Maximal Global Snapshot with Concurrent Initiators

Ravi Prakash and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210.
e-mail: {prakash, singhal}@cis.ohio-state.edu

Abstract

In a distributed system multiple nodes may initiate snapshot collection concurrently. In this paper we present a global snapshot collection algorithm that combines the information collected by each initiator. This generates a maximal, consistent global snapshot that is more recent than the snapshot collected by any initiator. Global snapshots are used to establish checkpoints for recovery from node failures. A maximal snapshot implies that the amount of computation lost during roll-back, after node failures, is minimized. We also present an efficient information dissemination strategy that nodes can employ to exchange snapshot information with each other.

Keywords: concurrent snapshot, consistent snapshot, recovery.

1 Introduction

A distributed system is composed of a set of n processors (henceforth, referred to as nodes) and a network of bidirectional communication links connecting the nodes. The links are FIFO in nature. Message transmission through these links takes unpredictable but a finite amount of time. During normal operation, no messages are lost or modified in transit. The system does not have any shared memory and no global clock. All the communication and synchronization takes place through messages.

A distributed computation consists of processes that communicate asynchronously with each other. These processes run on different nodes of the distributed system. The processes exchange information with each other through messages. For the application to run successfully, all the nodes, on which the modules of the application are running, should function properly

A node in the distributed system may fail during execution. We assume all the failures to be fail-stop in nature. When a node fails, the contents of its volatile memory are lost. This may leave the distributed computation in an inconsistent state. For example, a node P_i may fail after sending a message to P_j, but before recording the transmission of the message in a log maintained in its stable storage. Thus, the reception of the message will be recorded by P_j, while its transmission will not be recorded at P_i. In order to handle failures, algorithms have been proposed to collect consistent global snapshots of the system [1, 2, 4, 5, 7, 9, 10, 12, 15, 18]. The global snapshot is used to establish consistent checkpoints at the nodes. Recovery from a node failure is performed by rolling back the computation to a consistent checkpoint and then allowing it to proceed from there [5, 12, 14, 16, 17]. The more recent the checkpoint, the less the amount of computation that is lost during failure recovery. Hence, it is important to develop algorithms that collect a maximal global snapshot of the system (i.e., as latest as possible).

Koo and Toueg [5] have shown that if the nodes take their local snapshots in an uncoordinated manner, it may not be possible to construct a consistent global state from such snapshots. The rollback may lead to a *domino effect.* In synchronous snapshot collection algorithms, a node initiates snapshot collection by taking its local snapshot. Then it sends request messages to other nodes to take their snapshots. If the nodes maintain information about causal dependencies, a minimal number of nodes have to take their local snapshots in response to such requests. Such minimal snapshot collection algorithms have been presented in [5, 11].

Koo and Toueg's algorithm [5] involves suspending the underlying computation during snapshot collection. Once a node has taken its local snapshot for a snapshot initiator, it neither sends nor accepts any messages other than control messages corresponding to that snapshot initiation. The nodes resume the underlying computation when the snapshot collection terminates. In [11], Prakash and Singhal present a snapshot collection algorithm that allows the underlying computation to proceed while the snapshot is being collected. In addition, it does not force every node to take its local snapshot.

In a distributed system, multiple nodes may concurrently and independently initiate global snapshot collection. This is because a node may spontaneously initiate snapshot collection and a node's decision to initiate a global snapshot collection cannot be instantaneously conveyed to every other node in the system. Most of the snapshot collection algorithms presented in the past have assumed that at any given time only one snapshot collection is active. Koo and Toueg [5] and Spezialetti and Kearns [13] have proposed methods to handle concurrent initiations of snapshot collection.

Koo and Toueg [5] handle concurrent snapshot collections in the following manner: Once a node takes a local snapshot, it is unwilling to take a snapshot in response to another initiator's request. The node sends a negative response to all subsequent requests, until the snapshot is made permanent or until the snapshot collection is aborted. This approach has a drawback as shown in Figure 1. Let there be two nodes P_1 and P_2 such that both send a

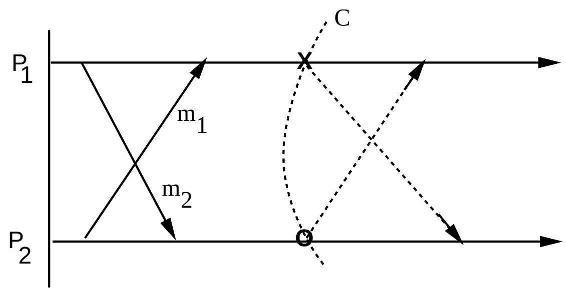

Figure 1: Concurrent snapshot collection for the Koo and Toueg algorithm.

message, indicated by a solid arrow, to each other. Node P_1 takes a tentative local snapshot at the instant marked by "X" and then sends a snapshot request to P_2. P_2 takes a tentative local snapshot at the instant marked by "O" and sends a request to P_1. The request messages are indicated by the broken arrows. Node P_1 refuses to take a snapshot in response to P_2's request and vice-versa. As a result both P_1 and P_2 receive a negative response and do not make their tentative snapshots permanent. Even though their tentative local snapshots put together would have constituted a consistent global snapshot, indicated by C, the effort is wasted.

Spezialetti and Kearns' algorithm [13] is based on the Chandy-Lamport snapshot collection strategy [1] and requires every node in the system to take its local snapshot. Mutually disjoint sets of nodes take their local snapshots in response to requests from different initiators. Then the local snapshots are patched together to construct a global snapshot.

In this paper we propose a new global snapshot collection algorithm to handle multiple concurrent snapshot initiations. The global snapshot collected by the proposed algorithm represents a much more recent state of the system than what is collected by Spezialetti and Kearns' algorithm. The more recent global snapshot, when used to establish checkpoints, will lead to less computation being lost during recovery from node failures. Unlike Koo and Toueg's algorithm, the proposed algorithm does not require the underlying computation to be suspended during snapshot collection. In addition, it does not suffer from the problem illustrated in Figure 1.

The paper is organized as follows: In Section 2, we describe the two basic approaches to handle concurrent snapshot collections. In Section 3, we present the maximal global snapshot collection algorithm. Section 4 describes a new strategy for the concurrent initiators to disseminate the state information, and to construct a global snapshot from this information. Section 5 compares our algorithm with Spezialetti and Kearns' algorithm. Finally, Section 6 presents the conclusion.

2 Concurrent Snapshot Collection Strategies

When a node decides to initiate a snapshot collection, this decision diffuses through the system in the form of requests to other nodes to take their local snapshots. As mentioned in Section 1, it is possible that multiple nodes may initiate snapshot collections concurrently. If each initiation is allowed to proceed independently, a node will receive multiple requests from different initiators to take its local snapshot. If a node has already taken its local snapshot for one initiation, then an issue is whether it should take local snapshots for requests from other concurrent initiations arriving later. There are different approaches to handle concurrent initiations of snapshot collection.

2.1 Classification of Approaches

There are two basic approaches to handle concurrent initiations of snapshot collection.

1. If a node has already taken a local snapshot for one initiation, then the diffusion of snapshot requests for other concurrent initiations is suppressed by that node — local snapshots are not taken in response to the requests arriving later and the node does not propagate those requests to its neighbors. Different nodes may take local snapshots in response to requests from different initiators. Once all the nodes have taken their snapshots, the local snapshots are combined to produce a global consistent snapshot. This approach has been employed by Spezialetti and Kearns [13].

   Let m nodes initiate snapshot collection independently and concurrently, and the snapshots collected by them be denoted as $C_1, \ldots, C_m$. Mattern [9] has shown that if C_1 and C_2 are consistent snapshots of a distributed computation, then $C_1 \cap C_2$ is also a consistent snapshot. The global snapshot obtained by taking the intersection of a set of snapshots is equivalent to the global snapshot obtained by suppressing the diffusion of snapshot requests. This global snapshot is the minimal of the snapshots collected by each individual initiator.

2. A node does not suppress the diffusion of subsequent concurrent snapshot requests after it has taken its local snapshot — it takes local snapshots in response to requests from concurrent initiations arriving later and propagates these requests. After all the nodes have finished taking their local snapshots, the latest local snapshot of each node is combined to produce a global consistent snapshot. This approach corresponds to taking the union of all the concurrent snapshots. Mattern [9] has shown that if C_1 and C_2 are consistent snapshots of a distributed computation, then $C_1 \cup C_2$ is also a consistent snapshot. The snapshot thus obtained is the maximal of the snapshots collected by the individual initiators.

Henceforth, we will refer to the two approaches as the *intersection approach* and the *union approach*, respectively. In the union approach, the diffusion of requests is not suppressed and the latest local snapshots at the nodes are selected. Therefore, the global snapshot obtained is more recent than that obtained from the intersection approach. In this paper we propose an algorithm that employs the union approach to collect the maximal global snapshot

Next we briefly describe Spezialetti and Kearns' algorithm that uses the intersection approach. The performance of the proposed algorithm will be compared with this algorithm.

2.2 Spezialetti and Kearns' Algorithm

In this algorithm, each node has a color associated with it which uniquely identifies the node. But the node can assume other colors, too. The messages bear the color of their respective senders at the time of sending the message.

Initially, all the nodes are white. A snapshot initiator node switches to its associated color, takes its local snapshot, and sends snapshot request messages to its neighbors. When a white node receives a colored snapshot request, it changes its color to that of the request message, takes its local snapshot, and forwards the request to its neighbors. When a non-white node receives a snapshot request of a different color, it infers the presence of more than one snapshot initiator in the system. The identity of the snapshot initiator that caused this request (of a different color) to be sent can be inferred from the color of the request message. The colored node adds the identity of the other initiator to a list called *border_list*. Once a node has received requests along all its incident edges, its coloring phase is complete. Then the node sends its *border_list* to the node from which it received the first snapshot request message. When a node receives *border_lists* from its neighbors, it takes their union and stores the resultant list as its own *border_list*. Ultimately, this list reaches the initiator node that caused this request message to be generated.

At the end of the coloring phase, the system is divided into regions of different colors; each region corresponding to an initiator. The *border_list* that ultimately reaches an initiator informs it about its neighboring initiators. The initiators exchange the partial snapshots of the nodes in their region with each other using a series of rounds. In each round an initiator sends to the initiators of its bordering regions any new state information that it obtained in the previous round. At the end of the information exchange phase, each initiator has a consistent global snapshot of the system.

3 A Maximal Snapshot Collection Algorithm

In this section, we present a new algorithm that employs the union approach to collect a maximal global snapshot. We first present the motivations for using the union approach.

3.1 Motivations

In Spezialetti and Kearns' algorithm, if a node has already taken a local snapshot for one initiator, it suppresses the diffusion of snapshot requests for other initiators. This intersection approach leads to the collection of a global snapshot that is not as recent as the global snapshot collected by the union approach. This is illustrated by Figure 2. Node P_2 initiates snapshot collection by taking its local snapshot at the instant marked by "X". It sends snapshot requests to both P_1 and P_3. Node P_3 concurrently invokes a snapshot collection by taking its local snapshot at the instant marked by "O" and sends requests to the other two nodes. The snapshot requests are indicated by broken arrows. The local snapshots taken by nodes in response to P_2's initiation are indicated by "X". In Spezialetti and Kearns'

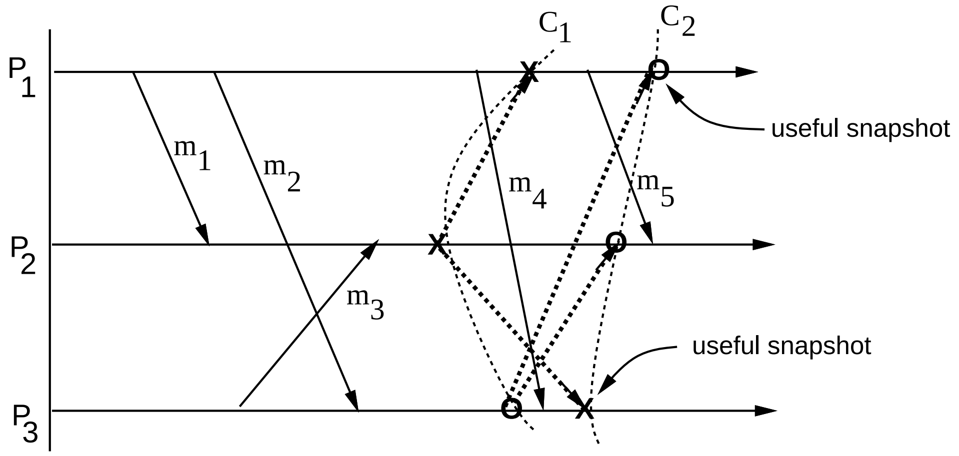

Figure 2: Global consistent snapshot composed of local snapshots taken for different snapshot initiators.

algorithm, P_1 takes its snapshot in response to P_2's request (marked by an "X") and ignores P_3's request. P_3 ignores P_2's request. The global snapshot thus collected is shown by the cut C_1, in which the transmission of message m_5 by P_1 and the reception of m_4 by P_3 are not recorded. Clearly, this snapshot is the minimal of the snapshots taken by P_2 and P_3. In the union approach, the request messages arriving at a node after it has taken its tentative local snapshot are not ignored. So, P_1 takes a local snapshot on receiving P_3's request and P_3 takes a local snapshot on receiving P_2's request. When the latest local snapshots taken by the nodes in the system are combined, the global snapshot obtained is indicated by the cut C_2. Consequently, the transmission of m_5 and the reception of m_4 are recorded. This snapshot reflects a more recent state of the system, as compared to the snapshot collected by Spezialetti and Kearns' algorithm. This snapshot is more recent than the global snapshot collected by individual initiators.

3.2 The Algorithm

The maximal snapshot collection algorithm consists of two phases: (a) independent snapshot collections by the initiators and (b) sharing snapshot information among initiators. In the first phase, each node in the system takes one or more tentative local snapshots (depending on the number of initiators). In the second phase the initiators exchange information with each and make one tentative local snapshot permanent at each node. Lamport's clock [8] is maintained by each node. Huang's termination detection algorithm [3], of associating weights with nodes and messages, is employed to detect the termination of snapshot collection.

When the snapshot information is being shared among the initiators, if the local snapshot taken by initiator P_i is being considered to find the latest local snapshot at a node P_j, then the local snaphot for P_i must be considered for finding the latest local snapshot at every other node, too. Partial sharing of sharing of snapshot information gathered by snapshot initiators may lead to the construction of an inconsistent global snapshot. The algorithm uses the snapshots taken by individual concurrent initiators to construct the maximal global snapshot. A node can have one of two possible colors: black or white. The colors are used to denote the causality relations between nodes. Unlike Speazialetti and Kearns' algorithm, colors are not used to mark regions.

3.2.1 Independent Snapshot Collections

All the nodes maintain a real-valued variable *weight* which is initialized to zero, and a variable *color* which is initially set to white. The nodes also have another variable *init_color* that is used to store the color of the node at the time it initiated a snapshot collection. All the messages sent by a node bear the color of the node at the time of sending the message.

When a node P_j initiates a snapshot collection, it sets its weight to one, stores its color in *init_color*, and then sets its color to black. Then the node takes its own tentative local snapshot and sends snapshot REQUESTs to all its neighbors. The REQUESTs and all the subsequent messages sent by the initiator will be colored black. With each REQUEST, P_j

sends *init_color* and a portion of its weight and decreases its own weight by the same amount. The REQUEST also carries the identity of the snapshot initiator — P_j. When a white node receives a black message, it changes its color to black.

When a node, P_i, receives its first snapshot REQUEST for an initiator, it takes its own tentative local snapshot. P_i associates the value of *init_color*, received in the REQUEST, with the local snapshot. Then it sends REQUEST messages on behalf of the initiator along all the outgoing edges, except the edge along which the REQUEST was received. A fraction of the weight received in the incoming REQUEST message is split equally among the outgoing REQUEST messages. The REQUESTs also carry the *init_color* value received by P_i in the incoming REQUEST message. Then P_i sends a RESPONSE to the snapshot initiator. The weight of this RESPONSE is equal to the difference between the weight received in the incoming REQUEST message for the initiator and the sum of the weights sent in the outgoing REQUEST messages. If there is no outgoing edge along which P_i could propagate the REQUEST, then the RESPONSE has a weight equal to the weight of the incoming REQUEST message. The RESPONSE contains the value of the local clock of P_i at the time it took its tentative snapshot on receiving the initiator's REQUEST.

If node P_i has already taken a tentative local snapshot for an initiator and receives another REQUEST on behalf of the same initiator, it sends a RESPONSE to the initiator with a weight equal to that of the REQUEST.

If a snapshot initiator P_i receive a snapshot REQUEST for another initiator P_j, it sends it own *init_color* in the RESPONSE. Then P_i compares the value of *init_color* received in the REQUEST with its own *init_color*. If they are both white then it implies that P_i and P_j initiated snapshot collection concurrently. So, P_i adds P_j to its list of concurrent initiators.

The initiator adds the weights received in the RESPONSEs. When the weight becomes one, it knows that all the nodes in the system have taken their tentative snapshots in response to its REQUESTs. The initiator also maintains an integer vector V of length n, where n is the number of nodes in the system. When the first RESPONSE is received from P_i, the value of P_i's local clock, carried by the message, is stored in V_i. Thus, the vector V stores the local clock components of all the nodes in the system at the time they took their tentative local snapshots for the initiator. If the RESPONSE received by P_j from P_i carries the *init_color* of P_i, then P_j compares that with its own *init_color* If they are equal then it implies that both

P_i and P_j initiated snapshot collection concurrently. So, P_j adds P_i to its list of concurrent initiators.

Once initiator P_j's weight becomes one, it is ready to share its snapshot information with other concurrent initiators so as to generate a maximal consistent snapshot. The list of concurrent initiators contains all the nodes with mutually concurrent snapshot initiations. The lists at all the concurrent initiators are identical. If all the nodes were initially white, then at the end of the independent snapshot collection phase, they are all black. The next time concurrent snapshot initiations are done, the same algorithm is followed, except the colors white and black are reversed.

3.2.2 Sharing Snapshot Information

When a snapshot initiator P_j's weight becomes one, its independent snapshot collection phase comes to an end. At this point of time P_j knows all the nodes that initiated snapshot collection concurrently with it. P_j sends its V vector to all these concurrent initiators and waits for the corresponding vectors to arrive from those nodes.

Once P_j has received all the vectors, it does the following: for every i, where $1 \leq i \leq n$, if its own vector component (V_i) is greater than the corresponding i^{th} component in the vectors received from the concurrent initiators, it means that the latest tentative local snapshot of P_i was taken on receiving P_j's REQUEST. So, P_j sends a COMMIT message to P_i. On receiving the COMMIT message, P_i turns that tentative local snapshot into a permanent one and discards all the other tentative snapshots of the same color.It is obvious that each node in the system will receive exactly one COMMIT message. Once all the nodes have made their latest tentative local snapshots permanent, the snapshot collection algorithm terminates. The pseudo-code for the snapshot collection algorithm is presented in Figure 3.

Assertion 1 *The concurrent lists at all the initiators, that exchange snapshot information, are identical. The snapshot initiations by all the nodes in the list (including the owner of the list) are pairwise concurrent.*

Let us assume that initially all the nodes are white. Each REQUEST generated by a snapshot initiator carries the initiator's color at the time of starting the snapshot collection. An initiator P_i, with *init_color* white, receives a white REQUEST due to another initiation by

first_req: bit vector of size n;
initiator: boolean; (initially *false*)
edge_num, clock: integer;
node_color, init_color: color; (initially white)
V: integer vector of size n;
weight, send_weight, recv_weight : real;
concurrent: set of node identifiers; (initially an empty set)

Snapshot Initiation by P_j

weight$\leftarrow$1.0; initiator$\leftarrow$true; init_color$\leftarrow$ node_color;
take local snapshot and associate init_color with it; node_color$\leftarrow$black;
edge_num$\leftarrow$number of edges incident on P_j;
send_weight$\leftarrow$weight/edge_num;
along all edges e incident on P_j do
        { weight$\leftarrow$weight$-$send_weight;
          send(P_j, P_j, REQUEST, send_weight, init_color, node_color); }

Operations at P_i on receiving REQUEST along edge e

On receiving (P_j, initiator_id, REQUEST, recv_weight, recv_init_color, msg_color) do
{ if node_color=white then node_color$\leftarrow$msg_color;
  if first_req[initiator_id]$\neq$ 1 then
      { first_req[initiator_id]$\leftarrow$ 1;
        if initiator=true$\wedge$init_color=recv_init_color then concurrent $\leftarrow$concurrent$\cup$initiator_id;
        clock$\leftarrow$local clock value;
        take tentative local snapshot for initiator_id and associate recv_init_color with it;
        edge_num$\leftarrow$number of edges incident on $P_i - 1$;
        if edge_num$\neq$0 then send_weight$\leftarrow$recv_weight/(2$\times$ edge_num);
        else send_weight$\leftarrow$0;
        along all edges incident on P_i except e do
              { send(P_i, initiator_id, REQUEST, send_weight, recv_init_color, msg_color);
                recv_weight$\leftarrow$recv_weight-send_weight;}
        if initiator=true then response_color$\leftarrow$init_color else response_color$\leftarrow$NULL;
        send(P_i, initiator_id, RESPONSE, clock, recv_weight, response_color) to initiator_id; }
  else send(P_i, initiator_id, RESPONSE, NULL, recv_weight, NULL) to initiator_id; }

Operations at P_j on receiving (P_i, P_j, RESPONSE, *clock$_i$*, recv_weight, recv_color)

{ weight$\leftarrow$weight+recv_weight;
  if init_color=recv_color then concurrent$\leftarrow$concurrrent$\cup P_i$;
  if $clock_i$ $\neq$NULL then V[i]$\leftarrow$ $clock_i$;
  if weight=1.0 then find_maximal_snapshot(concurrent, V, P_j); }

find_maximal_snapshot(concurrent, V, P_j)

{ to all $P_i \in$ concurrent send(P_j, V, COMPILE);
  while concurrent$\neq$ Φ do
        { recv(node_id, $\acute{V}$, COMPILE);
          concurrent$\leftarrow$concurrent-{node_id};
          for i = 1 to n do $\tilde{V}$[i]$\leftarrow$maximum(V[i], $\acute{V}$[i]); }
  $\forall$ i: $1 \leq i \leq n$ if $\tilde{V}$[i]=V[i] then send(P_j, V[i], COMMIT);
  reset first_req; }

Figure 3: Maximal snapshot collection in the presence of multiple initiators.

P_j only if both their initiations are concurrent. Let there exist another initiator P_k, such that initiations of P_i and P_k, and initiations of P_i and P_j are pairwise concurrent, but P_k's initiation is causally dependent on P_j's initiation. Then P_k's *init_color* will be black. This is because a black message sent by P_j, after its own initiation, to P_k would turn the P_k's color to black before it initiates snapshot collection. Consequently, P_i's concurrent list will contain P_j, and vice-versa, but neither would contain P_k. Also, P_k's concurrent list would contain neither P_i, nor P_j. P_i and P_j include each other in their concurrent list on receiving each others REQUEST or RESPONSE.

Assertion 2 *The global snapshot collected by the algorithm is consistent.*

Message communication across the communication links is in FIFO order. Hence, if the sender of a message has not recorded the transmission of a message in a tentative local snapshot, then the receiver will not record the reception of the message in its tentative local snapshot, corresponding to the same initiation. So, the snapshot collected by each initiation is consistent. The information sharing phase of the algorithm (subsection 3.2.2) has the effect of taking the union of the consistent snapshots collected by the initiators. According to [9], the union of consistent snapshots is itself a consistent snapshot.

3.3 Optimizations

The following optimizations can be made in the algorithm described in subsection 3.2.

Firstly, a node does not need to maintain multiple tentative snapshots. If multiple nodes initiate snapshot collection concurrently, each of them sends a snapshot REQUEST to a node P_i. Let us assume that P_i has taken a tentative snapshot on receiving initiator P_j's REQUEST. If later it receives a REQUEST from another concurrent initiator P_k, then P_i can take a tentative snapshot in response to P_k's REQUEST and discard the earlier tentative snapshot of the same color.

The algorithm requires a node to take a tentative local snapshot on the arrival of the first REQUEST message for every initiator. Let us look at the situation depicted in Figure 4. Nodes P_2 and P_3 initiate snapshot collection concurrently. The local snapshots of the nodes for P_2's initiation are represented by "X" marks. The local snapshots for P_3's initiation are represented by "O". The local snapshot taken by P_3, when it initiates snapshot collection,

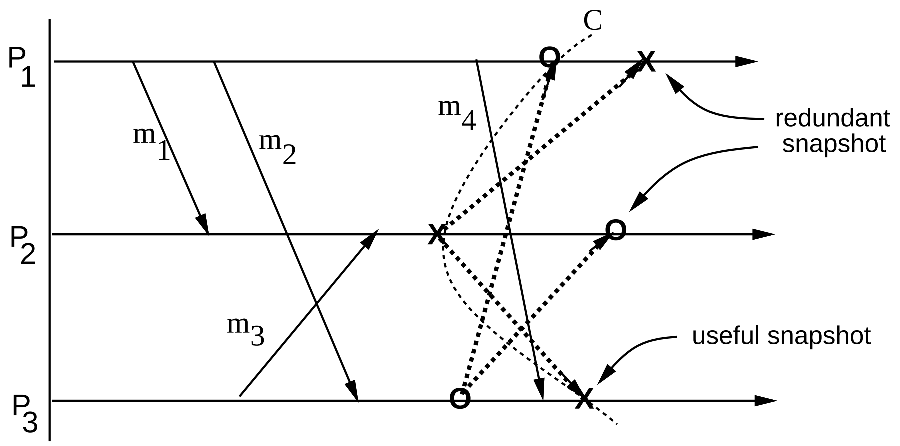

Figure 4: "Redundant" local snapshots on concurrent snapshot collection..

does not register the reception of m_4. But, the tentative local snapshot taken by P_3 on receiving P_2's request provides additional state information by registering the reception of m_4. P_1 has not sent or received any message in the time interval between its two tentative snapshots. So, the local snapshot taken by P_1 in response to P_2's request does not convey any additional information. Hence, it is redundant. By the same argument, the tentative snapshot taken by P_2 on receiving P_3's REQUEST is redundant. So, C represents a maximal consistent snapshot.

Based on the example presented above, the global snapshot collection algorithm can be optimized in the following manner: Let a node P_i take a tentative snapshot on receiving a REQUEST for snapshot collection initiated by P_j. Later P_i receives a REQUEST for another concurrent snapshot collection initiated by P_k. If P_i has not sent or received any computation message since taking the tentative snapshot for P_j's REQUEST, then it can ignore P_k's REQUEST to take a snapshot.

4 Alternative Information Dissemination Strategy for Spezialetti and Kearns' Algorithm

Spezialetti and Kearns use a wave-based information dissemination strategy. Once a snapshot initiator has collected state information about the nodes in its region, it has to share this information with all the other concurrent initiators so that all the initiators have the global snapshot of the system. This sharing requires several rounds of information exchange. In each round the initiator sends any new information, that it obtained in the previous round, to the initiators of the bordering regions. If there exist a pair of regions such that the least separation between them is r regions, then $r + 1$ rounds of information exchange will be required for them share each others information.

During a snapshot collection, it is not mandatory for all the nodes to take their local snapshots. A snapshot initiator can maintain a list of nodes on which it is causally dependent, and send snapshot requests only to those nodes [11]. On receiving a snapshot request, a node propagates the request (if required) only to those nodes on which it is causally dependent. Thus, the minimal set of nodes, required to maintain consistency, are made to take their local snapshots. This saves the effort involved in global snapshot collection. If such a strategy were to be followed, it is likely that an initiator knows the identity of only a subset of all the nodes that initiated snapshot collection concurrently. This situation is similar to Spezialetti and Kearns' algorithm where a snapshot initiator knows the identity of the initiators of its bordering regions only. An initiator can be said to border the initiator(s) whose identity it knows.

In the algorithm, described in Section 3, a snapshot initiator knows the identity of every other node that concurrently initiated snapshot collection. Therefore, a direct exchange of information amongst them is sufficient to construct a consistent global snapshot. In Spezialetti and Kearns' algorithm, or when a minimal set of nodes are made to take their local snapshots, as described above, an initiator may know the identity of only a subset of nodes that initiated snapshot collection concurrently with it. For such situations, we propose an alternative information dissemination strategy. The proposed strategy requires fewer messages than Spezialetti and Kearns' information dissemination strategy. This uses concurrent initiations of the *echo algorithm* by the snapshot initiators.

For the purpose of the proposed information dissemination strategy, we consider the set of concurrent initiators as the nodes of a graph. The initiators of neighboring regions are connected by edges in this graph. When an initiator P_j has collected state information about all the nodes in its region, it executes the following steps:

1. P_j sends *explore* messages along all the edges incident on it, and waits for acknowledgments (*echoes*) to arrive along all these edges.

2. Each echo carries state information about one or more regions in the system. Once echoes have been received along all the incident edges, the state information contained in all of them is combined with the state information of P_j's region. This combined information is the global snapshot of the system.

For every *explore* message received by initiator P_j from a neighbor P_i, along edge e, P_j executes the following steps:

1. The *explore* message is added to a list *wait_explore*.
2. If echo messages have been received by P_j along all the edges besides e, then the state information in those echo messages and P_j's own region state information is combined and sent as an echo message back to P_i. Also, the *explore* message is deleted from *wait_explore*.
3. If P_j has no other neighbor besides P_i, then P_j sends an echo message containing the state of the nodes in its region to P_i as soon as P_j has collected this information. Then *explore* is deleted from the list *wait_explore*.

When P_j receives an echo message, it checks if any *explore* message in the list *wait_explore* can be acknowledged, based on the second condition described above. If so, then the corresponding echo is sent for that *explore* message and it is deleted from *wait_explore*.

Therefore, a node P_j sends an *explore* along an edge e only once, after its own snapshot initiation has terminated. Having sent this *explore* message, P_j does not need to send any more *explore* messages along e to propagate the *explore* messages received along other edges. The echo message received by P_j, along e, in response to the *explore* message, contains the snapshot information at all the nodes in the spanning tree reachable from P_j, through e. P_j combines this information with that received in echoes along other edges to satisfy every *explore* message that it receives. So, only one *explore* and one echo message is sent along each edge in both directions. Ordinarily, if the graph has $\mid E \mid$ edges and one node starts the echo algorithm then $2 \times \mid E \mid$ messages are required. In the algorithm described above, if there are m snapshot initiators, each of which can concurrently and independently start the echo algorithm, then still only $4 \times \mid E \mid$ messages are required.

5 Comparison of the Algorithms

In this section we compare the performance of the proposed algorithm with Spezialetti and Kearns' algorithm in terms of message complexity and the currency of the state information conveyed by the global snapshot.

Unlike Spezialetti and Kearns' algorithm, the algorithm in Section 3 permits full propagation of snapshot REQUESTs generated by all the concurrent snapshot initiations. As a result, the global snapshot collected by the algorithm reflects a more recent state of the system than Spezialetti and Kearns' algorithm. This system state is the maximal of the snapshots collected by the individual initiators. If we refer back to Figure 2, the algorithm will record the reception of message m_4 by P_3, and the transmission of m_5 by P_1 in the global consistent snapshot, while Spezialetti and Kearns' algorithm will not.

One of the most important applications of snapshot collection is to establish checkpoints for site recovery. If a node crashes during the computation, then the recovery from failure is performed by rolling back the computation to a checkpoint (i.e., a recorded consistent global state). The more recent the checkpoint, the less the amount of computation that is lost during recovery. So, the proposed algorithm is more suitable for establishing checkpoints.

As our algorithm does not suppress the propagation of REQUESTs, it has a higher message complexity. Let us assume that there are n nodes in the system of which m concurrently initiate snapshot collection. In Spezialetti and Kearns' algorithm $O(n^2)$ messages are required to color all the nodes and make them take their local snapshots. In our algorithm, each initiator will lead to the generation of $O(n^2)$ REQUESTs. So, the message complexity of the tentative snapshot taking phase is $O(m \times n^2)$. Even though the number of messages required by our algorithm is higher, these messages are being sent concurrently. So, the time taken by our algorithm to collect the tentative snapshots will be comparable to that of Spezialetti and Kearns' algorithm. Our algorithm requires more number of messages. But the extra messages is a price paid for collecting a more recent global snapshot.

In our algorithm each initiator does not gather the global state information. For recovery purposes, each node only needs to know the local checkpoint to which it should roll-back its own computation on node failure. So, each initiator sends only an integer vector of n components to every other initiator after its own snapshot initiation terminates. Each such message may take a path as long as $O(n)$ edges. Thus, the message complexity for establishing a consistent checkpoint is $O(m^2 \times n)$. But, each message has a small size.

In the information dissemination phase, Spezialetti and Kearns make sure that each initiator gets the state information about the entire system. This requires $O(m^2 \times n)$ messages, each containing state information about $O(n/m)$ nodes.

The alternative information dissemination strategy presented in Section 4 has a much better average case performance than the state dissemination strategy of Spezialetti and Kearns. This is because in Spezialetti and Kearns' algorithm, an initiator communicates with its neighbors every time it gets new state information. Each of these communications may contain state information about a potentially small subset of the nodes in the system. In contrast, in our strategy an initiator compiles all the necessary state information, and only then conveys the information to the neighbors. So, an initiator needs to send state information to every neighbor only once.

The example in Figure 5 illustrates the difference between the two information dissemination strategies. The seven nodes, $P_1, \ldots, P_7$, represent seven concurrent snapshot initiators. The edges between the nodes represent the adjacency of the corresponding regions. In the proposed strategy, P_3 sends information to P_1 and P_2 only after it has collected information about its own region and has obtained information from P_4 about the regions corresponding to $P_4, \ldots, P_7$. In Spezialetti and Kearns algorithm, the state information dissemination will

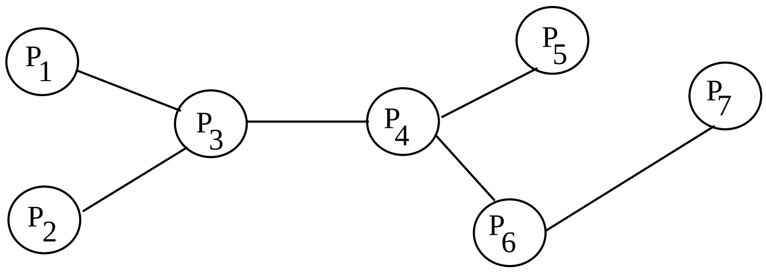

Figure 5: The adjacency graph of concurrent snapshot initiators.

require four rounds of message exchange, and a total of twenty-nine messages — each carrying state information about at least one region. Using the strategy describe in section 4, twenty-four messages will be required, of which twelve are lightweight *explore* messages that carry no state information. As an optimization, in some situations, the state information carried by the echo messages can be piggybacked on the *explore* messages, further cutting down the number of messages. Whenever the degree of connectivity of the concurrent initiator adjacency graph is low and the diameter is high, the proposed information dissemination strategy will perform much better.

The fewer number of messages in our alternative strategy has two advantages: (a) less traffic on the communication network and consequently, less congestion, (b) each message has a constant start-up time. Fewer messages will therefore lead to an earlier completion of the information dissemination phase.

6 Conclusion

Global snapshot collection of a distributed computation is a very important problem [6]. One of its primary applications is in establishing consistent checkpoints for recovery from node failures. On node failures, the computation can be restored to the most recent checkpoint.

The snapshot collection process should not interfere with the underlying computation. Therefore, some snapshot collection algorithms that require the computation to be temporarily suspended are not desirable. In a distributed computation, multiple nodes may concurrently initiate snapshot collection. Most previous snapshot collection algorithms have assumed that only one node initiates snapshot collection at a time. Koo and Toueg [5] consider concurrent snapshot initiations. But in their algorithm, all the initiations may end up aborting, in some situations, leading to a wastage of effort. Spezialetti and Kearns' algorithm [13] imposes restrictions on the propagation of snapshot requests. As a result the global snapshot collected is the minimal of the snapshots collected by all the initiators.

We have presented a global snapshot collection algorithm that handles concurrent initiation of snapshot collection by multiple nodes. It does not inhibit the underlying computation. Unlike Spezialetti and Kearns' algorithm, the propagation of snapshot requests by various initiators is not suppressed. The effect is equivalent to letting multiple nodes collect global snapshots concurrently and independently, and then combining the snapshots to obtain a maximal global snapshot. This maximal snapshot represents a more recent state of the system than the snapshot collected by any initiator.

We have also presented a new strategy for information dissemination that the concurrent initiators can use to share their recorded information and construct a global state. Its average case performance is better than that of the strategy of Spezialetti and Kearns in [13]. Both the strategies have the same worst case performance.

During snapshot collection, it is not mandatory that all the nodes take their local snapshots. Information about causal dependency can be used to make a minimal set of nodes take their local snapshots. If such a strategy is followed it is likely that a snapshot initiator knows the identity of only a subset of all the nodes that initiated snapshot collection concurrently. This situation is similar to Spezialetti and Kearns' algorithm where a snapshot initiator knows the identity of the initiators of its bordering regions only. An initiator can be said to border the initiator(s) whose identity it knows. The new information dissemination strategy proposed in this paper can be used to combine the snapshots collected by each initiator, and obtain a maximal global snapshot in such situations.

References

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

[2] J.-M. Helary. Observing Global States of Asynchronous Distributed Applications. In J-C Bermond and M. Raynal, editors, *Proc. of the* 3^{rd} *International Workshop on Distributed Algorithms*. Springer-Verlag LNCS 392, 1989.

[3] S. T. Huang. Detecting Termination of Distributed Computations by External Agents. In *Proceedings of the* 9^{th} *International Conference on Distributed Computing Systems*, pages 79–84, 1989.

[4] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 3(11):462–491, 1990.

[5] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[6] A. D. Kshemkalyani, M. Raynal, and M. Singhal. Global Snapshots of a Distributed System. Technical Report OSU–CISRC–1/94–TR2, The Ohio State University, 1994.

[7] T.-H. Lai and T.-H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153–158, 1987.

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

[9] F. Mattern. Virtual Time and Global States of Distributed Systems. In M.Cosnard et. al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithm*, pages 215–226. Elsevier Science Publishers B.V.(North-Holland), 1989.

[10] F. Mattern. Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-FIFO Systems. Technical Report SFB124–24/90, University of Kaiserslautern, 1990.

[11] R. Prakash and M. Singhal. Minimal Global Snapshot and Failure Recovery using Infection. Technical Report OSU–CISRC–12/93–TR42, The Ohio State University, 1993.

[12] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 223–238, 1989.

[13] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proceedings of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

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

[15] K. Taylor. The Role of Inhibition in Asynchronous Consistent-cut Protocols. In *Proceedings of the third International Workshop on Distributed Algorithms*, pages 280–291. Springer-Verlag LNCS 392, 1989.

[16] S. Venkatesan. Optimistic Crash Recovery Without Rolling Back Non-Faulty Processors. 1992.

[17] S. Venkatesan and Tony T.-Y. Juang. Low Overhead Optimistic Crash Recovery. Preliminary version appears in Proceedings of 11^{th} International Conference on Distributed Computing Systems, 1991.

[18] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Global state detection in non-FIFO networks. In *Proceedings of the* 7^{th} *International Conference on Distributed Computing Systems*, pages 364–370, 1987.