MESSAGE-OPTIMAL INCREMENTAL SNAPSHOTS

S. Venkatesan[*]

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

The problem of obtaining a global state or snapshot of a distributed processing system has been considered and efficient solutions have been proposed in the literature. They are useful for stable property detection, debugging distributed programs, monitoring distributed predicates and events, efficient rollback recovery in data bases, taking a check point of the system, etc. All of these applications require taking successive snapshots of the system. In this paper, this interesting problem is considered and a message efficient protocol is presented for obtaining an incremental snapshots. It is shown that the protocol uses the minimum number of additional messages possible, and hence it is message-optimal.

1. INTRODUCTION

The problem of constructing a *snapshot* or a *global state* of a *distributed processing system* is among the fundamental problems encountered in distributed systems. The snapshot may be thought of as the state of the entire distributed system at some instant of time assuming that the whole system is *frozen*. The need for snapshots arises not only for the purposes of debugging distributed software [12] and discarding obsolete information in distributed databases [10], but also because a snapshot can be used to monitor distributed events [12], set distributed break points and halt distributed computations [7], for protocol verification [5], and to detect any **stable** property of a distributed computation [2] such as deadlocks and protocol termination. A snapshot is also useful if a check point of a distributed system is to be recorded [1], and for taking a log of the distributed system to record events as the last snapshot taken (the most recent log) would provide crash recovery techniques similar to [8].

In order that the snapshot be accurate and consistent, an accounting be made not only of the state of each process in the network, but also of any messages in transit on the communication links. Chandy and Lamport [2] propose a theory of stable property detection in a distributed system. Their model is based on the partial ordering of events for achieving consistency. They also present a protocol to determine the global state of the system at *some consistent point* of computation.

All of the applications mentioned above require that a large number of successive snapshots be obtained when the distributed system is in operation to get vital information about the components of the system. One approach to obtaining successive snapshots is to run the protocol of Chandy and Lamport [2] every time a snapshot is required. However, this approach uses a large number of messages. Also, rerunning the complete protocol for snapshots many times is an overkill especially when the change between the successive snapshots is small. It is *this* interesting problem that is considered in this paper. Any snapshot protocol generates additional messages and these overhead messages should be minimized for good system performance as the snapshots of the system are taken several times. A widely accepted preformance measure of protocols for distributed systems is the number of overhead messages generated.

We present the notion of *incremental snapshots* in this paper and present two worst case lower bounds on the message complexity of any protocol for incremental snapshots. We then present a protocol for obtaining a snapshot of the system using the most recent snapshot. The message complexity of the incremental snapshot protocol presented in this paper matches the two lower bounds simultaneously, and hence the protocol is *asymptotically message optimal.* Also, our protocol is elegant and easy to implement because of its simplicity.

The paper is organized as follows. The computation model of a distributed system is presented in section 2. The snapshot protocol of [2] is reviewed and a description of the incremental snapshot problem is presented in section 3. Section 4 contains two worst case lower bounds on the message complexity. Section 5 presents a message-optimal protocol and section 6 concludes the paper.

2. SYSTEM MODEL

A distributed processing system can be represented by an undirected graph G = (V,E) where V represents the set of nodes and E represents the set of communication links or channels. At each node, there is a processor available and processes execute on the processors. Whenever no confusion arises, the terms node, processor and process will be used interchangeably. The nodes are connected to each other by an underlying *communication network* which consists of a set of communication links. A communication link connects two nodes of the network, and the two end nodes of a link are called **neighbors** of each other. Message ordering is preserved by the links - messages transmitted at one

[*] Current address: Department of Computer Science, University of Texas at Dallas.

end of a link are received at the other end in the same order in which they are sent. The communication links are bidirectional - the links can be used to transmit messages in either direction. Each bidirectional link can be viewed as a pair of unidirectional links going in opposite directions.

We consider point to point networks with no shared memory. Thus, communication between the nodes is by message-passing only. The nodes and the links incur unpredictable but finite delays in performing their tasks (that is, they are asynchronous).

A *distributed algorithm* or a *protocol* is a collection of *local algorithms* at each of the nodes participating in the protocol. Each local algorithm consists of several *steps*. In a step, a node reads some or all of the messages received from (a subset of) its neighbors, performs some local computation and sends a message to any subset of its neighbors. Each node has a unique *id* (of length $O(\log|V|)$ bits) associated with it, and before the protocol begins, each node knows its own *id* and the *ids* of its neighbors (and the link that leads to each of these neighbors)

To avoid trivial solutions, we assume that the messages are of length $O(\log|V|)$ bits. The performance of a protocol for distributed systems is measured by the **worst-case communication complexity** of the protocol expressed as a function of the number of nodes and/or links in the network - the maximum number of messages generated by the protocol during any execution on any network with the given number of nodes and/or links.

3. SNAPSHOTS OF A DISTRIBUTED SYSTEM

We first consider the problem of obtaining snapshots in a network. A snapshot of a given distributed system can be defined to be a *consistent state* of the entire system. Note that the state of the system consists of the node states and the link states. The state of a node is simply the contents of the local memory, and the state of a link consists of the set of messages sent on that link, but not yet received by the recipient at the other end of the link. Assume that the whole system is frozen at some instance. Now, the node and the link states can be put together, and one can form a snapshot of the system. However, an asynchronous system cannot be frozen simultaneously. Also, the system should not be frozen (even if it is possible to do so) as the distributed computation that is already in progress should not be altered. Thus, a separate protocol has to be used to obtain a snapshot.

In the network, there are at least two distributed protocols running - the original protocol of the *underlying computation* and the *snapshot protocol*. Messages generated by the underlying computation are called *primary* messages while the messages generated by the snapshot protocol are called *secondary* messages or *control* messages. Any snapshot protocol is to be superimposed on the underlying computation; it must run concurrently with, but not alter the underlying computation. A snapshot protocol first attempts to record the states of the processors and the links so that these states form a complete and consistent state of the system. Chandy and Lamport [2] have dealt with the problem of obtaining snapshots of a distributed system in a landmark work and we review their solution.

The snapshot algorithm of [2] has two phases - each node records its component of the global state in the first phase (recording phase), and all of these components are collected to construct the global state in the second phase (dissemination phase). The algorithm augments just one type of message called *markers* to the underlying computation in the first phase. Messages describing the components of the global states are used in the second phase. Assume that a single node p called the central processor initiates the snapshot protocol. First, p records its local state and sends *markers* on all of the links *before* it sends further messages of the underlying computation. The algorithm given in Figure 3.1 is used to process a *marker* message received by the node q from a neighbor on a channel c.
Since the network is connected, every node eventually receives *markers* on all of its links, and thus, each node records its state and the states of all links incident on it. This completes the first phase of the algorithm (recording phase).

Consider the sample network shown in Figure 3.2 with 4 processors and 6 channels.
Let the processor labeled 2 be the central processor. Assume that the distributed system represents an online banking system where the processor state represents the amount of funds available at the node, and the messages represent electronic funds in transit. A snapshot can be used to find the total amount of

---

```
/* executed q receives a marker on c */

if q has not recorded its state then begin
        q records its state.
        q sends one marker along every link after
                recordings its state but before q sends
                another primary message.
        q records state of channel c as empty.
        end
else
        q records the state of the channel c as sequence
                of messages received along c after
                q's state was recorded but before q
                received the marker along c
```

Figure 3.1

---

funds available within the system. Let the state of the system at some instant be as shown in Figure 3.3.
To construct a snapshot, node 1 sends *markers* on all of its links. Before node 4 receives a *marker*, the message in transit (with 10 units of funds) on the link (1,4) will be received and processed. In a similar manner, all of the other messages of the underlying computation are recorded correctly. Finally, when all of the messages in transit reach their respective destinations, and if the messages are combined with the processor states (funds are added), the system state is as shown in Figure 3.4.

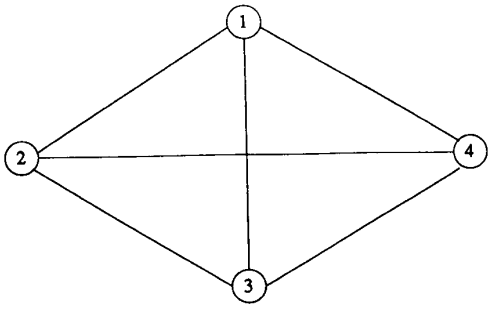

Figure 3.2

After recording its processor state and the states of the incoming links as explained, each node sends its component of the global state to p. Eventually, p receives all of the components and constructs the global state. This completes the second phase of the protocol (dissemination phase). The number of overhead messages used in the first phase (recording phase) of this algorithm is $O(|E|)$ as exactly two *markers* are sent on each link, one *marker* in each direction. In the dissemination phase, we assume that a spanning tree is used to send messages to the central processor for message efficiency. The message complexity of the second phase (dissemination phase) is $O(R|V|)$ where R is the total amount of state information and $|V|-1$ is the number of links in a spanning tree of the network. In the

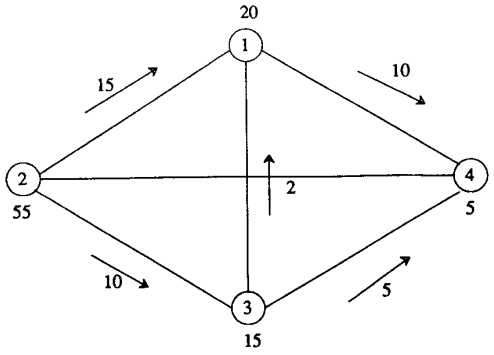

Figure 3.3

example above, the total amount of state information is 1, as the states of the processors and the links can be combined into one message. By piggy backing the *markers* on the messages of the underlying computation, *markers* can be avoided as shown by Lai and Yang [6]. However, this method uses a large amount of space and messages as the complete message history of the channels is sent.

INCREMENTAL SNAPSHOTS

We now consider the problem of obtaining *incremental snapshots*. Let p be a node in the system and

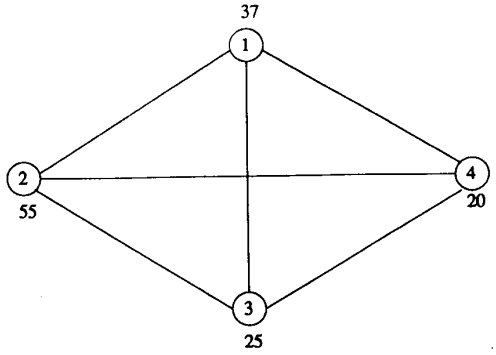

Figure 3.4

let $snap_{t_0}(p)$ be a snapshot of the system initiated by p when the time on the *local clock* of p is t_0. The *incremental snapshot* problem is to obtain $snap_t(p)$ for some $t > t_0$ given that $snap_{t_0}(p)$ is available. As mentioned in section 1, rerunning the protocol of [3] is very expensive in the number of messages. We should be exploit the fact that a recent snapshot of the network is already available, and the change in the system between successive snapshots is likely to be small. Thus, we need to obtain another snapshot of the system using the current snapshot in an efficient manner. For simplicity, assume that each snapshot has a unique version number associated with it, and these numbers are monotonically increasing. Our results remain correct even if this assumption does not hold. We first present two worst case lower bounds on the message complexity of *any* protocol for incremental snapshots running in an asynchronous system. In section 5, we present a protocol for solving the incremental snapshot problem whose message complexity (in the worst case) matches the two lower bounds simultaneously for a large class of applications. Thus, the protocol presented in this paper is asymptotically message-optimal.

4. LOWER BOUNDS ON INCREMENTAL SNAPSHOTS

In this section, we present two worst case lower bounds on the message complexity of *any* distributed algorithm for incremental snapshots operating in an asynchronous network. The first incremental snapshot will be taken only after a complete snapshot of the distributed system is obtained (by running the protocol of Chandy and Lamport [2]) and subsequent (incremental) snapshots are taken only after the completion of the previous incremental snapshots.

Let U be the set of links on which messages have been sent *after* the last snapshot was taken (incremental or otherwise). A message m is defined to be *sent on a link after the last snapshot was taken* if the state of the link recorded by the last snapshot does not include message m, and m was sent by the sender after it recorded its most recent processor state. A protocol is said to *use* a link if at least one message is sent on the link in either direction. Thus, U represents the set of

links of the network which were used by the underlying computation after the last snapshot was recorded. Let U_q be the set of links incident on q on which the processor q sent messages of the underlying computation *after* q recorded its most recent processor state. Thus, $U = \cup U_q$, $q \in V$. We now show that $\Omega(|U|)$ messages are generated in the worst case by *any* snapshot protocol.

Lemma 4.1: Any protocol for incremental snapshots uses at least $|U|$ messages in the worst case.

Proof: We will prove this lemma by contradiction using an adversary. The adversary will keep track of the set of links used by the underlying computation and the links used by the the protocol for incremental snapshots. Assume that there exists a protocol Π which generates less than $|U|$ messages. This implies that there exists at least one channel $c \in U$ such that no control message of Π is sent on c. Since $c \in U$, it is clear that a message m was sent on c by the sender *after* it recorded its most recent processor state. Now, the adversary simply delays the delivery of m to the recipient until Π completes the execution (the adversary can delay any message by an arbitrary but finite amount of time as the links are asynchronous and Π does not send any message on c). After Π completes its execution, the adversary delivers m to the recipient. Clearly, the (incremental) snapshot obtained by Π is incorrect as one message (m in this case) is not included in the snapshot. Thus, the lower bound follows.□

We now show a lower bound of $\Omega(|V|)$ on the message complexity of any protocol which obtains a snapshot (either completely or incrementally).

Lemma 4.2: A lower bound on the message complexity of any distributed protocol for (incremental) snapshots is $\Omega(|V|)$.

Proof: Given two processors i and j, let d(i,j) be the minimum number of links between i and j and let

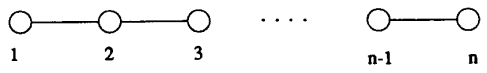

Figure 4.1

$r(i) = \max_j d(i,j)$. The network topology may be linear in the worst case as shown in Figure 4.1, and for the network shown, the value of r(i) is at least $\left\lceil \frac{|V|}{2} \right\rceil$. Thus, in the dissemination phase (where the local states are assembled to form the global state), at least $\Omega(|V|)$ messages are generated. Even if there is no dissemination phase, the lower bound holds because of the following reasoning:

Consider any two sets of processors separated by a distance of $|V|/2$. If these two are the only processors which changed their local states or sent/received messages, the processors in the two sets have to be informed about the need to take an incremental snapshot.□

Theorem 4.3: A worst case lower bound on the message complexity of any distributed protocol for incremental snapshots is $\Omega(|U|+|V|)$.□

In the next section, we present a protocol whose message complexity matches the lower bound of Theorem 4.3.

5. A MESSAGE-OPTIMAL PROTOCOL

In this section, we present a protocol for obtaining an incremental snapshot of a distributed system in a distributed manner. The protocol consists of several procedures. Since snapshots are taken successively, a version number is associated with each snapshot. For simplicity, we assume that the snapshot protocol is initiated by one node p called the central processor. (This assumption can be relaxed by modifying our protocol using the results of [13]). Whenever a processor needs a snapshot of the system, it sends a message to the central processor requesting it to initiate the incremental snapshot protocol. When a processor requests the central processor, or when the central processor decides to obtain a snapshot, the central processor checks if the previous snapshot protocol has completed its execution (the notion of completing a snapshot will be explained shortly). If the previous snapshot protocol has completed its execution, then p initiates the next snapshot (incremental). On the other hand, if the previous protocol is still executing, p waits until it is completed. The node p initiates the (incremental) snapshot protocol, and when it is constructed, the snapshot is sent to the processor(s) that requested a snapshot.

If a central processor is not available in a network, then the minimum spanning tree algorithm of Gallager et al. [4] can be run to elect a leader using $O(|E|+|V|\log|V|)$ messages. For reducing the message complexity of the protocol, a spanning tree of the network is assumed to exist (note that the algorithm of [4] constructs a spanning tree). Since this is a one-time preprocessing step, the message complexity of this step will not be included in analyzing the message complexity of the incremental snapshot protocol.

We now present the main idea of the protocol. In the protocol presented below, p represents the central processor while q and r represent arbitrary processors (including the central processor).

The main idea

We now give an informal description of the main idea of our protocol. Note that each node knows which of its outgoing links are in U (the set of links on which it sent messages after it recorded its processor state the last time), but it does not know exactly which of its incoming links are in U. In the algorithm of [2], all of the outgoing links are assumed to be in U; thus, *markers* are sent on all of the outgoing links and *markers* are received on all of the incoming links. However, sending *markers* on all of the outgoing links generates

O(|E|) messages.

To minimize the number of additional messages sent, we let the node q send *markers* on outgoing links that are in U_q, so that the states of such links are correctly recorded by the nodes at the other end of the links. An acknowledgement is requested for each *marker* sent. If q receives acknowledgements from the nodes at the other end for each *marker* sent, then it is clear that the states of all of the outgoing links of q have been recorded correctly (by the neighbors of q). If every node in the network receives acknowledgements for every *marker* sent, then it is clear that the recording phase of the current incremental snapshot is complete as the state of every link in U is recorded correctly. Termination of the current snapshot is detected using *snap_completed* messages as shown subsequently.

Description of the protocol

We integrate the snapshot protocol of [2] with our incremental snapshot protocol and present a *unified* solution. Each processor q has six local variables - VERSION (the version number of the current snapshot), U_q, the list of neighbors to which messages have been sent since the last snapshot was taken, P_STATE, the saved processor states of the (local) processor, STATE(c) for each incoming channel c (this is used to record the state of the channel c), LINK_STATES (completed channel states that belong to the current snapshot), and LOC_SNAP(i), the i^{th} global state as viewed by q. Initially, U_q is set to the list of neighbors of the node (since the underlying computation may have started before the snapshot protocol began), VERSION is initialized to 0, CHANNEL(c) is set to ϕ for all c and LINK_STATES is set to ϕ (procedure *initialize* in Figure 5.1).

```
/* executed by q before obtaining first snapshot */
procedure initialize;
begin
        VERSION ← 0;
        LINK_STATES ← φ;
        U_q ← list of outgoing links;
        for each incoming channel c do
                STATE(c) ← φ
end;
```

Figure 5.1

The protocol uses four types of control messages - *init_snap, snap_completed, marker* and *ack* messages. *Init_snap* messages are sent (with the current version number) on the tree links to inform all of the processors that a new snapshot is to be obtained. Thus, the processors start executing their respective local algorithms of the protocol. A *snap_completed* message is sent by a processor q to its parent when all of the descendants of q (including q) have recorded their processor states and the states of all of the channels in U that are incident on the descendants of q have been recorded. *Markers* are sent (with the current version number (value of VERSION) of the sender) to flush the messages of the underlying computation that are in transit so that the recipient of the *marker* will correctly record the state of the channel, and *ack*s are sent for each *marker* so that the sender of the *marker* knows if the states of all of its 'used' outgoing links have been recorded. Consider the sample network in Figure 3.2. Assume that the previous snapshot is as shown in Figure 3.4.

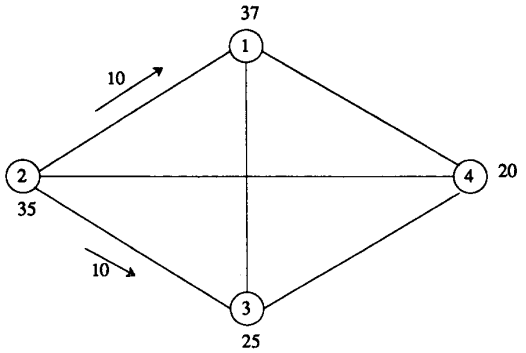

Figure 5.2

Now, if processor 2 sends a message (of the underlying computation) with a value 10 to processor 1 and a similar message to processor 3, then the system state is as shown in Figure 5.2. In this case, U contains only two channels (2,1) and (2,3), and hence only two *markers* need to be sent. If the algorithm of [2] is used, then the number of *markers* sent is 12 (two *markers* on each link). However, note that it is unnecessary to send *markers* on all of the links. Thus, the node 1 sends an *init_snap* message to nodes 2, 3 and 4. On receipt of such a message, processor 2 sends two *markers*, one to processor 1 and one to processor 3. Nodes 1, 3 and 4 do not send any *markers* as they have no link in U incident on them. When a *marker* is received by processor 1, it records the state of the channel (2,1) as the message with value 10 and sends an *ack* to processor 2. Similarly, processor 3 records the state of the channel (3,1) when a *marker* is received and then it sends an *ack* to processor 2. When processor 2 receives an *ack* from 1 and 3, the current snapshot can be collected to form a global snapshot. Our protocol uses only 7 messages (2 *markers*, 2 *ack*s and 3 *init_snap*) messages while the protocol of [3] uses 12 messages (12 *markers*). As shown subsequently, we can further reduce the number of messages to 5.

Init_snap and *marker* messages are sent with the current version number (value of VERSION) to avoid the problem of inconsistency that is created when a *marker* sent by the next invocation of the snapshot is received by a node *before* it receives the *init_snap* message from its parent. Thus, when q receives *ack*s for each *marker* sent, it is obvious that all of the messages sent by q have been recorded (either as part of the processor state of the recipient or as part of the incoming channel of the recipient). We now present the details of the protocol.

Whenever a message that belongs to the

underlying computation is sent by q on a link c, the sender adds c to U_q, the list of channels on which messages of the underlying computation have been sent by q (procedure *send_und* in Figure 5.3). This is recorded so that *markers* will be sent only on those channels that may have messages in transit.

```
/* executed when q sends a primary message to r */

procedure send_und(q:origin; r:destination; c:channel;
        m:message);
begin
        U_q ← U_q ∪ {c};
        send m to q on the channel c;
end;
```

Figure 5.3

Any time after procedure *initialize* is executed, whenever a message that belongs to the underlying computation is received on a channel, the message is stored as a possible part of the channel state and procedure *receive_und* is executed (Figure 5.4).
When a *marker* is received on a channel, it is clear that those messages that have been received after q recorded its most recent processor state but *before* the *marker* was received belong to the current snapshot. Hence, those message are saved in LINK_STATES. Also, an *ack* message is sent on c to inform the processor at the

```
/* executed when a primary message is received */

procedure receive_und(q:origin; r:destination; c:channel;
        m:message);
begin
        if RECORD(c)=true then
                STATE(c) ← STATE(c) ∪ {m};
                /* add m to state of c */
        pass m to the underlying computation
end;
```

Figure 5.4

end of c that the state of c had been fully recorded (procedure *receive_marker* in Figure 5.5).

To initiate the (incremental) snapshot protocol, the central processor p sends an *init_snap* message to itself. In response to this message, p completes the previous snapshot (with number VERSION) and stores the previously saved processor state and the set of link states (stored in the local variable LINK_STATES) in the local variable LOC_SNAP(VERSION). Since a new snapshot is needed, it saves its processor state, increments the value of VERSION, starts recording messages received on all of its incoming links (sets the state of each link to null and sets RECORD to true so that procedure *receive_und* will record the messages of the underlying computation), and sends an *init_snap*

```
/* executed when q receives a markerfrom a neighbor */

procedure receive_marker(r:origin; q:destination;
        c:channel; m:message);
begin
        if VERSION < VERSION in marker then
                begin
                /* a marker of next invocation is received
                before the init_snap message */
                send a init_snap(VERSION+1) to itself;
                receive_initiate;
                /* execute procedure receive_initiate */
                STATE(c) ← φ
                end;
        LINK_STATES ← LINK_STATES ∪ {STATE(c)};
        RECORD(c) ← false;
        /* stop recording messages received on c */
        send an ack on c as a reply
end;
```

Figure 5.5

```
/* executed when q receives a init_snap message
        from its parent */

procedure receive_initiate(q:destination; r:parent;
        c:channel; m:message);
begin
        if VERSION < VERSION in init_snap message
                then begin
                LOC_SNAP(VERSION) ← P_STATE(q) ∪
                        LINK_STATES;
                /* the previous snapshot is over */
                LINK_STATES ← φ;
                P_STATE(q) ← {current processor state};
                /* for the current snapshot */
                VERSION ← VERSION + 1;
                for each incoming channel c do begin
                        STATE(c) ← φ;
                        RECORD(c) ← true;
                        /* record messages received on c */
                        end;
                send an init_snap message to each child;
                for each c ∈ U_q do send a marker
                        on c and wait for an ack message;
                U_q ← φ;
                wait for a snap_completed message
                        from each child;
                if q is the central processor then
                        present snapshot is complete
                else
                        send a snap_completed message
                                to the parent;
                end
        else
                discard the init_snap message
                /* already received a marker with this
                version & called procedure receive_marker */
end;
```

Figure 5.6

message to all of its children. After this, the node p sets U_p and LINK_STATES to ϕ. It also sends a *marker* on each link in U_p and waits for an ack for each *marker* sent. When a *snap_completed* message is received from each child, and an ack is received for each *marker* sent, it is clear that all of the states have been recorded, and the current snapshot has been correctly recorded. These steps are given in procedure *receive_initiate* (Figure 5.6).

The behavior of the other processors is similar to that of the central processor. Thus, they all execute procedure *initiate* before beginning to execute any other procedure of the snapshot protocol. When they send a message that belongs to the underlying computation, procedure *send* will be executed, and for each message received, one of the procedures *receive_und, receive_initiate* or *receive_marker* will be executed depending on the type of the message received. When a *marker* with VERSION higher than that of a node is received, or a *init_snap* message is received, a node infers that its part of the previous snapshot is complete. Thus, each node can locally determine when its part of the snapshot procedure is complete.

When a node receives a *marker* with VERSION higher than the value of the local variable VERSION, it sends an *init_snap*(VERSION+1) message to itself (as per procedure *receive_marker* in Figure 5.5). Thus, p need not send an *init_snap* message to its child r if $(p,r) \in U_p$. Instead, it sends a *marker* on (p,r). Similarly, each interior node q in the spanning tree sends a *marker* on (q,s) to its child s instead of an *init_snap* message if $(q,s) \in U_q$. For the sample network in Figure 5.2, only five messages are sent - two *markers*, two acks, and one *init_snap* message.

We prove the correctness of the protocol before analyzing the communication complexity.

CORRECTNESS

To prove the correctness of the protocol, we show that every message in transit is recorded by the receiver of the message and then show that every message recorded by the protocol indeed belongs to the current snapshot.

Theorem 5.1: Every message in transit that belongs to the current snapshot is recorded by the recipient as part of the current snapshot.

Proof: This theorem can be proved by contradiction. Assume that there exists one message m that is not recorded. Let (q,p) be the link on which the message m was sent. Clearly, $(q,p) \in U_q$, and this fact is known to q. Thus, the node q sends a *marker* on (q,p) after q sends m and waits for an acknowledgement from p. If m is not recorded by p, then the only possibility is that the *marker* is received before m be the node p, a contradiction, as we assumed that the links obey the first-in-first-out order for messages.□

Theorem 5.2: Every message recorded by the processors as part of the current snapshot belongs to the current snapshot.

Proof: Consider a link (q,p) such that $(q,p) \in U_q$. When q receives an *init_snap* message, it sends a *marker* on (q,p) and waits for an acknowledgement from p (so that q knows that the state of (q,p) is correctly recorded by p) before q completes its part of the incremental snapshot protocol. If (q,p) does not belong to U_q, then no message has been sent by q, and hence the state of (q,p) is ϕ. In either case, the state of all of the links incident of the nodes are completely and correctly recorded. Thus, the proof of this theorem follows.□

The next two theorems are easy to prove.

Theorem 5.3: The processor states are consistent.□

Theorem 5.4: The snapshot constructed by the incremental snapshot protocol is consistent and correct.□

COMPLEXITY ANALYSIS

We now analyze the total number of control messages used by the protocol for obtaining one snapshot. For each invocation of the snapshot protocol, we count the number of additional messages. Since there are $|V|-1$ tree links in the network, and since one *initiate* message is sent downward on each tree link and one *snap_completed* message is sent upward on each tree link, the total number of *initiate* and *snap_completed* messages is $2(|V|-1)$. Also, exactly one *marker* and one *ack* message is sent on each link in U, and hence the total number of messages generated by our protocol is $O(|V|+|U|)$. Thus, the following theorem can be proved using the above argument.

Theorem 5.5: The message complexity of the incremental snapshot protocol is $O(|U|+|V|)$.□

Theorem 5.6: The message complexity of (the problem of) recording the incremental snapshots locally is $\Theta(|V|+|U|)$ for each invocation.

Proof: The proof of this theorem follows from Theorem 5.5 and the lower bounds presented in section 4.□

For many applications the states of the processors and the channel states can be combined into one message. For example, for discarding obsolete information, the nodes compute the min function [10]. In such applications, LOC_SNAP(VERSION) can be represented by one message (or a constant number of messages in general) for any invocation by combining all of the information into one value (for example, the function min can be computed using the local processor state and the states of the incoming links and the resulting value is used). Also, the links of the spanning tree are used so that the state of the children and the local states are combined to obtain one message. Since the number of links in a spanning tree is $|V|-1$, the dissemination phase uses $O(|V|)$ messages. For taking a log of the system, note that the local components of the snapshot need not be sent to the coordinator as the

system will start from the last snapshot that was successfully recorded in case of failure(s). Thus, our algorithm is message-optimal for the global state dissemination phase also.

Theorem 5.7: For a large number of applications, there exists a *message-optimal* protocol for obtaining incremental snapshots.□

6. CONCLUSIONS

In this paper, we first presented a problem that arises often in distributed systems, namely, obtaining incremental snapshots. This problem has immediate applications in database systems, debugging distributed programs, monitoring events, checkpointing, etc. For all of the applications, several snapshots are to be taken. An obvious solution is to run the algorithm of Chandy and Lamport [2]. However, this approach generates a large number of overhead messages. We first presented worst case lower bounds on the number of messages used by *any* protocol that solves the incremental snapshot problem and presented a protocol that is message-efficient. For all of the applications mentioned, our protocol is message-optimal. Because of its simplicity, our protocol can be used readily, and since it uses the minimum number of additional messages, the throughput of the distributed system is not adversely affected.

It is assumed that the communication channels transmit messages in the first-in-first-out manner. This assumption is valid in many distributed systems as the lower level protocols of the computer network ensure this using sequence number, etc [14]. Even if messages are sent out of order on channels, the messages of the underlying computation can be piggy backed with the value of VERSION, and the total number of messages sent between successive snapshots can be recorded by the sender of the messages can be sent to the recipient. With this information, it is easy to see that our protocol can be modified to operate correctly.

There are several possible directions in which future research can continue. For example, fault-tolerant incremental snapshots is an interesting and important problem that arises in faulty networks. Faulty networks are considered in [9, 11] in the context of obtaining obtaining complete snapshots. It is interesting to develop incremental snapshot protocols resilient failures.

References

1. Chandy, K.M., *Private communications.*
2. Chandy, K.M. and Lamport, L., "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems* **3**(1), pp. 63-75 (1985).
3. Cohen, S. and Lehmann, D., "Dynamic systems and their distributed termination," *Symposium on Principles of Distributed Computing*, pp. 29-33 (1982).
4. Gallager, R. G., Humblet, P.A., and Spira, P.M., "A distributed algorithm for minimum weight spanning trees," *ACM Transactions on Programming Languages and Systems* **5**(1), pp. 66-77 (1983).
5. Geihs, K. and Seifert, M., "Automated validation of a co-operation protocol for distributed systems," *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pp. 436-443 (1986).
6. Lai, T.H. and Yang, T.H., "On distributed snapshots," *Proceedings of the IEEE Infocom 87, Conference on Computer Communications*, pp. 342-346 (1987).
7. Miller, B. and Choi, J., "Breakpoints and halting in distributed programs," *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pp. 316-323 (1988).
8. Powell, M. and Presotto, D., "Publishing: a reliable broadcast communication mechanism," *Proceedings of the ninth ACM Symposium on Operating System Principles*, pp. 100-109 (1983).
9. Ramarao, K.V.S. and Venkatesan, S., "Fault-tolerant distributed snapshots," *Technical Report, University of Pittsburgh, Pittsburgh* (1987).
10. Sarin, S.K. and Lynch, N., "Discarding obsolete information in a replicated database system," *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 39-47 (1987).
11. Shah, A. and Toueg, S., "Distributed snapshots in spite of failures," *Technical report, Cornell University* (1987).
12. Spezialetti, M., "Global state as a paradigm for distributed control," *Ph.D. thesis, University of Pittsburgh (in preparation).*
13. Spezialetti, M. and Kearns, J.P., "Efficient distributed snapshots," *Proceedings of the sixth International Conference on Distributed Computing Systems*, pp. 382-388 (1986).
14. Tanenbaum, A., *Computer networks,* Prentice-Hall, Inc., Engelwood Cliffs (1981).