Checkpointing Facility on a Metasystem[⋆]

Yudith Cardinale and Emilio Hernández

Universidad Simón Bolívar,
Departamento de Computación y Tecnología de la Información,
Apartado 89000, Caracas 1080-A, Venezuela
{yudith,emilio}@ldc.usb.ve
http://suma.ldc.usb.ve

Abstract. A metasystem allows seamless access to a collection of distributed computational resources. Checkpointing is an important service in high throughput computing, especially for process migration and recovery after system crash. This article describes the experiences on incorporating checkpointing and recovery facilities in a Java-based metasystem. Our case study is SUMA, a metasystem for execution of Java bytecode, both sequential and parallel. This paper also shows preliminary results on checkpointing and recovery overhead for single-node applications.

1 Introduction

The access to distributed high performance computing facilities for execution of Java programs has generated considerable interest [3,1,12,7]. A metacomputing system, or metasystem, allows uniform access to heterogeneous resources, including high performance computers. This is achieved by presenting a collection of different computer systems as a single computer.

This work addresses some aspects related to the implementation of a checkpointing facility in a metasystem, with application to SUMA, a metasystem for execution of Java bytecode, both sequential and parallel. We compared several approaches for adding object persistence to the Java execution environment and selected the approach proposed in [2], which uses an extended Java Virtual Machine (JVM). We modified the architecture of SUMA for supporting remote checkpointing and recovery facility based on this extended JVM. This approach is semi-transparent at the user level, because the requirements for using this facility can be easily hidden. We obtained preliminary results on the overhead produced by the selected approach for single-node applications.

The rest of this document is organized as follows. Section 2 describes different approaches to implement persistent Java environments, as a base for Java checkpointing. Section 3 explains the implementation of the aforementioned services in SUMA. Section 4 presents the preliminary results of our research and section 5 presents the conclusions.

[⋆] This work was partially supported by grants from Conicit (project S1-2000000623) and from Universidad Simón Bolívar (direct support for research group GID-25)

2 Checkpointing Java on a Metasystem

Checkpointing involves capturing the state of a computation in terms of the data necessary to restart it from that state. An advantage of using Java, as far as checkpointing is concerned, is that checkpoints can be taken in a machine independent format. It is necessary to achieve Java object persistence for implementing architecture-independent Java checkpointing.

Several strategies have been proposed for adding persistence to the Java execution environment [4]. Some approaches are based on using language-level mechanisms through libraries [5,10]. Other approaches extend the JVM in order to make the computation state accessible from Java threads, which take the checkpoints [9,2]. Other approaches consist in running the whole JVM over an operating system that supports persistence or inserting a checkpointing layer between the JVM and a traditional operating system, as in [6].

All of the formerly described checkpointing approaches have their advantages and disadvantages. However, the implementation of a checkpointing facility in a metasystem requires some particular considerations. The selection of an appropriate checkpointing approach for a metasystem should take into account:

– Portability, due to the heterogeneity of the systems that comprise a metasystem. This consideration discards the approaches that save the machine state at the operating system level or the JVM level. From a portability viewpoint, the best options are those that keep the state of the computation in terms of Java objects, which can be restarted in another JVM.
– Transparency, which may also be a desirable feature for stand-alone JVM's. However, the need for efficient use of resources in a metasystem may be a reason for activating the checkpointing and recovery/migration services even if the user does not explicitly invoke them.
– Low intrusiveness of the checkpointing process, in terms of performance, especially if the metasystem design aims at high performance computing. Taking a checkpoint is itself a process that may consume a significant amount of time and should be optimized as much as possible. The checkpointing approaches based on adding instructions to the source code or bytecode have the risk of producing further performance reductions.

These aspects of metasystem design lead us to consider checkpointing approaches based on extending the JVM. We evaluated several projects that extend the JVM for implementing object persistence [11,8,9,2]. We are using the approach proposed in [2] because it provides a fine-grained Java thread state capture/restoration facility. This solution requires the checkpointable threads to be defined as extensions of the class "CapturableThread", contradicting the transparency requirement. However, it is potentially transparent because automatic preprocessing of the source code before compilation can easily be implemented.

3 Checkpointing and Recovering Services in SUMA

SUMA is intended to provide services for checkpointing and profiling, as an added value to on-line and off-line remote execution of Java applications. These services are mainly implemented within the Execution Agents, which are the SUMA components that actually execute the applications. On the client side, if a user wants to use the checkpointing service, under the current version she should program her applications based on threads that both extend the class CapturableThreads and implement the Serializable interface. More specifically, the programmer has only to include the statement "`import java.lang.threadpack.*`" and extend all threads from CapturableThreads.

Figure 1 shows the steps during the development and execution of an user application with checkpointing support on SUMA. On the client side there is no need to install the extended JVM. A stub package is provided for the programmer to compile the modified program. The code instrumentation on the client side can be done automatically, by using a preprocessor of source or byte code (step 1).

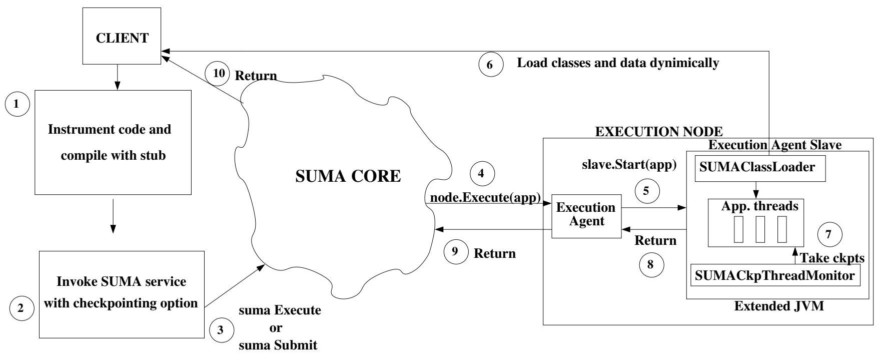

Fig. 1. Application execution on SUMA

The user requests the checkpointing service explicitly when submitting the application for execution in SUMA (step 2). This can be executed from a *suma-Client*, which can invoke either *sumaExecute* or *sumaSubmit* services. These services provide the checkpointing option.

After the SUMA client invokes the remote execution with the checkpointing option, SUMA builds an object that contains all information needed for application execution (step 3), finds a node with checkpointing support and sends the object to the Execution Agent at the selected node (step 4). Once this Execution Agent has received the object representing the application, it starts an Execution Agent Slave in a extended JVM (step 5). Two threads are started initially within this extended JVM: the SUMAClassLoader, whose function is to load classes and data from the client during the execution (it communicates with the client through CORBA callbacks), and the SUMACkpThreadMonitor, which will take the checkpoints. Then the main thread of the application is loaded directly from the client and started. The checkpoints will be taken from this moment on (step 7).

If the execution finishes successfully, the Execution Agent Slave informs the Execution Agent of this and finishes (step 8). The Execution Agent passes this information back to the SUMA core (step 9), which returns the results to the client (step 10). If the execution is interrupted, an exception is caught in the SUMA core, which launches the recovery process. The application will be restarted from the last checkpoint, typically in a different execution node.

4 Experimental Results

We conducted experiments to evaluate the checkpointing intrusiveness and the overhead of the recovery process. The platforms used in the experiments are several 143 MHz SUN Ultra 1 workstations connected through Ethernet. We executed a Java program ("Primes") to calculate the first "n" prime numbers. The prime numbers are saved in a vector, which means that the checkpoint size increases as the program progresses. Table 1 shows the time spent by the three last checkpoints and the recovery processes from these checkpoints. The time spent by the recovery processes is measured, by the SUMA core, between the point in which the failure exception is received and the point in which the "resume" service returns. The failure is simulated by killing the process. In all cases the application was restarted in a different node.

Table 1. Checkpointing and recovery overhead

Checkpoint No.	Checkpoint time	Recovery time	Checkpoint size
2	55.1 sec.	45 sec.	95KB
3	2 min. 5 sec.	1 min. 1 sec.	162KB
4	3 min. 8 sec.	1 min. 56 sec.	393KB

In this example every checkpoint is taken approximately two minutes after the previous one was saved. The overhead is currently very high, for instance, the case in which 4 checkpoints are taken incurs an overhead of about 40%. However, several improvements to this implementation can be done, such as the use of native threads instead of green threads. On the other hand, the number of checkpoints should be related to the probability of hardware failure.

5 Conclusions and Future Work

This work addressed some aspects related to the implementation of a checkpointing facility in a metasystem, with application to the SUMA metasystem.

This approach is almost transparent at the user level, because the requirements for using this facility can be easily hidden. This requirements are (1) modify the thread declarations, which can be done by a pre-processor just before compilation and (2) link locally with a stub that provides extensions for persistent threads. Future work will include the development of the aforementioned pre-processor.

The experiences shown in this article are limited to checkpointing of single-node applications on SUMA. Ongoing research is focusing on reducing performance overhead as well as incorporating a multiple-node (parallel) checkpointing algorithm, implemented with mpiJava. We are currently working on an Execution Agent for parallel checkpointing.

References

1. A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. *Future Generation Computer Systems*, 15(5–6):559–570, Octuber 1999. 75
2. S. Bouchenak. Making Java applications mobile or persistent. In *Proceedings of 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001. 75, 76
3. T. Brench, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the 7th ACM SIGOPS European Worshop*, 1996. 75
4. J. Eliot, B. Moss, and T. Hosking. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996. 76
5. S. Funfrocken. Transparent migration of Java-based mobile agents (capturing and reestablishing the state of Java programs). *Proceedings of Second International Workshop Mobile Agents 98 (MA'98)*, September 1998. 76
6. Jon Howell. Straightforward Java persistence through checkpointing. *In Advances in Persistent Object Systems*, pages 322–334, 1999. 76
7. Michael O. Neary, Bernd O. Christiansen, Peter Capello, and Klaus E. Schauser. Javelin: Parallel computing on the internet. *Future Generation Computer Systems*, 15(5–6):659–674, Octuber 1999. 75
8. J. Plank and M. Puening. Checkpointing Java. http://www.cs.utk.edu/˜plank/javackp.html. 76
9. T. Printezis, M. Atkinson, L. Daynes, S. Spence, and P. Bailey. The design of a new persistent object store for pjama. In *Proceedings of the Second International Workshop on Persistence and Java*, August, 1997. 76
10. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, 1(3):123–137, September 2000. 76
11. T. Suezawa. Persistent execution state of a Java Virtual Machine. *Proceedings of the ACM 2000 Java Grande Conference*, June 2000. 76
12. H. Takagi, S. Matsouka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: a migratable parallel object framework using Java. In *in Proc. of the ACM 1998 Worshop on Java for High-Performance Network Computing*, 1998. 75