RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols

Nuno Neves

Coordinated Science Laboratory
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801

W. Kent Fuchs

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907

Abstract

This paper describes the design, implementation, and evaluation of a run-time system for clusters of workstations that allows the rapid testing of checkpoint protocols with standard benchmarks. To achieve this goal, RENEW provides a flexible set of operations that facilitates the integration of a protocol in the system with reduced programming effort. To support a broad range of applications, RENEW exports, as its external interface, the industry endorsed Message Passing Interface (MPI). Three distinct classes of protocols were evaluated using the RENEW environment with SPEC and NAS benchmarks on a network of workstations connected by ATM. It was observed that the communication-induced protocol emulated the behavior of the coordinated protocol, with comparable performance. The message logging protocol degraded the performance. Even though the message logging protocol was slower due to log replay, all three protocols required a similar amount of time to restore the application to the same state as before failure occurred and recovery was initiated.

1 Introduction

During the past 20 years a large number of checkpointing and roll back recovery protocols have been proposed for distributed systems [1]. Most of these protocols, however, were never implemented or tested. System and application code complexity have made the implementation and evaluation of alternate protocols particularly difficult. Efficient implementation of recovery protocols has traditionally required intimate knowledge of low level system details such as communication channels, file system context, timers, and memory organization. This complexity has resulted in only a few protocol implementations being evaluated with relatively simple application programs.

Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract DABT 63-96-C-0069, and in part by the Office of Naval Research under contract N00014-97-1-1013.

This paper describes the design and implementation of RENEW - *REcoverable NEtwork of Workstations*, a run-time system that facilitates the development and testing of checkpoint protocols for parallel computing in clusters of workstations. RENEW has a flexible set of operations that provide for a protocol to be integrated into the system with a reasonable programming effort. The result is a high level performance that is comparable to a system specific implementation, without requiring the knowledge of the intrinsics of RENEW. The operations provide for a protocol to accomplish checkpointing and recovery tasks such as message tagging and logging, storage of data and checkpoints in the local disk or remote servers, and process restart and message replay. The application interface of RENEW is the industry endorsed MPI - *Message Passing Interface* [2]. Applications conformable with MPI can either be developed in RENEW or they can be run without modification.

Three different types of checkpoint protocols were implemented and tested in RENEW for this work: a coordinated protocol designed for mobile environments [3]; a recently proposed communication-induced protocol [4]; and an optimistic sender-based message logging protocol [5]. The applications utilized in the experiments were SPEC and NAS benchmarks and a large climate modeling code. It was observed that recently introduced communication-induced protocols behave in a manner that is very similar to coordinated protocols, but with the cost of loosing most of their flexibility. The message logging protocol performed worst with impact not only on the execution time but also on the amount of disk space that has to be allocated. The recovery results show that all the protocols take roughly the same time to restore an application to the prefailure state, even though the message logging protocol has perform numerous additional tasks.

2 Related Work

Checkpointing and roll back recovery techniques have been proposed for a wide range of applications, including shared-memory systems [6, 7], distributed debugging [8], and mobile computing [3, 9]. Even with this diversity, most checkpoint protocols for distributed systems can be divided into a small number of classes [1, 10]: uncoordinated [11], coordinated [12–15], communication-induced [4, 16] and message logging [5, 17, 18].

OTEC is an object-oriented simulator designed for the analysis of checkpointing and recovery techniques [19]. It uses DEPEND [20] and SIMPAR [21] for dependability and reliability evaluation in multicomputer systems. OTEC has a set of predefined classes for checkpointing, error detection and recovery, which can then be reused and composed to build new checkpoint protocols. The evaluation of the protocols can be done by varying parameters like checkpoint size and message rates.

Fail-safe PVM [22] and MIST [23] are enhanced versions of PVM capable of restarting distributed applications from failures. In both cases, the Chandy and Lamport [12] checkpoint protocol was implemented in the PVM run-time system to allow transparent recovery. During the checkpoint creation, processes are first stopped, then the communication channels are flushed from messages, and finally the checkpoints are independently saved. CoCheck [24] provides migration and checkpointing of parallel applications on a MPI environment. The solution adopted for recovery is similar to MIST.

RENEW has attributes of both a run-time system like MIST and a simulator like OTEC. It simplifies the development of parallel applications in distributed systems, with support for transparent recovery. It also provides a framework where checkpoint protocols can be designed and analyzed. Since RENEW is entirely implemented at user-level, it can be ported to any Unix-based system and tested in any network with TCP/IP[1]. This characteristic together with the support for standard benchmarks makes RENEW an environment where checkpoint protocols can be evaluated under realistic conditions.

Some checkpointing protocols have been implemented and evaluated in distributed systems [14, 15, 18, 25–27]. Most implementations have been done in specialized kernels or hardware systems making the accurate comparison between the protocols difficult. RENEW solves this problem since checkpointing and recovery schemes are analyzed in a common general purpose system. Section 5 compares our results with those reported by other authors.

3 RENEW

3.1 Overview

RENEW is a run-time system for clusters of workstations that supports the execution of message-passing parallel applications (Figure 1). The system is divided in two parts: a library that is linked with the application, and a set of checkpoint servers. Most of the functionality of RENEW is provided by the library; it spawns processes in remote machines, guarantees message deliveries, and recovers processes from failures. The server responsibility is to store and retrieve information from a remote disk.

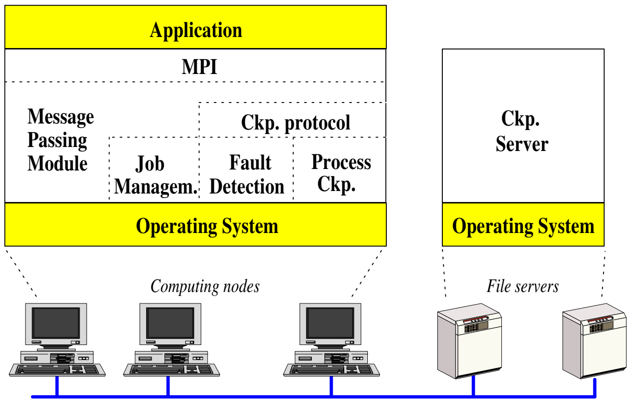

Figure 1: Architecture of RENEW.

The library is composed of a set of modules with well defined interfaces that cooperate to supply the services required by the applications and the checkpoint protocols. The MPI module is responsible for the external interface. It implements the various constructs necessary for the MPI specification (e.g., groups and data types) and does some initial processing on the application's messages before giving them to the message passing module. The job management module spawns processes in the computing nodes both when the application starts and during recovery. Process checkpointing and information storage are the responsibility of the checkpoint module. This module can either save the data in the local disk or in a remote machine using the checkpoint servers. The primary focus of RENEW is on checkpointing and recovery, however, basic fault detection and injection modules are also provided. The fault detection module locates process failures and initiates the recovery. The current implementation is based on watchdogs and uses the errors from the sockets to enhance fault location [28]. RENEW currently only provides minimal support for fault injection.

When a new request arrives, the checkpoint server starts a new process that handles all the communication with the checkpoint module. There are three main reasons why the servers are used in RENEW; First, they let the checkpoint module store data in remote nodes without requiring the disks to be exported with a network file system. Second, the load on the servers can be spread across multiple machines. Third, the specialized checkpoint server has significantly better performance than a network file system like NFS.

[1] The current version of RENEW has been ported to HP workstations with HP UX, Sun workstations running SunOS and Solaris, and PCs with Linux. RENEW has also been tested with 10 and 100 Mbit/s Ethernets and 155 Mbit/s ATM.

Table 1: Operations of the Checkpoint Interface.

Initialization and Ending:
[up-call] void `renew_initCkpProt`(int id, int n_procs)
[up-call] void `renew_endCkpProt`(void)
Message Tagging and Logging:
`#define CKP_INFO_SIZE` *size*
[up-call] void `renew_tagMesgR`(int dest, char *ckp_buf)
[up-call] void `renew_sentMesgR`(int dest, char *head, int h_size, char *msg, int m_size)
[up-call] void `renew_recvMesgR`(int source, char *ckp_buf, char *head, int h_size, char *msg, int m_size)
Process Checkpoint:
[down-call] int `renew_createCkp`(char *name, int n, exclHeap *exc)
Roll back and Log Replay:
[up-call] void `renew_processFailure`(int *ids, char **ckps)
[up-call] int `renew_replayMesg`(int source, char *head, int h_size, char *msg, int m_size)

3.2 Checkpoint Interface Specification

The current checkpoint interface takes into consideration the different requirements of the four basic classes of protocols: uncoordinated, coordinated, communication-induced, and message logging. It exports several groups of operations: message tagging and logging, checkpoint creation and data storage, roll back and log replay, message passing, and timers. Moreover, it supports the most common assumptions that are made about the communication system; reliable and unreliable channels. The operations are divided in *up-calls* and *down-calls*. Up-calls are operations from the checkpoint protocol that are invoked by the RENEW modules. Down-calls are operations belonging to RENEW. Up-calls not necessary to the implementation of the protocol can be defined as empty macros, to ensure that they are removed when RENEW is compiled.

Initialization and Ending

RENEW calls `renew_initCkpProt` when the initialization of the various modules is completed (see Table 1). The protocol can use this function to start the checkpoint timers and to initialize data structures. The function arguments are the total number of processes that are executing the application, n_procs, and a process identifier, id, with a value ranging between 0 and $n\_procs - 1$. RENEW assigns virtual identifiers to processes to make the physical location transparent to the protocol. During recovery, the process can be restarted on a different node without consequences to the protocol. The operation `renew_endCkpProt` is invoked when the application completes execution.

Message Tagging and Logging

A message is composed of a fixed sized header and data. The header contains a few fields (e.g., group identifier) that let the MPI module associate the sends with the corresponding receives. The data can be of any size, including 0 bytes. Most checkpoint protocols only add to the application messages fixed sized amounts of data. For this reason, it was decided to allocate a region in the header to hold the checkpoint tag, with a size specified by the `CKP_INFO_SIZE` macro. Since this number of bytes is sent on every message, its value should be carefully chosen. Otherwise, network bandwidth is wasted and performance is affected.

RENEW calls `renew_tagMesgR` before sending a message. Its arguments are the destination of the message, $dest$, and a pointer to the buffer where the checkpoint data should be added, ckp_buf. The message can be copied to a log, by a sender-based message logging protocol, when `renew_sentMesgR` is invoked. This operation is only executed when the message has been sent, and it has as arguments the header, $head$, and the message contents, msg. Performance is improved by separating the tagging from the logging since the copy is removed from the transmission critical path. This optimization, however, can not be done on the receive side since the copy has to be performed after the message arrival and before the delivery to the application. Operation `renew_recvMesgR` is called to allow the inspection of the tag and the logging at the receiver.

A protocol that uses the previous three operations sees a reliable FIFO ordered flow of messages. RENEW also provides an equivalent set of functions for protocols that assume unreliable communication channels. The main difference between the two sets is that the *unreliable* operations are also called when acknowledgments are sent or received. Furthermore, they may be invoked more than once for the same message, since there can be re-transmissions or duplicates. The unreliable functions have two extra arguments that are used to optimize the implementation of message logging protocols. The first is a sequence number that lets the protocol determine if the message has or has not already been logged. The second argument indicates if the message is from the application or an acknowledgment.

Process Checkpoint

The operation `renew_createCkp` lets the protocol create process checkpoints. The first argument, $name$, specifies the name of the checkpoint file. The file can be stored in the local disk or in a remote server. The choice is implemented at compile time. The other two arguments can be used to exclude memory regions from the process checkpoint. If n is set to zero, all memory of the process is saved in the checkpoint. `renew_createCkp` returns three kinds of values: > 0 if the checkpoint was stored correctly, $= 0$ if the process is being restarted from a checkpoint, and < 0 to indicate an error.

Roll back and Log Replay

The operation `renew_processFailure` is invoked when the fault detection module locates one or more process failures. This operation has two purposes; it notifies the protocol about the failures, and it lets the protocol specify which processes have to roll back. The arguments ids and $ckps$ are arrays with an entry for each process in the system. If $ids[proc]$ has a value different than zero, it indicates that process $proc$ has failed. Using the information in ids, and possibly with the cooperation of the other live processes, the protocol must determine the checkpoint names from which the processes should be restarted. If process $proc$ has to roll back, the entry $ckps[proc]$ should be set with the name of the checkpoint file. Otherwise, the entry should be set to zero.

After roll back, a process restarts the execution from the same state it had at the time of the checkpoint. Consequently, the protocol starts to re-execute from the last operation it called before checkpointing, which was `renew_createCkp`. Using the return value from the function, it can determine that the process is in recovery mode. The process stays in this mode until the protocol informs RENEW that recovery is completed. This is done by returning 0 when the operation `renew_replayMesg` is called.

In recovery mode, RENEW continues to transmit the messages sent by the application. However, when the application attempts to receive a message, RENEW calls `renew_replayMesg`, instead of trying to read it from the network. The protocol should then copy a header and the contents of a message to $head$ and msg, respectively. Messages must be returned in the same order that were logged, otherwise recovery may be incorrect.

Communication, Data Storage, and Timers

RENEW also exports operations for communication, data storage and retrieval, and timers. The communication functions let the protocol exchange data between processes in a FIFO ordered reliable manner. There are two sets of functions; one where sends have to be explicitly matched with the receives, and another where sends result in an up-call executed at the receiver. The operations for data storage and retrieval are similar to the Unix file functions. Their usage is recommended since they allow data to be saved in the remote servers. In Unix, only one timer can be active per process at a given time. Since the message passing module and the fault detection mechanism need to have timers simultaneously, RENEW implements a queue of timers on top of the system timer.

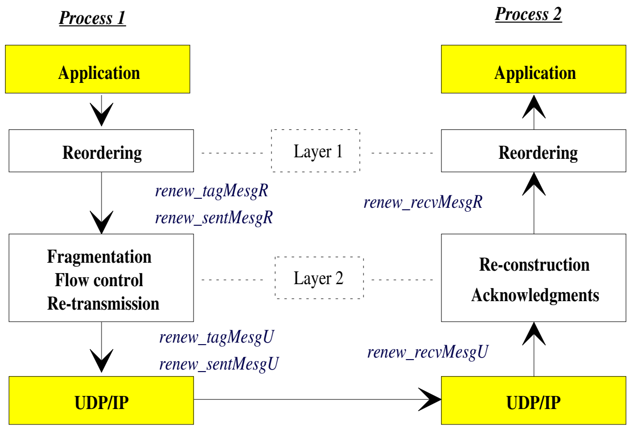

Figure 2: Message tagging and logging implementation.

Assessing Programming Costs

The Appendix shows the complete implementation of the coordinated protocol utilized in the experiments. The programs for the communication-induced and message logging protocols had 153 and 433 lines of code, respectively.

3.3 Implementation

Due to space limitations, this section will focus mainly on two aspects of RENEW implementation; the reliable and unreliable message tagging and logging operations and the recovery of processes.

3.3.1 Message Tagging and Logging

A message sent by an application traverses two software layers in RENEW before it is written to a datagram socket (see Figure 2). The first layer, reordering, is necessary due to the requirements for message progress and ordering from the MPI specification. MPI defines several types of send operations with blocking and non-blocking semantics. With non-blocking operations, a process can, for example, post a number of message sends on the system which can then be received in reverse order. To address this problem, RENEW uses two communication protocols; a short protocol for small messages and a long protocol for large messages [2].

The short protocol attempts to send a message as soon as there are no other messages waiting to be transmitted to the same process. When the message arrives at the process, it is saved in a queue until the application posts the matching receive. The receive queue can become potentially very large if the short protocol is also used for large messages. This problem can be especially important if many processes are utilized in the computation, with the adverse effect of increasing the process's checkpoint size. The long protocol starts by transmitting a $req - to - send$ message containing all information necessary to match receives to sends. On the receive side, $req - to - send$ is queued like a normal message until the matching receive is executed. When this happens, the receiver transmits a $ready$ message back to the sender, and then the application's message is sent.

Throughout the message transmission path, the only point where reliable FIFO order is guaranteed is between the two software layers. The message passing module calls `renew_tagMesgR` when it passes a message to the second layer. The message can belong to the application or it can be one of the auxiliary messages from the long protocol. Since $req - to - send$ and $ready$ are only 16 bytes in size, they do not create performance problems for message logging protocols. When the message is sent or queued for transmission, the module invokes `renew_sentMesgR`. On the receive side, `renew_recvMesgR` is executed when the message is transferred from the second to the first layer.

RENEW uses datagram sockets, based on the UDP transport protocol, for communication. Since UDP provides an unreliable communication service, the second layer implements message fragmentation, flow control and packet loss recovery. Stream sockets based on TCP were used in one of the earlier versions of RENEW. We decided to change because much of the functionality of TCP had to be replicated in RENEW. The operations `renew_tagMesgU` and `renew_sentMesgU` are called before and after the execution of the send. The `renew_recvMesgU` operation is invoked when a message arrives.

3.3.2 Process Recovery

RENEW relies on the fault detection module to locate process failures and initiate the recovery. In the current implementation, the module keeps a ring of stream sockets connecting all processes. Periodically, each process forwards a message along the ring and expects to receive a message from the ring. A failure condition is triggered if no message arrives for two intervals or if an error is returned from one of the two sockets. The fault detection module then runs an agreement protocol to determine which processes need to be recovered and to guarantee that all live processes agree on the failures. Next, process recovery is executed: first, new processes are started in the available nodes; second, the processes' states are restored using the checkpoints; and last, message replay is performed if necessary. The remainder of this section explains in greater detail the steps of fault recovery, using the example from Figure 3.

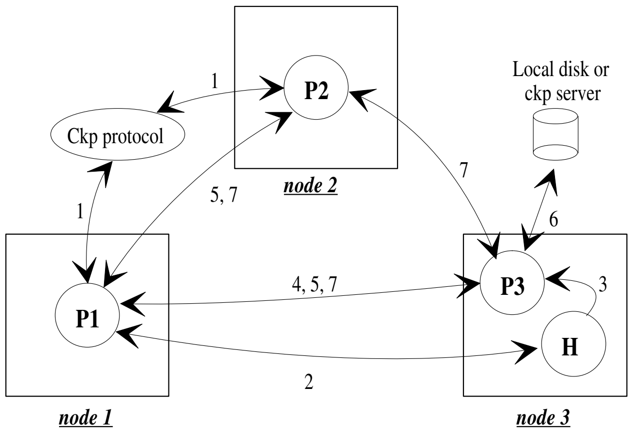

Figure 3: Recovery of process $P3$.

The first phase of recovery begins by selecting a coordinator, the live process with smallest identifier (process $P1$). The main responsibility of the coordinator is to ensure that all steps of recovery are completed properly. Then, the operation `renew_processFailure` of the checkpoint protocol is called to determine which processes have to be restarted. The protocol, either individually in each process or with the cooperation of all processes, should provide a list of checkpoint names, one for each failed process and possibly other names for processes that need to roll back. The coordinator then spawns a $helper$ program in one of the available nodes (process H). By default, an attempt is made first to start the helper in the same node where the failed process was executing because checkpoints might have been stored in the local disk. This attempt is assumed to have failed if the helper does not contact the coordinator within a specified interval of time. In this case, a new helper is spawned in another node. The arguments of the helper include an address of a coordinator socket and an identifier. The identifier is used to recognize helpers that take more than the specified interval to transmit the first message. These helpers are required to exit since they are no longer needed.

The main function of the helper is to guarantee that all information necessary for recovery is available in the local node. It receives from the coordinator the program and checkpoint names, a set of pathnames where the program might be found, and some environmental variables. The helper then checks if the checkpoint and program files can be accessed, and it initiates the new process (process $P3$). In the environment of the process, one variable is set with

Table 2: Applications Used in the Experiments.

Application	Problem Description	Messages		Ckp		Mesg Log	
		Mesg/sec	KBytes/sec	MBytes	sec	MBytes	sec
BT	Class A, 500 iterations	49.1	1053.3	81.2	17.7	75.1	15.7
LU	Class B, 250 iterations	124.3	448.8	50.2	11.8	31.7	4.3
SP	Class B, 400 iterations	74.4	1695.2	92.6	20.2	103.0	21.6
Seismic1	Small, 512 samples/trace	0.1	0.0	1.2	0.4	0.9	0.1
Seismic4		1.2	13.9	7.2	1.8	1.8	0.1
PCCM2	T42 resolution, one day	32.6	454.6	82.5	18.5	31.9	5.4

the checkpoint name to indicate that recovery is being performed. At this moment, the new process contacts the coordinator, and then the helper can terminate its execution. The coordinator next collects a socket address from each of the processes, and then distributes the addresses to allow independent communication among the processes ($step$ 5).

In the second phase of recovery, processes roll back using the information saved in the checkpoints. The checkpoint file can be stored in the local disk or in a remote server. In the second case, the server is requested to transmit the checkpoint. When the processes' state is reconstructed, new connections are established to prevent the reception of messages that might still be in the network ($step$ 7). Processes also exchange the current send sequence numbers to purge the receive queues from messages with sequence numbers larger then their senders. These messages could cause live lock problems in the checkpoint protocols [13]. RENEW then returns to the `renew_createCkp` operation, letting the protocol and application restart their execution.

During the last phase of recovery, the application program is unaware of the failure. Whenever it tries to receive a message, RENEW calls the `renew_replayMesg` operation of the checkpoint protocol. This operation should return the next message that was received during the failure-free period. This phase finishes when all messages have been replayed.

4 Experimental Setting

4.1 Checkpoint Protocols

Three types of checkpoint protocols were utilized in the experiments: coordinated, communication-induced, and optimistic sender-based message logging. The coordinated protocol used time to indirectly coordinate the processes at checkpoint time [3]. Instead of exchanging messages, the protocol keeps the processes' checkpoint timers roughly synchronized, and creates the checkpoints whenever they expire. To guarantee that the checkpoints form a consistent state and to adjust the timers, the protocol piggybacks in the messages a checkpoint number and a time interval. Failure recovery proceeds as in most coordinated protocols; processes roll back to the last available global state and then start to re-execute. An implementation of the protocol is presented in the Appendix.

The communication-induced protocol coordinates the creation of the checkpoints in a lazy fashion, by piggybacking a checkpoint sequence number in the messages [4]. If a process receives a message with a sequence number larger than the local one, it has to take a forced checkpoint. To avoid having to increase the sequence number, and consequently to reduce the number of forced checkpoints, the protocol tries to determine if new checkpoints are equivalent to previous ones with respect to the current recovery line. To track the equivalence between checkpoints, the protocol also needs to associate an equivalence number with the checkpoints, and it has to piggyback an array of equivalence numbers in the messages.

The optimist sender-based message logging protocol saves the application's messages in the volatile memory of the sender processes [5]. When a message is received, the protocol associates with it a sequence number, and then piggybacks the number in the next message returned to the sender. The log is saved to disk every time a checkpoint is created or when the allocated space is exhausted. During recovery, a process uses the sequence numbers to replay in the correct order the messages stored by the senders. Compared with more recent sender-based protocols, such as the one from Alvisi et al. [17], our implementation performs equally well since no extra messages are sent and no blocking is done at the receiver while it waits for the sequence numbers to be logged. The negative side is that receivers might transmit new messages before sequence numbers are stored, and consequently, a failed process might not be able to completely recover its state.

4.2 Applications and Environment

Five long-running parallel applications were used in the experiments. Table 2 presents, for each application, a description of the problem solved, communication rates and average values for checkpoint size and time. The values shown in the last two columns correspond to average log overhead incurred by a process using the message logging

Table 3: Failure-Free Results (Ckp. Period 5 min.)

Applic	No Ckp	Coordinated			Comm.-Induced			Message-Logging		
	sec	#	sec	%	#	sec	%	#	sec	%
BT	2530 (± 9)	8	2661 (± 4)	5.2	8	2671 (± 5)	5.6	9	2873 (± 10)	13.6
LU	2712 (± 3)	9	2805 (± 6)	3.4	9	2806 (± 19)	4.1	9	2902 (± 20)	7.0
SP	2841 (± 7)	10	3092 (± 5)	8.8	10	3082 (± 8)	8.5	11	3525 (± 22)	24.1
Seismic1	1379 (± 4)	4	1406 (± 24)	2.0	4	1405 (± 19)	1.9	4	1405 (± 10)	1.8
Seismic4	2147 (± 5)	7	2202 (± 5)	2.5	7	2296 (± 2)	6.9	7	2187 (± 2)	1.9
PCCM2	3582 (± 7)	12	3796 (± 12)	6.0	12	2813 (± 13)	6.4	13	3905 (± 26)	9.0

protocol, when the checkpoint period is 5 minutes.

- **BT**, **LU**, and **SP** are applications from the NAS benchmarks, developed by the Numerical Aerodynamic Simulation program, located at NASA Ames Research Center [29]. These applications reproduce much of the data movement and computation found in computational fluid dynamics codes.

- **Seismic** is an application from the high-performance computing SPEC benchmarks [30]. This application is used for seismic data processing, and reflects the current technology trends in the oil industry. One benchmark run consists of four executions of the **Seismic** program with different arguments. Since the second and third executions take less than one checkpoint period to run, they were not considered in the experiments.

- **PCCM2** is a parallel version of the NCAR Community Climate Model, developed by the CHAMMP program at the Oak Ridge and Argonne National Laboratories and the National Center for Atmospheric Research [31]. This application is a comprehensive three-dimensional global atmospheric model that has been improved over the past 15 years.

The experiments were performed on a cluster of four Sun UltraSparc workstations running the Solaris 2.5 operating system. Each machine had 512 MBytes of main memory and 4 GBytes of local disk. The interconnection network was an 155 Mbits/s ATM. All experiments were done during the night since the workstations are utilized for the daily work of the members of the research group. The values reported in the next section were obtained by averaging the five best results of at least ten experiments. More experiments were run in some cases to ensure that the confidence intervals were in the order of one percent of the mean values.

Using the reported values for the NAS benchmarks, the current version of RENEW shows, for the same number of nodes, better performance than an Intel Paragon or the UC Berkeley's NOW project, and worse performance than an IBM RS/6000 SP [32].

5 Performance Results

In the experiments, the applications were executed by four processes, each running on a different workstation. The checkpoints were saved in the local disk once every five minutes. The memory exclusion optimization was utilized only on the message logging protocol, to avoid writing the unused parts of the log. The memory size allocated for the log was 50 MBytes per process. Table 3 presents the failure-free execution times for the three protocols, together with the 95% confidence intervals.

The coordinated and communication-induced protocols displayed approximate performance, with overheads smaller than 9 % in all applications. This conclusion was confirmed using the t-test [33]: on the **LU** and **Seismic1** the protocols showed equivalent performance; on **BT**, **Seismic4** and **PCCM2** the coordinated protocol was better; and on **SP** the communication-induced protocol was better. Both protocols introduce primarily two performance penalties, the checkpoint storage and message tagging. The first one is the most important, and it accounts for the majority of the difference between the times with and without checkpointing. For instance, on the **SP** application the total overhead is 241 seconds, from which 202 seconds were spent on the writes (see Table 2). In practice, this penalty can be even higher since writes were asynchronous[2]. Even though the communication-induced protocol has a more complex tagging scheme than the coordinated, its influence on performance was not perceptible. In experiments with more processes, the communication-induced protocol would not scale as well since its tag includes an array with size proportional to the total number of processes.

During the execution of the communication-induced protocol, no forced checkpoints were observed since processes saved their states at similar times. To study the behavior of the protocol with un-synchronized timers, tests

[2] When a process attempts to store the memory contents to disk, the operating system only initiates the operation, and the actual writes are performed while the process continues to execute.

Table 4: Recovery Times for the Coordinated and Message Logging Protocols (values in sec).

	Phase 1		Phase 2		Phase 3 / Total	
	Coord	Mesg-L	Coord	Mesg-L	Coord	Mesg-L
BT	1.4 (± 0.3)	1.1 (± 0.3)	3.1 (± 0.3)	3.1 (± 0.2)	289.7 (± 3.8)	298.2 (± 1.3)
LU	0.8 (± 0.2)	1.1 (± 0.5)	2.1 (± 0.4)	2.4 (± 0.5)	308.1 (± 21.8)	285.4 (± 12.0)
SP	1.4 (± 0.6)	1.2 (± 0.5)	4.4 (± 2.1)	3.5 (± 0.5)	290.7 (± 11.8)	296.8 (± 8.1)
PCCM2	0.7 (± 0.0)	1.1 (± 0.5)	2.5 (± 0.0)	3.2 (± 0.5)	272.8 (± 7.7)	293.3 (± 16.7)

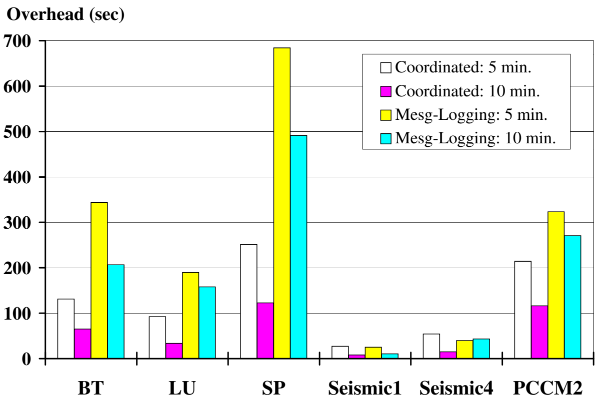

Figure 4: Failure-free overheads.

were performed where processes would start to store their states one minute apart. It was observed that after the first process terminated its checkpoint, it would induce a checkpoint in the other processes almost immediately. Then, the other processes would skip the scheduled checkpoint when the timers expired. The protocol incorporates several optimizations, and one of them skips a basic checkpoint if a forced checkpoint was already taken in the same interval. This optimization removed all the extra checkpoints, and as a result the failure-free overheads were much smaller. The cost of using this type of optimization is that processes loose their autonomy to schedule their own checkpoints.

The message logging protocol showed the worst performance due to the high message traffic of some applications (see Table 2). In two of the applications, **BT** and **SP**, the log had to be written to disk more than once between checkpoints. Since the log size is 50 MBytes, the performance costs due to the log and checkpoints were in the same order of magnitude. For longer checkpoint intervals the importance of log handling becomes even more significant, not only in terms of execution time, but also in terms of disk space. For instance, with an interval of one hour, the **SP** application would require more than 4.5 GBytes to hold the log. Figure 4 displays the overheads for checkpoint intervals of five and ten minutes. As expected, in the coordinated protocol the overheads were reduced close to half when the interval doubled. In the message logging protocol the performance was not improved as much, since the logging costs stayed the same.

Elnozahy and Zwaenepoel have analyzed the performance of message logging and coordinated protocols [18]. Even though many characteristics distinguish the two studies, e.g., hardware and applications, we reached the same basic conclusion that coordinated protocols introduce smaller overheads than message logging protocols. Our experiments, however, show that the performance gap between protocols has become wider.

Table 4 displays the recovery times for the coordinated and message logging protocols. No values are presented for the communication-induced since it emulated the coordinated protocol. In the experiments, processes were allowed to create their first checkpoint, and then, when they were about to save their second checkpoint, one of the processes would exit. Next, the fault detection would initiate recovery. The values for *Phase 1* correspond roughly to four operations: ask the checkpoint protocol which processes have to roll back, spawn the helper program, start the new process, and exchange configuration information. With the coordinated protocol, the first operation is accomplished relatively fast since all process are required to roll back. The message logging protocol takes a little longer since processes have to determine if roll back is possible. *Phase 2* corresponds to the interval starting from the fault detection until the application restarts execution. Coordinated protocols can tolerate new failures when this phase finishes. The last columns display the total time until the application has re-executed all the lost work or the time until the end of log replay.

6 Conclusions

This paper describes the design and implementation of RENEW, a tool that facilitates the development and testing of checkpoint protocols on clusters of workstations. RENEW offers a simple but powerful set of operations that allow the implementation of protocols. Three checkpoint protocols were evaluated on a cluster of workstations interconnected by ATM, with SPEC and NAS benchmarks and large climate modeling code. It was observed that the communication-induced and coordinated protocols had roughly equal performance. The optimistic sender-based message logging showed significant overheads for a few applications because of the high traffic rate. Failure recovery experiments indicate that both the coordinated and message-logging protocols require approximately the same amount of time to restore the application state.

Acknowledgments

We would like to thank Prof. W. Sanders for suggesting the use of confidence intervals in the experimental section, and our research group members that gave up to their nightly work during the the performance evaluation measurements. We also wish to thank the anonymous referees for their comments.

References

[1] E. Elnozahy, D. Johnson, and Y.-M. Wang, "A survey of rollback-recovery protocols in message-passing systems", School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-96-181, Oct. 1996.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The complete reference*, MIT Press, 1996.

[3] N. Neves and W. K. Fuchs, "Adaptive recovery for mobile environments", *Communications of the ACM*, vol. 40, no. 1, pp. 68–74, Jan. 1997.

[4] R. Baldoni, F. Quaglia, and P. Fornara, "Index-based checkpointing algorithm for autonomous distributed systems", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 27–34, Oct. 1997.

[5] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 14–19, Jul. 1987.

[6] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A recoverable distributed shared memory integrating coherence and recoverability", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 289–298, Jun. 1995.

[7] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory", *IEEE Trans. on Computers*, vol. 39, no. 4, pp. 460–469, Apr. 1990.

[8] R. Netzer and J. Xu, "Adaptive message logging for incremental program replay", *IEEE Parallel and Distr. Technology*, vol. 1, no. 4, pp. 32–39, Nov. 1993.

[9] D. K. Pradhan, P. Krishna, and N. H. Vaidya, "Recovery in mobile environments: Design and trade-off analysis", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 16–25, Jun. 1996.

[10] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: Models, characterization, and classification", Ohio State University, Tech. Rep. OSU-CISRC-5/96-TR33, 1996.

[11] B. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems – An optimistic approach", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 3–12, Oct. 1988.

[12] K. M. Chandy and L. Lamport, "Distr. snapshots: Determining global states of distributed systems", *ACM Trans. on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[13] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems", *IEEE Trans. on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[14] K. Li, J. F. Naughton, and J. S. Plank, "An efficient checkpointing method for multicomputers with wormhole routing", *Inter. Journal of Parallel Programming*, vol. 20, no. 3, pp. 23–31, 1991.

[15] N. Neves and W. K. Fuchs, "Using time to improve the performance of coordinated checkpointing", *Proc. of the Inter. Computer Performance & Dependability Symp.*, pp. 282–291, Sep. 1996.

[16] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 207–215, Oct. 1984.

[17] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and orphan-free message logging protocols", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 145–154, Jun. 1993.

[18] E. N. Elnozahy and W. Zwaenepoel, "On the use and implementation of message logging", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 298–307, Aug. 1994.

[19] B. Ramamurthy, S. Upadhyaya, and R. Iyer, "An object-oriented testbed for the evaluation of checkpointing and recovery systems", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 194–203, Jun. 1997.

[20] K. Goswami, R. Iyer, and L. Young, "Depend: A simulation-based environment for system level dependability analysis", *IEEE Trans. on Computer*, vol. 46, pp. 60–74, Jan. 1997.

[21] A. Hein and K. Bannsch, "Simpar - A simulation environment for performance and dependability analysis of user-defined fault-tolerant parallel systems", University of Erlangen-Nurnberg, Tech. Rep., 1995.

[22] J. Leon, A. L. Ficher, and P. Steenkiste, "Fail-safe PVM: A portable package for distributed programming with transparent recovery", School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-93-124, Feb. 1993.

[23] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MIST: PVM with transparent migration and checkpointing", *3rd PVM Users' Group Meeting*, May 1995.

[24] G. Stellner, "CoCheck: Checkpointing and process migration for MPI", *Proc. of the Inter. Parallel Processing Symp.*, Apr. 1996.

[25] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 39–47, Oct. 1992.

[26] J. Plank, "Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 76–85, Oct. 1996.

[27] Y.-M. Wang and W. K. Fuchs, "Scheduling message processing for reducing rollback propagation", *Proc. of the Inter. Symp. on Fault-Tolerant Comp.*, pp. 204–211, Jul. 1992.

[28] N. Neves and W. Kent Fuchs, "Fault detection using hints from the socket layer", *Proc. of the Symp. on Reliable Distr. Systems*, pp. 64–71, Oct. 1997.

[29] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0", NASA Ames Research Center, Tech. Rep. NAS-95-020, Dec. 1995.

[30] C. Mosher and S. Hassanzadeh, "ARCO Seismic Processing Performance Evaluation Suite: Seis 1.0 User's Guide ", Tech. Rep., Oct. 1993.

[31] J. Drake, R. Flanery, B. Semeraro, P. Worley, I. Foster, J. Michalakes, J. Hack, and D. Williamson, "Parallel Community Climate Model: Description and User's Guide", Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-12285, May 1996.

[32] NASA Ames Research Center, *NPB 2 Detailed Results: Graphs and Raw Data*, Nov. 1997, http://science.nas.nasa.gov/Software/NPB2Results.

[33] R. Jain, *The art of computer systems performance analysis*, John Wiley and Sons, 1991.

Appendix: Coordinated Protocol

```
/*---------------------------------------------------------
|   Coordinated.h
+---------------------------------------------------------*/
#define CKP_INFO_SIZE   4
#define renew_tagMesgR(d,cb)
#define renew_sentMesgR(d,h,hs,b,bs)
#define renew_recvMesgR(s,cb,h,hs,b,bs)
#define renew_sentMesgU(d,t,ssn,h,hs,b,bs)
#define renew_ckpRecvActive(b,bs,s,t)
#define renew_endCkpProt()
#define renew_tagMesgU(d,t,ssn,cb)  *(int *)cb = tag

extern unsigned int tag;

/*---------------------------------------------------------
|   Coordinated.c
+---------------------------------------------------------*/
#include "mpi.h"

#define CKP_INTERVAL   300
#define CKP_INTER(t) (ckpTime.tv_sec-(t).tv_sec)
#define SEQ_GT(a, b) (((char)((a) - (b))) > 0)

struct timeval ckpTime;
unsigned char cn = 0;
unsigned int tag, nprocs;

void createCkp(int code) {
    struct timeval t;

    cn++;
    if (!renew_createCkp((cn % 2) ? "CKP1" : "CKP2", 0, 0)) {
        renew_initCkpProt(0, nprocs);
        return;
    }

    /* Set the next checkpoint time and tag */
    ckpTime.tv_sec += CKP_INTERVAL;
    renew_timeAlarm(1, &ckpTime);

    gettimeofday(&t, 0);
    tag = (CKP_INTER(t) << 8) + cn;
}

void renew_initCkpProt(unsigned id, unsigned n) {
    nprocs = n;

    /* Set the timer signal handler */
    renew_associateAlarm(1, createCkp);
    gettimeofday(&ckpTime, (struct timezone*) 0);
    ckpTime.tv_sec += CKP_INTERVAL;
    renew_timeAlarm(1, &ckpTime);

    /* Set the tag for next checkpoint interval */
    tag = (CKP_INTERVAL << 8) + cn;
}

void renew_recvMesgU(int s, int t, int ssn, char *cb,
            char *h, int hs, char *b, int bs) {
    struct timeval t;
    unsigned char m_cn = (*(unsigned *)cb) & 0xff;
    unsigned m_inter = (*(unsigned *)cb) >> 8;

    gettimeofday(&t, 0);
    if ((m_cn == cn) && (m_inter < CKP_INTER(t))) {
        ckpTime.tv_sec = t.tv_sec + m_inter;
        renew_timeAlarm(1, &ckpTime);
    } else if (SEQ_GT(m_cn, cn)) {
        renew_holdAlarm(1);
        createCkp(0);
        ckpTime.tv_sec = t.tv_sec + m_inter;
        renew_timeAlarm(1, &ckpTime);
        renew_releaseAlarm(1);
    }
}

void renew_processFailure(int *ids, char **ckps) {
    int i;
    for (i = 0; i < nprocs; i++)
        ckpts[i] = (cn % 2) ? "CKP1" : "CKP2";
}

int renew_replayMesg(int s, char *h, int hs, char *m, int *ms) {
    return 0;
}
```