Smart Scene Management for IoT-based Constrained Devices Using Checkpointing

François Aïssaoui, Gene Cooperman, Thierry Monteil, Saïd Tazi

Smart Scene Management for IoT-based Constrained Devices Using Checkpointing

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

***Abstract*—Typical devices of the Internet of Things are usually under-powered, and have limited RAM. This is due to energy and cost concerns. Yet, IoT applications require increasingly complex programs with increasingly large amounts of data. In principle, an application could manage the increasing data within the limited RAM by saving and loading data from the file system as needed. But managing the use of RAM in this way is both time-consuming and error-prone for the code developer. We propose instead a novel architecture in which different semantic scenes are implemented as independent operating system processes. As the need arises to switch from one scene to another, the currently running process, which represents the current scene, is checkpointed and a process representing the new scene is restarted from a checkpoint image. This solution employs checkpointing to provide a simpler framework for the end programmer, while at the same time resulting in higher performance. For example, experiments show that restarting an old process from a checkpoint image is about 25 times faster than starting a new process. When using an mmap-based optimization (deferring the paging in of virtual memory pages until runtime), restarting an old process is about 500 times faster. Overall, checkpoint and restart each execute in less than 0.2 seconds on a Raspberry Pi B.**

***Index Terms*—Semantics, Scene Management, Advances Driver Assistance System, Internet of Things, Checkpointing, Constrained Devices**

I. Introduction

The Internet of Things (IoT) is fast becoming a critical technology in the evolution toward a "connected world". However, IoT faces some challenges in terms of optimization on performance-constrained devices and gateways. The CPUs are of low performance, with minimal associated RAM. This is intrinsic to many IoT scenarios, such as Advanced Driver Assistance Systems (ADAS: "smart cars"), drones, etc., which are constrained by limited battery size and by cost concerns.

In many applications, IoT software is consuming increasing amounts of memory, due in part to increasingly complex behavior with increasingly complex semantics as software requirements continue to evolve. Virtual memory is usually not available, since IoT devices involve real-time computation, and paging in virtual memory makes running times difficult to predict. So, the central question is:

> *how to write a large program with large data that does not naturally fit in the small RAM available.*

We propose a novel software architecture that is well-adapted to RAM-constrained devices for IoT. Specifically, we will target a simplistic model of Advanced Driver Assistance System (ADAS) in order to illustrate the architectural benefits of easier scene-management for the end programmer and improved performance through efficient checkpoint-restart. We implement this model on the ARM-based Raspberry Pi Model B computer, as an example of a low-performance CPU configuration with limited RAM.

The proposed software architecture consists of smart scene management, using semantic modelling and rule-based reasoning. Each scene is represented by an operating system process, and a checkpointing mechanism is applied to save the state of the process in a checkpoint image file.

Such a scene represents a partial view of the context. It contains information about the spatial context, the road conditions, the participants, etc. Since the memory for a single scene can be huge, one typically does not have sufficient RAM to load two scenes at the same time within small or embedded computers such as the Raspberry Pi. The novelty here consists of using checkpoint-restart to manage RAM by saving the old scene and restore a new scene. This performs better than writing a monolithic program that includes all scenes, which would have to rely on the random memory access of virtual memory to page in the memory of scenes on demand.

Semantic web technologies such as the Web Ontology Language (OWL) formalism allow one to represent knowledge using a description logic. It allows for the creation of vocabularies that can be shared, along with a set of rules to apply to the model using *semantic reasoners*. We propose a new model for Scene management, with a specific set of properties and a possible link to application model such as ADAS.

The remainder of the paper is organized as follows. Section II presents a brief overview of semantic web principles and technologies in the context of the IoT and some background on checkpointing. In Section III, the presentation of a Ideal Global Architecture of our Scene system is provided, presenting the models and rules used. Section IV presents the specific contributions of our work. Section V analyzes the experiment results. Then we conclude this paper in Section VI.

II. Background

This section provides an overview of the literature for the technologies employed in this paper. A brief overview of IoT is provided first. This is followed by a discussion of the requirements for semantics for IoT. Finally, we dis-

cuss checkpointing and introduce Distributed Multi-Threaded CheckPointing (DMTCP).

A. IoT Overview

The IoT is an important area for innovation due to the large numbers of possible applications [1]. This presents a vision of a world-wide network of interconnecting physical (sensors, activators, complex objects like cars). The vision also includes virtual objects able to interact and affect the real world create a significant number of challenges [2]. In most cases, these objects have strong constraints in term of energy, communication and/or processing [3].

B. Semantic Web Technologies for IoT

Semantic computing [4] 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 (software or hardware). It is a recommended best practice in the domain of IoT [5]. However, one of the problems facing users of semantic technologies is that the semantic information increases the complexity and processing time, and is therefore unsuitable for dynamic and responsive environments such as IoT. Complex models require greater CPU processing and therefore are not suitable for constrained environments such as IoT. The earlier proposal of the W3C [6] takes this difficulty into account by providing a lightweight ontology specially adapted for IoT.

We use the OWL formalism to represent the data and the associated knowledge. OWL is a description language based on linked data and share vocabularies. Semantic Web Rule Language (SWRL) is a rule-based language that describes what could happen when the knowledge base changes, or when an event happens. It allows one to express abstract rules to be applied in the model.

C. Checkpointing using DMTCP

A checkpointing mechanism consists in creating images (snapshots) of a process and being able to recreate the process from this image.

Checkpointing has a long history in HPC [7]–[10]. In 2012, a cluster of ARM CPUs was tested with respect to checkpointing as a basis for power-efficient HPC [11]. 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.

In Section V, DMTCP [12] is used to create checkpoint image files from running processes. DMTCP-style checkpointing is *transparent*, in that the original application binary is not modified, and the target process is not aware of DMTCP.

In this work, the latter approach is used to allow the application to change scenes on demand. The application program can be further modified through the use of DMTCP plugins [13]. Plugins are used to virtualize resources, so that the application can be restarted in a new environment, independently of changing physical names such as pathnames, process ids (PIDs), etc.

DMTCP also supports options for two well-known 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.

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.

III. Ideal Global Architecture: Scenes and Semantics

This section presents the semantic concepts associated with the Scene concept, along with the rules used to manage the Scenes.

A. Semantic Models Used

A Scene is defined as a partial view of the context. Several scenes are created according to the needs of the application. Only one Scene is loaded at any given time. The Scene includes the destination of the vehicle, along with a possible path. It also contains a partial representation of the map ontology used in [14]. Since the entire map is split among different scenes, this lowers the system complexity through rules to navigate from one scene to another.

Figure 1 shows a semantic model with some example relations in the semantic class Scene. The last relation of the Scene is its *Specificity*. This relation represents, for example, a location specificity, or a time-of-day specificity (e.g., day and night). This associates with the Scene specific characteristics that enable the reasoner to choose the best target scene to switch to.

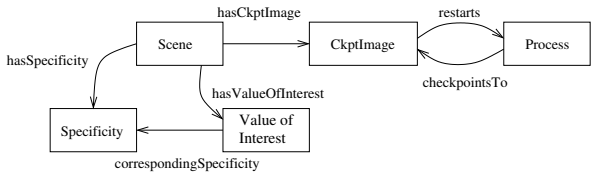

Fig. 1. Overview of Scene representation and link between Scenes and Checkpointing Mechanism

A second model that is linked to the Scene is used to guide the *Checkpointing*. Figure 1 shows the link to the Checkpointing model classes. The class *ValueOfInterest* is used to

characterize the values that are important for other Scenes. They are linked to the *Specificity* class by a *corresponding-Specificity* property for that class. This property links a value to a specific type of scene. Thus, 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. 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).

B. 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.

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.

C. Rules for Scene Management

Two models are considered in Section III-A. Specific rules for each model are used and will be described.

The first model includes a set of rules that affects the vehicle and its actions. This model is used to analyze the car sensor data (e.g., its position). It will allow the system to react according to the current context.

The second rule-based model from Section III-A is used to guide the checkpointing mechanism for the scene management. This model is in charge of gathering enough information from the system to infer that a change of scene is required. In this case, the rules cause the process in charge of the scene management to checkpoint the current scene and then to load the second one.

D. 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 must then be passed to the second scene.

As described in Section III-B, 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.

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 II-C.

The information to share can be retrieved from the model using the property *hasValueOfInterest* of the Scene. This relation is shown in Figure 1.

IV. Efficient RAM Management for IoT and Embedded Systems

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.

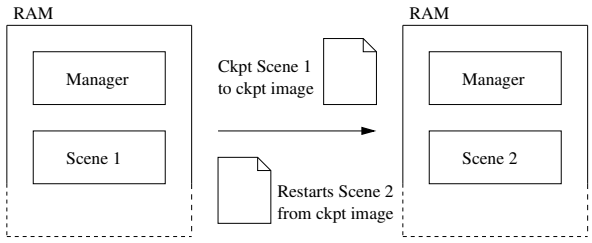

Fig. 2. Proposal of new architecture for Scene Management. Each rectangle represents a process.

Figure 2 illustrates the proposed architecture. The data to be handled is split into multiples scenes, which contain information and rules as described in Section III-A. 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 and is evaluated in the next section. We demonstrate the efficiency of such a system compared to a standard initialization of a process.

V. Experimental Evaluation

In this section we evaluate the system presented in Section III and Section V. 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. Finally, we discuss the runtime overhead introduced when the process is executed under the control of DMTCP, as opposed to executing the process natively.

A. Experimental Environment

These experiments use a *Raspberry Pi 2 Model B* with one 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.

For the checkpointing software, DMTCP version 3.0.0 was used, available at its github repository.

B. 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.

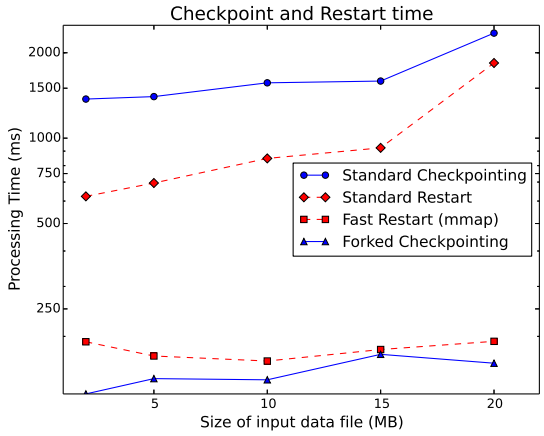

Fig. 3. Checkpoint and restart time

Two sets of experiments are discussed. First the standard checkpointing and restart mechanism is used. In Figure 3, 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 (see Section II-C) 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. The largest file used is about 20 MB, which serves as a placeholder for the actual scene-related data that would be used in a realistic ADAS application. 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 3 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 3) shows the impact of using the two DMTCP optimizations discussed in Section II-C: 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 3 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 3. 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 I
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 I 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.

C. 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 4 shows the execution times in different situations. The diamond-shaped and square plotted points represent the restart times for a checkpoint image. The square plot uses the

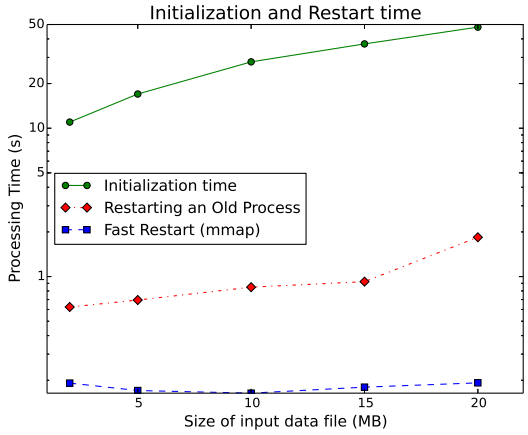

Fig. 4. Initialization of a new process versus restart of a previously checkpointed process. This compares the time for restarting a new process using the techniques of this work, versus the traditional alternative of starting (initializing) a new process for each new scene. Restarting an old process is about 25 times faster (and 500 times faster when using the mmap-based fast restart optimization). This is because restart avoids any data initialization that is executed by the Scene framework itself before it gives control to the end programmer.

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 of the ADAS framework. 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 4 shows that "Restarting an Old Process" is about 25 times faster than the standard execution startup of a new process in the ADAS framework. Further, the Fast Restart method is about 500 times faster than the standard initialization.

VI. Conclusion and Future Work

A new software architecture was presented that allows one to manage the RAM usage efficiently for Internet of Things (IoT) devices, and more generally for performance-constrained devices. A mechanism is used to checkpoint a process in order to make RAM available for a new process, which will be restarted from a checkpoint image file. A large, monolithic process would not be a good alternative, since the delays due to virtual memory paging are not consistent with real-time programming. We demonstrated that the proposed architecture is about 25 times faster than the standard startup of a new process (see Figure 4). 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. This work has been applied to an Advanced Driver Assistance Systems (ADAS) domain as an example, but the scene concept is generic and can be equally well applied to other problems in the Internet of Things.

This work has simulated the operating system characteristics and expected performance of an example scene-based architecture for ADAS as a proof-of-principle. The ADAS example itself is not intended as a realistic system for production. In future work, we will apply this to a full-fledged domain in the Internet of Things integrating management of the connectivity of multiple devices and real time constraints.

Acknowledgments

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

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] A. Whitmore, A. Agarwal, and L. Da Xu, "The Internet of Things — a survey of topics and trends," *Information Systems Frontiers*, vol. 17, no. 2, pp. 261–274, 2015.

[3] K.-C. Chen and S.-Y. Lien, "Machine-to-machine communications: Technologies and challenges," *Ad Hoc Networks*, vol. 18, pp. 3–23, 2014.

[4] P. C. Y. Sheu, H. Yu, C. V. Ramamoorthy, A. K. Joshi, and L. A. Zadeh, *Semantic Computing*. John Wiley and Sons, 2010.

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

[6] M. Bermudez-Edu *et al.*, "IoT-lite ontology, a submission to the W3C," Nov. 2015, https://www.w3.org/Submission/2015/SUBM-iot-lite-20151126/.

[7] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," Technical Report, Tech. Rep., 1997.

[8] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *Journal of Physics Conference Series*, vol. 46, pp. 494–499, Sep. 2006.

[9] J. Cao, G. Kerr, K. Arya, and G. Cooperman, "Transparent checkpoint-restart over InfiniBand," in *Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing*. ACM Press, 2014, pp. 13–24.

[10] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[11] K. L. Keville, R. Garg, D. J. Yates, K. Arya, and G. Cooperman, "Towards fault-tolerant energy-efficient high performance computing in the cloud," in *2012 IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 622–626.

[12] J. Ansel, K. Aryay, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, may 2009, pp. 1–12.

[13] 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, to appear.

[14] L. Zhao, R. Ichise, S. Mita, and Y. Sasaki, "Ontologies for Advanced Driver Assistance Systems," in *The 35th Semantic Web & Ontology Workshop (SWO)*, Mar. 2015, pp. 1–6, http://www.ei.sanken.osaka-u.ac.jp/sigswo/papers/SIG-SWO-035/SIG-SWO-035-03.pdf or http://sigswo.org/papers/SIG-SWO-035/SIG-SWO-035-03.pdf.