HAL
archives-ouvertes.fr

O2P: An Extremely Optimistic Message Logging Protocol

Thomas Ropars, Christine Morin

▶ **To cite this version:**

Thomas Ropars, Christine Morin. O2P: An Extremely Optimistic Message Logging Protocol. [Research Report] RR-6357, INRIA. 2007. <inria-00187682v2>

HAL Id: inria-00187682
https://hal.inria.fr/inria-00187682v2

Submitted on 15 Nov 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

O2P: An Extremely Optimistic Message Logging Protocol

Thomas Ropars — Christine Morin

N° 6357

Novembre 2007

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6357--FR+ENG

O2P: An Extremely Optimistic Message Logging Protocol

Thomas Ropars* , Christine Morin†

Thème NUM — Systèmes numériques
Projet PARIS

Rapport de recherche n° 6357 — Novembre 2007 — 16 pages

Abstract: Message logging is a transparent solution to provide fault tolerance for message passing applications. O2P is an extremely optimistic message logging protocol that is proved to tolerate multiple failures. Extremely optimistic message logging aims at combining the advantages of optimistic and pessimistic message logging to be well-suited for large scale applications while minimizing the overhead on failure free execution. In this paper, we present the O2P protocol and prove that it can handle concurrent failures.

Key-words: Fault Tolerance, Message logging, Message Passing Applications

* thomas.ropars@irisa.fr
† christine.morin@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

O2P: un protocole à enregistrement de messages extrêmement optimiste

Résumé : Les protocoles à enregistrement de messages sont une solution transparente pour fournir de la tolérance aux fautes aux applications à échange de messages. O2P est un protocole à enregistrement de messages extrêmement optimiste qui peut tolérer de multiples fautes concurrentes. L'objectif de l'enregistrement de message extrêmement optimiste est de combiner les avantages des protocoles optimistes et pessimistes pour être adapté aux applications de grande échelle tout en limitant le surcoût induit par le protocole en fonctionnement normal. Dans cet article nous présentons O2P et prouvons qu'il peut tolérer plusieurs fautes simultanées

Mots-clés : Tolérance aux fautes, Enregistrement de messages, Applications à Échanges de Messages

1 Introduction

Dealing with failures is one of the key issues in High Performance Computing (HPC) since the size the computing infrastructure is continuously growing. Increasing the number of computing nodes implied in an execution makes the Mean Time Between Failures (MTBF) decrease. Fault tolerance mechanisms are needed to enable long running applications to terminate despite failures.

Message passing is a programming paradigm widely used in HPC. Rollback-recovery [5] techniques are well known techniques to provide transparent fault tolerance mechanisms for message passing applications. Coordinated checkpointing is the solution which is mainly used. The first drawback of this technique is that the failure of one process implies a rollback all the application processes. Furthermore, coordinating the checkpoints may induce expensive concurrent accesses to stable storage. Message logging protocols have the advantage over coordinated checkpointing protocols to minimize the impact of a failure since they do not require every process to rollback in case of one failure. Furthermore they can be combined with uncoordinated checkpointing without the risk of domino effect. Solutions based on pessimistic message logging protocols log information synchronously on stable storage and thus induce an overhead on failure free execution.

In this paper we present O2P, an extremely optimistic message logging protocol tolerating multiple concurrent failures. Since it's an optimistic protocol, it induces no overhead on failure free execution. Furthermore, the extremely optimistic assumption used to log messages makes it scalable. Since O2P is a sender-based message logging protocol, it minimizes the amount of data stored in stable storage.

The paper is organized as follows. We explain the concept of extremely optimistic message logging in Section 2. We describe O2P in Section 3. In Section 4, we present and prove the recovery protocol. Additional remarks about O2P are given in Section 5. Related work is described in Section 6. Finally, we draw conclusion from this paper in Section 7.

2 Extremely Optimistic Message Logging Principles

In this section we first define the main concepts of message logging. Then we explain the extremely optimistic message logging principles.

2.1 Message Logging Basic Principles

Message logging protocols assume that process execution is piecewise deterministic [12], i.e. the execution of a process is a sequence of deterministic intervals started by a non-deterministic event, a message receipt. This means that starting from the same initial state and delivering the same sequence of messages, two processes inevitably reach the same state.

Determinants [1] describe messages. A determinant is composed of the message data and a tag. To identify messages, each process numbers the messages it sends with a sender sequence number (ssn) and the messages it receives with a receiver sequence number (rsn). The tag of a message is composed of the sender id, the ssn, the receiver id and the rsn. Message logging protocols save determinants into stable storage to be able to replay the sequence of messages received by a process in the event of failures.

The deterministic intervals composing the process execution are called state intervals. A state interval is identified by an index corresponding to the receive sequence number of the message starting the interval. Exchanges of messages create causal dependencies between the state intervals of the application processes. The state intervals are partially ordered by the Lamport's happen-before relation [8].

The determinant of a message is saved by the receiver into stable storage. A determinant is said stable when it is logged into stable storage.

Definition 1 *A state interval of a process is stable when all determinants of messages delivered by that process before this state interval are stable.*

Definition 2 *The maximum recoverable state interval of a failed process is the state interval it can reach by replaying the messages logged on stable storage before the failure.*

In case of failure, a message logging protocol has to restore the application in a consistent global state.

Definition 3 *In a consistent global state, if the state of a process reflects a message receipt, then the state of the corresponding sender process reflects the sending of that message [3].*

In pessimistic message logging, determinants are logged synchronously into stable storage. The sending of the next message is delayed until the determinant of the previous received message is saved into stable storage. Considering Property 1, the overhead induced by pessimistic protocols would be low. The advantage of this solution is that nothing need to be piggybacked on the messages.

Property 1 *The optimistic assumption made in pessimistic message logging is that logging a determinant is fast enough so that the probability of having saved it before the next message sending is high.*

Determinants are logged asynchronously into stable storage in optimistic message logging protocols. A process can send a message without waiting for determinants corresponding to previous message deliveries to be logged in stable storage. Thus the overhead induced by the protocol on failure free execution is negligible. In the event of a failure, determinants may be lost. In this case some of the messages delivered by a failed process before the failure can not be replayed. The maximum recoverable state interval of the process is not the state interval reached just before the failure. If a non failed process depends on one of the lost state intervals, it becomes orphan.

Definition 4 *An orphan process is a non-failed process whose state reflects the receipt of a message that has not been sent.*

In fact, the message has been sent before a failure but the sending state interval of the message can't be restored after the failure. The orphan process depends on a lost state interval. By extension, we call orphan state interval, a state interval which depends on a lost state interval and we call orphan message, a message sent from a lost state interval.

After a failure, an optimistic message logging protocol has to restore the application in a consistent global state, i.e. a state without orphan processes. In the example depicted in Figure 1, the logged messages are surrounded. Process p_1 fails before logging m_3. So the maximum recoverable state of p_1 is si_1. Processes p_0 and p_1 become orphan since their current state interval causally depend on the state interval si_2 of process p_1. The application must be rolled-back to the consistent global state represented by the dash line in the figure. Considering Property 2, the risk of orphan process creation with an optimistic message logging protocol is low.

Property 2 *The optimistic assumption made in optimistic message logging is that logging a determinant is fast enough so that the risk of experiencing a failure of the receiver between message delivery and the log of the corresponding determinant is small.*

For the need of the recovery protocol, optimistic protocols piggyback information on each message to trace dependencies between state intervals. Optimistic protocols that tolerate concurrent failures piggyback dependency vectors of size n, n being the number of processes in the application, on each message. This kind of solution is not well-suited for large scale applications.

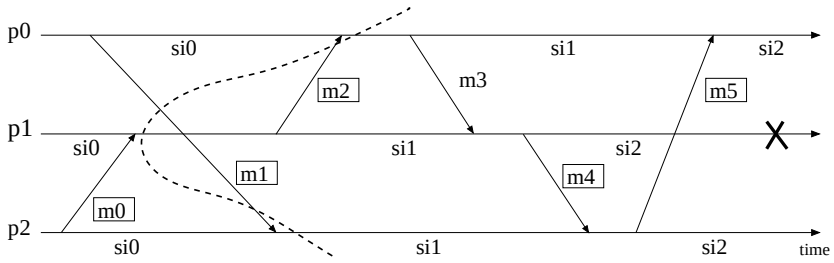

Figure 1: Example of a rollback induced by a failure

2.2 Extremely Optimistic Message Logging

The extremely optimistic assumption is a combination of Properties 1 and 2. O2P is an optimistic message logging protocol. It is based on the optimistic assumption of optimistic protocols (Property 2). Thus it induces no overhead on failure free execution.

But O2P is also based on the optimistic assumption of the pessimistic protocols (Property 1). O2P assumes that most of the time the current state interval of the sender is stable when it is sending a message. Thus O2P doesn't need to piggyback vectors of size n to trace dependencies between processes. It enables O2P to be well-suited for large scale applications. More details about dependency tracing in O2P are given in Section 3.2.

3 O2P Principles

3.1 System Model

We consider a distributed application composed of n processes communicating only through messages. Communication channels are FIFO and reliable but there is no bound on message transmission delay. A process receives messages from the network and then delivers it to the application. Each process has access to stable storage. Data saved in volatile memory is lost in a process crash and only the data saved on stable storage remains accessible. We assume a fail-stop failure model for processes. In this paper we only consider multiple concurrent failures and we assume that no failure occurs before the end of the recovery protocol.

3.2 Tracing Dependencies Between Processes

To be able to rollback orphan processes, O2P needs to trace dependencies between processes during failure free execution. Thanks to the extremely optimistic assumption, we can use a dependency list instead of a dependency vector. Each process lists the determinant of the messages it depends on. When a process delivers a message, it adds the determinant to its dependency list and it removes it when the determinant is saved on stable storage, i.e. when it receives the acknowledgment from stable storage. When sending a message, the sender piggybacks its dependency list on the message and the receiver adds this list to its own dependency list when it delivers the message. If a message becomes orphan then all the processes having the determinant of this message in their dependency list are orphan processes. The failure free protocol is briefly described in Figure 2.

Sending a message msg **by process** p_i **to process** p_j
  Piggyback $Unlog_i$ on msg // $Unlog_i$ *is the dependency list of process* p_i
  Send msg to p_j
  Save msg in volatile memory

Delivering a message (msg,$Unlog_{msg}$)
  Get (msg,$Unlog_{msg}$) from the incoming queue
  $si_i \leftarrow si_i + 1$ // *Changing of state interval*
  Deliver msg to the application
  Add det_{msg} to $Unlog_i$ // det_{msg} *is the determinant of* msg
  Update $Unlog_i$ with $Unlog_{msg}$
  Send det_{msg} to stable storage

Upon reception Ack(det_{msg}**) on process** p_i
  Remove det_{msg} from $Unlog_i$

Figure 2: The failure free protocol

In Figure 3, we show the protocol working on two scenarii. In Figure 3(a), the extremely optimistic assumption is valid. The dependency list of process p_2 is empty when message m_2 is sent. So there is nothing to piggyback on the message. In this case the protocol induces no overhead on failure free execution. In Figure 3(b), the assumption is not valid. The determinant det_1 of the message m_1 is piggybacked on message m_2. When process p_2 receives the acknowledgment of the logging of det_1, it forwards this acknowledgment to p_3 so that it can also remove det_1 from its dependency list. Thus size of dependency lists is kept minimal.

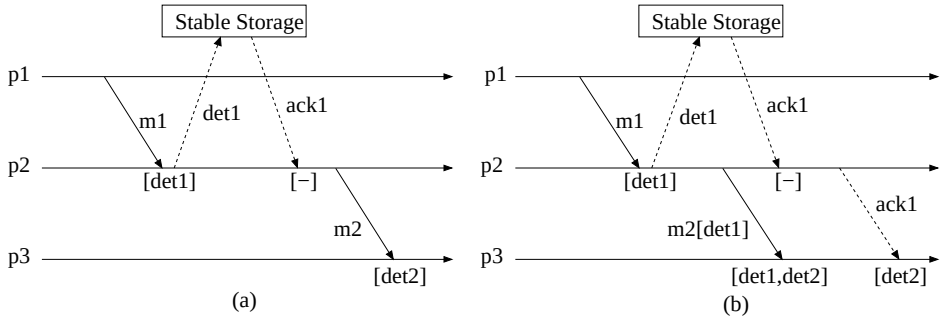

Figure 3: Two scenarii of execution

3.3 Sender-Based Message Logging

In sender-based message logging [6], determinants are saved in the sender volatile memory. Only one fault is tolerated because if the sender and receiver of a message both fail, all the information about this message is lost.

Sender-based message logging has been adapted to tolerate multiple failures in a pessimistic message logging protocol [2]. We use this solution for our optimistic protocol. The message data is saved in the sender volatile memory. If the receiver fails the message data is available in the sender memory for replay. If the sender fails, it will have to restart its execution and will re-create the message during recovery. So the determinant saved into stable storage only contains the message tag. Since the determinant size is minimal, we reduce the traffic on the network and increase the probability that the determinant is saved before the next message sending. Furthermore we save space on stable storage, while being able to tolerate n faults.

4 The Recovery Protocol

After a failure the application must rollback to a consistent global state. In the event of a single failure, finding a consistent global state is easy since every process has its dependency list and thus can detect if it has become orphan. When experiencing concurrent failures, it is more difficult, as illustrated in Figure 4. The problem is that the failed processes loose their dependency list, i.e. the knowledge of their causal dependencies. In this example, processes p_2, p_3 and p_4 fail while only messages m_1 and m_2 are stable. So the determinant of m_0 is lost in the failure. According to the information logged on stable storage, process p_2 can restore its state interval si_1. But message m_2 causally depends on m_0 and thus is orphan. State interval si_0 of process p_2 is its

maximum commitable state interval. Process p_2 must be restored in this state interval.

Definition 5 *A commitable state interval is a stable state interval that only depends on stable state intervals.*

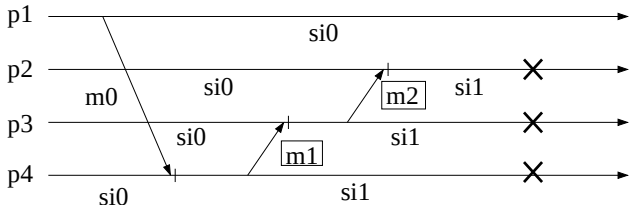

Figure 4: Concurrent failures

In this section, we prove through the presentation of the recovery protocol the following theorem:

Theorem 1 *After a failure, O2P restores the application in its maximum consistent global state.*

4.1 Data structures

To describe the recovery protocol, we need some variables and data structures. To be able to find the orphan state intervals, we associate with each determinant in a dependency list, $Unlog_i[det].si$ which is the index of the first state interval of process p_i depending on det. To compute the consistent global state, we use two n-vectors named $newCGS_i$ and $oldCGS_i$.

4.2 Restarting a failed process

Get Log_i from Stable Storage // *Log_i is the list of determinants logged by process p_i*
Broadcast message $RESTART(p_i, \|Log_i\|)$
$newCGS_i[p_i] \leftarrow \|Log_i\|$

Figure 5: Restarting a failed process p_i

On restart after a failure, a process first needs to get back from stable storage the list of determinants it has logged before the failure. Then it informs the other processes of its restart and gives its maximum recoverable state interval. These actions are summarized in Figure 5.

4.3 Orphan Process Detection

```
Upon reception of RESTART(p_j, si_j) do
    FP_i ← FP_i ∪ p_j     // FP_i is the list of failed processes
    if find_maximum_commitable_state() is not already running then
        max_comm_i ← si_i     // new maximum commitable state interval
        find_maximum_commitable_state()
    Let Det_orphan be the set of det ∈ Unlog_i such that (det.dest = p_j ∧ det.rsn > si_j)
    Let si_min be the min of Unlog_i[det].si for all det ∈ Det_orphan
    max_comm_i ← si_min − 1
    Remove Det_orphan from Unlog_i

On call to find_maximum_commitable_state()
    Wait until Unlog_i = ∅
    if max_comm_i < si_i then
        Rollback p_i to max_comm_i
    Send COMMIT(p_i, max_comm_i) to every p_j ∈ FP_i
```

Figure 6: Looking for orphan state intervals

When a non-failed process receives a failure notification, it has to check if it is orphan as described in Figure 6. If it is orphan, it has then to rollback to its maximum commitable state. Upon reception of the first `RESTART` message, a non-failed process starts a stabilization phase described in function `find_maximum_commitable_state()`. During this phase, it stops delivering messages. If its current state interval is not commitable, it may have become orphan. That's why delivering new messages and creating new state intervals that may need to be rolled-back later is useless.

Lemme 1 *Every orphan process will eventually rollback.*

Proof An orphan process is a non-failed process. So an orphan process has its dependency list. As channels are reliable, every non-failed process will eventually deliver the `RESTART` message sent by the failed processes. When a process delivers this message, it looks for orphan determinants in its dependency list. The dependency list has a complete list of causal dependencies of the process. So all orphan processes are detected.

Lemme 2 *In the event of a rollback, a non-failed process is restored in its maximum commitable state.*

Proof When a `RESTART` message is received by a non-failed process, orphan determinants are searched in the dependency list. In this set of orphan determinants, the determinant associated with the oldest state interval is selected. So the new estimated maximum commitable state is $si_{min} - 1$,

i.e. the first non-orphan state interval. When the dependency list becomes empty, it means that the estimated maximum commitable state interval does not depend anymore on non stable state intervals. We can conclude that it is the maximum commitable state interval of the process.

4.4 Negotiation Between the Failed Processes

```
On call to find_maximum_consistent_global_state()
  while the maximum consistent global state has not been found do
    Wait until newCGS_i is complete
    if newCGS_i ≠ oldCGS_i then    // Start an new round
      Send TRY(p_i,newCGS_i[p_i]) to every p_j ∈ FP_i
      oldCGS_i ← newCGS_i
      newCGS_i[p_j] ← ⊥ for all p_j ∈ FP_i
    else    // The maximum consistent global state has been found
      Rollback p_i to newCGS_i[p_i]

Upon reception of TRY(p_j, si_j) do
  if newCGS_i[p_j] = ⊥ then
    handle_rollback(p_j,si_j)
  else
    Deliver this message later

Upon reception of RESTART(p_j, si_j) do
  FP_i ← FP_i ∪ p_j
  handle_rollback(p_j,si_j)

Upon reception of COMMIT(p_j, si_j) do
  handle_rollback(p_j,si_j)

On call to handle_rollback(p_j,si_j)
  newCGS_i[p_j] ← si_j
  Remove from Log_i all det such that (det.src = p_j ∧ det.si > si_j)
  newCGS_i[p_i] ← ||Log_i||
```

Figure 7: Finding the maximum consistent global state

To find the maximum consistent global state of the application, O2P uses the algorithm presented in Figure 7. This algorithm is inspired from one proposed by Sistla and Welch [10].

We use two n-vectors $oldCGS_i$ and $newCGS_i$. $newCGS_i[p_j]$ is the current estimated maximum commitable interval of p_j known by p_i. The algorithm works in pseudo-rounds. A round finishes when $newCGS_i$ is complete. We don't need to use numbers to identify the rounds. Since channels are FIFO and reliable, we only need to ensure that each process delivers one message from every other processes in a round. The function `handle_rollback()` is used to remove orphan determinants from the list of messages to replay. For a failed process p_i, the first round finishes when it has received a message from every other process. It receives a `RESTART` message from all the other

failed processes and a `COMMIT` message from the non-failed processes. The non-failed processes are not included in the next rounds.

Lemme 3 *At the end of the first round, the maximum commitable state interval of at least one failed process is known.*

Proof Let $A = \{sc_1, sc_2, ..., sc_r\}$ be the set of maximum commitable state intervals for the failed processes. There is at least one sc_x in A that does not causally depends on any other sc in A since the Lamport's happen-before relation is a partial order on the set of state intervals in the system. Let sc_1 be this state interval. sc_1 only depends on maximum commitable state intervals of non-failed processes. We deduce from Lemme 1 and Lemme 2 that sc_1 is found at the end of the first round.

In a round, a failed process first sends its new estimated maximum commitable state in a TRY message to the other failed processes. Then it evaluates its new estimated commitable state interval according to the messages sent by the other failed processes.

Lemme 4 *In each round, at least one new maximum commitable state interval is known.*

Proof We follow the same reasoning as before. Let B be the set of maximum state intervals not found at the beginning of round k. There is at least one commitable state interval sc_y in B that does not depend on any other state interval in B. So sc_y is found at the end of round k.

Lemme 5 *The set of maximum commitable states form the maximum consistent global state.*

Proof By definition a consistent global state is a state without orphan process. So the set of maximum commitable states is a consistent global state. We conclude easily that it is the maximum consistent global state.

Proof of Theorem 1 Consider r concurrent failures. We deduce from Lemme 1 and Lemme 2 that at the end of the stabilization phase, the non-failed processes are rolled-back to their maximum commitable state intervals. We can conclude from Lemme 3 and Lemme 4 that the maximum commitable state interval of the failed processes is found in at most r rounds. It follows directly from Lemme 5, that O2P restores the application in its maximum consistent global state.

5 Additional Remarks

Some aspects of the protocol have not been addressed in the previous sections. They are briefly explained in this section.

5.1 Size of the Dependency Lists

We can imagine bad scenarii where communications between processes are so frequent that the acknowledgments of logging never arrive before next message sending. In this case, the dependency list of the processes would grow and the amount of data piggybacked on each message too. We could reach a state where dependency lists would be bigger than dependency vectors. To solve this problem, a solution can be to fix the maximum size of the dependency lists. If a process has a too big dependency list, its message sending is blocked until the size of its dependency list decreases.

5.2 Checkpointing

Combining message logging and checkpointing is very attractive since checkpoints don't need to be coordinated. Thus, problems of concurrent access to stable storage to write checkpoints are avoided. Checkpointing can also be seen as a way to limit the size of the logs since only the logs corresponding to messages delivered since the last checkpoint need to be stored. The only constraint is that to be valid a checkpoint must be done on a commitable state interval.

5.3 Fast Output Commit

The problem when sending message to the outside world is that the outside world cannot rollback. So before sending a message to the outside world, the application must be sure that it will never be rolled-back, i.e. the state interval of sending is commitable. Due to the use of dependency lists and of the forwarding of acknowledgments of stable determinants, a process knows as soon as possible when one of its state intervals becomes commitable. So messages sent to the outside world can be sent as soon as possible.

5.4 Detecting Duplicated Messages

When a process replays its execution after a failure, it sends messages that it has already sent before the failure. The receiver of the message may have already delivered it or may have lost it, if it has failed to. We need to be

sure that a process does not deliver the same message twice. Without FIFO channels, it would be very difficult to solve this problem because it would be impossible to differentiate a delayed message from a duplicated message without keeping a log of all the messages received by the process. With FIFO channels, a process only has to keep the ssn of the last message it has delivers from each process of the application. If it receives a message with a lower ssn than the one kept for the sender, it's a duplicated message.

6 Related Work

Strom and Yemini [12] were the first to present optimistic message logging. Their protocol uses dependency vectors piggybacked on each application message to keep up to date causal dependencies between processes. This dependency vectors have to be logged with the messages on stable storage. The main problem of this protocol is that a single failure can make another process rollback an exponential number of times. Sistla and Welch [10] presented a quite similar protocol but that do not suffer from the exponential rollback problem. In [10], they prove that there is a unique maximal consistent global state that can be recovered from stable storage. Peterson and Kearns [9] were the first to use vector clocks to qualify dependencies between state intervals. Vectors of size n are piggybacked on each message and saved on stable storage. But this first protocol only tolerates one fault. Other protocols improve the use of vector clocks to tolerate n faults by distinguishing user level time and system time [11] or by introducing fault tolerant vector clocks [4]. The amount of data piggybacked by these protocols on each message depends on the number of processes in the application.

Johnson and Zwaenepoel [7] manage to deal with multiple faults with piggybacking only state interval of sending of each message. But they use a centralized recovery algorithm which needs to be made fault tolerant. Sistla and Welch [10] also presented a protocol, from which is inspired our recovery protocol, that uses only direct dependencies and tolerates a single failure.

7 Conclusion

In this paper we have presented O2P, an extremely optimistic message logging protocol that tolerate an arbitrary number of concurrent failures. Extremely optimistic means that it applies the assumption made in pessimistic protocols to an optimistic protocol. The benefit is that the amount of data piggybacked on each message by the protocol do not depends on the number

of processes in the application. Thus the protocol is well adapted to large scale applications. When the extremely optimistic assumption is valid, there is nothing to piggyback on the messages. Since it's an optimistic protocol, it induces no overhead on failure free execution.

Using sender-based message logging, O2P limits as much as possible the size of the data needing to be saved on stable storage. Furthermore it ensures fast output commit due to its way of tracking commitable state intervals.

O2P has only been tested through simulations, but it would be interesting to test it in a real implementation to measure the benefits of using it according to the characteristics of the applications and of the infrastructure.

References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[2] A. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[3] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] O.P. Damani and V.K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. pages 108–115, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[5] E. N. (Mootaz) Elnozahy, L. Alvisi, Y.M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[6] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[7] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[8] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[9] S. L. Peterson and P. Kearns. Rollback Based on Vector Time. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 68–77, 1993.

[10] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 223–238, New York, NY, USA, 1989. ACM Press.

[11] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 361–371, Pasadena, California, 1995.

[12] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computing Systems*, 3(3):204–226, 1985.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399