Try, try till you succeed: Multiple checkpointing and rollback in distributed systems

Vinit A. Ogale

Dept. of Electrical and Computer Engineering
University of Texas at Austin, TX 78712

Abstract

We present a multiple checkpointing and recovery protocol for fault tolerance in distributed systems. This is useful in scenarios where a fault trigger may result in observable undesirable effects after indeterminate time, i.e., when *dormant faults* are observed. The primary assumption is that the fault trigger occurs in rare circumstances and it is highly probable that the fault will not reoccur in another run. To efficiently collect and store global snapshots we propose an online algorithm for slicing a computation w.r.t. conjunctive predicates.

1 Introduction

In this paper we deal with tolerating software faults in distributed systems. Software faults are extremely difficult, if not impossible to avoid. Hence a distributed system which can tolerate software faults with minimum runtime overhead, is highly desirable. In this paper we consider faults which can be circumvented by re-executing the program. We also assume that faults can be detected eventually. Some common examples of such faults are channel faults, resource availability and bad states in probabilistic algorithms. We allow the delayed detection of such faults, i.e., the fault may be detected long after it has actually occured. We call these type of faults *dormant faults*.

One of the popular techniques for fault tolerance is checkpointing and rollback. Many approaches have been proposed for global checkpointing and rollback in distributed systems [3]. Elnozahy et. al. [2] give a comprehensive survey of

Email address: ogale@ece.utexas.edu (Vinit A. Ogale).

many of the popular rollback recovery protocols for message passing networks. In [11], the authors describe an optimization of the normal 'checkpoint and recover schemes' where the recovery is carried out in different stages. If a local recovery does not overcome the bug then their protocol goes on increasing the scope of the rollback till it finally rolls back to a global consistent cut. This overcomes many of the transient non-deterministic faults, i.e., the *Heisenbugs* and also some deterministic bugs which can be overcome by reordering of messages [7].

However in some situations, this may not solve the problem. This is especially true when the bugs are *dormant*, i.e., the bug results in undesirable system behavior after executing normally for some time. A solution to this is maintaining checkpoints corresponding to multiple global consistent snapshots. If recovery from the latest snapshot does not circumvent the bug, then we go on rolling back to previous snapshots until the bug is bypassed. However the number of consistent global snapshots could be exponential in the number of events at each process. To efficiently store the snapshots we store the slice [6,9] of the set of all consistent global snapshots. Offline and centralized online algorithms for slicing have been presented in [10]. In this paper we present an online algorithm for slicing a distributed system. We use this algorithm to maintain a representation of the set of consistent global snapshots for rollback when required.

The main contributions we make in this paper are:

(1) We present an efficient protocol for fault tolerance when dormant bugs may be present.
(2) We give an online distributed algorithm for slicing a distributed computation. This can be used for other applications including predicate detection.
(3) We have implemented the framework, using the proposed protocol, for running a faulty distributed program. We conclude that the proposed scheme is practical and the overhead for fault tolerance is reasonably low.

Such a system would be very useful for distributed, computation intensive programs. In such programs a slow system recovery in case of faults is tolerable but a total system restart would be very undesirable. Some examples of such applications are weather modeling and movie special effect generation. Another important advantage of the proposed scheme is from a software engineering perspective. Using a fault tolerant framework, we could use simple algorithms which function correctly most of the the time. Whenever we detect an incorrect run we rollback and rerun the algorithm. This may lead to better programs and avoid unexpected behavior from incorrect implementations of complex algorithms.

Our protocol can be used in tandem with existing optimistic recovery algorithms including progressive retry [11]. The organization of this paper is as follows: Section 2 gives the overview of the multiple rollback scheme. In Section 3 we present the distributed online slicing algorithm. The overhead analysis is presented in Section 4 and Section 5 contains the simulation results from the implementation of our algorithm.

2 Overview

2.1 System Model and Notation

We assume a distributed asynchronous message passing system. The local trace at each process consists of a sequence of ordered events. Events can be of three types:

(1) Send: which sends a message to some process
(2) Receive: receiving a message from some process
(3) Internal: internal computation

We assume the presence of a distinguished process or unique ids for each of the processes.

2.2 Approach

The main idea of our rollback protocol is as follows:

(1) All processes take regular local checkpoints.
(2) We have a mechanism for determining which checkpoints can be a part of any consistent global snapshot and maintaining the list of consistent snapshots. In this paper we consider transitless global snapshots only. This simplifies the algorithm and eliminates the need for message logging.
(3) When a software fault occurs at process i, it initiates the rollback mechanism. If a previous fault had occurred in the system when the global state was v_{old} and the state when the new fault occurs is $v : \neg(v < v_{old})$ then the system recovers to the maximum global state.
(4) Otherwise the system rolls back to an older checkpoint, trying to avoid the trigger for the dormant fault. The older checkpoint can be chosen in multiple different ways and may be application dependent. We discuss this further in Section 7.

Hence with this protocol, many software bugs can be circumvented without user intervention. The overhead associated with the checkpoint and recovery mechanism is compensated by the added reliability it provides.

This protocol relies on an efficient and practical algorithm to maintain consistent cuts. The set of all consistent cuts in a distributed computation forms a distributive lattice. In [6] the authors define the *slice* of a computation with respect to a predicate as the set of join irreducible elements of the lattice of consistent cuts which satisfy the given predicate. Note that join irreducible elements in a lattice are those elements which can not be expressed as the join of two other elements [1].The lattice of all the cuts satisfying the original predicate can be reconstructed from the slice. The slice is, in many cases, exponentially smaller than the entire lattice. Offline and centralized online algorithms for slicing have been proposed in [9, 10]. Note that the problem of maintaining consistent global snapshots is equivalent to the problem of maintaining all the consistent cuts which satisfy a given regular predicate (the predicate being that each process has a checkpoint). Hence in this paper we present an online algorithm for slicing a computation with respect to a regular predicate and we use it to maintain the global snapshots efficiently. We describe this in detail in Section 3.

3 Distributed online slicing

Our algorithm uses a token to determine and maintain the slice. We know that the slice can be constructed by maintaining the smallest consistent cut satisfying the given predicate and containing an event, for all the events in the computation. In our algorithm the local predicate at a process is true if a checkpoint was stored immediately following that event. We could come up with a naive algorithm based on the distributed predicate detection algorithm [5]. The predicate detection algorithm detects the minimum consistent cut for which a given predicate is true. Hence we could run N versions of the algorithm concurrently on each process, which would result in N tokens in a system with N processes. Such an algorithm would involve a large number of messages and possibly be inefficient in the storage space required. We give an improved algorithm in Fig 1.

Our algorithm is run simultaneously with the application. A vector timestamp [8] is maintained at each process. The timestamps at the application events when a checkpoint is stored are added to a local queue. The function `getNextEvent(int localEvent)` returns the first element of the queue following `localEvent`. In our algorithm, we use a single token which contains two NxN arrays.

```
Token:
        cut[1..N][1..N] :Integer ;
        color[1..N][1..N] :Boolean(Red, Green);
        //The slice is the set of all join irreducible elements
        slice : List of consistent cuts which form the slice;
Initially:
    Token is at process P_0

If token is at process P_myId ::

Alogrithm:
  sendToken :Boolean = false;
  while(¬ sendToken){
        //Finds i such that color[i][myId] = RED
        currentProcess = findRed(color);
        //Get next event on process myId
        current[1..N] = getNextLocalEvent(cut[currentProcess][myId]);
        cut[currentProcess][myId] = current[myId];
        color[currentProcess][myId] = GREEN;
        for( i = 1 to N){
              if(cut[currentProcess][i] < current [i]){
                    cut[currentProcess][i] = current [i];
                    color[currentProcess][i] = RED;
        }
        if( ∀ j: color[currentProcess][j] == GREEN ) slice.add(cut[currentProcess]);
        currentProcess = findRed(color);
        // If no process is waiting for events on myId then send the token
        if(currentProcess == NULL) sendToken =true;
  }
  Find P_next such that color[currentProcess][next] == RED ;
  Send Token to P_next ;
```

Fig. 1. Algorithm

If the token is at process `myId`, the algorithm searches for an process i such that `color[i][myId]` is RED. In such a case we say that the process i '*waits on*' an event from process *myId*. The required event from process *myId* is then processed and the algorithm checks if the inclusion of the new event results in any other processes 'turning' RED. Finally the token is sent to another process when the algorithm determines that no other process is waiting for an event from process *myId*.

Our algorithm generalizes the online conjunctive predicate detection algorithm

[5]. The least cut excluding the initial cut that the slice contains is the output of the predicate detection algorithm.

4 Overhead analysis

4.1 Time overhead

Each event in the computation is processed at most N times . Let m be the maximum number of events at each process where a local checkpoint is stored. If there are N processes in all, the total number of events processed by the algorithm is mN. The processing of each event takes $O(N)$ time. Hence the running time of the algorithm is $O(mN^3)$. Note that we assume that the function `getNextEvent` can be efficiently implemented in time O(N).

4.2 Message overhead

Between any two local checkpoints the token can be sent to another process exactly once. Hence, if we have m checkpoints at each process, the number of extra messages required is $O(mN)$.

5 Observations on the implemented system

We implemented the described protocol in Java. The implementation uses the program framework given in [4]. We used the Java serialization API to store application state in files on the hard disk. This is a worst case scenario. In practical cases many of the checkpoints could be stored in memory resulting in lesser overhead. We have not implemented any garbage collection scheme for both the unused checkpoints. We do remove unnecessary events from the local event queue at each process. The implementation uses transitless checkpoints. Fig. 2 shows the overhead costs in the running time of a system with the checkpoint algorithm.

6 Recovery when the faults are dormant

The algorithm can use various strategies for choosing which old global snapshot to choose for rollback. One strategy would be to choose a snapshot that is

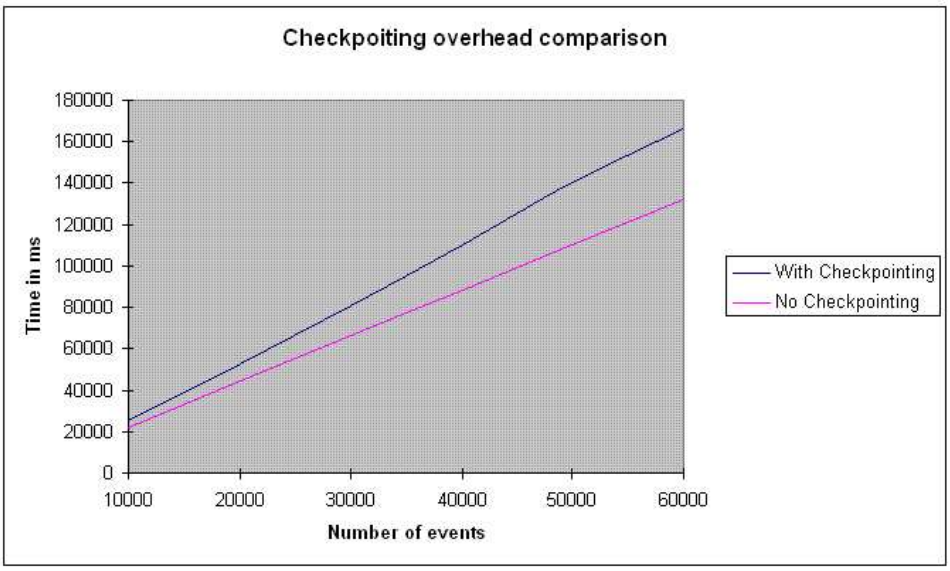

Fig. 2. Running time overhead comparison

incomparable to the last snapshot used. Also exponential rollback may result in circumventing the trigger quickly. Our implementation chooses only the join irreducible elements for rollback.

7 Applications

7.1 Computation intensive applications

For computation intensive distributed programs like protein-folding applications, it is essential to rollback to a valid system state with minimum loss of computation effort. The time taken for the rollback may be relatively unimportant, as the entire computation may take a very long time to complete.

7.2 Simplified algorithm design

Using the fault recovery protocol described in this paper, rare faults can be ignored in the applications. This can also be used to guarantee safety for algorithms which avoid undesirable states with high probability. This may lead to efficient and faster execution in most runs when faults do not occur with a safety guarantee when a fault does occur.

This can also be extended to message passing channels. If a corrupted message results in undesirable behavior in an application, our protocol could be used

instead of using a checksum for each message.

7.3 A simple mutual exclusion algorithm

We could implement a simple mutual exclusion algorithm by allowing each process to enter the critical section whenever it wants to. Periodically each process sends it critical section entry and exit timestamps to a monitor process. If the monitor process detects a safety violation it initiates a rollback. The overhead during an error free run is small. Such a scheme could result in a better performance than the currently used algorithms if the faults are rare.

8 Future work

Different strategies for global snapshot selection could be explored for varying applications. The implementation could be generalized to a fault tolerant framework which could be used to run the practical applications described in the previous section.

A useful extension of this would be to control the computation after recovery to try and avoid the undesirable system state in the new run.

References

[1] Davey, B., and Priestley, H. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[2] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (2002), 375–408.

[3] Garg, V. K. *Elements of Distributed Computing*. Wiley & Sons, New York, NY, 2002.

[4] Garg, V. K. *Concurrent and Distributed Computing in Java*. Wiley & Sons, New York, NY, 2004.

[5] Garg, V. K., and Chase, C. Distributed algorithms for detecting conjunctive predicates. In *IEEE International Conference on Distributed Computing Systems* (June 1995), pp. 423 – 430.

[6] Garg, V. K., and Mittal, N. On slicing a distributed computation. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)* (Washington - Brussels - Tokyo, Apr. 2001), IEEE, pp. 322–329.

[7] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Maeto, Calif, 1993.

[8] Mattern, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms* (1989), pp. 215 – 226.

[9] Mittal, N., and Garg, V. K. Slicing a distributed computation: Techniques and theory. In *5th International Symposium on DIStributed Computing (DISC'01)* (Oct. 2001), pp. 78 – 92.

[10] Mittal, N., Sen, A., Garg, V. K., and Atreya, R. Finding satisfying global states: All for one and one for all. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)* (Apr. 2004).

[11] Wang, Y.-M., Huang, Y., and Kintala, C. Progressive retry for software failure recovery in message passing applications. *IEE trans. on computers 46*, 3 (1997), 1137–1141.