Global Snapshots for Distributed Debugging: An Overview

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

Technical Report TR–92–03

Abstract

The widespread adoption of distributed computing has accentuated the need for an effective set of support tools to facilitate debugging and monitoring. 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.

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. Programming distributed systems is much

harder than for its counterpart — centralized systems, because of the parallelism inherent in a distributed program, nondeterminism of the execution behavior and non-predictable communication delays between processes. 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 works for a broad application domain. These uses include the detection of stable system properties such as deadlock and termination.

Informally, a snapshot of a distributed computation is a global state which could have been seen by some external observer with any reference point, and can thus 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 to establish a framework 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 channels via which processes communicate. 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 the 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 the one of Chandy and Lamport [CL85] in that the interprocess communications are assumed synchronous instead of 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 and all channel states within a system, such that in the global state:*

1. *Every message that a process receives would have been sent by the sending process of that message;*
2. *The channel state is a set of all messages sent by a process along that channel, but not yet received by the receiving process.*

Note. Because our model relies on synchronous communication, both the receiver and sender must be ready before the communication can take place. Thus no message will be held up in the channel, so only the process states need be counted.

The global state thus defined is always consistent in a sense that if a predicate function defined on S, *y(S)*, becomes true, then y remains 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 $\exists j, k : 0 \leq j \leq k \leq n, S = S_j \wedge S' = S_k$.

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

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

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

3 Global Snapshot Algorithms: A Survey

Our abstract framework of distributed debugging is based on global snapshots of the system. In view of this, we provide here a survey on the global snapshot algorithms. In the next section, we explain how global snapshots fit into our framework.

A global snapshot is taken at some point within a distributed computation that exhibits an interesting property. It is a two phase procedure: a recording phase where all processes are required to take respective local snapshots; and a dissemination phase where a global state is formed from local snapshots. To meet the global state consistency requirement, we need some way to delimit the "before" or "after" snapshot. There are two mechanisms that could be used for the demarcation: by using a control message (marker) or by setting a clock to a predefined time. Thus, the algorithms we survey fall into two categories: control–message–based and time–based global snapshot algorithms.

3.1 Control message Snapshot Algorithms

3.1.1 The Chandy and Lamport Algorithm

Chandy and Lamport's algorithm[CL85] is a landmark one for taking distributed snapshots. 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. Initially all processes are WHITE. All processes having the same color, namely RED, will yield a consistency snapshot. In the algorithm, a process wishing to initiate a global snapshot executes a procedure Turn_Red, thus sending out warning messages to begin the snapshot. The procedure Turn_Red is outlined as follows:

1. set color of itself to RED;
2. record current state of the process;
3. for each incoming channel, c_{pq}, begin recording messages on c_{pq};
4. for each outgoing channel, c_{qp}, send out a warning message before sending any more message on it;

A process, upon receipt of a warning message on an incoming channel, executes Get_Warning procedure as outlined as follows:

1. if color = WHITE, then execute Turn_Red;
2. stop recording incoming message on channel.

The Chandy and Lamport algorithm is concerned mainly with taking local snapshots. To disseminate back to the initiator the state information recorded individually, each process in the system is required to send its local snapshot to each of its neighbors (through all of its outgoing channels). As a process receives another process' local snapshots, it must, in turn, pass the snapshots to each of its own neighbors. All processes will eventually receive copies of local snapshots of all other processes. In this manner, a global snapshot can be formed by all the processes in the system.

This algorithm needs $O(N^2)$ messages for taking a local snapshot, and requires $O(N^3)$ messages for defusing the state images of the processes. This message complexity leads to several variants to improve the efficiency.

3.1.2 The Spezialetti and Kearns Algorithm

Spezialetti and Kearns noticed that the independence of the two phases in the Chandy and Lamport algorithm results in more message complexity — in particular, the dissemination phase is done exhaustively in the sense that all processes, even if they do not require the snapshot, will ultimately receive the total snapshots whereby they could form a global snapshot. They combined two phase approach into an integrated algorithm — called Spezialetti and Kearns Algorithm[SK86] in which they cleverly use a form of the Chandy and Lamport algorithm for taking local snapshots to assemble the global snapshot in an efficient way which we will summarize below.

In contrast to monochrome coloring of Chandy and Lamport algorithm, they utilize a multi-color scheme for the 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 setting *local_color* to its *id_color*, and then sending out a wave of warning messages which are also colored *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 on 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 identifiers of all initiators in all adjacent regions. The initiator in each region disseminates the local snapshot of processes received to the initiator 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 earlier phase provides useful information to the later phase. It needs $O(N^2)$ messages in the recording phase, and when there are m concurrent initiators needs $O(mN^2)$ exchanges of local snapshot in the worst case, so they are called "efficient distributed snapshots".

3.1.3 The Venkatesan Algorithm

Distributed applications often require that many successive snapshots be taken to get vital information about the computation. It is obvious that the Chandy and Lamport algorithm is not suitable for this purpose because the communication overhead is excessive in terms of control messages. Addressing this problem, Venkatesan proposed a notion of incremental snapshots and corresponding protocol for taking such incremental snapshots. The Venkatesan algorithm[Ven89] is intended to reduce the 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 message may not have been sent on some channel since the previous global snapshot). As in the Spezialetti and Kearns Algorithm for reducing the message complexity of the algorithm, a spanning tree is assumed to exist — this is a one time pre-processing step, and is not considered to be a component of the algorithm.

The Venkatesan algorithm assumes there is an initiator process to which all requests for a global snapshot are forwarded. The very first snapshot taken uses the Chandy and Lamport algorithm and is complete. Each subsequent snapshot is taken only if the previous one has completed (in-

cremental), and has a version number one more than that of previous one. In this algorithm, four types of control messages are used: initiation message, snapshot complete message, marker, and acknowledge (*ack*) message. An incremental global snapshot is taken by sending, along with a spanning tree rooted at the initiator, control messages, each of which carries the version number of the global snapshot.

When receiving an initiation message or a marker with a higher version number, each process records an incremental local snapshot by performing the following protocol steps:

1. sets states of each incoming channel to empty;
2. records its local state;
3. sends an initiation message to its each child;
4. records all received messages on incoming channels until a marker is received on that channel, at which point it sends back an *ack* to the marker sender.
5. sends markers on those outgoing channels that it sent at least one message since the most recent local state recording;
6. waits for an *ack* of the markers it sent in above step.
7. waits for snapshot complete message from each child;
8. when it has received all expected *ack* messages and snapshot complete message, it sends its parent a snapshot complete message.

When all the local snapshots as shown above have completed, the global snapshot is formed by the initiator.

The message complexity of this *incremental snapshot algorithm* is *O(N+U)* where U is the set of edges on which a message has been sent since the previous global snapshot.

3.1.4 The Li, Radhakrishnan and Venkatesh Algorithm

The above snapshots algorithms work with FIFO channels. The Li, Radhakrishnan and Venkatesh Algorithm[LRV87] gets a global snapshot in a non-FIFO channel. Accordingly, it needs to tag each message and marker sent along the channel. This tag is *Marker_no* which is the ordinal number of the latest global snapshot initiated as known to the sender process, and each process has a local *Marker_no*.

For simplicity, we consider the case where there is only a single initiator process. Each process in the system uses an observer process to keep track of all the messages sent and received on each channel, after it recorded its local state for the latest global snapshot initiated by the initiator. When initiating a global snapshot, the initiator process:

1. increases its Marker_no by one;
2. records its state and value of its observer;
3. sends a marker on each outgoing channel;

When receiving either a message or a marker, and if the tag on it is greater than value of its local marker counter, a process does the following:

1. it sets its Marker_no to that of message/marker tag received;
2. it records its state;
3. sends a marker tagged with Marker_no on all outgoing channel;
4. tags all messages with incremented value of Marker_no.

In this algorithm, we have the following state information:

$$\begin{aligned} \textit{local snapshot} &= \{\textit{recorded state}\} \cup \\ &\quad \{\textit{messages sent on each emanating outgoing channels}\} \cup \\ &\quad \{\textit{message received on each incident incoming channel}\} \end{aligned}$$

$$\begin{aligned} \textit{channel state} &= \{(\textit{message in transit at the previous global snapshot}) - \\ &\quad (\textit{message received by the sender's observer})\} \end{aligned}$$

The recorded local snapshots and messages recorded by the observer process are sent back to the initiator along the edges of a spanning tree rooted at the initiator to form a global snapshot.

In this algorithm, for each initiation of local snapshots, the number of marker messages is equal $O(N)$, whereas the message complexity in state assemble phase is $O(mN^2)$.

3.1.5 The Lai and Yang Algorithm

As mentioned earlier, to ensure the global snapshot scheme works correctly, processes in the system should be somewhat coordinated in taking local snapshots so that the formed global snapshot is "meaningful"[CL85]. That is where the control message marker plays the role of the coordination. Lai and Yang, however, argue that an "uncoordinated" global snapshot, in which processes take local snapshots without any coordination among them, can be obtained and is useful in some applications. Their algorithm requires no control message at all in taking a snapshot and does not require channels to be FIFO. 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. Lai and Yang's Algorithm[LY87] works according to the following three rules:

1. every process is initially white and turns red while taking a local snapshot;
2. every message sent by white(red) process is colored white(red);
3. every white process starts a snapshot at its convenience (no coordination) — but no later than a red message receipt. Thus, the arrival of a red message at a white process will invoke the process to take a snapshot before receiving the message.

The algorithm assumes that every process can instantaneously turn red (in contrast to the Chandy and Lamport algorithm in which only one process is required to have this "instantaneous" nature), and keep a log since the last snapshot. The behavior of a process upon receipt of a message of a new color is the same as that of a process in the Li, Radhakrishnan and Venkatesh Algorithm when it receives a message/marker of a higher Marker_no. Because of this similarity, we consider it to fall into the class of control–message schemes.

In this algorithm, if only one process can instantaneously initiate the algorithm, then $O(N)$ control messages are required.

3.2 A Time-based Algorithm: The Morgan Algorithm

It has been noted that if global time is available, the design and description of distributed algorithms can be significantly simplified. In the context of global snapshots, 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. Many researchers have proposed different schemes to make global time logically or physically available. For example, use of Lamport's logical clock [Lam78] and its variants, or use of a synchronized clock to guarantee with very high probability that the clocks have a skew of less than a δt seconds by using network time protocol[Mil88] which synchronizes clocks of nodes on a geographically distributed network. Assuming the availability of the global time, we have Morgan's time-based global snapshot algorithm[Mor85] in which at some predefined time t, all processes record their state and do the following:

1. mark all the outgoing messages with the time at which they were sent.
2. record the state before continuing the underlying computation;
3. record all messages received on or after t which were sent strictly before t. These are exactly the messages in the channels at time t;
4. On all input channels that carry a message sent on or after t, transmit the recorded state and message sequences to the process that has responsibility to form a global snapshot.

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

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

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

Now we further elaborate this framework. The notation is a liberal extension of the Communication Sequential Processes (CSP), as defined by 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:

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

that is, the guarded *sequence of statements* denotes the underlying computation, where $T = t_1, \cdots, t_m$ is an application dependent index set; the input/output commands can only appear in the guards 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*:

$$
\begin{array}{lll}
P_i & :: & INIT_i\,; \\
 & *[& \Box_{t\in T}\, guard_t \longrightarrow (sequence\ \ of\ \ statements)_t; \\
 & & \Box\ \ debugger; \\
 &] &
\end{array}
$$

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.

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

where $info \stackrel{\text{def}}{=} state(i) \cup done$, 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.

$$
\begin{array}{lll}
P_i & :: & INIT_i\,; \\
 & *[& \Box_{t\in T}\ \neg red;\ guard_t \longrightarrow (sequence\ \ of\ \ statements)_t; \\
 & & \Box_{j\in J}\ \neg received[j];\ c_{i,p}\ ?\ marker() \longrightarrow \\
 & & \qquad\qquad received_p := true; [red \longrightarrow skip \\
 & & \qquad\qquad\qquad\qquad \Box \neg red \longrightarrow red := true;\ \ recording;] \\
 & & \Box_{j\in J}\ red;\ \neg send[j];\ c_{c,j}\ !\ marker() \longrightarrow send[j] := true; \\
 & & \Box_{j\in J}\ red;\ \bigwedge send[j];\ \ c_{c,j}?\ (info, done) \longrightarrow \\
 & & \qquad\qquad\qquad\qquad Done := \bigwedge_{j\in J} done[j];\ \ collect\_state\_of\_subtree; \\
 & & \Box \qquad red;\ \ Done;\ \ c_{i,p}!\ state\_of\_subtree \longrightarrow skip; \\
 & & \Box \qquad red;\ Done;\ \ c_{i,p}\ ?\ halt() \longrightarrow halting := true; \\
 & & \Box_{j\in J}\ halting;\ \ c_{c,j}\ !\ halt() \longrightarrow halted \\
 &] &
\end{array}
$$

Case 3. P_i is a leaf.

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

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, the arrays *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 after a local snapshot has been taken. Whoever receives this marker will spread it down the tree and participate in a global snapshot. Eventually, each process delivers its state information and when done signals its parent. The whole program is now ready to halt for debugging.

theorem 1 *The global snapshot algorithm in the above framework gives a consistent global state.*

Proof:
(Correctness) After a global snapshot is taken, the underlying computation is stable. It is necessary to show that a global snapshot is feasible and meaningful. When initiating a global snapshot based on same predefined condition, a process takes a local snapshot, and sends a marker message traversing the spanning tree. The underlying computation in this process is frozen, no further communication in the underlying communication between processes is carried out. As the marker message traverses the spanning tree, all processes will do the same: recording local state and freezing the computation. because of synchronous communication, no channel state needs to be recorded. In finite time (because the communication delay is finite as we assumed in our model), the whole underlying computation is frozen. The system is in state S, thus the global snapshot is feasible. The rest of the proof, that is, the global state S is meaningful, is similar to the correctness of the Chandy and Lamport algorithm [CL85, page 72-73], and will not repeat here.

(Termination) Each process eventually takes its local snapshot. A process will propagate the marker message when it gets its marker. Hence, as long as there is an edge along the spanning tree to every other node, each process will send a marker down the tree. Because the reliable communication is assumed, markers are not lost, they eventually reach and all processes take the local snapshot. Same arguments hold for collecting the local snapshots and halting the computation. □

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 the 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 hope to report the research result in the future.

Acknowledgment

The research of first author is sponsored in part by an International Scientific Exchange Award and also through Grant OPG7902 held by the second author. Access to these sources through the Natural Science and Engineering Research Council (NSERC) of Canada is gratefully acknowledged.

References

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transaction on Computer Systems*, 3(1), February 1985. Pages 63-75.

[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 International, Series in Computer Science, 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.

[Lam78] Leslie Lamport. Time, Clock, and the Ordering of Events in a Distributed System. *Communication of ACM*, 21(7), July 1978. Pages 558-565.

[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 Programs. *ACM Transactions on Programming Languages and Systems*, 5(3), 1983. Pages 381-404.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(5), May 1987. Pages 153-158.

[Mil88] D. L. Mills. Network Time Protocol Specification and Implementation. DARPA internal report RFC 1059, DARPA, 1988.

[Mor85] Carroll Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20(5), May 1985. Pages 189-194.

[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.