Global Snapshots for Distributed Debugging [*]

Z. Yang and T. A. Marsland

Laboratory for Distributed and Parallel Computing
Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Email: (yang, tony)@cs.ualberta.ca

Abstract

The widespread adoption of distributed computing has accentuated the need for an effective set of support tools. In providing such support, one fundamental problem is that of constructing a global snapshot or global state of a distributed computation. This paper examines global snapshot algorithms from a distributed debugging perspective, and proposes an abstract framework based on global snapshots, which is defined to form a consistent state of the entire system. It is shown that by using a property preserving algorithm this framework can be superimposed on the underlying computation, but not interfere with it.

Keywords: Distributed computing, Distributed debugging, Global states, Snapshots.

[*]Presented at ICCI'92, Toronto, May 28-30, 1992, and to appear in *IEEE Proceedings of Fourth International Conference on Computing and Information (ICCI'92).*

Global Snapshots for Distributed Debugging [*]

Z. Yang and T. A. Marsland

Laboratory for Distributed and Parallel Computing
Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

Email: (yang, tony)@cs.ualberta.ca

1 Introduction

Interest in distributed computing has grown dramatically in recent years, because it has opened a cost-effective way to construct large systems from a collection of computers connected via networks. Such distributed systems exhibit great potential for increased performance, system extensibility, and increased availability[LPS81]. However, to bring this potential to full play, there is a growing need for an effective way to support distributed programming.

A distributed program may be viewed as a collection of processes residing, executing and communicating, but at geographically dispersed nodes which consist of one or more processors, one or more levels of memory, and several I/O devices. Because of the parallelism inherent in a distributed program, nondeterminism of the execution behavior and non-predictable communication delays between processes, programming distributed systems is much harder than for its counterpart — centralized systems. To meet this challenge, much research has been done from several aspects, for example, languages and semantics as illustrated in CSP at Oxford[Hoa85], MIT's Argus project[LS83] and Hermes in IBM[Str90]; algorithms and correctness proofs as advocated by Dijkstra[DS80]; and implementations of several distributed systems such as Conic in Imperial College[KMS87] and also Argus, as well as many others. This multitude shows the importance of obtaining effective solutions to problems which arise in distributed programming. Research has shown that many problems in distributed systems can be cast in terms of the problem of detecting global states. Indeed, constructing a global state or global snapshot of a distributed computation is viewed as a fundamental problem. Many researchers have proposed various algorithms for taking snapshots. Chandy and Lamport[CL85], in their landmark paper, proposed an elegant solution, called distributed snapshots, for detecting stable properties of distributed systems. This solution is general enough to be adapted to specific implementation requirements and is suitable in a broad application domain. These uses include the detection of stable system properties such as deadlock and termination.

[*]Presented at ICCI'92, Toronto, May 28-30, 1992, and to appear in *IEEE Proceedings of Fourth International Conference on Computing and Information (ICCI'92).*

Informally, a snapshot of a distributed computation is a global state which could have been seen by some external observer with any reference point, thus can be viewed as a point in the history of the computation. We can imagine taking a sequence of such snapshots during a distributed computation, initiated on command by a programmer or by the detection of a snapshot condition. If such a trigger is a breakpoint in a computation, the snapshot could be a valuable debugging tool — usually the system stops in a breakpoint, i.e. a snapshot state, until the debugging process permits resumption of execution.

A contribution of this paper is a framework established for distributed debugging. However, before we can discuss our framework in detail, we must present the system model on which snapshot algorithms are based. We then describe the snapshot algorithms which would become a component in our framework. The framework is presented in terms of breakpoints, local snapshots, global snapshots, and the halting and restarting of the underlying computation.

2 The system model

A distributed system is modeled by D = {P, C}, where P is a finite set of processes composing an underlying computation, C is a finite set of communication channels via which processes communicate with each other. The processes do not share memory but communicate exclusively by sending and receiving messages via channels. These channels are assumed to be reliable and synchronous. No assumptions are made about relative speed of the processes.

This system model forms a high level of abstraction of a distributed system. It abstracts away the physical organization of the system and the particular details of the underlying communication network, it even abstracts the details of the distributed programming environment.

Our model differs from one of Chandy and Lamport [CL85] in that the interprocess communications are assumed synchronous rather than asynchronous(buffered). However, the following definitions are derived from their work.

Definition 1. A process is defined as a 3-tuple: p = (S, init, E), where S is a set of process states, $init \in S$ is the initial state, and E is a set of events. The process state is represented by the program counter and all variable values in the memory.

Definition 2. An event of a process p is defined as a 5-tuple (p, s, s', m, c). We say an event occurs if a process transitions from state s to s' and sends (or receives) message m along the outgoing (incoming) channel $c \in C$, which is incident upon p; m and c are null symbols if no message is involved in the event.

For a process, three types of events are possible: intraprocess events, sending a message and receiving a message between processes.

Definition 3. A global state is a set of all process' states within a system. A global snapshot is a recorded global state. A global state S is consistent if there is a predicate function y defined on S, and y(S) is true for a global state S of distributed system D, then y is true at all later points in that computation. Also,

- If event e can occur in global state S, then the function *next*(S, e) returns the global state immediately after the occurrence of e in global state S.

- A global state S' is reachable from global state S (denoted as $S \rightarrow S'$) if and only if there is a computation $\{S_i : 0 \leq i \leq n\}$ such that $\exists j, k : 0 \leq j \leq k \leq n, S = S_j \wedge S' = S_k$.

In the remainder of this paper, global state and global snapshot are used interexchangeably, and refers to a consistent state.

Definition 4. A (distributed) computation is defined as a sequence of events:

$$comp = \{e_i : 0 \le i \le n\}$$

where e_i can occur in global state S. A computation is sometimes denoted by $\{S_i : 0 \le i \le n\}$ for brevity.

3 Global snapshot algorithms

Our abstract framework of distributed debugging is based on global snapshots of the system. A global snapshot is taken at some point within a distributed computation that has a stable property. It is a two phase procedure: a recording phase in which all processes are required to take their respective local snapshots; and a dissemination phase in which a global state is formed from local snapshots. Several global snapshot algorithms have appeared in the literature with different assumptions about the system. Among these, Chandy and Lamport algorithm[CL85] is a landmark one that assumes asynchronous and FIFO channels. It uses a process coloring scheme to enforce the consistency of a global snapshot. All events are defined to be WHITE if they precede the snapshot and RED if they occur afterwards. However, this algorithm is concerned mainly with taking local snapshots, and is highly inefficient. Spezialetti and Kearns [SK86] made a significant improvement to the Chandy and Lamport algorithm. They combined a two phase approach into an integrated algorithm and cleverly use a form of the Chandy and Lamport method for taking local snapshots, and so assemble the global snapshot in an efficient way. Vankatesan proposed an incremental snapshot algorithm [Ven89] to reduce message complexity. It uses the fact that a recent snapshot of the system is already available, and that the change in the system between successive snapshots is likely to be small (namely, a message may not have been sent on some channel since the previous global snapshot). The Li, Radhakrishnan, and Venkatesh Algorithm[LRV87] gets a global snapshot in a non-FIFO channel. Accordingly, it uses *Marker_no* to tag each message (including marker messages) sent along the channel. The Lai and Yang algorithm [LY87] also works with non-FIFO channels, and requires no control message at all in taking a snapshot. Instead, it uses an extra bit (illustrated as a message color: white and red) in all messages that are sent after a process records its state. Morgan provides an elegant treatment of the snapshot algorithm based on factorization [Mor85], i.e., the snapshot algorithm can be factored into two separate parts: a logical clock algorithm and a remaining–time–based algorithm. A detailed survey of all these methods is given in our report [YM92].

The snapshot algorithm in our distributed debugging framework is based on the Spezialetti and Kearns algorithm, which is summarized here to provide a basis of understanding for our work. In contrast to monochrome coloring of the Chandy and Lamport algorithm, Spezialetti and Kearns use a multi–color scheme for local snapshoting. Each process possesses two kinds of coloring variables: *id_color* and *local_color*. An *id_color* is a unique identifying color (not WHITE) which is its network name and does not change over the lifetime of the system; and a local color, initially WHITE. A process is said to be of the color of *local_color*. An initiator changes its color by using its identifying color and sets *local_color* to its *id_color*, and then send out a wave of warning messages which are colored its own *id_color*.

When a white process receives a colored warning, it sets *local_color* to the color of the received warning and follows the Chandy and Lamport algorithm, thus incorporating the process into the snapshot. As the warning wave travels through the system, a region of the initiator's color is established, and all these processes have their *local_color* set to the *local_color* of the initiator. The Spezialetti and Kearns Algorithm thus allows for several initiators of a global snapshot, in this case regions of various colors form. Processes at the border of the regions will have different colors depending on the color of the first warning received on any of the incoming channels. A snapshot is complete when all processes of the system are non-WHITE. It is assumed that there is a spanning tree rooted at the initiator in each region that was created by the initiator when it sent out a warning in the recording phase. Along the tree, each process sends its local snapshot to the initiator. The differently colored warning by different initiators lets the processes at the boundary of the regions identify the initiator in the neighbor region. This identification is also passed to the initiator of the region.

Once the initiator of a region has received local snapshots from all the processes in its region, it also knows the identities of all initiators in all adjacent regions. The initiator in each region disseminates the local snapshots received to the initiators in the adjacent regions. This goes the rounds until each initiator has received local snapshots from all non-adjacent regions as well.

The Spezialetti and Kearns Algorithm results in a more efficient global snapshot via phase-merging, in which each earlier phase provides useful information to the later phase, so it is named "efficient distributed snapshots".

4 An Abstract Framework of Distributed Debugging

As we pointed out earlier, a global snapshot should be taken in such a way that in the global state S the underlying computation possesses a stable property. An example of a stable property is "computation has terminated". This example indicates that we may partition the overall computation into a sequence of computational phases: $comp_1 \longrightarrow comp_2 \longrightarrow \cdots \longrightarrow comp_i \longrightarrow comp_k,$. So that "ith phase has terminated" is a stable property — called a breakpoint. Thus, a global snapshot can be used for distributed debugging. More formally, we define a distributed computation consisting of n processes $P_1, P_2, \cdots, P_i, \cdots, P_n$, each of which can reach a state S_i after a finite time, such that a predicate function $y(S_i)$ holds. Also the computation reaches a global state S and possesses the following properties, similar to those outlined by Chandy and Lamport [CL85]:

1. $\bigwedge_{i=1}^{n} y(S_i) \longrightarrow y(S)$
2. As soon as *y(S)* is true, the stable property holds and remains true within finite delay, so that the computation can be halted for debugging.
3. The next computation phase can be initiated from the state S such that
   $(y(S_i) \longrightarrow BPT_i) \wedge (BPT_i \longrightarrow y(S_{i+1}))$
   where BPT is breakpoint having a boolean value.

In practice, y is an externally defined function, and is usually defined by the programmer. During the execution of the computation, the value *y(S)* may be determined by a process in the system, by applying y to global state S. $BPT = true$ implies that the stable property holds and the breakpoint is reached.

From the above, we derive an abstract framework of distributed debugging as follows:

```
$Debugger$ :: [ $take\ a\ local\ snapshot$;
                $take\ a\ global\ snapshot\ and\ form\ a\ global\ state\ S$;
                $BPT := y(S)$;
                $if\ BPT = true\ then\{$
                                $halt\_to\_debug$;
                                $restart(S)\}$
              ]
```

Now we further elaborate this framework. The notation is a liberal extension of the Communication Sequential Processes (CSP), as defined by Tony Hoare[Hoa85]. Suppose that a distributed computation is a CSP program:

$$P = [\ P_1 \| \cdots \| P_i \| \cdots \| P_n\ ]$$

where for every $1 \leq i \leq n$,

$$P_i :: INIT_i; \quad STATEMENTS_i$$

We assume that each $STATEMENTS_i$ is defined as:

$$STATEMENTS_i \overset{\text{def}}{=} *[\ \Box_{t \in T}\ guard_t \longrightarrow (\ sequence\ \ of\ \ statements\ )_t\ ]$$

that is, the guarded *sequence of statements* denote the underlying computation, where $T = t_1, \cdots, t_m$ is an application dependent index set; and if for some alternatives there is no guard, then *guard = true* is assumed. For the purposes of debugging, we need some kind of arrangement in the underlying computation such that the global snapshot can be taken, and debugging can be carried out. The solution should meet some requirements, for example, it is superimposed on the underlying computation but is independent of the specific problem that is being solved; it should be a communication scheme to fulfill its duty but does not require additional channels; it should also be independent of the number of processes n and should not change specific neighborhood relationships among the P_i, determined solely by the underlying computation.

A natural solution to a CSP program above is to add another alternative, guarded by the debugging requirement, and we call this alternative *the debugger*:

```
$P_i$ :: $INIT_i\ ; \cdots$
     *[ $\Box_{t \in T}\ guard_t \longrightarrow (\ sequence\ \ of\ \ statements\ )_t$;
        $\Box\ debugger$;
     ]
```

Note that the CSP alternative debugger *is* the Debugger we defined in our abstract framework above.

We assume that we have designated an arbitrary process as an initiator in the underlying computation. It starts a global snapshot and collects the stable state information taken by all processes. There is a predetermined spanning tree rooted at the initiating process. In the algorithm described below, which is a combination of the Chandy and Lamport and the Spezialetti and Kearns Algorithms, we further denote $c_{i,p}$ as a channel leading to the parent of a process P_i, and $c_{i,j \in J}$ as a channel of a process P_i to one of its k children ($j = 1, \cdots, k$).

Thus, for the underlying computation,

$$P = [\ P_1 \| \cdots \| P_i \| \cdots \| P_n\ ],$$

we have three cases: P_i is a root (designated as an initiator), or is an intermediate node, or is a leaf. Using Hoare's notation [Hoa85], these cases are described as follows:

Case 1. P_i is the root.

```
$P_i$  ::  $INIT_i\ ; \cdots$
    $*[$  $\Box_{t \in T}\ guard_t; \neg red \longrightarrow (\ sequence\ \ of\ \ statements\ )_t;$
          $\Box\ \ B_i; \neg red \longrightarrow red := true;\ \ recording;$
          $\Box_{j \in J}\ \ red;\ \neg send[j];\ c_{i,j}\ !\ marker() \longrightarrow send[j] := true;$
          $\Box_{j \in J}\ red;\ \bigwedge_{j \in J} send[j];\ \ c_{i,j}?\ (info, done) \longrightarrow$
                          $Done := \bigwedge_{j \in J} done[j];\ \ form\_global\_state;$
          $\Box_{j \in J}\ Done;\ c_{i,j}!\ halt() \longrightarrow halting;$
    $]$
```

where $info \overset{\text{def}}{=} state(i) \cup done$, and $done$ is a boolean bit to inform a process' parent that all local snapshots in the subtree are complete and state information is being sent.

Whenever the root process is in a stable state, it may initiate a global snapshot by sending out a marker message along outgoing channels to its children, while recording its own local state (process state plus channel state), then it waits for all of them to participate in the global snapshot and collects the information from them. When all children return a "done" message, the global snapshot is complete, the root process may now halt the program by instructing the children to halt.

Case 2. P_i is an intermediate node. P_i plays a dual role: as a root (of a subtree) and as a child. So it must propagate the warning message to all its children, it also must collect and send the state information of the children to its parent.

```
$P_i$  ::  $INIT_i\ ; \cdots$
    $*[$  $\Box_{t \in T}\ \neg red;\ \ guard_t \longrightarrow (\ sequence\ \ of\ \ statements\ )_t;$
          $\Box_{j \in J}\ \neg received[j];\ c_{i,p}\ ?\ marker() \longrightarrow$
```

```
                    $received_p := true; [red \longrightarrow skip$
                                          $\Box \neg red \longrightarrow red := true; recording; ]$
        $\Box_{j \in J}\ red;\ \neg send[j];\ c_{c,j}\ !\ marker() \longrightarrow send[j] := true;$
        $\Box_{j \in J}\ red;\ \bigwedge send[j];\ c_{c,j}?\ (info, done) \longrightarrow$
                              $Done := \bigwedge_{j \in J} done[j];\ collect\_state\_of\_subtree;$
        $\Box$     $red;\ Done;\ c_{i,p}!\ state\_of\_subtree \longrightarrow skip;$
        $\Box$     $red;\ Done;\ c_{i,p}\ ?\ halt() \longrightarrow halting := true;$
        $\Box_{j \in J}\ halting;\ c_{c,j}\ !\ halt() \longrightarrow halted$
    ]
```

Case 3. P_i is a leaf.

```
$P_i$ :: $INIT_i; \cdots$
   $*[$ $\Box_{t \in T}\ \neg red;\ guard_t \longrightarrow (\ sequence\ of\ statements\ )_t;$
      $\Box$   $\neg received;\ c_{i,p}\ ?\ marker() \longrightarrow red := true;\ received := true;\ recording;$
      $\Box$   $red;\ c_{i,p}\ !\ state\_info \longrightarrow done := true$
      $\Box$   $done;\ c_{i,p}\ ?\ halt() \longrightarrow halted$
   ]
```

When a leaf process reaches a local stable state and is instructed to take a snapshot, it records and sends the local state to its parent, finally the leaf process halts as instructed.

In all three cases, array send(i) and receive(i) are used to ensure that marker is sent (received) only once. Thus the overall operation of P is as follows. At a certain point in the computation when its local state is stable, the root chooses to initiate a global snapshot by sending a marker message, to traverse the spanning tree, and to wait for a boolean result done, which should be true only if $\forall j \in J \;\; y(S_i) = true$ hold. Whoever receives this marker will spread it down the tree and participate in a global snapshot so that $y(S_i) = true$. Eventually, each process delivers its state information and when done signals its parent. The whole program is now ready to halt for debugging.

The global snapshot algorithm ensures that the snapshot state S could have occurred in the following two senses:

- it is possible for the program to reach S from initial state S_0;
- it is possible to reach a later state S_1 from S.

Hence, after the program halts at a breakpoint(program's state is in S), the restart algorithm should ensure that the program will eventually be in a state reachable from S (hence reachable from S_0).

The idea behind the restart algorithm is very simple. At the point when the program restarts, the algorithm reestablishes the state of all the channels recorded during the global snapshot, and puts the program into the same state as S. We omit the CSP representation of restart algorithm which can be readily found in the literature [Mor85].

Before we complete our discussion on a global snapshot–based distributed debugging framework, several points should be noticed:

1. The debugger always takes the system from one consistent state to another. This requires that breakpoints must be set in such a way that they are all consistent with each other, and with the interactions between them.

2. This framework is superimposed on the underlying computation, since it halts the computation at global state S reachable from S_{i-1}, and then restarts it eventually from a state S_i reachable from S, thus $S_{i-1} \longrightarrow S \longrightarrow S_i$. The framework, therefore, effectively preserves the computation upto and including state S. In this sense, the framework does not alter the underlying computation.

3. This framework carries out the debugging function without introducing additional communication channels.

5 Conclusion

Distributed computing systems have come into widespread use only recently. Experience with programming and debugging them is limited. This paper examines global snapshot algorithms from a distributed debugging perspective, and proposes an abstract framework based on a global snapshot which is defined to be a consistent state of the entire system. When a program, by means of some finite sequence of interprocess communication, reaches global stability, where each process is locally stable, the global stable property (i.e., breakpoint y) holds. The system could stay in this state for debugging, and from this state restart its execution. By using a property preserving algorithm it is shown that this framework is superimposed on the underlying computation, but does not interfere with it.

The presentation of our framework is inspired by CSP which provides a concise notation and has sufficient expressive power to serve its purpose. However, it is our belief that linguistic support (a language provision) is needed to allow for debugging of distributed programs that are supported by high level programming languages. We are working on this issue, and hoping to report the research result in the future.

Acknowledgment

The research of first author is sponsored in part by an NSERC International Scientific Exchange Award, and also through NSERC grant OPG7902. Access to these sources is gratefully acknowledged.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transaction on Computer Systems*, 3(1), February 1985.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computation. *Information Processing Letter*, 11(8), August 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[KMS87] J. Kramer, J. Magee, and M. Sloman. The CONIC Toolkit for Builiding Distributed Systems. In *IEE Proceedings D*, pages 73–82, March 1987. Vol. 134, No. 2.

[LPS81] B. W. Lampson, M. Paul, and H. J. Siegert. *Distributed Systems – Architecture and Implementation: An Advanced Course.* Springer-Verlag, 1981. LNCS 105.

[LRV87] H. F. Li, T. Radhakrishnan, and K. Venkatesh. Global State Detection in Non-FIFO Networks. In *Proceedings of 7th Conference on Distributed Computing Systems*, pages 364–370, 1987.

[LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Program. *ACM Transactions on Programming Languages and Systems*, 5(3), 1983.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(5), May 1987.

[Mor85] Carroll Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20(5), May 1985.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *Proceedings of Sixth International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[Str90] Robert E. Strom. Hermes: An Integrated Language and System for Distributed Programming. In *1990 Workshop on Experimental Distributed Systems*, Huntsville, AL, October 1990.

[Ven89] S. Venkatesan. Message-optimal incremental snapshots. In *Proceedings of nineth International Conference on Distributed Computing Systems*, pages 53–60. IEEE Computer Society PresS, 1989.

[YM92] Z. Yang and T. A. Marsland. Global Snapshots for Distributed Debugging: An Overview. Technical Report TR 92-xx, Computing Science Department, University of Alberta, (in preparation), 1992.