ERROR DETECTION FOR RELIABLE DISTRIBUTED SIMULATIONS

Taha Osman, Andrzej Bargiela

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

ABSTRACT

The field of distributed computing has witnessed an explosive expansion during the last decade. As the use of distributed computing systems for large scale simulations is growing, so is the requirement to increase their reliability. Despite the effort put in designing and maintaining distributed computing systems, non-negligible number of faults occur either due to external influences (e.g power failures), or indeed by reason of human error in the design of the software or the underlying hardware. The detection of these faults is the prerequisite of error recovery mechanisms which are designed as separate software modules.
This paper presents **E**rror **D**etection mechanism for **D**istributed **S**ystems (EDDS), which offers an efficient user-transparent mechanism for the detection of processor node crashes and hardware transient failures (e.g bus errors, segmentation faults, etc.). The system also enables integration of user-programmed error checks into the error detection mechanism. In addition, EDDS provides an approximate measurement of the error latency to allow for damage confinement and assessment. The current version of EDDS was implemented using the PVM message passing interface and was tested on SPARC workstations running SunOS 4.1 and SOLARIS 2.4 and HP workstations running HP-UX08.

1. INTRODUCTION

Computational complexity of many real-life problems necessitates the use of distributed computing systems. However, although the computing hardware is generally reliable, for a multi-processor distributed systems the probability of failure of an individual processing node is non-negligible [VoDe93]. Hence, it is necessary to develop mechanisms that prevent the waste of computations accomplished on distributed processing nodes when one of the nodes fails.

Many algorithms have been devised for fault-tolerant computing: e.g multiplication of the underlying hardware [BiDe94], full software redundancy [SiSw82], or checkpointing & rollback techniques [ELZw92]. Whichever fault-tolerance technique is used, the starting point, and probably the most crucial is the detection of an *erroneous* system state, i.e state which, in the absence of any corrective actions, could lead to the failure of the system. Thus the success of any fault tolerant system is critically dependent upon the effectiveness of the techniques for error detection. An important assertion of a fault tolerant system is its ability to determine the *latency* of the error (i.e the length of time between the occurrence of the error and the appearance of a system failure [John89]). This is vital for damage confinement and assessment, particularly because in distributed systems the error might propagate between system components (tasks, nodes) during the delay between the occurrence of the error and its detection.

While extensive research has been carried out on *fault-injection & error detection* in dedicated fault-tolerant systems, there is little reported work on *error detection mechanisms* (EDM) for distributed systems. Furthermore, most of the fault-tolerance techniques for distributed systems reported in the literature (e.g [ELZw92] [SeFo93]), assume that the processor nodes are *fail-fast*, which is a very restrictive assumption since it imposes that the error detection mechanisms should have zero latency.

This paper presents EDDS, an **E**rror **D**etection mechanism for **D**istributed **S**ystems. It offers an efficient user-transparent mechanism for the detection of processor node crashes, hardware transient failures (e.g bus errors, segmentation faults, etc.), as well as offers the application programmer the possibility to include his/her own error checks in the program and report the failure to the error detection mechanism. In addition, EDDS also provides an approximate measurement of the error latency to be used by subsequent error recovery procedures. The current version of EDDS was implemented and tested on SPARC workstations running SunOS 4.1 and SOLARIS 2.4 and HP workstations running HP-UX08.
The outline of this paper is as follows: Section 2 describes the system model; the message passing interface and system implementation are described in Section 3; Section 4 is concerned with evaluating the system performance and presents conclusions.

2. SYSTEM MODULE

2.1 Assumptions

We assume that the distributed system consists of a number of nodes (processors) that can run concurrent user-tasks. Nodes communicate via a message passing interface over an asynchronous network.
We assume that the processing nodes are fail-silent, i.e they only send correct messages, or nothing at all. However, we do not require the processes to be fail-safe, i.e with a zero error latency. We also assume that a central host will run the main error detection tasks. This central host must be fault-tolerant, i.e the probability of its failure is negligible. The required reliability of the central host might be obtained by hardware duplication, but the detail of achieving it is beyond the scope of this paper.

It's worth pointing out that this paper does not describe new (enhanced) error detection mechanisms at the CPU (kernel) level. Instead it focuses on the use of system-provided EDMs and offering an opportunity to include application-specific error checks, within an integrated Fault-Tolerant distributed environment.

2.2 System Design

Throughout the work, emphasis was put on the simplicity and modularity of the design to ensure that the EDDS system can be smoothly integrated into a Fault-Tolerance System that will provide process recovery from the detected errors.

At the initialisation stage the EDDS accepts validated system specifications (host & task specifications entered by user are checked against the network configuration) from the user/programmer as shown in Fig.1.
These specifications are then broadcasted to the *Host* and *user-tasks* monitoring subtasks, they detect host crashes_&_recovery and user-tasks failure_&_recovery and subsequently update the *active host* and *active task* tables. These tables are shared with the fault-tolerance system *recovery mechanism* and are used in the recovery process of the failed tasks. IDs of manually recovered hosts are also fed to the system for them to be subsequently monitored by the EDDS.

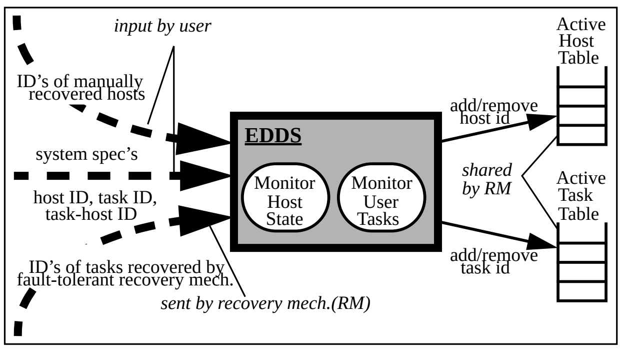

Fig. 1. Context Diagram of the EDDS system

2.2.1 Detecting Host Failures

Detection of host failures is based on a central host monitoring task, running on the master host, which is responsible of the coordination of host crash detection in all the system nodes. This central host monitoring task (Fig. 2) periodically sends acknowledgment requests to all the hosts in the system.
Each host must reply to the acknowledgment request within a predefined time interval "tack_timeout", otherwise it will be considered by the monitoring task as having "crashed". If the reply is received at a latter acknowledgment cycle (because of network delay) or if a message is sent by the user declaring that the host has been manually recovered, then the host is considered as "recovered", and it is added to the system host pool.

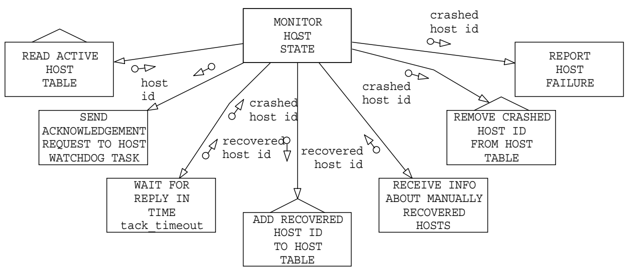
MONITOR HOST STATE
READ ACTIVE HOST TABLE
SEND ACKNOWLEDGEMENT REQUEST TO HOST WATCHDOG TASK
WAIT FOR REPLY IN TIME tack_timeout
ADD RECOVERED HOST ID TO HOST TABLE
RECEIVE INFO ABOUT MANUALLY RECOVERED HOSTS
REMOVE CRASHED HOST ID FROM HOST TABLE
REPORT HOST FAILURE
host id
crashed host id
recovered host id
recovered host id
crashed host id
crashed host id

Fig. 2. The Host Monitoring Task

This approach makes effective use of fault-tolerant master-host and it affords flexible mapping between the logical and physical connectivity of nodes.

An alternative technique used in STAR [SeFo93], operates a *logical ring of crash detection*, where each host only checks it's immediate successor in the ring. Although this technique reduces the message traffic, it introduces a limitation associated with the dependence on network structure. The logical structuring of the crash detection ring must match the physical connectivity of the nodes in the network, which is not always possible, e.g the user/programmer might choose to omit certain nodes from the logical structure of the distributed system.

2.2.2 Detecting User-Tasks Failures

Process failures caused by hardware errors (segmentation fault, bus error, etc.) are detected by the kernel (operating system) EDM. Upon the detection of an error the kernel EDM kills the affected process with a signal number that corresponds to the error type. Hence the user-tasks monitoring process operates as follows: It initially receives the ID's of the active user-tasks, and waits indefinitely for a task to exit. When a user-task exits, the monitoring process analyses the task exit status to determine whether it exited normally or due to a failure. In both situations the task is removed from the active tasks table, but in case of task failure, the fault is reported to the EDDS

system as illustrated in Fig. 3.
The monitoring process also receives notifications from the fault-tolerance recovery mechanism about the recovered tasks and includes their ID's in the active tasks table.

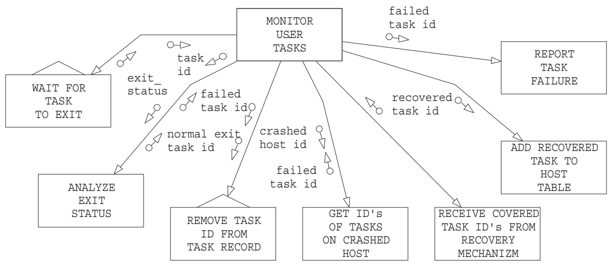

Fig. 3. User-Task Monitoring

With regard to the User (Application Programmer) organized error detection (e.g reversal checks, reasonableness checks, structural checks, etc. see [AnLe81] pg. 115), the application programmer must terminate the failed process with an appropriate exit status so that the EDM can identify the process as failed.

3. Implementation of the EDDS system

3.1 The message passing interface

In the interest of portability, the distributed computing system used for the implementation of EDDS consists of a network of workstations running **PVM** (**P**arallel **V**irtual **M**achine) software. PVM permits a heterogeneous collection of Unix computers linked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The software is very portable. The source, which is available free through netlib, has been compiled on a range of computing machines from laptops to super computers [GeBe94].

PVM offers via a high-level interface most of the facilities required for distributed programming: process control, inter-task communication, and process synchronization in a heterogeneous environment. The main PVM routines used for the implementation of the EDDS system are listed below:
pvm_spawn() - starts a new process on the specified host;
pvm_barrier() - blocks the calling process until all processes in a group have called it;
pvm_send() - sends a messages to another process (including processes in remote hosts);
pvm_recv() - receives a message (the calling process is blocked until msg is received);
pvm_nrecv() - non-blocking receive;
pvm_trecv() - receive with timeout;
pvm_notify() - requests notifications of a certain event in the parallel machine.

The use of the PVM interface for process control and interprocess communication requires a minimum level of application code modifications. In addition to linking applications with the EDDS library, the application code must include two statements at the start and the end of *the main()* function: *pvm_mytid()* and *ed_pvm_exit()*. The first routine is necessary to enrol the task in the PVM system and the second is for the EDDS to identify the task upon successful termination.
The following subsection gives brief description of the implementation of the EDDS system in a UNIX environment using the PVM message passing interface.

3.2 Terms and Definitions

The following terms are used in the functional description of the EDDS system below (Figures 4 and 5):

active_hosts: nodes that are currently considered as "operative" in the distributed system;
tack_timeout: maximum acceptable interval to wait for host acknowledgment. This variable is dynamically updated reflecting the network load;
host_group: dynamic group of processes that contains the *Host_Monitoring* task running on the master host and the *host_watchdog* tasks running on active_hosts;
hn: number of active_hosts;
treply: response time of the previous host to the ack._request;
active_tasks: user-tasks currently running in the distributed system;
received_message_type: identification tag attached to the message;
Successful_Termination: user-task was executed successfully;
Wait_Exit: user-task exited;
Task_Recovered: failed user-task recovered by the fault-toleracne recovery mechanism.

Functional description of the EDDS tasks is shown in Figures 4, and 5. Host crash detection is initialised by starting the *host_watchdogs* on the active hosts. The sole task of a host_watchdog is to send acknowledgment message to the central *HostCrashDetection* task upon the receipt of an ack._request. pvm_barrier() is used to synchronize communication between the *HostCrashDetection* task and *HostWatchdog* tasks.
After each detection cycle, the EDDS scans for recovered hosts. Firstly it checks if there is any pending acknowledgments from hosts considered as crashed at the previous detection cycle(s). The delay in receiving these messages might have been caused by network overload, but the host was considered failed regardless, because the delay exceeded *tack_timeout*. Contact with the recovered host is reinstated. Next the system checks if there is any notification sent by the user about manually recovered hosts.

The recovered host is added to the active_hosts pool, and the watchdog task is restarted on the recovered host.

```
Initialize()
    pvm_spawn( host_watchdog on active_hosts )
    pvm_barrier( host_group, hn+1 )
    HostCrashDetection()
end Initialize

HostCrashDetection()
    repeat
        pvm_send(ack._requests to host_watchdogs in active_hosts )
        for each active_host do
            pvm_trecv( replies to ack._requests from host_watchdog /
                                  in tack_timeout )
            if reply not received then
                remove host from active_host_table
                report the host failure
                set tack_timeout to zero
            else
                discount treply from tack_timeout
            endif
        end
        ScanForRecoveredHosts()
    until stopped
end HostCrashDetection

ScanForRecoveredHosts()
    pvm_nrecv( delayed reply from failed (considered crashed) host )
    if reply received then
        add host to active_host_table
    endif
    pvm_nrecv( notification about manual host recovery from user )
    if notification received then
        pvm_spawn( host_watchdog on recovered host )
        add host to active_host_table
    endif
end ScanForRecoveredHosts
```

Fig. 4a. Central Host Monitoring Task

```
HostWatchDog()
    pvm_barrier( host_group, n+1 )
    repeat
        pvm_recv( ack._request from detection task on master host )
        pvm_send( reply to the ack._request )
    until stopped
end HostWatchDog
```

Fig. 4b. Host Watchdog Task

Monitoring the user-tasks starts by executing pvm_notify(). With *TaskExit* as an argument, this function causes all user-tasks registered in the distributed system to send notification messages with message tag "Wait_Exit" to the calling task upon their exit. However, this notification does not manifest itself if the task exited abnormally or due to a fault. In order to overcome the last problem the pvm_exit() routine has been overloaded (see section 3.1), and it automatically sends a message tagged with "Successful_Termination" declaring the successful execution of the user-task. Because PVM guarantees the order of messages delivered from one source, if a Wait_Exit message is received before Successful_Termination message from the same task, this means that a failure occurred and the task was abnormally aborted.

As can be noticed from Fig. 5, the monitoring task is *message driven*, i.e managed by analysing messages sent either by the user-task (Wait_Exit and Successful_Termination) or the fault-tolerance recovery mechanism (Task_Recovered).

```
MonitorUserTasks()
    pvm_notify( about TaskExit of active_tasks with msg_type /
                  Wait_Exit)
    repeat
        pvm_recv( any message from active_tasks )
        case received_message_type of :
            Successful_Termination:
                add task to normal_exits record
                break
            Wait_Exit:
                if task in normal_exits record then
                report successful task termination
                else
                report task failure
                endif
                remove task from active_task_table
                break
            Task_Recovered:
                add task to active_task_table
                pvm_notify( about TaskExit of recovered_task )
                break
        end case
    until all tasks exited successfully
end MonitorUserTasks
```

Fig. 5. Monitoring User-Tasks

Taking into consideration the centralised error detection and the delays to the message passing interface that might be caused by the overload of the network, the error latency of detecting task failures can be calculated by the formula:

$$Tlatency = E(Tedm) + 3 \times \sigma(Tedm) + Trtt \qquad \textbf{(1)}$$

where: *E(Tedm)* - average reaction time of the kernel error detection mechanism;
σ*(Tedm)* - variance of the *Tedm*;
Trtt - estimated round-trip time from the master host to active_hosts

The error latency of host crash detection is determined by time intervals between the *host crash detection* cycles.

We envisage developing an error recovery mechanism based on periodic checkpointing and roll-back techniques [DeCu93], where the latency of the error will be crucial for limiting the extent of the process roll-back to previous checkpoints.

4. Test Results

The EDDS system performance was tested by simulating Hardware faults. Two *Host crash* simulations were performed:
1) powering down one of the distributed system nodes (SPARCstation IPC). In this case, the error latency was 1.5 seconds upon low network load (Trtt = 70 msec) and

acknowledgment timeout of ‘20*Trtt’ (Tack_timeout = 1.4 sec);
2) disconnecting a processing node from the Ethernet. The error latency was 1.03 seconds upon (Trtt = 51 msec, Tack_timeout = 1.02 sec). When the connection was restored, the EDDS system identified that the crash was caused by network delay and added the “recovered” host to the active host pool.

Three types of hardware transient failures were simulated to test error detection in user tasks: segmentation fault, bus error, and arithmetic exception.
Table 1 gives a representative sample of results for “bus error”.

Master Host & Arch. / OS	Remote Host & Arch. / OS	Latency (seconds)
Host1 SPARC multiprocessor SOLARIS 2.4	Host1	1.328
_″_	Host2 SPARCstation SOLARIS 2.4	0.802
_″_	Host3 SPARCstation SOLARIS 2.4	0.731
_″_	Host4 SPARCstation SUNOS 4.2	0.291
_″_	Host5 SPARCstation SUNOS 4.2	0.286
_″_	Host6 HP-APPOLO 400 HP-UX08	0.305
_″_	Host7 HP 340 HP-UX08	0.343

Table 1: Error Detection of User Tasks

Since measurements in the above table were taken with approximately the same network load implying the same Trtt, then according to formula (1), the error latency indicates the reaction time of the kernel EDM (Tedm).
It can be noted from table 1 that the nodes running the same operating system have been found to have similar error latency.

Conclusion and future work

This paper presented an efficient user-transparent error detection mechanism for open distributed systems. The detection mechanism covers processor node crashes and hardware transient failures (e.g bus errors, segmentation faults, etc.). The EDDS system also enables integration of user-programmed error checks into the error detection mechanism.
EDDS has been tested on a set of heterogeneous workstations connected by Ethernet and the results show that the system is viable for open distributed computing systems.

We implemented a simple and efficient centralised error detection structure, where the major remote-host & user-process monitoring tasks run on a failure-free master host. However, we assume that the processing nodes are fail-silent, and therefore, we do not consider faults in the communication link (e.g messages delivered with erroneous content, or the sending of extra messages). We do however, cater for a possibility of propagation of errors in the distributed computing system by allowing a non-zero error latency.

Future work will involve integrating the EDDS system into a “Fault-Tolerant Environment for Open Distributed Processing” where mechanisms will be developed for error recovery of failed user tasks.

REFRENCES

[AnLe81] T. Anderson and P.A. Lee 1981. “FAULT TOREANCE Principles and Practice”. Prentice-Hall Int., London.

[BiDe94] B. Bieker and G. Deconink 1994. “Reconfiguration and Checkpointing in Massively Parallel systems”. The FTMPS project.

[DeCu93] G. Deconink, R. Cuyvers, and others 1993. “Survey of Checkpointing and Rollback Techniques”. ESAT-ACCA Laboratory, Katholieke Universitei Leuven, Belgium.

[GeBe94] A. Geist, A. Beguelin, and others 1994. “PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing”. The MIT Press. Cambridge, Massachusetts.

[ElZw92] E. Elnozahy and W. Zwaenepoel 1992. “Mantheo: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit”. *IEEE Transactions on Computers, Vol. 41, No. 5, May 1992*

[John89] B.W. Johnson 1989. “Design and Analysis of Fault-Tolerant Digital Systems”. Addison Welsley Publishing Company Inc.

[SeFo93] P. Sens and B. Folliot 1993. “STAR: a Fault Tolerant System for Distributed Applications”. *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, pp. 656-660, Dec. 1993.*

[SiSw82] D. Siewiorek and R. Swarz 1982. “The Theory and Practice of Reliable System Design”. Digital Press, Bedford, MA, 1982.

[VoDe93] J. Vounckx, G. Deconink, and others 1993. “The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems”. Katholieke Universitei Leuven, Belgium.