On the Effectiveness of Distributed Checkpoint Algorithms for Domino-free Recovery

Franco Zambonelli
Dipartimento di Scienze dell'Ingegneria – Università di Modena
Via Campi 213-b – 41100 Modena – ITALY
franco.zambonelli@unimo.it

Abstract

The paper focuses on fault-tolerant distributed computations where processes can take local checkpoints without coordinating with each other. Several distributed on-line algorithms are presented which avoid roll-back propagation by forcing additional local checkpoints in processes. The effectiveness of the algorithms is evaluated in several application examples, showing their limited capability of bounding the number of additional checkpoints.

1. Introduction

In[1] a distributed computation composed of several communicating processes, the capability of recovering from a fault can be achieved by making processes periodically checkpoint and save their computational state to a stable storage [17, 7]. Then, in the case of a fault (either due to a node crash or to a failure in the communication network), the distributed execution can be restored by restarting the execution of each process from one of its local checkpoints, to form a so called *global checkpoint* [2].

The re-execution of a failed program requires the global checkpoint to be *consistent* [12, 14], i.e., not to include any local checkpoint that is dependent on an event happened after the global checkpoint. Inconsistency arises when there are messages that have been received before the global checkpoint but have not been sent yet: this situation cannot correspond to any past state of the failed execution. A further requirement relates to *in-transit messages*, i.e., messages that have been sent by a process before the global checkpoint and are going to be received after. Because these messages will not be re-sent in the restored execution, they must have been previously logged to a stable storage, to be delivered during re-execution. This paper focuses on the consistency issue and neglets message-logging, forwarding the interested reader to [1] for a detailed analysis of message-logging algorithms and related problems.

To permit one process to consistently restore its execution from its latest local checkpoint before the fault, one must grant that all its local checkpoints are $useful$, i.e., can belong to at least one consistent global checkpoint [18]. Otherwise, the execution of the process must be rolled-back in the past until a useful local checkpoint is found from which to build a consistent global checkpoint. Roll-back propagation, often called the $domino$ effect because of its recursive nature, limits forward execution progresses in presence of faults [16].

Several works in the area address the problems of building consistent global checkpoints by making communication drive the local checkpointing activity. Coordinated checkpointing schemes grant that all the local checkpoints belong to a consistent global one and, then, fully avoid domino effects, via a global coordination of the local checkpoint activity [5, 8]. However, those schemes require additional control messages to be included in the execution and are not applicable when processes need to checkpoint independently of each other. If processes are allowed to checkpoint independently of each other, the local checkpoint activity must be coupled with algorithms in charge of avoiding, or at least limiting, domino effects [9, 13, 19].

This paper deals with on-line algorithms that grant domino-free recovery by monitoring the application execution and by forcing additional local checkpoints in processes, when the arrival of one message is likely to make some local checkpoint useless. Several well known checkpoint algorithms are presented and integrated within a single theoretical framework. The effectiveness of the al-

gorithms is evaluated in a heterogeneous set of message-passing applications. The main result is that none of the algorithms shows itself capable of reasonably limiting the number of forced checkpoints, thus introducing a high overhead on applications. In addition, the paper shows that, in most cases, simple algorithms perform as well as complex ones.

The paper is organized as follows. Section 2 describes the adopted computational model and introduces some basic definitions and theorems. Section 3 presents the analyzed checkpoint algorithms. Section 4 evaluates their effectiveness in several applications.

2. Background Definitions and Theorems

We model a distributed computation as a finite set of processes $P_1, P_2, \ldots, P_n$ that communicate by message-passing. Processes can execute either internal events or communication events. Communication events include the sending of a message m ($Send(m)$) and the receiving of a message m ($Recv(m)$). Every receive event implies a send event. The send event of a message m is tied to the receive event of the same message by the $\xrightarrow{M}$ relation, i.e., $Send(m) \xrightarrow{M} Recv(m)$. Communications between two processes are not required to be FIFO.

The internal events of a process are sequentially ordered in time: this can be expressed by the $\xrightarrow{SO}$ relation.

Events in a distributed computation are partially ordered in time by the *happened before* relation, defined as the transitive closure of the union of the relations $\xrightarrow{M}$ and $\xrightarrow{SO}$:

$$\xrightarrow{HB} = (\xrightarrow{SO} \bigcup \xrightarrow{M})^+$$

If two events a and b are ordered w.r.t. to the $\xrightarrow{HB}$ relation, i.e., $a \xrightarrow{HB} b$, then there is a so called *causal path* from a to b.

2.1. Consistent Global Checkpoints

In a checkpointed execution of a distributed application, internal events of a process include the local checkpoints. Let us indicate as $C_{i,x}$ the x^{th} checkpoint taken by a process P_i. We call the x^{th} checkpoint interval of a process the set of all the events included between $C_{i,x}$ and $C_{i,x+1}$. Figure 1 shows a graphical representation of a checkpointed execution: horizontal lines represent the execution flows of processes (evolving in time from left to right), black dots the local checkpoint events, inter-process arrows the messages exchanged between processes.

A *global checkpoint* is a set of checkpoints, one for each process of the computation. A global checkpoint is *consistent* if there is not any local checkpoint included in it that happened before another local checkpoint included in the global one. With reference to figure 1, the global checkpoint including $C_{1,1}, C_{2,1}, C_{3,1}$ is consistent; the global checkpoint including $C_{1,3}, C_{2,2}, C_{3,2}$ is not because of the message $m7$, received by P_3 before the global checkpoint and sent by P_2 after it.

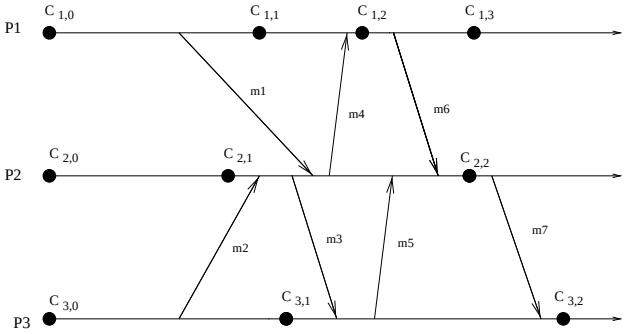

Figure 1. Graphical representation of one message-passing execution

Netzer and Xu (see [14]) analyzed the necessary and sufficient conditions for a local checkpoint to belong to a consistent global one. This analysis is based on the introduction of the ZigZag relation ($\xrightarrow{Z}$) between local checkpoints, that can be considered as an extension of the happened before relation. Given two checkpoints $C_{i,x}$ and $C_{j,y}$, a ZigZag-path exists between them ($C_{i,x} \xrightarrow{Z} C_{j,y}$) if:

- P_i sent a message m_1 after $C_{i,x}$
- there exist a chain of message m_2, ..., m_n, such that m_{i-1} is received by a process in the same checkpoint interval during which it sent m_i (either after or before) or in a previous one;
- m_n is received by process P_j before $C_{j,y}$

With reference to figure 1, $C_{3,2}$ has a ZigZag relation with $C_{1,0}$, because of the messages $m1$ and $m3$. Note that the ZigZag relation includes but it is not equivalent to the *happened before* relation.

As a peculiar case of the ZigZag relation, a local checkpoint is said to be involved in a ZigZag-cycle if it has a ZigZag-path to itself. Checkpoint $C_{1,2}$ in figure 1 is involved in a ZigZag-cycle, because of the messages $m6$ and $m4$.

Starting from the above definitions, Netzer and Xu proved that two local checkpoints can belong to the same consistent global checkpoint if and only if they are not tied by a ZigZag relation [14]. Consequently, it can be easily proved that a local checkpoint can belong to a consistent global one, i.e., it is useful, if and only if it is not involved in any ZigZag-cycle.

2.2. Global Checkpoints and Logical Clocks

Lamport's logical clocks [10] are a very simple and effective way to capture causal dependencies among processes. Let any process P_i have a logical clock lc_i associated with the internal events of its execution and incremented at each event. If the lc_i value is piggybacked with each message m (indicated as $m.lc$) and updated at the receipt of a message in the following way: $lc_i = max(lc_i, m.lc)$ then, if any two events of one execution, e_1 and e_2, are in a happened before relation ($e_1 \xrightarrow{HB} e_2$), one has necessarily: $e_1.lc < e_2.lc$

In the context of a checkpointed distributed computation, the only internal process events of interest are the local checkpoints. Then, apart from being updated at the receipt of a message, the logical clock of a process can be incremented only at local checkpoint events. When a process takes a checkpoint, it increments its local clock: $C_{i,x}.cl = cl_i = cl_i + 1$, where $C_{i,x}.cl$ indicates the logical clock of the checkpoint $C_{i,x}$. Note that with these assumptions, if two checkpoints $C_{i,x}$ and $C_{j,y}$ are connected by a causal path ($C_{i,x} \xrightarrow{HB} C_{j,y}$) one has necessarily: $C_{i,x}.cl < C_{j,y}.cl$

By considering the ZigZag relation between checkpoints as an extension of the happened before relation, one can think of extending the role of logical clocks too, in order to take into account the ZigZag relation, other than the happened before one. If an execution permits to have an extended logical clock mechanism with the property of capturing causal dependencies to ZigZag dependencies, the execution cannot have any checkpoint involved in a ZigZag-cycle and, then, any local checkpoint is granted to be useful. This is stated in the following thorem, firstly formulated in [6].

Theorem: the checkpointed execution of a distributed application does not have any local checkpoint involved in a ZigZag-cycle IF AND ONLY IF it is possible to assign logical clocks to processes so to grant that for any two checkpoints $C_{j,y}$ and $C_{k,z}$ such that $C_{j,y} \xrightarrow{Z} C_{k,z}$ then $C_{j,y}.cl < C_{k,z}.cl$.

Proof of necessity (ONLY IF part): Let us suppose that the computation does not include any ZigZag-cycle and still one cannot assign logical clocks to processes without having at least two checkpoints $C_{j,y}$ and $C_{k,z}$ such that $C_{j,y} \xrightarrow{Z} C_{k,z}$ and $C_{j,y}.cl >= C_{k,z}.cl$. However, the only reason why such an assignment cannot be done is because of the presence of a third checkpoint $C_{i,x}$ such that $C_{i,x} \xrightarrow{Z} C_{k,z}$ forcing $C_{i,x}.cl > C_{k,z}.cl$ and $C_{j,y} \xrightarrow{Z} C_{i,x}$ forcing $C_{j,y}.cl > C_{i,x}.cl$. For the transitivity of the ZigZag relation this would also imply: $C_{i,x} \xrightarrow{Z} C_{i,x}$, that contradicts the assumption.

Proof of sufficiency (IF part): If for any pair of checkpoints such that $C_{j,y} \xrightarrow{Z} C_{i,x}$ we have $C_{j,y}.cl < C_{i,x}.cl$, the presence of a checkpoint $C_{k,z}$ involved in a ZigZag-cycle, i.e., $C_{k,z} \xrightarrow{Z} C_{k,z}$, would also imply $C_{k,z}.cl > C_{k,z}.cl$, which is impossible.

3. Checkpoint Algorithms

The above theorem is a powerful framework for the design and the implementation of on-line algorithms to avoid roll-back propagation in parallel/distributed applications. If an algorithm is capable of granting an ordering relation on logical clocks for all the checkpoints involved in a ZigZag-path, it will also grant that no ZigZag-cycles will ever form and, then, that every local checkpoint will be useful.

To grant the above property, an algorithm must on-line monitor the evolution of an application execution and, on the basis of the available information, detect whether "unsafe" ZigZag-path, i.e., ZigZag-path between checkpoints whose logical clocks are not ordered, can possibly occur and, then, prevent them to be formed.

The algorithms presented in the paper are fully distributed (decisional activity is local to each process), and do not require additional messages to be exchanged between processes but the one intrinsic to the applications (the algorithms exchange information between processes by piggybacking it into the application messages). In addition to the piggybacked information, the algorithms can locally store information about the state of the computation. Depending on the amount of exploited information, either locally stored or piggybacked, several algorithms can be conceived.

The triggering event of the algorithms is the reception of a message. When a message is going to be received by one process, the algorithms, before effectively delivering the message to the process, analyze the locally stored information and the information piggybacked with the message. The aim is to detect whether delivering the message to the process would possibly create an unsafe ZigZag-path and, then, it would be likely to make some local checkpoint useless. In this case, to prevent the formation of the unsafe ZigZag-path, the algorithms force the receiver process to take an additional checkpoint before delivering the message. The receipt of piggybacked information may also make it necessary to update the locally stored information.

Summarizing, the general scheme of the algorithms can be sketched as follows:

```
msg=receive();

if(possibly_unsafe(msg.info,local_info))
   take additional local checkpoint;
fi
```

```
update_local_info(msg.info,local_info);

deliver msg;
```

where *msg.info* indicates the information piggybacked with the message msg and the *possibly_unsafe* function detects whether the incoming message is likely to create an unsafe ZigZag-path.

The following of this section details several algorithms with different amounts of locally stored and piggybacked information (from non-informed algorithms to highly-informed ones) and, consequently, with different conditions for detecting possibly unsafe ZigZag-path and different methods for updating the local information. Most of the algorithms have been already presented in the literature, though neither integrated within a single framework nor evaluated.

3.1. A $send$-based Algorithm

If no information at all about the state of the execution is either stored or piggybacked, the only way to grant all checkpoints to be useful is to checkpoint either before every receive or after every send event, to fully avoid the presence of ZigZag-path. However, the scheme is impracticable because of the too high overhead it imposes on applications. Nevertheless, still without piggybacking any information, it is possible to locally store within each process some information in order to reduce the number of additional forced checkpoints.

An incoming message is likely to make the receiver process involved in some ZigZag-path only if it has already sent other messages during the same checkpoint interval. Then, each process P_i can locally store a boolean information ($sent_i$) to keep track of the occurrence of send events during checkpoint intervals: $sent_i$ is set to $FALSE$ at each new local checkpoint and set to $TRUE$ at the first send event. A message going to be received by a process possibly forms an unsafe ZigZag-path and, then, has to force an additional checkpoint before its deliver, only if

$$sent_i = TRUE$$

A similar checkpointing scheme has been proposed in [17].

3.2. A $clock$-based Algorithm

By permitting to piggyback at least one scalar information with each message (i.e., the logical clock of the sender) each process can, at the receipt of a message, analyze it to detect in a more accurate way whether the message is likely to create some unsafe ZigZag-path.

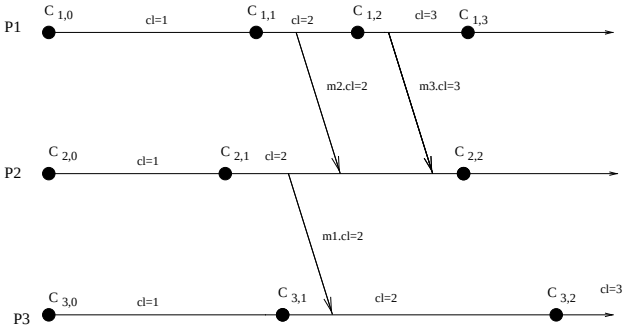

Figure 2. Unsafe ZigZag-path in an execution

When one message m is received by one process P_i, the piggybacked logical clock is checked against the local logical clock. A checkpoint is forced before delivering the message if:

$$m.cl > cl_i$$

In fact, the logical clock of the incoming message provides the information that some checkpoint exists, say in process P_k, with a clock greater than the current clock of P_i. Because P_i may have sent a message to a process (say P_j) in the same checkpoint interval, this is likely to create a ZigZag-cycle between a checkpoint x in P_k and a checkpoint y in P_j such that $C_{k,x}.cl \geq C_{j,y}.cl$.

With reference to figure 2, when P_2 receives message $m2$ with $m2.cl = 2$, its logical clocks is already 2. Then, the receipt of $m2$ cannot create any unsafe ZigZag-path. Instead, when message $m3$ is received with $m3.cl = 3 > cl_2 = 2$, this is likely to cause (to the knowledge of P_2) an unsafe ZigZag-path. In the figure, this unsafe ZigZag-path is effectively formed between $C_{1,2}$ and $C_{3,2}$. The algorithm prevents it by forcing an additional checkpoint before the deliver of $m3$, independently of $m1$.

The algorithm proposed in [9] follows a similar scheme, though the clock information is substituted by message coloring: both internal process events and computation messages are "colored"; a process is forced to take a local checkpoint if it receives a message with a color different from its current one.

3.3. A $clock + send$-based Algorithm

The above two algorithms can be integrated in a single scheme to prevent in a more informed way the formation of unsafe ZigZag-path and to reduce the number of additional forced checkpoints. Let any process locally store the boolean information about messages sent to other processes; let any message piggyback the logical clock information. Then, a message m incoming to a process P_i makes it necessary to force a checkpoint only if some message has been previously sent by P_i in its current checkpoint interval

and the clock of the incoming message is greater than the current clock of the receiver. Formally:

$$(sent_i = TRUE) \wedge (m.cl > cl_i)$$

With reference to figure 2, the algorithm forces a checkpoint only at the receipt of $m3$, because of both the message $m1$ sent in the same checkpoint interval and the greater logical clock of the incoming message $m3$.

Similar checkpointing schemes, specialized for particular application areas, are presented in [4] and [11].

3.4. A Full Informed Algorithm

The above described algorithms do not still exploit all the possible information about the state of the execution that could be made on-line available to a process.

When a process P_i receives a message m such that $(m.cl > cl_i) \wedge (sent_i = TRUE)$, a forced checkpoint may not be necessary if P_i has a way to detect that the logical clocks of the processes to which it has previously sent messages in the current checkpoint interval were already – in the checkpoint interval during which these processes received the above messages – greater than or equal to the logical clock of the incoming message m. In this case, the incoming message does not create any unsafe ZigZag-path, because the checkpoints involved in it are already ordered w.r.t. their logical clocks.

With reference to figure 2, when P_2 receives $m3$, it has to force a checkpoint because it does not know whether the logical clock of P_3 (to which it previously sent the message $m1$ with a smaller logical clock) were either equal to or less than the logical clock of $m3$ at the time it received (or will receive) $m1$. Figure 3 reports a similar communication pattern in which the logical clock of P_3, in the checkpoint interval during which it receives the message from P_2, is already equal to the one of the message $m3$: in this case, the ZigZag path is safe and a forced checkpoint is not necessary. The information about the logical clock of the process P_3 could have been made available to P_2 by a causal path started from P_3 and arriving at P_2 via $m3$. In figure 3, message m_x – represented by a dashed line – realizes this kind of causal path.

To keep track of the above identified condition, each process must store and piggyback a boolean vector of size N (where N is the number of application processes). The j^{th} components of the vector (called $increased$) keeps track of whether the current logical clock of the process that store the vector (or of the message that piggybacks it) is increased with respect to the last known value of the logical clock of process P_j. Furthermore, to permit one process to exploit the information provided by the $increased$ vector, it is necessary to detail in each process the identity of the processes

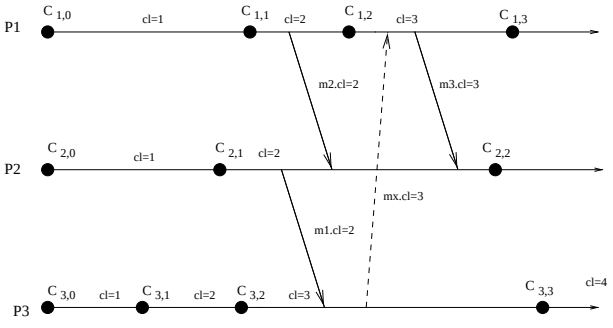

Figure 3. A safe ZigZag-path detected by the full informed algorithm

to which messages have been sent during a checkpoint interval, i.e., each process must store no longer a simple boolean $sent_i$ but, instead, a boolean vector of size N, whose j^{th} component takes into account whether a message has been sent to process P_j in the current interval.

The above introduced information leads to the following sufficient condition ($C1$) for forcing a checkpoint:

$$\forall k \neq i \exists k : (sent_i[k] = TRUE) \wedge$$

$$\wedge (m.cl > cl_i) \wedge (m.increased[k] = TRUE)$$

The condition $C1$ applies only in case the causal information $m.increased[k]$ arriving at a process P_i (that has sent a message $m1$ to process P_k in the current checkpoint interval) started from P_k during the same checkpoint interval in which P_k received $m1$ or in a previous one. Let us suppose, instead, that the information started from P_k in a later checkpoint interval than the one including the receive event of $m1$. In this case, independently of the value of $m.increased[k]$, the receipt of m would make useless all the checkpoints of P_k included between the receipt of $m1$ and the moment when the causal path, ending at P_i with m, starts from P_k. In fact, the receipt of m would complete a ZigZag-path from a checkpoint of P_k to a previous checkpoint of P_k that, by construction, has a smaller logical clock.

With reference to figure 1, the path that starts from P_2 with message $m4$ and completes back to P_2 in the same checkpoint interval (via message $m6$) involves $C_{1,2}$ in a ZigZag-cycle (though $m6.increased[1] = FALSE$ and, then, condition $C1$ is $FALSE$ at the receipt of $m6$).

To avoid these patterns, additional information has to be stored and piggybacked. In particular, it is necessary, for each process, to store and piggyback one integer vector of size N (called $ckpt$), updated on a component wise maximum scheme at a receipt of a message, as a vector of logical clocks [2]. This vector permits to detect all causal path. In addition, to detect whether a causal path includes

Program	Execution Time (sec)	Exchanged Data (Mbytes)	Avg. Message Size (bytes)
matrix determinant	48.3	43.1	3183
fast fourier transform	417.0	243.2	23233
finite differences	199.4	19.0	1241
circuit test generator	144.0	35.2	1641

Table 1. Description of the test programs

at least one checkpoint, an additional boolean vector (called $include$) has to be stored and piggybacked. The j^{th} component of the vector in process P_i ($include_i[j]$) takes into account whether a causal path from P_j to P_i exists that includes at least one checkpoint.

On the basis of the above information, a message m received by a process P_i has to force a checkpoint if (condition C2):

$$(include_i[i] = TRUE) \wedge (ckpt_i[i] = m.ckpt[i])$$

i.e., the message m derives from a causal path started from P_i during the same checkpoint interval and including at least one checkpoint, thus leading to a ZigZag-cycle.

By composing conditions $C1$ and $C2$, a message m received by a process P_i has to force a checkpoint if and only if:

$$(C1) \vee (C2)$$

The algorithm presented in [6] is equivalent to the above presented one, though requiring a larger amount of information to be stored and piggybacked.

4. Performance Evaluation

To evaluate the effectiveness of the presented algorithms, measured by the percentage of forced checkpoints they induce in applications in addition to the local checkpoints autonomously taken by the application processes, I adopted several message-passing programs (developed on a 16-nodes Intel $iPCS860$ hypercube) as testbeds and simulated the execution of the algorithms from message traces of their executions.

The first application computes the determinant of a 600×600 matrix. The second performs the fast fourier transform on a group of randomly generated set of data points. The third computes finite differences over a 900×900 grid to solve a differential equation. The last application is a circuit test generation program to detect faulty circuits.

The programs are heterogeneous in terms of both execution times and amount of data exchanged between processes (see table 1). In addition, the communication patterns they exhibit in executions are very different and representative of a larger class of parallel and distributed programs.

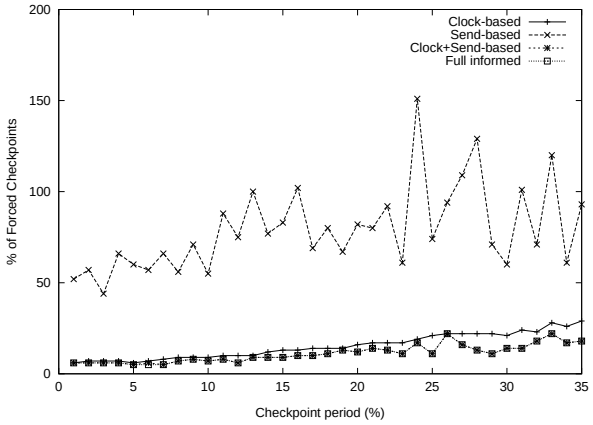

Figure 4. Percentage of forced checkpoints in the matrix determinant application

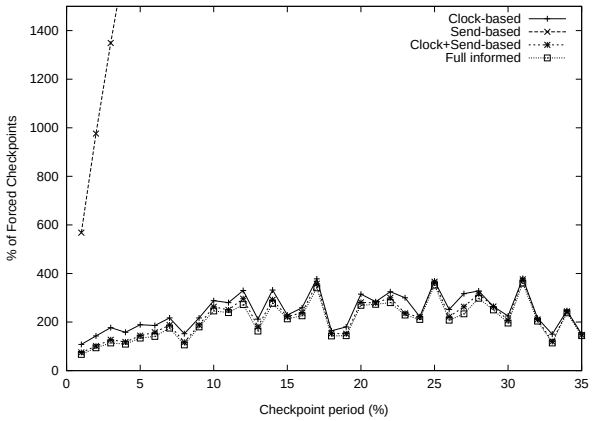

Figure 5. Percentage of forced checkpoints in the fast fourier transform application

To test the behavior of the checkpoint algorithms, I introduced artificial local checkpoints in the message traces. Different traces have then been produced with different checkpoint periods, from a 1% to a 35% of the application total execution time. To my opinion, this range covers all practicle cases: from very short checkpoints periods (of a few seconds) to checkpoint periods that cause applications to checkpoint only two or three times. To simulate the behavior of un-coordinated checkpointing, local checkpoints have been inserted in the traces with a small random skew from their basic time period.

4.1. Results

Figure 4 shows the results for the matrix determinant application. The send-based algorithm behaves badly, by forc-

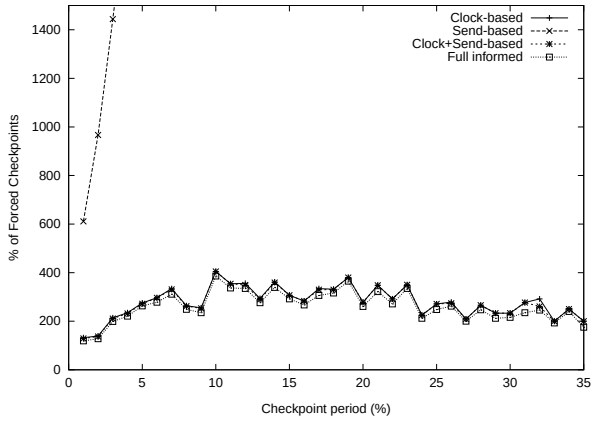

Figure 6. Percentage of forced checkpoints in the finite differences application

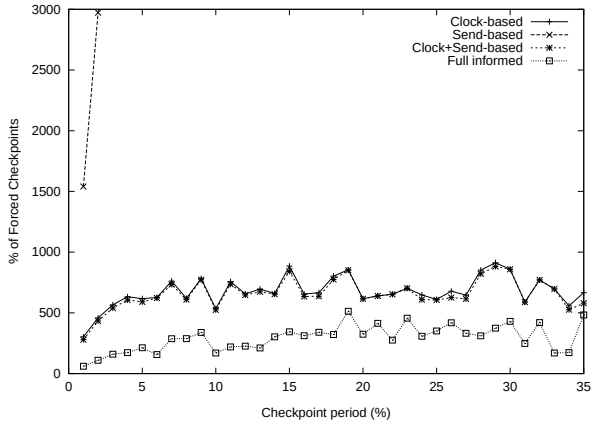

Figure 7. Percentage of forced checkpoints in the circuit test application

ing a high number of additional checkpoints (from 45% to 160%). This limited effectiveness is easy to explain: once a process has sent a message in a checkpoint interval, all further receive events in the same interval induce a forced checkpoint. The other algorithms works better and the number of forced checkpoints is highly reduced (from 5% up to 24%). The clock+send-based algorithm does not significantly improve the simpler clock-based algorithm: the few checkpoints that are saved w.r.t. to the clock-based algorithm correspond to those few receive events that follow no send events in the same checkpoint interval.

Surprisingly, the full informed algorithm does not achieve any improvement over the clock+send based one (the corresponding plots in figure 4 fully overlap). This result is justified by the fact that the communication patterns expressed in the application are very simple and regular (basically, divide and conquer trees). Then, the additional information potentially available to the algorithm cannot be exploited in any way to reduce the number of forced checkpoints.

In the fast fourier transform application (see figure 5), the higher complexity of the communication patterns makes the percentage of forced checkpoints grow for all the algorithms. For the send-based algorithm, this percentage grows over 6000%. For the other algorithms, it is between 180% and 400%. Again, as in the case of the matrix determinant application, the increase of the information available to algorithms does not lead to an analogous decrease of the number of forced checkpoints: the clock+send-based algorithm improves the clock-based of only a 5%-10%; the full informed algorithm achieves only a minimal further improvement. Very similar comparative results apply for the finite differences application (see Figure 6). These results stem from the fact that, though the communication patterns are somehow more complex in these two applications, they are still very regular and do not exhibits the peculiar causal path that would permit to exploit the additional information available to the full informed algorithm.

The only application in which the full informed algorithm performs significantly better than the other ones is the circuit tester one (see figure 7). Though the overall number of forced checkpoints is very high for all the algorithms, the fully informed algorithm takes about 50% less checkpoints than the clock+send-based algorithm. This can be explained by the fact that, in this application, the communication patterns are very complex (any process communicates with all other ones in an irregular way) and, then, the full informed algorithm can effectively exploit its additional information.

As a general remark, one can note that the performance of the algorithms are mostly independent on the local checkpointing period (apart in the case of the matrix determinant application, in which a slight dependency is evident).

4.2. Discussion

The above considerations can be summarized in the following points:

- the send-based algorithm is absolutely not capable of limiting the number of forced checkpoints;
- the clock- and the clock+send-based algorithms perform similarly and significantly decrease the number of forced checkpoints w.r.t. to the send-based algorithm;
- the full informed algorithm improves the simpler algorithms only in case of very complex and irregular communication patterns;

- the number of additional checkpoints forced by the algorithms is generally high w.r.t. to the number of local (non-forced) checkpoints, independently of the local checkpoint period.

The above results suggest to limit the use of the presented algorithms to exceptional cases. In fact, the very high number of checkpoints forced in applications may cause intolerable overhead and slow down the application execution. Coordinated checkpointing schemes have to be preferred: though they have to pay the cost of additional control messages in applications to coordinate the checkpoint activity, they induce a controllable number of checkpoints while guaranteeing their usefulness. Uncordinated checkpointing schemes should be adopted only when coordinated ones would be not applicable, i.e., when application processes need to checkpoint independently to each other. In these cases, unless the communication patterns expressed in the application are very irregular, the adoption of the simple $clock + send$-based algorithm would be enough.

5. Conclusions and Future Work

The paper presents several distributed algorithms to grant domino-free recovery in message-passing applications based on un-coordinated checkpointing schemes. This is done by forcing, when necessary, additional local checkpoints in processes.

The paper evaluates and compares the effectiveness of these algorithms for several message-passing applications. The main result is that all algorithms, even highly-informed ones, force a large number of additional checkpoints. This may cause intolerable overhead on applications and can make, in most cases, coordinated schemes preferred.

I am currently investigating heuristic checkpoint algorithms that, even if they cannot grant that all the local checkpoints are useful, can effectively limit the domino effect while forcing a very limited number of additional checkpoints [13]. In addition, I am evaluating whether the presented checkpoint algorithms can be exploited to support – together with message logging algorithms – efficient incremental replay of distributed applications [15]. Both the above research topics are likely to benefits from recent works on roll-back dependency tractability, aimed at better formalizing the characteristic of checkpointed executions in dependence of the expressed communication patterns [3, 18].

Acknowledgments

I'd like to thank Robert Netzer for his precious suggestions and criticisms and for having made me available the trace files. This work has been supported by the Italian Ministero dell' Universita' e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications".

References

[1] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[2] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems. In S. J. Mullender, editor, *Distributed Systems*, pages 55–96. ACM press, 1993.

[3] R. Baldoni, J. Helary, A. Mostefaoui, and M. Raynal. Consistent checkpointing in message passing distributed systems. Technical Report 2564, INRIA, Rennes, FR, June 1995.

[4] D. Briatico, A. Ciufoletti, and L. Simoncini. Distributed domino-effect free recovery algorithm. In *4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, October 1984.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 9(1):63–75, February 1985.

[6] J. Helary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *16th IEEE Symposium of Reliable Distributed Systems*, October 1997.

[7] K. Kim, J. You, and A. Abouelnaga. A scheme for coordinated execution of independently designed recoverable distributed processes. In *16th Symposium on Fault-Tolerant Computing Systems*, pages 130–135, 1986.

[8] R. Koo and S. Toueg. Checkpointing and roll-back recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[9] T. Lai and T. Yang. On distributed snapshots. *Information Processing Letters*, 3(25):153–158, May 1987.

[10] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] D. Manivannan and M. Singhal. A low overhead recovery technique using quasi-synchronous checkpointing. In *16th Conference on Distributed Computing Systems*, pages 100–107, May 1996.

[12] D. Manivannan, M. Singhal, and R. H. B. Netzer. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6), June 1997.

[13] R. Netzer and J. Xu. Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symposium on Parallel and Distributed Processing*, pages 754–761, December 1993.

[14] R. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[15] R. Netzer and Y. Xu. Replaying distributed programs without message logging. In *6th IEEE Symposium on High-Performance Distributed Computing*, August 1997.

[16] B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 1(2), February 1975.

[17] D. L. Russell. State restoration in systems of communication processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, February 1980.

[18] Y.-M. Wang. Consistent global checkpoints that contains a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.

[19] Y.-M. Wang and W. Fuchs. Scheduling message processing for reducing rollback propagation. In *IEEE Fault-Tolerant Computing Symposium*, pages 204–211, July 1992.