Improvements and Reconsideration of Distributed Snapshot Protocols

Adnan Agbaria
IBM Haifa Research Lab
Mount Carmel, Haifa 31905, Israel
Email: adnan.agbaria@gmail.com

Abstract

Distributed snapshots are an important building block for distributed systems, and, among other applications, are useful for constructing efficient checkpointing protocols. In addition to the imposed overhead of the existing distributed snapshot protocols, those protocols are not trivially applicable (if at all) in many of today's distributed systems, e.g., grid, mobile, and sensors systems. After presenting the shortages and the inapplicability of the most popular existing distributed snapshot protocols, this paper discusses improvement directions for the protocols. In addition, it presents a new and an important improvement for the most popular distributed snapshot protocol, which was presented by Chandy and Lamport in 1985. Although the proposed improvement is simple and easy to implement, it has significant benefits in reducing the software and hardware overheads of distributed snapshots. Then, the paper presents proofs for the safety and progress of the new protocol. Lastly, it presents a performance analysis of the protocol using stochastic models.

1. Introduction

Computer systems are improving rapidly as showed by the many new technologies and computing environments that have appeared recently. In the last two decades, we have observed many new computer systems running on those new computing environments. Grid computing, mobile computing and wireless computing are only few examples of such new environments. Without doubt, new technologies are needed to meet the requirements of those environments for building efficient and dependable computer systems. On the other hand, those new systems are more conservative than traditional systems on hardware and software overheads, such as high performance and resources consumptions. We believe that many distributed protocols, originally designed for traditional distributed systems, cannot be applied efficiently (if at all) in the new systems [4, 14]. Several researchers have tried to re-design many existing distributed protocols for the new computing environments; however, those re-designed protocols may not be efficient and may run only in few particular environments with specific assumptions. Therefore, the challenge here is to design new protocols or improve existing protocols to meet the dependability and efficiency requirements of the new environments.

In this paper we focus on *distributed snapshots*, which is a basic problem in distributed systems [6]. Distributed snapshots provide a traditional technique for ensuring persistence and fault tolerance in distributed systems. In addition, distributed snapshots are key building blocks for implementing checkpoint/restart (C/R) protocols [9] and many other properties in distributed systems. *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.

Traditionally, there are two well-known distributed snapshot protocols. In the first protocol, *Sync-and-Stop* (denoted by SaS) [2, 16], a distributed snapshot is taken after all the processes have been synchronized and have stopped their execution. The second protocol is the Chandy and Lamport's (denoted by CL) distributed snapshot protocol [6], which does not stop the execution of the application. It was shown in [2] that these two protocols are the most efficient among a set of distributed checkpointing protocols in two different traditional systems. In addition to the efficiency of CL comparing with SaS [2], considering the real world, the applicability of the CL protocol is more common than the SaS protocol. CL is implemented in many distributed systems such as Starfish [3] and Condor [1].

Since the last two decades, many researchers tried to improve the SaS and CL protocols. Elnozahy et al. [10] presented an improvement on the SaS protocol. They proposed a protocol that does not stop the execution of the application. Instead, it piggybacks timestamps on the application messages to construct distributed snapshots. On the other hand, Mattern [15] and Schulz et al. [19] eliminated the FIFO restrictions in the CL protocol. They proposed

25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)
0-7695-2677-2/06 $20.00 © 2006 IEEE

a non-FIFO CL protocol by piggybacking control information onto the application messages. Actually, it was proven by [7] that in non-FIFO systems any distributed snapshot protocol is either *inhibitory* (i.e., suspending the application execution, in particular suspending the sending of application messages while waiting for a control message from another process), or it relies upon piggybacking control information onto the application messages. Therefore, we believe, as was shown in [2], that the original CL protocol for FIFO systems is the most efficient protocol among a large set of distributed checkpointing protocols. However, as we show later, this protocol still needs to be improved to meet the efficiency requirement in many new environments.

In this paper, we describe the SaS and CL protocols. We outline the main disadvantages of these protocols in the traditional distributed environment and show how and why these protocols cannot be applied easily (if at all) in some new environments like grid and wireless ad-hoc networks. Furthermore, we present a new and important improvement for the CL protocol. This improvement aims at reducing the number of log accesses by the CL protocol. Regardless of any distributed system, a log access imposes both software and hardware overheads. Due to log accesses, the system performance is degraded, therefore the software overhead could be intolerable in high-performance systems. Since log accesses require saving data to a persistent memory (either internal or external memory), hardware resources are consumed. For example, in wireless network systems, the power consumption and memory usage due to log accesses is considered very costly. To the best of our knowledge, we are the first to consider the log accesses issue in the CL protocol. Therefore, our scheme is a new and novel scheme that reduces dramatically the number of log accesses in CL.

2. System Model and Definitions

We consider a distributed system consisting of n processes, denoted by $P_1, P_2, \cdots, P_n$, connected by a network (a process P_q may be denoted by q). Processes communicate via asynchronous reliable message passing. 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 occurring *event*. The *normal* possible events are `computation`, `send`, and `receive`. In addition, we define two more possible events: `log` and `checkpoint`. The `log` event consists of saving a message in a persistent memory (e.g., secondary storage), and the `checkpoint` event consists of additionally saving the local state in secondary storage. The *local history* of a process is a sequence of such events. An *execution* is a collection of local histories, one for each process.

By default, we assume that the network delivers messages reliably, in FIFO order [6]. We mention explicitly If FIFO is not supported. For each `receive` event in an execution, there is a corresponding `send` event; and for each `send` event, there is, at most, one `receive` event. If the execution is infinite, 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 [13]; 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. Each checkpoint taken by a process is assigned a unique sequence number. The ith checkpoint of process p is denoted by $C_{p,i}$.

When a failure occurs in a distributed system, it maybe recovered 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 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) $\in E|_S$.

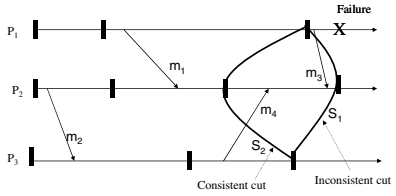


Figure 1. An example of distributed execution

For example, let E be an execution as presented in Figure 1. 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) $\notin E|_{S_1}$. m_3 is called an *orphan* message *relative* to 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) $\notin E|_{S_2}$. m_4 is called an *in-transit* message relative to S_2. In this 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, R is called a *distributed snapshot* if every in-transit mes-

sage in E relative to R is logged and can be retransmitted for further recovery from R.

Messages generated by the application are called *data* (or *application*) messages, and the messages generated by the distributed snapshot protocol are called *control* messages.

Definition 2.4: The *checkpoint overhead*, denoted by o_p, is the increase in the execution time of a process p because of a single checkpoint. Consequently, the *distributed snapshot overhead*, denoted by O, is the maximum value of o_p for all the processes. Namely, $O = \max_{1 \leq p \leq n}\{o_p\}$.

Definition 2.5: The *checkpoint latency*, denoted by l_p, is the duration required to take a single checkpoint in a process p. Consequently, the *distributed snapshot latency*, denoted by L, is the maximum value of l_p for all the processes. Namely, $L = \max_{1 \leq p \leq n}\{l_p\}$.

3. Existing Distributed Snapshot Protocols

3.1. The Sync-and-Stop (SaS) Protocol

The SaS protocol produces distributed snapshots. One process, called the *coordinator*, invokes a barrier synchronization among the processes to take a distributed snapshot. The protocol works as follows. The coordinator broadcasts a special message, called INIT, to the other processes. Upon receiving INIT, each process stops running its target program and sends the READY message to the coordinator after making sure that its sent data messages have been received. When the coordinator collects the READY messages, it broadcasts the DO message and takes a local checkpoint. Upon receiving the DO message, each process takes a checkpoint and sends back the `DONE` message to the coordinator. When the coordinator collects the DONE messages from all the processes, it broadcasts the COMMIT message to resume the application.

The main advantage of the SaS protocol is that it does not assume FIFO in the network channels. Although there are many systems and environments that do not support FIFO in the network channels, there are few distributed snapshot protocols that do not assume FIFO. For example, as was mentioned in [4], in the wireless mobile environment, FIFO does not necessarily hold. Moreover, many systems and applications in grid environment may not assume FIFO for efficiency and performance issues. On the other hand, the SaS protocol has some significant disadvantages. The first disadvantage is that it suspends the application execution during a distributed snapshot. Such suspension could be intolerable for some applications, especially in high-performance applications. In addition, as in the wireless mobile environment, many applications cannot even suspend the execution at all. The second disadvantage is the requirement that each process should be able to verify if every data message sent before receiving the INIT message is received or not. Although there are many systems that can support this verification [16], there are many systems that cannot (as it is very costly). For example, in systems that work on radio broadcast transmission, a process cannot verify that a message it sends is received. In addition, it could be very costly and difficult to verify the message arrival by the sender in the wireless mobile environment [11]. Similarly, in the Grid and WAN environments, this property is not cheap to implement.

We believe that the improvement presented by Elnozahy et al. [10] gets rid of the main disadvantages of SaS. By this improvement, each distributed snapshot is identified by a monotonically increasing Consistent Checkpoint Number (CCN). Every application message is tagged with the CCN of its sender, enabling the new protocol to run in the presence of message re-ordering or loss.

3.2. The Chandy-Lamport (CL) Protocol

The CL protocol is considered a *coordinated checkpointing* protocol in which processes need to coordinate with each other to form a snapshot. Unlike the SaS protocol, the CL protocol does not suspend the application execution while taking a snapshot. The *coordinator* process in CL (which could be any process) initiates a distributed snapshot by broadcasting *marker* messages and then taking local checkpoints. When process p receives the marker from channel c and it has not taken a checkpoint yet, it broadcasts the marker, takes a checkpoint, and records the state of c as being empty. Otherwise, p records the state of the channel c as the sequence of messages received along c after it has taken a checkpoint and before it has received the marker.

We describe here the CL protocol in more detail using a pseudo code. We start by defining two data structures to help the reader understand the protocol's description: (1) `MARKER` - this is the marker message used to coordinate a distributed snapshot and (2) S_p - A n-vector of integers. An entry $S_p[u]$ indicates the state of process u as known by process p. We consider two different states for every process. The states are *Normal* and *Saving*. Intuitively, a process is in the *Saving* state during the creation of the distributed snapshots and in the *Normal* state otherwise. In the CL protocol, we say that process p is in the Saving state between the time it receives the first `MARKER` message until it receives the `MARKER` back message from all the processes.

Figure 2 presents the pseudo code of the CL protocol. We describe the CL protocol by observing the the behavior of process p upon receiving a message from another process q. The message could be either the marker or data.

```
CL_recv(msg, q)
1: recv(msg, q)
2: if (msg.Type == MARKER)
3:    if (S_p[p] == Normal) {p didn't see the MARKER}
4:       ∀u, 1 ≤ u ≤ N,
             send(MARKER, u); S_p[u] = Saving; checkpoint()
5:    S_p[q] = Normal {receiving the MARKER back}
6:    if (∀u, u ≠ p, S_p[u] == Normal), S_p[p] = Normal
7: else {This is a data message}
8:       if ((S_p[p] == Saving) and (S_p[q] == Saving)), Log(msg)
```

Figure 2. p's behavior according to CL

As presented in Figure 2, in the CL protocol, when a process p receives the MARKER message, it switches to the Saving state, takes a local checkpoint, and forwards the MARKER message to its neighbors. Notice here that p identifies and logs every in-transit message. p identifies an in-transit message as an incoming message from a process q such that p has sent the marker to q, but has not received it back yet. Since we assume FIFO, such a message should be an in-transit message.

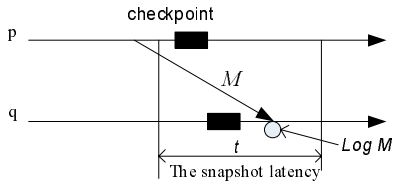
The main advantage of the CL protocol is the ability to take distributed snapshots "on-the-fly" without piggybacking control information onto data messages. In fact, this is a very important property for high-performance systems and many other systems. The price of not suspending the application execution in CL, however, is the obligation to identify and log the in-transit messages. Of course, in-transit messages can be logged in the internal memory (which is a persistent memory) during the snapshot. Then, they should be moved to secondary storage at the end of a distributed snapshot. Therefore, secondary storage will be accessed several times for checkpointing and logs. Moreover, in systems with limited internal memory, logs should be directed to secondary storage. Obviously, the log events could contribute significantly to the parameters O and L in the CL protocol. In some applications, especially those with a high rate of communication and big messages, the overhead of logging in-transit messages could be unacceptable.

4. The Modified Chandy-Lamport Protocol

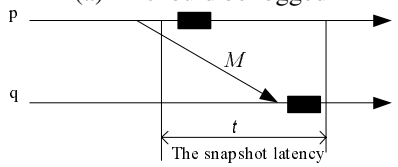
In sections 3.1 and 3.2 we presented the main disadvantages of SaS and CL protocols, respectively. Although there are already some works to cope with these disadvantages, yet we don't have efficient distributed snapshot protocols that cope with all of these advantages. The open question remains if there is an optimal protocol that can be obtained at all. Our direction here, therefore, is to improve the existing protocols to cope with some of these disadvantages and try to converge to optimal protocol.

In Section 3.2, we identified that the main disadvantage of the CL protocol, that have not addressed yet, is the possible large number of accesses to a persistent memory (e.g. secondary storage) in order to log any in-transit message. Actually, accessing secondary storage is one of the main contributions to the checkpointing overhead [2, 17]. Regardless of where a message is logged, to secondary storage or the main memory, the large number of logs just increase the overhead dramatically. We present here a modified version of the CL protocol, called the Modified Chandy-Lamport (MCL) protocol, where we try to reduce the number of in-transit messages. In Section 5, we show that this reduction could be as big as 90% in some applications.

Suppose that M is a data message from process p to another process q. The reason that M may be considered as an in-transit message relative to a distributed snapshot S is that q takes a checkpoint (belonging to S) before it receives M, but p has sent M before taking its checkpoint. Figure 3(a) shows how M could be an in-transit message relative to S. We assume here that the latency of S is t. Therefore, if we succeed in delaying the checkpoint of q after receiving M, M will not be an in-transit message and the latency of S remains t. As presented in Figure 3(b), the challenge here is to delay the checkpoints in each process as long as possible without increasing O (the overhead) and L (the latency).



(a) M should be logged



(b) M is not logged

Figure 3. A scenario where we can avoid a log

Indeed, in our new MCL protocol, we have the same value of L, but fortunately with less overhead (O). Figure 4 shows the pseudo code of a process behavior upon receiving a message according to the new MCL protocol. As presented in Section 3.2, we use the same data structures used in describing the CL protocol for describing the MCL protocol. In the CL protocol, a process can be in one of two states: Normal or Saving. In the MCL protocol, however, we define one more state, called *Ready*. This new state indicates that a process is ready to take a checkpoint after receiving the marker, but did not take it yet.

```
MCL_recv(M, q)
1: recv(M, q)
2: if (M.Type == MARKER)
3:   S_p[q] = Saving  /* Assume that q's state is Saving */
4:   if (S_p[p] == Normal)  /* p didn't see the MARKER yet */
5:     ∀u, 1 ≤ u ≤ N and u ≠ p, send(MARKER, u)
6:     ∀u, 1 ≤ u ≤ N and u ≠ q, S_p[u] = Ready
7:   else  /* p is not in the Normal state */
8:       if (∀u, u ≠ p, S_p[u] == Saving)
9:         if (S_p[p] == Ready), checkpoint()
10:        ∀u, 1 ≤ u ≤ N, S_p[u] = Normal  /* Terminate the snapshot */
11:else  /* The message is a data message */
12:    if ((S_p[p] == Ready) and (S_p[q] == Saving))
13:      S_p[p] = Saving; checkpoint()  /* Take a checkpoint before delivering M */
14:    if ((S_p[p] == Saving) and (S_p[q] == Ready)), Log(M)
```

Figure 4. The behavior of process p according to the MCL protocol

In MCL, once process p receives (or initiates) the marker, it switches from the Normal state to the Ready state. Then, p broadcasts the marker to all the other processes and assumes that they are in the Ready state. As presented in Line 3 of Figure 4, when process p receives the marker from process q, p recodes q's state as Saving. This is because p does not know if q is still in the Ready state or not. Process p switches to the Normal state after receiving the marker from all the other processes (from Line 8 to Line 10 of Figure 4). Of course, if p has not taken the checkpoint yet, it does so.

As presented in Line 12 of Figure 4, if process p is in the Ready state and receives a message from q which is in the Saving state, then p switches to the Saving state. Again, p does not know if q has taken the checkpoint or not, so it takes the checkpoint before delivering M to ensure that M will not be an orphan message. Similarly, if p is in the Ready state and wants to send a message M to process q such that q is in the Ready state (according to p's knowledge), then p switches to the Saving state before sending M. This is because p does know when exactly q switches to the Saving state. Note that the only place where process p logs a data message M, that is sent from process q, is when p is in the Saving state and q is in the Ready state (Line 14 of Figure 4).

4.1. Protocol Properties

Now we prove the safety and the progress of the MCL protocol. For safety, we show that the protocol produces distributed snapshots. Showing the progress of the protocol is straightforward. Note that every process broadcasts the marker right after receiving it. Therefore, the marker message is propagated to all the processes as fast as possible. Furthermore, once a process p receives the marker from all the other processes, it terminates the current distributed snapshot.

Lemma 4.1: For any two processes p and q and for an integer i, if p sends a message M to q after sending the marker, then q will take the checkpoint $C_{q,i}$ before delivering M.

Proof: By the MCL protocol, every checkpoint $C_{p,i} \in S$ is created only after p has seen the marker (p switches to the Saving state only from the Ready state). We assume that p has already seen the marker and sends it to all the other processes. By the MCL protocol, if p is in the Ready state, regardless of q's state and before p sends M, p takes $C_{p,i}$ and switches to the Saving state. Due to FIFO, q should receive the marker before receiving M. Then, once q receives M, it takes a checkpoint before delivering the message (Line 12 of Figure 4). Therefore, by the MCL protocol, M cannot be an orphan message. □

Lemma 4.2: Every in-transit message is logged by the MCL protocol.

Proof: By the MCL protocol, message M between p and q is an in-transit message only if p sends M before receiving the marker (p is in the Normal state) and q receives M after takeing the checkpoint (q is in the Saving state).

By Line 2 (in Figure 4), once process q receives the marker at the first time, it broadcasts the marker and determines that the state of all the other processes (apart from the process who sent the marker) are in the Ready state. Therefore, by Line 14 (in Figure 4), if process q receives message M after being in the Saving state and before receiving the marker from process p, process q logs the message M. Notice here that due to FIFO, q knows that M is in-transit message, therefore, the MCL protocol logs every in-transit message. □

Theorem 4.3: The MCL protocol produces distributed snapshots.

Proof: By the discussion on the progress of the protocol, we showed that for every marker M, each process takes a local checkpoint. By Lemma 4.1, there are no orphan messages relative to any cut of checkpoints that are created due to M. In addition, by Lemma 4.2, every in-transit message relative to the cut is logged. Therefore, this cut is a distributed snapshot. □

5. Performance Analysis

In this section we present a model-based performance analysis to show the amount of overhead that is saved due to the new protocol for distributed snapshot comparing with the old protocols. The plan of performance analysis is as follows: first, we select a specific message-passing program; then we define different distributed executions of the program with different distributed snapshot protocols; next, we build a stochastic model for each execution; and lastly, we simulate those executions and present the results.

The message-passing program that we consider here is a parallel version of the Jacobi method for solving a system of linear equations. The program has one main loop. In every iteration of the loop, a process performs three steps: sends messages to the left and right neighbors, receives messages from the left and right neighbors, and performs a local computation. Please note that the left and right neighbors do not exist for P_1 and P_n, respectively.

Using Stochastic Activity Networks (SANs) [18], we have modeled four different executions of the Jacobi program. The first execution is just the Jacobi program without any distributed snapshot protocol (denoted by Basic). The second execution is the Jacobi program with the SaS protocol (denoted by SaS). The third execution is the Jacobi program with the CL protocol (denoted by CL). The forth execution is the Jacobi program with the MCL protocol (denoted by MCL).

SANs are a convenient and high-level language for capturing the stochastic behavior of a system. Möbius [8] is a tool dedicated to creating and simulating (or solving) SAN-based models. A SAN has the following components: *places* (denoted by circles), which contain tokens; *tokens*, which indicate the "value" or "state" of a place; *activities* (denoted by vertical ovals), which change the number of tokens in places; *input arcs*, which connect places to transitions; *output arcs*, which connect transitions to places; *input gates* (denoted by triangles pointing left), which are used to define complex enabling predicates and completion functions; *output gates* (denoted by triangles pointing to the right), which are used to define complex completion functions; and *instantaneous activities* (denoted by vertical lines), which are used to specify zero-timed events. An activity is enabled if for every connected input gate, the enabling predicate contained in it is true, and for each input arc, there is at least one token in the connected place. When an activity completes, one token is added to each place connected by an output arc, and functions contained in connected output gates and input gates are executed. The output gate and input gate functions are usually expressed using pseudo-C code. The times between enabling and firing of activities can be distributed according to a variety of probability distributions, and the parameters of the distribution can be a function of the state.

In the following sections we present the SAN models of all the executions apart from the SaS execution. Although we consider SaS in our results for the comparing reason, our new MCL protocol improves CL but not SaS.

5.1. The Basic Execution

The left side of Figure 5 shows the composed model for Basic. The model consists only of one atomic SAN submodel, which is **basicReplica**. **basicReplica** is *replicated* NUM_PROC times to form a message-passing application with NUM_PROC processes. The right side of Figure 5 shows the SAN representation of the **basicReplica** submodel. The SAN model models the behavior of a single process, including assignment of a process identifier, execution of the main loop of Jacobi, and failures in the process. The activity *setID_act* fires at the beginning of each process execution to set an ID to the place *myRank*. The shared place *numProc* is initialized with NUM_PROC, and every time a new process is started, *numProc* is reduced by one until it becomes zero.

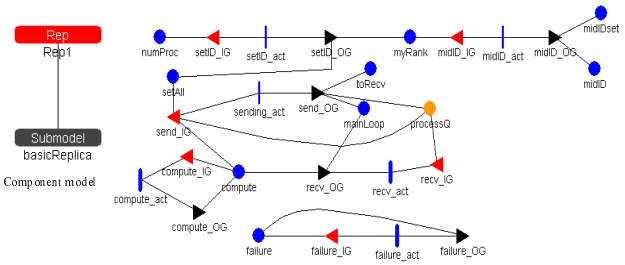


Figure 5. The SAN model for Basic

A process starts the main loop only after all the other processes are ready. The shared place *setAll* indicates that all the processes have assigned their IDs. The local place *mainLoop* traces the number of iterations of the main loop in the Jacobi program. This place is initialized to 0 and incremented by one for each completed iteration. A process starts a new iteration by sending messages to its left and

right neighbors using the *send_act* activity. Then, it waits to receive messages from those neighbors using the *recv_act* activity. The place *processQ* is a shared array. Each index i in this array represents process i, and the entry $processQ[i]$ represents the number of queued messages for process i. The activity *compute_act* represents the local computation by a process. When a failure occurs, the activity *failure_act* is fired and puts a token in the local place *failure*. Since there is no checkpoint/restart in this model, after a failure occurs, the simulation of Basic stops immediately. The activity *failure_act* fires using an exponential distribution with a rate that differs for different studies.

5.2. The CL Execution

Figure 6 shows the SAN representation of **basicReplica** of the CL execution. The composed model looks exactly like the model in Figure 5. The SAN model in Figure 6 models the behavior of a single process. It assigns an identifier for each process and then executes the main loop of the Jacobi program. The execution starts with assigning an identifier for each process and execution of the main loop of Jacobi. Each process sets its identifier and executes the main loop exactly as described in Section 5.1, but with consideration of the CL checkpointing protocol. The activity *chkptInit_act* initiates a distributed snapshot in a particular process. This activity fires periodically using a deterministic time distribution function. Once the activity has fired, the activity *broadcastMarker_act* fires in the same process to broadcast the marker. Then, the activity *chkpt_act* indicates the actual local checkpoint. On the other hand, the activity *recvMarker_act* is fired in each process to receive the marker. The places *L_Sp* and *G_Sp* represent the S_p data structure. After a failure occurs, as indicated by place *failure*, the activity *recovery_act* fires after some delay indicating the fault detection delay. *recovery_act* recovers the faulty process by rolling back its execution to the latest checkpoint. Unlike the Basic execution, if a failure occurs, a recovery starts after the failure detection.

5.3. The MCL Execution

The SAN model of the MCL execution is similar to the SAN model of the CL execution presented in Figure 6. The main difference is the actions taken in the input gate *chkpt_IG*, when a new token has arrived. By MCL, upon a new token, checkpoint is not enabled as in CL, but the process enters the Ready state instead. Checkpoints are enabled only when the process receives back all the tokens from its children, or receives a message from another process which is in the Saving state. These conditions are checked in the output gates *recvMarker_OG* and *recv_OG*.

5.4. Results

We used the Möbius tool [8] to design the SANs, define performance, measure distributed snapshots, design studies on the models, simulate the models, and obtain values for the measures defined on various studies. Mainly, the Jacobi program runs the main loop that includes communication with neighbors and computation. To measure the overhead of the four different models, we defined a performance measure that counts the completed iterations of the loop. In this simulation, we set the execution time to 1000 seconds and counted the completed iterations within this time. In addition, we set the following parameters: NP: this parameter indicates the number of running processes, CI: this parameter indicates the periodic time of initializing a new snapshot in the execution (*checkpoint interval*), CO: this parameter indicates the latency of taking a local checkpoint (the parameter is assigned to 2 seconds), LD: this parameter indicates the time of a log event to complete (this overhead is used by the SAN activity *compute_act*), TR: this parameter indicates the network delay, DR: this parameter indicates the rate at which a failure is detected (we assigned the value to 10), FR: indicates the failure rate that occurs in every process (this rate is used by the **failure_act** activity with an exponential distributed function), and RO: indicates the time of recovery (we assigned this parameter as 3.5 seconds).

For every execution, we ran the simulation 1000 times and the results presented here had a 95% confidence interval. Mainly, in these studies, we observed the number of completed iterations. For every execution we recorded the *average number of iterations*, denoted by ANI, which is the sum of the completed iterations in every process divided by the number of processes. In addition, for the executions that may have message logging, we recorded the total number of log events.

Figure 7 shows ANI versus different number of processes (NP) in all the four executions. First of all, as we see in Figure 7(a), the MCL execution achieves almost the same ANI values as the Basic execution in the failure-free case. In this figure, we can see that while SaS and CL have 6% and 4% more ANI than Basic, respectively, MCL has only 2% ANI over the Basic execution. On the other hand, as presented in Figure 7(b), MCL achieves the maximum ANI with $FR = 0.0001$. This figure shows that our new protocol MCL is better than the CL protocol. In addition, MCL continues to gain the maximum ANI number with higher failure rate.

Figure 8(a) shows ANI versus different values of failure rate (FR). ANI in all the executions drops linearly with a linear increase of the failure rate. The number of processes is important here since we assume a failure rate of Poisson distribution. As presented in Figure 8(a), we can see that

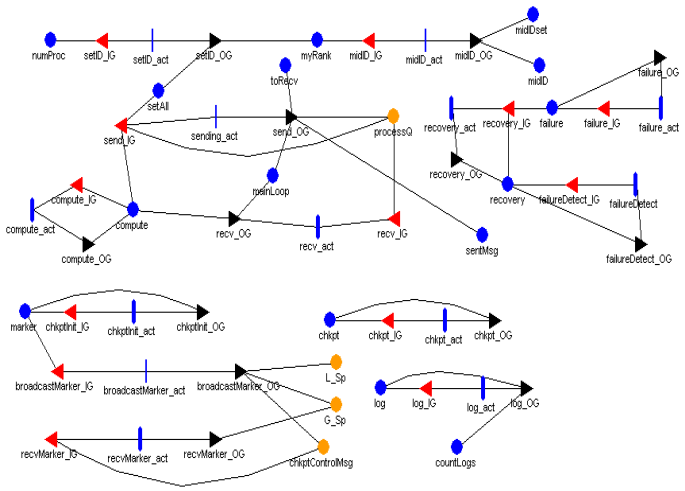
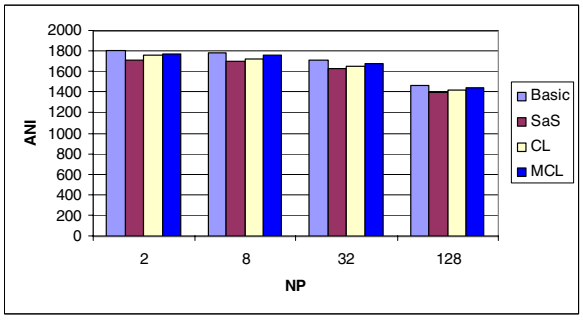
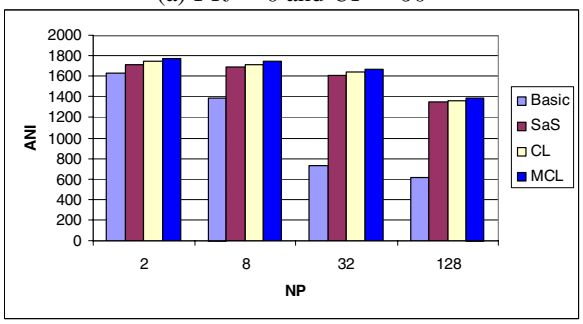


Figure 6. The SAN model for the CL execution



(a) $FR = 0$ and $CI = 50$

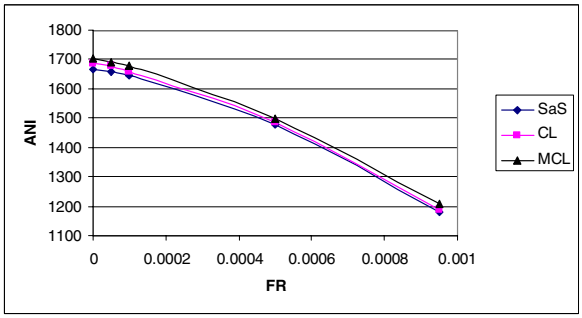


(b) $FR = 0.0001$ and $CI = 50$

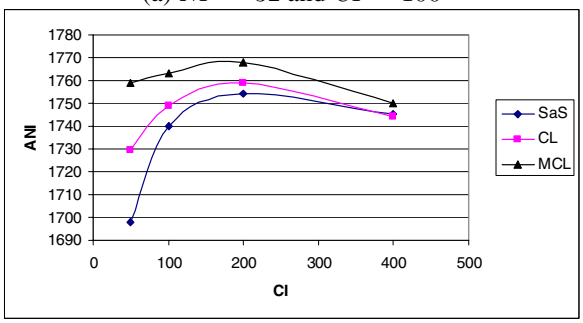
Figure 7. ANI of all the executions VS. NP

the MCL execution achieves the largest ANI.

Figure 8(b) presents ANI of the executions versus different values of CI. First of all, we can see that the MCL execution achieve greater ANI than SaS and CL. In addition, as was determined in [2], we can see that the CL protocol achieves more ANI than the SaS protocol in any value of CI. Figure 8(b) shows that in case of failures, each execution has a different optimum CI value in which the execution achieves its maximum ANI. As was shown by [2], this claim is true for any execution with checkpoint events.



(a) $NP = 32$ and $CI = 100$

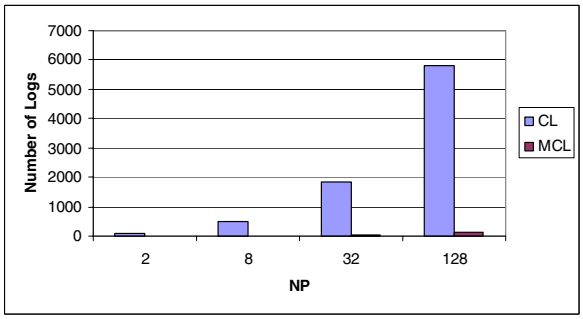


(b) $NP = 8$ and $FR = 0.0001$

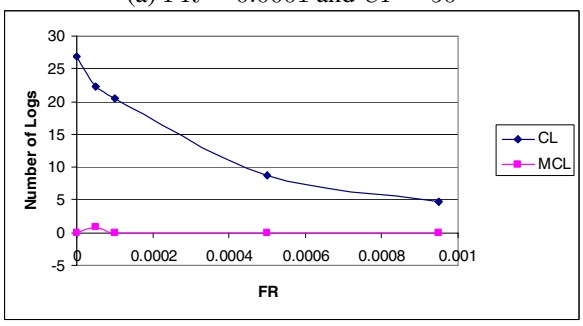
Figure 8. ANI vs. FR and checkpoint intervals

Figure 9(a) and Figure 9(b) shows how many log events CL and MCL have, versus NP and FR, respectively. There is no doubt that the MCL protocol reduces the number of log events, but what we see here is a dramatic reduction (more than 95%). Although I simulated a message-passing program in which a process only communicates with its neigh-

bors, the CL protocol had a large number of log events. So, the question here is how many log events does the CL protocol have in applications in which every process communicates with all the others?



(a) $FR = 0.0001$ and $CI = 50$



(b) $NP = 8$ and $CI = 50$

Figure 9. The number of logs vs. NP and FR

6. Related Work

A great deal of work has been done on distributed snapshots, for example [6, 12]. Those papers presented summaries of different checkpointing protocols. In addition, a considerable body of work is available on C/R for traditional distributed systems, e.g., [9, 16]. As pointed out in [9], the C/R protocols for traditional distributed systems are classified into three classes: coordinated checkpointing, uncoordinated checkpointing, and communication-induced checkpointing. For the C/R protocols for distributed systems, this major classification still holds; most of the protocols are of the communication-induced checkpointing type. Those papers presented summaries of different checkpointing protocols.

After extensive analysis work on the distributed C/R protocols and systems, it has been discovered that the main parameters that contribute to the C/R overhead are control messages, rollback propagation, and induced checkpoints [2, 5]. Therefore, it was concluded in [2], that the distributed snapshot protocols (which belong to the coordinated checkpointing class) impose less overhead than many of other C/R protocols. This is simply because distributed snapshot protocols do not have induced checkpointing and the rollback propagation is the shortest. However, as mentioned above, the distributed snapshot protocols (SaS and CL) have many restrictions that limit their use in different distributed systems.

Some work has been done to improve the SaS and CL protocols. Plank [16] has introduced a new distributed snapshot protocol, called the *Network Sweeping* (NS) protocol. This protocol is an improvement on the CL with less control messages. In fully connected processes, the CL protocol has $O(n^2)$ control messages (markers) in every snapshot. Using the property of wormhole routing to "sweep" the fully connected systems, the NS protocol has only $O(l(n))$ control messages, where $l(n)$ is the number of physical communication links in the system. Moreover, Vaidya [20] has improved the CL protocol by ensuring that the processes do not take the checkpoints at the same time in order to avoid any extra delay due to concurrent access to the same secondary storage. Notice here that these two improvements on the CL are different from the MCL improvement. The MCL protocol reduces the access to the secondary storage due to message logging by more than 95%.

Agbaria and Sanders [4] have presented a new version of the CL protocol that runs on the mobile environment without the FIFO assumptions. This protocol, however, behaves exactly like CL and has the same number of message logging as CL. In this paper, in addition to reducing the number message logging by MCL, I introduced the SaC protocol that can run on a mobile environment (does not require FIFO) and has a negligible amount of message logging.

7. Conclusions

In a previous work [2], it was shown that the distributed snapshot protocols are the most efficient protocols in distributed systems among a set of known checkpointing protocols for distributed systems. With the belief that distributed snapshot protocols are very important in different distributed systems (not only the traditional distributed systems) and concentrating on the improvement of the distributed snapshots, we continue to seek improvements for distributed snapshot protocols. In this paper, we came up with new distributed snapshot protocol, called MCL, that have better performance than the previous protocols and can be applied in more distributed environments.

To the best of our knowledge, the new MCL protocol is the first protocol that reduces the number of accesses to the secondary storage due to message logging. The MCL protocol is efficient and its improvement could be very crucial for many systems such as high-performance and limited-resources systems. In addition to presenting this new protocol, we present proofs of correctness and progress of the

protocols.

To show the effectiveness and the low overhead of the new protocol, we compared the new protocol with well-known distributed snapshot protocols (SaS and CL). Using stochastic models, we were able to model different distributed executions, each with a different distributed snapshot protocol. We simulated those different models to obtain the expected results that show the low overhead of the new protocol as compared to the other protocols.

References

[1] Condor Project Homepage. http://www.cs.wisc.edu/condor/.

[2] A. Agbaria, A. Freund, and R. Friedman. Evaluating Distributed Checkpointing Protocols. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 266–273, Providence, Rhode Island, May 2003.

[3] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 6(3):227–236, July 2003.

[4] A. Agbaria and W. H. Sanders. Distributed Snapshots for Mobile Computing Systems. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, pages 177–186, Orlando, Florida, March 2004.

[5] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An Analysis of Communication Induced Checkpointing. In *Proceedings of the 29th Fault-Tolerant Computing Symposium*, pages 242–249, Madison, Wisconsin, June 1999.

[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] C. Critchlow and K. Taylor. The inhibition spectrum and achievement of causal consistency. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 31–42, 1990.

[8] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, October 2002.

[9] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[10] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[11] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Comput.*, 27(4):38–47, April 1994.

[12] L. He and Y. Sun. On Distributed Snapshot Algorithms. In *Proceedings of the Advances in Parallel and Distributed Computing*, pages 291–297, March 1997.

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

[14] N. Malpani, J. L. Welch, and N. H. Vaidya. Leader Election Algorithms for Mobile Ad Hoc Networks. In *Fourth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Boston, MI, August 2000.

[15] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[16] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Department of Computer Science, Princeton University, January 1993.

[17] J. S. Plank and M. G. Thomason. Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems. *Journal of Parallel and Distributed Computing*, 61(11):1570–1590, November 2001.

[18] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In *Lectures on Formal Methods and Performance Analysis*, LNCS 2090 (E. Brinksma, H. Hermanns, and J. P. Katoen), pages 315–343. Springer-Verlag, Berlin, 2001.

[19] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *SC'04 Conference CD*, Pittsburgh, PA, November 2004.

[20] N. H. Vaidya. On Staggered Checkpointing. In *Proceeding of the 8th IEEE Symposium on Parallel and Distributed Processing*, New Orleans, LA, October 1996.