Distributed Snapshots for Mobile Computing Systems*

Adnan Agbaria William H. Sanders
Coordinated Science Laboratory and
Electrical and Computer Engineering Department
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana IL 61801, USA
{adnan, whs}@crhc.uiuc.edu

Abstract

Distributed snapshots are an important building block for distributed systems, and are useful for constructing efficient checkpointing protocols, among other uses. Direct application of these algorithms to mobile systems is not feasible, however, due to differences in the environment in which mobile systems operate, relative to general distributed systems. This paper presents a distributed snapshot algorithm that is well-suited to mobile systems, which often have limited bandwidth and long latencies, and where the mobile hosts may roam among the different cells within the system. In addition to presenting the protocol itself, we prove its liveness and safety. As was the case for the classical distributed snapshots protocol and distributed systems, we believe that this protocol will be an important building block for mobile systems.

1. Introduction

The mobile computing environment introduces new challenges in the area of fault-tolerant computing. Compared to traditional distributed environments, wireless networks are typically slower, providing lower throughput and latency, comparing to wireline networks. In addition, the mobile hosts have limited computation resources, are often exposed to harsh operating environment that makes them more likely to fail, and can roam while operating.

Distributed snapshots is a traditional technique for providing persistence and fault tolerance in distributed systems. More specifically, they are key building block for implementing checkpoint/restart (C/R) protocols [7]. *Checkpointing* is the act of saving an application's state to stable storage during its execution, while *restart* is the act of restarting the application from a checkpointed state. If checkpoints are taken, then when an application fails, it can be restarted from its most recent checkpoint. This limits the amount of computation lost because of a failure to the time that elapsed between the last checkpoint and the failure. Checkpointing can also be used in migrating a process from one computer to another, and for debugging purposes.

One of the main challenges in implementing C/R mechanisms is that of maintaining low overhead, since otherwise the cost of taking a checkpoint will outweigh its potential benefit. Another problematic challenge is that of finding a collection of checkpoints, one from each process, that corresponds to a *consistent* view of the distributed application's state for restarting. A distributed application's state is not consistent if it represents a situation in which some message m is received by a process, but the sending of m is not in the checkpoint collection. A collection of checkpoints that corresponds to a consistent distributed state forms a *recovery line*.

Over the past two decades, intensive research work has been carried out on providing efficient C/R protocols in traditional distributed computing, e.g., [7, 14]. Recently, more attention has been paid to providing C/R protocols for mobile systems [15, 11]. Some of these protocols have been adapted from the traditional distributed environment; others have been created from scratch for mobile systems. In both cases, the majority of the protocols can be classified as *communication-induced checkpointing* (CIC) protocols, in which processes take local checkpoints in an uncoordinated manner. The key challenge in creating practical protocols of this type is finding and maintaining a recovery line and minimizing the drop in performance due to the large number of *forced* checkpoints, where usually some processes are forced to take checkpoints to ensure that a recovery line exists [2, 7].

*This research has been supported by DARPA contract F30602-00-C-0172.

We proved in [3] that the classical distributed snapshots protocol presented by Chandy and Lamport [6], which is a *coordinated* protocol, is the most efficient of the well-known protocols. This protocol takes the checkpoints *on-the-fly*, such that it does not suspend any execution of the system as well as C/R. In addition, the distributed snapshots protocol determines the *global state* of the system. Such global state helps us to detect some properties, such as deadlock and termination.

In this paper we present a distributed snapshot protocol for the mobile environment, that is a robust adaptation of the classical distributed snapshots technique and that can be used for achieving an efficient C/R protocol. It is an *adaptive distributed snapshots* (hereafter, *ADS*) protocol for mobile environments. We first discuss why the classical distributed snapshots protocol cannot be applied straightforwardly to the mobile environment. Based on some observations related to the mobile environment and the distributed snapshots protocol, we provide an adaptation to allow the protocol to be implemented for mobile computing systems. In addition, we prove the properties of liveness and safety.

The remainder of this paper is organized as follows. Section 2 describes the system model and basic definitions. In Section 3, we present our protocol, some running examples, and proof of liveness and safety for the protocol. In Section 4, we describe previous related work, and conclude our work in Section 5.

2. Preliminaries

2.1. System Model

We use the system model presented in [4, 11]. In this model, a mobile computing system consists of n *mobile hosts* (*MHs*), $h_1, \cdots, h_n$, and m *mobile support stations* (*MSSs*), $M_1, \cdots, M_m$, where $n > m$. Figure 1 shows an example of a typical mobile computing system in which the MSSs are connected by a static wired network and each MH is connected by a wireless network to one MSS. A *cell* is a logical or geographical coverage area under an MSS. An MH can *directly* communicate with an MSS M_i only if it is present in the cell serviced by M_i. At any time, every MH h_i, $1 \leq i \leq n$, belongs to only one cell. The static network provides reliable First-In-First-Out (FIFO) delivery of messages between any two MSSs with arbitrary message latency. Similarly, the wireless network within a cell ensures reliable FIFO delivery of messages between an MSS and an MH.

Consider Figure 1 when an MH h_1 wants to send a message to another MH h_4, h_1 first sends the message to its local MSS M_1 over the wireless network. Then, M_1 forwards the message to M_3 over the static network.

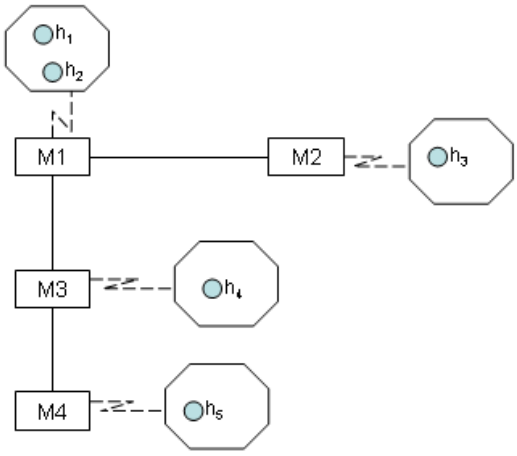

Figure 1. An example of a mobile computing system

Finally, M_3 forwards the message to h_4 over the wireless network. Since the location of an MH is not fixed, before M_1 forwards the message to M_3, it should first determine the MSS that currently serves h_4. This problem has been tackled through several routing protocols for mobile networks [5, 8]; we don't address it here.

When an MH h_i leaves a cell served by an MSS M_i, it sends a *leave(r)* message to M_i, where r is the sequence number of the last message received by h_i. After sending this message, h_i neither sends nor receives any other message from M_i. Then, M_i deletes h_i from its ID list of MHs that are local to its cell. On the other hand, when h_i enters a new cell served by M_j, it sends a *join(h_i)* message to M_j. Then, M_j adds h_i to its ID list.

In this paper, we consider a distributed computation in a mobile computing system that consists of N processes, $P_1, P_2, \cdots, P_N$, running concurrently on different MHs. For simplicity, we assume that each MH runs one process and $n = N$. Message passing is the only way of communication. The computation is asynchronous. Each process P_i is modeled as an automaton with a predefined initial state e_i, and a deterministic transition function from its current state to the next state based on the current state and the *event* it occurs. The *normal* possible events are `computation`, `send`, and `receive`. In addition, we define another two possible events that can happen: `log` and `checkpoint`. The *log* event consist of saving a message in secondary storage, and the *checkpoint* event consist of saving the local state in secondary storage.

Given a process p, we say that the process p is in a *normal* state if it performs only normal events, and we say that p is in a *saving* state if it is able to perform

the log and checkpoint events in addition to the normal events.

The *local history* of a process is a sequence of such events. An *execution* is a collection of local histories, one for each process. For each receive event in an execution, there is one corresponding send event, and for each send event, there is at most one receive event. Moreover, if the execution is infinite, then for each send event there is exactly one corresponding receive event. For a message m in the execution, **Send**(m) denotes the send event of m, and **Recv**(m) denotes the receive event of m. Events in an execution are related by the *happened before* relation [9]; this relation is defined as the transitive closure of the process order and the relation between the send and receive events of the same message.

2.2. Definitions and Notations

When a failure occurs in a distributed system, we need to recover from a *cut* of checkpoints (i.e., a set of checkpoints consisting of one checkpoint from each process). However, not all cuts of checkpoints are *consistent*, i.e., correspond to a state that could have been reached in the execution. A consistent cut of checkpoints is called a *recovery line*.

Definition 2.1: Given an execution E and a cut of checkpoints $S \in E$, the *partial* execution of E corresponding to S, denoted by $E|_S$, is the collection of local histories of each process $p \in E$ up to the checkpoint event in S.

Definition 2.2: Given an execution E and a cut of checkpoints $S \in E$, S is a *recovery line* if for every message m, if **Recv**(m) $\in E|_S$, then **Send**(m) $\in E|_S$.

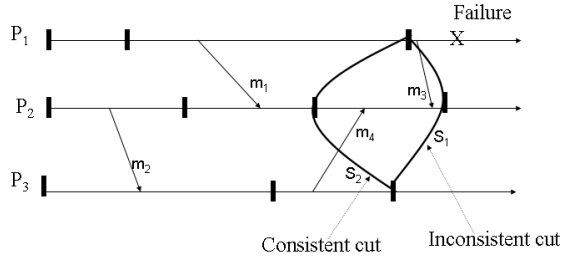

Figure 2. An example of distributed execution

For example, let E be an execution as presented in Figure 2. If a failure occurs in process P_1 after m_3 is sent, the execution cannot be recovered from the latest cut of checkpoints S_1, since it is not a recovery line. The reason is that **Recv**(m_3) $\in E|_{S_1}$, but **Send**(m_3) $\notin E|_{S_1}$. Thus, the execution needs to rollback to the latest recovery line, which is S_2. Notice here that **Send**(m_4) $\in E|_{S_2}$, but **Recv**(m_4) $\notin E|_{S_2}$. We call m_4 an *in-transit* message relative to S_2. In our example, m_4 should be *logged* in order to be retransmitted by the recovery mechanism.

Definition 2.3: Given an execution E and a recovery line $R \in E$, R is called a *distributed snapshot* if every in-transit message in E relative to R is logged and can be retransmitted for further recovery from R.

Given a mobile system $\mathcal{M}$ and an MSS $M \in \mathcal{M}$, we use $\mathcal{N}(M)$ to denote all the MSSs in $\mathcal{M}$ that are *neighbors* to M. Two MSSs M_1 and M_2 are neighbors if there is a direct *channel* between them in the static network. Moreover, we use $\mathcal{C}(M)$ to denote all the MHs that belong to the cell served by M. Notice here that since an MH can leave/join a cell dynamically, we assume that $\mathcal{C}(M)$ is updated dynamically according to any join/leave operation.

3. A Distributed Snapshot Protocol

In this section, we describe our distributed snapshots protocol for mobile systems. To simplify the presentation of the ADS protocol and show the differences between this protocol and the classical distributed snapshots protocol [6], we first present the latter protocol as given by Chandy-Lamport [6] (hereafter, the C-L protocol), and then show why this protocol cannot be implemented straightforwardly in mobile systems. We start by defining some data structures to help us describe the protocols.

3.1 Data Structures and Variables

In presenting the distributed snapshots protocol, we consider the following data structures and variables:

Marker - This is the marker message used to coordinate a distributed snapshot. It can be sent by any machine in the system. It contains an integer number, which is **Marker.num**. This number is attached to the marker to indicate the corresponding snapshot number.

x**.num -** An integer number, which indicates the latest snapshot number that the machine x knows.

x**.state -** A flag variable, which indicates the state of the machine x. The state could be either normal or saving.

P_M - An integer number, maintains the ID of the first MSS that sends the new marker to M. Such MSS is called the parent of M in the current round of the protocol.

N_M, C_M - A data structures for maintaining $\mathcal{N}(M)$ and $\mathcal{C}(M)$ respectively for an MSS M.

3.2. The C-L protocol

The classic distributed snapshots protocol [6] works as follows. Let $\mathcal{N}(p)$ be the set of processes that have direct communication with a process p. If p receives a marker message **Marker** from another process q such that $q \in \mathcal{N}(p) \cup \{\emptyset\}$, p deals with the marker by calling the function **receiving** presented in Figure 3.

```
/* p receives a message from process q */
receiving(msg, q)
/* The marker could be sent by any process */
1: If (msg = Marker), then
2:    If ((p.state = normal) and (Marker.num > p.num))
         p.state = saving
         p.num = Marker.num
         N_p = N(p) /* N is the set of all the processes */
         Checkpoint()
         ∀x ∈ N_p, send(Marker, x)
3:    If ((p.state = saving) and (Marker.num = p.num))
         If (q ∈ N_p), then N_p = N_p \ {q}
         If (N_p = ∅), then p.state = normal
4: Else /* The message is an application message */
         If (p.state = saving) and (q ∈ N_p), then Log(msg)
```

Figure 3. The behavior of every process p according to the C-L protocol

As presented in Figure 3, in the C-L protocol, when a process p receives a marker, it switches to the saving state, takes a local checkpoint, and forwards the marker to its neighbors. p logs all the intransit messages. p identifies an intransit message as an incoming message from a process $q \in \mathcal{N}(p)$ such that p has sent the marker to q, but has not received it back yet.

In [6], it was specified that this protocol works only if all the channels provide FIFO delivery. In Figure 4 we show an example in which the C-L protocol produces an inconsistent global state if FIFO is violated. In this example, process p sends the marker and then the message m after recording its state to q. However, process q receives m before the marker; hence, the cut $\{C_1, C_2\}$ is not consistent.

In mobile systems, FIFO communication between the MHs may not naturally occur, for many reasons. For example, suppose that h_1 and h_3 (in Figure 1) are communicating. Assume that h_1 sends m_1 and then moves to $\mathcal{C}(M_2)$. After it joins $\mathcal{C}(M_2)$, assume that h_1 sends m_2 to h_3. In that case, m_2 may arrive at h_3 before m_1 does.

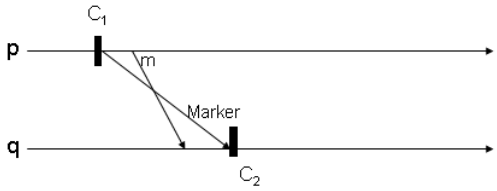

Figure 4. The cut $\{C_1, C_2\}$ represents an inconsistent global state

A straightforward way to support distributed snapshots for mobile computing would be to provide FIFO among MHs. However, providing FIFO could be impractical if it causes delays in message delivery and thus decreases system performance. Instead, we claim that the FIFO property should be satisfied only between the marker messages and the application messages, but not for any two messages in the system. In other words, all the application messages that have been sent after a marker should be received after the marker, and vice versa.

We now present a distributed snapshots protocol for mobile computing that makes uses of this idea. The protocol does not require a FIFO channel between any two MHs; instead, it insures that for any two MHs h_1 and h_2 on a mobile system, if h_1 sends a message m to h_2 after receiving the marker, then h_2 will receive the marker before receiving m.

3.3. Adaptation to Mobile Systems

In the adaptive distributed snapshots (ADS) protocol, the MSSs are responsible for forwarding the marker and logging the intransit messages. Since in mobile systems the MHs have limited resources [4], we have designed the protocol such that most of the work done by the MSSs rather than the MHs. The ADS protocol works as follows:

- If an MSS $M \in \mathcal{M}$ receives the marker message **Marker** from another machine x such that $x \in \mathcal{N}(M) \cup \mathcal{C}(M) \cup \{\emptyset\}$, M processes with the message by calling the function **MSS_receiving** presented in Figure 5.
- For every MH $h \in \mathcal{C}(M)$ for an MSS M, if h receives the marker message **Marker** from M such that **Marker.num** $>$ h.**num**, h updates its snapshot number and sends **Marker** back to M with its

local state. Otherwise, if **Marker.num** $\leq h$.**num**, h updates **Marker.num** and sends it back to M.

- An MSS M that is in the saving state saves any incoming messages from machine x, where $x \in N_M$.
- If an MSS M receives a local state of an MH $h \in \mathcal{C}(M)$, M saves the local state in stable storage.
- If an MH h joins a $\mathcal{C}(M)$ group, h's first action is to exchange the marker values between h and M. Subsequently, if h receives an updated marker, it does a local checkpoint and sends it back to M. On the other hand, if M receives an updated marker from h, it behaves as if it was receiving a new marker.

Notice here that the marker could be initialized by any MSS machine on the system. This is reflected in the assumption indicated in Step 1, where a marker message may come from an empty set. Figure 5 shows the **MSS_receivingMarker** function that is called by an MSS machine when it receives a message.

MSS_receiving(msg, x) /* $x \in \mathcal{N}(M) \cup \mathcal{C}(M) \cup \{\emptyset\}$ */
1: If (msg = Marker), then
2: If ((M.state = normal) and (Marker.num $> M$.num))
M.state = saving
M.num = Marker.num
$C_M = \mathcal{C}(M)$
$N_M = \mathcal{N}(M) \setminus \{x\}$
$P_M = x$ { Update the parent }
$\forall y \in (N_M \cup C_M)$, **send**(Marker, y)
3: If ((M.state = normal) and (Marker.num $< M$.num))
Marker.num = M.num
send(Marker, x)
4: If (M.state = saving) and (Marker.num $= M$.num))
If ($x \in C_M$), then $C_M = C_M \setminus \{x\}$
If ($x \in N_M$), then $N_M = N_M \setminus \{x\}$
If ($C_M = \emptyset$), then **send**(Marker, P_M)
If ($N_M = \emptyset$) and ($C_M = \emptyset$), then M.**state** = normal
5: Else /* The message is an application message */
If (M.state = saving) and ($x \in N_M$), then **Log**(msg)

Figure 5. An MSS machine M receives a marker message from another machine x.

Figure 6 shows the **MH_receivingMarker** function that is called by an MH machine when it receives a marker from its corresponding MSS machine.

MH_receivingMarker(Marker, M)
If (Marker.num $> h$.num), then
h.num = Marker.num
C = **Checkpoint**()
send(M, ⟨ Marker, C ⟩)
Else
Marker.num = h.num
send(M, Marker)

Figure 6. An MH machine h receives a marker message from its corresponding MSS machine.

3.4. Running Examples

We illustrate here our distributed snapshot protocol with an example in which we use it with a set of running on the mobile environment. In these examples, we illustrate the correct functioning of the protocol under different scenarios that may happen in a mobile system. Formal proofs of the protocol properties are given in the next section.

Consider the example of a mobile system presented in Figure 1. In the system there are four MSSs, $M_1, \cdots, M_4$, and five MHs, $h_1, \cdots, h_5$. Assume that at the beginning of the run, all the local snapshot numbers are zero. Assume that a snapshot is started by M_1. Below we describe in detail a scenario that could happen in the system during the taking of the snapshot, according to our protocol.

1. M_1 increments the marker number to 1 and broadcasts the marker to $\mathcal{N}(M_1) = \{M_2, M_3\}$ and $\mathcal{C}(M_1) = \{h_1, h_2\}$.
2. h_1 sends the message m_1 to h_3 before receiving the marker.
3. h_1, h_2, and h_3 receive the marker and take their local checkpoints. Since M_2 is still in the saving state (assume it does not receive the marker back from h_3), then in Step 4 it saves the message m_1.
4. h_3 receives m_1, but only after taking its local checkpoint.
5. The marker continues to propagate until all the MHs have taken their checkpoints.

At the end of this run, we have a snapshot that consists of a consistent cut (S_1 in Figure 7), and the intransit message m_1 is saved with global checkpoints.

Another running example of the system presented in Figure 1 is as follows.

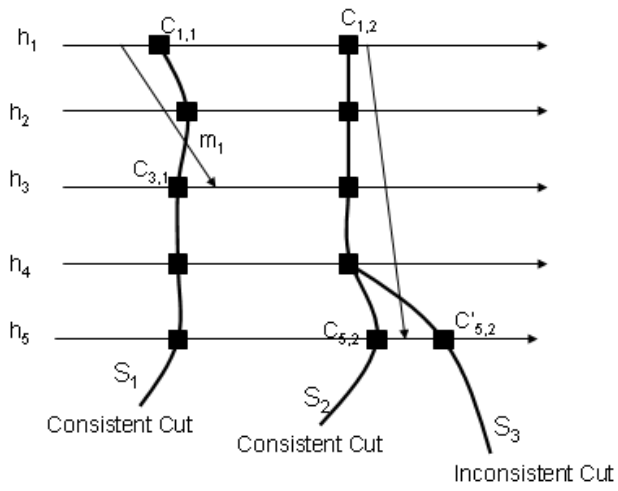

Figure 7. An example of a mobile system for illustrating the protocol

1. M_1 increments the marker number to 2 and broadcasts the marker to $\mathcal{N}(M_1) = \{M_2, M_3\}$ and $\mathcal{C}(M_1) = \{h_1, h_2\}$.
2. h_1 takes a local checkpoint, leaves $\mathcal{C}(M_1)$, and joins $\mathcal{C}(M_4)$. The token does not reach M_4 at this point.
3. h_1 sends the marker to M_4. M_4 then switches to saving state and starts the snapshot.
4. h_1 sends a message to h_5.
5. h_5 receives the message sent by h_1 only after taking its local checkpoint.
6. The marker continues to propagate until all the MHs have taken their checkpoints.

At the end of this run, we have a snapshot that consists of a consistent cut (S_2 in Figure 7). Notice that if h_1 does not propagate the marker with it when it joins $\mathcal{C}(M_4)$, then we will not get a snapshot, but it will consist of an inconsistent cut S_3.

3.5. Protocol Properties

In this section we prove the safety and progress of the adaptive distributed snapshots protocol. For safety, we show that the protocol produces distributed snapshots. Specifically, we show that all the checkpoints on the MHs form a recovery line and every in-transit message relative to this recovery line is logged as well. For progress, we show that after every snapshot, all the MSSs and MHs switch to normal state. We start by proving how fast the marker propagates on the mobile systems.

Although the routing protocols in a mobile system are beyond the scope of our work, we believe that the marker propagation mechanism used here is the fastest mechanism in mobile environments.

Claim 3.1: Given a mobile system $\mathcal{M}$, once a marker is initialized in $\mathcal{M}$, it propagates faster than any other message in $\mathcal{M}$.

Proof: We prove this claim by observing the marker routes in $\mathcal{M}$. By the ADS protocol, once an MSS M initializes (or receives) a marker, it broadcasts it to its neighbors. Then, every neighbor broadcasts it. Obviously, any message m that the MSS M wants to send, regardless of the used routing protocol, it should send m to its neighbor(s) first. Then, the neighbor(s) continues according to the routing protocols. Hence, m will be transmitted to some particular paths from M to its target. On the other hand, since the marker is sent to all the neighbors, it will be transmitted to all the possible paths from M to its target, where it meets the optimum path in the network during propagation.

Furthermore, since by the ADS protocol, the first operation that an MH does during the join process to a new cell is to exchange its marker with the corresponding MSS. □

Lemma 3.2: Within a distributed snapshot of the ADS protocol, every MH h takes a local checkpoint.

Proof: Once a marker is initialized in an MSS M, by Claim 3.1, the marker propagates to all the MSSs in the system, where every MSS in $\mathcal{N}(M)$ sends it to its neighbors. As a result, since we assume that there is reliable point-to-point communication between the MSSs, eventually every MSS will receive the marker.

In Step 2 of the **MSS_receiving** function (see Figure 5), when an MSS receives the marker, it broadcasts the marker to the MHs in its cell. Therefore, by the reliable assumption within the same cell, every MH will receive the marker. As a result, by function **MH_receivingMarker** (presented in Figure 6), every MH takes a local checkpoint within the current distributed snapshot. □

Lemma 3.3: For any two MHs h_1 and h_2 on a mobile system, if h_1 sends a message m to h_2 after receiving the marker, then h_2 will receive the marker before receiving m.

Proof: We start this by contradiction. In particular, assume that for some message m, h_2 receives the marker after m while h_1 has sent m after receiving the marker. Figure 4 illustrates such a situation, where p takes the place of h_1, and q takes the place of h_2. We examine now all the possible forms h_1 and h_2 could take in the system.

First, assume that h_1 and h_2 belong to the same cell, for example $h_1, h_2 \in \mathcal{C}(M)$, for some MSS M. In Step 2 of Figure 5, h_1 receives the marker after M has sent it to $\mathcal{C}(M)$. By the protocol, M should send the marker to h_2 at the same time it sends the marker to h_1. On the other hand, since the message m is sent by h_1 after receiving the marker, then M should forward m to h_2 after sending the marker to h_2. By the FIFO assumption within the same cell, h_2 should receive the marker before m, contradicting our assumption.

Second, assume that $h_1 \in \mathcal{C}(M_1)$ and $h_2 \in \mathcal{C}(M_2)$, where $M_1 \neq M_2$. By [6] and the FIFO assumption between the MSSs, the marker is received consistently by the MSSs M_1 and M_2 with any application message between them. In other words, if an application message m' is sent by M_1 to M_2 after M_1 has the marker, then M_2 should receive the marker before receiving m'. In Step 2 of Figure 5, when M_2 receives the marker, it forwards it directly to h_2. Then, by the FIFO assumption within the same cell, h_2 should receive the marker before receiving the message m, again contradicting our assumption.

Lastly, assume that h_2 has not yet received m and joins the cell of h_1 after the marker has been distributed in this cell. Then by the ADS protocol, h_2 will receive the latest marker that was distributed in this cell before receiving any further messages, including the message m, contradicting our assumption. □

Lemma 3.4: Every in-transit message is logged by the ADS protocol.

Proof: By definition, a message m from h_1 to h_2 is in-transit if h_1 sends m before taking its checkpoint, but h_2 receives m after taking its checkpoint. By Step 5 in Figure 5, an MSS M, which is in a saving state, logs every incoming message. Since M switches to the saving state before the MHs in its cell (Step 2, Figure 5). By Step 4 in Figure 5, M remains in the saving state until all its MHs have taken their checkpoints and it receives the marker back from its neighbors. Thus we guarantee that every incoming message that could be received by an MH $h \in \mathcal{C}(M)$ after a checkpoint is logged by M. In other words, every in-transit message is logged. □

Theorem 3.5: The ADS protocol produces distributed snapshots for mobile systems.

Proof: By Lemma 3.2, every MH takes a local checkpoint during the ADS protocol. By Lemma 3.3, the cut of checkpoints is a recovery line. Finally, by Lemma 3.4, every intransit message relative to the recovery line is logged. Therefore, the ADS protocol produces distributed snapshots. □

Now we prove the progress of the ADS protocol. We show that if an MSS enters a saving state by the ADS protocol, it switches to the normal state after a finite period of time.

Lemma 3.6: If an MSS M is in a saving state, then it receives the markers from all the hosts in $\mathcal{N}(M) \cup \mathcal{C}(M)$.

Proof sketch: By Step 2 in Figure 5, when an MSS M switches to the saving state upon receiving (or initializing) a marker, it defines the set N_M to be $\mathcal{N}(M) \cup \mathcal{C}(M)$. By Step 4, when the marker returns to M from $x \in N_M$, M extracts x from N_M. M switches to the normal state when it receives the markers back from all the members of N_M.

By Figure 6, if an MH $h \in \mathcal{C}(M)$ receives a marker, it sends the marker directly back to M. By assuming of reliable communication between M and $\mathcal{C}(M)$, all the markers should be received back by M. Similarly, by Step 4 in Figure 5 and the reliable assumption, eventually every $M' \in \mathcal{N}(M)$ sends the marker back to M.

Notice here that while M is in the saving state, the set N_M may change dynamically to reflect the dynamic change of $\mathcal{C}(M)$, e.g., an MH can join/leave the cell. By the ADS protocol, if an MH h joins the cell $\mathcal{C}(M)$, it initially exchanges the marker with M. Therefore, if M was in the saving state, it updated N_M accordingly. □

Corollary 3.7: If an MSS M enters the saving state, then it will switch back to the normal state after a finite time.

Proof: Since we assume reliable point-to-point communication between M and its MHs, then eventually every $h \in \mathcal{C}(M)$ will send back the marker. In addition, due to the reliable assumption among the MSSs, eventually every $M' \in \mathcal{N}(M)$ will send back the marker to M. As a result, eventually M will switch to the normal state. □

4. Related Work

A considerable body of work is available on C/R for traditional distributed systems, e.g., [7, 12, 13]. As pointed out in [7], the C/R protocols for traditional distributed systems are classified into three classes: coordinated checkpointing, uncoordinated checkpointing, and

CIC. Observing the C/R protocols for mobile systems, this major classification still holds, where most of the protocols belong to CIC.

Pradhan et al. [15] presented an evaluation for C/R protocols in mobile computing systems. They analyzed some well-known C/R protocols to determine the main parameters that affect their performance. These parameters include the wireless bandwidth, the communication-mobility ratio of the user, and the failure rate of the mobile host. In order to minimize the overhead of our protocol, we tried not to minimize the communication through the wireless bandwidth. Also we keep the major work of the protocol to be done by the MSSs, but not the MHs.

Manivannan and Singhal [11] presented a CIC C/R protocol for mobile systems. This protocol focuses on solving the challenging problem of finding and maintaining a recovery line. Therefore, the protocol produces some force checkpoints that could increase the overhead. On the other hand, since our protocol does not take any additional checkpoints, the recovery line is limited to the latest cut of checkpoints.

Lin and Dow [10] presented a three-phase hybrid C/R protocol designed for mobile systems. This protocol requires coordination among MSSs in the first phase. Then it applies a CIC technique among the MHs in the second phase. Lastly, a timeout interval is considered in the third phase. The ADS protocol does not use the CIC technique, which complicates the recovery mechanism, and there are no timeouts.

Acharya and Badrinath [1] presented a CIC checkpointing protocol for mobile computing systems, where a process takes a checkpoint whenever a message reception is preceded by a message transmission. Obviously, this protocol causes as many forced checkpoints as the number of messages if the message reception and transmission are interleaved, resulting in high overhead. The ADS protocol does not have forced checkpoints at all.

5. Conclusions

We have presented a robust adaptation of the classical distributed snapshots protocol to mobile systems. In addition to presenting the protocol itself, we present a proof of correctness and progress of the protocol. As was the case for the classical distributed snapshots protocol [6], this protocol facilities solutions for important problems in distributed systems, such as checkpointing, and global state detection.

In previous work [3], we identified the distributed snapshots protocol as the most efficient protocol in distributed systems among a set of known checkpointing protocols for distributed systems. We believe that the adaptive distributed snapshots protocol presented in this paper is likewise an efficient checkpointing protocol for the mobile environment, in which efficiency is very important in meeting the needs of limited resources in the mobile hosts.

Acknowledgments

We would like to thank Kaustubh Joshi for his helpful comments and Jenny Applequist for her editorial assistance.

References

[1] A. Acharya and B. R. Barinath. Checkpointing Distributed Applications on Mobile Computing. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 73–80, September 1994.

[2] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg. Quantifying Rollback Propagation in Distributed Checkpointing. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, pages 36–45, New Orleans, USA, October 2001.

[3] A. Agbaria, A. Freund, and R. Friedman. Evaluating Distributed Checkpointing Protocols. In *Proceedings of the 23rd International Conference of Distributed Computing Systems*, pages 266–273, Providence, Rhode Island, May 2003.

[4] B. R. Badrinath, A. Acharya, and T. Imielinski. Structuring Distributed Algorithms for Mobile Hosts. In *Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS)*, pages 21–28, June 1994.

[5] P. Bhagwat and C. E. Perkins. A Mobile Networking System Based on Internet Protocol (IP). In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 69–82, August 1993.

[6] 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.

[7] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Department of Computer Science, Carnegie Mellon University, June 1999.

[8] J. Ioannidis, D. Duchamp, and G. Q. Maguire. IP-based Protocols for Mobile Internetworking. In *Proceedings of ACM SIGCOMM Symposium on Communications, Architectures and Protocols*, pages 235–245, 1991.

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

[10] Cheng-Min Lin and Chyi-Ren Dow. Efficient Checkpoint-based Failure Recovery Techniques in Mobile Computing Systems. *Journal of Information Science and Engineering*, 17:549–573, 2001.

[11] D. Manivannan and M. Singhal. Failure Recovery based on Quasi-Synchronous Checkpointing in Mobile Computing Systems. Technical Report OSU-CISRC-7/96-TR36, The Ohio State University, Department of Computer and Information Science, 1996.

[12] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[13] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, January 1993.

[14] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

[15] D. K. Pradhan, P. Krishna, and N. H. Vaidya. Recovery in Mobile Environments: Design and Trade-Off Analysis. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, June 1996.