there are Eclipse Leshan and Eclipse Wakaama.

Eclipse Leshan[10] is a client and server implementation of LWM2M in Java. Eclipse Wakaama[11] is a light client written in C. Both projects propose extensions of the software in order to integrate this technology in new devices.

We observe a large diversity in term of possible protocols and device technologies used in the IoT. Each type of application will rely on multiple technologies and will have different constraints. Some literature contributions focus on data handling and its representation in the IoT domain. Moreover, the autonomous approach is suggested with the usage standards for the communication protocols. Those decisions will help to handle the extensibility and heterogeneity of the domain.

The device management technologies are useful in order to monitor an IoT system. Some of them, such as LWM2M, are based on CoAP, a protocol suggested for use in IoT to reduce the network overload. This technology, and in particular LWM2M will be used in this thesis to monitor the IoT software infrastructure.

1.2 Runtime software platforms

This thesis discusses how to manage the running software processes on IoT infrastructure. For this purpose, a set of platforms that enable the interaction with

[8] `https://www.omaspecworks.org`

[9] List of specifications: `http://openmobilealliance.org/wp/index.html`

[10] `https://www.eclipse.org/leshan/`

[11] `https://www.eclipse.org/wakaama/`

the execution of the software entities is described.

We evaluated several existing technologies that enable an important operation for the IoT, "software migration". It corresponds to the possibility of stopping the execution of a software entity on a machine and pursuing it on another [Milojičić 2000].

1.2.1 Checkpointing Mechanism

1.2.1.1 Definition

The checkpointing mechanism consists mainly of two operations, **checkpoint** and **restart**, on an **operating system process**. Figure 1.1 shows an example of both operations.

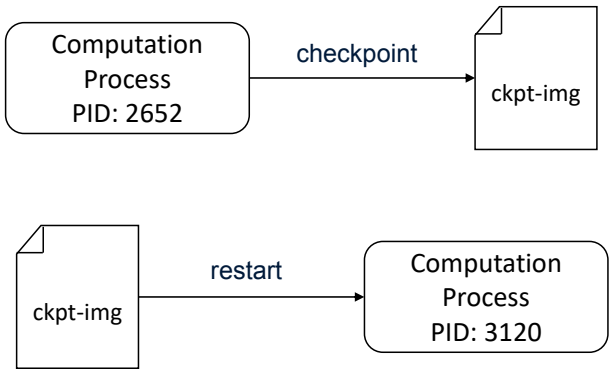


Figure 1.1 – Checkpoint and restart operations from checkpointing mechanism

The *checkpoint* operation corresponds to the saving of a running process into a file. It aims at saving the whole state of the process with its memory, open files, created threads, etc. in a *checkpoint file*.

Thus, this file can be used to perform the *restart* operation — the reverse operation that recreates the running process from the checkpoint file. The process is in a state semantically equivalent to the state at the time of checkpoint.

Checkpointing has a long history in HPC [Litzkow 1997, Hargrove 2006, Cao 2014, Cappello 2014]. In 2012, a cluster of ARM CPUs was tested with respect to checkpointing as a basis for power-efficient High Performance Computing (HPC) [Keville 2012]. This used the more powerful ARM Cortex-A9 CPU, whereas the current Raspberry Pi Model B uses the less powerful ARM Cortex-A7. In those earlier experiments, checkpoint times from 3.4 to 138 seconds were observed on various NAS parallel benchmarks for MPI — a standard test suite for parallel applications. In comparison, the experiments of this work apply checkpointing only to a single process.

1.2.1.2 DMTCP as an implementation of the checkpointing mechanism

In this work, a checkpointing mechanism is used, implemented by Distributed Multi-Threaded CheckPointing (DMTCP) [Ansel 2009]. DMTCP provides a *transparent* checkpointing mechanism that provides for checkpoint/restart without any modification of the original application code or operating system. However, it is important to note that the checkpointing mechanism is not only for HPC, as presented before. It is used in several other cases such as:

- Fault tolerance
- Process migration
- Debugging, by creating a checkpoint right before the bug
- Fast startup, using an already initialized checkpoint

DMTCP uses a system based on a coordinator. Each coordinator corresponds to a single computation with one or more system processes that may be checkpointed. To checkpoint the computation, a checkpoint command must be sent to the coordinator, which will notify the managed processes. The DMTCP library will perform the checkpoint operation on each processes and save its state. The inverse operation can be performed on the same or a different machine, using a new coordinator to restart the processes.

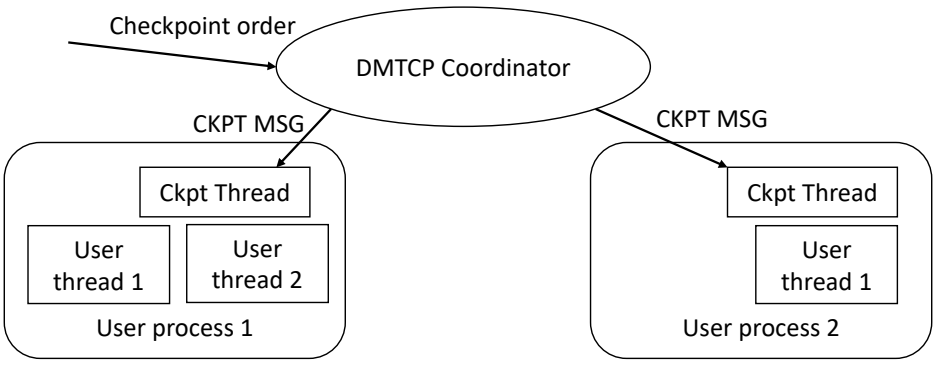


Figure 1.2 – Example of DMTCP coordinator with two processes

An example of the use of the DMTCP coordinator is given in Figure 1.2. It shows the DMTCP coordinator at the top that is managing two processes. The first process has two threads executing its computation while the other one only has one thread. Moreover, we note that the checkpointing thread is present in both processes, although it is not active when the user thread are computing. When the coordinator receives a checkpoint request, it sends a checkpoint message to the DMTCP thread. This will trigger the checkpoint operation, which will freeze the user threads in their current computation and save the process in a file.

It is important to note that during the checkpoint operation, the user threads are **frozen**. This means, if a new message arrives from a network connection for instance, it will be handled only after the checkpoint operation is complete.

However, a problem exists in such a mechanism when the process is interacting with other external entities: the external entities may change between the time of checkpoint and restart. For instance, the PID of the process itself is not the same during a checkpoint and after a restart. Moreover, the communication with other processes may be interrupted in the middle of a message during a checkpoint request. A plugin mechanism is proposed by DMTCP to solve this problem.

DMTCP 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 [Arya 2016]. A plugin in DMTCP can have multiple functions. It can act as a wrapper for system functions. For instance, when the process uses a system call such as `getpid`, a virtual value is given to the process and not the system one. The plugin is in charge of keeping track of the PIDs and providing wrapper functions that interpose on any system calls invoking PIDs, in order to hide any alteration in PIDs from the process.

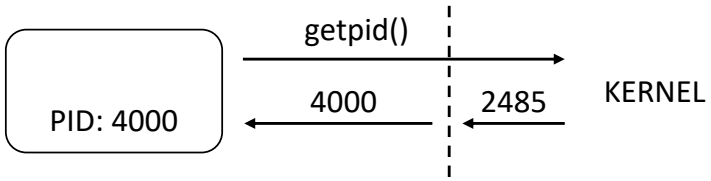


Figure 1.3 – Example of DMTCP plugin with `getpid` system call

Figure 1.3 shows an example with the `getpid` call. The kernel returns the value but it is intercepted by the DMTCP plugin. A translation table keeps track of the PIDs and translate it when necessary. The virtual value is sent to the user process on the left. This value is still correct after the restart of the process, and the plugin is in charge of keeping this virtualization. The translation is also performed on the other system calls that uses the PID, such as `kill` for instance.

1.2.1.3 Existing checkpointing strategies

The checkpointing mechanism manages the software entities. On top of this mechanism, a decision needs to be made on **when** to checkpoint.

In [Salehi 2016], the authors propose a two-state checkpoint policy (TsCp) in order to reduce checkpoint overhead for real-time applications. The policy separates the state of the application into two cases: *fault-free* and *faulty execution*. In fault-free execution, non-uniform intervals of checkpointing are used to reduce the number of unnecessary checkpoints. In the second state, checkpoints are performed in a uniform manner. The aim is to start the application in the non-uniform intervals until the system fails. At this moment, the system is restarted at the last valid

checkpoint, and the second state starts with uniform checkpoint intervals to prevent future failures.

[Ghit 2017] proposes three types of checkpoint policies: 1) greedy checkpoint policy; 2) size-based checkpoint policy; and 3) resource-aware checkpoint policy. The greedy policy performs a checkpoint at each steps of the application. This also uses a *budget* representation of the possible checkpoints. This policy tries to limit the effort of re-computing a result previously obtained by the program. The second policy is denoted "size-based". This policy aims at checkpointing straggler tasks that are slower than the main computation. It eliminates the possible time lost when a long-running task fails but is not directly linked to the main computation. The last policy proposed is a resource-aware checkpointing policy. It evaluates the cost of the checkpoint as compared to the cost of the computation that has been performed. If the cost of the checkpoint is lower, than it is performed. Otherwise, the computation continues until a failure occurs, since it would be faster to use the previous checkpoint and re-compute the lost result.

In [Naksinehaboon 2008], the authors propose an incremental approach to the checkpointing mechanism. Since their contribution concerns checkpointing over the network, its usage would grow when a large checkpoint is performed. In their approach, a full checkpoint is performed first, and then incremental checkpoints are done over the network. On important steps, full checkpoints are also created. When a failure occurs, the last stable state is constructed with the last full checkpoint and the incremental ones are also added.

1.2.2 Cloud: Virtual Machine Migration

In cloud computing, the migration of virtual machines is a commonly used mechanism [Zhang 2010]. It is also called "live migration". It has evolved from process migration techniques [Osman 2002].

Several implementations of live migration have been built, as, for example, in Xen in [Clark 2005]. Other virtualization software such as VMWare has also implemented this live migration[12].

Regarding the technical context of the VM migrations, a cloud hypervisor is required. For the virtual machines, a virtualization is required and the operating system of the machines is migrated alongside. This requires machines with enough computational power in order to perform this virtualization and is difficult to apply to IoT gateways.

It is also important to note that the state of a running application in the virtual machine can be saved. The snapshot mechanism of virtual machines enables the serialization of the state of the whole machine. This snapshot can be used to migrate the machine along with the software application, when needed.

[12] `https://www.vmware.com/products/esxi-and-esx.html`

1.2.3 Docker

Docker[13] is a virtualization software package based on containers on an operating system level virtualization. The program allows the user to download images of containers from an on-line repository.

Docker uses a layer system for the representation of the images. This means that an image has a set of layers that defines it. In order to reduce the number of layers a user has to download, a copy-on-write optimization is performed on the underlying layer. This means, if an image is based on the same layer as another image, it will not make a copy of it when the base image is not changed.

However, Docker by itself does not directly allow the process migration. Several other contributions that support Docker enable this process management of images on multiple machines. Consul[14] proposes an orchestration of the services deployed on Docker. Kubernetes[15] is a Google proposal for the orchestration of microservices deployed on distributed machines. It facilitates the migration of the services, but requires the components to be completely stateless.

An important note on the Docker environment is that microservices are required to be stateless. This assumption is used a lot in the management of Docker services that facilitate the start and stop of the images.

The authors of [Ismail 2015] recommend the use of Docker for the Fog computing domain. The results of that work found that Docker has a fast deployment, good performances, and a small footprint on the target machines.

1.2.4 OSGi

Open Services Gateway initiative (OSGi) is a standard proposed by the OSGi Alliance[16] that defines a service platform for Java programs. It defines another layer of abstraction on the top of the Java Virtual Machine (JVM) that provides standard services such as logging, communication (e.g., HTTP), etc.

The architecture proposes the use of *Bundle*. A bundle is a software entity that contains a set of Java packages that can be exported and provides a set of services. A bundle also depends on other bundles based on the packages it imports and the services it uses.

There are several runtime implementations of OSGi, such as Eclipse Equinox[17], which supports the Eclipse IDEs, or Knopflerfish[18].

A strong point of OSGi is the possibility of installing or uninstalling bundles at runtime. This enables the management of an OSGi instance depending on the current needs of the applications and the context. However, no mechanism is present in the standard to migrate the state of the bundle between two instances. It is

[13] `https://www.docker.com/`
[14] `https://www.consul.io/`
[15] `https://www.aquasec.com/wiki/display/containers/Kubernetes+Architecture+101`
[16] `https://www.osgi.org/`
[17] `http://www.eclipse.org/equinox/`
[18] `https://www.knopflerfish.org/`

possible to uninstall a bundle from one instance and install it in another. But a supplementary mechanism is needed to migrate the bundle data. In the literature, OSGi has been used by the authors of [Pan 2011] to enable task migration.

Moreover, OSGi has a dedicated working group for the IoT[19]. They aim at defining base services and an OSGi architecture for the integration of IoT devices.

1.2.5 Comparison of the approaches

Table 1.1 shows a comparison of the different existing solutions for process migration. It describes several parameters to consider when using a migration technique:

- Requires resources to perform the migration
- Integration to already existing solution
- Migration cost
- Virtualization level required by the technology
- State of restart if an initialization is needed

[19] https://www.osgi.org/about-us/working-groups/internet-of-things/

	Checkpointing (DMTCP)	**Cloud VM Migration**	**Docker**	**OSGi**
Required resources	Low footprint on the memory [Ansel 2009].	Requires machines that handle the VM virtualization.	Low footprint [Ismail 2015].	Java Virtual Machine and OSGi runtime.
Integration to already existing solutions	Transparent integration and adaptation possible with the plugin mechanism	Easy integration with a VM that possesses the runtime environment of the application.	Need the separation in micro-services and stateless to facilitate the migration.	Needs to adapt the architecture of the Solution to bundles and force the usage of Java or JNI.
Migration cost	Process frozen during the checkpointing. Needs to transfer the checkpoint file over the network. The size of the file depends on the size of the memory used by the process.	Creation of a snapshot that contains the whole VM is required to be performed and sent over the network.	Needs to deploy the images on the target machine.	Needs to download and install the bundles requires to deploy the services on the destination machine.
Virtualization level	Virtualization of system resources when system calls are used by the processes.	Hardware virtualization level.	Operating system virtualization level	JVM and OSGi runtime.
State of restart	The process is already ready after the restart has been performed.	When using a snapshot, the application is already initialized. When restart from a VM, standard initialization of the process is required.	The initialization has to be done after a restart of a Docker image but may be fast if it is stateless.	The bundle has to be started after it has been installed. Depending on the business application, it may be instant or take some time.

Table 1.1 – Comparison of technologies enabling process migration.

1.3 Semantic Web and Technologies

Semantic technologies are used in this thesis in order to represent the system to manage with its entities in a high level model. This section provides a definition of the base concepts of those technologies. Moreover, we study its usage in the IoT.

In a survey of multiple context modeling and reasoning techniques, [Bettini 2010] gives several purposes to semantic technologies: 1) the expressiveness of the language enables the description of complex context data; 2) provides formal representation of the knowledge that is sharable to other entities; and 3) the existence of reasoning tools that checks the consistency of the knowledge base and also generate new knowledge based on the complex description of the system.

1.3.1 Definition

In [Berners-Lee 2001], the authors proposed the **Semantic Web** as an extension of the regular web but with information understandable by the humans and the machines. It aims at providing context information that is understandable by a program when it retrieves a web resource. Semantic computing [Sheu 2010] is an emerging and rapidly evolving interdisciplinary field that originated from artificial intelligence. It consists of applying models and standardized technology describing the semantics of the linked objects to enable interactions and interoperability between different components.

1.3.1.1 Ontology

The standard recommendations are provided by the World Wide Web Consortium (W3C)[20], an organism also responsible for the web standards. In order to understand the context information, a vocabulary is required. The use of *ontologies* has been proposed by [Gruber 1991] and then standardized by the W3C. An ontology defines a set of concepts of a domain. Also, it defines properties that are used to characterize the concepts. Two types of properties are possible: *object properties* that links two instances of a concept ; or *data properties* that links an instance of a concept to a value, e.g., an integer.

Moreover, it is possible to define one ontology based on another one, thus extending the previous ontology. This is commonly used when an ontology defines high-level concepts, one needs to refine the ontology to the needs of the considered application.

1.3.1.2 Representation and serialization

The Linked Data principle is used in the Semantic Web. This defines the use of a Uniform Resource Identifier (URI) to identify all the resources. Those resources are connected together via semantic properties and creates a knowledge graph, also called a knowledge base. To represent this graph, the W3C proposes the use of

[20] `https://www.w3.org/standards/semanticweb/ontology`

Ressource Description Framework (RDF)[21] as a model. The latter is based on triples composed of a *subject*, a *property* and an *object*. The subject corresponds to the entity that is concerned by the triple. This property defines the type of the relation. The object is the entity in relation with the subject by the relation. The subject and the property are entities identified by a URI. The object can be another individual of the graph identified by a URI, in the case of an object property. Otherwise, the object can be a literal in the case of a data property. Its type may vary depending on the property: integer, string, etc.

RDF only enables the description of resources with a graph representation. However, it does not contain any semantics. For this purpose, RDFS[22] enables the description of taxonomic and non-taxonomic relationships between classes. Ontologies defined only with RDFS are considered lightweight ontologies.

To express more complex concepts and relations, the Web Ontology Language (OWL) formalism has been defined[23]. It enables the definition of complex classes and properties by using an extended set of logical axioms.

In order to exchange those information between the users and the machines, a serialization format is required. The most common one is in Extensible Markup Language (XML), a widely used format to exchange data on the web. Other more compact formats exist such as Turtle[24]. This format is more oriented in the representation of the triples, thus making it less verbose than the XML representation.

1.3.1.3 Query possibilities with SPARQL

To retrieve information from the knowledge graph, a querying mechanism is needed. For this purpose, the W3C proposes SPARQL Protocol and RDF Query Language (SPARQL)[25]. This defines a query language to retrieve information in the knowledge graph by using a graph pattern-matching mechanism. SPARQL has been extended to enable the insertion, modification and deletion of the triples in the knowledge base.

1.3.2 Semantic Inference Engines

In order to infer new knowledge, semantic inference engine are used. They base their reasoning on the logical axioms from RDFS and OWL expressed in the ontology. Additionally, they can determine if there is an inconsistency in the knowledge graph. This kind of engine is called a **reasoner**.

Moreover, the RDFS and OWL logical axioms are not the only mechanisms to enable the creation of new knowledge. Other mechanisms, based on rules, enable the creation of new relations in the knowledge base on certain condition.

[21] `https://www.w3.org/RDF/`
[22] `http://www.w3.org/2000/01/rdf-schema`
[23] `https://www.w3.org/OWL/`
[24] `https://www.w3.org/TR/turtle/`
[25] `https://www.w3.org/TR/sparql11-overview/`

1.3.2.1 SWRL

Semantic Web Rule Language (SWRL)[26], presented in [Horrocks 2004], is a rule system that allows the creation of new relations. They are compounded of two parts. First, a precondition has to be evaluated by a reasoner in order to find out a matching pattern of resources. In its matches, the postconditions are applied on the knowledge graph.

The SWRL rules can be represented in RDF, thus making it possible to embed them in an ontology alongside the description of the concepts. Moreover, since SPARQL enables the insertion and modification of the knowledge graph, it is possible to change the rule dynamically.

An example of SWRL rule is shown in Listing 1.1.

```
hasParent(?x1,?x2) ^ hasBrother(?x2,?x3) -> hasUncle(?x1,?x3)
```

Listing 1.1 – Example of SWRL rules

This rule aims at representing the *hasUncle* object property between *x1* and *x3*, where *x2* is the parent of *x1*. The parent relation is represented by the *hasParent* object property and the brother relationship by the *hasBrother* object property. When a graph pattern matches the left part of the rule, meaning we find a *x1* that has a parent *x2*, and the same second individual has a brother *x3*, then the last relation *hasUncle* between *x1* and *x3* is inferred.

It is important to note that SWRL has not been standardized (stayed at the submission status) by the W3C even if it has been supported by several reasoners and used by the community. For instance, Pellet[27] allows the inference of SWRL rules alongside the RDFS and OWL logical axioms. [Hashmi 2014] presents an usage of SWRL rules for the automation of negotiation in web services. The rules are defined in order to find out the issues from the input data of the services, and also to match the policies for the agreements in the negotiation.

1.3.2.2 SPIN and SHACL

SPARQL Inference Notion (SPIN) is another submission to the W3C. It aims at representing rules using SPARQL to define constraints to represent in the ontology. It has quickly transited to Shapes Constraint Language (SHACL), a recommendation of the W3C[28]. SHACL aims at representing constraints in the knowledge base by the definition of "shapes" and infer new knowledge with a rule-based system[29].

1.3.3 Usage of Semantics in the IoT

In [Hachem 2011], the authors propose the usage of semantic technologies to provide interoperability and flexibility in IoT systems since they are highly dynamic and heterogeneous. Their approach uses three levels of representation in the

[26] https://www.w3.org/Submission/SWRL/
[27] https://www.w3.org/2001/sw/wiki/Pellet
[28] https://www.w3.org/TR/shacl/
[29] https://www.w3.org/TR/shacl-af/#rules

ontologies. They discuss the use of physics and mathematics domain ontologies that represent the physicals concepts by their relations. Another ontology provides estimation models for their system. The last ontology focuses on devices.

Moreover, the use of semantic technologies is a recommended best practice for IoT [Serrano 2015] for the interoperability approach.

Different types of data can be formalized by semantic models. Sheth et al. provide in [Sheth 2008] 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. [Barnaghi 2012] 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. [Desai 2015]. 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 facilitates the creation of new services. They also describe the gateways as the primary 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.

[De Paola 2014] proposes an ontology-based autonomic system for ambient intelligence scenarios. The author discusses the needs for IoT applications to enable the self-management of the system and the high-level representation of the system. In this work, the proposed ontology focuses on ambient intelligence scenarios and the interaction with the users.

1.3.4 Known ontologies in the IoT

The literature also proposes ontologies that aim at representing different aspects around the devices of the IoT and their data.

SSN / SOSA: Semantic Sensor Network (SSN)[30] is an ontology dedicated to the representation of IoT sensors and their observations. It has been enhanced by Sensor, Observation, Sample and Actuator (SOSA)[31], another W3C initiative, to integrate other concepts such as the actuation.

SAREF: The ontology Smart Appliance REFerence (SAREF)[32] aims at managing the energy and the services of smart homes. It is supported by the European Commission and by European Telecommunications Standards Institute (ETSI). Afterwards, the ontology has been enhanced in order to be applied to other IoT domains.[Daniele 2016]

oneM2M Base Ontology: The oneM2M standard also proposes its own vocabulary in the Technical Specification TS-0012. It defines high level concepts

[30] `http://purl.oclc.org/NET/ssnx/ssn`
[31] http://www.w3.org/ns/sosa/
[32] `https://w3id.org/saref`

in the IoT such as Thing or Service, and aims at being aligned with other domain specific ontologies.

IoT-O: based on the work of [Alaya 2015b], the ontology aims at orchestrating several concepts for the description of IoT data [Seydoux 2016b]. It is based on several ontologies such as SSN for the sensing devices description, SAN for the actuation part, MSM and WSMO for the service descriptions, etc. It allows the representation of the device deployment on an IoT scenario and is based on other known ontologies.

In conclusion regarding the semantic technologies, we observe a large usage of those technologies in order to represent the data of the IoT devices. Since IoT is a heterogeneous system and handle a large panoply of data types, a high level representation is required. Some efforts are made in order to represent the IoT system from the point of view of the devices and the data they produce.

However, there is a lack of representation of the IoT machine and software infrastructure. This corresponds to the machines where the device are connected and where the software processes are executed. A representation of them is required in order to analyze the issues concerning the infrastructure.

1.4 Autonomic Computing

1.4.1 Definition

In a manifesto of IBM from 2001 [Horn 2001], 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. [Kephart 2003]. They propose an approach based on living organism that is able to manage a system and also manage itself.

In [Kephart 2003], the authors propose features that an autonomic system has to implement:

Seft-configuration: this feature represents the capability of the system to reconfigure itself depending on the evolution of the monitored system.

Self-optimization: the management system needs to optimize itself .

Self-healing: when the system has issues, the management system is able to detect and repair them based on high-level policies.

Self-protection: the system is able to protect itself from malicious attacks and error that would disable its operation.

Moreover, an architecture is proposed to implement an autonomic computing system. Figure 1.4 shows this architecture, called MAPE-K.

The framework is composed of the following components:

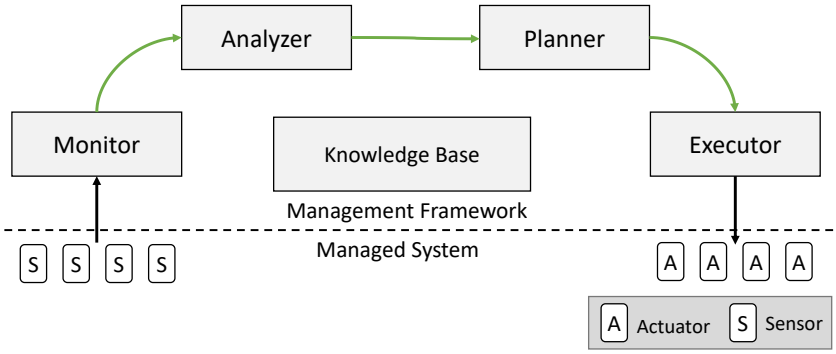


Figure 1.4 – MAPE-K loop for Autonomic Computing

Sensors: they represent entities gathering metrics and sending them to the management system.

Actuators: *(also called effectors)* these components are in charge of changing the managed system when the autonomic framework detects an issues. They perform basic actions on the managed system, following the orders of the management framework.

Knowledge Base: this component stores the information of the monitored system. It contains a description of the elements of the system, along with their current state. It also possesses the high level policies to apply when a decision has to be taken in the system.

Monitor: this component aggregates the metrics received from the sensors. It has to update the Knowledge Base of the framework when a change is detected.

Analyzer: the Analyzer is in charge of finding out the problems in the system. Based on the description of the entities in the system and their current state retrieved by the Monitor. It will infer the *Symptoms*. With this information, it will send a Request For Change (RFC), a high-level representation of the parameters to change in the system, to the Planner.

Planner: this component bases its reasoning on the RFC received from the Analyzer. It aims to find a plan of actions to perform on the system in order to apply the given changes. The choices made by the planner are influenced by the high-level policies defined in the Knowledge Base.

Executor: this receives the plan of actions inferred by the Planner. It uses this plan to determine the correct actuators to use in the system in order to perform the actions.

1.4.2 Usage in the IoT

In an IoT context, a large number of entities are distributed and need to be monitored and controlled when a change appears in the system. With the growing

complexity of such a system, the management task is difficult.

However, some contributions aim at providing a framework of autonomic computing. In [Alaya 2015a], the authors provide a framework designed to manage M2M systems, called FRAMESELF. It is structured around the MAPE-K loop and implements the several autonomic computing data such as event, symptoms, RFC and actions. For the inference system, a logical model of the policies with a rule based system is used. The inference engine is DROOLS [Bali 2009]. This enables the expression of high-level policies with logical rules make possible to coincide several elements to perform an inference. However, this approach does not allow for the representation of semantic entities in the knowledge base.

The usage of autonomic computing in order to manage the devices of the IoT has been suggested in [Aïssaoui 2016b]. Moreover, the combination of the autonomic computing with standard protocol and semantic representation allows for interoperability in terms of communication and interpretation of data. This is an important property for IoT, due to the strongly heterogeneous and dynamic environment.

The application of those principle has been carried out in [Seydoux 2016a], in order to manage the devices from a connected apartment.

Conclusion

This chapter provided an overview of the different concept used in this thesis.

First, the IoT domain, along with its constraints, has been described. This thesis aims at handling this diversity and representing the complexity of the application in order to manage the software processes present on the machines. The device management technologies have been presented and are used to monitor the IoT software infrastructure.

Then, several runtime software platforms that enable the migration of software processes have been presented. We studied multiple technologies that require different levels of virtualization and need different levels of effort in order to adapt to already existing solutions. In this work, we focus mainly on the checkpointing mechanism, which provides a transparent and efficient migration mechanism. based on two available operations, *checkpoint* and *restart*. It provides the flexibility to manage the software processes depending on their needs.

However, the mechanism of migration of the software entities is not enough by itself. An orchestration of this mechanism is required in order to determine *when* to migrate an entity and *where*. For this purpose, we chose to use the web semantic technologies. This provides reasoning capabilities, based on a high-level description of the IoT entities with a common vocabulary, and an ontology. The inferences are done by reasoners based on the definition of the entities in the vocabularies and the rules present in the knowledge base.

Finally, to structure this approach, the autonomic computing paradigm has been discussed. It was shown in the literature to be an architectural approach well-suited for IoT. It enables a clear separation of components, and it specifies a set of properties that need to be implemented in order to manage a system.

The next chapter presents the first contribution of the thesis. It covers the interface between the autonomic manager and the managed system. Two components are described, the *Monitor* and the *Executor*. The Monitor uses device management technologies to retrieve metrics from the software infrastructure, and the Executor uses the checkpointing mechanism to perform the required modifications to the system.

Chapter 2

The IoT System: Monitoring and Migration mechanism

Contents

In the previous chapter, we expressed the need to manage the software processes executed in an IoT environment. The entities present in this IoT system, the software processes, the devices, the machines, etc., are changing over time. The resource consumption of the machines are increasing due to the new deployed processes, or the devices physically moving and changing their network connections. This dynamic environment requires to be *monitored* and *repaired* when an issue is detected. In order to perform this management of this environment, we choose the autonomic computing approach.

The autonomic computing with the MAPE-K loop has been presented in the previous chapter, Section 1.4. This approach structures the **management framework** in several components. The latter has to interact with the **managed system**

in order to retrieve information or metrics about its "health". Moreover, the managing system needs to be able to perform actions on the software infrastructure in order to fix the issues.

For this purpose, the *Monitor* component of the MAPE-K loop is in charge to perform the gathering of metrics on the system. The *Executor* component acts on the system when an action has to be performed. As presented in the context, this thesis focuses on the migration of IoT processes. This chapter presents the *Monitor* and the *Executor* components of the autonomic computing approach. (See Figure 2.1)

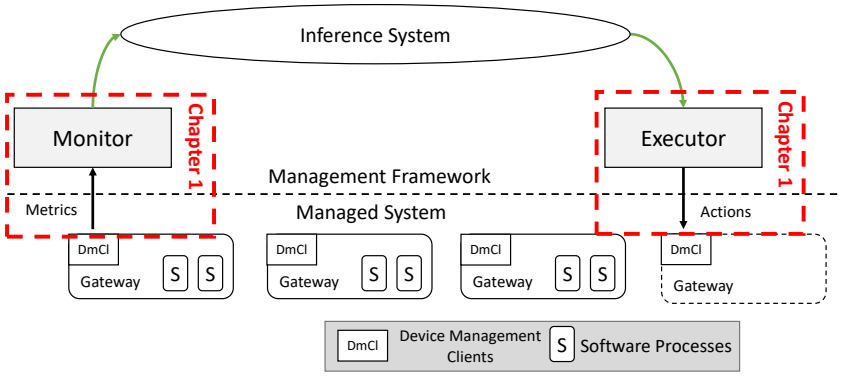


Figure 2.1 – Framework Architecture — Monitor & Executor

The monitoring is presented with two aspects. First, the technologies used to perform the monitoring in the system are briefly presented. For this purpose, Device Management technologies are used. They are industry-approved and standard technologies that allows the gathering of information on distributed machines. Then, a discussion on the monitored parameters is given in order to determine which information is relevant for IoT software process management.

Subsequently, the executor component is presented. Its role is to perform the software migration of the IoT software processes. Such migration is performed via a checkpointing mechanism. However, as shown in the previous chapter, this mechanism has not yet been used in an IoT environment. Thus, an adaptation of the checkpointing mechanism is discussed. The IoT has resource constrained machines that may lack RAM when used by several processes, or processes with large amount of data to handle. To solve this issue, we propose a new software architecture approach, based on *Scenes* and checkpointing mechanism, in order to enable the handling of large data. This scene mechanism is modeled with an ontology that enables the extension for new applications.

As a final point, an evaluation of the checkpointing improvement in an IoT context is conducted. This aims at showing that the checkpoint/restart mechanism is faster than standard initialization and restart of a process. Moreover, the usage of scene mechanism with DMTCP as a checkpointing software is evaluated. Finally, a specific mechanism used with the scenes that enables the passing of information

between them is evaluated in order to determine its overhead.

2.1 IoT Software Process Monitoring

In order to manage the software infrastructure, one need is to collect enough data to determine the potential issues. For this purpose, the contribution proposed in this thesis is based on an approach based on standard, and widespread in the industry, as discussed in previous work [Aïssaoui 2016b]. However, the standard defines *how* to transfer the data and not to interpret it.

First, this section describes the Device Management standard used for this approach. Then, the list of monitored parameters that will be used by other autonomic components to infer the issues of the infrastructure are given.

2.1.1 Device Management technologies

To collect data from the system, a set of sensors is needed. *Device Management* technologies allow to collect this kind of management data. In our framework, we use LWM2M as a Device Management protocol. It has been described in Section 1.1.3.

The standard uses a client-server architecture. This allows the integration of the server along side the Monitor component of the Autonomic loop. The clients are executed on the different entities that needs to be managed. The main targets of those clients are the IoT gateways. They are the entities executing the software processes this thesis aims at managing.

The sensors are then deployed on the monitored gateway using LWM2M clients. Several implementations are available as described in Section 1.1.3 and can be chosen depending of the type of equipment that is required to be managed. Figure 2.1 displays the Device Management clients in the gateways.

2.1.2 Monitored parameters

With Device Management technologies, the connection between the gateways and the Monitor component is established. We are now looking at which information has to transit through this mechanism.

A distinction of two kind of data can be made. The first one are the *available and used resources* by the system. This can correspond to physical properties of the monitored gateway such as the battery level, or numerical properties such as RAM usage. We provide a non-exhaustive list of those parameters.

Battery level: The *Battery level* of the gateway, if it is autonomous in energy, is an interesting parameter for several scenarios. Actually, knowing that the battery is almost depleted leads to ensure that the services present of the named gateway are migrated. This migration can have priorities depending on the kind of service available and can be expressed in a global migration policy.

Processor/RAM/Disk usage: Several parameters such as *Processor usage*, *RAM Usage* or *Disk usage* are parameters of the current gateways that vary in time. Those parameters have to be used to determine the *"health"* of the machine. If too many resources are used on a single gateway, maybe distributing the running processes on other gateways may enhance their operation.

Network Connectivity: The *Network Connectivity* is another parameter to monitor on the gateways. In fact, depending on the kind of current network connection, a process may have to be stopped or halted. For instance, a heavy-consuming network bandwidth process have to be stopped on cellular connection, but may be restored when Wi-Fi is available.

Network usage: From the previous parameter flows the *Network usage* in term of bandwidth usage. Even on a Wi-Fi network, if a gateway is having a high network usage compared to the others, a new distribution of the network consuming processes has to be performed.

Connected Devices: The Monitor has to retrieve the gateways where the devices are connected. This information is used for the processes that requires a specific device or type of device. In the IoT, device can have mobility with wireless connection to the gateways, e.g., bluetooth, and swap from a gateway to another when the first one is out of bound.

Executed Processes: The currently executed processes on each gateway is a crucial information. It represents the current distribution of the process in the software infrastructure. This is the base information that we want to act on when an issue is detected in the system. The software migration will have a direct impact on this information.

The second kind of data that needs to be monitored are the application specific information. They vary in nature and purpose depending on the goal of the software process. Since Device Management technologies allow the declaration of custom object models, those application information can be embedded in those objects.

2.1.3 Data Interpretation

We listed a set of parameters that defines the state of the software infrastructure. Some more parameters may be added in the future, depending on the new technologies. The advantage of using Device Management technologies for the transport of those information is that the data models are extensible. It means that the addition of new parameters to send to the Monitor component can be done by extending already existing models or create a new one.

Now that the data is present in the Monitor component, it has to be interpreted in order to be analyzed later on. This step requires a model to do so. In Autonomic Computing, the model is described into the knowledge base. As described

in the general introduction, this thesis proposes semantic representation of the IoT software infrastructure using Semantic Web technologies and ontologies.

This leads to the goal of the Monitor to semantically describe the provided information with a vocabulary. This vocabulary, stored in an ontology, will be described in the next chapter.

2.2 Process Migration: Execution and Enhancement

The Executor component has to perform the actions on the system to fix it. Those actions are based on migration mechanisms such as checkpointing.

In order to use the Checkpointing mechanism in the IoT context, some optimization is advised. The goal behind this optimization is to reduce the process "freeze" time when a checkpoint is created. During this freeze time, the process is not responding. Depending on the type of application, this can cause timeout issues with network connections, the lost or depreciation of data in other cases. Moreover, the restart time of a process is also important when it contains a lot of data stored in RAM. Indeed, since the restart operation has to remap the memory of the checkpointed process into the RAM, the larger the memory, the longer the restart operation will be.

In order to solve the large RAM usage issues by using the checkpointing mechanism, the thesis proposes a new software architecture approach based on a scene representation. A *Scene* corresponds to a partial view of the data used by the application which depend on the application domain.

First, this section presents the checkpoint mechanism optimization used for this adaptation to the IoT domain. Then, the scene approach coupled with the checkpointing mechanism is set in an IoT software process.

2.2.1 Checkpointing optimization

The DMTCP software is used to perform the checkpointing operations. During the compilation or the execution of the DMTCP operations, several parameters can be used to configure the checkpoint and the restart operation. Two possible options are described in this section: *Forked Checkpointing* and *Fast Restart*.

DMTCP also supports options for two optimizations that enhance the speed of checkpoint and restart. The first is "*Forked Checkpointing*". DMTCP forks a child process, which executes the checkpoint. This takes advantage of the well-known operating system support for copy-on-write between the parent and child processes. The parent process continues to execute without blocking, while the child process writes memory and other state into the checkpoint image file. This allows to bypass the freeze time during the checkpointing operation.

The second optimization option is "*Fast Restart*", based on the Linux *mmap* system call. The mmap call maps the checkpoint image file to RAM, but the data is not actually copied to RAM until the virtual memory subsystem pages it in.

Thus, execution begins early after restart, paging in only the actively used pages, and without waiting for all of the checkpoint image file to be loaded.

Some other parameters are used during the real deployment in the experimentation. Those options are described in Section 2.4.1 but have less impact on the efficiency of the checkpointing.

2.2.2 Scene in action with the Checkpointing mechanism

In principle, the use of scenes within a large, global hierarchy can be implemented as a single large process. However, typical IoT-based embedded systems are restricted to small RAM without any virtual memory. For this reason, we represent each scene of the global hierarchy as a separate operating system process. Only one process (the current scene) runs at a time. We demonstrate that switching between scenes can be made efficient through the use of checkpointing. The original scene (with all of its internal state) is checkpointed, and a new scene is restarted from a previous checkpoint image.

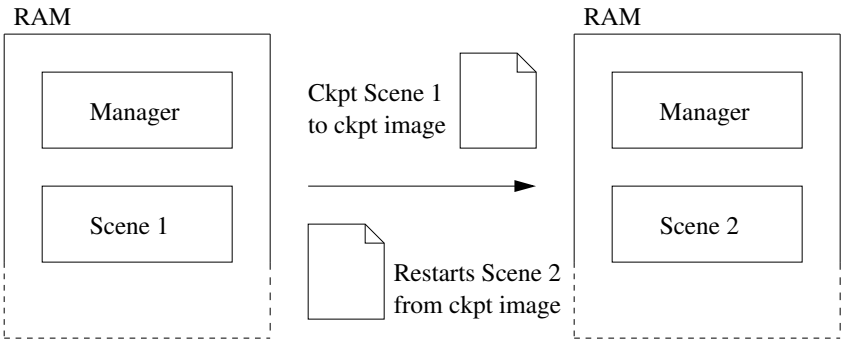


Figure 2.2 – Proposal of new architecture for Scene Management. Each rectangle represents a process.

Figure 2.2 illustrates the proposed architecture. The data to be handled is split into multiples scenes, which contain information. Each scene is represented as an individual process. A *Scene Manager* is used to checkpoint and restart the process that represents a scene.

This enhancement provides a simpler way for the end programmer to design the architecture and the data handling of its program. It also enables the possibility to handle large amount of data in an optimized way with the checkpointing mechanism used to swap the scenes.

However, swap from a scene to another is a complex task. It requires the consideration of multiple parameters that may have different descriptions, formalism, etc. A high level representation of such information is needed.

In order to determine when to swap from a scene to another, we propose a semantic model of the scene mechanism. This allows the representation of the scenes with a vocabulary and the possibility to extend its behavior with application specific rules. This model is provided in the next section.

2.3 Software Process Architecture based on Scenes

This section presents the global architecture with semantics and scenes that is used as a testbed in this work. It defines the "scene" approach and describes the models used to handle the scene management through policies on when to checkpoint and when to restart. Then the rules used to change from one scene to another are presented and some examples of application rule will be given.

This leads to a specific need when a process is swapping from a scene to another: transfer some information from the previous scene to the new one. For this purpose, we introduce the concept of information sharing between scenes.

2.3.1 Semantic Models Used

A Scene is defined as a partial view of the application context. Several scenes are created according to the needs of the application. In order to have a fast swapping between the Scene, it is required to have multiple pre-saved scenes. However, only one Scene is loaded at any given time.

Figure 2.3 shows a semantic model with some example relations in the semantic class Scene.

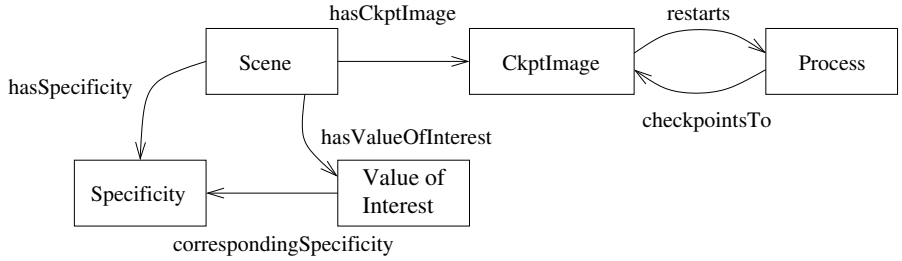


Figure 2.3 – Overview of Scene representation and link between Scenes and Checkpointing Mechanism

Scene: This class represents the considered Scene. For each Scene that the application needs, an instance of this class will be created.

Specificity: In order to determine how to switch between scenes, one needs to provide the specificity of a scene. This class represents this specificity. Regarding the evolution of the process, the current context may change and we will have to change scene regarding the new specificity. The Specificity class is linked with the Scene class with a relation *hasSpecificity* from the Scene to the Specificity class. This relation represents, for example, a location specificity, or a time-of-day specificity (e.g., day and night).

Value of Interest: The class *ValueOfInterest* is used to characterize the values that are important for other Scenes and need to be monitored. They are linked to the *Specificity* class by a *correspondingSpecificity* property for that

class. This property links a value to a specific type of scene. Following the evolution of this parameter, we can infer that a scene change is required. This associates with the Scene specific characteristics that enable the reasoner to choose the best target scene to switch to.

CkptImage: The Scene is linked to a *CkptImage* class by the *hasCkptImage* object property. This allows the reasoner to identify the available checkpoint images for a Scene.

Process: The Process class represents a process in term of operating system. The checkpoint image is then linked to a process. Two types of relations are possible: 1) the process has been checkpointed into a CkptImage (shown via *checkpointsTo*); or 2) a CkptImage is used to restart a process (a *restarts* relation is created between the CkptImage and the restarted process).

2.3.2 Scene Hierarchy

Since each scene represents a partial view of the global state, a classification of the scenes is needed. A hierarchy is used in which each scene (except for the root scene) has a parent scene. This representation allows the definition of more specific scenes depending on the application requirements.

A "child" scene inherits parameters and rules from its parent scene and adds additional, more specialized information. That information might be, for example, information about the type of location (e.g., what city, or what neighborhood in a vehicular context) and is considered to be *static* in the sense that it does not change over time. In contrast, each specialized scene also has *dynamic* information. An example is the specific road conditions, which might depend on road work in progress.

A hierarchical classification of this type allows one to create lightweight scenes, each of which has more specialized information than the parent in the hierarchy.

Figure 2.4 gives an example of scene hierarchy. This example is based on a smart application for a vehicle. The root scene is located at the top of the representation. The first specificity in place is the location. Two child scenes with different value for the location are depicted in Figure 2.4. Then, depending on the scene, we have more specification for the same concept, i.e., the location, or different specificity. For instance, in the bottom layer two scenes are defined when the street is crowded or not. This allows the definition of different behaviors for the application. Moreover, the piece of code handling the "crowded" situation may not be shared with the "not crowded" situation. Thus, the scene architecture provides a easy way to encapsulate the behavior of the application.

2.3.3 Shared Information

The checkpointing mechanism allows the state of a running process to be serialized into a file. But some information and knowledge acquired by the first scene

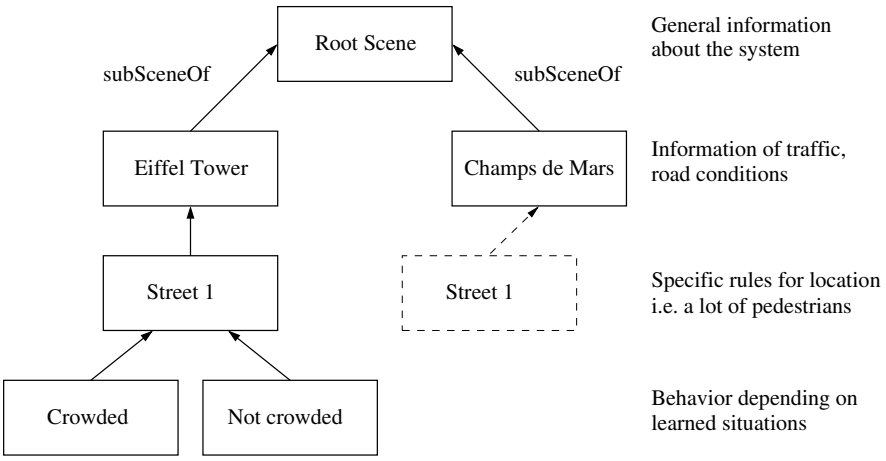


Figure 2.4 – Example of a possible scene hierarchy. The root scene is at the top with general information, then child scenes are underneath with more specific application information.

must then be passed to the second scene.

As described in Section 2.3.2, the scenes are derived from a hierarchical classification. This classification allows the system to provide relevant information to the next scene. For instance, the whole system shares information from the car sensors and geographical location. This general information is stored and defined by the root scene of the system, which will be shared by all sub-scenes.

In considering a sub-scene in the scene hierarchy, note that it is not necessary to pass all of that sub-scene information to other scenes. For instance, if a sub-scene with a specialization of the geographical location, e.g., the scene of the city of Paris, is unloaded and another sub-scene with another location is loaded, the system will not pass information of specific road conditions of the city of Paris, since it is not relevant for the new scene.

On the other hand, if a scene of Toulouse is being replaced by a child scene that is specialized for night driving, road condition must be applied to the new scene, since the vehicle is not changing its location.

With such a mechanism, the system is able to share information between different scenes, according to the relevance of the data for the next scene. Such a mechanism allows one to reduce the amount of information handled by the system and the reasoner. This mechanism is implemented using the DMTCP plugins discussed in Section 1.2.1.

The information to share can be retrieved from the model using the property *hasValueOfInterest* of the Scene. This relation is shown in Figure 2.3. The storage system is based on *name:value* pairs. This means that when a process is about to be checkpointed, a set of data is stored with this format. Those information are stored locally.

In the next section, several evaluations are provided. The scene mechanism is

compared to a standard loading in Section 2.4.3. The overhead of the *name:value* mechanism is discussed in Section 2.4.5.

2.4 Experimental Evaluation

In this section we evaluate the scene system presented in Section 2.3 and the cost of the checkpointing mechanism in an IoT context. Here, we discuss the additional time needed when a checkpoint is invoked, and the time needed to restart a scene from a checkpoint image file. Then we compare this restart time to a traditional approach, which consists of dynamically reading the data files. After, we discuss the runtime overhead introduced when the process is executed under the control of DMTCP, as opposed to executing the process natively. Finally, the overhead of passing information between scenes with the *name:value* pairs is discussed.

2.4.1 Experimental Environment

These experiments use a *Raspberry Pi 2 Model B* with 1 GB of RAM. In these experiments, we emphasize the limited RAM of a constrained embedded system by restricting ourselves to a more limited 256 MB of RAM. This was also the RAM provided with the earlier Pi 1 Model A+. The files containing the scenes and the images files for the experiment are stored in the file system of the SD card of the Raspberry Pi. The *Raspbian-4.4.11–v7+* operating system is used. The program testbed requires a semantic reasoner. Our testbed is based on Java, using the Oracle JVM version 1.8.0_65 (Standard Edition). Version 3.0.0 of the *Apache Jena* library is used to load the ontology and the data files. The loading, checkpointing and restarting times are evaluated for a process consisting of the JVM along with the ancillary libraries and the files that are loaded. Finally, for the checkpointing software, we use DMTCP version 3.0.0, compiled with GCC version 4.9.2.

By default DMTCP uses gzip to compress the checkpoint image of a process. This compression makes the checkpoint process and restart process slower since the checkpoint image has to be compressed and uncompressed for both operations. In the experiment, we use the "no-gzip" option of DMTCP to skip the compression of the checkpoint image in order to speed up the checkpoint and restart operations.

During the checkpoint operation, DMTCP has to save the state of open files used by the target process being checkpointed. If the current process has "write permission" for the open files, it is possible that an open file has been modified by the process but the modified data has not yet been written out to the file system. To avoid any error in the checkpointed process, DMTCP makes a copy of the potentially modified files for which the process has "write permission". In our experiments, one such open file is the jar file of the Java program. But making a copy of this file can slow down the checkpoint operation. Since the jar file is never modified, we removed the write permission that is assigned by default by the operating system. This optimization speeds up the checkpoint-restart operation.

2.4.2 Checkpoint and Restart

As a first case for evaluation, we analyze the checkpoint and restart times on the Raspberry Pi. The size of the input files is varied in order to find the relation between the size of the files and the checkpoint-restart time.

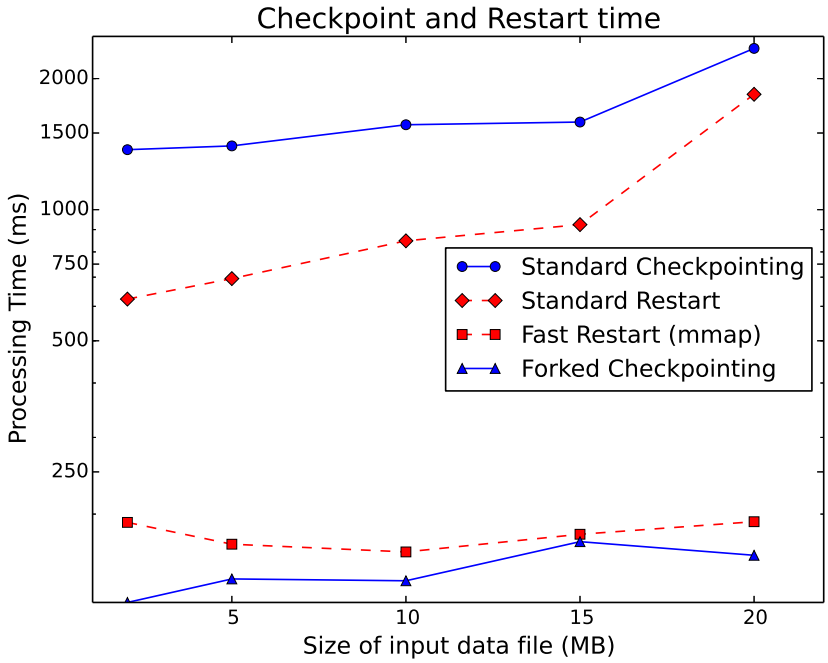


Figure 2.5 – Checkpoint and restart time

Two sets of experiments are discussed. First the standard checkpointing and restart mechanism is used. In Figure 2.5, the two lines at the top of the graph show the time needed for a standard checkpoint and restart the Java program with the Jena library and the data files loaded. The "standard" times refer to the case when the DMTCP optimizations of forked checkpoint and mmap-based fast restart are not used. The times vary as the size of the scene file is varied. Note that a logarithmic y-axis is used for the checkpoint and restart times. It is assumed that the operating system must execute in RAM along with the application in a real-time system. Recall that the goal of these experiments is to simulate a low-cost embedded device, with only 256 MB of RAM.

The time to checkpoint and restart grows slightly when the size of the scene-related data increases. This is expected, since DMTCP must map the process image to the checkpoint file (or reverse for restart operation) and this operation is slower if there is more data to save to a file (or to load from a file). The unoptimized checkpointing times of Figure 2.5 vary from 1.5 s to about 2 s. This is reasonable for energy-constrained devices such as the Raspberry Pi, but it can be improved to be more responsive. Similarly, the unoptimized restart times vary from about 600 ms to 1.5 s.

In order to further improve responsiveness, a second experiment (also presented in Figure 2.5) shows the impact of using the two DMTCP optimizations discussed in Section 2.2: forked checkpointing and mmap-based fast restart. These optimizations improve the checkpoint/restart times (and hence the responsiveness) by a further factor of ten.

The first line from the bottom of Figure 2.5 shows the time for the Forked Checkpointing. This Forked Checkpointing operation is about 5 to 10 times faster than the Standard Checkpointing and allows the running process to be available more time – since the Checkpointing operation freezes all threads to avoid any error in the memory of the process. The checkpoint operation is done by the child process and the time to make this operation is equivalent to the Standard Checkpointing. The times are reduced to about 150 ms to 200 ms for the running process. Since the times are close to the minimum quantum of times given to the thread, we expect some variations in the checkpoint time, as exemplified by the slightly higher checkpoint time for a file size of 15 MB.

The Fast Restart time is the second curve from the bottom in Figure 2.5. The time for fast restart operation is nearly constant as the file size varies. This is the mmap optimization defers loading of most of the virtual memory pages. From our experiment, we see that the Fast Restart operation is about 3 to 10 times faster than the Standard Restart.

Table 2.1 – Size of Ckpt Image depending on Input file

Input file (MB)	2.0	5.1	10.2	15.4	20.5
Ckpt image (MB)	86.8	98.4	143.5	157.1	179.7

Table 2.1 shows the checkpoint image size as a function of the input file size. The checkpoint image size increases with the size of the input file, since the file data has been loaded into RAM during initialization. The image is large compared to the 2 MB input file, since the process is Java-based. The JVM must be checkpointed along with the loaded classes. The checkpoint image file size is also large because of the large Java classes running in the JVM. The size of the checkpoint image file increases more in absolute terms than the increase in size of the input file. This is because the data loaded are submitted to a semantic reasoner. This reasoner infers new knowledge that has been stored into the RAM and then must be saved as part of the checkpoint image.

2.4.3 Startup Times

In the second experiment, we discuss the difference in execution times between a restart and launching a fresh, new process that need to load data from a file.

Figure 2.6 shows the execution times in different situations. This compares the time for restarting a new process using the techniques of this work, versus the traditional alternative of starting (initializing) a new process for each new scene. The diamond-shaped and square plotted points represent the restart times for a

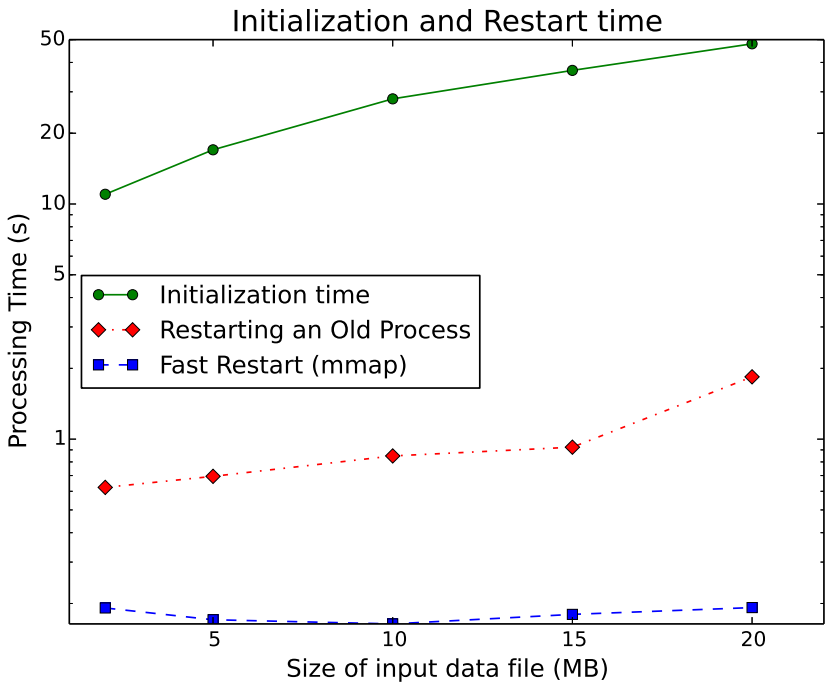


Figure 2.6 – Initialization of a new process versus restart of a previously checkpointed process.

checkpoint image. The square plot uses the mmap-based Fast-Restart option. The round plot represents the initialization time of the process when reading the data from the file. The initialization and restart times grow with the size of the input file that is loaded. Of the total initialization time, about 2 to 4 seconds is required solely to start the JVM before reaching the "main" method. The remaining time is used to load the Java-based semantic libraries and the input data.

Collecting together JVM startup, semantic library startup and loading the initial data, Figure 2.6 shows that "Restarting an Old Process" is about 25 times faster than the standard execution startup of a new process. Further, the Fast Restart method is about 500 times faster than the standard initialization. This is because restart avoids any data initialization that is executed by the Scene framework itself before it gives control to the end programmer.

2.4.4 Runtime Overhead when Running under DMTCP

In this software architecture, the process that handles processing of the scenes is larger since it is launched under the control of DMTCP. The use of DMTCP causes one or more DMTCP libraries to be loaded and a DMTCP checkpoint thread to be run within the target process (see [Ansel 2009]). However, the size of the resident RAM for the DMTCP libraries is quite small, about 2.6 MB, and does not vary with application inputs.

The DMTCP library also can slow the process at runtime due to interposition on

system calls from the target application to the runtime library. (See Section 1.2.1: process virtualization.)

Hence, the impact of DMTCP on the process depends especially on the number of system calls made by the process. This is small, since the model is dominated by computation, rather than by system calls in the given example.

Table 2.2 – DMTCP Overhead over 2000 OWL operations

File Size	2.05 MB	10.25 MB	20.50 MB
Time w/ DMTCP (ms)	3679	4252	3378
Time w/o DMTCP (ms)	3656	4262	3306

Table 2.2 shows the time required by the scene to execute 2000 OWL operations in the model. An operation correspond to adding a new triple of data into the semantic knowledge base. The execution of the target process is carried out in two different regimes: 1) with DMTCP that can checkpoint the program and system call wrappers; and 2) without DMTCP such as a native process. The aim is to determine if the use of DMTCP adds significant runtime overhead to the main process.

The last table shows that the times required to process the OWL operations is only 2.4% more with DMTCP than without, for a file size of 20.5 MB, and DMTCP has almost no effect on times for smaller input files. Notice that the 20.50 MB file is faster to process the operations than the others. This is due to the type of data that is used as input. Since we are using semantic reasoners and models, depending of the inserted data, the knowledge inferred is different, and the data will trigger different rules.

The test with the 10.25 MB input file is actually slightly faster with DMTCP. This is due to the natural random variation in processing time of a process allocated by the operating system scheduler.

2.4.5 Overhead of Passing name:value Pairs between Scenes

Section 2.3.3 presented the sharing of information between different scenes through a scene change. This is implemented using a DMTCP plugin [K. Arya 2016]. Such a plugin is able to define a set of functions to call when a checkpoint or restart operation is performed. We define a plugin to save that part of the state of the current scene to be checkpointed that has to be shared with other scenes. The plugin writes into a file the information to share when a checkpoint operation is performed. Then, on restart, the plugin will read this shared file and, depending of the hierarchy of the scene, will load into the memory the saved state. This mechanism slows down the system and this section evaluates the cost of this mechanism.

Table 2.3 shows the time required for the DMTCP plugin to save and load information from an information sharing file. The time depends on the number of

Table 2.3 – Average save and load time of DMTCP plugin in charge of the sharing of information between scenes.

Nb of pairs name:value	25,000	50,000	100,000	200,000
Save time (ms)	18	36	83	145
Load time (ms)	10	19	40	81

name:value pairs that have to be shared. In this experiment, we show that the time increases when more information has to be shared.

The average time for I/O for $200,000$ name:value pairs is about 145 ms for saving and 81 ms for loading. If we assume a typical 16 bytes per name:value pair, and a read/write speed of 100 MB/s for the SSD, then we would estimate 32 ms to save or load $200,000$ name:value pairs. The longer times for save/load occur because of the overhead of system calls and random access to the SSD. Nevertheless, the save/load times are acceptable, since they do not dominate over the times for checkpoint/restart times shown in Figure 2.5.

Conclusion

In order to manage the IoT software infrastructure, the definition and role of monitoring and execution component are required. Those components are the only ones interacting with the managed system, making them important interfaces to design.

The Monitor is gathering metrics from the managed system to determine its health. The Executor is performing actions to repair the issues. In the approach of this thesis, those actions are based on software migration operations with a checkpointing mechanism. Those operations allows the Executor to checkpoint a running process into a file, and restart it later on the same machine, or migrate it to another one. We also observed that the checkpointing mechanism had to be adapted in order to fit the resource constrained IoT machines.

Therefore, the technologies used for the monitoring, based on Device Management, have been presented in this chapter. An emphasis on the parameters that are required to be monitored by the component has been given. A clear need to have an extensible model for the representation of the application metrics has been highlighted.

Moreover, the execution has been discussed with the optimization of the checkpointing mechanism. The enhancements of the *Forked Checkpointing* and *Fast Restart* are well-suited mechanism for an IoT application. Subsequently, the issues of large RAM usage of processes is risen and answered by the proposition of a Scene mechanism. As for this mechanism, it allows the distribution of the application data in several scenes that can be swapped efficiently with the checkpoint/restart mechanism.

Finally, several experimentation are conducted to validate this approach. We demonstrated that the proposed scene architecture is about 25 times faster than the standard startup of a new process. When used with mmap-based fast restart (thus deferring paging in of virtual memory until runtime), the proposed architecture can even be 500 times faster. Moreover, the overhead of the checkpointing software DMTCP and the passing of *name:value* pair is evaluated. It has been demonstrated in previous works that the DMTCP is quite small (about 2.6 MB) [Ansel 2009], and the overhead in term of execution time depends on the number of system calls of the target process. It has been demonstrated that the passing of large number of pairs has a quite small execution time even for 200,000 pairs. Usually, applications does not handle this number of parameters.

This work has been published in [Aïssaoui 2016a].

The data gathered by the Monitor needs to be interpreted. For this purpose a model is required. Moreover, this model has to be extensible due to the diversity of possible application in the IoT. Chapter 3 provides the model used in the knowledge base of the autonomic framework. The given ontology aims at providing a vocabulary to describe the considered IoT software infrastructure. The different policies to apply to repair the system are also discussed.

Chapter 3

Knowledge Base for IoT Infrastructure Management

Contents

In the previous chapter, we demonstrated how to retrieve management data from the IoT infrastructure. Those information has to be stored by our management framework. Those data have to be interpreted in order to find out the potential problems in the system. This interpretation will lead to the creation of a plan of execution to fix the software infrastructure.

However, to interpret the data in an efficient manner, a model is required. Moreover, the description of the managed system has to be given to the management framework in order to acknowledge the existing entities and their capabilities. This means that a description of the currently available machine with their computation capabilities is required. We also need to represent the set of devices present in the environment, along side with the processes that are executed on this infrastructure.

In summary, we need the static description of the entities present in the system and the representation of the current state of the system that is retrieved by the monitor. In addition, this model has to be extensible depending on the specific application domain. It is required to be able to extend the model to fit several kind of scenarios.

To respect the given arguments, we suggested in Section 1.3 the usage of semantic technologies and representation. For this purpose, an ontology representing

the IoT software infrastructure is required. This vocabulary allows one to represent the targeted system to manage in a formal way, and store it into a knowledge base.

Moreover, this knowledge base contains the state of the entities present in the monitored system, allowing several components of the autonomic loop to retrieve relevant information for the problem inference. The Figure 3.1 shows the management framework with the knowledge base. This component is used by the inference system in order to determine the issues.

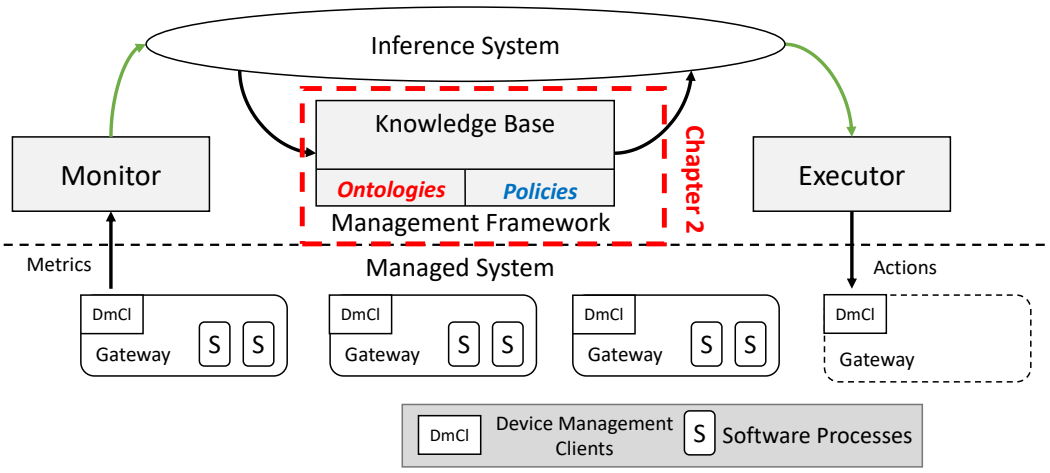


Figure 3.1 – Framework Architecture – Knowledge base

This chapter discusses the proposed ontology representing the different entities of an IoT software infrastructure, focusing on the software capabilities of the entities. Moreover, a description of several autonomic computing elements such Symptoms and Request for Change are provided. Finally, an instance of the model is given, based on a scenario on logistics.

3.1 IoT System Representation

This section presents the proposed ontology representing the different entities present in an IoT software infrastructure. The vocabulary aims at representing key parameters and metrics that will help the reasoner to find out the issues of the several processes.

An overview of the proposed ontology[1] is given in Figure 3.2. Note that all classes and relations are not represented in this figure. It displays the main modules of the ontology and some relations between the entities.

In order to create this ontology, we went through several steps in the representation. Each "part" of the ontology correspond to a module representing a type of entity.

[1]Ontology available at: `http://w3id.org/laas-iot/cpiot-o`

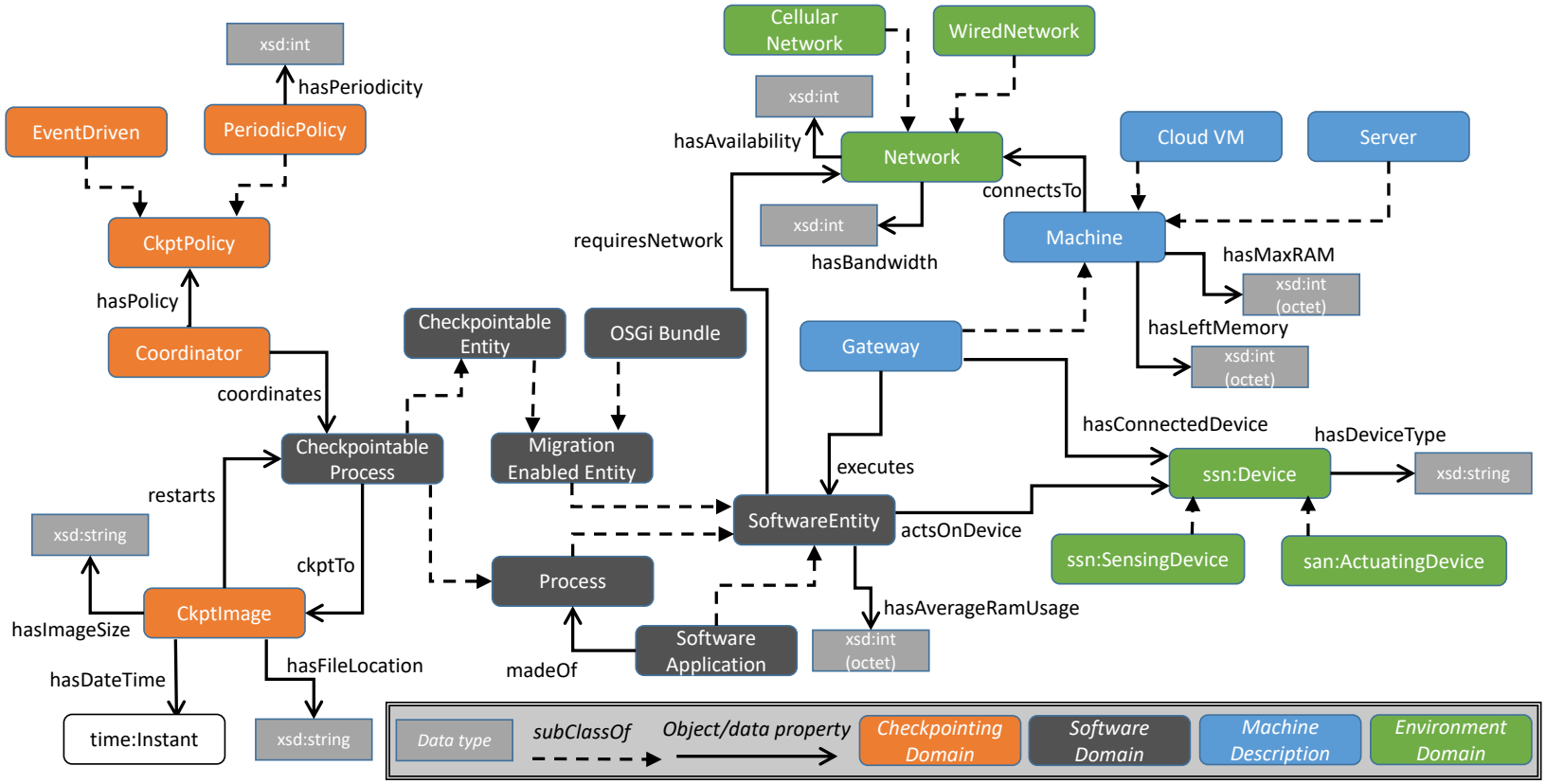


Figure 3.2 – Overview of the proposed ontology CpIoT-O

First, we defined the "Machine" module, aiming at representing the entities that are able to host and execute software processes. The type of machine can be important depending on the scenario and has to be represented alongside their capabilities. Then, we need to represent the environment with which the machine are interacting. This can include physical entities such as the devices, or the virtual networks. Additionally, the software processes that are executed on the machines are represented in a separated module. The migration capabilities of the software entities is also provided in the vocabulary. Finally, for the checkpointable entities, a checkpoint domain representation is given.

Moreover, in the vocabulary we consider two types of properties: "static properties" and "dynamic information properties". The *static properties* correspond to **descriptive information** provided by the IoT infrastructure 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 and the changes are tracked by the monitor component. 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.

3.1.1 Machine Description Module

The first step to represent the IoT software infrastructure is to provide a vocabulary for the hosts of the processes. The ontology provides several classes to represent a host and its capabilities.

In the ontology, a *Machine* corresponds to a physical (or virtual) entity that runs an operating system capable of hosting and executing some software. Several subclasses are defined: Gateway, Cloud Virtual Machine (CloudVM) or Server. There are other possibilities but since the focus is on the IoT domain, the main interest is to represent a Gateway. Servers and VMs are also interesting for an IoT approach since some processes may be placed temporally on this kind of equipment during the physical maintenance of the gateways.

The taxonomy of the machines is not enough for the problem. One needs to represents the computational capabilities in order to evaluate the available resources. For this purpose, several data properties are used.

A set of general properties linked to the top level Machine class contains: *hasMaxRAM*, *hasMaxDiskSpace* or *hasAverageEnergyConsumption*. Those relations gives a description of the machine for several parameters and will not change overtime. They are part of the *descriptive information* of the system.

The last relation, *hasAverageEnergyConsumption* correspond to a naive approach of representing the energy consumption. The machines are ranked by their energy consumption and this rank is stored in this relation. The lower this property is, the fewer the machine consumes energy. This approach allows the selection of a machine depending its energy consumption.

After the description of the machines has been given, their state needs to be

represented in the ontology. For this purpose, several relations are defined: *isMachineOnline*, *hasRamUsage* or *hasDiskUsage*. Those relations are part of the dynamic data that are gathered by the monitoring component. They define the current state of the machine.

A specific relation is describing the gateway. Since this kind of equipment may have a battery has a power source, the battery level has to be monitored. This parameter is given with the property *hasBatteryLevel*.

3.1.2 IoT Environment Module

The next module represent the physical entities that are interacting with the machines. This aims at representing the possible physically changing entities that the software processes are going to interact with. The ontology does not cover all possible IoT scenarios but proposes an approach for most common usages. This part focuses mainly on the representation of the devices in the IoT, and the network for the connectivity of the gateways.

A *Device* is a physical connected entity which is able to sense or act on the environment. Several kind of connected device can be represented such as a temperature sensor, a light sensor, a lamp, a heating system, etc. They can be defined as sub-classes of the *Device* definitions in the ontology. Moreover, some already published ontologies on the IoT already provides relevant vocabularies on the types of devices such as SSN[2] (other ontologies have been presented in Section 1.3.3). However, the definition of a general device is necessary in our approach. Additionaly, to facilitate the integration of some application in our management framework, we added an data property named *hasDeviceType* allowing one to define a type with a string.

A Device can be connected to a Gateway to send its information to an application. This connection is represented by the object property *isConnectedTo* that has for range the *Gateway* class. It may use a communication protocol, such as Bluetooth or Wi-Fi, in order to send or receive information from a gateway. This is translated in sub-properties of the *isConnectedTo* object property with relations such as *hasCellularConnection* or *hasBluetoothConnection*. The inverse property is defined as *hasConnectedDevice* and it lists the Devices connected to a Gateway. The Device concept is aligned with the definition of Sensor from the SSN ontology[3].

The *Network* class corresponds to a communication Network. It allows one to determine which entity is reachable through the Network. The connection of a gateway to a network is represented in the ontology by the object property *connectsTo*. The *Network* class has some attributes to specialize the network considered bandwidth or availability through the relations *hasBandwidth* or *hasAvailability*. Another relations is used to determine the usage of a network with the relation *hasAverageRoundTripTime*. This allows one to have an estimation of a network compared to others.

[2] `https://www.w3.org/TR/vocab-ssn/`

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

3.1.3 Software Domain Module

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.

However, all software entities are not possible to migrate. To make a distinction between those entities, we define the *MigrationEnabledEntities*. This class corresponds to an abstract concept on a piece of software that is possible to migrate by a specific mean. For instance, several sub-classes on this concept are possible such as an OSGi bundle or a checkpointable entity. Those concepts correspond to their own classes in the ontology.

For specific processes that are possible to migrate through the checkpointing mechanism, specific representation are given. This characteristic is described by a *CheckpointableEntity* class that represent a checkpointable Process or Application. *CheckpointableProcess* and *CheckpointableApplication* represents the corresponding checkpointable entities in term of classes in the ontology. Figure 3.3 illustrates this hierarchy.

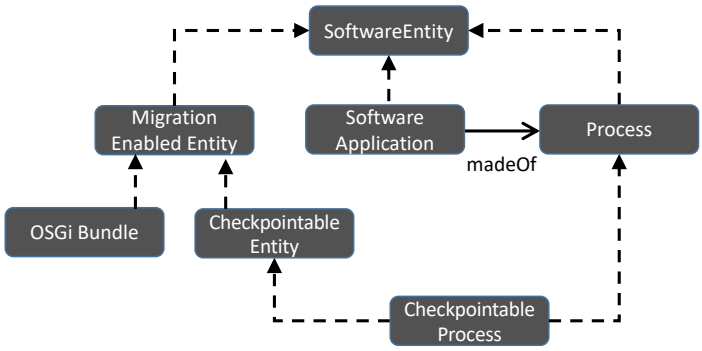


Figure 3.3 – SoftwareEntity taxonomy

Regarding the description of the software entities, some resource usage data properties are defined. Two data properties are *hasAverageRamUsage* and *hasAverageCpuUsage* that represents the average consumption of the machine resources. The RAM usage is expressed in mega bytes and the CPU usage is an approximate percentage of the process usage. This is not a totally precise metric but helps ranking the processes by their usage of the CPU.

Moreover, the software processes, depending on the application domain, have a set of requirements. A top level object property is defined as *hasFunctionalRequirementTo* and represents a generic requirement between two entities. This property has several sub properties defined in the ontology. This will allow an inference system to determine when they are not satisfied. For this purpose, we declare object

properties that link the software entities to their constraint. In this thesis we present two main constraints that are represented with the object properties *actsOnDevice* and *requiresNetwork*.

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. The *requiresNetwork* relation represent the need of a software entity to be executed on a gateway that is connected to the targeted network. If any of those functional requirements are not satisfied, a symptom needs to be raised.

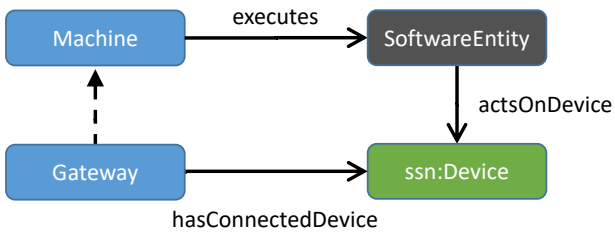


Figure 3.4 – Relation illustration between Machine, Device and Software classes. Also represents the possible constraint between a SoftwareEntity and a Device by the *actsOnDevice* object property.

A connection to a service ontology is possible with the software processes. A *Service* class is defined in the ontology and has a relation with the SoftwareEntity. The *providesService* object property links the software entitiy to provided services. Moreover, the Service concept is aligned with the eponymous class of MSM ontology.[4]

3.1.4 Checkpointing Module

The last module represents the checkpointing mechanism concepts present in the system. It is specifically targeted at representing the behavior of the DMTCP software.

Several classes are defined to represent the concepts handled by DMTCP. Indeed, a checkpointable process in DMTCP is connected to a *Coordinator*. A class representing this coordinator is defined and has a set of properties to represent its access point (IP address, port). This coordinator may have a checkpointing policy representing by another class (CheckpointingPolicy). The link between those entities is represented by the relation *hasPolicy*. We can find two types of policies: event driven policy and periodic policy. The event driven policy is equivalent to an application driven policy. This means that the application itself, i.e., the checkpointed process, will send signals to the coordinator to indicate when to per-

[4] `http://iserve.kmi.open.ac.uk/ns/msm#Service`

form a checkpoint operation. The periodic policy performs a checkpoint after a period. This parameter is also stored in the knowledge base with the data property *hasPeriodicity*.

Another part of the checkpointing modules represents the output of a checkpoint operation: a Checkpoint Image. This concept has an eponymous class in the ontology with several relations for its parameters: image size, image location and a time stamp. This checkpoint image is linked to the *CheckpointableProcess* with two object properties. The first one is *checkpointsTo* meaning that the image is the result of the checkpoint operation on the running process. The other relation is *restarts* which point out that the process has been restarted using the specified checkpoint image.

3.2 Representation of MAPE-K data in the ontology

In the previous section, the description of the system with its current state is represented in the ontology. Moreover, some functional properties can be expressed using semantic object properties. The aim is now to represent an issue in the system using the autonomic computing vocabulary.

The commonly used data from the autonomic computing are *Symptoms* and *Requests For Change (RFCs)*. Then, when those information has been inferred, a solution has to be defined in order to solve the issues. This is called a *Plan* and has to be affected by a high-level policy to guide the choices.

This section presents the set of symptoms and RFCs that are used in our management framework. Note that with the semantic technology approach, new symptoms and RFCs can be defined depending on the application domain.

3.2.1 Symptom and RFC representation

3.2.1.1 Symptoms

The *Symptom* class is an abstract representation of something outside the normal operation of the system. It aims at pointing out the defaulting entities or parameters.

By itself, this class is not enough to define the symptom of the system.

For each kind of symptom, sub-classes can be defined representing the specific issue. Moreover, a set of object properties can be added to the describe that will point out the deficient entities.

First, we defined a set of symptoms regarding the resource usage of the machines. Another abstract class, sub-class of Symptom is defined as *LackOfResource*, representing a generic lack of resource on a machine. The object property *hasSymptom* links the *Machine* class to the lack of resource symptom. With this definition, several sub-classes for different resources can be defined.

In the ontology, the proposed lack of resource are: *LackOfRam*, *LackOfMemory*, *LackOfEnergy* and *LackOfProcessingPower*. Depending of the lack of resource, dif-

ferent policies can be applied. For instance, when the RAM is almost full on a gateway, or the CPU usage is close to 100%, the migration of some processes is required for the gateway to be computing more efficiently. However, if the energy is lacking due to the battery depleting, only important or emergency processes have to be migrated to other gateways. All other "minor" processes can be stopped or moved to a server while the battery is recharging. This scenario is possible is the gateways are powered with solar panels. During the night, the energy may run low if during the day not enough power has been gathered. Figure 3.5 shows the different classes and the *hasSymptom* relation.

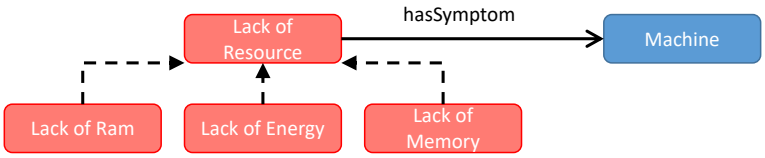


Figure 3.5 – Lack of Resource symptom with hasSymptom relation

The second type of symptom is related to the function requirements of the software processes. Indeed, when one requirement of a process is no more satisfied due to a change in the system, the process is considered in a defective state. This symptom is represented with the class *WrongSoftareLocation*, pointing out the defective process with the object property *concernsSoftwareEntity*. The inverse relation is also described in the ontology as *hasWrongLocation*. Additionally, a proposition of migration target are provided with the symptom. This is represented with the object property *potentiallyMigratesTo*. However, this symptom is not checking for the migration capabilities of the software entities. The expression of this parameter does not mean the migration will be possible. Figure 3.6 shows the WrongSoftware-Location symptom with its relations.

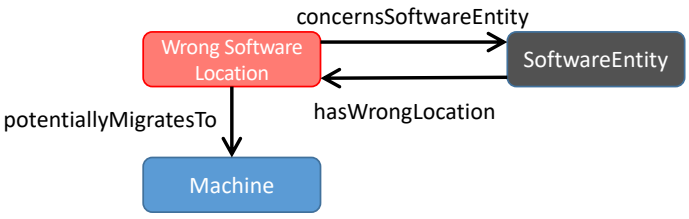


Figure 3.6 – Wrong Software Location symptom illustration

3.2.1.2 Requests for Change (RFCs)

After the defective entities of the system has been pointed out by the symptoms, the next step is to define the set of changes that needs to be applied.

This does not yet correspond to actions to perform on the system. It represents the general change that needs to be applied to the system in order to fix the issues.

In this ontology, we defines mainly two requests for change. Other kind of RFCs may be defined depending on the application scenario.

The first RFC proposed is represented by the class *LightenMachine*. This RFC represents the need to migrate some processes from a machine to another in order to free some resources. The object property *targetsMachine* represents the machine that is targeted by this request.

The second RFC has for semantic class *MigrateEntity*. It represents the need to migrate a software entity to a new machine. The concerned software entity is linked to the request with the relation *hasMigrationRequest*, and the inverse relation is also described as *targetEntity*. This entity has to be possible to migrate in order to create this migration request. Moreover, a list of possible migration targets is given with the object property *migrationTarget*.

Figure 3.7 shows the RFCs and the relations with the other classes. The classes with a light-red background are Symptoms. The classes with a light-purple background are RFCs. The other colors corresponds to the description of Figure 3.2.

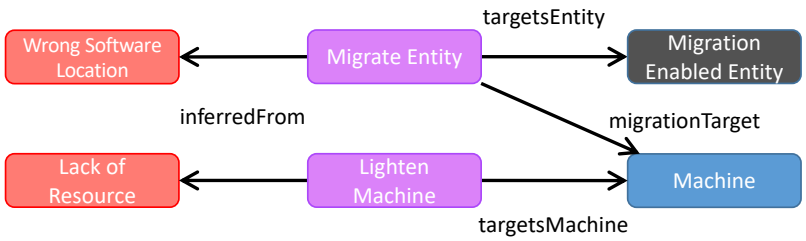


Figure 3.7 – Illustration of the RFCs and their relations

3.2.2 Policies

After the RFC has been created, the autonomic framework has to find a set of action in order to resolve the issues of the system. However, multiple solutions are possible in this problem, since multiple gateway may fulfill the requirements of the deficient software processes. For this purpose, policies are created to rank and allows a more precise selection of the migration targets when a migration request is raised.

In the ontology, a high-level *Policy* concept is defined. This class is extended with several sub-classes representing the different policies to apply.

An example of a sub-class of Policy is *RamPolicy* which defines the minimum available RAM required on the gateway of the system. This policy has a RAM percentage threshold that the management framework need to avoid to overtake. This managing policy can be applied to other resources such as the disk space value, or processing power.

Another kind of policy are handling the energy management of the machines. When two gateways satisfy the constraints for a process, the less energy consuming one has to be chosen in order to apply this policy, even if the second one has more processing power.

3.3 Model instance: Box of Vaccines

3.3.1 Scenario description

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. With a semantic description of each type of good, the specific policy can be applied on the system when needed.

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 over-loaded.

Figure 3.8 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 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.

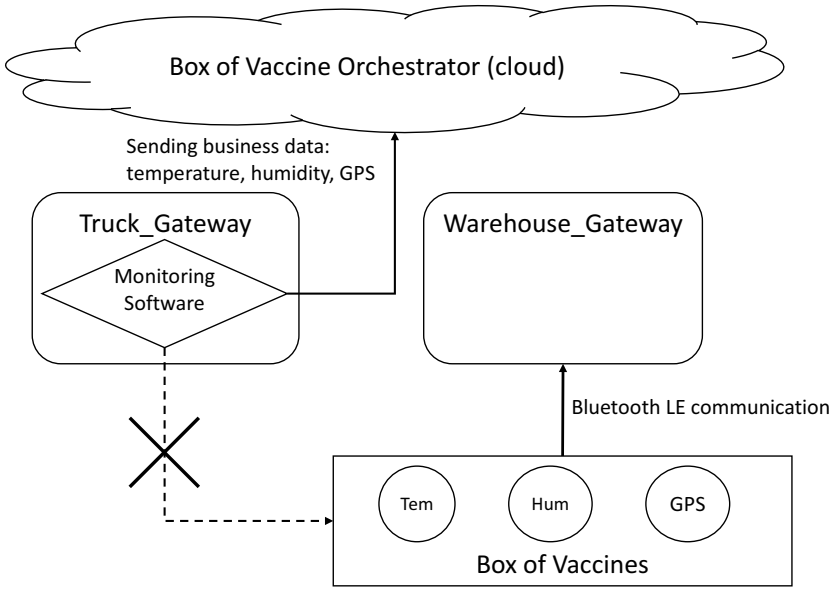


Figure 3.8 – Architecture of the logistics scenario based on box of vaccines.

3.3.2 Application on the model

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.

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 3.9 shows the instances created in the knowledge base. It represents a snapshot after the truck arrives at the site. The box is linked to its vaccine 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 must raise a symptom.

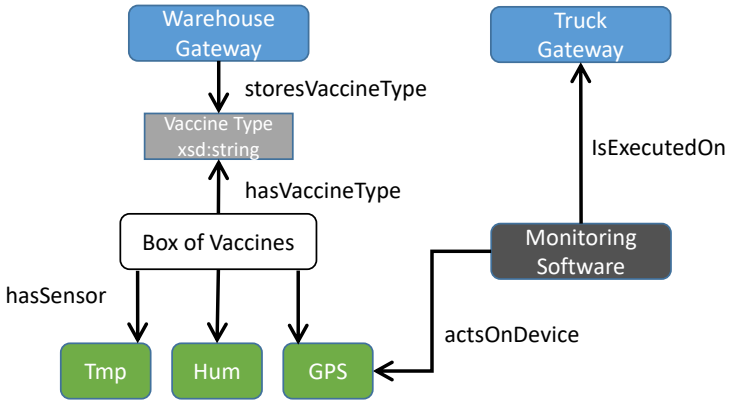


Figure 3.9 – Representation of the ontology instances used for the first logistics scenario.

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.

After receiving the information, our approach suggests to 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.

Conclusion

This chapter presented the model used in the knowledge base of the autonomic framework. It also provided the representation of symptoms and requests for change.

The needs arises to use a model to interpret the gathered data from the monitored system. This model has to provide the definition of the system alongside its state. Moreover, it has to be extensible in order to adapt the management framework to different IoT applications.

For this purpose, the semantic technologies have been used. This chapter has proposed an ontology that aims at describing the important entities of an IoT software infrastructure. With a module structure, the ontology has defined several concepts: machines that are hosting the processes the framework aims at managing ; software entities that are executed on the aforesaid machines ; the IoT environment comprising the physical entities interacting with the machines such as the devices ; and the checkpointing mechanism.

The ontology does not only describe the available entities in the system, but it also represents the current state of the system based on the data received in the Monitor. This is part of the role of the Monitor to maintain the data in the knowledge base updated and semantically represented.

In addition, the representation of the issues of the system are given with a set of possible Symptoms. Those symptoms allows the inference engine to determine which parameter the management framework need to have an impact on. The aforesaid step gives as a result a set RFCs also described in this chapter.

Finally, an instance of the proposed ontology is given using a logistics scenario. This shows the extensibility of the approach that can be extended to several IoT applications.

The main components of the proposed ontology have been presented in [Aïssaoui 2017].

The last part in need to be handled in the loop is the inference engine. Indeed, using the provided description of the entities and their current state, one needs to infer the issues of the system and creates the aforesaid symptoms. From this symptoms, a set of RFcs need to be defined. Finally, a plan of actions need to be defined in order to migrate the processes to solve the issues.

The next chapter presents the semantic rule engine, alongside the metaheuristics approach used to infer the previous cited information.

Chapter 4

Semantic Analyzer for Symptom and RFC inference

Contents

Several components of the autonomic computing approach has been covered in the previous chapters. The monitoring one aims at gathering metrics from the IoT software infrastructure. The need to represent those information in a formal way has been expressed. For this purpose, the knowledge base includes an ontology described in the previous chapter. Moreover, the representation of the possible issues of the system are represented in the vocabulary. The execution component has also been presented. The latter is based on software migration mechanism such as the checkpointing mechanism.

Two components of the autonomic computing are still missing: the *Analyzer* and the *Planner*.

The *Analyzer* uses the information gathered from the *Monitor* and the descriptive information of the entities in order to infer the *Symptoms* of the system. When the issues of the system have been found, another inference is performed in order to find out the required changes. Those changes indicates the parameters to change in the system, or a general idea of the action to perform in order to solve the symptoms. They are called Requests for Change (RFCs).

The Planner uses the inferred RFCs to define a plan. This plan is composed of the actions that will be performed by the executor component.

Figure 4.1 shows the complete architecture of the framework with all the components. Moreover, it highlights the approach taken to design the *Analyzer* and *Planner* components.

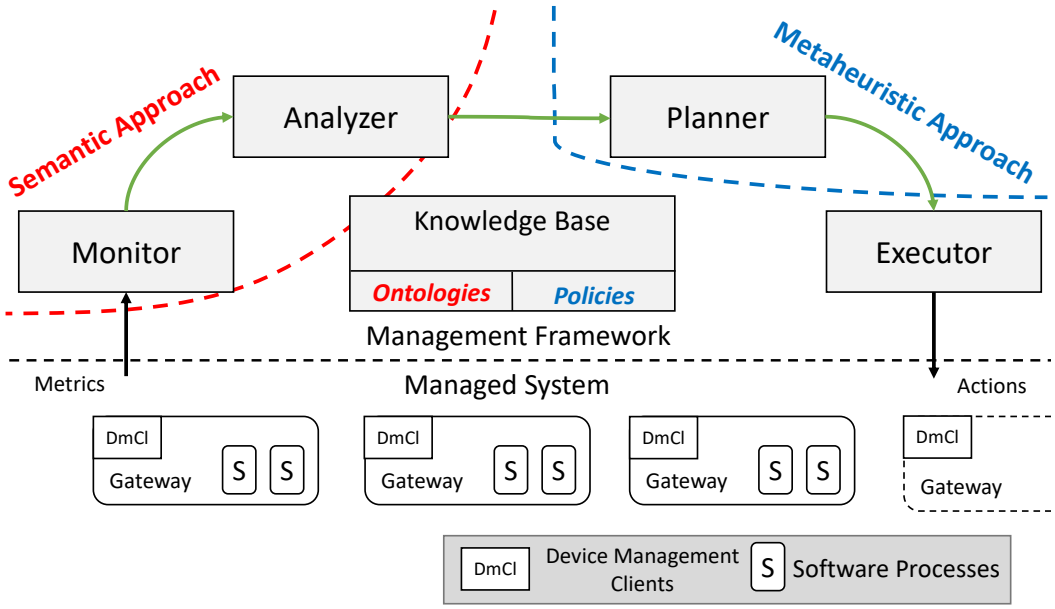


Figure 4.1 – Framework Architecture – Final architecture

In this thesis, we propose the usage of semantic reasoner in order to infer the symptoms and then the RFCs. This approach allows an easy and extensible representation of the issues, as demonstrated in the previous chapter, and rule system. In semantics, the rules are embedded with the model. They are represented directly in the semantic knowledge base. This means that they can be added directly in the application model.

This chapter presents the set of rules used to infer the symptoms presented in the previous chapter, and the rules to infer the corresponding RFCs. With this approach, we propose a simple algorithm that is able to infer a plan of action with the requested RFCs. The planing is handled by an algorithm also described in this chapter.

An evaluation of this approach is proposed with the box of vaccines scenario that has been presented in the previous chapter (See Section 3.3).

4.1 Analyzer: Semantic inference with SWRL rules

The first step in the inference part of the autonomic computing approach is the *Analyzer*. It uses the description of the system entities with their current state in order to find out the symptoms. Then, Requests for Change (RFCs) are inferred to solve those symptoms. A set of generic symptoms and RFCs have been defined in the previous chapter (see Section 3.2).

In order to perform this inference, a semantic reasoner is used. From the system description and state, the semantic reasoner is going to infer new knowledge. First, the definition and the inference rules of Symptoms and RFCs are provided. Then,

the algorithm used to find the actions to perform on the system is described.

4.1.1 Symptom Inference Rules

The inference of those symptoms is performed with SWRL rules. As described in Section 1.3, an SWRL rule is compounded of two parts. Each part contains a set of atom representing an OWL axiom. The first part defines a graph that the reasoner will try to match in the knowledge base. The second part are axioms that will be inserted in the knowledge base if the first part completely matches.

Moreover, in this approach we use an extension of SWRL providing the *swrlx:createOWLThing* atom.[1] The role of this atom is to create a new individual in the knowledge base. This individual is linked to the other arguments used in the atom. It means, if a rule is matching several times for other variable, it will create only one individual for each tuple of the other arguments.

In the previous chapter, the different generic symptoms are described in Section 3.2. Two kinds of symptoms are defined: resources usage violations and software entities functional property violations.

4.1.1.1 LackOfResource symptoms

For the first type of symptoms, the resources usage violations, we have to compare the current usage of the resources to the capabilities of the machines. The threshold to consider a machine in a critical state is stored in a Policy. In order to infer this kind of issues, the rule needs to retrieve the policy, the considered machines with their current state.

An example is given with the detection of exceeding RAM consumption of the machine. The rule for the inference of the symptom LackOfRam is provided in Listing 4.1.

The first three axioms are used to bind the variables to specific classes: *SystemContext*, *RamPolicy* and *Gateway*. This means, the possible graph pattern to match this rule will requires those variables to have the defined type (by the semantic relation `rdf:type`). With the *hasPolicy* relation, we retrieve the RAM policy linked to the system if it exists in the knowledge base. The *hasMinAllowedRamLeft* retrieves the value to compare with the machines of the system. The *hasMachine* ensures that the policy is applied to a machine in the same system and the relation *hasRamLeftPercent* retrieves the RAM left in percent of the machine variable. Then, a comparison is made between the RAM left on the machine and the threshold from the policy with the axiom `swrlb:lessThan`. The `swrlx:createOWLThing` is used to create the symptom individual in the knowledge base when the rest of the left part of the rule is true. The right part of the rule add the *LackOfRam* type to the `symptom` variable and the relation *hasSymptom* is added to the machine and points out the created symptom.

[1] *swrlx* defined in `http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl`

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

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

Listing 4.1 – Inference rule for LackOfRam symptom in SWRL.

The result of this rule is the creation of a new individual, linked to the gateway and the RAM policy. This means, only one individual will be created by pair of *Gateway+RamPolicy* matching this rule. The created individual is given a type by the second part of the rule, which is *LackOfRam*. Moreover, this created symptom is linked to the gateway with the relation *hasSymptom*.

Other similar rules are defined for other type of resource consumption. An example with the LackOfEnergy symptom is given in Listing 4.2.

```
SystemContext(?system) ^
EnergyPolicy(?policy) ^
Gateway(?gw) ^
Battery(?battery) ^
hasPolicy(?system, ?policy) ^
hasMinBatteryLeft(?policy, ?minBatteryLeft) ^
hasMachine(?system, ?machine) ^
hasBattery(?machine, ?battery) ^
hasBatteryLeftPercent(?battery, ?batteryLeft) ^
swrlb:lessThan(?batteryLeft, ?minBatteryLeft) ^
swrlx:createOWLThing(?symptom, ?gw, ?policy)

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

Listing 4.2 – Inference rule for LackOfEnergy symptom in SWRL.

Compared to the previous rule, the difference is that the battery level of the machine is stored in a separated entity. It is represented by the class *Battery* and is linked to the machine with the relation *hasBattery*. The battery level is retrieved with the data property *hasBatteryLeft*. For the policy, the same graph pattern

is retrieved but the class is not the same, i.e., *EnergyPolicy*. The data property *hasMinBatteryLeft* gives the threshold of the policy.

4.1.1.2 WrongSoftwareLocation inference with actsOnDevice

The second kind of symptoms is related to the software entities functional properties. The rule has to retrieve the functional property of the software entity, and find out if it is not satisfied. This symptom is represented with the semantic class *WrongSoftwareLocation*.

An instance of such a rule is given in Listing 4.3 with the verification of the *actsOnDevice* functional property. At first, the rule has to retrieve the software entities that has this constraint and the linked device. For this purpose, the software entities executed on the machines are retrieved with the relation *isExecutedOn* and are bound to the variable `softwareEntity`. The functional requirement of this entity is retrieved with the *actsOnDevice* object property and the concerned device is bound to the `device` variable. The machine where the device is connected is retrieved with the object property *isConnectedTo* and the machine is bound to the variable `deviceGw`. In order to determine if there is a violation, a comparison between the gateway where the software entity is executed and the gateway where the device is connected to has to be done. This comparison is performed by the axiom `differentFrom`, a base SWRL axiom. It checks if the individuals are not the same.

However, in semantic technologies, having a different URI for the individuals does not mean they are different. Because of the open world assumption, not expressing the fact that they are same does not mean they are different. The clear expression that the gateways are different individuals needs to be expressed in the knowledge with a *AllDifferent*[2] axiom between the gateways.

The symptom creation is linked to the software entity and the gateway where the device is. The rules creates the WrongSoftwareLocation symptom, linked to the software entity with the relation *concernsSoftware*. Moreover, the *potentiallyMigratesTo* relations is defined between the symptom and the gateway on which the device is connected.

As presented in Section 3.3, the *actsOnDevice* functional requirement is used in the box of vaccines scenario. This approach is evaluated in Section 4.2.

Other similar rules can be applied on other functional requirements. Listing 4.4 shows an example with the requirement *requiresNetwork*. The same approach as the previous rule is taking, making the comparison if the current gateway is connected to the network the software entity is executed on.

4.1.1.3 WrongSoftwareLocation inference with requiresDeviceType

Another functional requirement of the software entities is expressed with the data property *requiresDeviceType*. This property defines the required type of device

[2] `http://www.w3.org/2002/07/owl#AllDifferent`

```
SystemContext(?system) ^
Gateway(?softwareGw) ^
Gateway(?deviceGw) ^
SoftwareEntity(?softwareEntity) ^
hasMachine(?system, ?deviceGw) ^
hasMachine(?system, ?softwareGw) ^
actsOnDevice(?softwareEntity, ?device) ^
isConnectedTo(?device, ?deviceGw) ^
isExecutedOn(?softwareEntity, ?softwareGw) ^
differentFrom(?deviceGw, ?softwareGw) ^
swrlx:createOWLThing(?symptom, ?softwareEntity, ?deviceGw)

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

Listing 4.3 – Inference rule for WrongSoftwareLocation symptom in SWRL.

with a string.

The property means that if no device of the required type is connected to the gateway of the software entity, a symptom has to be raised. However, because of the open-world assumption, SWRL rules do not have negative axioms to represent the absence of a property.

To counter balance this assumption, a specific approach is taken. When the inference is performed, the framework takes in consideration that all device connections are already represented in the knowledge base. Also, the framework is aware of the possible current device types. With those information, a new data property is added to the Gateway where the device are connected: *absentDeviceType*. This property is added by the framework by subtracting the present device types on the Gateway to the complete set of device types.

With this new information added to the knowledge base, a new rule can be written to determine if the gateway does not have the required device for the process. The Listing 4.5 shows the SWRL rule.

This rule has two functionalities.

First, it creates the symptom *WrongSoftwareLocation* that represents the issues on the process. This is done by the first part of the rule that checks the equality between the required device type of the process and the absent device types of the gateway. To perform this creation, the machine is retrieved with the *hasMachine* relation and the software entity linked to it with the *isExecutedOn* relation. Then, the required device type is bound to the `requiredDeviceType` variable with the relation *requiresDeviceType*. The missing device types of the gateway are retrieved with the data property *hasAbsentDeviceType* and bound to `absentDeviceType`. The equality between the absent types and the required type

```
SystemContext(?system) ^
Gateway(?softwareGw) ^
Gateway(?networkGw) ^
Network(?network) ^
SoftwareEntity(?softwareEntity) ^
hasMachine(?system, ?networkGw) ^
hasMachine(?system, ?softwareGw) ^
requiresNetwork(?softwareEntity, ?network) ^
connectsTo(?networkGw, ?network) ^
isExecutedOn(?softwareEntity, ?softwareGw) ^
differentFrom(?deviceGw, ?softwareGw) ^
swrlx:createOWLThing(?symptom, ?softwareEntity, ?deviceGw)

-> WrongSoftwareLocation(?symptom) ^
potentiallyMigratesTo(?symptom, ?networkGw) ^
concernsSoftware(?symptom, ?softwareEntity)
```

Listing 4.4 – Inference rule for WrongSoftwareLocation symptom in SWRL due to the missing network.

is performed by the relation *swrlb:equal*, a base SWRL axiom. If all this matches, then the *swrlx:createOWLThing* axiom creates the symptom.

The second part retrieves the set of gateways that has a connected device with the required type. This is done by the *hasConnectedDevice* and *hasDeviceType* that links the device with the correct type to the gateway where it is connected. This allows to retrieve a gateway where the correct device type is available and place it in the symptom. This set of gateways is linked to the symptom with the relation *potentiallyMigratesTo*. This information will be used to easily extract potential migration targets when the reconfiguration of the software infrastructure will be defined.

4.1.2 Request for Change inference

When the symptoms are inferred, a second inference for the RFCs can be performed. Those RFCs represents a modification required for the correct operation of the system. It does not contain *how* to perform this change.

The first RFC proposed is named *MigrateEntity*. It represents the required migration of the software to a new location. The latter is inferred from the WrongSoftwareLocation symptom. The rule retrieves the symptoms and the concerned software with its maximum RAM usage. It compares this information with the RAM left on the possible migration targets. Also, the rule makes sure that the software entity is possible to migrate with the type *MigrationEnabledEntity*.

The second RFC is *LightenMachine*. The request represents the need to "lighten" a machine by removing some software. This RFC has an object prop-

```
Gateway(?gateway) ^
hasMachine(?system, ?gateway) ^
isExecutedOn(?softwareEntity, ?gateway) ^
requiresDeviceType(?softwareEntity, ?requiredDeviceType) ^
hasAbsentDeviceType(?gateway, ?absentDeviceType) ^
swrlb:equal(?absentDeviceType, ?requiredDeviceType) ^
hasMachine(?system, ?gatewayWithDeviceType) ^
hasConnectedDevice(?gatewayWithDeviceType, ?device) ^
cpiot-o:hasType(?device, ?requiredDeviceType) ^
swrlx:createOWLThing(?symptom, ?softwareEntity,
                                    ?requiredDeviceType) ->

WrongSoftwareLocation(?symptom) ^
concernsSoftware(?symptom, ?softwareEntity) ^
potentiallyMigratesTo(?symptom, ?gatewayWithDeviceType)
```

Listing 4.5 – Inference rule for WrongSoftwareLocation due to the missing Device Type on a Gateway.

erty that shows the target of the request: *targetsMachine*. This RFC is inferred when a LackOfResource 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.

The Listing 4.7 demonstrates the SWRL rule used to infer the LightenMachine RFC.

4.1.3 Actions to Perform on the System

When the Symptoms and RFCs are inferred by the reasoner, we need to define what actions on the system to resolve the issues. Based on the inferences, we propose an algorithm that creates a set of migration plans to repair the system state.

At first, we need to consider the gateway to lighten, in order to create some space for the incoming software. For this, we look at the currently running software on the gateway to be lightened and try 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

```
cpiot-o:WrongSoftwareLocation(?symptom) ^
cpiot-o:concernsSoftware(?symptom, ?softwareEntity) ^
MigrationEnabledEntity(?softwareEntity) ^
cpiot-o:potentiallyMigratesTo(?symptom, ?machine) ^
cpiot-o:hasMaxRamUsage(?softwareEntity, ?maxRamUsage) ^
cpiot-o:hasRamLeft(?machine, ?ramLeft) ^
swrlb:lessThan(?maxRamUsage, ?ramLeft) ^
swrlx:createOWLThing(?migrateRfc, ?softwareEntity, ?machine)
->
cpiot-o:MigrateEntity(?migrateRfc) ^
cpiot-o:toMigrate(?migrateRfc, ?softwareEntity) ^
cpiot-o:migrationTarget(?migrateRfc, ?machine)
```

Listing 4.6 – Inference rule for MigrateEntity RFC.

```
LackOfResource(?symptom) ^
cpiot-o:Machine(?machine) ^
cpiot-o:hasSymptom(?machine, ?symptom) ^
cpiot-o:potentiallyMigratesTo(?entity, ?machine) ^
swrlx:makeOWLThing(?rfc, ?machine) ->

cpiot-o:targetsMachine(?rfc, ?machine) ^
cpiot-o:LightenMachine(?rfc)
```

Listing 4.7 – Inference rule for LightenMachine RFC.

done in the previous part, it is not required to check again if there are enough resources for the migrations under consideration.

Algorithm 4.1 Algorithm for the establishment of Migration Plan using DMTCP

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

The algorithm presented in Algorithm 4.1 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.

4.2 Experimental Evaluation: Box of Vaccines Scenario

This section presents a scalability study performed on this first scenario, geographical migration, as described in Section 3.3. First, the experimental environment is specified, followed by the scalability study.

4.2.1 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 is its use in the IoT domain in general, we need to evaluate the size of the system that can be managed in a reasonable time.

To evaluate the scalability, we consider the first scenario, which was presented previously in Section 3.3. 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 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.

4.2.2 Scalability study

Figure 4.2 shows the results of the experiment with the detailed values in Table 4.1. 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.

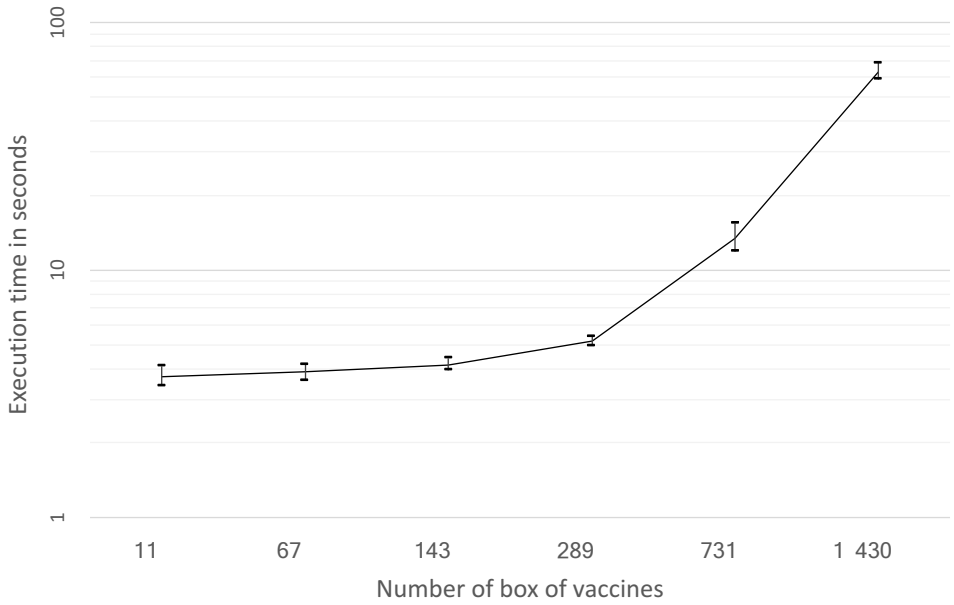


Figure 4.2 – Average-min-max chart of execution time of semantic reasoner depending on the number of box of vaccines.

Table 4.1 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. It 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.

Table 4.1 – This table displays the data shown in the figure 4.2.

Truck gateways	Box of vaccines	Average (s)	Minimum (s)	Maximum (s)
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 semantic reasoner has a static cost of 3 seconds, due

to the choice of technology. In particular, the creation of the rule engine with the SWRL API and Drools takes about 3 seconds to be instantiated, independently of the number of box of vaccines. However, this initialization has to be performed only one time at the start of the framework, it does not impact the performances of the latter executions.

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 approach has the goal of providing a quick analysis of the system in order to perform changes in reasonable time.

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.

Conclusion

This chapter presented the inference system used in the management framework to infer the Symptoms and the RFCs.

The Analyzer uses the system description with the data gathered by the monitor to infer the symptoms and RFCs. This is done by a set of SWRL rules and a semantic reasoner. Since the rule are part of the ontology, they can be easily extended for new scenarios.

We also evaluated this approach in term of scalability with a scenario based on logistic domain. We demonstrated that the response time grows exponentially depending on the number of box of vaccines as expected. Thus, more separated and lower infrastructures needs to be considered by this approach is order to functional in an acceptable time.

However, in order to create a plan of action the algorithm presented lacks the consideration of the optimization of the system. It can find a possible solution that match the functional requirements of the software processes but will not evaluate the quality of the solution. Moreover, since the inference system result correspond to a list of possible migration targets for each software process, it is difficult to create the optimal combination.

A combinatorial problem arises from this result when the optimization of the system is at stake. To solve this problem, a meta-heuristics approach is presented in the next chapter that aims at implementing the Planner component with a genetic algorithm.

The model including the semantic rules with the box of vaccine evaluation has been published in [Aïssaoui 2017].

Chapter 5

Software processes optimization in an IoT system

Contents

The previous chapter demonstrated how to infer the symptoms and RFCs of the software process infrastructure using web semantic technologies. The description of the entities in the knowledge base with their state allows the semantic reasoner to infer the issues of the system.

However, the inferred RFCs only point out the need to migrate a software entity, or lighten a machine. It does not explicitly indicate where to migrate the entities. Some RFCs hints possible locations for the migration but a decision needs to be made. The impact of a migration can be huge. Indeed, if a process has to be migrated to another machine, it will fill the target machine and may not leave enough resources for the processes to function properly. This means it will require the migration of other entities to another machine, but the same problem could appear again. In summary, this problem is combinatory and the optimization of the system parameters is difficult to perform with a standard algorithm in finite time.

Therefore, another kind of approach is taken in complement. We propose the use of meta-heuristics in order to find the plan of actions to perform on the system. More specifically, a genetic algorithm is used [Davis 1991]. The genetic algorithm accesses to the knowledge of the previous components and extracts the description of the system with a subset of constraints to reason on. Moreover, genetic algorithm has been proven to be efficient in the optimization of multiple objectives [Fonseca 1993].

The definition of an objective function, called fitness function, needs to be defined. The definition of the fitness function is based on a penalty, inspired by cloud

placement techniques such as [Yusoh 2010]. This approach has also been used in the literature to optimize the energy of cloud infrastructures [Wu 2012].

To conclude, this approach is evaluated in term of execution time compared to a brute force approach, and to the quality of the solution found.

5.1 Planner: System optimization through meta-heuristics

From the given requests for change inferred from the Analyzer and the knowledge base, the Planner has to define a plan of action in order to repair the system.

The previous sections shows how the RFCs are inferred in the framework. If we take in consideration the MigrateEntity RFC, we notice that it provides a possible set of migration targets for the software entity. This allows the selection for a specific entity but the placement of one software entity on a gateway may have some side effect. Indeed, placing several software entities on the same gateway may overload it. In that case, migrating other entities already present on the gateway is required and may have exactly the same side effect.

This decision of new software entities placement is quite complex. Moreover, in order to reduce the cost, we need to reduce the number of migrations since the software entities are not executed during the migration.

In order to solve this combinatorial problem, a meta-heuristics approach is taken. Since this approach is not implemented with semantic technologies, a translation of the knowledge is required. This section presents how the knowledge extraction is performed and serialized.

Those information are given to a genetic algorithm. Its goal is to find a new placement for the software entities in the system. The model used in the genetic algorithm and its execution are presented in this section and then evaluated.

5.1.1 Knowledge extraction and transformation

When the Symptoms and RFCs are inferred in the knowledge base, they are used by the planner component to find out the new process placement. However, the planner component that is executing the genetic algorithm is not using semantic technologies for execution time optimization purposes. To transfer those information from the semantic knowledge base to the planner component, the framework needs to extract the data with SPARQL queries and transform in a structured format.

The semantic knowledge extraction with SPARQL

The aim of this operation is to extract only the necessary information for the planner component. Unnecessary data for the inference of the new process placement has to be omitted if we want a faster execution of the genetic algorithm. For

this extraction two kinds of data has to be extracted: 1) the current process placement with the capabilities of the entities (e.g., the available RAM of the gateways) ; 2) the functional constraints of the software processes.

To represent the current placement of the software processes, at first the set of machines present in the system is retrieved with their capabilities. This information allows the genetic algorithm to be aware of the available machines where the software processes can be migrated. Moreover, the capabilities are used to determine if the inferred placement of the algorithm is possible in term of resource usage.

In the semantic knowledge base, this is done by retrieving all instances of the class *Machine* with their linked object properties such as *hasMaxRam* or *hasAverageEnergyConsumption*. Then, the link with the software processes is retrieved with the relation *executes* from the machines to the processes. Then, some information about the resource consumption of the softwares are fetched such as the *averageRamComsumption*.

The second part to extract are the expression of the functional requirements in the system. However, we do not want to overload the genetic algorithm with a lot of different information. Indeed, we defined some functional requirements in this work but it can be extended for other type of IoT applications. In that case, we do not want to change the extraction when the model evolves. Thus, we are using the information extracted from the Symptoms and the RFCs. They contains the set of possible machines where the software entities can be migrated (represented by the relation *potentiallyMigratesTo*). The extraction of such information is performed with a SPARQL query presented in Listing 5.1.

```
PREFIX cpioto: <http://w3id.org/laas-iot/cpiot-o#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT * WHERE {
 Type(?softwareEntity, cpioto:MigrationEnabledEntity),
 Type(?migrationTarget, cpioto:Gateway),
 Type(?symptom, cpioto:WrongSoftwareLocation),
 PropertyValue(?softwareEntity, cpioto:hasWrongLocation,
                                                ?symptom),
 PropertyValue(?symptom, cpioto:potentiallyMigratesTo,
                                            ?migrationTarget)
}
```

Listing 5.1 – SPARQL-DL query to extract *WrongSoftwareLocation* symptoms

This SPARQL request retrieves the set of software entities that have a specific type: *MigrationEnabledEntity*. This is expressed with the clause `Type(?softwareEntity, MigrationEnabledEntity)`. The same approach is taken to retrieve the gateways and the symptoms with the *WrongSoftwareLocation* type. Then, the requests filters the software entities that have a symptom with the relation *hasWrongSoftwareLocation*. If a symptom is found that way, the list

of migration targets is retrieved by the relation *potentiallyMigratesTo*. This list is ensured to be gateways by the previous type statement.

Serialization in XML

The extracted knowledge is serialized in XML. We did not use RDF representation for the serialization but provide our own structure. The reason is due to the technologies used for the genetic algorithm. It is written in C++ and the libraries to handle RDF and query the graph are not well developed. The ecosystem is more maintained around Java technologies. That is why we extract first the knowledge and then serialize it in our own structure.

The root node of the XML is `ga-model` and envelops all the information. Since the knowledge is compounded of two parts, the XML has a similar structure.

The first part corresponds to the description of the system. It first lists the set of machines under the tag `machines`. Underneath, several `machine` tags are used with the capabilities of the concerned machine as attributes. Also, the semantic URI of the machine is added as an XML attribute. If the machine executes a software process, a child node `softwareInstances` is added representing the process. Similarly to the machines, the node contains the resource usage as attributes.

The second part of the XML are the functional constraints. This is represented by the nodes named `constraints`. The latter correspond to a list of node called `softwareConstraints` having a parameter that points out the concerned software entity. As a child node, it has a list of possible migration targets embedded in the node `migrationTargets`.

Listing 5.2 shows an example of extracted knowledge in XML in a simplified case. There is two machines, with equivalent resources. However, a software entity is executed on the second one. In the constraints, we notice that this software entity has a constraint and needs to be executed on the gateway `GW_0`. For the readability, the URIs have been simplified with `URI#`.

A more complex and relevant example is provided in Appendix **??**.

5.1.2 Genetic Algorithm Resolution

In order to find out the new software processes placement, a genetic algorithm is used. It is considered as an optimization algorithm, meaning that it tries to enhance a set of inputs into a *better* set of output. The quality of a placement is given by a fitness function. Depending on its definition, the output of the algorithm can vary.

The genetic algorithm is part of evolutionary computation algorithms. It is based on genetic transformations and natural selection mechanisms. The base data are *chromosomes* that will be transformed with specific *operators* and inserted in the global *population*. Each chromosome has a set of *genes* that have a *allele*. Thus, each chromosome can be represented as an array, with each indexes representing a gene. The value stored in the array corresponds to the allele.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ga-model>
    <machines>
        <machine maxRam="256" ramLeft="256" energyFactor="1"
                 uri="URI#GW_0"/>
        <machine maxRam="256" ramLeft="240" energyFactor="1"
                 uri="URI#GW_1">
            <softwareInstances averageRamConsumption="16"
                               uri="URI#SE_0"/>
        </machine>
    </machines>
    <constraints>
        <softwareConstraints concernedSoftwareUri="URI#SE_0">
            <migrationTargets>URI#GW_0</migrationTargets>
        </softwareConstraints>
    </constraints>
</ga-model>
```

Listing 5.2 – Example of extracted knowledge serialized in XML.

The aim of the algorithm is to generate several chromosomes in an almost random manner, following genetic operators, and find out the best one.

5.1.2.1 Genetic Algorithm Model

We need to specialize the data structure of the genetic algorithm to our problem. A chromosome corresponds to a placement of the software processes on the machine. Each gene corresponds to a software entity. This means that an array representing a chromosome has for size the number of software entities in the system. The allele of the genes correspond to the machine where the software entity is executed.

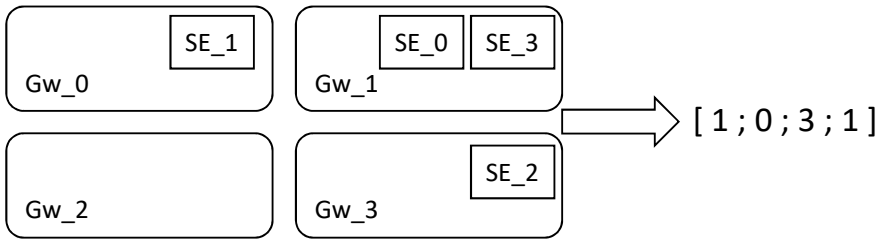


Figure 5.1 – Example of encoding for the Genetic Algorithm

Figure 5.1 shows an example of encoding done for the genetic algorithm. The `Gw_i` corresponds to the gateways and `SE_i` to the software entities. There are 4 software entities executed on the 4 gateways. The representation is a chromosome with 4 genes. The first index corresponds to the `SE_0` and then the index is incrementing at the same time as the name of the software entity. The value corresponds to the gateway where the entity is executed.

5.1.2.2 Algorithm execution

Algorithm 5.1 shows the execution of the genetic algorithm.

The input is the current state of the system. It is encoded in a chromosome called Ch_{init}. The output of the algorithm is another chromosome called $Ch_{optimum}$. In order to evaluate the quality of the chromosome, a fitness function is defined. It takes a chromosome as argument and returns a score. In order to optimize the system, this score has to be minimized.

Algorithm 5.1 Genetic Algorithm execution

Require: Ch_{init}, initial chromosome
Ensure: $Ch_{optimum}$, optimized solution
  population initialization from Ch_{init}
  $Ch_{optimum}$ such as $f_{fitness}(Ch_{optimum}) = +\infty$
  while stop criteria not met **do**
    Crossover on the population
    Mutation on population
    Survivor selection
    Find *best*
    if $f_{fitness}(best) < f_{fitness}(Ch_{optimum})$ **then**
      $Ch_{optimum} = best$
    end if
  end while
  return $Ch_{optimum}$

To start the algorithm, an **initial population** is required. To create population, random mutations are performed on the initial chromosome in order to reach a population of 50 individuals.

After the initialization, the algorithm enters in the main loop which is performed while the stop criteria are not met. For each iteration, multiple **crossovers** between the chromosomes of the population followed by **mutations** are performed.

The number of crossover performed is five times the size of a chromosome. For each operation, two different random chromosomes are chosen. This helps generating more possible solutions when the problem is larger. Then, for each chromosome present in the population, the mutation operator is applied. This double the size of the current population.

At the end of the iteration, the survivor selection is performed. It is composed of two steps: 1) **validation** and 2) **evaluation of the chromosomes**.

For the **validation**, the algorithm checks if the functional requirements of the software entities are satisfied in the generated solutions. If it is not the case, the concerned chromosomes are removed from the population. Moreover, the resource usage of the machine is checked by the component. If the resource usage exceeds the limit on a machine, the chromosome is also removed.

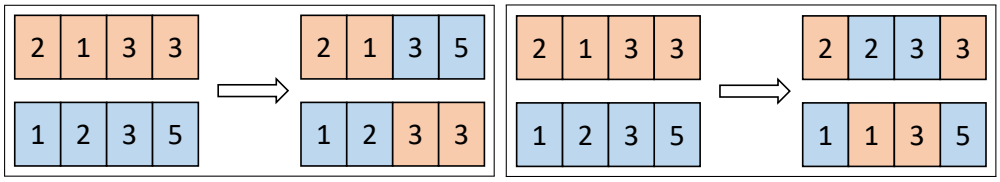
When the validation is done, the **evaluation** step retrieves the fitness value of the chromosomes with the eponymous function. Then, the chromosomes are sorted

by their fitness value and the n bests are kept, with $n = 10 * sizeOfChromosome$. After that, the stop criteria are evaluated and the loop repeats.

5.1.2.3 Genetic Operators

As we have seen in the algorithm execution, two main genetic operators are defined.

Crossover operator: It takes in argument two chromosomes. The aim is to mix the two chromosomes in order to create two others chromosomes. The aim is to take a part of the first chromosome and a part of the second chromosome is order to create another one. Several breaking points can be taken when performing this operation. Figure 5.2a shows an example of a one point crossover while Figure 5.2b shows an instance of multi-point crossover.



(a) One point crossover (b) Multi-point crossover

Figure 5.2 – Crossover operator examples

Mutation operator: It takes two arguments, a chromosome and the list of functional constraints. The aim of this operator is to change randomly some allele of the chromosome. In order to reduce the number of incorrect chromosome, the allele (the machine) is selected depending on the constraints of the gene (the software entity).

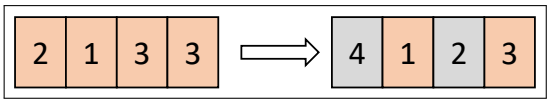


Figure 5.3 – Mutation operator example

5.1.2.4 Stop criteria

In order to stop the algorithm, some criteria has to be defined. In our approach, we use mainly 2 criteria.

The first criterion is the **number of iterations**. This standard stop criteria allows to define the maximum number of iterations the algorithm has to perform. In the framework, the maximum number of iterations is set to 10. This number does not depend on the dimension of the problem because the number of generated chromosomes scales with it. 10 iterations also allows the algorithm to get through a lot of elements in the solution space while having to much iterations would be useless

on low dimensions. It would cause the algorithm to evaluate more chromosome than the dimension of the solution space.

The second criterion is the **variation of the solution** found. Indeed, the solution did not vary for several iterations, it is possible to have reach a local minimum. In that case, if the minimum has not changed for 3 iterations, the algorithm stops. This enables a faster execution of the algorithm when the solution is not progressing. The choice of the number is based on the total number of iterations. We estimate if the algorithm does not find more interesting solution in 3 iterations, which correspond to the third of total iteration, then there is few possibilities to find a new minimum. Of course, with the random part of the genetic algorithm, it is possible to find out a new global minimum in the set of solution, in that case the algorithm is continued.

5.1.2.5 Fitness function

The fitness function serves as a measuring function of the quality of the chromosomes. It takes as an argument a chromosome and return a integer value.

We defined the fitness function in two parts. The first one computes the differences between the initial chromosome and the provided one. Indeed, each difference represents a software migration in the real system. Each migration has a cost. Therefore, the number of migration has to be reduced. To represent this in fitness, we increment the value by the RAM usage of the processes that needs to be migrated. Since the cost of the migration is proportional to the RAM usage, it directly reflects this cost in the fitness.

The second part correspond to the global resource usage. For instance, we want to promote a system where the energy consumption is the lowest possible. Thus, we define a cost independent of the initial chromosomes that evaluate the new state of the system. We use the average energy consumption of the machines and multiply it to the RAM usage of the processes executed on the machine.

In a mathematical approach, we defined the set $P = \{p_1, p_2, \ldots, p_n\}$, with n the number of software processes. The set of hosts (machines) is represented with $H = \{h_1, h_2, \ldots, h_m\}$, with m the number of machines.

For mathematical representation, a placement function based on the chromosome is defined such as:

$$Ch, p \rightarrow f_{placement}(Ch, p) = h, p \in P, h \in H$$

The fitness function is defined as:

$$f_{fitness}(Ch) = \Delta_{Ch_{init},Ch} + f_{cost}(Ch)$$

With,

$$\Delta_{Ch_{init},Ch} = \sum_{p_i \in P} \delta_{p_i}$$

$$\delta_{p_i} = \begin{cases} mem^{p_i}_{usage} & \text{if } f_{placement}(Ch_{init}, p_i) \neq f_{placement}(Ch, p_i) \\ 0 & \text{otherwise.} \end{cases}$$

With $E(h)$ the function representing the energy consumption of a machine, the function f_{cost} represents the static cost of the chromosome and is defined as:

$$f_{cost}(Ch) = mem^{p_i}_{usage} \times E(f_{placement}(Ch))$$

In order to normalize the fitness function, we only used the software entities RAM usage with some modifiers. Moreover, the addition between the two elements allows an easy comparison. For example, if the migration cost is the same for two chromosomes, but on one, a process will be hosted on a more machine using more energy, the other solution will be prioritized.

It is important to note that the result of the fitness function is only used to rank the chromosomes and does not represent anything.

5.2 Experimental Evaluation

This section presents an experimental evaluation of the framework. The aim is to show that the response time needed to perform an iteration is satisfying in term of execution time and the quality of the solution found.

First, the experimental context is provided. Then, the evaluation of the performances of the framework is performed in term of execution time. Finally, the quality of the solution found by the genetic algorithm is discussed in comparison to the optimal solution found by a brute-force algorithm.

5.2.1 Experimental context

5.2.1.1 Model used in the Scenario

To evaluate the framework, a simulation of the IoT software infrastructure is performed. A fixed number of gateway that can host the processes is taken.

We consider a fixed number of gateway of available in the architecture to 10. In those gateways, half of them shares a profile, and the other half another profile. The first profile has a low energy consumption but with only 256 MB of RAM. The second profile has 1024 MB and a larger energy consumption.

Regarding the devices, 27 devices are deployed and have a specific device type. Five types of device are used and are linked to different gateways. The device repartition is also shown in Figure 5.4.

The number of software entities will be the parameter varied in the experiments. The entities are also separated in four profiles. The functional requirement used for all entities is the *requiresDeviceType*. The class repartition is the following:

Class 1 : 16 MB of RAM usage and 1 required data type

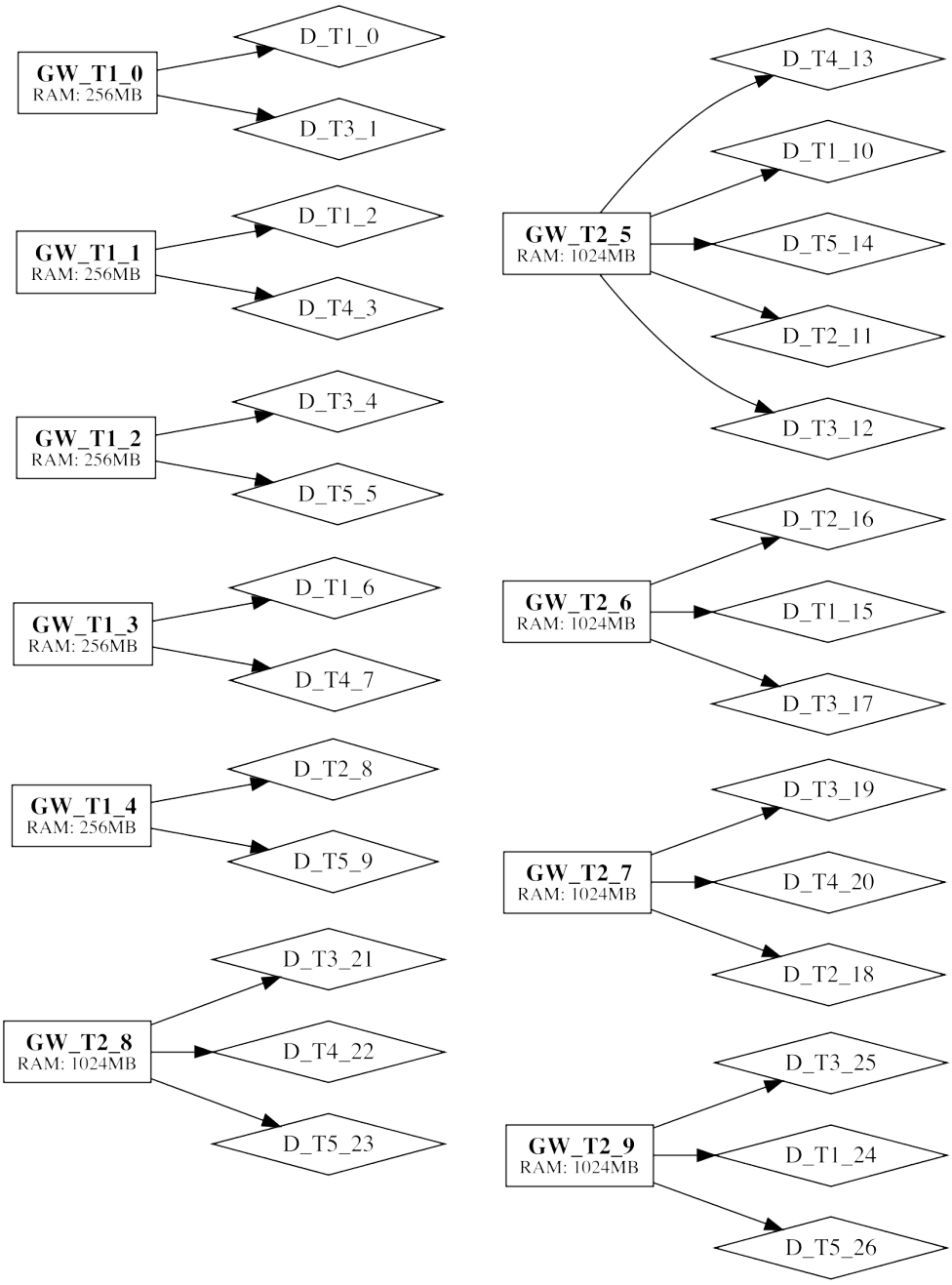


Figure 5.4 – Fixed gateway architecture used in the experimental evaluation.

Class 2 : 64 MB of RAM usage and 1 required data type

Class 3 : 16 MB of RAM usage and 2 required data type

Class 4 : 64 MB of RAM usage and 2 required data type

In the following, the number of software entities generated will be used as the **dimension** of the problem. The repartition of software entities in the classes is done randomly. Thus, several instances of the same dimension will lead to different configurations since the repartition of classes will be different.

For each dimension, a hundred of instances of this model are generated. The set of dimension chosen comprises all values between 5 and 10, and from 10 to 150 with steps of 5, for a total of 34 dimensions evaluated. We did not push after 150 dimensions because the system is already almost full in term of resource usage .

5.2.1.2 Experimental environment

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 Monitor and the Analyzer are written in Java. The JVM used is OpenJDK JVM version 1.8.0_171. 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 knowledge base 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. The SparqlDl library (version 2.0.0) is used to perform the Sparql requests on the knowledge base. Protégé version 5.1.0 has been used to create the model, but it is not used in the experiment.

The Planner with the Genetic Algorithm is written in C++, and has been compiled with `g++` version 4.8.4. The standard level used for the code is C++11, and optimization flag was `-O2`. The Boost library (version 1.66) has been used to parse the XML representation and to parse the command line arguments.

5.2.2 Performance evaluation

5.2.2.1 Analyzer execution

Figure 5.5 shows the execution time of the semantic reasoner for the inference, and the time for the extraction. The values corresponds to the average execution time over the 100 models for each dimensions. For each model, the experiment is run 10 times. The data table is available at Table 5.1.

Dimension	Mean of Extraction time (ms)	Mean of Inference time (ms)
5	935	764
6	939	769

Table 5.1 continued from previous page

Dimension	Mean of Extraction time (ms)	Mean of Inference time (ms)
7	952	776
8	950	781
9	956	784
10	956	784
15	979	806
20	997	819
25	1017	830
30	1041	849
35	1055	864
40	1077	871
45	1094	884
50	1121	898
55	1139	910
60	1163	917
65	1184	922
70	1200	934
75	1223	942
80	1245	949
85	1261	954
90	1275	963
95	1299	967
100	1323	972
105	1327	978
110	1355	981
115	1370	989
120	1381	991
125	1403	995
130	1430	1002
135	1453	1011
140	1478	1019
145	1494	1013
150	1512	1015

Table 5.1 – Mean of the inference time by the semantic reasoner on the base model and mean of extraction time of the knowledge to transform the data for the genetic algorithm.

The inference time does **not** include the genetic algorithm. It is only the inference of the semantic rules also presented in Section 4.1 but with the new scenario.

The inference time goes from 764 ms in average for the dimension 5 to 1015 ms for the dimension 150. The semantic reasoner has also a 3 s initialization time

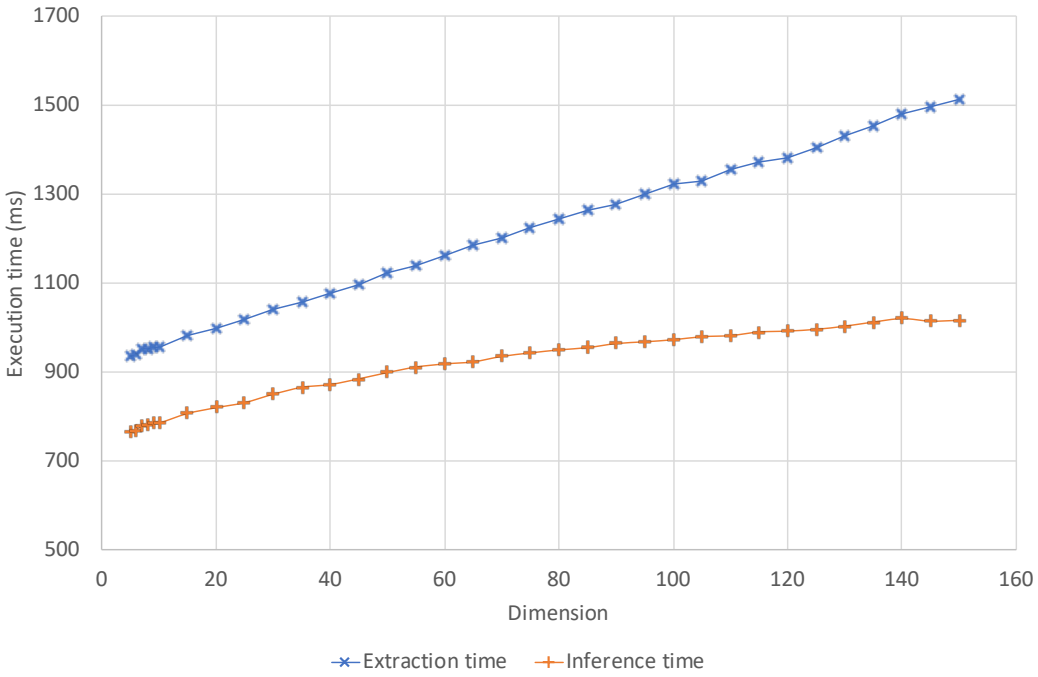


Figure 5.5 – Execution time of inference and extraction in the semantic knowledge base.

but it is not represented in the data since it can be initialize at the start of the framework. We observe that the inference time increases with the dimension of the problem but stays around 1 s which is a quite acceptable value.

For the extraction time, it starts at 935 ms for the dimension 5 and goes up to 1512 ms. We notice that the extraction time is longer than the inference time. It also increases with the dimension of the problem since there is more entities to extract from the knowledge base.

5.2.2.2 Genetic algorithm evaluation

First, the execution time of the genetic algorithm is studied on the presented scenario.

Figure 5.6 shows the mean of execution time of the genetic algorithm depending on the dimension of the problem. Moreover, the mean of chromosome generated is given. A view of the distribution of the data is given with box plot figures.

The execution time of the genetic algorithm goes from 3 ms on average for the dimension 5 to 2275 ms for the dimension 150. The time is correlated to the number of chromosome generated during the algorithm. The number does not increase linearly since in some iterations of the genetic algorithm, the same chromosome is generated multiple times but it is only added one time to the population.

We can see on the distribution that there are some points that are significantly lower than the mean of the series. This is due to the different stop criteria of

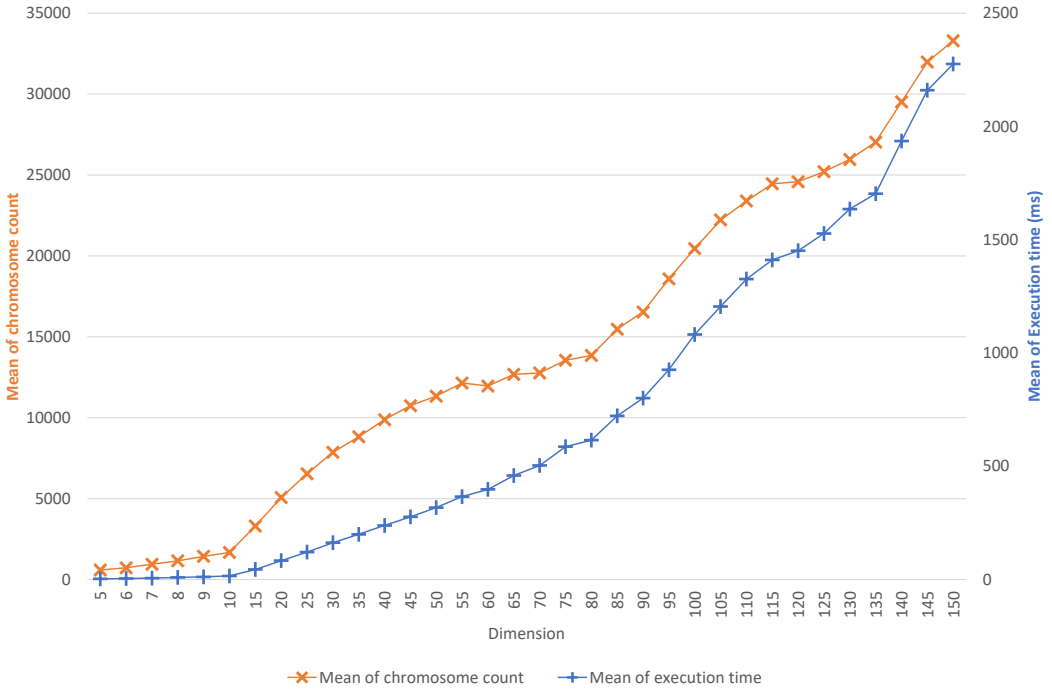


Figure 5.6 – Mean of execution time and number of generated chromosome of the genetic algorithm.

the algorithm. If the minimum found has not changed for 3 successive iterations, the algorithm stops. Thus, in some experiments the algorithm is faster when this criterion is triggered early in the computation.

Moreover, even for a high number of dimensions, the response time around 2 seconds, with the higher time being 3016 ms. We notice a gap in the execution time from the dimension 140 where the execution time is higher. This is due to the fact that the system is almost overloaded. For this reason, the algorithm tries to find possible solutions that are not overloaded and almost never trigger the previously mentioned stop criterion. There are still some exception as the distribution shows at Figure 5.8.

5.2.2.3 Comparison with brute force algorithm

In order to compare the results of the genetic algorithm, we implemented a brute force algorithm. It explores all possible members of the solution set and find out the one with the lowest fitness value. However, the algorithm does not explore impossible solutions from the algorithm. It uses the same constraints given to the genetic algorithm in order to reduce the combinatorial possibilities. Therefore, the execution time of the brute force algorithm depends on the dimension of the problem and the problem constraints.

Figure 5.9 shows a comparison of the mean of execution time for the dimensions 5 and 10. It is only displayed for 10 problems for each dimension.

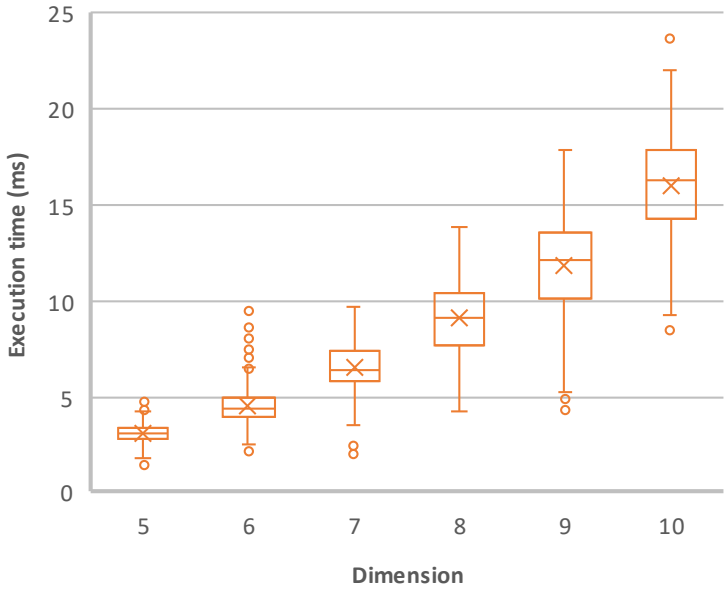


Figure 5.7 – Execution time of the genetic algorithm for dimension 5 to 10.

As a general trend, the genetic algorithm is faster than the brute force, even on really small problems (dimension 5). Only on the problem $[5; 10]$, the brute force is faster. This is caused by the number of expressed constraints. If there is a lot of constraints to apply, the set of possible solution is quite small and the brute force may take less time.

For the dimension 10, the genetic algorithm has an execution time of about 16 ms when the brute force takes from 24 seconds to 729 seconds for the problems with fewer constraints. Even in the best case for the brute force, the genetic algorithm is 1500 times faster for this dimension. Moreover, for the next dimension of 15, the combinatorial possibilities are strongly higher. With 5 more processes, it means in the worst case that there is 10^5 times more possible solutions in the set, since there are 10 possible machines. Going through all those possibilities makes the brute force impossible to compute a solution in a finite time. However, the genetic algorithms only takes on average 44 ms.

We clearly see the need to use meta-heuristics for this kind of problems in order to find a near-optimal solution in finite time.

Additionally, some optimizations can be added to the genetic algorithm and the brute force. The given execution times are determined with single threaded implementation of the algorithms. A multi-threaded approach would reduce the execution time of both algorithms since the generation of the chromosome is easily parallelizable.

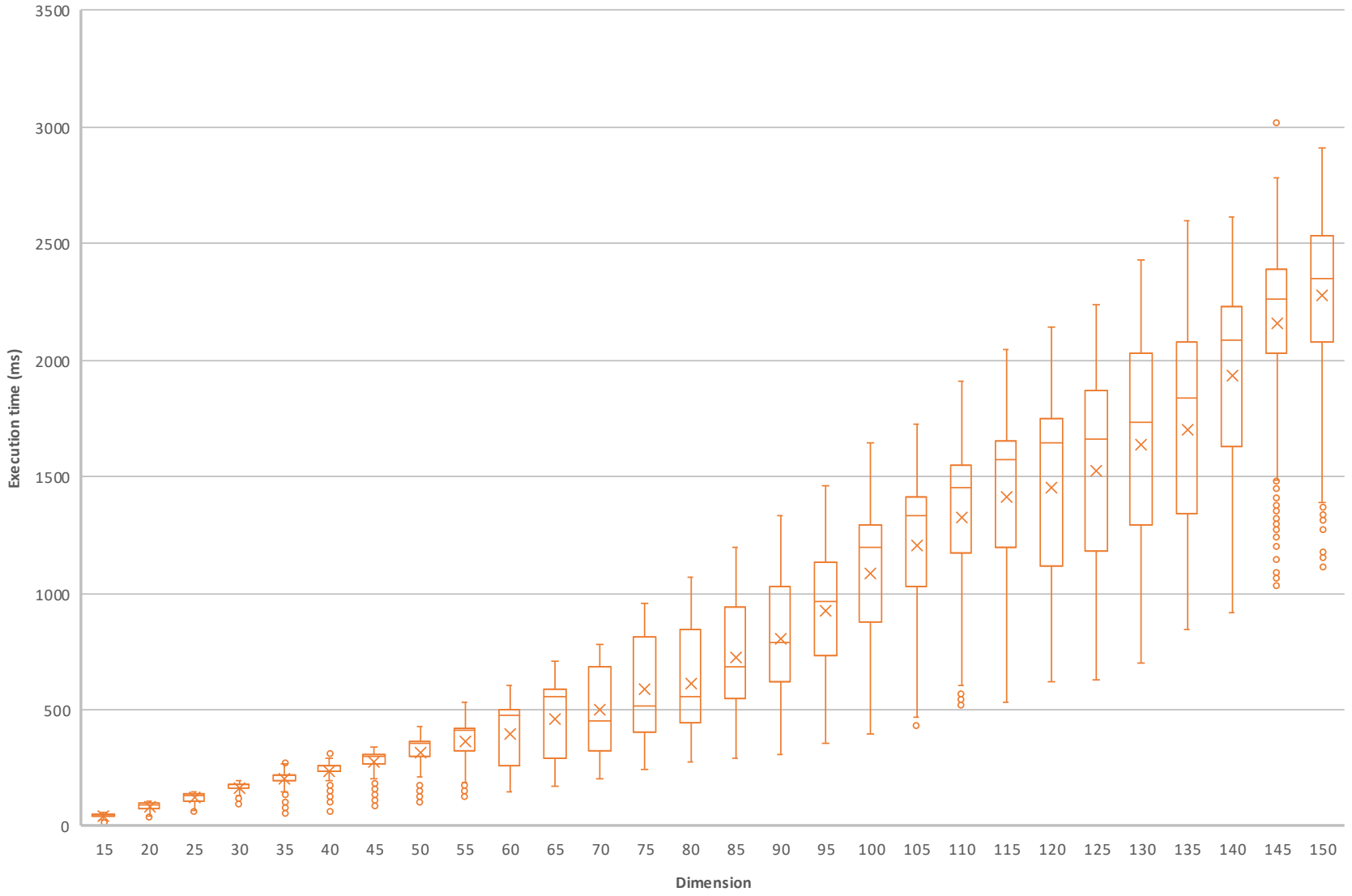
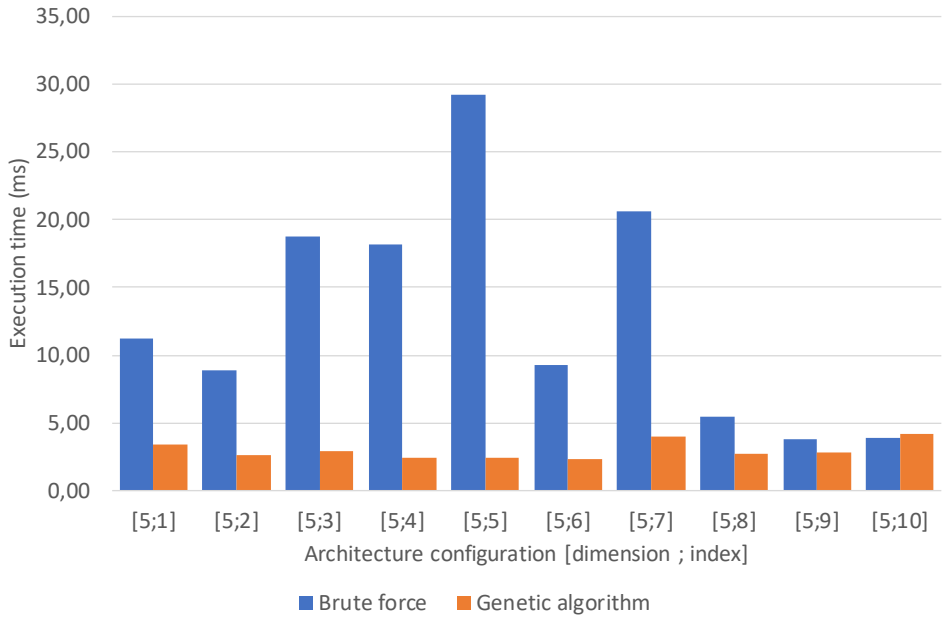
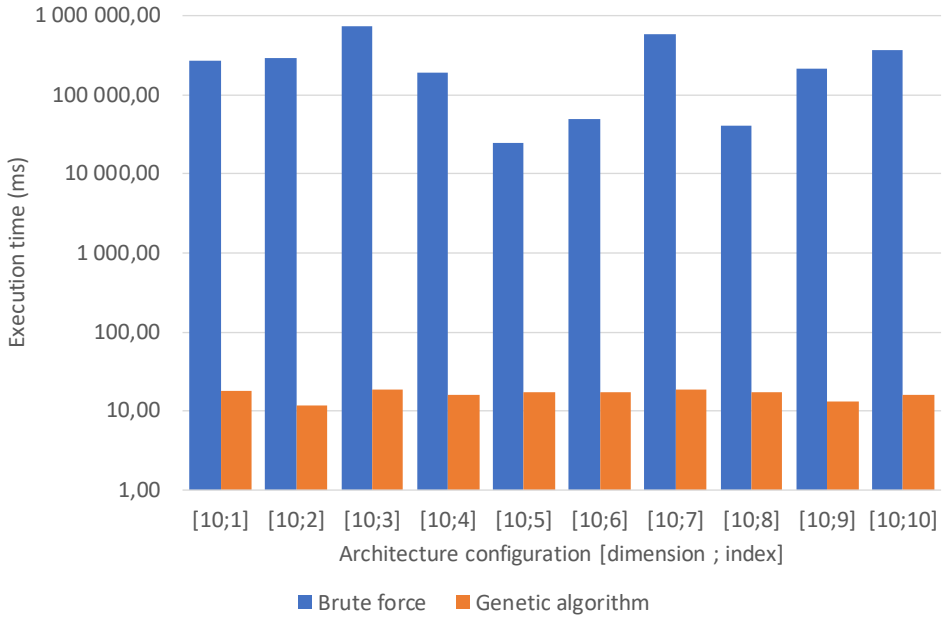


Figure 5.8 – Execution time of the genetic algorithm for dimension 15 to 150 (steps of 5).



(a) Dimension of 5



(b) Dimension of 10 (logarithmic scale)

Figure 5.9 – Execution time comparison between Genetic Algorithm and Brute Force

5.2.2.4 Quality of the solution found

Even if the genetic algorithm has been demonstrated to be faster than the brute force, we need to evaluate the quality of the solution found. Meta-heuristics approach tends to find near-optimal solutions and not the best solution of the problem.

We compared the result found by the genetic algorithm with the optimal one. We were able to validate this with only the dimensions where the brute force algorithm is applicable, dimensions $\{5, 6, \ldots, 10\}$. For those dimensions, and for the 10 iterations on the problem, the genetic algorithm always find the best solution (or an equivalent in term of fitness value). We start to see different values found in the algorithm in dimensions 20 and higher.

This demonstrates that the genetic algorithm finds an equivalent to the optimal solution for the given fitness function. We note that this fitness function creates a lot of chromosomes with the same fitness value. With a function having more disparity in the values, the genetic algorithm would not find the optimal solution so easily. Table 5.2 shows the summary of solutions found by the algorithms.

Dimension	**5**	**6**	**7**	**8**	**9**	**10**	**15**
Number of base model	10	10	10	10	10	10	10
Number of solutions found by Genetic Algorithm	10	10	10	10	10	10	10
Number of solutions found by Brute Force	10	10	10	10	10	10	NA
Fitness difference in %	0	0	0	0	0	0	NA

Table 5.2 – Comparison of solutions found by the genetic and brute force algorithms

Conclusion

This chapter presented how the planner is implemented with the genetic algorithm approach to optimize the system. This algorithm is using the data extracted from the semantic knowledge base in order to find out a placement of the software processes in the system. The decision of the algorithm is enhanced by the reasoning of the semantic representation, since all specific constraints of the software are solved by the reasoner. This collaboration enables a fast execution of the genetic algorithm that has been proven to be efficient.

This meta-heuristics approach has been validated with a set of evaluations. A comparison in term of execution time and quality of the solution has been performed between the genetic and a brute-force algorithm. The genetic algorithm has a faster execution time than a brute-force approach (exception for some really small size problems, where an optimized brute-force outperform it). Moreover, an equivalent solution to the optimal one has been found by the genetic algorithm for the dimensions where the brute-force is applicable, i.e., where the optimal solution is found.

This chapter presented the last components left from the MAPE-K loop, in the implementation of the management framework. It also displays the collaboration possible between a high-level description of the system, with the semantic representation and reasoning, and a meta-heuristic algorithm that is not semantic aware. It enables the extension of this framework for several applications for the management of IoT software processes.

The planner with the genetic algorithm approach and the evaluation are in submission.

In the next chapter, a conclusion of the thesis is given with a summary of the contributions from the previous chapters. Moreover, a discussion on the future work of short and long term is provided.

Conclusion and Future Work

Conclusion

This thesis aims at providing an autonomous approach based on semantic web technologies and checkpointing mechanisms in an IoT software system.

The increasing number of devices implies the need to multiply the number of gateways to handle their connection. Those gateways are hosting software processes that may experience failure or lack of resources. Moreover, the dynamicity of the environment leads to modifications to the state of the entities.

Detecting defective software in this distributed and complex infrastructure is difficult. We illustrate the diversity of applications in the IoT that implies a lot of possible scenarios that a management framework has to take care of.

First, some information and metrics from the managed system have to be gathered in order to determine its *health*. This is the role of the **Monitor** component of the MAPE-K loop. This is done through the use of a well-known protocol in IoT, LWM2M (a **device management protocol**).

To manage the processes, this thesis proposes the use of a migration mechanism. For this purpose, an evaluation of several mechanisms has been given, and we suggest the use of the checkpointing. This mechanism allows one to create checkpoints of running processes, serialized into a file. This file can be transfered to another machine and the process restarted in the same state as it was when checkpointed. Moreover, an autonomic approach, with the implementation of the MAPE-K loop, has been shown to be efficient to handle the management of complex systems.

The first contribution discusses the adaptations needed to fit the **checkpointing mechanism for the IoT**. Using DMTCP, we suggest several optimizations in order to use this mechanism in an optimized manner. On top of that, we propose a **scene mechanism** aiming at splitting the program data into separated checkpoint files. This **enables the use of large data in a constrained environment**. In fact, the IoT gateways hosting the software processes are low-powered and do not have large computing capabilities. This scene mechanism is coupled with a **semantic representation**, allowing one to describe the different scenes of the process and their specificities. With such description, a reasoner is able to determine when to change from a scene to another depending on the gateway environment.

The use of the DMTCP checkpointing mechanism on IoT processes has been evaluated and it was shown that the overhead of DMTCP is low. The RAM overhead is about $2,6$ MB and does not vary with application inputs. Moreover, the computation cost depends on the number of system calls, since they are intercepted by the DMTCP library and plugins to provide the correct virtualization. With the use of the scene mechanism, this computation overhead is about $2,4$ % slower with DMTCP. Additionally, the improvement by using the forked checkpointing and fast restart has been evaluated. We showed that the forked checkpointing does not freeze

the process for a long time (less than 250 ms), and forked checkpointing allows the application to resume much faster than otherwise. Moreover, we demonstrated that a standard restart using checkpointing is 25 times fast than the initialization time of the process. The fast restart optimization of DMTCP brings this optimization to 500 times faster than the standard initialization. This work has been presented in [Aïssaoui 2016a].

In the autonomic approach, the **Executor** component assumes the role of interacting with the system to change its behavior.

The second contribution focuses on the representation of the IoT software infrastructure. Indeed, a model is required to be used in the **knowledge base** of the autonomic loop. The **proposed ontology aims at representing the IoT software infrastructure** with entities such as the devices, the software entities or the machines where the processes are executed. Moreover, the functional requirements of the software entities can be expressed through several object and data properties described in the vocabulary. On top of that, the definition of the possible symptoms are defined with the requests for change.

The role of the **Analyzer** is to **determine the symptoms of the system and the RFCs to apply**. For this purpose, this thesis proposes the usage of SWRL rules. This system allows the system manager to define high-level rules depending on the application, which will be stored alongside the model. This thesis evaluates this approach with an instance of the ontology and the rules on a logistics scenario. This ontology with the rule system and the evaluation of the logistics scenario have been published in [Aïssaoui 2017].

With the result of the analyzer, the **Planner** has to **define a plan of actions** that will be performed by the executor. This task corresponds to **finding a new software entity placement** of the set of available machines. This new placement needs to **fulfill the functional requirements** of the software entities while **not overloading the machines in term of resource usage**. Moreover, several parameters can be optimized at the same time, such as the energy consumption, to enhance the system operation. Thus, the definition of the problem leads to a combinatorial set of possible solution and finding the best solution is a difficult task. Therefore, the use of a **meta-heuristics** is proposed by this thesis. More precisely, a **genetic algorithm** has been used to find a near-optimal solution. This algorithm is commonly used in placement problems in cloud computing.

The genetic algorithm uses a simplified description of the system to be aware of the available resources and the functional constraints of the software entities. This description is extracted from the semantic knowledge base alongside the current state of the system. Using this information, it generates a certain amount of placement and evaluates it with a fitness function. The number of generated chromosomes scales with the number of problems. Moreover, a stop criterion halts the execution of the algorithm when no new minimum is found, resulting in a faster execution time.

In comparison with a brute-force algorithm, we demonstrated that the genetic algorithm is able to find solutions for much larger problems than by using brute-

force. While brute-force stops functioning at a dimension of 15 software entities, the genetic algorithm is able to find a solution in about $2,5$ seconds even for a dimension of 150. We did not evaluate at still higher dimensions, since the definition of the problem would lead to an impossible solution, due to the gateways beginning to be overloaded.

In the dimensions where the brute-force is able to find a solution (from 5 to 10), this thesis demonstrates that the genetic algorithm finds an equivalent of the optimal solution.

All of the above contributions used together create the autonomic loop and enables the management of software entities in an IoT context.

Future Work

Short-term work

Semantic inference enhancements As we demonstrated in the evaluation of Chapter 4, the semantic inference system based on SWRL is not extremely efficient. Indeed, the execution time can be quite long when the problem gets larger. However, this complexity is not that high compared to real deployment of IoT systems. For this purpose, we suggest to evaluate more up to date technologies for the future, such as SHACL rule system. This semantic inference system is standardized and will imply more optimization focus from the semantic web community, compared to SWRL which is just a submission to the W3C.

Genetic algorithm improvements The genetic algorithm has been proven to be efficient in finding new software entities placement. An enhancement of the algorithm could use a local search. Indeed, this approach enforce the algorithm to look "around" the solutions found in order to find a new local minimum. This improvement has been suggested in several contributions [Dengiz 1997, Ishibuchi 1998] and could help the decision in our approach.

The currently used fitness function in the genetic algorithm is based on two factors, the cost of the migration and an evaluation of the new placement. This definition may be altered in order to adapt the placement to new policies, depending on the applications needs. This dynamic definition of the fitness function corresponds to more multi-objectives is an interesting aspect to integrate in this work. Since the IoT is a highly dynamic domain, the possibility to influence the new placement depending on new parameters is important.

Long-term work

Other possible IA approaches This thesis uses semantics web technologies and meta-heuristics algorithm as IA approaches to solve the problem. This contribution does not include prediction of the environment. With some literature models, such as machine learning, it could be possible to anticipate the events that will lead to

new issues in the system. Moving the software processes in advance could have a positive impact on the operation of the system, leading to a higher update and error proof software infrastructure.

Integration of the Cloud infrastructure The Cloud infrastructure has been mentioned in the context of this thesis as the higher layer of the IoT architecture. Indeed, this infrastructure composed of machines with high computation capabilities could be used in several scenarios. For instance, if we take in consideration gateways that are powered by a solar panel, they may experience energy down time depending on the production. During this down time, one needs to ensure that the service provided by the gateway are still available. Therefore, migrating the service to a Cloud virtual machine could replace the service on the gateway while the energy production is not sufficient.

Evaluation on real deployment The current evaluation of this thesis takes each component one by one to determine their performance. In order to have a better idea of the efficiency of the contribution, a deployment on a real case would enable a better evaluation of the contribution. Moreover, the current architecture performs its iteration on the current data representing the state of the system. An interesting question is, when to perform the autonomic iteration? Regarding the evolution of the system, some changes may impact a large need of reconfiguration while some are minor. Defining a policy regarding the trigger of the iteration of the autonomic loop in a real deployment is an interesting topic.

Bibliography

[Aïssaoui 2016a] François Aïssaoui, Gene Cooperman, Thierry Monteil and Saïd Tazi. *Smart Scene Management for IoT-based Constrained Devices Using Checkpointing*. In 15th IEEE International Symposium on Network Computing and Applications, 2016. (Cited in pages 40 and 90.)

[Aïssaoui 2016b] François Aïssaoui, Guillaume Garzone and Nicolas Seydoux. *Providing interoperability for autonomic control of connected devices*. In 2nd EAI International Conference on Interoperability in IoT, 2016. (Cited in pages 23 and 27.)

[Aïssaoui 2017] François Aïssaoui, Gene Cooperman, Thierry Monteil and Saïd Tazi. *Intelligent Checkpointing Strategies for IoT System Management*. In 5th IEEE International Conference on Future Internet of Things and Cloud, Pragues, 2017. (Cited in pages 54, 67, and 90.)

[Al-Fuqaha 2015] Ala Al-Fuqaha, Abdallah Khreishah, Mohsen Guizani, Ammar Rayes and Mehdi Mohammadi. *Toward better horizontal integration among IoT services*. IEEE Communications Magazine, vol. 53, no. 9, pages 72–79, 2015. (Cited in page 6.)

[Alaya 2015a] Mahdi Ben Alaya. *Towards interoperability, self-management, and scalability for machine-to-machine systems*. PhD thesis, Universite Toulouse III Paul Sabatier, 2015. (Cited in page 23.)

[Alaya 2015b] Mahdi Ben Alaya, Samir Medjiah, Thierry Monteil and Khalil Drira. *Toward semantic interoperability in oneM2M architecture*. IEEE Communications Magazine, vol. 53, no. 12, pages 35–41, 2015. (Cited in page 21.)

[Ansel 2009] Jason Ansel, Kapil Aryay and Gene Cooperman. *DMTCP: Transparent checkpointing for cluster computations and the desktop*. In 2009 IEEE International Symposium on Parallel & Distributed Processing, pages 1–12. IEEE, 5 2009. (Cited in pages 11, 16, 37, and 40.)

[Arya 2016] Kapil Arya, Rohan Garg, Artem Y Polyakov and Gene Cooperman. *Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization*. In IEEE Int. Conf. on Cluster Computing (Cluster'16), pages 402–412. IEEE Press, 2016. (Cited in page 12.)

[Atzori 2010] Luigi Atzori, Antonio Iera and Giacomo Morabito. *The Internet of Things: A survey*. Computer Networks, vol. 54, no. 15, pages 2787–2805, 2010. (Cited in pages 1 and 6.)

[Bali 2009] Michal Bali. Drools JBoss Rules 5.0 Developer's Guide. Packt Publishing Ltd, 2009. (Cited in page 23.)

[Barnaghi 2012] Payam Barnaghi, Wei Wang, Cory Henson and Kerry 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, pages 1–21, 2012. (Cited in page 20.)

[Berners-Lee 2001] Tim Berners-Lee, James Hendler and Ora Lassila. *The semantic web.* Scientific american, vol. 284, no. 5, pages 34–43, 2001. (Cited in page 17.)

[Bettini 2010] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan and Daniele Riboni. *A survey of context modelling and reasoning techniques.* Pervasive and Mobile Computing, vol. 6, no. 2, pages 161–180, 4 2010. (Cited in page 17.)

[Cao 2014] Jiajun Cao, Gregory Kerr, Kapil Arya and Gene Cooperman. *Transparent Checkpoint-restart over InfiniBand.* In Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing, pages 13–24. ACM Press, 2014. (Cited in page 10.)

[Cappello 2014] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer and Marc Snir. *Toward Exascale Resilience: 2014 Update.* Supercomputing Frontiers and Innovations, vol. 1, no. 1, pages 5–28, 2014. (Cited in page 10.)

[Chen 2014] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu and Hucheng Wang. *A vision of IoT: Applications, challenges, and opportunities with China Perspective.* IEEE Internet of Things Journal, vol. 1, no. 4, pages 349–359, 2014. (Cited in page 8.)

[Clark 2005] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield. *Live migration of virtual machines.* In Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 273–286. USENIX Association, 2005. (Cited in page 13.)

[Compton 2009] Michael Compton, Cory Andrew Henson, Laurent Lefort, Holger Neuhaus and Amit P Sheth. *A survey of the semantic specification of sensors.* International Conference on Semantic Sensor Networks, vol. 522, pages 17–32, 2009. (Cited in page 8.)

[Daniele 2016] Laura Daniele, Monika Solanki, Frank den Hartog and Jasper Roes. *Interoperability for smart appliances in the iot world.* In International Semantic Web Conference, pages 21–29. Springer, 2016. (Cited in page 20.)

[Davis 1991] Lawrence Davis. *Handbook of genetic algorithms.* 1991. (Cited in page 69.)

[De Paola 2014] Alessandra De Paola. *An Ontology-Based Autonomic System for Ambient Intelligence Scenarios*. In Advances in Intelligent Systems and Computing, pages 1–17. 2014. (Cited in page 20.)

[Dengiz 1997] Berna Dengiz, Fulya Altiparmak and Alice E Smith. *Local search genetic algorithm for optimal design of reliable networks*. IEEE Transactions on Evolutionary Computation, vol. 1, no. 3, pages 179–188, 1997. (Cited in page 91.)

[Desai 2015] Pratikkumar Desai, Amit Sheth and Pramod Anantharam. *Semantic Gateway as a Service Architecture for IoT Interoperability*. In 2015 IEEE International Conference on Mobile Services, pages 313–319. IEEE, 6 2015. (Cited in pages 8 and 20.)

[Fonseca 1993] Carlos M Fonseca, Peter J Fleming and others. *Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization.* In Icga, volume 93, pages 416–423, 1993. (Cited in page 69.)

[Ghit 2017] Bogdan Ghit and Dick Epema. *Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks*. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pages 105–116. ACM, 2017. (Cited in page 13.)

[Gorlatova 2014] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis and Gil Zussman. *Movers and shakers: Kinetic energy harvesting for the internet of things*. In ACM SIGMETRICS Performance Evaluation Review, volume 42, pages 407–419. ACM, 2014. (Cited in page 7.)

[Gruber 1991] Thomas R Gruber. *The role of common ontology in achieving sharable, reusable knowledge bases*. KR, vol. 91, pages 601–602, 1991. (Cited in page 17.)

[Gyrard 2014] Amelie Gyrard, Soumya Kanti Datta, Christian Bonnet and Karima Boudaoud. *Standardizing generic cross-domain applications in Internet of Things*. In Globecom Workshops (GC Wkshps), 2014, pages 589–594. IEEE, 2014. (Cited in page 8.)

[Hachem 2011] Sara Hachem, Thiago Teixeira and Valérie Issarny. *Ontologies for the Internet of Things*. ACMIFIPUSENIX 12th International Middleware Conference, no. June 2009, page 3:1–3:6, 2011. (Cited in page 19.)

[Hargrove 2006] Paul Hargrove and Jason Duell. *Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters*. Journal of Physics Conference Series, vol. 46, pages 494–499, 2006. (Cited in page 10.)

[Hashmi 2014] Khayyam Hashmi, Erfan Najmi, Zaki Malik, Brahim Medjahed, Amal Alhosban and Abdelmounaam Rezgui. *Automated negotiation using*

semantic rules. Proceedings - 2014 IEEE International Conference on Services Computing, SCC 2014, pages 536–543, 2014. (Cited in page 19.)

[Horn 2001] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Technical Report, IBM, 2001. (Cited in page 21.)

[Horrocks 2004] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean and others. *SWRL: A semantic web rule language combining OWL and RuleML*. W3C Member submission, vol. 21, page 79, 2004. (Cited in page 19.)

[Ishibuchi 1998] Hisao Ishibuchi and Tadahiko Murata. *A multi-objective genetic local search algorithm and its application to flowshop scheduling*. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 28, no. 3, pages 392–403, 1998. (Cited in page 91.)

[Ismail 2015] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke and Ong Hong Hoe. *Evaluation of docker as edge computing platform*. In Open Systems (ICOS), 2015 IEEE Confernece on, pages 130–135. IEEE, 2015. (Cited in pages 14 and 16.)

[K. Arya 2016] K. Arya, R. Garg, A. Y. Polyakov and G. Cooperman. *Design and Implementation for Checkpointing of Distributed Resources using Process-level Virtualization*. In Proc. of 2016 IEEE Computer Society International Conference on Cluster Computing. IEEE Press, 2016. (Cited in page 38.)

[Kamalinejad 2015] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor C M Leung and Yong Liang Guan. *Wireless energy harvesting for the Internet of Things*. IEEE Communications Magazine, vol. 53, no. 6, pages 102–108, 2015. (Cited in page 7.)

[Kephart 2003] Jeffrey O Kephart and David M Chess. *The Vision of Autonomic Computing*. Computer, vol. 36, no. 1, pages 41–50, 2003. (Cited in page 21.)

[Keville 2012] Kurt L Keville, Rohan Garg, David J Yates, Kapil Arya and Gene Cooperman. *Towards Fault-tolerant Energy-efficient High Performance Computing in the Cloud*. In 2012 IEEE International Conference on Cluster Computing, pages 622–626. IEEE, 2012. (Cited in page 10.)

[Kim 2015] Seong-Min Kim, Hoan-Suk Choi and Woo-Seop Rhee. *IoT home gateway for auto-configuration and management of MQTT devices*. In Wireless Sensors (ICWiSe), 2015 IEEE Conference on, pages 12–17. IEEE, 2015. (Cited in page 8.)

[Litzkow 1997] Michael Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny. *Checkpoint and Migration of UNIX processes in the Condor Distributed*

Processing System. Technical Report, Technical Report, 1997. (Cited in page 10.)

[Liu 2015] Jiaqiang Liu, Yong Li, Min Chen, Wenxia Dong and Depeng Jin. *Software-defined internet of things for smart urban sensing.* IEEE communications magazine, vol. 53, no. 9, pages 55–63, 2015. (Cited in page 8.)

[Ma 2011] Hua Dong Ma. *Internet of things: Objectives and scientific challenges.* Journal of Computer Science and Technology, vol. 26, no. 6, pages 919–924, 2011. (Cited in page 8.)

[Milojičić 2000] Dejan S Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler and Songnian Zhou. *Process migration.* ACM Computing Surveys (CSUR), vol. 32, no. 3, pages 241–299, 2000. (Cited in page 10.)

[Naksinehaboon 2008] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, Stephen L Scott and others. *Reliability-aware approach: An incremental checkpoint/restart model in hpc environments.* In 2008 8th International Symposium on Cluster Computing and the Grid (CCGRID, pages 783–788. IEEE, 2008. (Cited in page 13.)

[Osman 2002] Steven Osman, Dinesh Subhraveti, Gong Su and Jason Nieh. *The design and implementation of Zap: A system for migrating computing environments.* ACM SIGOPS Operating Systems Review, vol. 36, no. SI, pages 361–376, 2002. (Cited in page 13.)

[Pan 2011] Gang Pan, Yuqiong Xu, Zhaohui Wu, Shijian Li, Laurence Yang, Man Lin and Zhong Liu. *TaskShadow: toward seamless task migration across smart environments.* IEEE Intelligent Systems, vol. 26, no. 3, pages 50–57, 2011. (Cited in page 15.)

[Perumal 2015] Thinagaran Perumal, Soumya Kanti Datta and Christian Bonnet. *IoT device management framework for smart home scenarios.* In Consumer Electronics (GCCE), 2015 IEEE 4th Global Conference on, pages 54–55. IEEE, 2015. (Cited in page 8.)

[Salehi 2016] Mohammad Salehi, Mohammad Khavari Tavana, Semeen Rehman, Muhammad Shafique, Alireza Ejlali and Jörg Henkel. *Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 7, pages 2426–2437, 2016. (Cited in page 12.)

[Serrano 2015] Martín Serrano, Payam Barnaghi, Francois Carrez, Philippe Cousin, Ovidiu Vermesan and Peter Friess. *Internet of Things IoT Semantic Interoperability: Research Challenges, Best Practices, Recommendations and Next Steps EUROPEAN RESEARCH CLUSTER ON THE INTERNET OF THINGS.* Technical Report, 2015. (Cited in page 20.)

[Seydoux 2016a] Nicolas Seydoux, Khalil Drira, Nathalie Hernandez and Thierry Monteil. *Autonomy through knowledge: how IoT-O supports the management of a connected apartment.* In Semantic Web Technologies for the Internet of Things (SWIT). CEUR-WS, 2016. (Cited in page 23.)

[Seydoux 2016b] Nicolas Seydoux, Khalil Drira, Nathalie Hernandez and Thierry Monteil. *IoT-O, a core-domain IoT ontology to represent connected devices networks.* In European Knowledge Acquisition Workshop, pages 561–576. Springer, 2016. (Cited in page 21.)

[Sheth 2008] Amit Sheth, Cory Henson and Satya S. Sahoo. *Semantic Sensor Web.* IEEE Internet Computing, vol. 12, no. 4, pages 78–83, 7 2008. (Cited in page 20.)

[Sheu 2010] Phillip C Y Sheu, Heather Yu, C. V. Ramamoorthy, Arvind K. Joshi and Lotfi A. Zadeh. Semantic Computing. John Wiley and Sons, 2010. (Cited in page 17.)

[Szilagyi 2016] Ioan Szilagyi and Patrice Wira. *Ontologies and Semantic Web for the Internet of Things-a survey.* IECON, IEEE, 2016. (Cited in page 8.)

[Whitmore 2015] Andrew Whitmore, Anurag Agarwal and Li Da Xu. *The Internet of Things — A Survey of Topics and Trends.* Information Systems Frontiers, vol. 17, no. 2, pages 261–274, 2015. (Cited in page 8.)

[Wu 2012] Grant Wu, Maolin Tang, Yu-Chu Tian and Wei Li. *Energy-efficient virtual machine placement in data centers by genetic algorithm.* In Neural Information Processing, pages 315–323. Springer, 2012. (Cited in page 70.)

[Yashiro 2013] Takeshi Yashiro, Shinsuke Kobayashi, Noboru Koshizuka and Ken Sakamura. *An Internet of Things (IoT) architecture for embedded appliances.* 2013 IEEE Region 10 Humanitarian Technology Conference, R10-HTC 2013, pages 314–319, 2013. (Cited in page 8.)

[Yokotani 2016] Tetsuya Yokotani and Yuya Sasaki. *Comparison with HTTP and MQTT on required network resources for IoT.* In Control, Electronics, Renewable Energy and Communications (ICCEREC), 2016 International Conference on, pages 1–6. IEEE, 2016. (Cited in page 6.)

[Yusoh 2010] Zeratul Izzah Mohd Yusoh and Maolin Tang. *A penalty-based genetic algorithm for the composite SaaS placement problem in the cloud.* In Evolutionary Computation (CEC), 2010 IEEE Congress on, pages 1–8. IEEE, 2010. (Cited in page 70.)

[Zanella 2014a] a Zanella, N. Bui, a Castellani, L. Vangelista and M. Zorzi. *Internet of Things for Smart Cities.* IEEE Internet of Things Journal, vol. 1, no. 1, pages 22–32, 2014. (Cited in page 8.)

[Zanella 2014b] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista and Michele Zorzi. *Internet of things for smart cities*. IEEE Internet of Things journal, vol. 1, no. 1, pages 22–32, 2014. (Cited in page 7.)

[Zhang 2010] Qi Zhang, Lu Cheng and Raouf Boutaba. *Cloud computing: state-of-the-art and research challenges*. Journal of internet services and applications, vol. 1, no. 1, pages 7–18, 2010. (Cited in page 13.)

[Zhu 2010] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu and Weijun Qin. *Iot gateway: Bridgingwireless sensor networks into internet of things*. In Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on, pages 347–352. Ieee, 2010. (Cited in page 8.)

Abstract:

The Internet of Things (IoT) has to handle more and more connected, communicating and moving devices. The software infrastructure needs to insure a set of parameters to keep the system in a correct state. This infrastructure comprises a set of software processes executed on complex hardware platforms such as servers, gateways or things. The dynamic property of the IoT requires a perpetual adaptation and reconfiguration of the software infrastructure. In this thesis, we propose the usage of another migration mechanism: the "checkpointing" mechanism on both the servers and the gateways. This mechanism is light and able to store the software process state during the migration. The problem addressed by the thesis is to use this checkpointing mechanism in an efficient and autonomous way to preserve the properties expected from an IoT software infrastructure. A first contribution discusses the optimization of the checkpointing mechanism on an IoT gateway. A second contribution provides an autonomic and semantic approach to orchestrate the checkpointing mechanism. A third contribution discusses the optimization performed by a meta-heuristic algorithm on the software distribution. The contributions presented have been validated on several use cases for the IoT including optimization of software processes placement depending on the computing and energy capacity of IoT equipment in a logistic scenario.

Keywords: Internet of Things, Semantics, Ontology, Software process migration, Checkpointing mechanism, Genetic algorithm, Autonomic management

---

Résumé :

L'Internet des Objets intègre de plus en plus d'objets connectés, communicants et mobiles. L'infrastructure logicielle qui doit être déployée pour connecter ces objets et traiter leurs données doit répondre à différents critères. La nature dynamique de l'IoT nécessite une adaptation et une reconfiguration de cette infrastructure logicielle en cas de changement. Dans ce travail de thèse, nous proposons l'utilisation du mécanisme de « checkpointing » permettant aussi de conserver l'état des processus lors du déplacement. La problématique abordée dans cette thèse est comment utiliser ce mécanisme de checkpointing de manière efficace et autonome pour conserver les propriétés de l'infrastructure logicielle. Une première contribution concerne l'optimisation du checkpointing pour les équipements de l'Internet des Objets. La deuxième contribution concerne l'utilisation d'une approche autonomique et sémantique pour orchestrer les mécanismes de checkpointing. La troisième contribution concerne l'optimisation de la répartition des précessus par un algorithme de meta-heuristique. L'ensemble de ces contributions est validé dans différents cas d'usage de l'IoT tel que l'optimisation du déploiement des processus sous des contraintes de capacité de calcul des équipements, de mémoire ou de consommation énergétique dans un scénario de logistique.

Mots clés :

Internet des Objets, Sémantique, Ontologie, Migration de processus, Mécanisme de Checkpointing, Algorithme génétique, Gestion autonomique