Abstract
Minimum-process synchronous snapshot compilation is a suitable approach to introduce fault tolerance in mobile distributed systems transparently. In order to balance the snapshot compilation overhead and the loss of computation on recovery, we propose a composite snapshot compilation algorithm, wherein an all-process synchronous snapshot is capture after the execution of minimum-process synchronous snapshot compilation algorithm in a fixed number of times. In synchronous snapshot compilation, if a single procedure fails to capture its snapshot; all the snapshot compilation effort goes waste, because, each process has to recompute its tentative snapshot. In order to capture the tentative snapshot, an Mobile Host (M_HOST) needs to transfer large snapshot data to its local Mobile Support Station (MOB_SUPP_ST) over wireless channels. Hence, the loss of snapshot compilation effort may be exceedingly high. Therefore, we propose that in the first phase, all concerned M_HOSTs will capture Temporary snapshot only. Temporary snapshot is similar to mutable snapshot, which is stored on the memory of M_HOST only. In this case, if some procedure fails to capture snapshot in the first phase, then M_HOSTs need to recompute their Temporary snapshots only. The effort of capturing a Temporary snapshot is negligibly small as compared to the tentative one. In the minimum-process synchronous snapshot compilation algorithm, an effort has been made to minimize the number of useless snapshots and blocking of procedures using probabilistic approach.

Keywords: snapshot, distributed system, synchronous snapshot compilation, probabilistic procedures

1. Introduction
Mobile Hosts (M_HOSTs) are increasingly becoming common in distributed systems due to their availability, cost, and mobile connectivity. They are also considered suitable for effective and efficient disaster management. In case of disaster, the static connectivity may not work; therefore, we have to depend on mobile computing environments in such cases. An M_HOST is a computer that may retain its connectivity with the rest of the distributed system through a wireless network while on move. An M_HOST communicates with the other nodes of the distributed system via a special node called mobile support station (MOB_SUPP_ST). A “cell” is a geographical area around an MOB_SUPP_ST in which it can support an M_HOST. An MOB_SUPP_ST has both wired and wireless links and it acts as an interface between the static network and a part of the mobile network. Acharya-Badrinath [1] proposed that Static nodes are connected by a high speed wired network. A snapshot is a local state of a procedure saved on the stable storage. In a distributed system, since the procedures in the system do not share memory, a global state of the system is defined as a set of local states, one from each procedure. The state of channels corresponding to a global state is the set of messages sent but not yet received. A global state is said to be “coherent” if it contains no orphan message; i.e., a message whose receive event is recorded, but its send event is lost as proposed by Chandy-Lamport [5]. To recover from a failure, the system restarts its execution from the previous coherent global state saved on the stable storage during fault-free execution. This saves all the computation done up to the last snapped state and only the computation done thereafter needs to be redone.

In synchronous or synchronous snapshot compilation, procedures capture snapshots in such a manner that the resulting global state is coherent. Mostly it follows the two-phase commit structure [2]. In the first phase, processes capture tentative snapshots, and in the second phase, these are made stable. The main advantage is that only one stable snapshot and at most one tentative snapshot is required to be stored. In the case of a fault, processes rollback to the last snapped state [5]. The Chandy-Lamport algorithm [5] is the earliest non-blocking all-process synchronous snapshot compilation algorithm. In this algorithm, markers are sent along all channels in the network which leads to a message complexity of $O(N^2)$, and requires channels to be FIFO.

We have to deal with various issues while designing a snapshot compilation algorithm for mobile distributed systems [1]. These issues are mobility, disconnections, finite power source, vulnerable to physical damage, lack of stable storage etc. Prakash-Singhal [19] proposed a nonblocking minimum-process synchronous snapshot compilation protocol for mobile distributed systems. They proposed that a good snapshot compilation protocol for mobile distributed systems should have low overheads on M_HOSTs and wireless channels; and it should avoid awakening of an M_HOST in doze mode operation. The disconnection of an M_HOST should not lead to infinite wait state. The algorithm should be non-intrusive and it should force minimum number of procedures to capture their local snapshots. In minimum-process synchronous snapshot compilation algorithms, some blocking of the processes captures place [2] or some useless snapshots are captured [2].

2. Basic Idea
In the proposed snapshot compilation scheme, initiator procedure collects the dependency vectors of all processes and computes the tentative minimum set [2]. Suppose, during the execution of the snapshot compilation algorithm, $P_i$ captures its Temporary snapshot and sends $m_i$ to $P_j$ as shown in Figure 3.1. $P_j$ receives $m_i$ such that it has not captured its snapshot for the current initiation and it does not know whether it will get the snapshot request or not. If $P_j$ captures its snapshot after processing $m_i$, $m_i$ will become orphan. In order to avoid such orphan messages, we use the following technique.

Temporary-Snapshot Based Composite Synchronous Checkpointing Protocol for Mobile Distributed Systems
Author(s): 1Dyuti Banerjee, 2Praveen Kumar
Affiliation: 1Research Scholar, 2Professor, NIMS University, Jaipur (Rajasthan India)
P\textsubscript{i} sends \(m_3\) to \(P\textsubscript{j}\) after capturing its Temporary snapshot. \(P\textsubscript{j}\) receives \(m_3\) such that i) \(P\textsubscript{j}\) has received the mset[\(i\)] from the initiator process, ii) \(P\textsubscript{j}\) does not belong to mset[\(i\)] and iii) \(P\textsubscript{j}\) has not captured its snapshot for the current initiation. In this case, we have two options: (i) \(P\textsubscript{j}\) \ may capture mutable snapshot before processing \(m_3\), ii) \(m_3\) is buffered at \(P\textsubscript{j}\) till \(P\textsubscript{j}\) captures its Temporary snapshot or \(P\textsubscript{j}\) receives the tentative snapshot request, whichever is earlier. We propose the probabilistic approach as follows. Suppose \(P\textsubscript{k}\) has sent \(m_5\) to \(P\textsubscript{i}\) and \(P\textsubscript{k}\) belongs to mset[]. In this case, if \(P\textsubscript{j}\) receives \(m_2\) before capturing its Temporary snapshot, then \(P\textsubscript{j}\) will be included in the minimum set. Alternatively, if \(P\textsubscript{j}\) receives \(m_2\) after capturing its Temporary snapshot (shown by dotted message \(m_2\) in the figure 3.1), then \(P\textsubscript{j}\) will not receive snapshot request due to \(m_2\). Hence we can say that if \(P\textsubscript{j}\) has sent \(m_3\) to \(P\textsubscript{k}\) such that \(P\textsubscript{k}\) belongs to mset[] then most probably \(P\textsubscript{j}\) will get the snapshot request. In this case, we propose that \(P\textsubscript{j}\) should capture its mutable snapshot before processing \(m_3\). Here, if \(P\textsubscript{j}\) gets the regular snapshot request it will convert its mutable snapshot into Temporary one. Alternatively, if \(P\textsubscript{j}\) does not receive the snapshot request, it will discard its mutable snapshot on receiving tentative snapshot request. Suppose there does not exist any process \(P\textsubscript{k}\) such that \(P\textsubscript{j}\) has sent some message to \(P\textsubscript{k}\) and \(P\textsubscript{k}\) belongs to mset[]. In this case, we can say that most probably \(P\textsubscript{j}\) will not get snapshot request for the current initiation. Here, if \(P\textsubscript{j}\) captures its mutable snapshot before processing \(m_3\), then most probably \(P\textsubscript{j}\) will have to discard its mutable snapshot. Therefore, we propose that \(P\textsubscript{j}\) should buffer \(m_3\). \(P\textsubscript{j}\) will process \(m_3\) only after getting the tentative snapshot request or after capturing the Temporary snapshot whichever is earlier.

In minimum-process snapshot compilation, some processes may not be included in the minimum set for several snapshot initiations due to typical dependency pattern; and they may starve for snapshot compilation. In the case of a recovery after a fault, the loss of computation at such processes may be unreasonably high. In Mobile Systems, the snapshot compilation overhead is quite high in all-process snapshot compilation [5]. Thus, to balance the snapshot compilation overhead and the loss of computation on recovery, we design a composite snapshot compilation algorithm for mobile distributed systems, where an all-process snapshot is captured after executing of minimum-process algorithm for fifteen number of times.

In the first phase, the relevant M_HOSTs are required to capture Temporary snapshot only. Temporary Snapshot is stored on the disk of the M_HOST and is similar to mutable snapshot [8]. If any procedure fails to capture its snapshot in coordination with others, then all relevant procedures need to repudiate their Temporary snapshots only. In case of repudiate, the loss of snapshot compilation effort will be very low as compared to two phase algorithms. In mobile distributed systems, we may expect frequent repudiates due to exhausted battery, abrupt disconnections etc.

3. Proposed Protocol

We explain the proposed minimum-process snapshot compilation algorithm with the help of an example. In Figure 2, at time \(t_0\), \(P\textsubscript{4}\) initiates snapshot compilation process and sends request to all processes for their dependency vectors. At time \(t_0\), \(P\textsubscript{4}\) receives the dependency vectors from all processes and computes the tentative minimum (mset[]) set, which in case of Figure 2 is \(\{P\textsubscript{5}, P\textsubscript{6}, P\textsubscript{5}, P\textsubscript{6}\}\) due to messages \(m_3\), \(m_2\) and \(m_4\). \(P\textsubscript{4}\) sends this tentative minimum set to all processes and captures its own Temporary snapshot. A process captures its Temporary snapshot if it is a member of the mset[]. When \(P\textsubscript{3}\), \(P\textsubscript{5}\) and \(P\textsubscript{6}\) get the mset[], they find themselves to be the members of mset[]; therefore, they capture their Temporary snapshots. When \(P\textsubscript{6}\), \(P\textsubscript{1}\) and \(P\textsubscript{2}\) get the mset[], they find that they do not belong to mset[], therefore, they do not capture their Temporary snapshots.

\(P\textsubscript{5}\) sends \(m_8\) after capturing its Temporary snapshot and \(P\textsubscript{1}\) receives \(m_8\) after getting the mset[]. When \(P\textsubscript{5}\) sends \(m_8\) to \(P\textsubscript{1}\), \(P\textsubscript{5}\) also piggyback \(pr\textsubscript{csn}\) and \(pr\textsubscript{c_state}\) along with \(m_8\). When \(P\textsubscript{1}\) receives \(m_8\) it finds that \(csn[5]<m.pr\textsubscript{csn}\) and \(m.pr\textsubscript{c_state}=1\). \(P\textsubscript{1}\) concludes that \(P\textsubscript{5}\) has captured its snapshot for some new initiation. \(P\textsubscript{1}\) also finds \(rec\textsubscript{mset}=1\); it implies \(P\textsubscript{1}\) has received the mset[] for the new initiation and \(P\textsubscript{1}\) is not a member of mset[]. Further, \(P\textsubscript{1}\) has not sent any message to any process of the mset[]. In this case, \(P\textsubscript{1}\) concludes that most probably it will not be included in the minimum set for the current initiation; therefore \(P\textsubscript{1}\) buffers \(m_8\) and processes it only after getting the tentative snapshot compilation request.
After capturing its Temporary snapshot, \( P_4 \) sends \( m_{11} \) to \( P_2 \). At the time of receiving \( m_{11} \), \( P_2 \) has received the \( mset[] \) and it \( P_2 \) is not the member of the \( mset[] \). \( P_2 \) finds that it has sent \( m_1 \) to \( P_3 \) and \( P_3 \) is a member of \( mset[] \). Therefore, \( P_2 \) concludes that most probably, it will get the snapshot request in the current initiation; therefore, it captures its mutable snapshot before processing \( m_{11} \). When \( P_3 \) captures its Temporary snapshot, it finds that it is dependent upon \( P_2 \), due to \( m_3 \), and \( P_2 \) is not in the \( mset[] \); therefore, \( P_3 \) sends Temporary snapshot request to \( P_2 \). On receiving the snapshot request, \( P_2 \) converts its mutable snapshot into Temporary one. It should be noted that the Temporary snapshot and mutable snapshot are similar. Mutable snapshot is a forced snapshot and Temporary snapshot is a regular snapshot captured due to snapshot request. In order to convert the mutable snapshot into Temporary snapshot, we only need to change the data structure (\( mob_supp_st_local_Temporary[2]=1 \)).

After capturing its snapshot, \( P_1 \) sends \( m_{13} \) to \( P_1 \). \( P_1 \) finds that it has not sent any message to a process of tentative minimum set. It captures the bitwise logical AND of \( pr_sendv[1][] \) and \( mset[] \) and finds the resultant vector to be all zeroes (\( pr_sendv[1][]=[0000001] \); \( mset[]=[1111000] \)). \( P_1 \) concludes that most probably, it will not get the snapshot request in the current initiation; therefore, \( P_1 \) does not capture mutable snapshot but buffers \( m_{13} \). \( P_1 \) processes \( m_{13} \) only after getting the tentative snapshot request. \( P_0 \) processes \( m_{14} \) because it has not sent any message since last stable snapshot (\( pr_send=0 \)). After capturing its snapshot, \( P_4 \) sends \( m_{12} \) to \( P_3 \). \( P_3 \) processes \( m_{12} \), because it has already captured its snapshot in the current initiation. At time \( t_2 \), \( P_4 \) receives positive responses to Temporary snapshot requests from all relevant processes (not shown in the Figure 2) and issues tentative snapshot request along with the exact minimum set \( [P_2, P_3, P_4, P_5, P_6] \) to all processes. It should be noted that if any process fails to capture its Temporary snapshot, then all the relevant processes need to repudiate their Temporary snapshots and not the tentative snapshots. The effort of capturing a tentative snapshot is exceedingly high as compared to Temporary snapshot in mobile distributed systems. In this way, we try to reduce the loss of snapshot compilation effort if any process fails to capture its snapshot in coordination with others. On receiving tentative snapshot request, all relevant processes convert their Temporary snapshots into tentative ones and inform the initiator. A process, not in the minimum set, discards its mutable snapshot, if any; or processes the buffered messages, if any. Finally, at time \( t_3 \), initiator \( P_4 \) issues commit. On receiving commit following actions are captured. A process, in the minimum set, converts its tentative snapshot into stable one and discards its earlier stable snapshot, if any.

4. Conclusion

We propose a composite snapshot compilation algorithm, wherein, an all-process synchronous snapshot is captured after the execution of minimum-process synchronous snapshot compilation algorithm for a fixed number of times. In minimum-process snapshot compilation, we try to reduce the number of useless snapshots and blocking of processes using a probabilistic approach. Concurrent initiations of the proposed protocol do not cause its
concurrent executions. In the first phase, all concerned processes capture Temporary snapshots only. In this way, we try to reduce the loss of snapshot compilation effort when any process fails to capture its snapshot in coordination with others. We have also reduced the size of the integer snapshot sequence number to four bits. It is piggybacked onto normal computation messages. The present algorithm can be modified for its application in distributed systems and ad hoc networks. The actual number of useless snapshots and number of messages blocked can be computed by simulation results.

References