TIME-BASED COORDINATED CHECKPOINTING

BY

NUNO F. NEVES

Licenciatura, Universidade Técnica de Lisboa, 1992
Mestrado, Universidade Técnica de Lisboa, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

TIME-BASED COORDINATED CHECKPOINTING

Nuno Ferreira Neves, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1998
W. Kent Fuchs, Advisor

Distributed systems are being used to support the execution of applications ranging from long-running scientific simulators to e-commerce on the Internet. In this type of environment, the failure of one of its components, either a computer or the network, may prevent other components from completing their tasks. Since the probability of failure increases with the number of computers and execution time, it is likely that these applications will be interrupted unless provision is made for failure handling. In this thesis we address the problem of fault recovery in distributed systems.

The thesis describes two variations of a coordinated checkpoint protocol that uses time to remove most causes of overhead, and to avoid all types of direct coordination. The time-based protocol does not have to transmit extra messages, does not need to tag the application messages, and only accesses the stable storage when the checkpoints are saved. The thesis also describes a new coordinated checkpoint protocol that is well adapted to mobile environments. It uses time to indirectly coordinate the creation of new global states, and it saves two different types of checkpoints to adapt its behavior to the current network characteristics.

Traditional techniques for fault diagnosis in distributed systems, either based on watch-dogs or polling, exchange performance with detection latency. The thesis introduces a complementary mechanism that uses the error codes returned by the stream sockets. Since these errors are generated automatically when there is communication with a failed process, the mechanism incurs only in small overheads. Our results show that, in most cases, failures could be located using only the errors from the sockets.

A large number of checkpoint-based recovery protocols have been proposed in the literature, however, most of them were never evaluated. The thesis describes the design and implementation of a run-time system for clusters of workstations that allows the rapid testing of checkpoint protocols with standard benchmarks. *RENEW - Recoverable Network of Workstations* provides a flexible set of operations that facilitates the integration of checkpoint and rollback recovery protocols.

To my father, Marinho Ferreira Neves.

Acknowledgments

To my advisor, Professor W. Kent Fuchs, for his comments and suggestions. This research would not have been possible without his invaluable support and guidance.

To the members of my dissertation committee, Professors Gul Agha, Ravishankar Iyer, Jane Liu, Laxmikant Kale, and William Sanders, for their many contributions to this thesis. Special thanks are due to Professor William Sanders for his help on the treatment of experimental data.

To Jenny Applequist and Jill Comer for carefully reading the papers and for correcting their imperfections.

To Pedro Trancoso for his constant encouragement and friendship. Thanks are also due to my friends Enamul Amyeen, Kuo-Feng Ssu, Vamsi Boppana, Bin Yao, Chen Wang, and Srikanth Venkataraman for their help during my graduate studies.

To my parents, brother and grandfather, for the enthusiasm and support. A special thanks for my wife, Cristina, for her patience and encouragement.

Table of Contents

Chapter **Page**

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1

Introduction

Distributed applications can be found in a large number of fields, such as automated banking systems, parallel simulators, aircraft control systems, cooperative editors and games. The main characteristic of these applications is that they are executed by a group of processes that cooperate to carry out a common goal [1]. Typically, the processes run on a set of machines, possibly heterogeneous, connected by a network. The physical separation among the processes increases the probability of an application crash since the network or any of the machines can fail. Whenever this happens, live processes might not be able to continue their work unless the system provides support for fault tolerance.

This thesis addresses the problem of fault recovery in distributed systems. The techniques described are based on checkpointing and roll back recovery. Two types of computing environments are considered: a cluster of workstations and a mobile system.

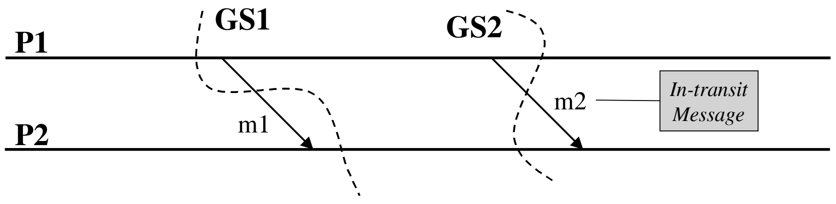


Figure 1.1: Two global states: *GS1* is inconsistent, and *GS2* is consistent.

1.1 Definitions and Performance Metrics

1.1.1 Consistent and Recoverable Global States

A distributed application is run by a group of processes executing on several machines that cooperate by exchanging messages. The main responsibility of a checkpoint protocol is to save, during the failure-free intervals, information about the application execution. When a failure occurs, the protocol uses the stored information to recover the failed processes, bringing the application to a state that is *consistent*. Intuitively, a global state is consistent if it could have occurred during a failure-free execution of the application [2]. A global state includes the state of each process and the messages that are in transit in the network. The following property is verified by a consistent global state,

Consistency : If the reception of a message m_i is reflected in the global state then the send of message m_i must also be reflected in the global state.

The previous requirement does not consider the messages that are in-transit in the network and that are also part of the global state (e.g., message *m2* in Figure 1.1). A message is in-transit if it was transmitted *before* the global state and is received *after* the global state. The checkpoint protocol must be able to restore all the in-transit messages (e.g., by logging and re-sending them) to guarantee that no messages become lost during the application re-execution. Therefore, the application should be recovered to a global state that also verifies the following property,

Recoverability : If the send of message m_i is reflected in the global state then the reception of message m_i must also be reflected in the global state or the checkpoint protocol must be able to restore the message.

1.1.2 Checkpoint Overhead and Latency

The performance metric that is most widely used to compare checkpoint protocols is the overhead. By definition, overhead is equal to $(Exec\_Time\_With\_Ckp/Original\_Exec\_Time) - 1$, and it accounts for the performance degradation on the application execution time as a result of the introduction of a checkpoint protocol. Typically, users will utilize a protocol with less overhead, unless their application requires a special characteristic that is only provided by a specific protocol. Some of the factors that contribute to the checkpoint overhead are the following:

- *Checkpoint Storage*: Checkpoint protocols need to save periodically the state of the application. Whenever this happens, processes have to create a snapshot of their state, and then they have to write it in stable storage. Checkpoint storage is the only overhead that can not be avoided, however it can be decreased with optimizations like *copy on write* [3] and *main memory exclusion* [4].

- *Extra Accesses to Stable Storage*: In some cases, protocols might have to make extra accesses to stable storage to save information necessary for recovery (e.g., message log or in-transit messages). The protocol should merge several accesses into one, since it is normally more efficient to do fewer writes with more information.

- *Extra Messages*: Coordinated protocols usually exchange extra messages during checkpoint creation to ensure that all processes initiate their checkpoints. The actual number of transmitted messages is highly dependent on the protocol; in the well-known protocol by Chandy and Lamport [2] it is in the order of O(n^2), where n is the number of processes.

- *Message Tagging*: Sometimes protocols piggyback information on the messages of the application. This information is used, for instance, to enforce the consistency property or to minimize the number of processes that have to roll back. The amount of added information can be a fixed quantity or can be proportional on the number of processes.

- *Blocking*: Processes might be required to suspend their normal execution or to stop transmitting messages during certain periods of time, while they wait for one task to be finished. For example, a pessimistic sender-based message logging protocol disallows message sends until all *receive sequence numbers* have been saved [5]. The blocking interval is usually proportional to the message delivery times.

An important concern of high-availability applications is to minimize down time. With checkpointing, the length of recovery is proportional to the checkpoint period. Larger checkpoint periods result on average in larger rollbacks, and consequently in longer recovery times. Typically, a protocol only begins to create a new application checkpoint when the previous one has been completely stored. Therefore, the checkpoint period is lower bounded by the checkpoint latency. The checkpoint latency is defined as the time a protocol takes to save a new checkpoint [6, 7].

1.2 Types of Checkpoint Protocols

1.2.1 Coordinated Protocols

A coordinated protocol saves during the failure-free periods consistent and recoverable global states of the application. After a failure, recovery is performed in a relatively simple way: first, the checkpoints of the failed processes are reloaded in the available nodes and the surviving processes roll back to their last checkpoints. Next, processes re-execute the application program and re-read the logged in-transit messages. Examples of this type of protocol can be found in [2, 3, 8–26].

The major difficulty in designing this type of protocol comes from the distributed nature of the systems. In general, it is impossible to stop and save the state of all application's processes and in-transit messages at once. The usual solution to this problem consists of dividing the storage of a new global state into two (or more) phases [2, 3, 16, 19, 22, 23]. In the initial phase, processes try to agree on a new consistent state by exchanging a few messages. Since the whole procedure is not atomic, failures can occur in the middle, processes start by saving a tentative global state [12]. In the second phase, the tentative global state is made permanent. In case of a failure, processes roll back at most to their last permanent checkpoint. Therefore, the system only needs to keep the last checkpoint of each process in the intervals between the creation of new checkpoints, and the previous ones can be removed. Coordinated protocols also have to ensure that the recoverability property is verified. This is usually done by logging the in-transit messages as they arrive at the receiver [2, 19], or by including them in the senders' checkpoints [12].

The various coordinated protocols that have been described in the literature differ, for instance, in terms of the amount of concurrency allowed during checkpoint creation, and on the assumptions of the underlying support system (e.g., communication channels are FIFO or not). Protocols have also been proposed that try to minimize the number of processes that have to take part on the checkpoint creation, and that have to roll back after a failure [8, 11, 12, 27]. Other protocols have assumed synchronized clocks and bounded message delivery times to avoid the exchange of the coordination messages [10, 18, 24].

1.2.2 Uncoordinated Protocols

Uncoordinated protocols decrease the checkpoint overhead by avoiding the message exchanges during the creation of the checkpoints [28–37]. With these protocols, each process saves its state independently from the others. After a failure, processes have to find a consistent global state to restart the computation, which in some cases may require some sort of coordination among the surviving processes and the new processes, and in others it may involve only the new processes.

A protocol based on the previous description has as a main advantage the small overhead that it introduces during the failure-free periods [28]. The protocol only has to save the checkpoints and to keep some information about the messages that were transmitted. Unfortunately, during recovery, it can suffer from the *domino effect* [38], which can result in an unbounded number of roll backs as the processes attempt to find a consistent global state.

To prevent the domino effect, *communication-induced protocols* store extra checkpoints when processes exchange information. A protocol can guarantee that only a bounded number of roll backs will occur by storing a checkpoint before delivering a message, if the application had sent a message since the last checkpoint [32, 39]. This protocol also has to log all received messages so that they can be replayed during recovery. When a process fails, it returns to the last checkpoint, and then it propagates the roll back to the other processes until consistency is found.

Another way of avoiding the domino effect is to create a checkpoint whenever a process sends data to another process. The advantage of this solution is that only the failed process needs to rollback, and the other processes can continue their execution without being disturbed. A second advantage is that only one checkpoint has to be kept per process. Variations of this basic idea have been proposed for shared memory multiprocessor systems [40, 41], for distributed shared memory systems [30, 34, 37], and message passing systems [33, 42].

Communication-induced protocols can create a large number of checkpoints. One way to address this problem requires that only some of the checkpoints are consistent [35]. The protocol defines a *laziness* Z, and associates an increasing number to each process' checkpoint. Then, it only guarantees that the checkpoints with numbers multiple of Z are consistent. This technique, however, involves a tradeoff between the amount of roll back and checkpoint overhead. More recently, other techniques have been proposed to lower the number of induced checkpoints [43, 44].

1.2.3 Log-Based Protocols

Log-based checkpoint protocols assume that processes execute in a piecewise deterministic manner [45]. Under this assumption, the execution of a process can be divided into a sequence of deterministic intervals, started by a non-deterministic event. By logging the outcome of each non-deterministic event, these protocols are able to recreate during recovery the state of a process affected by a failure. Most of these protocols also require that the only non-deterministic events are the receive events. Even though this assumption might appear too strong, for several classes of applications the assumption is valid.

A log-based protocol has to save only two things to be able to recover a process: the contents of all messages and the order of their reception. If all this information is available after a failure, only the failed process needs to roll back, and the surviving processes can continue their execution without being disturbed. A process is recovered by loading its last checkpoint on a free processor, and then by letting it run the program. Whenever it tries to receive a message, the checkpoint protocol returns the same message that was originally read. The protocol also detects and removes the messages that the process tries to transmit to avoid duplicates.

Logging can be done at the receiver, at the sender, or on a dedicated machine [46]. Simplicity is the main advantage of logging at the receiver [47–51]. The protocol saves a copy of the messages as they arrive. The reception order usually does not have to be logged since messages are replayed in the same order that they were stored. Logging at the sender is more complex because one process saves the message contents, the sender, and another knows the order of reception, the receiver [5, 45, 52–55]. The main advantage of these protocols is the low logging overhead. By assuming that the sender fails independently from the receiver, the sender process can log the messages in the volatile memory, instead of writing them to stable storage. The messages are only stored in stable storage when the sender process creates a new checkpoint. The receiver has to guarantee that the reception order is not lost with failures by saving it in stable storage [45], at the sender [5, 54], or in a remote process [52].

Log-based protocols can be pessimistic or optimistic. Pessimistic protocols guarantee that all information related to the non-deterministic events is logged before the process is allowed to communicate [5, 46, 47, 49, 52–54, 56]. With these protocols only the failed processes have to roll back. Moreover, processes only need to keep their most recent checkpoint, and they do not need to run a garbage collection protocol to remove the data that is no longer necessary for recovery. Optimistic protocols decrease the failure-free overheads by logging the messages asynchronously [45, 48, 50, 57–60]. These protocols usually group several receives and save them at the same time to reduce the number of accesses to stable storage. The "optimism" comes from the assumption that data is logged before a failure occurs. If this assumption is not verified, processes can experience a *bounded domino effect*, and several processes might have to roll back not only to the last checkpoint but also to previous ones.

1.3 Contributions of the Thesis

1.3.1 Optimizing Coordinated Protocols

The first problem addressed by this thesis is the design of a coordinated checkpoint protocol with overheads comparable to an uncoordinated protocol, and at the same time with weak assumptions about the underlying support system. Uncoordinated protocols, as was mentioned in the previous section, attempt to reduce the failure-free overheads to the minimum since they only have to store of the processes' states. However, this comes with the cost that several checkpoints have to be kept per process, a garbage collector has to be run periodically, and the domino effect is always a possibility. Coordinated protocols do not suffer from any of these problems, but require some coordination during checkpoint creation. This thesis describes two variations of a coordinated checkpoint protocol that uses time to avoid all types of direct coordination. The time-based protocol does not have to exchange coordination messages, does not need to tag the application messages, and only accesses the stable storage when the checkpoints are saved.

1.3.2 Checkpoint Protocol for Mobile Environments

Mobile computing allows ubiquitous and continuous access to computing resources while the users travel or work at a client's site. The flexibility introduced by mobile computing brings new challenges to the area of fault tolerance. Failures that were rare with fixed hosts become common, and host disconnection makes fault detection and message coordination difficult. The thesis describes a new coordinated checkpoint protocol that is well adapted to mobile environments. The protocol uses time to indirectly coordinate the creation of new global states, and it uses two different types of checkpoints to adapt its behavior to the current network characteristics, and to trade off performance with recovery time.

1.3.3 Fault Detection Using the Socket Errors

Traditionally, two techniques have been used to detect failures in distributed systems: watchdogs [61, 62] and polling [63]. In both techniques a tradeoff has to be made between performance and speed of fault detection. By decreasing the timeout interval, faults are discovered faster but with higher overheads, because more communication is necessary, and the tested process is more frequently disturbed. The thesis describes a complementary mechanism that uses the error codes returned by the stream sockets. Since these errors are generated automatically when there is communication with a failed process, the mechanism does not incur in any failure-free overheads. However, for some types of faults, detection can only be attained if the surviving processes use certain communication operations. Nevertheless, our results show that in most cases, faults can be found using only the errors from the socket layer.

1.3.4 A Tool for Implementation and Evaluation of Checkpoint Protocols

During the past 20 years a large number of checkpointing and roll back recovery protocols have been proposed for distributed systems [64]. Most of these protocols, however, were never implemented or tested. The thesis 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 - Recoverable Network of Workstations* 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 *MPI - Message Passing Interface* [65].

Chapter 2

Time-Based Checkpointing

Coordinated checkpointing and rollback recovery is a technique for fault tolerance in distributed systems [12, 16, 22]. During the failure-free periods, the coordinated checkpoint protocol stores periodically in stable storage the state of the application and the messages that are in-transit in the network. When a failure occurs, recovery involves rolling back the application to the last available state, and then restarting its execution. One of the main advantages of coordinated checkpointing, when compared with other checkpointing methods, is its simplicity. For instance, log-based checkpoint protocols have to save the reception order and the contents of all messages, and then have to garbage collect this information [5, 35, 50, 52, 59, 60]. Other positive aspects of coordinated checkpointing are that concurrent failures can be tolerated, and applications do not have to execute in a piece-wise deterministic manner [45].

Arguments commonly used against coordinated checkpointing have been the overhead and lack of independence of checkpoint creation. Traditionally, two types of coordination have been employed: extra message exchanges and message tagging. A coordinated protocol sends extra messages, for example, to ensure that all processes start to save their states [2, 23]. One reason for using message tagging is to minimize the number of processes that have to roll back after a failure [27]. Time-based coordinated protocols do not need to exchange extra messages, because they assume synchronized clocks and bounded message deliveries [10, 24]. In these protocols, processes initiate their checkpoints whenever a local timer expires, and tag the application messages to identify in-transit messages.

This chapter presents two variations of a new coordinated protocol that uses time to avoid all types of direct coordination [66, 67]. Although processes save their states independently, the protocol is able to store application checkpoints that are consistent and recoverable. The protocol also does a minimal number of accesses to stable storage, since it only executes one write per process in each application checkpoint. The protocol is specified using two procedures; one that saves the process state whenever a local timer expires, and another that keeps timers approximately synchronized. The second procedure is also used to detect failures in the processor clocks that might lead to incorrect behavior of the protocol.

The chapter ends by describing a second coordinated protocol that is well adapted to mobile environments [68, 69]. This protocol also utilizes time to indirectly coordinate the processes, however due to the specific characteristic of mobile environments, it uses other techniques to ensure that application checkpoints are consistent and recoverable. Since mobile hosts might be connected to different kinds of networks as they roam between places, the protocol adapts its behavior by creating two different types of process checkpoints; it saves checkpoints in the local host or in a remote stable storage. Locally stored checkpoints are used in most cases if the network bandwidth is small, to reduce overheads and network consumption. Sometimes checkpoints also have to be sent to stable store to guarantee that permanent failures can be tolerated.

2.1 Distributed System Model

An application is executed by a set of processes running on several nodes. A node contains a processing unit, local memory and a local hardware clock. Clocks do *not* have to be synchronized, however, they drift from real time with a maximum drift rate, ρ. This assumption requires that local clocks have at most an error of $\rho(e-s)$ time units at the end of the real-time interval $[s,e]$. Correct clocks obey to the following requirement:

Bounded Drift : If $C_n(t)$ is time returned by the local clock of node n at real-time t, then $\forall s, e$:

$s < e;\ \ (1-\rho)(e-s) \leq C_n(e) - C_n(s) \leq (1+\rho)(e-s).$

Processes can set timers to schedule future executions of operations. A timer started at real time t with an initial value T expires at time $C_n(v) \geq C_n(u) = C_n(t) + T$. It is assumed that a process does not execute the program during the interval $[u, v]$, which is called the *scheduling delay*. This interval is limited by a constant δ, and it accounts for the situations when the operating system suspends a process until it finishes another task. The bounded drift equation can be used to derive a maximum deviation that separates two timers initiated in different nodes. If two timers are started in two nodes exactly at the same time with the same initial value T, then they will expire at most $(2\rho T + \delta - \delta\rho^2)/(1-\rho^2)$ time units from each other. This value will be approximated by $2\rho T + \delta$ because drift rates are very small. For most quartz clocks available in commodity workstations the drift rates are in the order of 10^{-5} or 10^{-6}, and for high precision clocks ρ is in the order of 10^{-7} or 10^{-8} [70]. The value of the scheduling delay is normally in order of tens of milliseconds, however, we will only require that δ is smaller than the checkpoint period.

Processes communicate by exchanging messages. Messages may be lost while in transit, arrive out of order, be duplicated, or may be discarded because of insufficient buffering space. To guarantee in-order reliable message delivery, the application utilizes a communication protocol. Typically, the communication protocol keeps a copy of each sent message until an acknowledgment arrives. If the acknowledgment is lost or delayed, it retransmits the messages after a timeout interval. After retransmitting the message a number of times, if no acknowledgment is received, the protocol assumes that the remote process has failed and returns an error that can be used to initiate recovery (see next chapter). The communication protocol associates with each message a sequence number. When a message is received, it compares the associated sequence number with the expected sequence number to identify communication problems (e.g., duplicates). Messages take less than td_{max} time units to be delivered, otherwise it is considered that there was a failure. An upper bound on this value is equal to the maximum number of times a message is retransmitted multiplied by the timeout interval.

It is assumed that it is possible to save, at checkpoint time, a copy of the messages that have not yet been acknowledged and the current values of the send and receive sequence number counters. This information can be easily obtained if the application is built on top of unreliable communication channels (e.g., sockets over UDP). In this case, the reliable communication protocol is implemented as part of the application, which means that the unacknowledged messages and sequence counters are automatically stored when processes create their checkpoints. On the other hand, if the application uses reliable communication channels (e.g., sockets over TCP), the checkpoint protocol has to be able to extract the necessary information. In this case, the checkpoint protocol might have to be implemented together with the communication protocol (normally in the operating system).

Processes can suffer from crash and performance failures. However, processes can store data in stable storage, and this data can be obtained after a failure by the correct processes.

2.2 No-Logging and Logging at the Sender Protocols

Time-based protocols are implemented using two procedures, one that saves the processes' states, and another that keeps the checkpoint timers approximately synchronized. This section starts by describing the checkpoint creation procedure, and then it presents the re-synchronization procedure.

2.2.1 Create Checkpoint Procedure

The protocols use time to indirectly coordinate the creation of new global states. Processes determine the instants when they should save their states using a local timer, without needing to exchange messages. By keeping the timers approximately synchronized, the protocols are able to guarantee that the independently created checkpoints form a consistent and recoverable global state. In this section, it is assumed that timers are initially synchronized in such a way that they expire at most D seconds apart.

2.2.1.1 Consistency Property

A coordinated protocol has to guarantee that processes create their checkpoints in such a way that the consistency property is verified. This property is ensured if no process, after storing its checkpoint, sends a message that is read before the receiver saves its state. If timers were exactly synchronized, all processes would initiate their checkpoints at exactly the same time, and no consistency problems could occur. In a distributed system, however, timers are never perfectly synchronized, and they expire within an interval DEV. Therefore, one way to avoid consistency violations consists in guaranteeing that no messages are sent during DEV. If the checkpoint period is T, and n checkpoints have been created since the last timer re-synchronization, then the interval DEV is bounded by a *maximum deviation* $MD = D + 2\rho nT + \delta \geq DEV$. The value of MD is composed of three factors: the initial deviation among the timers D; the drift of the clocks $2\rho nT$; and the scheduling delay δ. Processes, however, do not know if their timers expire at the beginning or at the end of the interval MD. A conservative approach to address this problem consists in preventing message sends during MD time units after the timers expire. This is expressed with the following requirement,

CR1 : The consistency property is verified if, after the timer stops, a process does not send messages during $MD = D + 2\rho nT + \delta$ time units.

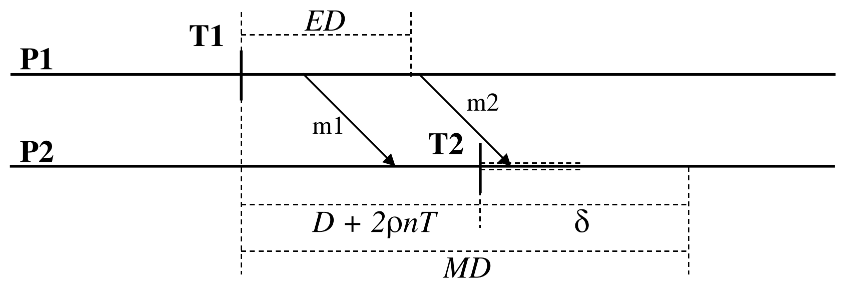


Figure 2.1: Message *m1* creates a consistency problem, and message *m2* does not violate the consistency property.

In practice, the previous requirement is too conservative for two reasons: first, since messages take a minimum time to be delivered (td_{min}), processes can start to send messages earlier without causing consistency violations. A message transmitted at the end of the interval $MD - td_{min}$ will arrive later than all timers expired, and consequently after the receiver process has initiated its checkpoint. Second, since it is assumed that processes do not execute during the scheduling delay, messages that arrive during this period will not be read. Therefore, it is possible to define a tighter interval where messages should not be transmitted. This interval, called *effective deviation*, is equal to $ED = D + 2\rho nT - td_{min}$. The following requirement is implemented by the time-based protocols,

CR2 : The consistency property is verified if processes do not send messages during $ED = D + 2\rho nT - td_{min}$ time units after their timer stops.

Figure 2.1 shows the execution of two processes with checkpoint timers expiring at times *T1* and *T2*. Message *m1* is sent during the critical interval ED and violates the consistency property. The protocol guarantees that no such message sends can occur.

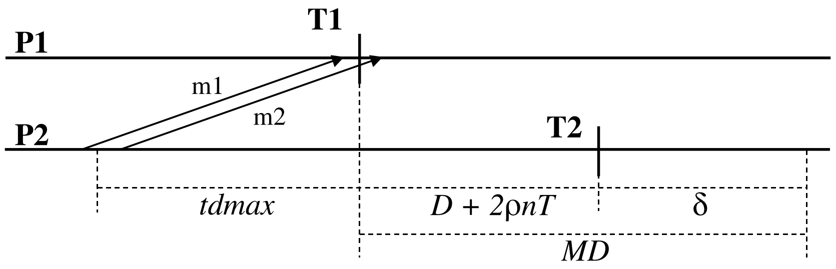


Figure 2.2: No-logging protocol: In-transit messages are prevented from occurring by disallowing message sends during $td_{max} + D + 2\rho nT$ seconds before checkpoint time. In this protocol message *m2* would not be sent.

2.2.1.2 Recoverability Property

No-Logging Protocol

A coordinated protocol must be able to reconstruct all messages that are in-transit at checkpoint time. These are messages that were sent before a process saved its state, and are read after the receiver stored its checkpoint. One solution for the recoverability problem consists in avoiding the creation of messages that might become in-transit. If in-transit messages can not occur, the recoverability property is automatically satisfied. This approach simplifies the implementation of the protocol during both the failure-free periods and recovery, since the protocol does not need to log any messages or to re-send or re-read messages. It also avoids the overhead of storing the in-transit messages.

The example from Figure 2.2 can be used to illustrate how in-transit messages can be prevented from occurring. In the figure, process $P2$ sends two messages, $m1$ and $m2$. Message $m1$ does not have to be stored since it arrives before $T1$, but message $m2$ would have to be logged if process $P2$ was allowed to send it. Therefore, a global state will satisfy the recoverability property if no process is allowed to send messages td_{max} time units *before* the first timer is scheduled to finish. It should be noted that this condition does not prohibit processes from continuing their executions

```
   procedure stopSMesg()
1     $stopSendMesg$ = TRUE;
2     setTimer($stopSMesg, ckpTime + T - (D + 2\rho(N+1)T + td_{max})$);

   procedure createCkp()
3     saveProcessState();
4     $N$ = $N + 1$;
5     $ckptTime$ = $ckpTime + T$;
6     setTimer($createCkp, ckpTime$);
7     if $((D + 2\rho(N-1)T - t_{dmin}) >$ (getTime() $- (ckpTime - T)))$
8        requestResyncTimers();
9     $stopSendMesg$ = FALSE;
10    sendQueuedMessages();
```

Figure 2.3: Checkpoint creation procedure for the no-logging protocol.

until they start to save their checkpoints. Whenever a process attempts to send a message, the protocol queues the message and then lets the process continue with the computation. The actual transmission of the message is done at the end of the critical interval.

Since timers do not terminate at exactly the same instant, processes need to prevent message sends during an extra interval, corresponding to the maximum time that separates two scheduled timers. This interval is equal to $D + 2\rho nT$. The time-based protocol with no logging satisfies the following requirement,

RR1 : The recoverability property is verified if each process k, with a timer scheduled to end at T_k, does not transmit messages during $D + 2\rho nT + td_{max}$ time units before T_k.

The checkpoint creation procedure for the no-logging protocol can be implemented using the code from Figure 2.3. The procedure uses two timers, one that expires $D + 2\rho nT + td_{max}$ seconds before the checkpoint, and another that expires at checkpoint time. Whenever the first timer terminates, it calls the `stopSMesg` function. This function sets a flag indicating that messages should be queued, and then resets the timer (Lines 1-2). The function `createCkp` is executed when the second timer stops. It saves the process state, increments the checkpoint time with the checkpoint period T, and resets the timer (Lines 3-6). Variable N keeps the number of checkpoints created since the last re-synchronization, and variable $ckpTime$ holds the current checkpoint time. Next, `createCkp` tests if ED seconds have passed since the checkpoint time (Line 7). If the condition is not satisfied, this means that the term $2\rho nT$ has grown too large, and that timers need to be re-synchronized. The frequency of re-synchronization, however, is usually small. For reasonably good clocks ($\rho = 10^{-6}$), checkpoint sizes of 1 MBytes, and stable storage with bandwidth of 1 MBytes/s, the re-synchronization procedure only needs to be run once every 5 days. Before returning, `createCkp` resets the flag and sends the messages that were queued (Lines 9-10).

Logging-at-Sender Protocol

Typical communication protocols, either implemented as part of the application or in the operating system, ensure reliable message deliveries by keeping a copy of each sent message until an acknowledgment arrives [71]. Lost or corrupted messages are recovered by re-transmitting the messages if the acknowledgment is not received within a given interval, and duplicate messages are detected using sequence numbers. An in-transit message is a message that was sent before the sender process saved its state, and is received after the receiver process created its checkpoint. Therefore, unless acknowledgments violate the consistency property, a copy of each in-transit message exists in the sender machine at checkpoint time.

A checkpoint protocol can ensure that all in-transit messages are logged by including the unacknowledged messages in the sender checkpoint. During recovery, the sender re-transmits the logged messages, avoiding message losses. Logging at the sender, however, might save a few other messages besides the in-transit messages; a process can receive a message before its checkpoint, but the acknowledgment might only arrive after the sender has started to save its state (see message *m1* in Figure 2.7). These extra messages have to be detected and discarded during re-

```
  procedure createCkp()
1    saveProcessState();
2    $N$ = $N + 1$;
3    $ckpTime$ = $ckpTime + T$;
4    setTimer($createCkp$, $ckpTime$);
5    if $((D + 2\rho(N - 1)T - td_{min}) >$ (getTime() $- (ckpTime - T)))$
6       requestResyncTimers();
```

Figure 2.4: Checkpoint creation procedure for the logging-at-sender protocol.

covery, otherwise they are read twice. This can be accomplished by saving, in the checkpoints, the value of the send and receive sequence number counters, which are used to detect duplicate messages due to re-transmissions. By resetting these counters during recovery, the extra messages will be considered duplicated messages and will be removed automatically.

The code from Figure 2.4 can be used to implement the checkpoint creation procedure. The procedure starts by saving the process state and by preparing the timer for the next checkpoint (Lines 1-4). The function `saveProcessState()`, stores the process state in stable storage, including all unacknowledged messages and the send and receive sequence number counters. Next, it verifies whether the critical interval ED has grown larger than the time necessary to save the process checkpoint (Line 5). In the affirmative case, function `requestResyncTimers()` sends a request for timer synchronization, and then executes the re-synchronization procedure.

2.2.2 Re-synchronization Procedure

Time-based coordinated protocols utilize the resynchronization procedure to keep the checkpoint timers approximately synchronized. In addition, the procedure also detects failures in the clocks that might result in incorrect behavior of the protocols.

The resynchronization procedure selects one of the processes to act as coordinator. The coordinator adjusts the other processes' timers in such away that they expire at most D time units apart

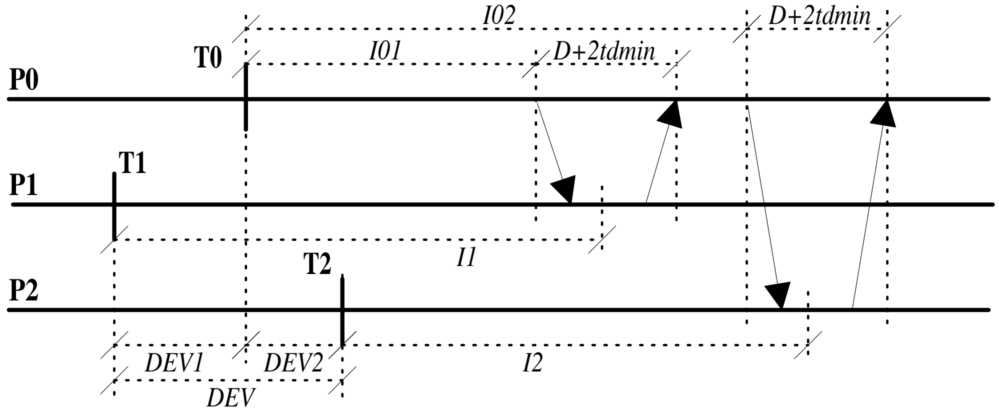


Figure 2.5: Estimate the value of DEV.

from its timer (if ρ is zero). Since clocks might not be synchronized, it can not ask another process to set timer to a given absolute time. Instead, it uses a simple iterative method; first, the coordinator sends to the other process the interval until the next checkpoint, $interval_c = ckpTime_c - currentTime_c$. Second, the process saves the $interval_c$ and the current time, $currentTime_p$, and sends an acknowledgment back to the coordinator. Third, if the time that separates the transmission of $interval_c$ and the reception of the acknowledgment is smaller than $D + 2td_{min}$ time units, the coordinator sends an END synchronization message. Otherwise, it returns to the first step and repeats the same operations. When the remote process receives the END message, it resets the timer with the new checkpoint time $ckpTime_p = currentTime_p + interval_c - td_{min}$.

The resynchronization method has to be done in an iterative way because delivery times are not constant. In some cases the first message of the coordinator can take more time to arrive than in others. Therefore, the coordinator can only guarantee a deviation smaller than D if it rejects all iterations where the round trip time is larger than $D + 2td_{min}$. Nevertheless, messages usually have short delivery times, which allows D to be kept small. The constant D should be made equal to a multiple of the round trip-time of a small message. As a rule of thumb, we normally set D to 10 ms, which works well for a 10/100 Mbit/s Ethernet and 155 Mbit/s ATM networks.

The time-based protocols are only able to ensure that global states are consistent if timers expire within the expected deviation, $(D+2\rho nT) = DEV_{expected}$. For this reason, it is important to detect clock failures. While the coordinator adjusts the timers, it estimates the maximum deviation between two timers during the last checkpoint creation, DEV. If DEV is larger than $DEV_{expected}$, then at least one of the clocks is not working correctly.

The example from Figure 2.5 is used to illustrate the calculation of DEV. From the three processes represented in the figure, $P0$ is the coordinator. $T0$, $T1$, and $T2$ were the times when the timers were scheduled to end during the last checkpoints. The coordinator estimates the value of $DEV1$ while it resynchronizes the timer of process $P1$. Before sending the $interval_c$, the coordinator calculates the interval $I01$. After receiving $interval_c$, process $P1$ computes the value of $I1$ and returns it to the coordinator in the acknowledgment. If the coordinator accepts the resynchronization, $DEV1$ can be estimated in the following way:

$$|\Delta_1| - D \leq DEV1 \leq |\Delta_1| + D, \; with \; \Delta_1 = (I01 + td_{min}) - I1$$

Using an equivalent method, the coordinator obtains the estimate of $DEV2$. Next, it computes DEV using the estimates $DEV1$ and $DEV2$. Two cases have to be considered: both timers expired before or after $T0$, or one of the timers expired before and another after $T0$. The following bounds can be derived for DEV,

$$\begin{cases} max(|\Delta_1|, |\Delta_2|) - D \leq DEV \leq max(|\Delta_1|, |\Delta_2|) + D & if \; ((\Delta_1 < 0) and (\Delta_2 < 0)) \\ & or \; ((\Delta_1 > 0) and (\Delta_2 > 0)) \\ |\Delta_1| + |\Delta_2| - 2D \leq DEV \leq |\Delta_1| + |\Delta_2| + 2D & otherwise \end{cases}$$

The coordinator can make the following conclusions by comparing the expected deviation with the estimated maximum and minimum deviations:

$$\begin{cases} DEV_{expected} > DEV_{max} & OK \\ DEV_{expected} < DEV_{min} & Failure \\ DEV_{min} \leq DEV_{expected} \leq DEV_{max} & No\ conclusion \end{cases}$$

Since DEV is not exactly determined, there is a window of uncertainty of size $4D$ time units where conclusions can not be made about the failure of the processor clocks. A pessimistic or optimistic approach can be used to address this problem; with the pessimistic approach, the protocol assumes that there was a failure if $ED_{expected} \leq DEV_{max}$. The optimistic approach uses the condition $ED_{expected} < DEV_{min}$ to detect clock failures. If a clock failure is found then a system warning can be issued and the protocol can increase the assumed value for the drift rate (e.g., $\rho_{new} = \rho_{old} * 2$). If, after a few resynchronizations, the value of ρ does not converge to the *real* drift rate, then another warning must be sent saying that the protocol is unable to recover from the clock failure.

The resynchronization procedure is implemented using the code from Figure 2.6. The code adjusts the checkpoint timers and detects the clock failures. For this reason, it should only be utilized after processes have created at least one checkpoint. The first synchronization can be done using an equivalent procedure with the lines corresponding to the fault detection removed. The coordinator executes the `while` loop to synchronize the remote timers (Lines 2-3). It begins by sending the $interval_c$, next waits for the acknowledgment, and then sees if the round-trip time is smaller than $D + 2td_{min}$ (Lines 4-7). In the affirmative case, it breaks the loop and begins to set another timer (Line 11). The END message is sent when all timers have been adjusted (Line 14). The code extends the fault detection method described previously to a number of processes larger than three. The coordinator estimates the two deviations by receiving the interval from the other process (Lines 6, and 8-10). The pessimistic approach is used to determine if there was a clock

```
procedure ResynchronizeTimers()
  Coordinator:
    $\Delta_a$ = $\Delta_b$ = 0;
    for each($p \in Processes$) do
      while (TRUE) {
        $currentTime_c$ = getTime();
        send($p, ckpTime - currentTime_c$);
        receive($p, I$);
        if ((getTime() $- currentTime_c) < (D + 2 * td_{min})$) {
          $I0$ = $(currentTime_c + td_{min}) - (ckpTime - T)$;
          if ($I > I0$) $\Delta_b$ = max($\Delta_b, I - I0$);
          else $\Delta_a$ = max($\Delta_a, I0 - I$);
          break;
        }
      }
    broadcast($-1$);
    if$((\Delta_a + \Delta_b + 2D) \geq (D + 2\rho(N-1)T))$ Error();
    $N$ = 1;

  Process p:
    receive($coord, loop$);
    do {
      $currentTime_p$ = getTime();
      $interval_c$ = $loop$
      send($coord, currentTime_p - (ckpTime - T)$);
      receive($coord, loop$);
    } while ($loop > 0$);
    $ckpTime$ = $currentTime_p + interval_c - td_{min}$;
    setTimer($createCkp, ckpTime$);
    $N$ = 1;
```

Figure 2.6: Timer resynchronization procedure.

failure (Line 15). The other processes receive $interval_c$ (Lines 17-23), and then they reset the timers (Lines 24-25).

2.2.3 An Example

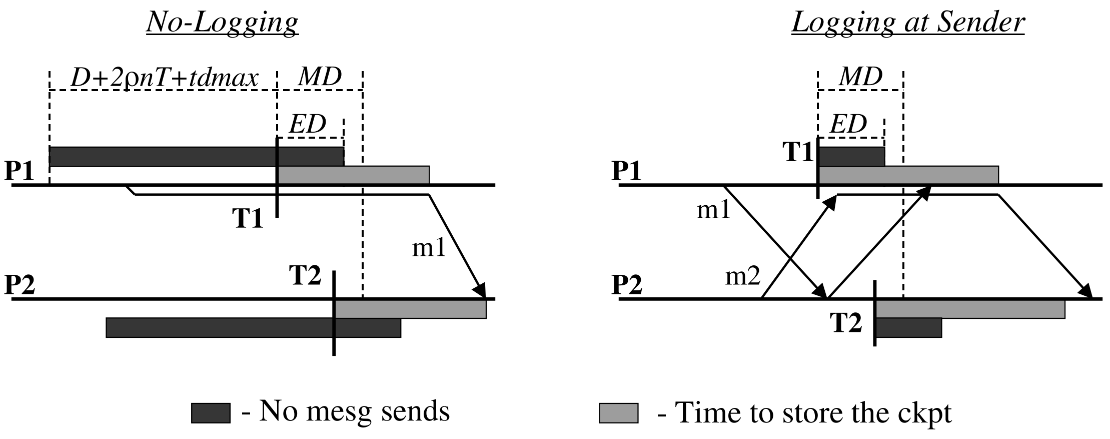
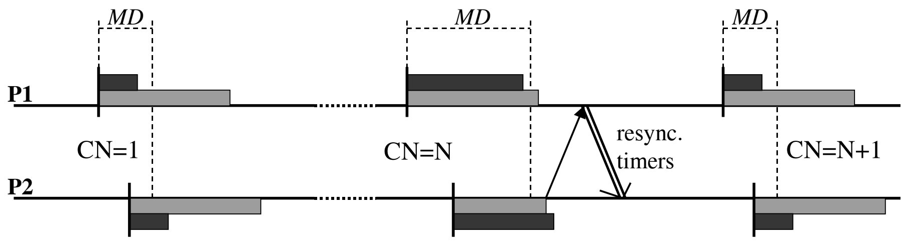


Figure 2.7: Example with the creation of the i checkpoint.

The example from Figure 2.7 shows the execution of the protocol with no-logging and with logging at the sender during the creation of an application checkpoint. In both cases, the two processes begin to save their states at different instants, because checkpoint timers are not exactly synchronized. However, since timers stop at most *MD* seconds apart, the two checkpoints are separated by a small interval. On the execution represented on the left part of the figure, the protocol with no logging postpones the send of message *m1*. When process *P1* attempts to send $m1$, the flag $stopSendMesg$ is set, and the message is queued. The message is sent when process $P1$ finishes saving its checkpoint.

The protocol with logging at the sender saves both messages represented in the right part figure. Message *m1* is sent by process *P1* and is received by process *P2* before the creation of the checkpoints, which means that *m1* is not an in-transit message. However, the acknowledgment of *m1* only arrives to process *P1* after the timer expires. At *T1*, process *P1* does not know if *m1* is an in-transit message or not, so it includes the message in the checkpoint. Later, if there is a failure, processes will have to roll back to the stored checkpoints, and process *P1* will re-send *m1*. Process *P2* will detect that *m1* is a duplicate using the receive sequence number counter that was included in its checkpoint, and will remove the message. (Although the acknowledgment of *m1* is an in-transit message, the protocol does not need to log acknowledgments.) Message *m2* is an in-transit message, and is included in the checkpoint of process *P2*. Since process *P1* is storing its state when *m2* arrives, the acknowledgment is sent after the checkpoint is completed. To avoid violations to the consistency property, the acknowledgment only had to be postponed until the end of the critical interval.



■ - Effective deviation (ED) ▨ - Time to store ckpt

Figure 2.8: Example with the execution of the protocol through several checkpoints.

The example from Figure 2.8 illustrates the behavior of the protocols throughout a long period of time. When the application starts, timers are well synchronized, and ED is much smaller than the time to save a checkpoint. As the application continues its execution, the value of *ED* increases because of the clock drifts. On the *N*'s checkpoint, the value of *ED* becomes larger than the time to store the checkpoint of process *P2*. After saving its state, process *P2* sends to process *P1* a request for timers' re-synchronization (in the example, *P1* is the coordinator). When processes create their next checkpoints, the value *ED* is again small.

2.2.4 User Initiated Checkpoints

Time-based protocols save new global states periodically, whenever the timers reach the checkpoint period. Sometimes it is necessary to create extra global states either because the user wants to save its work or because the application needs to guarantee that a specific task will not be rolled back. As examples, the user might want to swap the application to disk so that it can be restarted in a new group of machines, or the application might want to prevent the roll back of interactions with the outside world. The creation of extra global states requires the exchange of messages among the processes; at least a broadcast message has to be sent to announce the new checkpoint, and then an acknowledgment has to be returned by each process. The acknowledgments are used to detect delivery problems, and if they are not received within a given timeout, the broadcast has to be retransmitted.

Time-based protocols piggyback in the broadcast message the information necessary to schedule an extra checkpoint. The *initiator*, the process called by the user or that executed the operation `checkpoint()`, sets the new checkpoint time, $extraCkpTime_i = currentTime_i + \mu$. Then, it sends to the other processes the interval between the normal and the extra checkpoint, $interval_i = ckpTime_i - extraCkpTime_i$. When a process receives the message, it schedules the extra checkpoint for the following time, $extraCkpTime_p = ckpTime_p - interval_i$. If for some reason the extra checkpoint can not be created (e.g., the broadcast arrives after $extraCkpTime_p$), the process returns to the initiator an error message indicating that distinct time should be used. As a result, the initiator has to select a later time and has to repeat the whole process again.

The value of μ should be chosen carefully. It should be sufficiently large to let the broadcast and acknowledgments to be delivered before $extraCkpTime$, otherwise several checkpoints might be aborted. On the other hand, the user should not have to wait too much time for the checkpoint, which means that μ has to be smaller than a few hundreds of milliseconds. In typical networks, like Ethernet or ATM, round trip times are in the order of 1 - 2 ms. Therefore, the value of μ can be set to 100 ms.

2.2.5 Garbage Collection and Recovery

Time-based protocols save regularly global states of the application, containing the processes' checkpoints and the in-transit messages. This information should be garbage collected whenever it stops being useful for recovery to avoid exhausting the stable storage. Since protocols always recover the application to the last *complete* available state, this means that a global state can be deleted as soon as a new one is created.

The entity responsible for the garbage collection can be, for instance, a special process in the stable storage or the processes executing the application. In the first possibility, the process would have to check periodically if new checkpoints had been saved and then remove the old checkpoints. The second solution is not as simple as the first one because an application process does not know when a global state is completely stored; it only knows when its checkpoint is finished. In any case, the creation of a global state is upper bounded by $tc_{max} + MD$ time units, where tc_{max} is the maximum time to save a process checkpoint. Therefore, a process can delete its old checkpoint whenever this interval has passed since the last timer expiration.

2.2.6 Related Work

Time plays a fundamental part in any fault tolerant real-time system, like MARS [72] and DELTA-4 [73]. In the area of checkpoint-based recovery protocols it has been used to avoid extra exchanges of messages in coordinated protocols [10, 24].

Tong et al. [24] proposed the first time-based protocol. This protocol assumes loosely synchronized processor clocks and relatively small message delivery times. Processes use a positive acknowledgment retransmission scheme to be able to communicate reliably. A process starts to save its state whenever the local clock reaches a multiple of the checkpoint period. The checkpoint of a process includes all messages that have been sent and have not been acknowledged. The protocol adds to each application message and acknowledgment a checkpoint number to detect in-transit messages, and then it stores them in stable storage as they are received. Processor clocks are resynchronized periodically.

Cristian and Jahanian [10] also proposed a protocol that uses time to initiate the creation of the checkpoints. This protocol requires stricter assumptions about the synchronization of the clocks and assumes that message delivery times are small with high probability. During the creation of a new checkpoint, the protocol defines a critical interval during which all in-transit messages have to arrive. If an in-transit message takes longer than the bounded delivery time to be transmitted, it can be received later than the critical interval. In this case, the protocol considers that there was a communication failure, and the application has to roll back. After saving its state, each process broadcasts a special message so that all bounded delivery violations can be detected. To identify in-transit messages, the protocol tags each message with a checkpoint number and the time of the local clock. In-transit messages are logged as they are received and are written to stable storage at the end of the critical interval.

The protocols described in this chapter use synchronized checkpoint timers instead of synchronized clocks. This characteristic is important in systems where, for security reasons, the applications are not allowed to change the value of the processor clocks. Contrary to the previous time-based protocols, the protocols do not need to tag the application's messages with any information, and they avoid the extra accesses to stable storage that were necessary to save the in-transit messages. The protocols are also able to adapt the frequency of timer re-synchronization to the application, and to detect clock failures that might lead to incorrect behavior.

The no-logging protocol assumes small and bounded message delivery times. It prevents the existence of in-transit messages by disallowing message sends during an interval before the checkpoint creation. The logging at the sender protocol has to save the in-transit messages, but it does not need the assumption that message delivery times are small. This assumption is common to all previous time-based protocols, and is the most important limitation to the applicability of these protocols. The protocol also has a checkpoint latency completely independent of message delivery times, which enables it to function correctly even with small checkpoint periods. To our knowledge, this is the first coordinated protocol with this characteristic.

2.3 Adaptive Protocol

This section describes a coordinated checkpoint protocol that is well adapted to the characteristics of mobile environments [68, 69]. The protocol uses time to be able to store consistent recoverable states of the application without having to exchange messages. It creates two different types of process checkpoints to adapt to the current characteristics of the network and to provide differentiated recoveries. Process checkpoints are saved in stable storage or locally in the hosts. Locally stored checkpoints do not consume network bandwidth and take much less time to be created. However, they can be lost due to permanent failures in the mobile hosts. During the application execution, the protocol keeps a global state in stable storage and has another global state that is dispersed through the mobile hosts and stable storage. The first global state is used to recover permanent failures, and the second, transient failures.

2.3.1 Unique Aspects of Mobile Environments

Mobile hosts have several characteristics that make them different from fixed hosts. Checkpoint protocols designed for mobile environments should consider these distinguishing features in their definition. Otherwise, they will incur high overheads, or they will simply not work correctly.

1. *Location is not fixed*: As the user moves from one place to another, the location of the mobile host in the network changes. The checkpoint protocol can store the processes' states in a well known site or in a computer near the current location of the mobile host. In the second case, the checkpoint protocol has to keep track of the places where processes' states were saved.

2. *Disconnection*: A mobile host can become disconnected. While disconnected, the mobile host is not able to send or receive any messages. Protocols that need to exchange messages will not work correctly in this situation. During disconnection, the checkpoint protocol should provide a local recovery mechanism that allows the mobile host to recover from its own failures.

3. *Batteries store a limited amount of power*: The mobile host is often powered by batteries. Network transmissions and disk accesses are two of the most important sources of power consumption [74]. To minimize power consumption, the checkpoint protocol should reduce the amount of information that it adds to the application's messages, and it should avoid sending extra messages. The protocol should also make a small number of accesses to disk.

4. *Network characteristics are not constant*: The various wireless technologies have completely different qualities of service [75, 76]. For instance, a radio frequency LAN can have bandwidths between 2 and 20 Mbps, but a wide-area LAN using cellular digital packet data (CDPD) may have a bandwidth of 19.2 Kbps. Other different characteristics are cost, packet loss rates, and latency. The checkpoint protocol should adapt its behavior to the current network.

5. *Different types of failures*: Mobile host failures can be separated into two different categories. The first one includes all failures that can not be repaired; for example, the mobile host falls and breaks, or is lost or stolen. The second category contains the failures that do not permanently damage the mobile host; for example, the battery is discharged and the memory

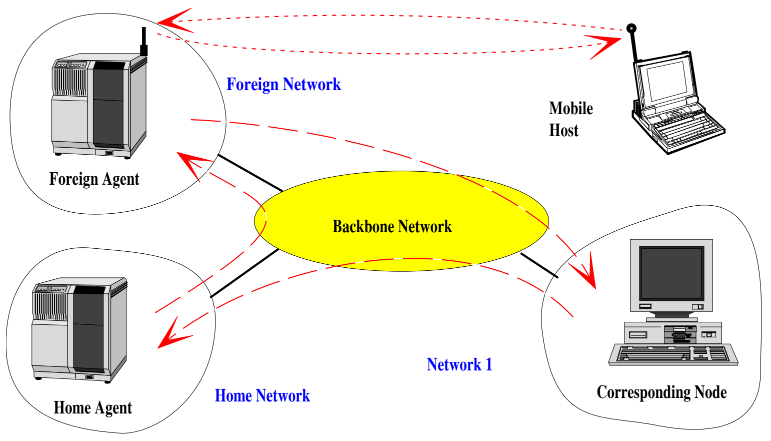


Figure 2.9: Mobile environment.

contents are lost, or the operating system crashes. The first type of failures will be referred to as *hard failures*, and the second type as *soft failures*. The protocol should provide different mechanisms to tolerate the two types of failures.

2.3.2 Mobile Environment and Terminology

The terminology that is going to be used is based on the internet draft for mobile IP [77]. The system contains both fixed and mobile hosts interconnected by a backbone network (see Figure 2.9). A mobile host uses a wireless interface to maintain network connections while it moves, and it is identified by a long-term address. The address also serves to localize the mobile host's *home network*. While at home, the mobile host receives the packets as a normal fixed host. When it moves to another network, the mobile host relies on the services of a *foreign agent* to be able to communicate. Typically, the foreign agent has a wireless interface and is able to forward packets to and from the mobile host (the mobile host can also be directly connected to the wired network). The geographical cover area of the wireless interface is called the *cell*. Disconnection occurs when the mobile host moves outside the range of all the cells. The mobile host can request the services of another foreign agent if the current one fails. The *home agent* represents the mobile host when it is away from the home network. The home agent intercepts the packets directed to the mobile host and forwards them to the current foreign agent[1]. The mobile host informs the home node about foreign agent changes.

The example from Figure 2.9 can be used to illustrate the communication between the mobile host and another host. The *corresponding host* sends packets to the long-term address of the mobile host. The backbone network routs these packets to the home network. The routing protocol is the same as for packets that are sent to a fixed host. On the home network, the home agent intercepts the packets and forwards them to the foreign agent. The foreign agent transmits the packets through the wireless network to the mobile node. Packets sent by the mobile node do not have to be forwarded by the home agent. The foreign agent sends the mobile host's packets directly to the corresponding node.

2.3.3 Protocol Specification

The checkpoint protocol uses time to indirectly coordinate the creation of global states. Processes save their states periodically, whenever a local checkpoint timer expires. The protocol can set different checkpoint intervals to ensure distinct recovery times. Higher checkpoint intervals require on average larger periods of re-execution, but reduce the protocol's overheads.

The protocol creates two distinct types of checkpoints. The protocol uses checkpoints saved locally in the mobile host to tolerate soft failures, and it uses checkpoints stored in stable storage to recover hard failures. The first type of checkpoint is called *soft checkpoints*, and second type *hard checkpoints*. Soft checkpoints are necessarily less reliable than hard checkpoints, because they can be lost with hard failures. However, soft checkpoints cost much less than hard checkpoints because they are created locally, without any message exchanges. Hard checkpoints have to be sent through

[1]Mobile IP also allows messages to be directly forwarded to the mobile host, if it has a temporary address belonging to the foreign network.

the wireless link, and then through the backbone network, until they are stored in stable storage. A soft checkpoint with 1 Mbyte would take just a few seconds to be saved. On the other hand, a similar hard checkpoint would take more than 7 minutes just to be transmitted through a cellular link[2].

The protocol uses the distinct creation costs of the two checkpoint types to adapt its behavior to the quality of service of the current network. For different network configurations, the protocol saves a distinct number of soft checkpoints per hard checkpoint. If the network is slow, the protocol creates several soft checkpoints to avoid the network transmissions. By making a correct balance between soft and hard checkpoints, the protocol can keep its overheads approximately equal across the various types of networks.

For a given network configuration, the protocol can exchange hard failure recovery time with performance costs. Hard failures are recovered with global states containing only hard checkpoints. If the protocol creates hard checkpoints frequently, the amount of rollback due to hard failures is small on average. However, the performance of the protocol can be poor.

Soft checkpoints let the protocol continue to function correctly while the mobile host is disconnected. Conceptually, a disconnected mobile host can be viewed as a host connected to a network with no bandwidth. In this case, the number of soft checkpoints per hard checkpoint is set to infinity, which means that all processes' states are stored locally. The local checkpoints are used to recover the mobile host from soft failures.

Recovery from soft failures should be faster than from hard failures, since soft failures will occur more frequently. This can be accomplished with soft checkpoints because recovery can be made completely local, or at most, it will be necessary to send a rollback request message to the other processes.

[2]The actual transmission of the hard checkpoint can be done in the background, but it will always be a considerable burden to the user.

```
// $P_m$ = Sender's identifier
// $CN_m$ = Current checkpoint number of the sender
// $timeToCkp_m$ = Time interval until next checkpoint
// $mesg_m$ = Message contents
procedure receiveMesg($P_m, CN_m, timeToCkp_m, mesg_m$)
   if (($CN = CN_m$) and (timeToCkp() $>$ $timeToCkp_m$))
      resetTimer($timeToCkp_m$);
   else if ($CN < CN_m$) {      // orphan message
      createCkp();
      resetTimer($timeToCkp_m$);
   }
   deliverMesgToApplication($mesg_m$);
```

Figure 2.10: Message reception.

2.3.3.1 Checkpoint Creation Procedure

As with the protocols already described, the adaptive protocol creates checkpoints whenever the timers stop. However, it uses other techniques to ensure the consistency property and to re-synchronize the timers. The previous methods can not be utilized because, during certain periods of time, it might be impossible to adjust the timers and consequently, it is not viable in general to assume small timer deviations.

The protocol maintains a *checkpoint number* counter, CN, at each process to guarantee that the independently saved checkpoints verify the consistency property [29, 78]. The value of CN is incremented whenever the process creates a new checkpoint and is piggybacked in every message. The consistency property is ensured if no process receives a message with a CN_m larger than the current local CN. The process creates a new checkpoint before delivering the message to the application if CN_m is larger than the local CN (see Figure 2.10). The recoverability property is guaranteed by logging all messages that might become in-transit, as in the logging at the sender protocol.

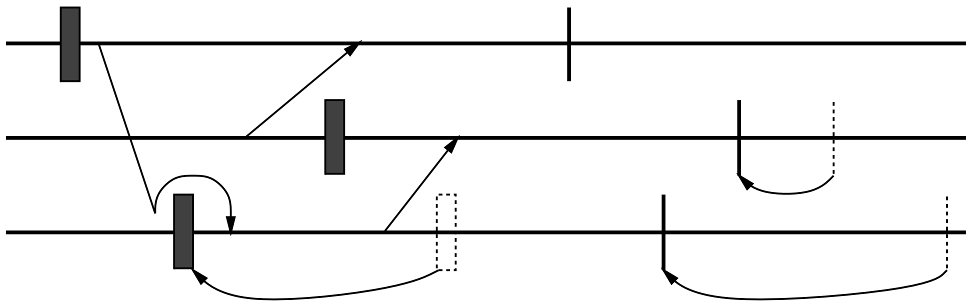


Figure 2.11: Time-based checkpointing.

The protocol uses two techniques to keep the timers roughly synchronized. When the application starts, it sets the timers in all processes with the checkpoint period. Since processes do not begin to execute in exactly the same instant, timers will end at different times. The protocol has a re-synchronization mechanism that adjusts timers during the application execution. Each process piggybacks in its messages the time interval until the next checkpoint. When a process receives a message, it compares its local interval with the one just received (see Figure 2.10). If the received interval is smaller, the process resets its timer with the received value. The re-synchronization mechanism also serves to solve other causes of timer inaccuracies, such as clock drifts.

The example from Figure 2.11 will be used to illustrate the execution of the protocol. This figure represents the execution of three processes (to simplify the figure, message acknowledgments are not represented). Processes create their checkpoints at different instants because timers are not synchronized. After saving its CN checkpoint, process $P1$ sends message $m1$. When $m1$ arrives, process $P3$ is still in its $CN - 1$ checkpoint interval. To avoid a consistency problem, $P3$ first creates its CN checkpoint and then delivers $m1$. $P3$ also resets the timer for the next checkpoint. Message $m2$ is an in-transit message that has not been acknowledged when process $P2$ saves its CN checkpoint. This message is logged in the checkpoint of $P2$. Message $m3$ is a normal message that indirectly re-synchronizes the timer of process $P2$. It is possible to observe in the figure

Table 2.1: Configuration Table for *maxSoft*.

Quality of Service	*maxSoft* Low	*maxSoft* High	Network Example
QoS > 10	1	2	ethernet, ATM
6 < QoS <= 10	2	8	radio, infrared
3 < QoS <= 6	4	32	cellular
0 < QoS <= 3	8	128	satellite
QoS = 0	∞	∞	disconnected

the effectiveness of the re-synchronization mechanism. Timers are better synchronized after the three messages have been received.

2.3.3.2 Soft vs. Hard Checkpoints

The protocol adapts its behavior to the current characteristics of the network. For instance, if the network has a poor quality of service, the protocol saves several soft checkpoints before it sends a hard checkpoint to stable storage. The quality of service of a network depends on several factors, e.g., bandwidth and packet loss rate. Its value can be estimated by the protocol, or it can be provided by the underlying communication layers [76, 79].

The number of soft checkpoints that are stored per hard checkpoint is called $maxSoft$, and it depends on the quality of service of the current network. The assignment of $maxSoft$ values to the different networks is made statically and saved in a table. Table 2.1 gives two examples of possible assignments. The minimal quality of service corresponds to a disconnected mobile host. In this case, $maxSoft$ is set to infinity, which means that only soft checkpoints are created. The low $maxSoft$ column represents an assignment where hard checkpoints are created frequently, which guarantees a small re-execution time after a hard failure. The high $maxSoft$ column corresponds to the opposite case.

Application processes run on hosts that might be connected to different networks, each corresponding to a distinct $maxSoft$ value. This means that a global state can include both soft and hard checkpoints. To ensure that recovery is always possible, the protocol has to keep at each moment a global state containing only hard checkpoints. This global state is used to recover the application from hard failures. Otherwise, the domino effect [38] can occur, and recovery might not be possible. The protocol guarantees that new hard global states are saved by correctly initializing the $maxSoft$ table. The process that creates hard checkpoints less frequently is the one running in the host connected to the network with worse quality of service (the disconnect case will be discussed in the next section). The protocol guarantees that a new hard global state is stored every time this process creates a hard checkpoint, by initializing the table in such a way that $maxSoft$ values are multiples of each other. For example, if processes $P1$ and $P2$ have $maxSoft$ values 4 and 8, this means that a new hard global state is stored every 8 checkpoints. Process $P1$ creates hard checkpoints whenever *CN* is equal to 4, 8, 12, 16, ..., and process $P2$ whenever *CN* is equal to 8, 16, ... The protocol also keeps the last global state that was stored (which can include soft checkpoints) to recover from soft failures.

The functions from Figure 2.12 are used to create a new checkpoint. Function `createCkp` is called to save a new process state. It starts by incrementing the CN, and then it resets the timer with the checkpoint period (Lines 1-2). Next, the function determines if the checkpoint should be saved locally or sent to stable storage (Lines 3-4). The function `storeState` stores locally the process state, and the function `sendCkpST` sends the process state to stable storage. The function `receiveCkp` is called by the stable storage to store newly arrived checkpoints. It first writes the received state to the disk, and then updates the local checkpoint counter (Lines 5-6). Then, it determines if a new hard global state has been stored using a *checkpoint table* (Lines 7-11). The checkpoint table contains one row per CN, and one column per process. The table entries are initialized to zero. An entry is set to one whenever the corresponding checkpoint is written to

```
// Application process:
procedure createCkp()
    $CN := CN + 1$;
    resetTimer($T$);
    if (($CN$ mod $maxSoft$) = 0) sendCkpST(getState());
    else storeState(getState(), $CN$);

// Stable storage:
// The function arguments are the same as in receiveMesg
procedure receiveCkp($P_m, CN_m, timeToCkp_m, state_m$):
    storeState($state_m, CN_m$);
    $CN$ := max($CN, CN_m$);
    setBit($CN_m, P_m$);
    if (row($CN_m$) = 1) {
        $CN_{hard} := CN_m$;
        garbageCollect($CN_{hard}$);
    }
```

Figure 2.12: Functions to create a new checkpoint.

disk. The table only needs to keep one bit per entry, which means that it can be stored compactly. A new hard global state has been saved when all entries of a row are equal to one. The variable CN_{hard} keeps the checkpoint number of the new hard global state. The function `garbageCollect` removes all checkpoints with checkpoint numbers smaller than CN_{hard}.

2.3.3.3 Mobile Host Disconnection

A mobile host becomes disconnected whenever it moves outside the range of all the cells, or whenever the user turns off the network interface. While disconnected, the mobile host can not access any information that is stored in the stable storage. For this reason, the protocol must be able to perform its duties correctly using only local information. The protocol continues to save soft checkpoints in order to recover from soft failures. Two different types of disconnection can be considered. An *orderly disconnection* allows the protocol to exchange a few messages with the stable storage just before the mobile becomes isolated. Examples of this type of disconnection include situations in which the user calls a logout command, or the communication layers inform the protocol when the mobile is about to move outside the range of the cells (when the wireless signal becomes weaker). A *disorderly disconnection* corresponds to the opposite case, in which the protocol is not able to exchange any messages with stable storage. This happens, for instance, when the user unplugs the ethernet cable without turning off the application.

There are two reasons why the mobile host should create a new global state before it disconnects. From the mobile host owner's point of view, the protocol should create a new global state because it prevents the rollback of work done while the mobile was disconnected. If a failure occurs after the mobile's disconnection and before the creation of a new global state, the application rolls back to the last global state that was stored (without warning the mobile host). Later, during re-connection, the mobile's process will also have to roll back to this global state, undoing all the work executed while the mobile host was isolated. The creation of a new global state is also advantageous to the owners of the other hosts. If the mobile host (soft) fails after disconnecting and before saving its state, the protocol recovers the application process by doing a local rollback to the last checkpoint. When the mobile host re-connects, it will inform the other hosts about its failure (if the failure was hard, the user will have to tell the system administrator), and all the other processes will also have to roll back.

The mobile host cooperates with the stable storage to create a new global state before disconnection. Just before the mobile host becomes isolated, the protocol sends to stable storage a request for checkpoint, and saves a new checkpoint of the process (hard or soft, depending on the network). Then, the stable storage broadcasts the request to the other processes. Processes save their state as they receive the request. New global states can only be created before the mobile host detaches from the network if disconnections are orderly. Otherwise, the protocol is not able to determine when disconnections occur. In any case, the protocol can always create a local checkpoint. This soft checkpoint allows independent recovery of soft failures, and minimizes the second problem that was mentioned in the previous paragraph.

When the mobile host re-connects, the protocol sends a request to stable storage, asking for the current checkpoint number and the CN of the last hard global state. When the answer arrives, the protocol updates the local CN using the current checkpoint number. The protocol also creates a hard checkpoint if the mobile host has been isolated for a long time. If the difference between the received CN and CN_{hard} is larger than the maximum $maxSoft$ (in the example from Table 2.1, 8 or 128 depending on the assignment), the mobile sends a new hard checkpoint to stable storage. This checkpoint allows the hard global state to advance.

2.3.4 Related Work

Log-based checkpoint protocols save the processes' states without having to exchange messages [5, 50, 54, 56]. This is an interesting characteristic for mobile environments because processes can continue to create checkpoints while they are disconnected. On the other hand, these protocols usually have to save a reasonable amount of information to guarantee deterministic execution after a failure. This information includes the reception order and the contents of the messages. This can be a problem if the information has to be saved in a mobile host, since there is typically a limited amount of flash memory or disk. Log-based protocols also have the problem that processes usually need to exchange messages to garbage collect the stored information [80]. During recovery, processes also send messages to find a consistent global state [57] or to obtain information stored by the other processes [53, 54].

Most of the coordinated checkpoint protocols exchange messages during the checkpoint creation [19, 22, 81]. Messages are needed to guarantee that processes' checkpoints form a consistent recoverable global state. This characteristic makes these protocols inadequate for mobile environments. Time-based coordinated protocols do not need to send the coordination messages [10, 24].

However, they rely on synchronized clocks, which is something that will be difficult to guarantee in mobile environments.

Two checkpoint protocols designed for mobile environments have been proposed [39, 42]. The protocol by Acharya and Badrinath [39] uses a two-phase rule to determine when processes need to save their state. The two-phase rule requires processes to create new checkpoints whenever they receive a message after having sent a message. Processes also have to create a checkpoint whenever the mobile host switches from foreign agents, and prior to disconnection. The protocol logs all messages that are exchanged between processes. Both the checkpoints and the messages are stored in the current foreign agent. As the mobile host roams between places, the checkpoints and logged messages become scattered through the foreign agents.

Pradhan et al. [42] proposed two uncoordinated checkpoint protocols. The first protocol creates a checkpoint every time a process receives a message. The second approach creates checkpoints periodically, and logs all messages that are received. As in the protocol by Acharya and Badrinath, checkpoints and message logs are stored in the foreign agents. Pradhan et al. also propose three ways to deal with the problem of checkpoints becoming distributed through several nodes as the mobile host moves among cells.

The protocol described in this section creates checkpoints whenever a local timer expires, and it only logs the unacknowledged messages at checkpoint time. Two of the previous protocols create checkpoints based on the messages exchanged by the processes, which for certain patterns of communication results in the creation of a large number of checkpoints. Two of the previous protocols also need to log *all* messages, which can consume a large amount of disk. The proposed protocol saves checkpoints in the mobile host and in stable storage to recover from different types of failures. The previous protocols always assume hard failures, which means that they will always pay the extra cost of sending the checkpoints through the network. The proposed protocol does not rely on the foreign agents to store the checkpoints. In many cases, foreign agents will belong to some external organization that provides a mobile networking service. The protocol is not likely to be able to save the processes' states in these foreign agents. Having a computer in the home network that serves as stable storage also has two other advantages. First, the checkpoint protocol does not have to keep information about the location of the processes' states. Second, the computer can be made as reliable as the applications executed by the mobile hosts require (it is not possible to make *all* foreign agents highly reliable).

Vaidya has proposed previously a two-level scheme that uses two checkpoint protocols to tolerate distinct failures [82]. The two-level scheme relies on a log-based protocol to tolerate single process failures, and on a coordinated protocol to recover multiple process failures. The protocol described in this section is a coordinated protocol that saves two different types of global states. The first type of global state is able to recover single or multiple soft failures, and the second type of global state is used to tolerate single or multiple hard failures.

Chapter 3

Fault Detection Using the Socket Errors

One of the important components of any system that wants to provide transparent fault recovery is the fault detection module. Traditionally, two techniques have been employed to locate faults in distributed systems: polling [63] and watch-dogs [61, 62]. In the polling technique, a process periodically sends a message to the process being tested and then waits for an acknowledgment. If the acknowledgment does not arrive after a certain amount of time, the tested process is assumed to have failed. In the watch-dog technique, the tester process keeps a timer and expects to receive a message from the tested process before the timer expires. The tested process is considered to have failed if it is unable to send the message that resets the timer. In both techniques a tradeoff has to be made between performance and speed of fault detection. By decreasing the time out interval, faults are discovered faster but with higher overheads, because more communication is necessary, and the tested process is more frequently disturbed.

This chapter describes a way to detect failures in distributed systems, whose main advantage is having minimal overheads during failure-free operation [83]. Conceptually, the fault detection mechanism is very simple. It looks at the values returned by the stream socket functions as a process exchanges messages. If one of these values belongs to the set of errors associated with process failures, the mechanism can assume that the remote process was terminated. Fault detection based solely on these errors is not accurate since some errors might indicate a process failure, when in reality there was a network problem [84]. Nevertheless, the errors imply that at least the socket has to be recovered. To assess the coverage and latency of the proposed mechanism, faults were injected during the execution of two parallel applications. Our results show that in most cases, faults could be found using only the errors from the socket layer. Depending on the type of fault that was injected, detection occurred in an interval ranging from a few milliseconds to less than 9 minutes.

3.1 Background

Stream sockets based on the TCP/IP network protocols offer a full-duplex connection between two processes [85]. Before communication can be initiated, each process must establish the connection by opening a *socket*. Processes can exchange messages by writing to the socket or by reading from the socket. Stream sockets deliver messages reliably and in order, but they do not preserve message boundaries.

TCP uses checksums, sequence numbers, and acknowledgments to guarantee that messages are not lost or damaged [71, 86]. Conceptually, each byte of a message is assigned a sequence number. When TCP wants to transmit a message, it tags in the header with the sequence number of the first byte and the size of the message. Then, it saves a copy of the message in a send queue and starts a timer. The message is removed from the queue as soon as the acknowledgment arrives. If for some reason the acknowledgment is lost or delayed, TCP re-transmits the message when the timer expires. The receiver side saves the messages on a queue until a read call is issued and uses the sequence numbers to detect duplicate messages due to re-transmissions.

3.2 Types of Faults

Four distinct types of faults were examined, each resulting in the termination of a process, but with different behaviors observed at the socket interface. The four types of faults that were considered were:

Kill : The fault terminates the process, but does not affect the rest of the system. Examples of this type of fault are the following: the process is aborted because it executed an illegal instruction; the owner of the machine kills the process; or one of the assertions of the program is violated, and the process exits.

Crash : The machine where the process is running crashes permanently or stays down for a long period of time. Examples of this type of fault include a permanent failure in one of the machine's components or a situation in which an unattended machine crashes and no one is available to restart it.

Reboot : The machine where the process is executing shuts down, and then boots. A machine might be rebooted because a new software version requires rebooting in order to be installed, or because the machine is not performing as expected.

Crash & boot : The process is running on a machine that crashes, and then boots. Possible causes of such faults include power failures, incorrect usage by users, or operating system bugs.

3.3 Fault Detector

This section describes in which circumstances the stream sockets generate an error after a process failure. Two stream socket implementations were studied, both built on top of the TCP/IP communication protocols. The first implementation is based on the Berkeley sockets (SunOS

Inputs:		Error values:	
RST	-- Reset	0	-- Read returns 0 bytes
TOUT	-- Timed Out	BP	-- Broken Pipe
SQ=0	-- Send Queue empty	CT	-- Connection Timed Out
SQ!=0	-- Send Queue not empty	CRP	-- Connection Reset by Peer
RQ=0	-- Receive Queue empty	IA	-- Invalid Argument
RQ!=0	-- Receive Queue not empty	SBP	-- Signal Broken Pipe
R	-- Read		
W	-- Write		
R_Block	-- Read Blocks in the OS		
W_Block	-- Write Blocks in the OS		

Figure 3.1: Terminology used in the state diagrams.

4.1.3) and the second implementation is based on the streams from the UNIX System V R4 (Solaris 2.5). From the various functions of the socket interface that can be used to send or receive information, we chose to study the `read` and `write` system calls. The other functions provide similar error codes. To simplify the presentation, it is assumed that sockets are configured for blocking I/O (e.g., a read blocks when there are no messages available). The same errors would be seen if the sockets were set for non-blocking I/O.

A state diagram that explains the behavior of the sockets after the termination of a process was developed for each type of fault. The diagrams were derived by looking at the values returned by the read and write system calls, and at the messages exchanged by the surviving machines and the machine where the failure occurred. In the following subsections the term *local* is used to identify the failed machine, and the term *remote* denotes the machine that detects the fault by trying to communicate with the failed one. The terminology used in the state diagrams is presented in Figure 3.1.

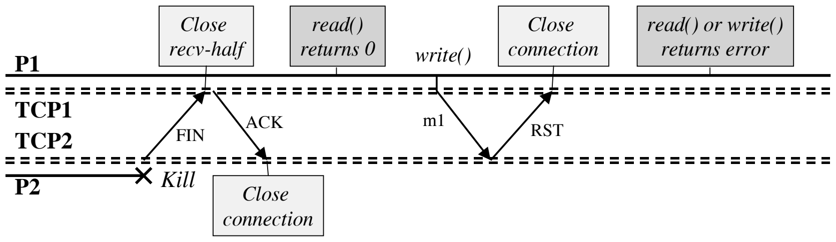


Figure 3.2: Process P2 suffers a kill fault and process P1 makes the detection.

3.3.1 Kill

Whenever a kill fault terminates the execution of a process, the operating system asks the local TCP to close all connections associated with the process (see Figure 3.2). A connection allows messages to travel in two directions, therefore, both directions have to be shut down before the connection is completely closed. During the closing procedure, the local TCP sends a special *FIN* message to the remote TCP indicating that no more messages will be sent.[1] Next, it completely closes the connection. After receiving the *FIN* message, the remote TCP closes the receive-half of the connection, but leaves the send-half open. When it tries to transmit a message, the local TCP responds with a *RST* message. The remote TCP completely closes the connection when the *RST* arrives.

The low-level messages exchanged between the TCPs result in errors returned by the socket interface. These errors are used by the fault detection mechanism to locate process failures. Figure 3.3 displays the various stages that a connection can undergo after a failure, as perceived by a remote process running on a machine with the Solaris operating system. The connection goes from state `OK` to one of the first four states depending on the status of the send and receive queues when the *FIN* message arrives. If the send queue is empty (SQ=0), there is a transition to either

[1] A process can also close a connection by calling the `close` or `shutdown` system calls. When this happens, the same *FIN* message is sent by the local TCP. Therefore, the fault detection mechanism must be disabled on a particular connection, before that connection is closed.

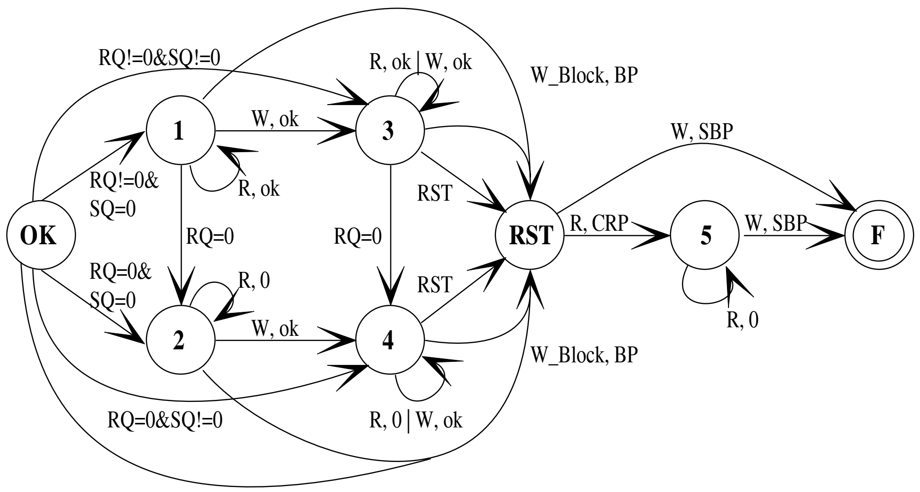


Figure 3.3: State diagram for kill fault on Solaris.

state `1` or `2`. The connection goes to state `1` provided that there are messages to be read (RQ!=0); otherwise, it goes to state `2` (RQ=0). In state `1`, the process reads the queued messages without being informed about the failure. When the receive queue becomes empty (RQ=0), the connection goes to state `2`. In this state, the failure is detected by a read call because the function *returns 0 bytes*.

A connection remains in state `1` and/or `2` as long as no messages are transmitted to the failed process. There is a transition to state `3` or `4` if the process executes a write. The connection can also leave state `OK` to one of these two states if the send queue was not empty at the moment of the failure (SQ!=0). A connection stays in states `3` and/or `4` during a short period of time, corresponding to the interval limited by the send of the message and the reception of the *RST*. Then, the connection moves to state `RST`. If the process blocks in the operating system while doing a write (W_Block), the function returns the error *broken pipe* when the *RST* arrives. This error can be used to detect the failure. The blocking happens if the process attempts to send a message larger than the available space in the send queue. The arrow from `OK` to `RST` corresponds to the case when the process is blocked in a write before the *FIN* message arrives. Any read or write call from `RST` state produces an error that can be used to detect the failure. The first read gives *connection reset by peer*, and the

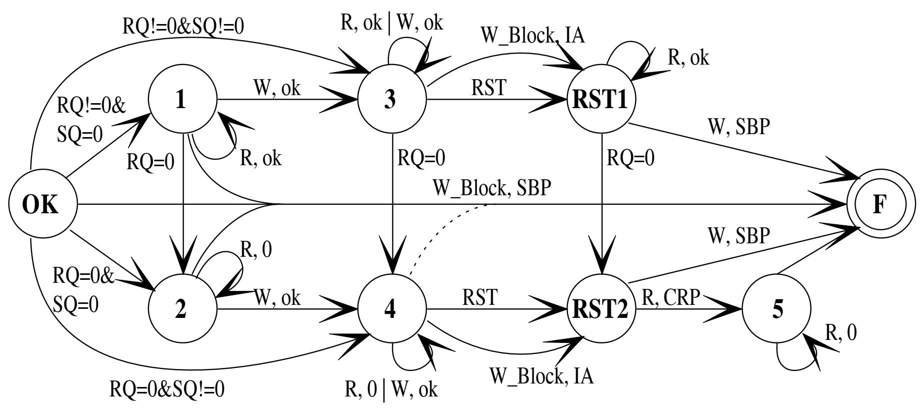


Figure 3.4: State diagram for kill fault on SunOS.

subsequent ones *return 0 bytes*. A write from the state 5 or RST generates a signal *broken pipe* that is thrown to the process. Unless this signal is caught, it terminates the execution of the process.

Figure 3.4 displays the state diagram for the case when the fault is detected by a process running on a machine with SunOS (we will present only this diagram for the SunOS). There are two main differences between SunOS and Solaris. The first one is related to the writes that block in the operating system (W_Block). In SunOS, these writes usually generate a signal *broken pipe* when the *RST* arrives. The only exceptions are in states 3 and 4, where the error *invalid argument* is returned (in state 4, the process sometimes received the signal *broken pipe*). The second difference is related to the messages stored in the receive queue. In SunOS, a process can continue to read these messages even after the reception of the *RST*, without generating any errors.

3.3.2 Crash

When a crash failure occurs, all the processes that were running on the machine terminate their execution. With this type of failure no warnings are transmitted to the remote TCPs. However, they can be detected if a remote TCP tries to communicate with the failed one. Since no acknowledgments are received in response to the sent messages, the remote TCP re-transmits the messages

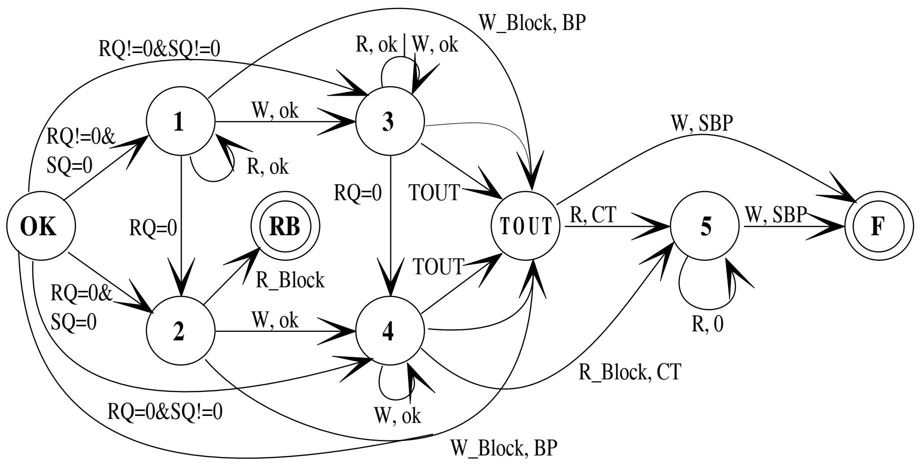


Figure 3.5: State diagram for crash fault on Solaris.

a certain number of times until it gives up, and then closes the connection. At that moment, an error is passed to the socket layer indicating a communication problem.[2]

After a crash, a process running on a Solaris machine observes the connection going through the stages depicted in Figure 3.5. As with the kill fault, the transition to one of the first four states depends on the condition of the send and receive queues. The connection goes from state `2` to state `RB` if the process attempts to read a message. In this state, the process blocks indefinitely in the operating system while it is waiting to receive a message from the failed process. The `RB` state corresponds to a case where the failure is not detected using only the errors from the sockets.[3] The connection goes to states `3` and `4` if there is a message to be transmitted. This can happen because the send queue was not empty when the failure occurred, or because the process tried to send a message. The connection stays in either one of these states until there is a time out and TCP closes the connection. If the process reads a message while the connection is in state `4`, it blocks in the operating system as in state `2`. However, the read returns the error *connection timed out* after the

[2]Even though a network partition does not terminate a process, it gives the same type of errors as a crash. This is a problem that has to be solved by any fault detection mechanism for distributed systems. Typical solutions require the processes that were assumed to have failed to terminate their execution.

[3]The optional keepalive mechanism of TCP would allow the detection of the crash fault when the process blocks in a read. However, the detection latency is typically around 2 hours, which makes this mechanism not very useful.

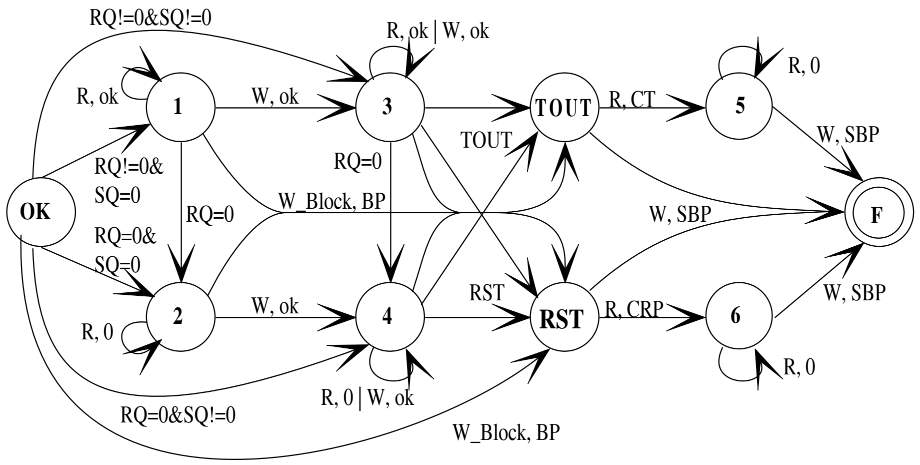


Figure 3.6: State diagram for reboot fault on Solaris.

time out takes place. The write system call issues the error *broken pipe* if the process blocks in the operating system while sending a message in states `OK` through `4`. Once the connection reaches the `TOUT` state, the next read or write returns an error.

3.3.3 Reboot

The reboot of a machine can be divided into three phases. During the first phase, while the machine is shut down, the operating system syncs all disks and tells TCP to close all connections. TCP transmits a *FIN* message through each connection, and then awaits an acknowledgment (for a few milliseconds). The second phase corresponds to the initial period of the booting procedure. During this period, no messages are sent, even to respond to remote requests. In the last phase, messages start to be transmitted and received as usual; however, all the knowledge about previous connections is lost. The TCP layer answers with a *RST* message to all incoming messages received at the end of the first phase (when the connections are closed), or after the third phase has started.

Figure 3.6 displays the various stages that a connection can experience after a reboot fault. Following the reception of the *FIN* message, the connection goes to one of the first four states; the specific state depends on the condition of the send and receive queues, as was explained for the

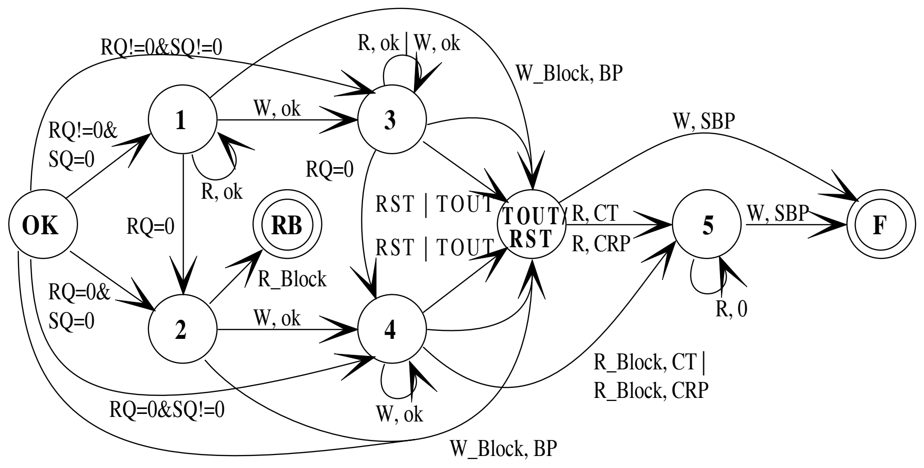


Figure 3.7: State diagram for crash & boot fault on Solaris.

kill fault. There is a transition from state `3` or `4` to state `RST`, if TCP receives a *RST* in response to a sent message. Once in state `RST`, the fault is detected by the next read or write. The connection can also go from state `3` or `4` to the state `TOUT` if no messages were transmitted in the first phase of the reboot that resulted in a *RST*, and if the second phase of the reboot takes longer than the time out period. This situation usually does not occur; however, it was added for completeness. A read or write after the connection has reached the `TOUT` state produces an error.

3.3.4 Crash & boot

With a crash & boot fault the machine crashes, terminating all processes, and then boots. As with crash faults no *FIN* messages are transmitted. Therefore, detection is only possible if the remote TCPs try to send messages to the failed TCP. During the booting of a machine, there is an initial period where incoming messages are not acknowledged; then, in a second phase, communication is restarted. Messages belonging to previous connections that are received in the second phase are answered with a *RST*. The reception of the *RST* closes the connection of the remote TCP.

Table 3.1: Summary of the Conditions and Errors for the Stream Sockets on Solaris.

	Condition	Function	Error
Kill	*FIN* arrived & Receive queue empty	read	0
	message send & *RST* arrived	read/write	CRP / BP, SBP
Crash	message send & Time out	read/write	CT / BP, SBP
Reboot	*FIN* arrived & Receive queue empty	read	0
	message send & (*RST* arrived or Time out)	read/write	CRP, CT / BP, SBP
Crash & boot	message send & (*RST* arrived or Time out)	read/write	CRP, CT / BP, SBP

The state diagram for the crash & boot fault is represented in Figure 3.7. This fault can be seen as a special crash fault, for which detection can be accomplished earlier if a *RST* is received before the time out occurs. The rest of the diagram should be interpreted like the one for the crash fault.

3.3.5 Summary

The previous four subsections explain, for each type of fault, in which circumstances the stream sockets generate errors. Failures are located using basically two methods: in the first, the TCP from the machine affected with the fault sends a *FIN* message informing the other TCPs about the process termination; in the second, one of the surviving TCPs attempts to send a message to the failed TCP, and then either receives a *RST* as response or receives no answer until the connection times out. When TCP determines that there was a failure, it informs the socket layer, which subsequently returns an error to the application. Table 3.1 presents a summary of the conditions for fault detection and the errors reported by the stream sockets on a Solaris machine. On a SunOS machine the error *invalid argument* also has to be considered.

3.4 Fault Injection Results

3.4.1 Applications and Environment

To assess the coverage and latency of the fault detection mechanism, faults were injected during the execution of a particle simulator and a raytracer. These applications were chosen because they represent two of the most common parallel programming models. These applications also have different communication frequencies and are sufficiently large to be considered *complete* parallel applications. The particle simulator, `ising`, simulates in two dimensions the spin changes of Spin-glass particles at different temperatures [87]. `Ising` is a geometric decomposition application where each process solves a sub-region of the total particle surface. In each step, a process first calculates the new spin values of its particles, and then exchanges the boundary particles with two other processes. The second application is a parallel implementation of the raytracer POVRAY 2.2 [88]. `Povray` is programmed using a master-slave model. The slaves receive from the master a certain number of pixels of the image, then compute the color of each pixel, and return the results to the master. The master distributes the pixels and saves the results on disk.

The experiments were performed on two machines running SunOS 4.1.3, a SPARCstation ELC and a SPARCstation IPC, and on a third machine running Solaris 2.5, an UltraSPARC 1. Faults were injected on the process executing on the SPARCstation ELC, which left two processes, one on the SunOS machine and another on the Solaris machine, to detect the faults. The master process of the `povray` application always ran on the SPARCstation ELC, which means that the slaves did the fault detection. The network and machines were lightly loaded when the experiments were done.

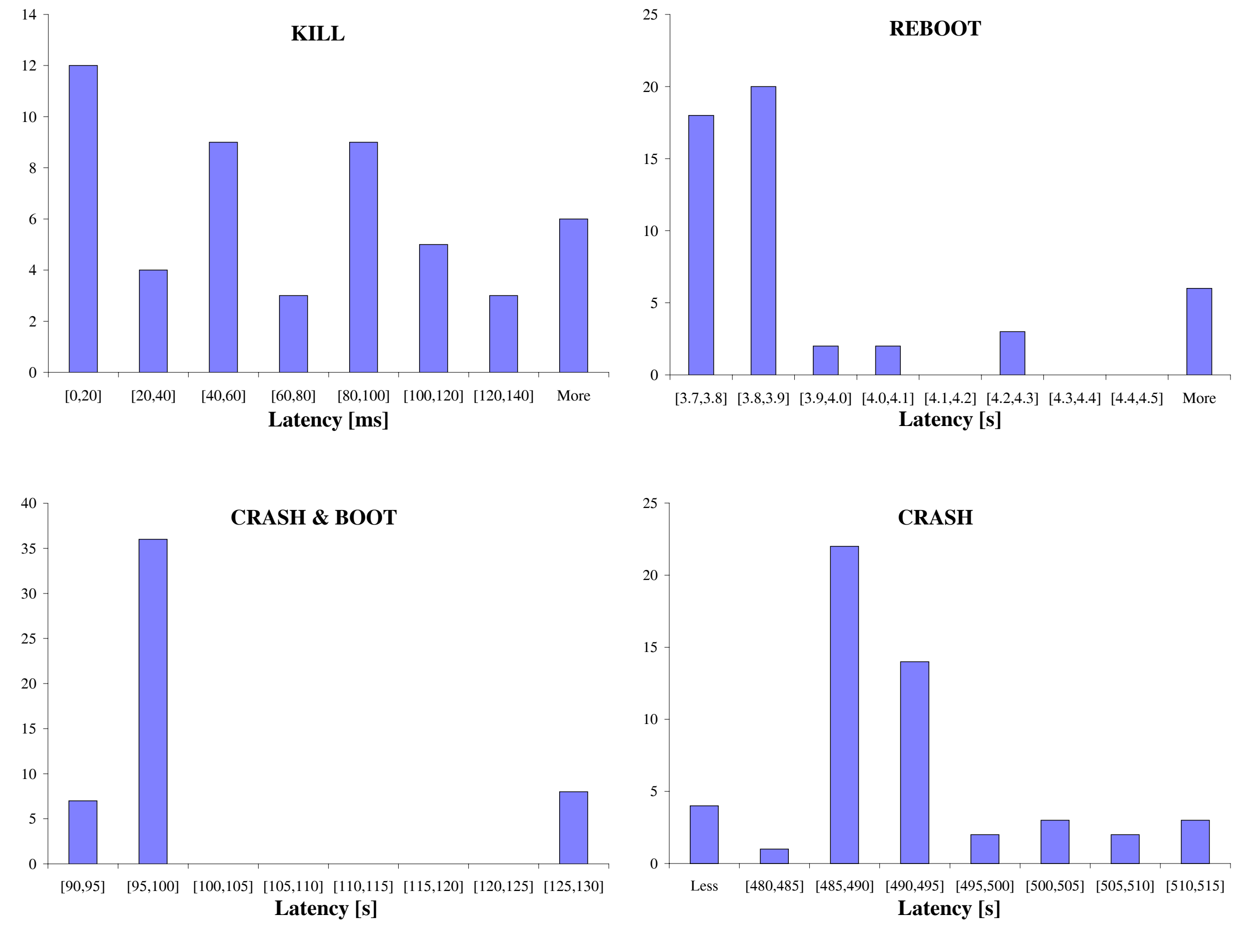


Figure 3.8: Histograms of the fault injection results on `ising`.

3.4.2 Coverage and Latency

The histograms of fault detection latencies collected in the experiments are shown in Figures 3.8 and 3.9. The values displayed correspond to the latencies observed by the first process that discovered the fault. In most cases, the Solaris machine detected the faults since it is faster than the SunOS machine. From the total number of faults that were injected, only three of them were undetected, all in the `povray` application. The detection latencies ranged from a few milliseconds for the kill faults to 511 seconds for the crash faults.

As was explained in Section 3.3, kill faults can be located rapidly because surviving machines are informed about the failure. Typically, they are detected as soon as the surviving processes had

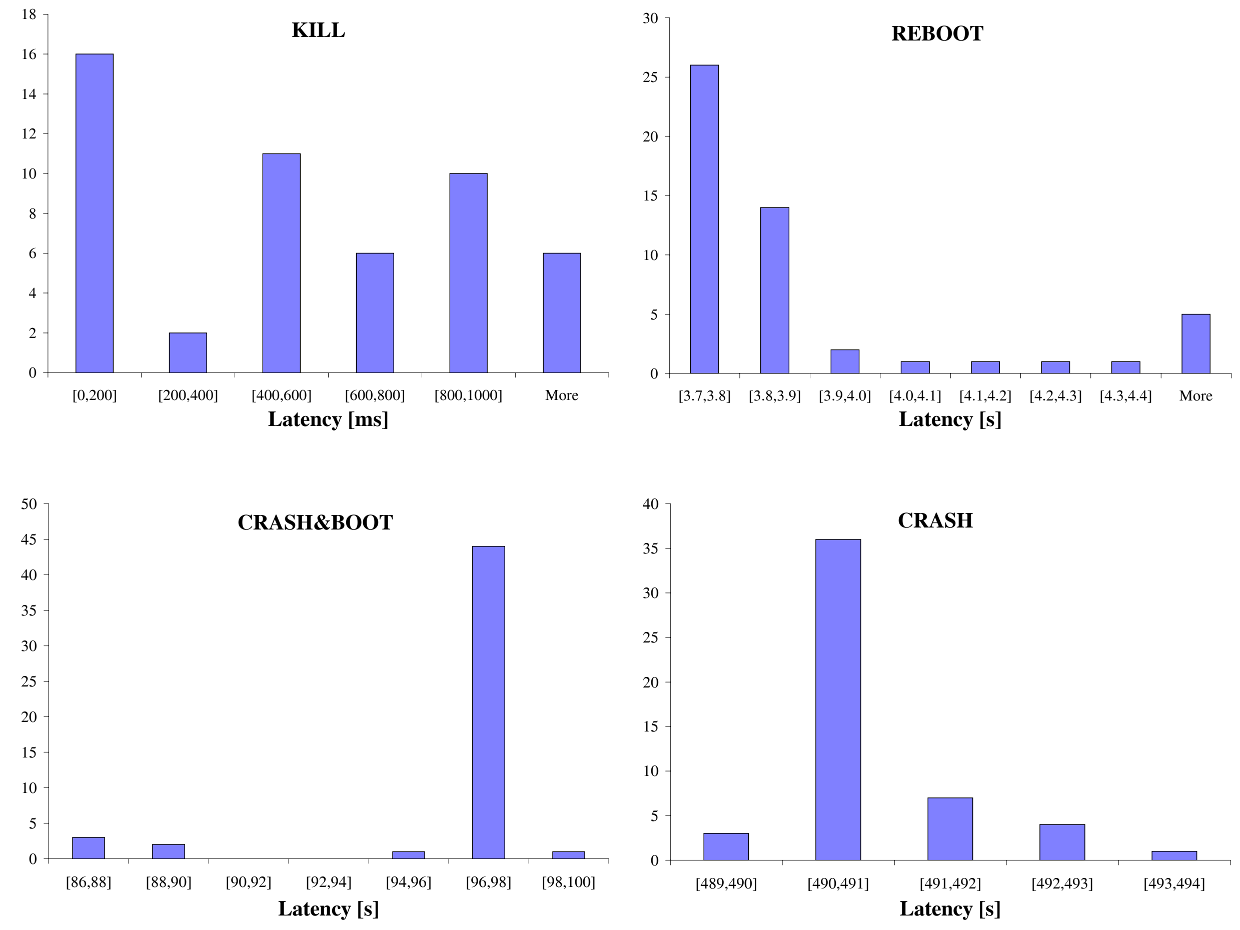


Figure 3.9: Histograms of the fault injection results on `povray`.

to exchange data with the failed one. On the `ising` application, the Solaris process sent its particles 15 ms after the beginning of the step, and then it waited on a receive. The SunOS process, which is slower, exchanged particles every 150 to 330 ms (computation time is smaller when the spins start to converge). On the `povray` application, slaves communicated with the master every 850 ms to 4.1 s in Solaris, and 11.5 s to 35 s in SunOS (certain parts of the image require more computation than others). Faults were detected with random latencies because they were injected at random times. However, the maximum latency was limited by the largest period without communication. The observed minimum and maximum latencies for `ising` were 2 and 195 ms, and for `povray` were 9 and 2443 ms. In both applications, the error reported most frequently was *read returned 0* *bytes*. Other returned errors were *connection reset by peer* with 6 cases for `ising`, and 2 cases for `povray`; and *signal broken pipe* with 2 cases for `povray`.

In the initial part of the reboot procedure, the operating system syncs the file systems and then closes all TCP connections. Reboot faults were discovered with a latency of 3.7 to 3.8 s, in most cases, for both applications. However, in a few other cases, fault detection took longer since the connections were closed later because of loaded file servers. The machine where faults were injected had several remote file systems that were mounted locally. Therefore, during the sync operation, it had to exchange many messages with a number of servers, so that all information that was cached in memory (e.g., modified superblocks) could be written to the remote disks. The error used to detect all faults was *read returned 0*.

Crash & boot failures can only be detected if the surviving processes attempt to send messages to the failed machine. However, the detection can not be done immediately after the crash because, during the initial part of the booting procedure, the crashed machine does not respond to the arriving messages. Messages have to be re-transmitted several times before the *RST* is returned. Consequently, faults are detected at discrete points of time, only at the end of the re-transmissions. Re-transmissions were usually done after the following intervals (in seconds): Solaris = 0.18, 0.38, 0.75, 1.5, 3, 6, 12, 24, 48, 56.25, 56.25, ...; SunOS = 0.5, 2, 4, 8, 16, 32, 64, 64, ... The detection latencies for `ising` can be divided into 3 clusters, one below 95 s, another around 96.5 s, and the last one at 127.5 s. The faults corresponding to the second cluster were located by the Solaris process (0.18 + 0.38 + ... + 48 = 95.81), and the faults belonging to the third cluster were found by the SunOS process (0.5 + 2 + ... + 64 = 126.5). The SunOS process also detected the faults from the first cluster. The SunOS machine sometimes used a different set of re-transmission intervals that resulted in an earlier detection (1.25 + 3 + 6 + 12 + 24 + 48 = 94.25). On `povray`, the majority of faults were located by the Solaris process roughly 96.5 s after fault injection. A few other faults were found before 95 s by the SunOS process. As was mentioned previously, the SunOS process had computation intervals as large as 35 s . Therefore, it could send a message to the failed process several seconds after the crash, resulting in re-transmissions in the period between the time when the crashed machine started to respond to incoming messages and the 96.5 s. The error reported for all faults was *connection reset by peer*.

Crash faults are detected when the surviving machines quit re-transmitting messages. The usual time out intervals for Solaris and SunOS machines are 490 s and 511 s, respectively. On the `ising` application, three faults were found by the SunOS process at roughly 511 s, and the rest were discovered by the Solaris process. Most of the faults located by the Solaris process had a latency of 490.5 s; however, a few others were detected with latency somewhat smaller or larger. By looking at the re-transmission times of the Solaris machine, we derived the following formula for the re-transmission intervals: $I_{n+1} = min(I_n * 2, 56.25)$. The usual value observed for I_0 was 0.18, which gives 490 s for the time out. However, sometimes Solaris used a distinct I_0, or one of the first I_k was larger than expected because of a delay. These small differences in the intervals explain the detection latencies that are smaller or larger than 490 s (e.g., I_0 = 0.20 results in a time out of 501 s). On the `povray` application, all faults were detected by the Solaris process at approximately 490.5 s. The error reported for all faults was *connection reset by peer*.

3.5 Integration and Related Work

The fault detection problem in distributed systems has been usually solved either using a process membership service, or a distributed system-level diagnosis protocol. Process membership services have been developed in the context of group-based systems, and their main objective is to provide a consistent view of which processes are currently members of a group, despite process joints, departures or failures [61, 62, 89–95]. Distributed system-level diagnosis protocols locate process failures, and then distribute this information in such a way that each node can independently determine the set of failure-free processes [63, 96–101]. Membership services normally assume that processes fail by crashing, and that the communication subsystem can suffer from performance or omission failures. Failures of this type lead to detection mechanisms based on watch-dogs. On the other hand, distributed diagnosis uses a polling-based mechanism, since the assumed failure model requires explicit testing to detect failed processes [102, 103].

The specification of a membership or a diagnosis protocol can usually be divided in two parts. In the first part, the protocol uses the watch-dogs or polling techniques to determine which processes are functioning correctly. After a failure is detected, the second part of the protocol ensures that live processes are informed about the failure within a limited amount of time. The detection mechanism described in this chapter can be used together with the watch-dogs and polling implementations since they require the exchange of messages among the processes. If the messages are sent through stream sockets, then the errors will be generated as previously described. Errors that result from the normal reads and writes during the application execution can be used to initiate the second parts of the protocols.

Chapter 4

The RENEW Tool

This chapter 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 [104]. 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* [65]. Applications conformable with MPI can either be developed in RENEW or they can be run without modification.

4.1 Overview

RENEW is a run-time system for clusters of workstations that supports the execution of message-passing parallel applications (Figure 4.1). The system is divided in two parts: a library that is linked

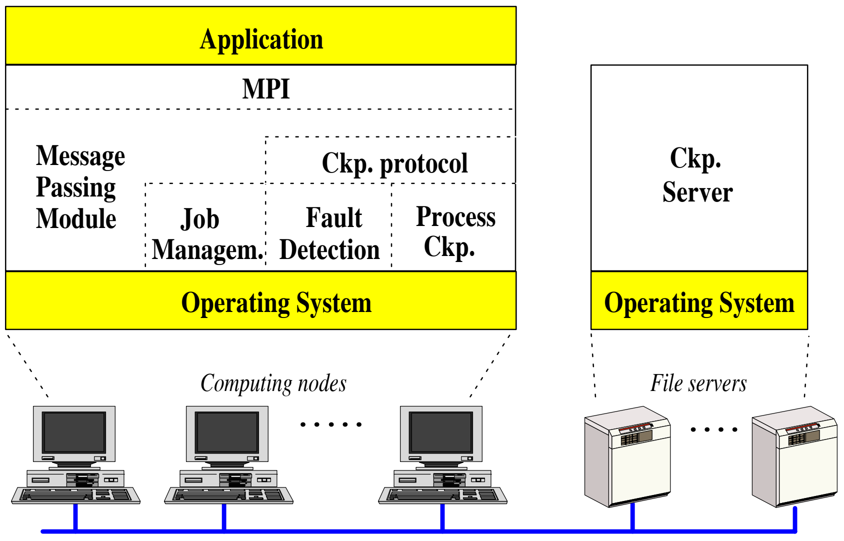


Figure 4.1: Architecture of RENEW.

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.

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 the fault detection

Table 4.1: 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)

mechanism described in the previous section. 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.

4.2 Checkpoint Interface Specification

The current checkpoint interface takes into consideration the different requirements of the three basic classes of protocols: uncoordinated, coordinated, 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.

4.2.1 Initialization and Ending

RENEW calls `renew_initCkpProt` when the initialization of the various modules is completed (see Table 4.1). 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.

4.2.2 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.

4.2.3 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.

4.2.4 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 the value 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.

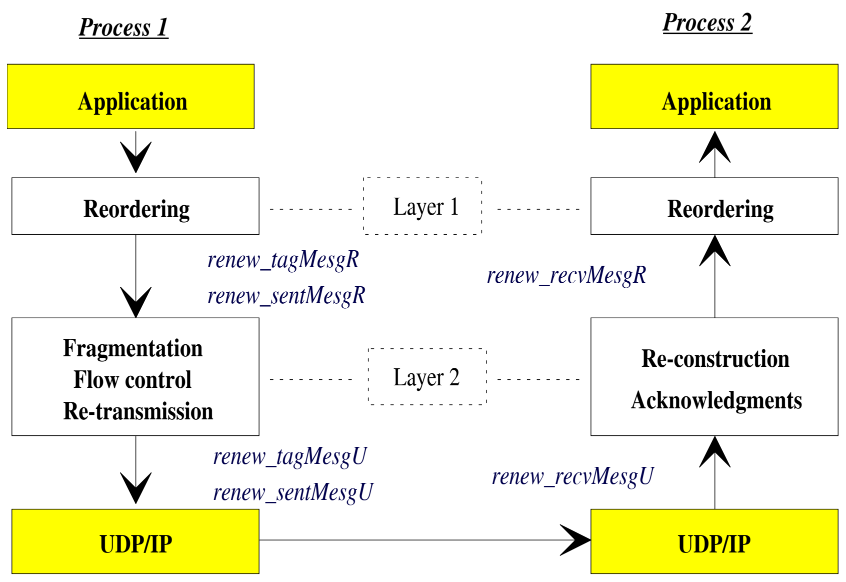


Figure 4.2: Message tagging and logging implementation.

4.2.5 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 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.

4.3 Implementation

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.

4.3.1 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 4.2). 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 [65].

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.

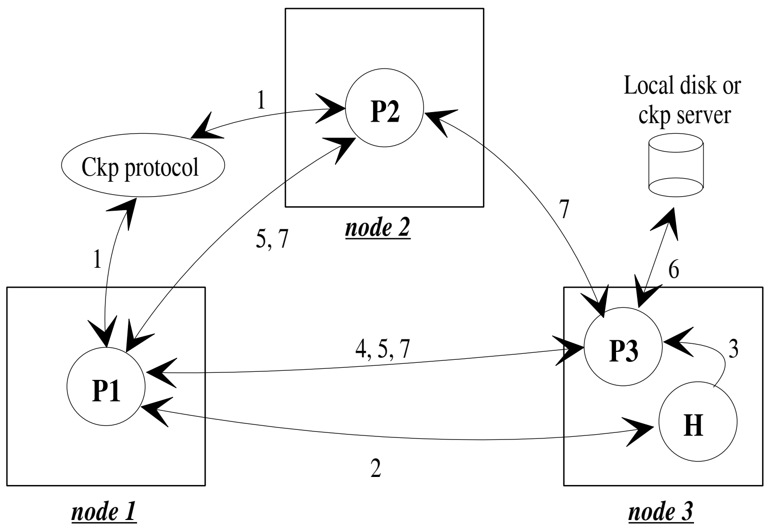


Figure 4.3: Recovery of process $P3$.

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.

4.3.2 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 4.3.

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 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 [12]. 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.

4.4 Related Work

Checkpointing and roll back recovery techniques have been proposed for a wide range of applications, including shared-memory systems [37, 105], distributed debugging [106], and mobile computing [42, 69]. OTEC is an object-oriented simulator designed for the analysis of checkpointing and recovery techniques [107]. It uses DEPEND [108] and SIMPAR [109] 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 [110] and MIST [111] are enhanced versions of PVM capable of restarting distributed applications from failures. In both cases, the Chandy and Lamport [2] 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 [112] 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 [3, 16, 66, 113–115]. Most implementations have been performed 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.

[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.

Chapter 5

Evaluation

This chapter presents three sets of experimental values. The first set shows the performance of the time-based protocol with no-logging and compares some of the overheads that are present in most coordinated protocols. Our results indicate that the most important overheads are the storing of the in-transit messages and the addition of information to messages. The less important overhead is the exchange of messages during the checkpoint creation. These values were obtained on the CM5 multiprocessor of the National Center for Supercomputer Applications. In second set of experiments, the performance of the time-based protocol with logging at the sender is analyzed when checkpoints are written to local and remote disks. These tests were performed on the cluster of Sun workstations connected by a 155 Mbit/s ATM.

The last set of experiments compare the performance of the adaptive protocol with an optimistic sender-based message logging protocol [5], and a communication-induced protocol [43]. It was observed that communication-induced protocol behaves in a manner that is very similar to the coordinated protocol, but with the cost of loosing most of its 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 reveal that all the protocols take roughly the same time to restore an application to the pre-failure state, even though the message logging protocol has performed numerous additional tasks.

5.1 Time-Based with No-Logging

This section describes experiments made on a 32-node partition of a CM5 [116]. Two versions of a two-phase protocol and the time-based protocol were implemented in the RENEW tool. The RENEW library was built on top of the CM5 communication library, CMMD, instead of the usual Unix sockets. Whenever an application attempted to send a message, it first passed the message to the library, which would execute all the steps required by the checkpoint protocols. Then, the message was transmitted using one of the CMMD operations. The inverse procedure occurred on the receive side.

The experimental results were obtained using the average of 3 to 5 runs of the benchmarks. The execution times were on the order of 10 minutes and the checkpoint period was set to 1 minute. The execution times were kept short because there was limited amount of CM5 time that had to be used for both the code development and experiments. This limitation, however, does not interfere with our conclusions, because they are mainly comparisons between different types of overheads. These results do not include the time to write the processes' checkpoints.

5.1.1 Two-Phase Implementation

The two-phase protocol utilized in the tests is similar to the protocol that was used in a previous study on coordinated checkpoint protocols [3]. With the exception of the bounded delivery times, the time-based and the two-phase protocols make similar assumptions about the communication channels.

The two-phase protocol maintains in each process a checkpoint counter, CN. Whenever is time to create a new global state, a coordinator increments its counter and broadcasts a special start message $< START, CN >$. After receiving the start message, a process updates the local CN and writes its checkpoint. Then, it returns a $< SAVED, CN >$ message. The coordinator, after saving its state and collecting all $< SAVED, CN >$ messages, broadcasts the $< END, CN >$ message to conclude the checkpoint procedure. To detect in-transit messages, the protocol tags each application message with the value of the local CN. A process stores in stable storage all messages that arrive with a CN_m smaller than the current CN. A process initiates the creation of a checkpoint if a message has a CN_m larger than the current CN. This last scenario can occur because communication channels are not FIFO.

Two versions of two-phase checkpoint protocol were implemented. The first version follows exactly the procedure just described. In that version, the checkpoint library has to make an extra copy of each message passed by the application in order to add or remove the CN tag. The CMMD library does not provide a *gather send* function for noncontiguous buffers with different sizes. If this function was available, the copy could be avoided by saving in one buffer the CN tag and in another the message supplied by the application. In general, the extra copy has to be made in any system in which *gather send* functions are not available.

The second implementation avoids the copy in most cases, at the cost of making the protocol less general and less transparent. This implementation uses two bits of the application message tags to piggyback the CN. In this case, CN can only take four different values, 0 through 3. This optimization could only be used in the communication operations that had a tag. In the other cases, it was necessary to make a copy. This restriction to the CN values requires the assumption that message delivery times are bounded and less than three times the checkpoint period

5.1.2 Time-Based Implementation

The time-based protocol with no-logging makes the assumption that delivery times are bounded. This assumption can not be guaranteed in a system like the CM5, although it can be approximated. Our implementation uses two characteristics of the CM5 to approximate bounded delivery times; the minimum I/O bandwidth guaranteed to each node and the small number of processes that execute concurrently in each node. The following formula was used to calculate the maximum delivery time:

$$td_{max} = \frac{maxMesgSize}{5M} + extraTime$$

The first term in the formula corresponds to the maximum time that a message takes to be transmitted through the network. It depends on the size of the largest message that is sent by the application, $maxMesgSize$, and on the minimum bandwidth that is guaranteed by the network between two nodes, 5 Mbytes/s. The second term is an upper bound on all the other delays that a message can experience. On some machines, it is difficult to determine this bound, because of the scheduling delays (e.g., a message can be received by the operating system, but the process is only scheduled after a long period of time). However, in the CM5 this problem is less important, because only a small number of processes are executed concurrently in each processor (usually only one or two processes). In the experiments, the value of td_{max} was set to 65 ms (25 ms to send the largest message and 40 ms for the extra time).

5.1.3 Applications

Six compute-intensive applications were used in the experiments (see Table 5.1). These applications are either kernels of larger programs, or complete programs. Each application has different characteristics in terms of frequency of communication, amount of information exchanged, or pattern of communication. The applications were the following:

Table 5.1: Applications Used in the Experiments on the CM5.

	Problem Description	Messages		
		Mesg/sec	KBytes/sec	Max Bytes
lu	50 512x512 matrices.	1763.1	3871.5	36864
mult	40 512x512 matrices.	6.0	200.8	34816
sor	3000 iterations 1024x1024 points.	748.6	3080.5	135432
tsp	10 problems with 20 cities.	143.2	586.6	4096
ga	popul. 1600, 4x10^6 funct. evaluations	19.3	79.6	4096
ising	1200 iterations 1024x1024 grid.	329.2	1348.6	4096

- `lu`: Performs the LU factorization of a matrix using the Gaussian elimination with partial pivoting. In each step, a node computes a column's multipliers and broadcasts them to the other nodes. Next, all nodes update the remaining columns.

- `mult`: Multiplies several matrices by the same initial matrix A ($A * B_i = C_i$, $i = 1, 2, ...$). Each node starts with the same copy of a matrix, and calculates a few contiguous rows of the result matrix.

- `sor`: Uses the red-black successive overrelaxation iterative method to solve the Laplace equation. The problem is parallelized by giving to each node a certain number of contiguous rows of the resulting linear system of equations. In each iteration, a node calculates its points, and then exchanges the two boundary rows with two other nodes.

- `tsp`: Solves the traveling salesperson problem. A master node keeps a queue with a number of tours that still have to be evaluated. The other nodes request tours from the master and try to find a minimum length tour.

- `ga`: Is a parallel implementation of the genetic algorithm system GENESIS 5.0 [117]. `ga` solves a non-linear optimization problem. This application divides the initial population among the nodes. A node executes the genetic algorithm on its individuals, and after a

Table 5.2: Experimental Results on a 32-Node Partition of the CM5.

	No Ckp.	Time-Based			Two-Phase		
	sec	#	sec	%	#	sec	%
lu	451.5	7	455.5	0.9	8	510.6	13.1
mult	418.1	7	420.3	0.5	7	440.7	5.4
sor	497.5	8	499.6	0.4	9	547.7	10.1
tsp	449.3	7	450.3	0.2	7	462.9	3.0
ga	418.0	6	419.1	0.3	7	424.4	1.5
ising	467.0	7	471.8	1.0	8	492.4	5.4

few generations, it sends a few individuals to one node, and receives a small number of individuals from another node.

- `ising`: Is a parallel simulation model of physical systems, such as alloys and polymers [87]. `ising` simulates in two dimensions the spin changes of Spin-glass particles at different temperatures. In each step, a node computes the spin values of a sub-region of the total particle surface, and then exchanges the boundary particles with two other nodes.

5.1.4 Individual Overheads

Table 5.2 presents the various results for executions with and without the checkpoint protocols. It is possible to observe that the time-based protocol performs better than the two-phase protocol in all applications. The time-based protocol shows overheads smaller than 1%, and the two-phase protocol has overheads between 1.5% and 13.1%. It is also possible to notice that in most applications the two protocols took different numbers of checkpoints (due to the periodicity of the checkpoint creation). However, even if we calculate the overhead per checkpoint[1], the time-based protocol outperforms the two-phase protocol.

[1]The reader should notice that this measure has to be used with caution, since the overhead of the checkpoint protocol does not occur only during the checkpoint creation, but also while the application is executing. In fact, a reasonable part of the overhead of the two-phase protocol occurs between the creation of the checkpoints.

The time-based protocol achieves good results because it avoids most overheads while the application is executing. The major cost of the protocol occurs only when the application is about to take its checkpoint. At checkpoint time, there is one interval in which processes can not send any messages. However, there is no blocking if the checkpoint happens to be taken in a computing phase of the process execution. The experiments show that in most cases only a small number of processes had to block, usually less than 10%. Also, the amount of blocking depends on the instant when the process attempts to communicate. If the message is sent at the end of the interval, the process experiences almost no blocking.

The size of the blocking interval is proportional to clock drift and the maximum message delivery time. Since clock drifts are small, their contribution to the interval is usually modest. However, the term corresponding to the clock drift grows with time. As was mentioned previously, timers can be re-synchronized to solve this problem. It is more difficult to guarantee that the maximum delivery time will always be small. For instance, it depends on the size of the largest message sent by the application and on the network bandwidth. The size of the messages can be kept within reasonable bounds if the checkpoint library implements message fragmentation. The applications that were used did not send messages larger than 135432 bytes, so there was no need to implement fragmentation in this early version of RENEW.

The performance costs introduced by the two-phase protocol can be divided mainly into message coordination, addition of information to messages, and in-transit message storage. Figure 5.1 presents, for each application, four bars showing how the costs have been sub-divided. The first bar shows the total overhead of the time-based protocol. The bar "No Copy, No In-transit" corresponds to the executions of the two-phase implementation with the no-message-copy optimization. This bar does not contain the time spent to save the in-transit messages (the function where the write occurs was removed), which means that it mainly shows the performance costs associated with the message coordination. It is possible to notice that this overhead is relatively small. The CM5 has

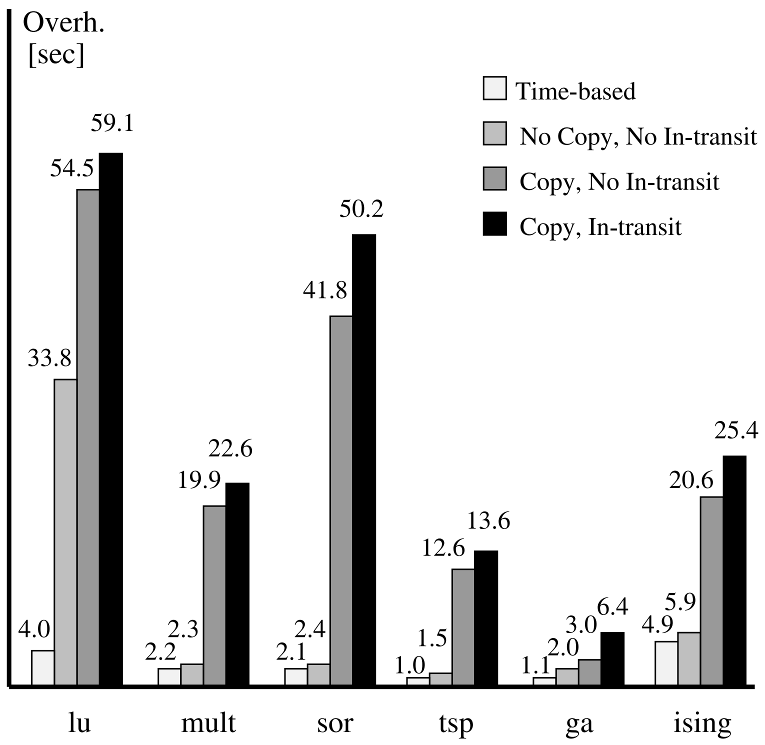


Figure 5.1: Distribution of the overheads.

a fast network, and we have implemented a message coordination completely asynchronous, that removes most of the costs, since no process is required to block while waiting for the messages from the other processes. `lu` is the only application that shows a high overhead because it was not possible to remove all message copies. `lu` distributes the matrix multipliers through the nodes using the broadcast primitive of the CMMD. The broadcast function is synchronous and does not associate a tag with the messages. In this case, it was necessary to make a message copy to add the CN number. The `lu` bar contains copying overheads in addition to the message coordination.

The bar "Copy, No In-transit" corresponds to executions of the two-phase implementation that makes a message copy. This bar does not include the time taken to save the in-transit messages. It mainly shows the elapsed time required to make the message copies and to coordinate the processes. The difference between the third and the second bar should be roughly equal to the time wasted while copying the messages. This time depends heavily on the application; applications

Table 5.3: In-Transit Messages.

	lu	mult	sor	tsp	ga	ising
Number	27	11	40	9	18	45
Kbytes	393	405	325	74	150	370

that exchange many and large messages have a higher copying overhead (see Table 5.1). The time also depends on the communication pattern that is used by the application. `tsp` sends more information per second than `mult`, but it has a smaller copy overhead. In the `mult` application, nodes send to the master the computed rows, and then wait for new rows of another matrix. The master only distributes new work after receiving all the previous rows. Therefore, each message copy made by the master not only makes the receiving process wait, but also all the other processes that are expecting new rows. Communication is less synchronous on `tsp`. A node sends its results to the master, and then receives new work without having to wait for the other nodes.

The last bar adds to the third bar the time required to save the in-transit messages. These values are the average of the 3 best execution times of 5 experiments. The number of in-transit messages that have to be stored can change dramatically from one run to the next. For instance, with the `ga` application, there was an execution for which it was necessary to save 207 messages with a total size of 1.7 Mbytes. These values are quite different from the averages shown in Table 5.3. The in-transit storage overhead depends on several factors, such as the number of in-transit messages, the size of the messages, and disk contention while the writes are being done. It also depends on the type of communication that is used by the applications. As was mentioned previously, if several nodes are expecting to receive a message from the same node at the same time, a disk access made by the sender can make several nodes wait. The use of different communication primitives can change the probability that in-transit messages will occur. For instance, `lu` sends more messages than `sor`, but has a smaller number of in-transit messages (see Tables 5.1 and 5.3). This is because `lu` uses a synchronous broadcast primitive.

Table 5.4: Description of the Applications Used in the Experiments.

	Problem Description
ga	1600 individuals, 10x10^6 function evaluations
ising	1500 iterations 2500x2500 grid
povray	400x400 pixels, music.pov image

5.2 Time-Based with Logging at the Sender

The implementation of the protocol was done on the RENEW run-time system, and was based on the procedures presented in the second chapter of the thesis. The values of the maximum clock drift rate and minimum message delivery time were set to 10^{-5} and 0, respectively. The reader should notice that setting td_{min} equal to zero is a `conservative` assumption since it increases the size of critical interval. The value of D was set to 10 ms and the checkpoint period to 5 minutes.

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. The processes' checkpoints were either saved in the local disk or in a remote HP workstation (also connected to the ATM network). All experiments were done during the night when the load in the network and machines was light.

5.2.1 Applications

Three compute-intensive applications were used in the experiments (see Table 5.4): `ga` and `ising` are the same applications that were used in the CM5 tests; `povray` is a parallel implementation of the raytracer POVRAY 2.2 [88]. This application uses a master-slave programming model. The main responsibility of the master is to distribute pixels of the image to the slaves. The slaves

Table 5.5: Performance Results on a Cluster of Workstations.

	No Ckp	Checkpoint				Time-Based			
				Local	Remote	Local		Remote	
	sec	#	KBytes	sec	sec	sec	%	sec	%
ga	2872	9	858/681	0.2/0.3	0.7/0.8	2972	3.5	2904	1.1
ising	3198	10	6828	1.5	3.3	3224	0.8	3232	1.1
povray	3091	10	683/22715	0.2/4.5	0.8/11.1	3120	0.9	3156	2.1

repeat the following steps: receive a number of pixels, calculate the color of the pixels, and return the results to the master.

5.2.2 Failure-Free Results

Table 5.5 displays the values obtained during the execution of the applications. The second column of the tables displays the execution times without the checkpoint protocol. In the column `size` is shown two checkpoint sizes for the `ga` and `povray` applications. The process that starts the `ga` application has a larger checkpoint size than the other processes because it allocates some extra data structures during initialization. The master of the `povray` application does not have to parse the image description file resulting in a smaller checkpoint size. In the column `Ckp Time` are presented the average elapsed times necessary to store a process checkpoint in the local disk or remote file server. These values correspond to execution of the instruction `saveProcessState` in the procedure. As expected, it takes more time to write the checkpoints in a remote disk than in a local disk. However, since the network is fast and only a small number of processes execute concurrently, the difference between the two write times is not too large (it doubles). It is possible to observe in all cases that the overhead introduced by the checkpoint protocol is very small, less than 3.5%. During the failure-free operation, the protocol only needs to create process checkpoints periodically, and to adjust the timers. All the other overheads were removed from the protocol.

5.3 Adaptive Protocol

Three types of checkpoint protocols were utilized in the experiments: a coordinated, an optimistic sender-based message logging, and a communication-induced. The coordinated protocol was the adaptive time-based protocol described for mobile environments. All the tests were done on the same cluster of workstations as the previous experiments. 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.

5.3.1 Other Checkpoint Protocols

The communication-induced protocol coordinates the creation of the checkpoints in a lazy fashion, by piggybacking a checkpoint sequence number in the messages [43]. 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. [52], our implementation performs equally well since no extra messages

Table 5.6: Applications Used in the Experiments.

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

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.

5.3.2 Applications

Five long-running parallel applications were used in the experiments. Table 5.6 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 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 [118]. 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 [119]. 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`

Table 5.7: Failure-Free Results (Ckp. Period 5 min.)

	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	3813 (13)	6.4	13	3905 (26)	9.0

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 [120]. This application is a comprehensive three dimensional global atmospheric model that has been improved over the past 15 years.

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 [121].

5.3.3 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 5.7 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 [122]: 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 5.6). 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 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.

[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.

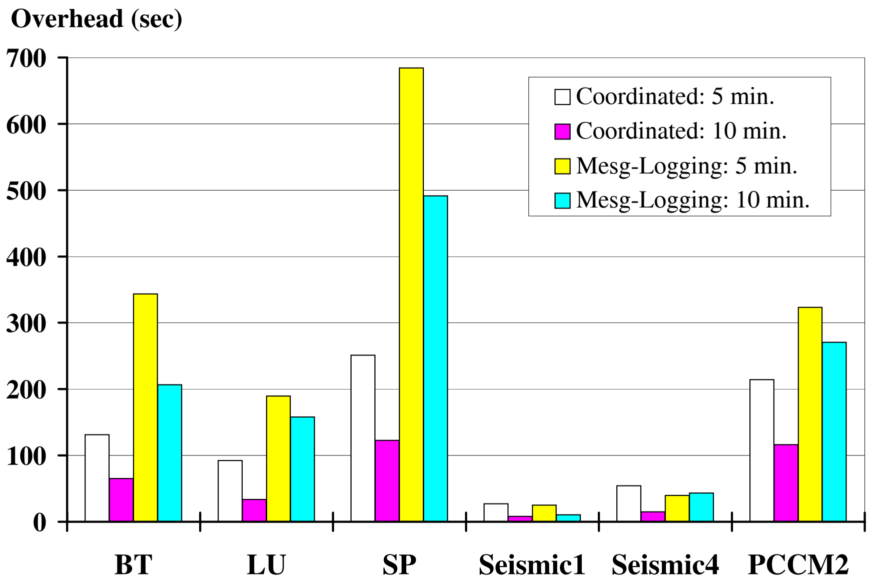


Figure 5.2: Failure-free overheads.

The message logging protocol showed the worst performance due to the high message traffic of some applications (see Table 5.6). 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 5.2 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 [113]. Even though many characteristics distinguish the two studies, e.g., hardware and applications, we reached the same basic conclusion that coordinated protocols introduce smaller

Table 5.8: 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)

overheads than message logging protocols. Our experiments, however, show that the performance gap between protocols has become wider.

Table 5.8 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.

Chapter 6

Conclusions

6.1 Summary

The thesis describes two variations of a new checkpoint protocol that uses time to coordinate the creation of application checkpoints. The protocol is optimal in the sense that all types of direct coordination, extra message exchanges and application's message tagging, have been removed. The protocol also does a minimal number of accesses to stable storage, since it only executes one write per process in each application checkpoint. Two procedures are used to implement the protocol, one that saves the process states, and another that keeps the timers approximately synchronized. The checkpoint creation procedure is executed locally by each process whenever a timer expires. The re-synchronization procedure requires the cooperation of the processes, but is executed infrequently. This procedure is also used to detect failures in the processor clocks that might lead to incorrect behavior of the protocol.

Both protocol variations have their own advantages; the protocol with no-logging prevents the existence of in-transit messages, avoiding the storage of this type of messages. This is accomplished by disallowing message sends during an interval before the checkpoint time. The size of the interval is proportional to the maximum message delivery time. The protocol with logging at the sender is more general since it does not require the assumption of small bounded message deliveries. However, it needs to include all potential in-transit messages in the sender's checkpoint.

The thesis also presents a coordinated checkpoint protocol well adapted to the characteristics of mobile environments. The protocol is able to save consistent recoverable global states even when mobile hosts are disconnected, because processes create new checkpoints whenever a local timer expires. A simple message tagging mechanism is used to keep the checkpoint timers approximately synchronized. Two types of process checkpoints are kept by the protocol; soft checkpoints are stored locally in the mobile host, and hard checkpoints are saved in stable storage. The protocol adapts its behavior to networks with different qualities of service by changing the number of soft checkpoints that are created per hard checkpoint. When the mobile host is disconnected, the protocol creates soft checkpoints to be able to recover from soft failures.

The thesis introduces and examines the effectiveness of a fault detection mechanism based on the errors from the stream sockets. It explains in which circumstances the errors can be used to locate four types of process failures, and it shows that all faults are detected as long as the live processes try to communicate with the failed ones. Failures are found mainly due to two reasons: first, in some cases, the faulty machine transmits a message informing the other TCPs about the process termination. Second, if a live machine attempts to send a message to the failed one, then it either receives a *RST* as response or it receives no answer until the connection times out. Fault injection experiments were made during the execution of two parallel applications to determine the coverage and latency of the fault detection mechanism. It was observed that most faults could be found using only the errors from the socket layer. Depending on the type of fault that was injected, detection occurred in an interval ranging from a few milliseconds for the kill faults to less than 9 minutes for the crash faults.

The thesis 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 with reduced programming effort. To support a broad range of applications, RENEW exports, as its external interface, the industry endorsed MPI interface. Several types of protocols were evaluated on the RENEW environment using a variety of benchmarks. Two coordinated protocols, a two-phase and the time-based with no logging, were compared on a 32-node partition of a CM5. It was observed that the time-based protocol outperformed the other protocol. On a network of workstations connected by ATM, three protocols were tested: the time-based for mobile environments, a communication-induced and an optimistic sender-based message logging. The experiments showed that the communication-induced protocol emulated the behavior of the coordinated protocol, with comparable performance. The message logging displayed 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 state of the applications.

6.2 Future Directions

Most of the performance studies on checkpointing that have been described in the literature, including our experiments, use complete or kernels of parallel applications. This type of applications has significant requirements in terms of computing power and a wide range of frequencies and types of communication patterns, but has modest interactions with the *outside world*[1]. Parallel applications usually read some configuration parameters at the beginning, then do some computation, and at the end write the results. Nowadays, a large number of applications are appearing, for example web-based or cooperative applications, that have many and different types of exchanges with the outside world, but much less requirements on the machines or network. These applica-

[1]The *outside world* is all systems and processes that do not belong to the application.

tions are not likely to be well supported by traditional checkpoint protocols or characterized by the previous performance studies.

Applications are also changing in the way they are implemented and on the systems where they are being run. Multi-threading is becoming common, which poses several difficulties to message logging protocols. The number of non-deterministic events will rise considerably since the outcome of interactions among the threads, for instance contention for a lock, will also have to be replayed during recovery. Multiple types of communication paradigms will be used by the applications, for example message passing and shared memory. Current clusters of workstations are starting to include both uniprocessors and shared-memory multiprocessors. In this type of environment, processes located in different machines will continue to exchange data using messages, but processes executing in the same multiprocessor will tend to communicate using shared memory because it is faster. Checkpoint protocols will also have to be adapted to support more dynamic environments like mobile systems.

References

[1] S. J. Mullender, *Distributed Systems*, Addison-Wesley, 1993.

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

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

[4] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix", *Usenix Winter 1995 Technical Conference*, pp. 213–233, January 1995.

[5] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging", in *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, July 1987, pp. 14–19.

[6] N. H. Vaidya, "On checkpoint latency", in *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995, pp. 60–65.

[7] N. H. Vaidya, "Another two-level failure recovery scheme: Performance impact of checkpoint placement and checkpoint latency", Texas A&M University, Tech. Rep. 94-068 (revised), January 1995.

[8] G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points", in *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, June 1983, pp. 48–55.

[9] F. Chao and J. R. Kenevan, "A non-fifo checkpointing protocol for distributed systems", in *Proceedings of the 1991 Symposium on Applied Computing*, April 1991, pp. 266–272.

[10] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations", in *Proceedings of the 10th Symposium on Reliable Distributed Systems*, September 1991, pp. 12–20.

[11] J. L. Kim and T. Park, "An efficient protocol for checkpointing recovery in distributed systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 231–240, August 1993.

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

[13] S. Krishnan and L. Kale, "Efficient, language-based checkpointing for massively parallel programs", Parallel Programming Laboratory, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. TR94-2, 1994.

[14] P.-J. Leu and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems", in *Proceedings of the Fourth International Conference on Data Engineering*, February 1988, pp. 154–163.

[15] P.-J. Leu and B. Bhargava, "A model for concurrent checkpointing and recovery using transactions", in *Proceedings of the 9th International Conference on Distributed Computer Systems*, 1989, pp. 423–430.

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

[17] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems", in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, June 1990, pp. 97–104.

[18] P. Ramanathan and K. G. Shin, "Use of common time base for checkpointing and rollback recovery in a distributed system", *IEEE Transactions on Software Engineering*, vol. 19, no. 6, pp. 571–583, June 1993.

[19] L. M. Silva and J. G. Silva, "Global checkpointing for distributed programs", in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 155–162.

[20] S. H. Son and A. K. Agrawala, "A non-intrusive checkpointing scheme in distributed databases systems", in *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, June 1985, pp. 99–104.

[21] S. H. Son and A. K. Agrawala, "Distributed checkpointing for globally consistent states of databases", *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1157–1167, October 1989.

[22] Y. Tamir and C. H. Séquin, "Error recovery in multicomputers using global checkpoints", in *Proceedings of the International Conference on Parallel Processing*, August 1984, pp. 32–41.

[23] Y. Tamir and T. M. Frazier, "Application-transparent process-level error recovery for multicomputers", in *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Services*, January 1989, pp. 296–305.

[24] Z. Tong, R. Y. Kain, and W. T. Tsai, "A low overhead checkpointing and rollback recovery scheme for distributed systems", in *Proceedings of the 8th Symposium on Reliable Distributed Systems*, October 1989, pp. 12–20.

[25] K. Venkatesh, T. Radhakrishnan, and H. F. Li, "Optimal checkpointing and local recording for domino-free rollback recovery", *Information Processing Letters*, vol. 25, pp. 295–303, July 1987.

[26] Z. M. Wójcik and B. E. Wójcik, "Fault tolerant distributed computing using atomic send-receive checkpoints", in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, December 1990, pp. 215–222.

[27] L. M. Silva and J. G. Silva, "On the optimum recovery of distributed systems", in *Proceedings of the EUROMICRO Conference*, September 1994, pp. 704–711.

[28] B. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems – An optimistic approach", in *Proceedings of the 7th Symposium on Reliable Distributed Systems*, October 1988, pp. 3–12.

[29] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm", in *Proceedings of the 4th Symposium on Reliable Distributed Systems*, October 1984, pp. 207–215.

[30] B. Janssens and W. K. Fuchs, "Relaxing consistency in recoverable distributed shared memory", in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 155–163.

[31] K. H. Kim, J. H. You, and A. Abouelnaga, "A scheme for coordinated execution of independently designed recoverable distributed processes", in *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, July 1986, pp. 130–135.

[32] D. L. Russell, "State restoration in systems of communicating processes", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 183–194, March 1980.

[33] N. A. Speirs and P. A. Barrett, "Using passive replication in Delta-4 to provide dependable distributed computing", in *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, June 1989, pp. 184–190.

[34] G. Suri and B. Janssens W. K. Fuchs, "Reduced overhead logging for rollback recovery in distributed shared memory", in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995, pp. 279–288.

[35] Y.-M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation", in *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993, pp. 86–95.

[36] W. G. Wood, "A decentralised recovery control protocol", in *Proceedings of the 11th International Symposium on Fault-Tolerant Computing*, June 1981, pp. 159–164.

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

[38] B. Randell, "System structure for software fault tolerance", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, June 1975.

[39] A. Acharya and B. R. Badrinath, "Checkpointing distributed applications on mobile computers", in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, September 1994, pp. 73–80.

[40] D. B. Hunt and P. N. Marinos, "A general purpose Cache-Aided Rollback Error Recovery (CARER) Technique", in *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, July 1987, pp. 170–175.

[41] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 231–240, April 1990.

[42] D. K. Pradhan, P. Krishna, and N. H. Vaidya, "Recovery in mobile environments: Design and trade-off analysis", in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996, pp. 16–25.

[43] R. Baldoni, F. Quaglia, and P. Fornara, "Index-based checkpointing algorithm for autonomous distributed systems", in *Proceedings of the 16th Symposium on Reliable Distributed Systems*, October 1997, pp. 27–34.

[44] D. Manivannan and M. Singhal, "A low-overhead recovery technique using quasi synchronous checkpointing", in *Proceedings of the International Conference on Distributed Systems*, May 1996, pp. 100–107.

[45] R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile logging in n-fault-tolerant distributed systems", *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pp. 44–49, June 1988.

[46] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism", in *Proceedings of the Nineth ACM Symposium on Operating System Principles*, October 1983, pp. 100–109.

[47] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance", in *Proceedings of the Nineth ACM Symposium on Operating System Principles*, October 1983, pp. 90–99.

[48] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon, "Transparent recovery of Mach applications", in *Proceedings of the Usenix Mach Workshop*, July 1990, pp. 169–184.

[49] G. Richard III and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory", in *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993, pp. 86–95.

[50] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems", *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, August 1985.

[51] Y.-M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems", in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 138–144.

[52] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and orphan-free message logging protocols", in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 145–154.

[53] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit", *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, May 1992.

[54] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system", in *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Systems*, August 1994, pp. 121–129.

[55] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro, "Lightweight logging for lazy release consistent distributed shared memory", in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[56] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX", *ACM Transactions on Computer Systems*, vol. 7, no. 1, pp. 1–24, February 1989.

[57] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing", *Journal of Algorithms*, vol. 11, no. 3, pp. 462–491, September 1990.

[58] S. L. Peterson and P. Kearns, "Rollback based on vector time", in *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993, pp. 86–95.

[59] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging", in *Proceedings of the Eighth Annual Symposium on Principles of Distributed Computing*, August 1989, pp. 223–238.

[60] S. W. Smith, D. B. Johnson, and J. D. Tygar, "Completely asynchronous optimistic recovery with minimal rollbacks", in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995, pp. 361–370.

[61] F. Cristian, "Reaching agreement on processor group membership in synchronous distributed systems", *Distributed Computing*, vol. 4, pp. 175–187, 1991.

[62] F. Jahanian, R. Rajkumar, and S. Fakhouri, “Processor group membership protocols: Specification, design and implementation”, in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, October 1993, pp. 2–11.

[63] R. Bianchini and R. Buskens, “An adaptive distributed system-level diagnosis algorithm and its implementation”, in *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991, pp. 222–229.

[64] 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, October 1996.

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

[66] N. Neves and W. K. Fuchs, “Using time to improve the performance of coordinated checkpointing”, in *Proceedings of the International Computer Performance & Dependability Symposium*, September 1996, pp. 282–291.

[67] N. Neves and W. K. Fuchs, “Coordinated checkpointing without direct coordination”, in *Proceedings of the International Computer Performance & Dependability Symposium*, September 1998.

[68] N. Neves and W. K. Fuchs, “Adaptive recovery for mobile environments”, in *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, October 1996, pp. 16/1–16/8.

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

[70] F. Cristian and C. Fetzer, “Probabilistic internal clock synchronization”, in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, October 1994, pp. 22–31.

[71] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.

[72] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, “Distributed fault-tolerant real-time systems: The Mars approach”, *IEEE Micro*, pp. 25–41, 1989.

[73] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, “Active replication in Delta-4”, in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 28–37.

[74] G. H. Forman and J. Zahorjan, “The challenges of mobile computing”, *Computer*, vol. 27, no. 4, pp. 38–47, April 1994.

[75] M. Nemzow, *Implementing wireless networks*, McGraw-Hill Series on Computer Communications. McGraw-Hill, Inc., New York, 1995.

[76] R. Katz et al., “The Bay Area Research Wireless Access Network (BARWAN)”, in *Proceedings of the Spring COMPCON Conference*, 1996.

[77] C. Perkins, “IP mobility support”, *Internet Engineering Task Force, Internet Draft (work in progress)*, February 1996.

[78] T. H. Lai and T. H. Yang, “On distributed snapshots”, *Information Processing Letters*, vol. 25, pp. 153–158, May 1987.

[79] V. Bharghavan, “Challenges and solutions to adaptive computing and seamless mobility over heterogeneous wireless networks”, *International Journal on Wireless Personal Communications*, 1996.

[80] Y.-M. Wang and W. K. Fuchs, “Optimistic message logging for independent checkpointing in message-passing systems”, in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 147–154.

[81] J. S. Plank, *Efficient checkpointing on MIMD architectures*, PhD thesis, Princeton University, June 1993.

[82] N. H. Vaidya, “A case for two-level distributed recovery schemes”, in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 64–73.

[83] N. Neves and W. Kent Fuchs, “Fault detection using hints from the socket layer”, in *Proceedings of the 16th Symposium on Reliable Distributed Systems*, October 1997, pp. 64–71.

[84] K. P. Birman and B. B. Glade, “Consistent failure reporting in reliable communications systems”, Dept. of Computer Science, Cornell University, Tech. Rep. CS TR 93-1349, May 1993.

[85] W. R. Stevens, *Unix Network Programming*, Prentice Hall Software Series, 1990.

[86] J. Poster (ed.), “Transmission control protocol”, *RFC 793*, September 1981.

[87] J. G. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira, “Experimental assessment of parallel systems”, in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996, pp. 415–424.

[88] POV-Ray Team, *Persistency of vision ray tracer (POV-Ray): User's Documentation*, 1993, http://povray.org.

[89] Y. Amir, D. Dolev, S. Kramer, and D. Malki, “Transis: A communication sub-system for high availability”, in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 76–84.

[90] F. Cristian and F. Schmuck, "Agreeing on processor group membership in asynchronous distributed systems", Dept. of Computer Science and Engineering, University of California, San Diego, Tech. Rep. CSE95-428, 1995.

[91] C. Fetzer and F. Cristian, "Agreeing on who is present and who is absent in a synchronous distributed system", in *Proceedings of the 16th Symposium on Reliable Distributed Systems*, October 1997, pp. 157–164.

[92] K. H. Kim, H. Kopetz, K. Mori, E. H. Shokri, and G. Gruensteidl, "An efficient decentralized approach to process-group membership maintenance in real-time LAN systems: The PRHB/ED scheme", in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992, pp. 74–83.

[93] A. Mishra, L. L. Peterson, and R. D. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs", *Distributed Systems Engineering Journal*, vol. 1, no. 2, pp. 87–103, 1993.

[94] A. M. Ricciardi and K. P. Birman, "Using groups to implement failure detection in asynchronous environments", in *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Systems*, August 1991, pp. 341–351.

[95] L. Rodrigues, P. Verissimo, and J. Rufino, "A low-level processor group membership protocol for LANs", in *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993, pp. 541–550.

[96] S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, "A diagnosis algorithm for distributed computing systems with dynamic failure and repair", *IEEE Transactions on Computers*, vol. C-33, no. 3, pp. 223–233, 1984.

[97] R. Bianchini, K. Goodwin, and D. S. Nydick, "Practical application and implementation of distributed system-level diagnosis theory", in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, June 1990, pp. 332–339.

[98] R. Bianchini, M. Stahl, and R. Buskens, "The Adapt2 on-line diagnosis algorithm for general topology networks", in *Proceedings of GLOBECOM*, December 1992, pp. 610–614.

[99] G. Masson, D. Blough, and G. Sullivan, "System diagnosis", in *Fault-Tolerant Computer System Design*, D. Pradhan, Ed., chapter 8, pp. 478–536. Prentice-Hall, 1996.

[100] S. Rangarajan, A. T. Dahbura, and E. A. Ziegler, "A distributed system-level diagnosis algorithm for arbitrary network topologies", *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 312–334, 1995.

[101] E. P. Duarte and T. Nanya, "Hierarchical adaptive distributed system-level diagnosis applied for SNMP-based network fault management", in *Proceedings of the 15th Symposium on Reliable Distributed Systems*, October 1996, pp. 98–107.

[102] M. A. Hiltunen, “Membership and system diagnosis”, in *Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995, pp. 208–217.

[103] F. P. Preparata, G. Metze, and R. T. Chien, “On the connection assignment problem on diagnosable systems”, *IEEE Transactions on Electronic Computing*, vol. EC-16, no. 12, pp. 848–854, 1967.

[104] N. Neves and W. K. Fuchs, “RENEW: A tool for fast and efficient implementation of checkpoint protocols”, in *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.

[105] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, “A recoverable distributed shared memory integrating coherence and recoverability”, in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995, pp. 289–298.

[106] R. Netzer and J. Xu, “Adaptive message logging for incremental program replay”, *IEEE Parallel and Distributed Technology*, vol. 1, no. 4, pp. 32–39, November 1993.

[107] B. Ramamurthy, S. Upadhyaya, and R. Iyer, “An object-oriented testbed for the evaluation of checkpointing and recovery systems”, in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, June 1997, pp. 194–203.

[108] K. Goswami, R. Iyer, and L. Young, “Depend: A simulation-based environment for system level dependability analysis”, *IEEE Transactions on Computer*, vol. 46, pp. 60–74, January 1997.

[109] 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.

[110] 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, February 1993.

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

[112] G. Stellner, “CoCheck: Checkpointing and process migration for MPI”, in *Proceedings of the International Parallel Processing Symposium*, April 1996.

[113] E. N. Elnozahy and W. Zwaenepoel, “On the use and implementation of message logging”, in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, August 1994, pp. 298–307.

[114] J. Plank, "Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques", in *Proceedings of the 15th Symposium on Reliable Distributed Systems*, October 1996, pp. 76–85.

[115] Y.-M. Wang and W. K. Fuchs, "Scheduling message processing for reducing rollback propagation", in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, July 1992, pp. 204–211.

[116] Thinking Machines Corporation, *Connection machine CM-5 technical summary*, November 1993.

[117] N. Neves, A.-T. Nguyen, and E. L. Torres, "A study of a non-linear optimization problem using a distributed genetic algorithm", in *Proceedings of the International Conference on Parallel Processing*, August 1996, vol. II, pp. 29–36.

[118] 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, December 1995.

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

[120] 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.

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

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

Vita

Nuno F. Neves was born in Lisbon, Portugal, on March 13, 1969. He received a *licenciatura* (5 year degree) and a *Mestrado* in Electrical and Computer Engineering in 1992 and 1995 from the Universidade Técnica de Lisboa. In 1991 he joined *INESC - Instituto de Engenharia, Sistemas e Computadores* where he worked in the Distributed Systems Group. Throughout his doctoral studies he was supported with a Fulbright Scholarship and a PRAXIS XXI Fellowship, and from 1996 he was a research assistant at the Center for Reliable and High-Performance Computing at the University of Illinois. During 1997 and 1998 he worked as a research consultant at the Electrical and Computer Engineering Department at Purdue University. His research was recognized with the William C. Carter award at the 1998 IEEE International Fault-Tolerant Computing Symposium. His research interests include fault-diagnosis and recovery techniques, and tools for the development of distributed applications. After completing his doctoral dissertation, he will return to Portugal for a faculty position.