Transparent Fault Tolerance for Grid Applications

Paweł Garbacki[1], Bartosz Biskupski[2], and Henri Bal[3]

[1] Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands
p.garbacki@ewi.tudelft.nl

[2] Department of Computer Science, Trinity College, Dublin, Ireland
biskupski@cs.tcd.ie

[3] Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
bal@cs.vu.nl

Abstract. A major challenge facing grid applications is the appropriate handling of failures. In this paper we address the problem of making parallel Java applications based on Remote Method Invocation (RMI) fault tolerant in a way transparent to the programmer. We use globally consistent checkpointing to avoid having to restart long-running computations from scratch after a system crash. The application's execution state can be captured at any time also when some of the application's threads are blocked waiting for the result of a (nested) remote method call. We modify only the program's bytecode which makes our solution independent from a particular Java Virtual Machine (JVM) implementation. The bytecode transformation algorithm performs a compile time analysis to reduce the number of modifications in the application's code which has a direct impact on the application's performance. The fault tolerance extensions encompass also the RMI components such as the RMI registry. Since essential data as checkpoints are replicated, our system is resilient to simultaneous failures of multiple machines. Experimental results show negligible performance overhead of our fault-tolerance extensions.

1 Introduction

Computational grids become increasingly important to solve computationally intensive and time consuming problems in science, industry, and engineering [3, 4, 21]. Since the failure probability increases with a rising number of components, fault tolerance is an essential characteristic of massively parallel systems. Such systems must provide redundancy and mechanisms to detect and localise errors as well as to reconfigure the system and to recover from error states.

Java's platform independence is well suited to a heterogeneous infrastructures that typify grid architectures. The object oriented nature of Java facilitates and code reuse significantly reduces development time. There is a wide variety of interfaces and language extensions that simplify parallel programming in Java not to mention Java threads and Remote Method Invocation (RMI) [2] which are

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 671–680, 2005.
© Springer-Verlag Berlin Heidelberg 2005

parts of the Java specification [10]. Despite all these facilities, the Java Virtual Machine (JVM) [12] standard does not support fault tolerance.

In this paper we present the design, implementation and evaluation of a system that provides coordinated checkpointing and recovery for RMI-based parallel Java applications with a focus on grid computing. Our approach is novel in that it was designed to be transparent to the programmer, requiring minimal changes to the application code.

The contributions of this paper are the following: first we design and implement a Java bytecode transformer that makes the application programs resilient to failures by means of checkpoint-recovery. The comprehensive compile time analysis significantly reduces the overhead incurred by the application code modifications. Second, we present a technique that allows checkpointing even inside (nested) remote method calls. Third, we provide mechanisms which enable RMI components to recover from a system crash in a completely transparent manner. Finally, we present performance results for classic parallel applications. As a yardstick, we compare the performance of Java applications with and without the fault tolerance extensions.

The rest of this paper is organized as follows. Section 2 lists the related work. Sections 3 and 4 describe the capturing and reestablishing of the state of a single process and the whole application, respectively. Section 5 introduces the subsystem responsible for handling checkpoints initiated inside remote method calls. Section 6 describes fault tolerant RMI. Section 7 presents the performance of our system. Finally, Section 8 concludes this paper.

2 Related Work

The importance of fault tolerance in grid computing has already been recognised by the establishment of the Grid Checkpoint Recovery Working Group [16]. Its purpose is to define user-level mechanisms and grid services for fault tolerance.

We are not aware of any other project that is specifically addressing the provision of transparent fault tolerance support for grid applications written in Java. There is, however, a considerable amount of work in various related areas. The problem of saving the complete state of a Java program was investigated in the context of mobile agents migration. The Nomads [17] and Sirac [5] projects modify the JVM to capture the execution state of the application, which makes them inappropriate for the heterogeneous grid environment where different machines may have different JVMs installed. The CIA [11] mobile agents platform uses Java Platform Debugger Architecture (JPDA) API [1] which is not compatible with most Just In Time (JIT) compilers and thus causes increased execution overhead. The last category includes projects that insert additional instructions to the application code. This is the case for the Brakes [8], JavaGo [15], and Wasp [9] projects.

3 Capturing and Reestablishing Local State

A grid application is composed of a set of (multithreaded) processes distributed over a number of nodes, usually on one process per node basis. A *local checkpoint* is then a snapshot of the local state of a process. A *global checkpoint* is a set of all local checkpoints saved on nonvolatile storage to survive system failures.

The process execution state consists of the Java stacks of all local threads. Each Java stack consists of frames that are associated with method invocations. A new frame is pushed onto the stack each time a method is invoked and popped from the stack when the method returns. A frame includes the local variables, the operand stack with partial results of the computations, and the program counter which indicates the next instruction to be executed.

A thread has access only to data stored in the currently executing method frame due to the strict Java security policies, and therefore there is no straightforward way to preserve the whole Java stack. In our system we use an approach similar to that proposed by the Brakes [8] project. The state capturing in Brakes is initiated explicitly by placing a checkpoint function call in the application source code. Brakes provides a post-compiler which instruments the application bytecode by inserting additional code blocks that do the actual capturing and reestablishing of the execution state. The Brakes bytecode transformer inserts a code block after every method invocation, which saves in the checkpoint the stack frame of the current method and returns control to the previous method on the stack. This process continues until the whole Java stack is saved. The process of reestablishing a thread's state is similar but restores the stack frames in reverse order. The bytecode transformer inserts code at the beginning of each method, which restores the stack frame of the current method and invokes the next method whose state is saved in the checkpoint, consequently restoring the next stack frame. The process continues until the whole stack is rebuilt.

A significant improvement that we made to the original Brakes algorithm is that we added the analysis of the methods call graph in order to detect and modify only these methods and method invocations that could lead to the execution state capturing. This modification has been proven (see Sect. 7) to dramatically decrease the number of rewritten methods and thus reduce the overhead caused by transforming the application. The methods call analyser finds a set of all methods in all user classes whose invocation can lead to state capturing. Initially, the set contains just one method — the internal method that directly initiates state capturing. In each iteration the algorithm adds to the set new methods that can invoke methods already in the set. The algorithm stops when no methods were added in the last iteration.

4 Capturing and Reestablishing Global State

The checkpointing mechanisms described in Sect. 3 apply only to threads running within the same address space. We extend the applicability of our checkpointing mechanisms to the class of distributed applications by using the *coordinated*

checkpointing scheme [18]. As its name suggests, in coordinated checkpointing all the threads running on different nodes have to synchronise before writing their states to stable storage. Global coordination guarantees that the saved state is automatically *globally consistent* [6].

The global thread coordination is performed in two phases. In the first phase threads are synchronised locally at each node. The goal of the second phase is to synchronise all nodes. The coordination of threads running on the same node is performed with the help of the *local coordinator*, a component deployed on each node. When a thread is ready for a global checkpoint it notifies its local coordinator. Once all local coordinators receive confirmation from all threads the distributed phase of the global coordination process begins. The distributed synchronisation algorithm proposed by us is based on software trees [19]. A tree-based barrier is known to provide excellent performance and to scale well to large machines [14]. Although this method was originally designed for multiprocessors, it can easily be adapted to a distributed environment.

In order to make our system not only resilient to software faults but also to hardware faults, we replicate the checkpointed data among different machines. This way even in a situation when a stable storage device on one of the nodes crashes, the checkpoint can be still retrieved from a remote location.

When a failure occurs, the processes that were running on the crashed nodes are restored from the latest checkpoint on the backup nodes. Each of the backup nodes runs a simple service capable of obtaining the local checkpoint of a crashed process and initiating its recovery (described in Sect. 3).

There are several situations in which the failure may be detected. First, the user application can explicitly invoke the scanning procedure that will check which nodes are down. Second, the failure may be detected during the global barrier synchronisation. Third, a crash of a remote object may be discovered by the fault-tolerant RMI described in Sect. 6. Finally, there is a dedicated thread running on every node that checks periodically whether all remote processes are up and running.

5 Capturing and Reestablishing Distributed Thread's State

To increase the level of programming transparency, we allow the programmer to initiate the state capturing at any stage of the program execution, also when some of the threads perform remote method calls. To our knowledge our approach is the first to manage this problem in a completely distributed way, without any central components. Before presenting our solution, we first explain why saving and restoring of a state of a thread performing a remote method call is challenging, and also introduce some terminology.

Since Java threads are bound to the virtual machine in which they were created, a remote method execution is mapped to two different Java threads: the *client thread* that initiated the call, and the *server thread* that executes the remote method on the remote node. These two threads are both representatives

for the same *distributed thread of control* [22, 7]. The checkpoint initiated inside a remote method call should contain the state of both server and client threads. In a local execution environment, the JVM thread identifier offers a unique reference for a single computation entity. Java, however, does not offer any interfaces that allow us to recognise two threads as parts of the same distributed thread. To cope with this problem we extend Java programs with the notion of *distributed thread identity* [22]. Propagation of globally unique identifiers allows for the identification of local computations as parts of the same distributed computation entity.

Using the introduced terminology we describe the idea of a distributed thread state capturing. Each local thread in our system has an associated identity of the distributed computation of which it is part. The identity is generated when the distributed computation is started, that is, when the oldest local thread which is part of the distributed computation is instantiated. The remote thread identity is sent along with the remote method call. It is done by extending the signature of the remote method with a parameter representing the distributed thread identity. Now suppose that the checkpoint was requested inside a remote method call. We start from capturing the state of the server thread which initiated the checkpoint. The context object containing the serialized state of the server thread is stored on the server machine under the key representing the distributed thread identity. After its state was captured, the server thread throws a special type of exception notifying the client thread that the checkpoint was requested. This procedure is repeated until the contexts of all local threads have been saved. The distributed thread state reestablishing is a reverse process.

6 Fault Tolerant RMI

A strong point of Java as a language for grid computing is the integrated support for parallel and distributed programming. Java provides Remote Method Invocation (RMI) [2] for transparent and efficient [13, 20] communication between JVMs. However, it does not have any support for fault tolerance. Therefore, we developed mechanisms that provide fault tolerance for the Java RMI components. We provide a replicated *RefStore* server that maintains remote objects references, mechanisms that allow remote objects and their stubs to transparently recover from a system crash, and a fault tolerant RMI registry.

The replicated RefStore server was developed for the purpose of storing remote references to recovered fault tolerant remote objects. The remote reference is a standard Java object (a component of every stub) containing the address of the hosting node, a communication port number, and a key that together uniquely identify the remote object. When a fault tolerant remote object recovers from a crash, it automatically registers its new remote reference in the RefStore server. When a stub cannot connect to the object using its old remote reference, it retrieves the new remote reference from the RefStore.

In order to release the programmer from the burden of detecting failures and updating remote references, a transformation algorithm that analyses a user's

stub classes and automatically generates exception handlers was developed. The exception handler is invoked when the stub cannot connect to the fault tolerant remote object. When it happens, the recovery process is initiated. After successful recovery, the stub automatically retrieves the new remote reference from the RefStore server and reconnects to the remote object using its new location.

The Java RMI registry [2] is typically used for exchanging stubs for remote objects. However, since the node on which the registry is running may also fail, we provide an implementation of a fault-tolerant RMI registry service that is checkpointed together with the whole application.

To summarize, each component of the fault-tolerant RMI system is either replicated or checkpointed, and so there is no single point of failure. Moreover, since no modifications in the application code are needed, our fault-tolerant RMI is completely transparent to the programmer.

7 Performance Evaluation

In this section we study the impact of our fault-tolerance extensions on the performance of the distributed applications. All tests were performed on the DAS2 cluster[1] of 1GHz Pentium III processors, running Linux, and connected by a Myrinet network.

We investigate the overhead incurred by the checkpointing extensions on two applications, namely Successive Over Relaxation (SOR) and Traveling Salesman Problem (TSP). These applications were selected as being challenging for the checkpointing system. Complicated control flow scheme and non-negligible amounts of temporary data pose difficulties for making these applications fault tolerant manually.

SOR is an iterative method for solving discretised Laplace equations on a grid. The program distributes the grid row-wise among the processors. Each processor exchanges its row of the matrix with its neighbors at the beginning of each iteration.

TSP finds the shortest route among a number of cities using a parallel branch-and-bound algorithm, which prunes large parts of the search space by ignoring partial routes already longer than the current best solution. We divide the whole search tree into many small ones to form a job queue. Every worker thread will get jobs from this queue until the queue is empty.

We measure the performance overhead during the normal execution (without initiating the state capturing) introduced in these applications by the fault-tolerance extensions. This overhead is generated by additional conditional statements placed after method calls which may initiate state capturing, and by replacing the standard Java RMI with its fault-tolerant counterpart. We argue that the overhead caused by the extra conditional statements is negligible since they are sensibly placed and thus rarely invoked.

[1] A detailed description of the DAS2 cluster architecture can be found at *http://www.cs.vu.nl/das2*

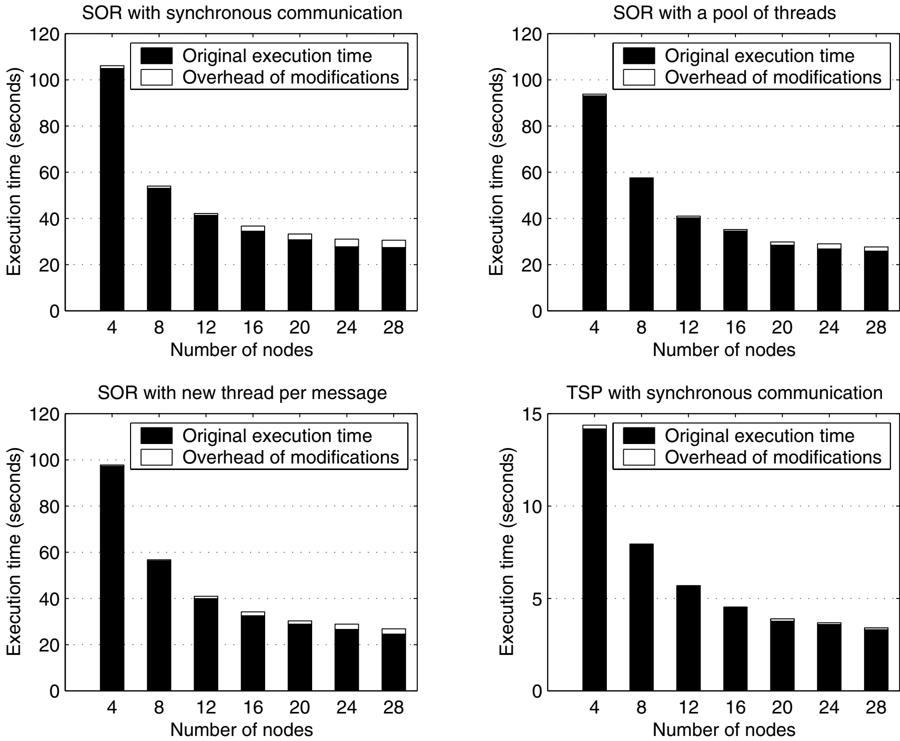

Fig. 1. The performance overhead incurred by our fault-tolerance extensions

We first asses the performance when no checkpoints are taken. Figure 1 presents the comparison of the execution times of the original and the transformed applications in relation to the number of nodes used for the computation in that case. Three variants of the SOR algorithm (for a matrix 1000x10000 and 200 iterations) and one variant of TSP (for 17 cities) were used during the measurements. The first version of the SOR algorithm uses synchronous communication. The second version is based on asynchronous communication scheme with a pool of threads which are reused for sending messages. The last variant of the SOR algorithm starts a new thread for every data send request. The TSP application uses synchronous communication.

The measurements show that overhead in SOR incurred by the fault-tolerance extensions increases with the number of nodes. The overhead caused by the stubs transformations is proportional to the number of remote method invocations which is higher for larger number of nodes. The highest performance overhead of 9% was observed for the SOR application with synchronous communication. In the case of synchronous communication, the overhead incurred by the stubs transformations affects directly the computation thread, thus slowing down the whole application. The performance degradation of the asynchronous versions

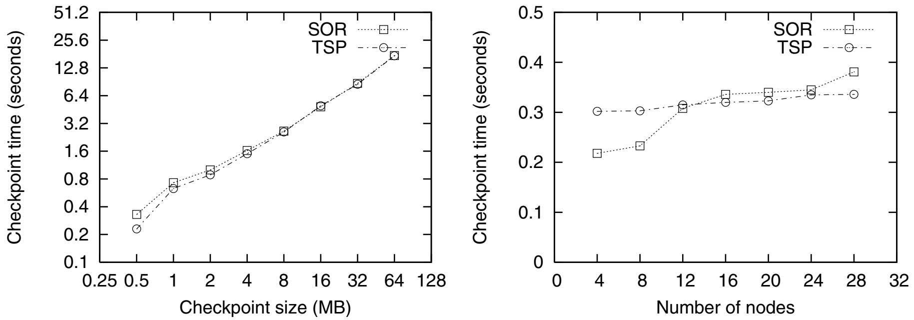

Fig. 2. The duration of the checkpointing process as a function of the checkpoint size (left) and the number of nodes (right). Note the logarithmic scale on both axes of the left plot

of SOR affects directly only the communication threads that are running in the background. The highest observed performance losses in the asynchronous versions with a thread pool and a new thread per message were 5% and 6%, respectively. A slightly higher overhead in the latter case comes from the fact that creating a new thread for each message requires registering this thread in our system. The bottom-right plot of Fig. 1 presents the results of the TSP application. The fault-tolerance extensions in this application hardly influence its performance. The highest observed overhead was less than 2%. Since processes do not communicate with each other as frequently as in the SOR algorithm, the overhead caused by the fault-tolerant RMI is much lower.

We now turn to the time that is needed to take a distributed checkpoint (Fig. 2). Clearly, capturing the state of a process that contains more data takes longer. Similarly the number of nodes in the system is not without influence on the overall performance of the checkpoint. The delay introduced by the global barrier synchronisations grows with the number of nodes involved.

The left half of Fig. 2 shows how the size of the checkpointed data relates to the time needed to take a globally consistent checkpoint of the applications running on 20 nodes (note the logarithmic scale on both axes). For the distributed checkpoint performance evaluation we used the most complex variant of the SOR algorithm, namely the asynchronous communication with the thread pool. As one could expect, the time of a checkpoint can be approximated by a linear function of the data size. The linear dependency on this logarithmic plot is however much weaker for smaller (less than 3MB) than for bigger checkpoints. For smaller checkpoints the state capturing time is dominated by the efficiency of the state saving extensions. Checkpoints that contain more data move the overhead from the data serialisation to the data replication phase, which is much more data size dependent.

The plot presented in the right half of Fig. 2 shows the influence of the number of nodes on the performance of the checkpoint. The size of the checkpointed

data was approximately the same for all applications — 500KB. The overhead of the checkpointing phase is determined by two factors. The performance of the global barrier synchronisation algorithm depends on the number of nodes involved. Furthermore, different threads need different amount of time to capture their states. The variations of the state serialization times among different threads accumulate resulting in considerable delays. The results of the experiments show however that our system can deal with a higher number of nodes without excessive performance loss.

As we described in Sect. 3, we optimised the original Brakes algorithm by rewriting only those methods that may lead to the checkpoint request. We measured the performance gain due to this optimization for the SOR and TSP applications running on 8 nodes. The number of rewritten methods was reduced for SOR from 53 to 5 and for TSP from 37 to 2. This resulted in a performance gain of over 10% in the case of SOR and over 30% in the case of TSP.

8 Conclusions

In this paper we have presented a complete solution for making regular grid applications written in Java fault tolerant. The high level of programming transparency and independence from a particular JVM enable easy integration with existing applications and grid services. The experiments show that our approach has a very low performance overhead during normal program execution, and scales well with the system size.

Acknowledgments

The authors would like to thank Jim Dowling, Dick Epema, Thilo Kielmann, and Tim Walsh for their valuable comments on the draft version of this paper. This work was partly supported by the DBE Project, IST 6th Framework Programme.

References

[1] Java platform debugger architecture (jpda). `http://java.sun.com/products/jpda/`.
[2] Java remote method invocation specification. revision 1.10, jdk 1.5.0, 2004. `http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf`.
[3] G. Allen, W. Benger, T. Goodale, H. Ch. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The cactus code: A problem solving environment for the grid. In *The Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, PA, USA, August 2000.
[4] D. C. Arnold and J. Dongarra. The netsolve environment: Progressing towards the seamless grid. In *International Workshop on Parallel Processing*, Toronto, Canada, August 2000.
[5] S. Bouchenak. Making java applications mobile or persistent. In *Conference on Object-Oriented Technologies and Systems*, San Antonio, TX, USA, January 2001.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
[7] R. Clark, E. Jensen, and F. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *USENIX Winter Conference*, San Diego, CA, USA, January 1993.
[8] T. Coninx, E. Truyen, B. Vanhaute, Y. Berbers, W. Joosen, and P. Verbaeten. On the use of threads in mobile object systems. In *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, June 2000.
[9] S. Fuenfrocken. Transparent migration of java-based mobile agents. In *Second International Workshop on Mobile Agents*, Stuttgart, Germany, September 1998.
[10] J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000. http://java.sun.com/docs/books/jls/.
[11] T. Illman, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. In *The Fifth IEEE International Conference on Mobile Agents*, Atlanta, GA, USA, December 2001.
[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999. http://java.sun.com/docs/books/vmspec/.
[13] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient java rmi for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, November 2001.
[14] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
[15] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *3rd International Conference on Coordination Models and Languages*, Amsterdam, The Netherlands, April 1999.
[16] Nathan Stone, Derek Simmel, and Thilo Kielmann. GWD-I: An architecture for grid checkpoint recovery services and a GridCPR API. Grid Checkpoint Recovery Working Group Draft 3.0, Global Grid Forum, http://gridcpr.psc.edu/GGF/docs/draft-ggf-gridcpr-Architecture-2.0.pdf, May 2004.
[17] N. Suri, J. Bradshaw, M. Breedy, P. Groth, A. G. Hill, and R. Jeffers. Strong mobility and fine-grained resource control in nomads. In *Agent Systems and Applications / Mobile Agents*, Zurich, Switzerland, September 2000.
[18] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
[19] P. Tang and P. C. Yew. Algorithms for distributing hot spot addressing. Technical report, Center for Supercomputing Research and Development, University of Illinois Urbana-Champaign, January 1987.
[20] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: An efficient java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, WA, USA, November 2002.
[21] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based grid programming. In *AGridM 2003 Workshop on Adaptive Grid Middleware*, New Orleans, LA, USA, September 2003.
[22] D. Weyns, E. Truyen, and P. Verbaeten. Distributed threads in java. In *International Symposium on Parallel and Distributed Computing*, Iasi, Romania, July 2002.