FADI: A FAULT TOLERANT ENVIRONMENT FOR OPEN DISTRIBUTED COMPUTING

Taha Osman, Andrzej Bargiela

Real-Time Telemetry Systems
Department of Computing, The Nottingham Trent University
Burton Street, Nottingham NG1 4BU
Email: taha@doc.ntu.ac.uk, andre@doc.ntu.ac.uk

FADI is a complete programming environment that serves the reliable execution of distributed application programs. FADI encompasses all aspects of modern fault-tolerant distributed computing. The built-in user-transparent error detection mechanism covers processor node crashes and hardware transient failures. The mechanism also integrates user-assisted error checks into the system failure model. The nucleus non-blocking checkpointing mechanism combined with a novel selective message logging technique delivers an efficient, low-overhead backup and recovery mechanism for distributed processes. FADI also provides means for remote automatic process allocation on the distributed system nodes.

1. Introduction

It is generally accepted that distributed systems represent the backbone of today's computing world. One obvious benefit of distributed systems is their ability to efficiently solve complex problems that require vast computational power. The execution of computation-hungry applications such as neural-network training or various systems modelling can only be significantly speeded-up by exploiting parallelism. Another benefit of distributed systems is that they reflect the global business and social environment in which we live and work. The implementation of electronic commerce, flight reservation systems, satellite surveillance systems or real-time telemetry systems is unthinkable without the services of intra and global distributed systems.

The deployment of distributed systems in these areas has put extra demand on their reliability. For distributed processing of number-crunching applications it is important to implement fault-tolerant measures to avoid the waste of computations accomplished on the whole distributed system when one of its nodes fails. In on-line and mission-critical systems a fault in the system operation can disrupt control systems, hurt sales, or endanger human lives. Distributed environments running such applications must be highly available, i.e they should continue to provide stable and accurate services despite faults in the underlying hardware.

FADI was developed to harness the computational power of interconnected workstations to deliver reliable distributed computing services in presence of hardware faults affecting individual nodes in a distributed system. To achieve the stated objective, FADI has to support the automatic distribution of the application processes and provide means for user-transparent fault-tolerance in a multi-node environment.

Tel: +44 (0) 115 9418418 ext. 2948, Fax: +44 (0) 115 9486518

Addressing the reliability issues of distributed systems involves tackling two problems: error detection and process recovery. Error detection is concerned with permanent and transient computer hardware faults as well as faults in the application software and the communication links. The recovery of failed distributed applications requires recovering the local execution state of the processes, as well as taking into consideration the state of the communication channels between them at the time of failure.

The rest of this paper is organised as follows: section "2" gives an abstract view of FADI; section "3" presents the error detection mechanism; sections "4 & 5" introduce a detailed description of the backup and recovery mechanism; FADI performance evaluation is discussed in section "6" and concluding remarks are made in section "7".

2. Overview of the FADI System

2.1 Target Domain of Applications

FADI is designed to support computation-intensive applications executing on networked workstations. While the "economy of effort" stipulates the introduction of measures that prevent the loss of the long-running computation results executing on distributed nodes, frequently it is difficult to justify the use of expensive hardware replication fault-tolerance techniques. This frequently is the case for off-line computationally intensive applications such as simulations of mass and heat transfer [1], graphics rendering [2], etc. Another relevant area for the application of FADI is on-line distributed programs such as decision support control mechanisms [3] and flight reservation systems. Such systems can tolerate a short delay in services - for fault-management, but a complete halt of the system upon the occurrence of faults cannot be accepted. This paper presents a cost-effective and efficient fault-tolerant solution for running these applications on network-based systems.

2.2 Structural Design of FADI

Figure 1 represents an abstract view of FADI operation. The *Error Detection* and *Fault Recovery* modules run on a central node (usually the server) that is assumed to be fault-tolerant, i.e the probability of its failure is negligible. The required reliability of the central node might be obtained by hardware duplication, but the detail of achieving it is not relevant to the discussion of the FADI itself. To store the

processes checkpoints, a centralised NFS file system that is visible and identical to all the network nodes is also assumed.

Upon system start-up, FADI identifies the configuration of the underlying network and presents it to the user, which selects the network node(s) on which the application processes should be executed. FADI spawns the application processes on the specified nodes and periodically triggers their checkpointing to save their execution image together with any inter-process messages are saved into stable storage.

Nodes participating in the application execution are continuously monitored, and in the event of node crash, checkpoints of all the processes running on the failed node are extracted from stable storage and the processes are restarted on an operative node. Interprocess messages received since the last checkpoint are also replayed from the log. FADI also detects user-processes that have exited prematurely due to a transient hardware failure and similarly recovers them.

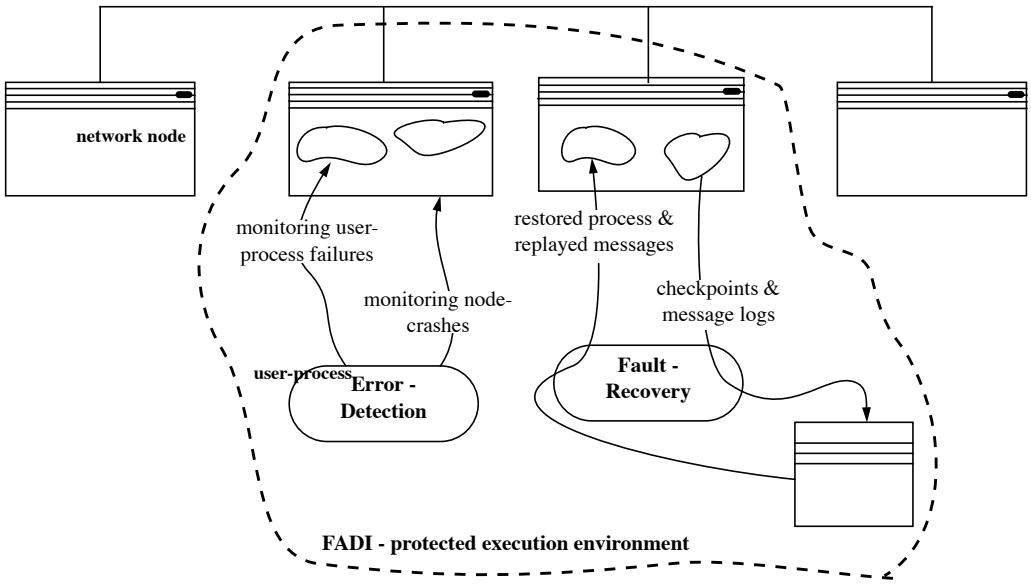

Figure 1 **FADI General Structural Design**

3. Detection of Faults in the Hardware Environment

The starting point for all fault-tolerant strategies is the detection of an erroneous state that in the absence of any corrective actions could lead to a failure of the system. FADI Error Detection Mechanism (EDM) identifies two types of hardware faults: processor node crashes (as caused by power failure) and transient hardware failures (temporary memory flips, bus errors, etc.) that cause the failure of a single application process, and also allows the integration of user-programmed (application-specific) error checks.

It is assumed that the processing nodes are fail-silent, i.e they only send correct messages, or nothing at all. However, the processes are not required to be to be fail-safe, i.e with a zero error latency.

3.1 Detecting node Failures

Detection of node failures is based on a central node monitoring task that periodically sends acknowledgment requests to all the nodes in the system. Each node must acknowledge within a predefined time interval (acknowledgment timeout), otherwise it will be considered as having "crashed".

The difficulty associated with this detection technique is the calculation of the acknowledgment timeout. Research in the Delta-4 system [4] asserted that the total message traffic between components of a dynamically evolving system (e.g, a shared communication inter-link as the ethernet) cannot in practice be assumed determenistic and varies from an instance to the other. To accommodate this variation, a background task was implemented to dynamically calculate the round-trip time *(rtt)* of inter-node messages with relation to the current network traffic. Each new round-trip sample *(Srtt)* is filtered into a "*smoothed*" estimate according to the formula:

$$\boxed{RTT_{i+1} = \alpha \times RTT_i + (1-\alpha) \times Srtt} \qquad \textbf{(Equation 1)}$$

where RTT_i is the current estimate of round-trip time, RTT_{i+1} is the new computed value, and α is a constant between 0 and 1 that controls how rapidly the estimated *rtt* adapts to the change in network load and is chosen to make the estimation more responsive to upward-going trends in delay and less responsive to downward-going trends. The calculation of the optimum value of α is beyond the scope of this paper.

3.2 Detecting Application-Process Failures

Transient hardware failures can cause the failure of the application processes. Most of these errors are detected by mechanisms built in the system hardware and the operating system kernel. These mechanisms include traps for illegal instructions, bus errors, segmentation faults, interrupts, arithmetic exceptions, etc.

Upon the detection of errors, the OS kernel kills the affected process with a signal number that corresponds to the error type. When a user-process exits, FADI analyses the process exit status to determine whether it exited normally or prematurely due to a failure, in which case the failed process recovery is initiated.

With regard to the user-assisted error detection, a special signal handler was dedicated to service the detection of such errors [5]. All the programmer has to do is to raise an interrupt with a predefined signal

number and the detection mechanism will handle the error as if it was raised by the kernel (OS) detection mechanism (KDM).

For a centralised detection mechanism - such as FADI's, it is vital to consider the latency of detecting errors on the distributed system nodes. In addition to the time necessary for delivering the fault report to the central detection server (approximately half the network round-trip time), we also need to consider the reaction time of the OS kernel to transient hardware faults. The latter is calculated once by a *worm-whole* type task that simulates a transient fault to measure the OS kernel reaction time. Round trip time is calculated dynamically as explained in the previous section. Hence, we can calculate the error latency of detecting process failures with 99% confidence by the formula:

$$\boxed{t_{latency} = E(t_{edm}) + 3 \times \sigma(t_{edm}) + \frac{rtt}{2}} \qquad \textbf{(Equation 2)}$$

where:
E(Tedm) - average reaction time of the kernel error detection mechanism;
σ(Tedm) - variance of the Tedm;
rtt - current estimate round-trip time from the master node to active_nodes.

The next section discusses how to recover the distributed user application processes from processor and network fault.

4. Recovery of Distributed Application Processes

Fault-tolerance methods for distributed systems have developed in two streams: checkpointing/rollback recovery and process-replication mechanisms. Process replication techniques have been widely studied by many researchers [6] [7]. With such techniques, each process is replicated and executed on a different computer such that the probability that all replicas would fail is acceptably small. Although these techniques incur a smaller degradation in performance when compared to checkpointing mechanisms, they are not overhead-free. The failure-free overhead is caused by using multicasting mechanisms to deliver every outgoing message to the *troupe* of replicated destination processes [8]. However, the main hindrance of wide adoption of process-replication methods in various areas is the heavy cost of the redundant hardware needed for the execution of the replicas.

In contrast with process replication mechanisms, this method does not require the duplication of the underlying hardware or the replication of the application processes. Instead, each process periodically records its current state and/or some history of the system in stable storage, an action called *checkpointing*. If a failure

occurs, processes return to the previous checkpoint (*rollback*) and resume their execution from this checkpoint. The overhead that this technique incurs is greater than that of process replication mechanisms because checkpoints are taken during failure-free operation, and rollback-recovery requires certain actions to be taken to ensure consistent recovery when processes crash. Nevertheless, this overhead is getting smaller as the computers and communication networks become more efficient. Hence, the degradation in performance caused by the checkpointing mechanism is quite acceptable for the class of applications FADI is targeting.

We will discuss FADI backup and recovery technique in two stages: The first looks into the process of saving the execution state into stable storage and restoring it in case of failure. The second investigates the coordination of the checkpointing and rollback in a distributed environment taking into account possible inter-process communication.

4.1 FADI Non-Blocking Checkpointing Mechanism

FADI checkpointing mechanism is based on a technique developed by *Condor* known as *Bytestream checkpointing* [9]. The idea is that the checkpointed application process should write its state information directly into a file descriptor, and a restarting process should read that information directly from a file descriptor.

For the targeted long-running class of applications, reducing the failure free overhead is of essential importance. Checkpointing contributes to this overhead by suspending the execution of the application program while the checkpoint is taken and written out to disk. In an attempt to minimize the checkpointing overhead, a *non-blocking* checkpointing technique was introduced. A copy is made of the program's data space and an asynchronous thread of control is used to perform the checkpointing routines, i.e reads the process state and records it to disk, while the user process continues the execution of the program code.

Figure 2 gives a high-level view of FADI checkpointing and rollback mechanism. Upon normal run (start) of the user program, FADI checkpointing prologue (linked with the checkpointed process) sets the local system timer to invoke the checkpointing of the user process at regular intervals specified by the user in the parameters, it then calls the user-main() routine.

Checkpointing starts by recording information about the process signal state (blocked, ignored, handled, etc.) and the status of opened files (mode with which opened, offset at which positioned) into a data struc-

ture allocated in the process heap - hence, part of the data segment. Then the entire data segment is written to the checkpoint file at checkpoint time, and is read back into the same address space at restart time in the event of failure.

To accomplish this, the start and end address of the data segment are needed. The start address is a constant location that is usually platform specific and can be found as a linker directive in the man pages. The end address is effectively the top of the heap, and can be returned by the UNIX sbrk() system call.

Preserving the stack requires saving and restoring the *stack context* (data structure containing the stack pointer amongst other stack related information), and the *actual data* that makes up the stack itself.

To save and restore the stack context, standard *"C"* functions setjmp() and longjmp() are used. setjmp() is called to save the current stack context into a buffer. If longjmp() is then called with a pointer to the saved buffer, the stack context is restored and the execution returns back in the code to the point where the original setjmp() was made. To save the stack data, the stack's start and end points are required. The start of the stack is a well known static location which is defined as a constant in the OS specification, the end of the stack, by definition, is pointed by the stack pointer held in the saved stack context.

Restoring the stack is more difficult because the stack space is used (for local variables) while being replaced. Directly replacing a process stack space with the space saved in the checkpoint file will irreversibly corrupt the executable image. To avoid this, the stack pointer is moved into a safe buffer reserved in the process data area. Moving the stack pointer is accomplished with yet another call to setjmp(), manually manipulating the stack pointer in the saved context to point to our buffer in the data area, followed by a longjmp(). Then, the reserved space in the data area is used as a temporary stack to safely replace the process original stack area with the one previously saved in the checkpoint file. So when moving the execution stack pointer back to its original place, we will effectively be using the restored stack.

FADI maintains the integrity of user-files across the checkpointing/rollback process by tracking appends and updates to user-files. This is achieved by augmenting UNIX system calls and *"C"* library functions that write to user-files (e.g write, fwrite, printf, etc.) with in-house functions that add the file to a list before calling the original routine. At checkpoint time, files in this list are checked and if the file was open for update, then a shadow copy of the file is created, otherwise the file was open for append and a note is made of its current size. When the checkpoint is taken, the forked thread is terminated with a user signal.

If a hardware fault was detected and a rollback of the user program is requested, then the checkpointing prologue calls the restore routine. It first replaces user files that were opened for update with their shadow copies and truncates files that were opened for append to the size recorded at the most recent checkpoint. These operations are performed only if the contents of the file were modified since the last checkpoint. Then the data segment is replaced with the one stored in the checkpoint file. Now the restore routine has the list of open files, signal handlers, etc. in its own data space, and restores those parts of the program state. Next it switches the current program stack with that saved at checkpoint time and returns to the user code at the same instruction that was interrupted when the last checkpoint was invoked. Finally, because checkpointing was performed by a signal handler, UNIX OS restores all CPU registers to their state before checkpointing took place.

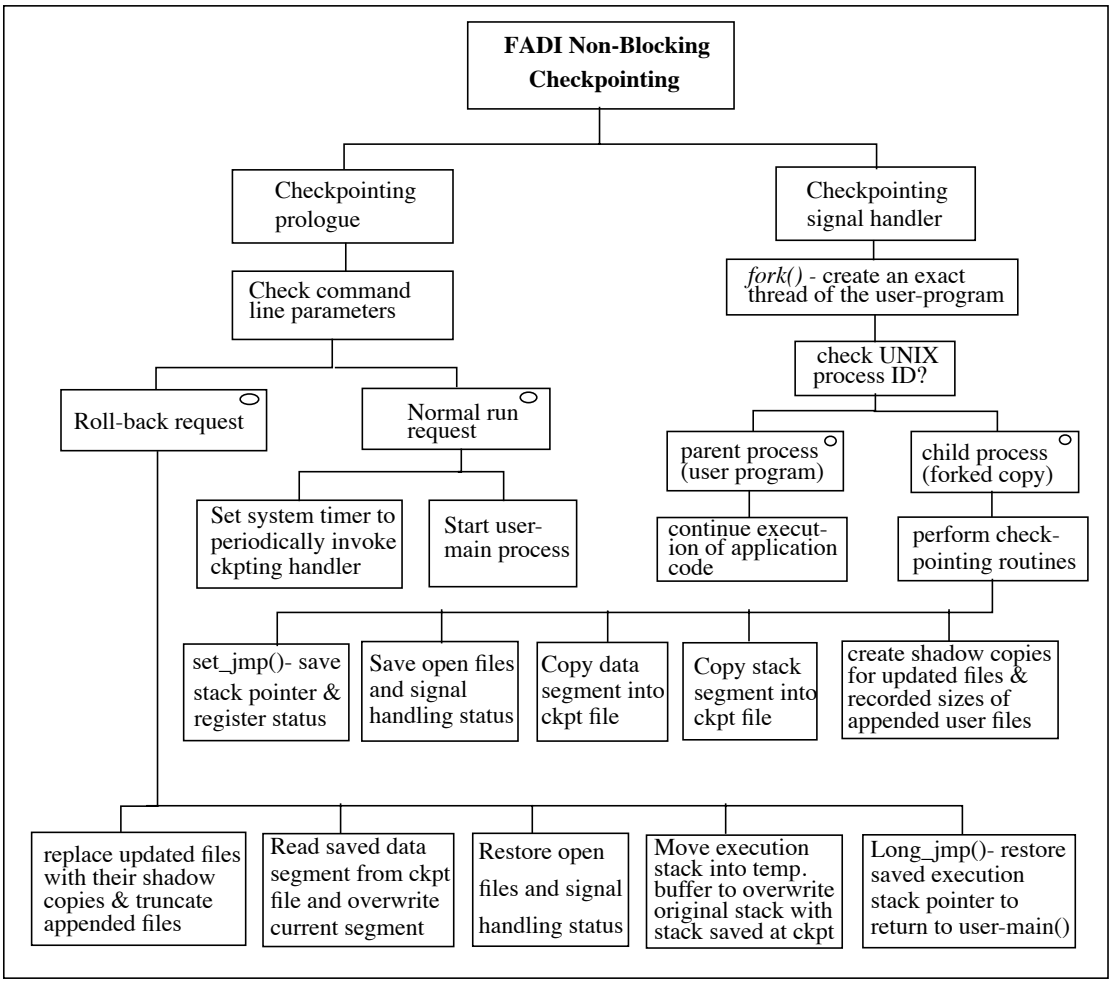

Figure 2 **Checkpointing & Rollback in FADI**

5. Coordination of Checkpointing and Rollback for Message-Passing (interactive) Applications

Conventional distributed checkpointing/rollback methods fall into three categories: pessimistic message logging, optimistic message logging and consistent (coordinated) checkpointing. Message logging schemes allows the processes to take their checkpoints independent of each other, and they record all inter-process messages in a message log. After a failure is detected, the previous checkpoint is restored and the logged messages are replayed (in the same order) to bring the failed processes back to a consistent system state. The disadvantage of message logging techiques is the overhead of logging all interprocess messages to disk.

coordinated checkpointing protocols checkpoint all (or interacting) processes together to form a consistent recovery line beyond which rollback is unnecessary. The advantage of coordinated checkpointing schemes is that they do not cause failure-free overhead (because no logging is necessary). The main computational cost of this approach is associated with blocking the applications while the recovery-lines are being stored. This overhead can increase significantly with higher checkpointing frequencies.

This paper presents a novel checkpointing/rollback algorithm that combines the advantages of the message logging techniques to take checkpoints for a single process at a time (without blocking the application), with the benefit of coordinated checkpointing techniques which gives complete recovery lines. The resulting algorithm is characterised by small failure-free execution overhead. The algorithm is called *Coordinating Checkpointing with Selective Message Logging*.

5.1 System and Environmental Model

The basic model is that all processes cooperate to create a global consistent recovery line, beyond which rollback is unnecessary. No determinism of the application processes is required, and messages are delivered via reliable FIFO channels.

Messages *received* by a node might not be *delivered* to the destination process until they are *requested*. i.e messages might have already been *received* by the message passing daemon, but not yet *consumed (requested)* by the destination process.

A centralised process running on a *fail-safe* node will coordinate the initiation and validation of checkpointing in all the application processes to form a *consistent recovery line*. This *Coordinating Task* will

also be responsible for maintaining the information logged to stable storage necessary for the selective rollback of interacting processes (logged messages, communication trees, etc.).

5.2 Recovery Line Consistency

The fundamental concepts of FADI reliable distributed computing algorithm are devised from solutions to the recovery-line consistency problem of coordinated checkpointing. The following notations will be used throughout the rest of this section:

Pα	: Process with index α;
Pset	: Set of all the application's distributed processes;
Cα	: the ith checkpoint of process α;
Ri	: the recovery line containing the ith checkpoint of all processes;
GSti	: the ith global state interval between recovery line i, i+1.

Two execution scenarios need to be considered to ascertain the consistency of a recovery line:

Scenario 1 Messages sent in one state interval and received in a subsequent one**:**

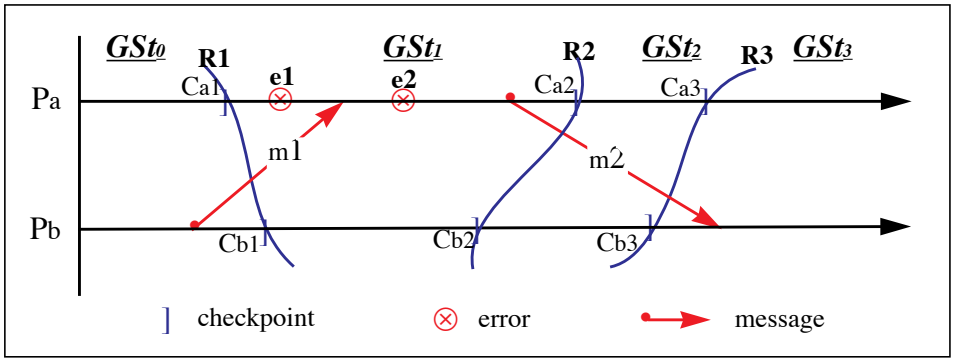

Figure 3 **Recovery-line Consistency (a)**

In Figure 3, if an error occurs in *GSt1 (e1 or e2)*, then processes *Pa, Pb* will roll back to the recovery line *R1*. Hence, the receipt of *m1* by *Pa* will be undone, but *m1* will not be resent by *Pb* after the rollback.

The most commonly adopted approach to solve this problem was suggested by Silva in [10] and Elnozahy in [8]. The messages are appended with the sender's state interval *sSt* which is compared with the receiver's local (current) state interval *lSt* upon message receipt. If *lSt* > *sSt*, this indicates that the message was sent from a previous state interval across the last recovery line so it is logged. Upon rollback, these messages are replayed to maintain consistency.

However, these algorithms do not distinguish between the *delivery* and the *consumption* of the message. Messages *delivered* to destination processes (on remote or local nodes) - whether using a low-level transfer

protocol like TCP/IP or a high-end message passing interface like PVM [11] - are *buffered* by the transfer protocol at the receiver's end until they are *requested* by the destination process *(consumed)*.

What follows is that the above approach compromises two aspects of the algorithm's reliability:

- It applies only to errors occurring after message consumption (*e2* in Figure 3), but does not take into account errors occurring after the message was delivered but before its consumption (*e1*), in which case the message (*m1*) will not be logged because the destination process (*Pa*) has not received it yet.
- It does not consider messages that might cross more than one recovery line (*m2*). All sent messages are assumed to be delivered at most in the next state interval.

A principal prerequisite for recovery line consistency is the *logging of all messages that cross it from left to right*. In order to avoid the shortcomings of the above approach, we have two realistic options to maintain the consistency of the recovery line:

1) indiscriminate sender-based message logging;

2) Identification of messages that crossed the recovery line, waiting for their receipt, logging them and only then declaring that the recovery line is consistent.

The first option is quite expensive in terms of failure-free overhead, as explained in the previous section, so we propose a solution based on the second option.

Scenario 2 Messages sent in one state interval but received in a previous one:

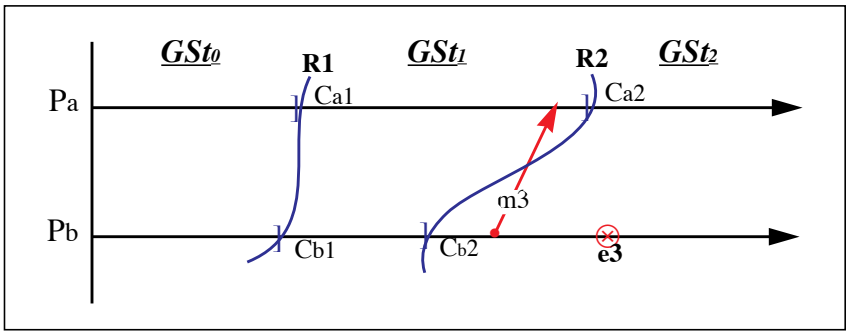

Figure 4 **Recovery-line Consistency (b)**

Figure 4 illustrates the case when the error occurs in *GSt2 (e.g at e3)*, then process *Pb* will roll back to the recovery line *R2 and re-run*. The sending of *m3* by *Pb* will be repeated, but will not be matched by a corresponding receive request from *Pa*. Possible solutions to this problem include:

1) Freezing of all processes until each have taken a local checkpoint. This approach implies a time and communication overhead during the failure-free operation of the application that can be computationally expensive for long-running algorithms.

2) Identifying messages sent from a subsequent state interval using state interval labels [10]. If *(lSt < sSt),* in which case the receiving process takes a checkpoint before consuming the message to preserve the algorithms consistency. Obviously determinism here is not required because the sending and receiving of message m3 will be performed within the same state interval *"GSt2"* and both operations (send and receive) will be redone (if needed) upon rollback, which makes this approach the most attractive.

5.3 Coordinated Checkpointing with Selective Message Logging. An Algorithm for Checkpointing & Rollback of Distributed Applications

The algorithm is based on conclusions drawn from discussing the system model and recovery line consistency of recovery schemes based on coordinated checkpointing with selective message logging in the previous section.

5.3.1 Sending and Receiving Messages

Upon *message send*, the message is augmented with three pieces of information: send sequence number of the "α" destination process *SSN*α, the *sender ID*, and the sender current local state interval *LSt*. After the message is successfully delivered, *SSN*α is incremented. If this is the first message sent to that destination since the last checkpoint, the Coordinator task is informed to update its list of communication flags *C* for this pair *(sender, dest)* of application processes indicating that they have communicated in the current state interval. This flag is set if any of the pair *(l,m)* sends a message to the other. The flags are cleared at the start of each new global state interval.

```
procedure send_msg()
begin
    append [SSN(dest_id) + sender_id + dest_id + sSt] to message body;
    if (first message to dest since last checkpoint)
       inform Coordinator task;
    SSN(dest_id) ++;
end
```

Upon *message receipt,* the *SSN* included in the received message is compared with the sequence number of the latest message received from the same process (stored locally in *RSN*). If the previous sequence number (*RSN*) is greater or equal to the newly received *SSN*, then a local checkpoint of the receiver state is taken to preserve the algorithm consistency. Next the receiver process compares the received *sSt* with the current local *lSt*, if the local state interval is bigger, then the message was sent from a previous state interval crossing the recovery line *LRmsg* and it should be logged.

```
procedure receive_msg()
begin
   unpack [SSN + sender_id + dest_id + sSt & message body];
   if (SSN <= RSN(sender_id)) then
         take a local checkpoint;
   else
         RSN(sender_id) = SSN;
   endif
   if (sSt < lSt) then
         log msg to LRmsg in stable storage;
   endif
end
```

5.3.2 The Checkpointing Protocol

At regular intervals the Coordinator task sends a checkpointing request *(ckpt_req)* to the application processes. The *ckpt_req* is sent together with the global state interval *GSt*. Each process takes a local checkpoint and sends an acknowledgment message *ckpt_ack* which contains local arrays of received messages *(RM)* and sent messages *(SM)*. Each *RM* array holds for every message originator two items of information: the "*sender_id"* and the receive sequence number of the last received message *"RSN"*. Similarly to the *RM* array, the *SM* array contains: *"dest_id"* and the send sequence number *"SSN"* of last message sent to that destination.

After the Coordinator task receives *ckpt_ack* from the application process, it checks if all the sent messages were received by the correspondent destination processes. If that is the case, then the recovery line is declared consistent and the global state interval is incremented.

Otherwise, information about unbalanced (unreceived) messages is added to the inconsistency (unbalanced messages) list "*UBmsg*". In this case, no new checkpoints are requested, and the Coordinator task inquires the processes on the receiving end of the unbalanced messages to check if the messages have been

received *(recv_req)*. When all messages in *UBmsg* are received, the last recovery-line is declared consistent, *Gst* is incremented, and the normal checkpointing operation is resumed.

It is worth to note that during checkpointing, flushing the transient messages data is not necessary to commit output as in traditional message logging techniques since in the case of failure the message can be replayed from the log, or resent if the sender is forced to rollback [8]

```
procedure checkpoint()
begin
  at periodic intervals do
  begin
     send ckpt_req[GSt] to all processes in Pset;
     await* for ckpt_ack[SM[sn], RM[rn]] from every p(i) in Pset;
     for each p(i) in Pset
        for each SM(k) in SM[sn]
           if not exists RM(j=1..rn) that ((RM(j).RSN == SM(k).SSN) &&
                    (RM(j).sender_id == SM(k).dest.id)) then
              add SM(k) to UBmsg;
           endif
        end
     end
     if UBmsg is not empty then
        at periodic intervals repeat
           send recv_req to all SM(k).dest_id in UBmsg;
           await for recv_ack [RM[rn]];
           for each SM(k) in UBmsg
              if exists RM(j=1..rn) that ((RM(j).RSN == SM(k).SSN) &&
                    (RM(j).sender_id == SM(k).dest.id)) then
                 remove SM(k) from UBmsg;
              endif
           end
        Until (UBmsg is empty)
     endif
     Declare Recovery-line(GSt) consistent;
     free logged messages from GSt-1;
     Gst++;
     clear communication flags C(l,m)
  end
end
```

5.3.3 The Rollback Protocol

When a process fails, the Coordinator task looks up the communication table C(l,m) and only the processes that interacted with the failed process since the last global checkpoint are rolled-back. Messages addressed to the rolled back processes are replayed from the log.

```
procedure rollback(fail_processID)
begin
  look up processes in C(fail_processID, *)
  for all * processes
          rollback to Gst
          unset C(fail_processID, *)
          replay all messages to * from LRmsg
  end
  rollback fail_process
  replay messages from LRmsg to fail_process
end
```

6. Performance Measurements

The performance tests were carried out on a network of three workstations connected by a 10 Mbit/sec Ethernet. The main node running the Fault-management tasks is a SPARC_20 server running SOLARIS 5.5.1 (60 MHz clock rate, 32 Mbytes of main memory, 5 Gbyte SCSI disk drives). The applications were distributed between two diskless SPARCstation *IPC*s running SunOS 4.1.3 (40 MHz clock rate, 8 and 16 Mbyte of main memory).

6.1 Performance Metric Requirements

The main overheads that can affect the run-time of the distributed applications are:

a) The message logging overhead: it includes the overhead of tracking the dependencies of sent/received messages exchanged between the application processes (i.e augmenting bookkeeping information to messages, processing this information to maintain the consistency of the communication channels, etc.), and the overhead of logging inconsistent messages into stable storage.

b) The checkpointing overhead: it includes the overhead of taking the checkpoint (i.e saving the process image: open-files, communication sockets state, signal handling information, process data and stack segments), and the time spent on writing the checkpoint in a disk.

Because of the long MTBF of modern computer systems, failures are considered an exception rather than a rule, therefore the overhead of application rollback can hardly affect its execution time. However, for on-line distributed applications (e.g flight reservation systems, industrial decision support systems), the disruption of distributed services must be within acceptable limits to the system operator. The average rollback time for the synthetic applications experiment was *6.5* seconds.

6.2 Benchmarking FADI using a Synthetic Application

This study evaluates FADI's performance for managing fault-tolerance for message-passing applications. Therefore, a test application was designed which is communication-intensive and with small computation cycles, so that the measured overhead will mainly be due to interprocess communication. Four processes

were distributed amongst two SPARCstation *IPCs*. The first generates random numeric data, passes it to the second, where it is hashed by a simple arithmetic operation, and the same is repeated by the third process. The fourth process holds the hash keys for the second and third processs. It decodes the received data package and sends it back to the first process where it is checked against the original values to verify that the data integrity was maintained throughout the pipeline.

6.2.1 Experimental Results

Figure 5 shows the affect of varying the rate of messages exchanged between the application processes on the failure-free overhead of the application. Incrementing the message rate mainly affects the time spent on augmenting *each* message with bookkeeping information (12 bytes containing: State interval, sender ID, send sequence number), and tracking the dependencies of the exchanged messages, i.e keeping records of the last sent and received message for every application. This overhead is persistent as long as there are messages sent/received between the application processes, while the message logging overhead affects the execution of the application only if a message crosses the recovery line due to our selective message logging policy. Subsequently the failure-free overhead gradually grows, but even with a considerable message exchange rate (6.4 Kbyte/sec) the overhead was measured at *2.9%* which comfortably falls within the widely acceptable *"10%"* overhead on application execution time [14].

Sens in [13] implemented an independent checkpointing scheme with pessimistic message logging, and for a much less communication-demanding application (0.06 Kbyte/sec) the overhead was *3.0%* with a similar checkpointing interval of *2min*.

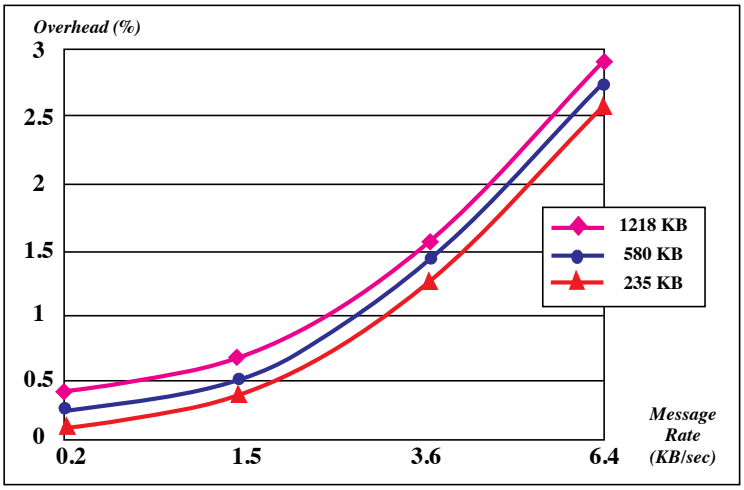

Figure 5 **Failure-Free Overhead as a Function of Message Exchange Frequency**

From Figure 5 it can also be noticed that changing the size of the application processes while varying the message rate has little affect on the failure-free overhead, the curves for various heap sizes of the application process almost overlap.

Figure 6 (a) illustrates the checkpointing overhead of the application for three image sizes versus the frequency of taking the checkpoints. For the three variations of the application the overhead increases as the checkpointing interval is reduced, the reason being that with smaller intervals (higher checkpointing frequencies), more checkpoints are taken. When employing the non-blocking checkpointing policy, the checkpointing procedures are *virtually* performed by a forked thread while the main program concurrently continues execution. However, some execution time is still lost when the OS kernel makes the context switch and swaps one of the processes out. This slight increase in the overhead is escalated with large image sizes (more Kbytes to write to disk) and hence the deviation in the overhead of the three curves in Figure 6 (a) as the checkpointing interval decreases.

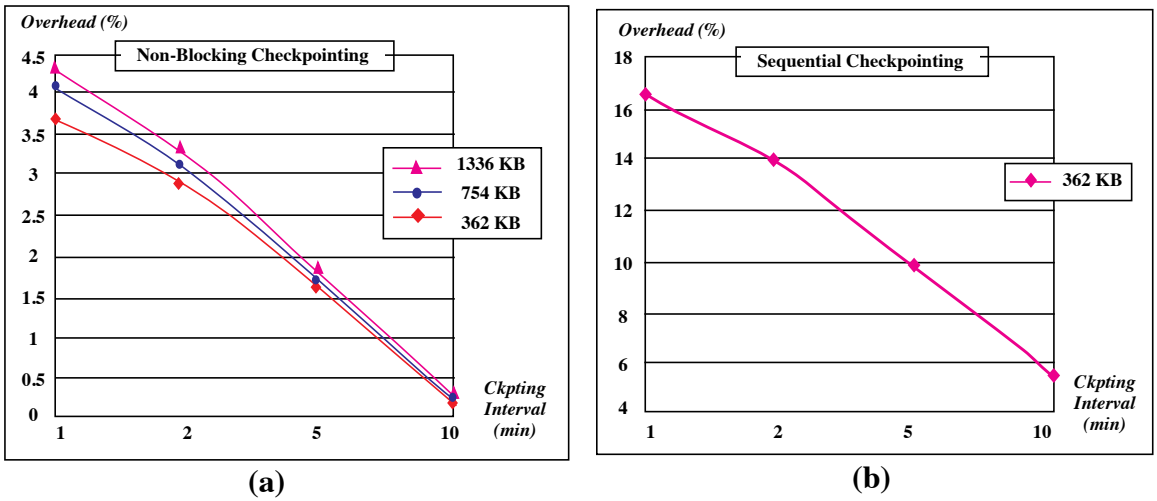

Figure 6 **Failure-Free Overhead as Function of the Checkpoint Interval**

Under similar message rate and image size conditions, Elnozahy and Zwaenepoel's *message logging with coordinated checkpointing* algorithm [8] scored less overhead for higher checkpointing frequencies (*1.8%* for 2min interval), but our algorithm performs better as the checkpointing frequency decreases(*0.2%* overhead as opposed to *0.3%* for 10min checkpointing interval). This is despite of the added complexity to our algorithm caused by catering for failures occurring whilst interprocess messages are in transient. Figure 6 (b) highlights the advantage of non-blocking checkpointing. For the most moderate overhead conditions (362

KB applications size and 10 min. checkpointing interval) the overhead of sequential checkpointing was *5.72%* as opposed to *0.12%* for non-blocking checkpointing.

This performance study indicates that FADI's reliable distributed computing protocol exhibited low overhead even with overestimated message exchange rates, which proves the system feasibility for communication-intensive distributed applications. However, larger overheads (although still acceptable) were measured for shorter checkpointing intervals. Because of the transparency of the implementation of our reliable distributed computing algorithm, the checkpointing frequency can be tuned without any modification to the application code. The requirements of FADI targeted applications (extended execution time) allow for higher checkpointing frequencies and will therefore bear low overhead when run under FADI.

We conjecture that these results will hold on modern hardware platforms as well. Fast Ethernet networks can deliver communication rate up to a Gigabit per second and the latest Sun SPARCstation *"ULTRA 80"* clocks 450 MHz and can accomodate 4GB of RAM.

7. Status and Conclusions

FADI was designed to provide a fault-tolerant distributed environment that provides distributed system users and parallel programmers with an integrated processing environment, where they can reliably execute their concurrent (distributed) applications despite errors that might occur in the underlying hardware. This integrated environment encompasses all aspects of modern fault-tolerant distributed computing: automatic remote process allocation, detection of hardware errors, and a technique to recover distributed user processes running under FADI from these errors.

FADI user-transparent error detection mechanism covers processor node crashes and hardware transient failures, and allows for the integration of user-programmed error checks into the detectable errors database. A limitation of the algorithm is that it assumes that the processing nodes are fail-silent, i.e they deliver correct output or none at all. Therefore, certain faults in the communication link (e.g messages delivered with erroneous content or network partitioning) are not considered. However, duplicate message are tolerated.

A non-blocking checkpointing policy was adopted to backup and restore the state of the application processes. The checkpointing mechanism forks an exact copy (thread) of the application program, this thread performs all the checkpointing routines without suspending the execution of the application code, thus sig-

nificantly reducing the checkpointing overhead. In addition, the checkpointing mechanism comprises a model for restoring the user-files in case of failure. It uses a combination of copy-shadowing and file size bookkeeping to undo modifications to user-files upon rollbacks.

In order to coordinate the operation of the checkpointing mechanism in distributed computing environments, a novel approach to reliable distributed computing for messages-passing applications was devised. It takes advantage of the low failure-free overhead of coordinated checkpointing methods with logging messages that cross the recovery line to avoid blocking the application process during the checkpointing protocol. The low failure-free overhead is at the expense of a longer rollback time which is admissible because of the extended execution time of the targeted application. In contrast with similar algorithms, this technique is tolerant to errors occurring whilst messages are in transit, i.e messages are delivered to the destination (queued at message passing daemon or transport protocol thread), but not yet requested (consumed) by the receiving process. The algorithm requires only one global checkpoint to be recorded in stable storage and avoids multiple rollbacks (domino effect).

The experimental results gained by exhaustive testing of the FADI environment with synthetic applications showed that the system compares favourably with similar fault tolerant environments and exhibits low-overhead even with an over-estimated message exchange rate.

8. References

[1] L. Riberio *et. al.* "Numerical simulations of liquid-liquid agitated dispersions on the VAX 6250/VP", *Computing Systems in Engineering"*, Vol. 6, Iss. 4-5, p. 465-469, Oct. 1995.

[2] W. Lamotte, K. Ellens. "Surface Tree Caching for Rendering Patches in a Parallel Ray Tracing System", *Proceedings of the Conference on Scientific Visualisation of Physical Phenomena*, p. 189-207, 199

[3] A. Bargiela and J. Hartley, "Parallel Simulation of Large Scale Water Distribution Systems", *Proceedings of the 9th European Simulation Multiconference,* 1995.

[4] D. Powell *et al.* "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", *Proc. of the 18th International Symposium on Fault-Tolerant Computing, p. 246-251,* June 1988.

[5] Taha Osman, and Andrzej Bargiela, "Error Detection For reliable Distributed Simulations", *In proceedings of the 7th European Simulation Symposium*, p. 385-362, 1995.

[6] K. P. Birman and R. V. Renesse. “Reliable Distributed Computing with the ISIS Toolkit”, *IEEE Computer Society Press,* 1994.

[7] B. Appel *et al*. “Implications of Fault Management and Replica Determinism on the Real-Time Execution Scheme of VOTRICS”.

[8] E. Elnozahy and W. Zwaenepoel. “On the Use and Implementation of Message Logging”, *Digest of Papers. The 24th International Symposium on Fault-Tolerant Computing,* p. 289-307, June 1994.

[9] T. Osman and A. Bargiela, “Process Checkpointing in an Open Distributed Environment”, *In the Proc. of the 11th European Simulation Multiconference*, p. 536-541, Turkey, June 1997.

[10] L. Silva and G. Silva. “Global Checkpointing for Distributed Programs”, *Proc. of the 11th Symposium on Reliable Distributed Systems,* Huston, Texas, p. 155-162, Oct. 1992.

[11] A. Geist *et al*. “PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing”. *The MIT Press,* Cambridge, Massachusetts, 1994.

[12] G. Janakiraman and Yuval Tamir, “Coordinated Checkpointing-Rollback Recovery for Distributed Shared Memory Multicomputers”, *Proc. The 13th Symposium on Reliable Distributed Systems, p. 42-51,* CA, 1994

[13] P. Sens, “The performance of Independent Checkpointing in Distributed Systems”, *Proc. The 28th Hawaii International Conference on Systems Sciences,* January 1995.

[14] J. S. Plank and K. Li., “A Consistent Checkpointer for Multi-computers”, *IEEE Parallel & Distributed Technology,* 2(2), p. 62-67, 1994

.

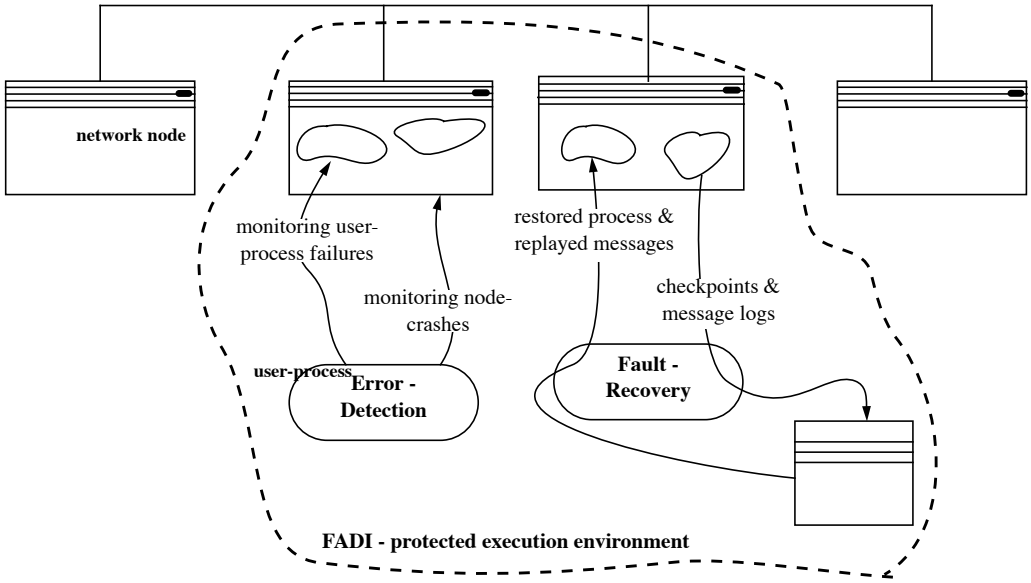

Figure 1 **FADI General Structural Design**

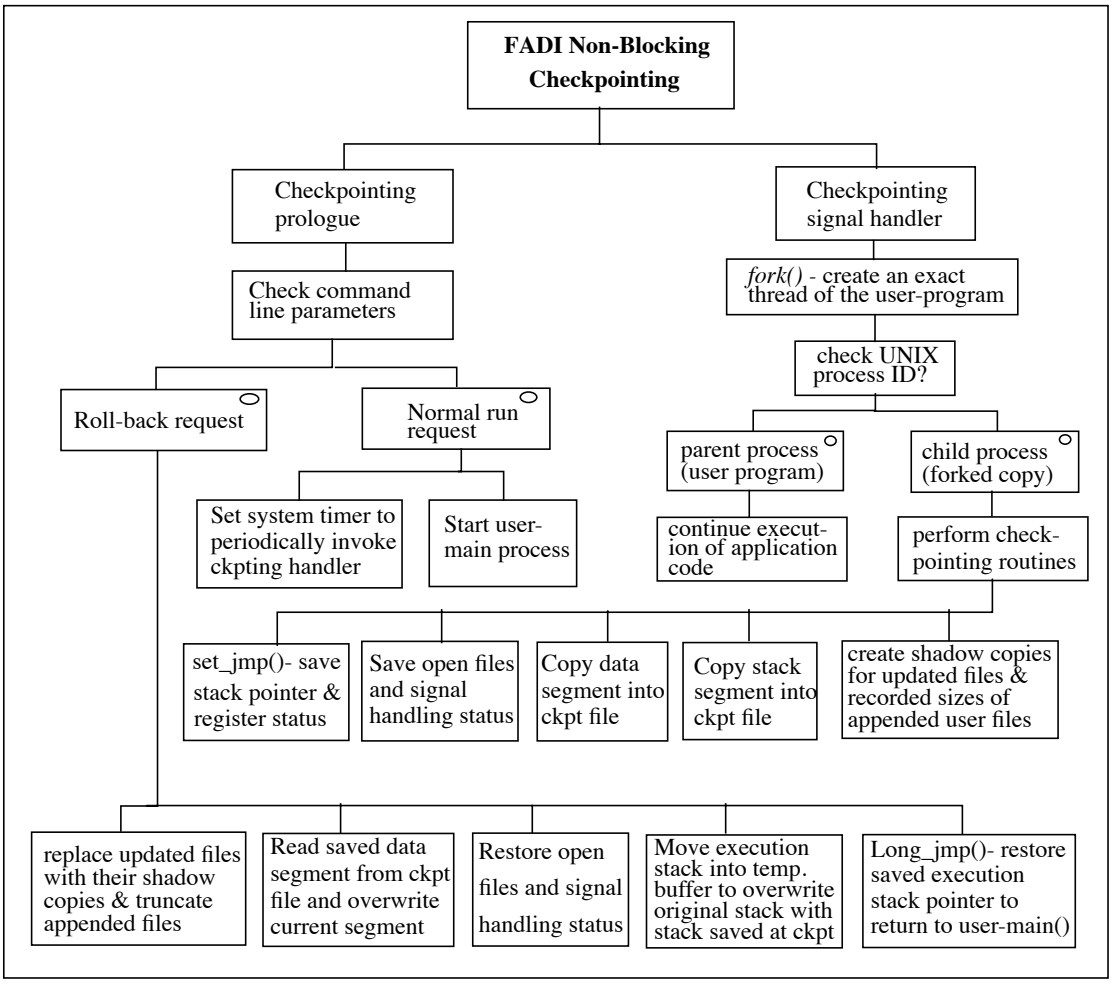

Figure 2 **Checkpointing & Rollback in FADI**

:

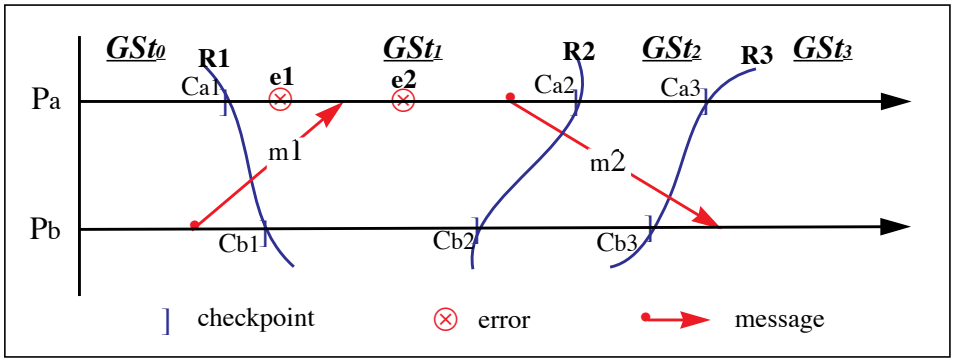

Figure 3 **Recovery-line Consistency (a)**

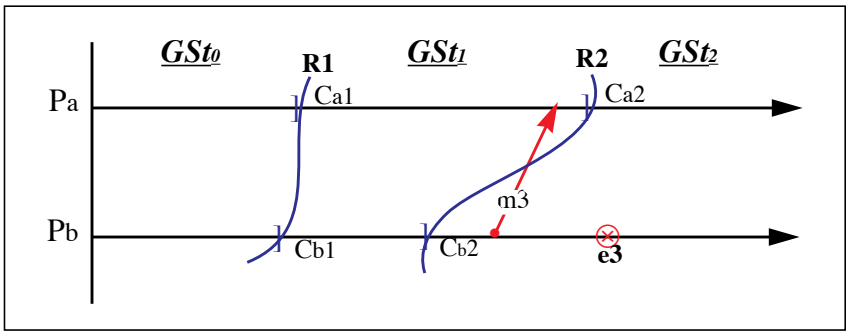

Figure 4 **Recovery-line Consistency (b)**

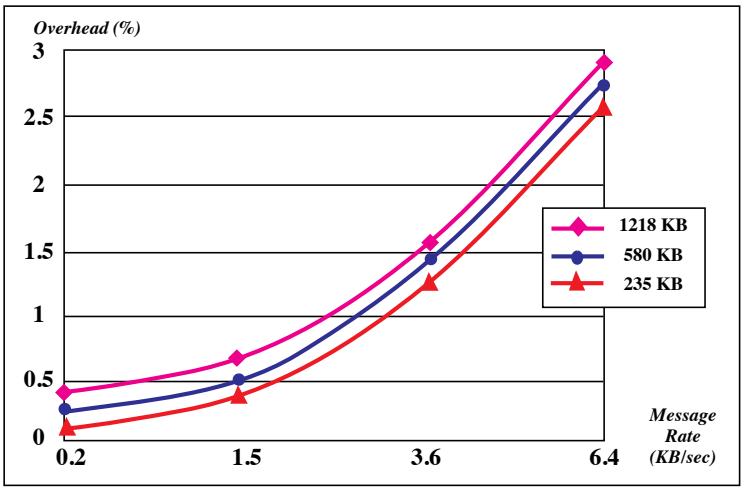

Figure 5 **Failure-Free Overhead as a Function of Message Exchange Frequency**

.

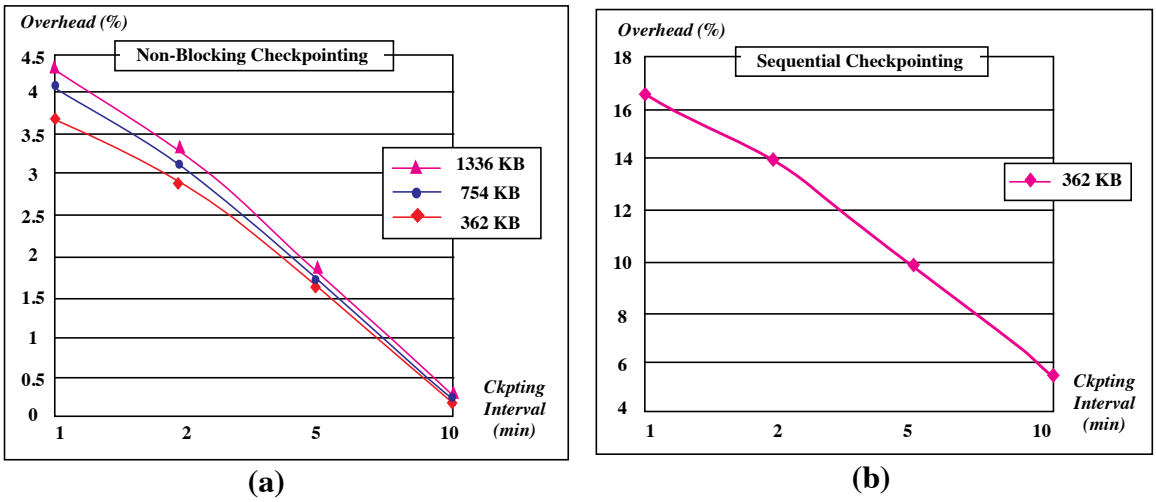

Figure 6 **Failure-Free Overhead as Function of the Checkpoint Interval**