A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability

Françoise Baude[1], Denis Caromel[1], Christian Delbé[1], and Ludovic Henrio[1,2]

[1] INRIA – CNRS, Univ. Nice-Sophia Antipolis
2004, Route des Lucioles, BP93, 06902 Sophia Antipolis Cedex, France
{Francoise.Baude,Denis.Caromel,Christian.Delbe,Ludovic.Henrio}@inria.fr
[2] Harrow School of Computer Science, Univ. of Westminster, Harrow HA1 3TP UK

Abstract. Communication Induced Checkpointing protocols usually make the assumption that any process can be checkpointed at any time. We propose an alternative approach which releases the constraint of always checkpointable processes, without delaying any message reception nor altering message ordering enforced by the communication layer or by the application. This protocol has been implemented within ProActive, an open source Java middleware for asynchronous and distributed objects implementing the ASP (Asynchronous Sequential Processes) model.

1 Introduction

To ensure consistency of recovery lines, Communication-Induced-Checkpointing (CIC) protocols [5, 8, 9] usually make the assumption that every process of the system can be checkpointed at any time: a reception might lead to a *forced* checkpoint. But this assumption can fail for complex or particular systems where a process is not always in a state that can be checkpointed. In particular, in the context of Java middlewares like ProActive [7], persistence can be obtained in a convenient and portable way by standard Java serialization. But, as a thread cannot be serialized, an important part of the activity[1], the threads' stacks, cannot be checkpointed without special arrangements which are discussed below.

A first solution is to use specific tools that make checkpoints possible at any time: threads persistence can be achieved by modifying the execution environment at the OS level [14] or at the virtual machine level [18], or by using a native code-based persistence library [13]. Persistence capabilities can also be added using customized compilers: they add code to capture enough informations to characterize the state of a process [2], or use compile-time reflection to provide persistence functions [12]. But those tools usually involve a loss of portability and/or efficiency. In the context of Java, it is rather unfortunate to lose portability.

A more portable and convenient solution is grounded on the possibility to identify program points at which a checkpoint is possible. In the context of a multi-threaded programming environment like in Java, it concretely means

[1] We prefer the term activity rather than process to identify the runtime entity.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 644–653, 2005.

that, at those points, the state of an activity is fully characterized without any knowledge about the state of its thread(s). For instance, the existence of such states grounds the *weak* migration capability of mobile agents, as provided for instance in Voyager or Aglets [1]: a mobile agent is able to migrate only when it reaches such a state. We have identified such program points in the ASP model [6], which are called *stable states* in the following.

As already said, in CIC checkpointing, a message reception might lead to a forced checkpoint. In a fully asynchronous message-passing context, message receptions are unpredictable. So, message receptions can be artificially and simply delayed without consequence, until a checkpoint can be taken (i.e., the execution reaches a stable state). But, as soon as synchronization mechanisms through blocking message reception exists, like the *wait-by-necessity* mechanism in the ProActive middleware, or in MPI with the blocking receive routine [11], this simple solution is not applicable. Indeed, postponing a message reception could lead to a deadlock (e.g. considering that a message could be awaited by the program in order to continue the execution and finally reach a stable state, postponing this message to the next stable state would obviously yield to a deadlock).

Our work has thus consisted in reconsidering the initial simple solution of postponing message receptions, given the constraints raised by Java middlewares like ProActive and its associated computation model, the ASP calculus. Eventually, it has appeared that this new protocol applies for a wider range of contexts.

2 Context

This section describes the hypothesis for which our protocol have been designed and circumscribes the more general context in which it can be applied.

2.1 Constrained Checkpointability

ProActive being a Java middleware, it is impossible to store the state of a thread, thus to take a checkpoint at any time. However, some stable states where a checkpoint is possible can be either automatically identified at the middleware level, or explicitly defined at the application level.

Middleware Level. An activity is in a stable state when its state can be represented without any information about its thread (particularly the stack). The checkpointing can be thus performed using standard Java serialization; the persistence capability is then fully-transparent to the programmer. We have identified such states in the ASP model (see Section 2.3).

Application Level. The proposed protocol can be also used at the application level: stable states could also be specified by the programmer as in [11] ; in that case, the application would be responsible for restoring the state of the activity upon recovery. Although this second approach loses transparency for the programmer, it allows the programmer to save the *minimum* amount of data necessary to recover the activity state.

2.2 Distribution and Communication Model

ASP object calculus is based on concurrent mono-threaded activities communicating using two kinds of messages: request and reply. Each activity consists of one thread, a set of objects (which we call its *applicative state*), and a request queue. There is a master object among the applicative state that is called the *active object*. Note that the applicative state of an active object cannot be shared with another active object: there is no shared memory in our model.

In ASP, when an activity calls a method on an active object, a new request is added to the request queue of this active object. When the signature of the called method has a return value, a future is created on the sender side: this future represents the result of the request that is not known yet. Futures are generalized references that can be manipulated as classical objects. However, some operations (e.g. field access) need a real object value to be performed. Performing such operations on future objects leads to a blocked state called *wait-by-necessity*. When the receiver ends the service of a request, the associated future can be updated: the receiver sends a reply that will replace the future. Note that the impact of a message reception is different depending on the kind of the message. On one hand, a request reception modifies only the request queue of the receiver until it serves the request ; this alteration is reversible by removing this request. On the other hand, a reply reception modifies, in a *non reversible* manner, the applicative state of the receiver.

Causally ordered communications are achieved using a *rendez-vous* taking place at the beginning of each communication [4]. When an activity sends a message to another, it stops its execution until the message is in the context of the receiver. The rendez-vous implies that communications are acknowledged and has the advantage to always ensure point-to-point FIFO ordering of messages.

Our model then guarantees causal ordering of messages; the fault-tolerance protocol thus has to preserve this ordering in case of recovery of the system. More generally, as any synchronization primitive is sufficient to ensure any (partial) ordering of messages at the application level, a protocol with constrained checkpointability has to preserve during a recovery the communication order enforced by the application.

2.3 Properties and Assumptions

The ASP Calculus: In [6], we proved using the ASP calculus the two following main properties:

Property 1 *The relative order of reception of replies during a distributed execution has no consequence on the behavior of the program, assuming that no deadlock is caused by wait-by-necessity.*

Property 2 *An execution can be characterized only by the ordered lists (one for each activity) of request sender identifiers.*

The first property would not be necessary in a middleware that does not have futures. A weaker version of the second one seems to be verifiable in most service-oriented platforms: *An execution can be characterized only by the ordered lists of requests.* With such a property, one would just have to store more informations inside *promised requests* (Section 3.1) in order to use our protocol.

Assumptions on the System: We also make the following assumptions:

- activities are piecewise-deterministic [17] and fail-stop [16],
- failures are detected in an arbitrary but finite time [19],
- an available host always exists, in order to restart a failed activity,
- a stable storage, known by each activity, exists in order to save checkpoints.

Presence of Stable States: An activity is in a *stable state* when it does not serve any request. Indeed, between two request services, the thread state is not necessary to fully characterize the state of the activity. Consequently, the stable state of an activity can be recorded through standard Java serialization of the applicative state and of the request queue. Note that in practice, the presence of stable states requires that the activities never serve a request which service does not end. But this restriction is easy to tackle since an infinite service can always be imitated by a infinite sequence of self-sent requests.

2.4 Notations

Figure 1 shows two activities i and j. j calls a method on i: a request Q is sent. Eventually, this request is served on i and a reply, result of the service, is sent. A rectangle drawn using dotted lines represents the period of service of a request. Conversely, a period of stable state is represented by a simple line. Figure 2 shows a checkpoint C_i^n on an activity i, its sequence number (n) and the request queue of i ($[Q^1, Q^3, Q^4]$). An empty queue is denoted by $[\emptyset]$.

3 Principle of the Protocol

The proposed protocol is an adaptation of [5] and [8] for constrained checkpointability. A parameter TTC, the checkpointing time counter, allows each activity to periodically take checkpoints: if an activity has not taken any checkpoint

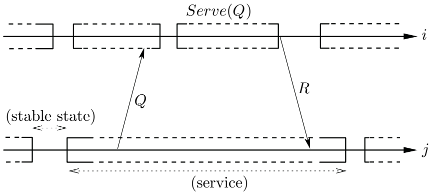

Fig. 1. Communicating activities

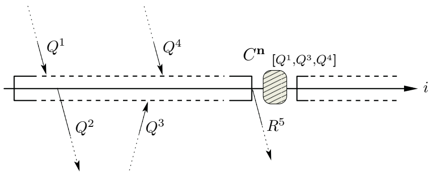

Fig. 2. Checkpoint on an activity

during TTC seconds then a checkpoint is triggered as soon as a stable state is reached. This time counter is reinitialized *each time* a checkpoint is taken. On an activity, each checkpoint is identified by an index which monotonically increases. In case of recovery, all activities have to restart from the same checkpoint index; a set of checkpoint with the same index is called a recovery line. The current checkpoint index of the sender is piggybacked on every message, and the current index of the receiver is piggybacked on every acknowledgment message. These piggybacking allow to identify potential *orphan messages* (message that have been sent after but received before the recovery line, then duplicated in case of recovery) or *in-transit messages* (messages that have been sent before but received after the recovery line, then lost in case of recovery).

In classical CIC protocols, such messages should trigger a *forced* checkpoint on the sender or on the receiver so as to ensure consistency of the currently built recovery line. In our context, it is not possible to take those forced checkpoints; the consequence of this constrained checkpointability is that recovery lines are *inconsistent*. Indeed, there might be orphan and in-transit messages. Compared to classical CIC protocols, our protocol must then handle those messages *a-posteriori*, as soon as a checkpoint is possible, to avoid lost or duplicated messages in case of recovery: we then introduce an *additional message-logging mechanism*.

Since the sending of logged messages after a recovery is obviously triggered by the protocol, the message-logging mechanism could lead to a loss of causal dependencies between messages, and then break the message ordering. Thus, as long as there might exists a message that can be logged during the first execution, the protocol has to record enough informations to be able to ensure execution equivalence in case of recovery. For that, we introduce the *request reception history*, a list of *promised requests*.

3.1 Promised Requests

A promised request is a local substitute for a request that is not yet received in the re-execution; it only contains the identity of the activity from which a request is awaited. A promised request awaited from i in the request queue of j is denoted by $Q_{i,j}^{pmd}$. The service of a promised request is subject to synchronization through a wait-by-necessity mechanism: if an activity tries to serve a promised request, it is blocked until the awaited request is received and updates the promised one.

To summarize, a promised request is a place holder for a request that will be received after a recovery and has already been received in the first execution.

3.2 Orphan Messages

The reception of an orphan request should trigger a checkpoint *before* the delivery of this message. As this is not possible, we replace in the next possible checkpoint the request by a promised one inside the request queue. When resent during recovery, this request will thus automatically be inserted at the right place in the request queue, like Q in case of recovery from $n+1$ in Figure 3: the use of a promised request allows to preserve the relative order of requests during the two executions, thus ensures execution equivalence.

Concerning replies, their reception order is not significant thanks to Property 1 but, as stated in Section 2.2, we cannot cancel their effect on the applicative state. However, activities being piecewise-deterministic, ensuring the equivalence of executions is sufficient for guaranteeing that the reply sent during the re-execution is the same as during the first execution, and thus can be ignored.

3.3 In-transit Messages

In-transit messages (requests and replies) can be logged in the next possible checkpoint and re-sent during the re-execution. We introduce the *re-send queue*, denoted by $\Uparrow\{Q^n, Q^m....\}$, a queue of messages that have to be re-sent during a recovery. Figure 4 shows the logging (noted $\Uparrow\{Q\}$) of the in-transit request Q in the checkpoint n.

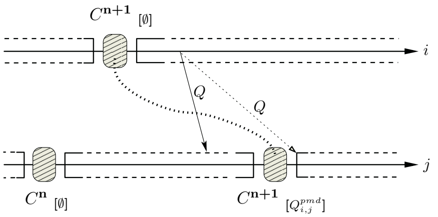

Fig. 3. The request Q is replaced by a promised request in C_j^{n+1}

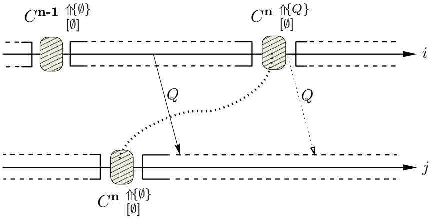

Fig. 4. The request Q is logged for re-sent in the checkpoint n of i

3.4 Request Reception History

To preserve message ordering, the protocol must ensure equivalence between the first execution and the re-execution in case of recovery. By doing this, it also ensures that orphan replies are identical in the first execution and in the re-execution (Section 3.2). This equivalence must last until the completion of the currently built recovery line. Indeed, after this completion, there cannot be any in-transit nor orphan messages: there cannot be anymore causal relation loss between messages nor duplicated reply.

So as to ensure execution equivalence, we introduce the request reception history. Thanks to the Property 2, this history just needs to record the ordered list of the identity of activities that have sent requests; this information is sufficient to ensure execution equivalence. Consequently, a request reception history for an activity i and for its n^{th} checkpoint is a list of promised requests standing for the requests received between this local checkpoint n and the *history closure*. The only constraint on this closure point is that it must occur *after* the completion of the recovery line n.

The history closure can thus be triggered by a message sent by the stable storage as soon as all the checkpoints with the same index have been received.

The consistency of the history closure line is crucial for preventing infinite wait on a promised request. It is ensured by avoiding orphan message as in [5]: the reception of a message from an activity that has already closed its history triggers on the receiver the closure *before* the delivery of this message.

Finally, when an activity recovers from a checkpoint n, it just has to append to its request queue the history of the checkpoint n: execution equivalence is then ensured as long as an inconsistency could appear.

4 Experiments

A prototype has been implemented within the ProActive Java library. These experiments are a first experimental validation of our protocol, but [3] and [10] provide also a formal presentation of the protocol and the main steps of the correctness proof. [10] proves that our protocol ensures that any re-execution from any recovery line eventually reaches a consistent global state that occurs in the first execution.

4.1 Test Applications

We choose to evaluate the overhead induced by our protocol within two representative applications:

- "Sieve of Eratosthenes" computes the n^{th} prime number in a master-slaves configuration. The communication pattern is 1-to-n for the master node.
- "Jacobi" performs an iterative computation on a square matrix of floats. On each iteration, the value of each point is computed as a function of its value in the last iteration and the values of its neighbors. A square sub-matrix is allocated to each activity. The communication pattern is 1-to-m for all nodes; each activity communicates with its direct neighbors. Each activity is equivalent to the others.

Note that the benchmarks are performed with the same source code for standard and fault-tolerant executions, since there is no need to alter nor recompile the source of an application to make it fault-tolerant. The tests have been performed on a cluster of bi-Xeon @ 2Ghz 1 Gb RDRAM - 512 Kb L2 cache, Linux 2.4.17, interconnected with a 1 Gb/s Ethernet, on the Sun Java Virtual Machine 1.4.2.

4.2 Performance Overhead

Table 1 shows the overhead ($\frac{ExecTime_{tolerant} - ExecTime_{non-tolerant}}{ExecTime_{non-tolerant}}$) induced by the protocol for respectively the Eratosthenes and the Jacobi application running with 8 slaves (one slave per CPU) for Eratosthenes, and with 9 sub-matrix (one sub-matrix per CPU) for Jacobi. The checkpointing time counter is initialized with $TTC = 100\ sec$ for each activity. For each data size (the computed prime number or the matrix size) and *for one activity*, average checkpoint size (checkpoint of the slave for Eratosthenes), number of checkpoints performed, cumulated checkpointing time (the maximum among all activities) and average received message rate are given.

Table 1. Overhead for Jacobi (9 CPUs) and Eratosthenes (8 CPUs), $TTC = 100sec$

	Erathostenes (# Prime Number)					Jacobi (Matrix Size)			
Data Size	**1000**	**2000**	**3500**	**5000**	**10000**	**500**	**1000**	**2000**	**3000**
Exec. Time (s)	201	420	785	1152	2477	153	253	563	1315
Msg Rate (msg/s) (Slave)	42	42	42	42	42	78	47	22	9
Msg Rate (msg/s) (Master)	316	331	332	337	338	n/a	n/a	n/a	n/a
Ckpt Size (Mb)	0.39	0.82	1.4	1.95	4.01	0.68	1.07	7.16	15.07
# Ckpts	3	5	9	13	27	2	3	6	14
Ckpting Time (s)	0.6	1.1	3.5	7.1	33.5	0.9	1.3	3.7	3.9
Overhead (%)	4.95	6.56	6.38	7.77	7.50	2.87	2.94	4.07	4.41

The measured overhead is low: it varies from about 3% to 8% in the worst case. This overhead can be decomposed in two parts: overhead due to message treatment, and time spent for checkpointing (mainly serialization and communication with the stable storage). The higher overhead observed for Eratosthenes application is due to the higher received message rate of the master. Both overheads increase with data size because of checkpoint size. The smoother increasing of Jacobi overhead is explained by the fact that the growing of checkpoint size is counterbalanced by the decreasing of the received message rate for each activity, while the rate of the master for Eratosthenes does not lower with data size.

We also notice that the number of checkpoints performed by an activity *linearly* increases with the fault-tolerant execution time, and that each activity performs the same number of checkpoints. This stability (observed for all the applications we have experimented) is an interesting property of our protocol since a large an unpredictable number of *additional* checkpoints forced by the protocol is known to be the Achilles' heel of CIC protocols [15].

4.3 Scalability

Figure 5 presents the overhead induced by the protocol for Eratosthenes and Jacobi applications regarding the number of CPUs. Eratosthenes computes the 2000^{th} prime number and Jacobi iterates on a 2000*2000 matrix. We observe that the overhead remains roughly constant up to 25 CPUs for both applications: this result demonstrates that the proposed protocol scales well.

4.4 Faulty Execution

Figure 6 shows the recovery time, i.e. the time spent for recovering every activities after a fault, for Eratosthenes and Jacobi regarding the number of CPUs.

The recovery time remains low, up to 25 CPUs (38 *sec* for Eratosthenes and 18 *sec* for Jacobi) and smoothly increases with the number of CPUs from 9 CPUs. The higher recovery time for Eratosthenes is linked to the higher message rate for both master and slaves. Indeed, a higher message rate leads to a longer history, then the synchronization due to promised requests lasts longer.

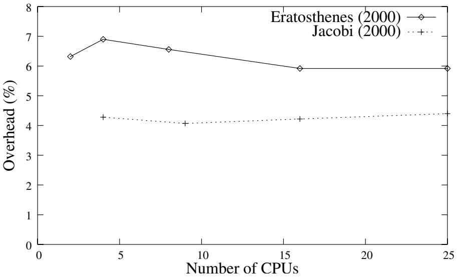

Fig. 5. Execution overhead for Eratosthenes and Jacobi

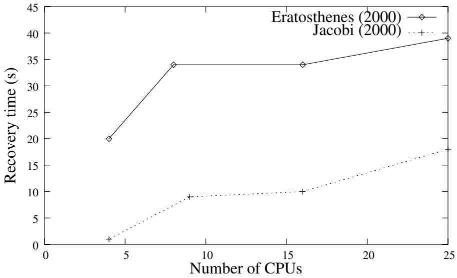

Fig. 6. Recovery time for Eratosthenes and Jacobi

5 Conclusion

In this paper, we have presented a new hybrid CIC-message logging protocol for Java middlewares that *does not assume* permanent checkpointability. It allows recovery from lines made of restrictively placed checkpoints, without delaying any message reception nor breaking message ordering. The proposed protocol:

- deals with in-transit messages thanks to message logging,
- deals with orphan messages thanks to promised requests and history,
- performs low number of checkpoints,
- is fully transparent since there is no need to alter nor recompile code application to make it fault-tolerant.

It has been implemented in a 100% Java compatible way within the middleware ProActive; as a consequence, the usage of dedicated tools for persistence can be avoided, and portability is total.

Even if the presented protocol has been designed and implemented in the context of ProActive, its main idea is the ability to recover from inconsistent recovery line without breaking any message ordering. As such, this work is applicable to other middlewares, even those using applicative-level persistence. Overall, the location of checkpoints is no more a strong constraint.

The context of this article is somehow similar to [11]; but, contrarily to Bronevetsky et al., our protocol focuses on ensuring the message ordering at recovery. Indeed, a given ordering is always ensured by ProActive but may also be enforced by some applications, even over MPI. The [11] approach may lead those applications into a state that should not exist, since *only* the receptions of orphan or in-transit messages are logged. Introducing a message reception history in [11] would allow one to also cope with this category of applications.

The practical target of our research is also large-scale distributed programming such as grids. In this context, CIC protocols are maybe not the best choice. Indeed, these protocols are more efficient for small systems with low failure rate. On the contrary, grids are large systems with a high failure rate, and a grid application is often partitioned into loosely coupled components, each component being based upon more strongly cooperating processes. In this case, we think of an adaptive approach that autonomously chooses the best combined usage of message logging and hybrid CIC-message logging. The protocol proposed here can thus be considered as a step towards such a *single parameterized* protocol.

References

1. Aglets Software Development Kit. IBM, 1999. http://www.trl.ibm.com/aglets/.
2. B.Ramkumar and V.Strumpen. Portable checkpointing for heterogenous architectures. In *Fault-Tolerant Parallel and Distributed Systems*, pages 73–92, 1998.
3. C.Delbé. Causal ordering of asynchronous request services. In *Dependable Systems and Networks - Student Forum*. IEEE, June 2004.
4. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communications. *Distributed Computing*, 9(4):173–191, 1996.
5. D.Briatico, A.Ciuffoletti, and L.Simoncini. A distributed domino-effect free recovery algorithm. In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, pages 207–215, 1984.
6. D.Caromel, L.Henrio, and B.Serpette. Asynchronous and deterministic objects. In *31st ACM Symposium on Principles of Programming Languages*. ACM Press, 2004.
7. D.Caromel, W.Klauser, and J.Vayssiere. Towards seamless computing and metacomputing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd., November 1998.
8. D.Manivannan and M.Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Proceedings of the 16th ICDCS*, pages 100–107, 1996.
9. D.Manivannan and M.Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. In *IEEE Transactions on Parallel and Distributed Systems*, volume 10, pages 703–713, 1999.
10. F.Baude, D.Caromel, C.Delbé, and L.Henrio. A fault tolerance protocol for asp calculus: Design and proof. Technical Report RR-5246, INRIA, 2004.
11. G.Bronevetsky, D.Marques, K.Pingali, and P.Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, 2003.
12. J.C.Ruiz-Garcia, M.O.Killijian, J.C.Fabre, and S.Chiba. Optimized object state checkpointing using compile-time reflection. In *Workshop on Embedded Fault-Tolerant Systems*, pages 46–48, 1998.
13. J.Howell. Straightforward java persistence through checkpointing. In *Proceedings of the 3rd International Workshop on Persistence and Java*, pages 322–334, 1998.
14. J.S.Plank, M.Beck, G.Kingsley, and K.Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.
15. L.Alvisi, E.N.Elnozahy, S.Rao, S.Husain, and A.De Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.
16. R.D.Schlichting and F.B.Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. In *ACM Transactions on Computer Systems*, volume 1, pages 222–238, 1983.
17. R.E.Strom and S.Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems*, volume 3, pages 204–226, 1985.
18. S.Bouchenak. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems*, 1999.
19. T.D.Chandra and S.Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.