ELSEVIER

Information Processing Letters 74 (2000) 257–262

Information Processing Letters

www.elsevier.com/locate/ipl

Resolving error propagation in distributed systems

Jenn-Wei Lin, Sy-Yen Kuo [*]

Department of Electrical Engineering, Rm. 415, National Taiwan University, Taipei, Taiwan

Received 2 May 1998
Communicated by F. Dehne

Abstract

This paper investigates the problem of error propagation in distributed systems. To resolve this problem, a state preservation scheme is presented to save process states in main memory. Based on the state preservation, the processes suffering from error propagation can be recovered without involving stable storage. The recovery overhead is significantly reduced. In addition, a well-known *single-source-all-destination* graph algorithm is also utilized to find the optimal recovery points of the processes suffering from error propagation. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Error propagation; Distributed systems; State preservation; Stable storage; Graph algorithms

1. Introduction

Most of the existing literature investigating the error recovery problem in distributed systems often assumes that an error is detected immediately after it has occurred [1–3]. This assumption is not realistic since it requires that error detection mechanisms must have high detection rate and zero latency. In practice, there is a nontrivial interval between the occurrence of an error and the moment of its detection. This interval is called the *error detection latency*. If a message is sent during error detection latency, the message may be contaminated by the error. Subsequently, when this contaminated message arrives at a destination process and is used in its computation, the destination process will suffer from contamination. This situation can be conceived as an error being propagated to the destination process.

The previous research on the error propagation is based on the checkpointing recovery techniques. There are two main types of checkpointing recovery techniques: coordinated checkpointing and independent checkpointing with message logging. Each of these two techniques has its benefits and drawbacks. Silva and Silva utilized a coordinated checkpointing technique to resolve the error propagation problem [4]. When an error is detected, the system first determines a consistent recovery line. Then, all the processes are rolled back to that recovery line. If a failure-free process does suffer from error propagation, it will be rolled back to its most valid checkpoint to nullify contamination. However, the processes that do not suffer from the error propagation are also rolled back. Krishna et al. employed an independent checkpointing with message logging technique to deal with the error propagation problem [5]. Upon a failure, recovery is initiated to assess the damage caused by the error propagation. If a process is suspected of suffering from error propagation, it is rolled back to one of its previous checkpoint. The previous recovery approaches [4,5] for the processes suffering from error propagation are the same as that for the failed process.

[*] Corresponding author. Email: sykuo@cc.ee.ntu.edu.tw.

Valid checkpoints are first loaded from stable storage, and then the processes suffering from error propagation are restarted from that checkpoints.

In this paper, we will extend the independent checkpointing with message logging technique to provide the capability for handling the error propagation problem. A state preservation scheme is proposed to preserve some process states in main memory. Based on the state preservation, the processes suffering from contamination can be recovered by using the states saved in main memory without involving stable storage. This can significantly reduce the recovery overhead since the stable storage access is an expensive operation. In addition, a well-known *single-source-all-destination* graph algorithm is used to find the optimal recovery points of the processes suffering from error propagation.

The rest of the paper is organized as follows. Section 2 gives some preliminaries. Section 3 presents the approach to resolving the error propagation problem. Section 4 evaluates the overhead of the proposed approach. Finally, we give concluding remarks in Section 5.

2. Preliminaries

The system considered in this paper consists of a set of processing nodes connected through a communication network. The distributed application program is partitioned into a set of processes, and each process is executed in a node. The communication network is assumed to guarantee FIFO delivery of messages between any pair of nodes. The following notations and definitions are used in the rest of paper.

$D_{\max}$: the maximum error detection latency.

$R_{\max}$: the maximum communication delay.

Definition 1. A process is an *error-occurred* process if a failure occurs in the node executing the process.

Definition 2. A process is an *error-contaminated* process if it has received a contaminated message directly or indirectly.

Definition 3. The maximum error notification $N_{\max}$ is the interval between the occurrence of an error and the moment that the event of error occurrence is known by all processes. After an error is detected, an *Error-Notified* message is broadcast to notify an error occurrence. The maximum error notification can be represented as $D_{\max} + R_{\max}$. If a process receives an *Error-Notification* message at time t_N, the earliest time of error occurrence may be $t_N - R_{\max} - D_{\max}$.

3. Error propagation

In the previous research on error propagation [4,5], the error-contaminated process is recovered by loading its most valid checkpoint from stable storage and then restarting from that checkpoint. However, stable storage is usually implemented by disks. Loading a checkpoint from a disk is a time-consuming operation. To eliminate the loading time from disks, a state preservation scheme is proposed in this section. In addition, a *single-source-all-destination* graph algorithm is also utilized to find the optimal recovery points of error-contaminated processes.

3.1. State preservation

An error-contaminated process is injured due to receiving a contaminated message (not due to an internal error). The injured area in the node executing the error-contaminated process is confined to the address space related to that process. If a second address space can be allocated from main memory and is not used to execute user processes, this address space will be intact after receiving a contaminated message. Therefore, if a correct process state is also saved in the intact part in advance, the error-contaminated process can be recovered from main memory instead of stable storage. This significantly reduces the recovery time of the error-contaminated process. The problem next is how to find a correct process state. Since a contaminated message is introduced by a *receive* instruction, the process state prior to executing the receive instruction is a correct state. However, a contaminated message is not identified until the error is detected. To be conservative, whenever a process receives a message, its current process state is preserved in the allocated second address space in advance. Then, if a process is notified to have received a contaminated message, it can be restarted from the point prior to receiving a con-

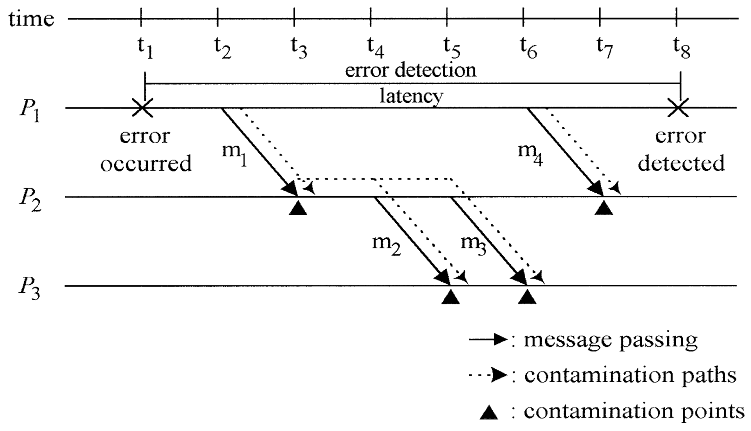

Fig. 1. Interactions among processes during error detection latency.

taminated message and then re-receive a clean message. Here, the process state with respect to a receive message is only preserved for a limited time interval, defined as follows:

Definition 4. The *preservation period* for the process state with respect to a *receive* instruction is $D_{\max} + R_{\max}$.

If a process does not receive an *Error-Notification* message within the maximum error notification latency $D_{\max} + R_{\max}$ after a *receive* instruction was issued, that means the receive instruction was not issued after the error occurrence. The received message can be also guaranteed to be uncontaminated. Therefore, the process state with respect to the issued *receive* instruction is not preserved further.

To reduce both the impact on the regular process execution and the amount of the address space for state preservation, the copy-on-write checkpointing technique [6] is used to assist the state preservation. The copy-on-write checkpointing technique can overlap the execution of a process with the preservation of a process state, as follows.

- Allocate a second address space in main memory.
- The current state of the process is first divided into the CPU state and the memory state.
- The CPU state is entirely saved in the second address space.
- The memory state is incrementally saved in the second address space by individually saving each memory block when it is modified.

3.2. Recovery

Based on the state preservation scheme, we know that the error-contaminated process can be recovered from its contamination point. However, if an error-contaminated process receives more than one contaminated message, it has several contamination points. The recovery point of the error-contaminated process must be the point before it received the first contaminated message. For example, in Fig. 1, the recovery point of process P_2 is the point before message m_1 is received. In addition, the first contaminated message of each error-contaminated process is not necessarily received from the error-occurred process. The first contaminated message may be received from other error-contaminated processes. For example, in Fig. 1, the first contaminated message m_2 of process P_3 is received from error-contaminated process P_2. The recovery point of an error-contaminated process must be determined from the view of the overall system. To systemically find the exact number of error-contaminated processes and their respective recovery points, two process stages: *early analysis stage* and *final determination stage* are done as follows.

The early analysis stage is to collect all *error-suspected* messages (the messages which may suffer from contamination). To accomplish this work, whenever a message is received, some information (the identifier of the sending process and the reception time) pertaining to the received messages is kept. After an error is detected, a failure-free node is elected as the error-contamination recovery manager in addition to broadcasting an *Error-Notification* message. Upon receiving the *Error-Notification* message, each process uses the reception time t_N to determine the earliest time $t_N - R_{\max} - D_{\max}$ of error occurrence (see Definition 3). If a process has received a contaminated message, the reception time of this message must be in the range of $(t_N - R_{\max} - D_{\max}, t_N)$. Next, each process filters its kept information to select the items whose timestamps are in the range of $(t_N - R_{\max} - D_{\max}, t_N)$ and then sends selected items to the error-contamination recovery manager. The messages corresponding to these selected items are error-suspected messages.

The final determination stage is to find error-contaminated processes and their optimal recovery points. The error-contaminated process must be restarted from the point before the first contaminated message is received. The optimal recovery point of an error-contaminated process is the point where it received the first contaminated message. To achieve this work, the relationship among the error-suspected messages is modeled as a weighted directed graph $G(V, E)$. Each vertex in V represents a process, and each edge in E represents an error-suspected message. The weight of a edge is the reception time of the corresponding message. Based on the graph G, if the error-occurred process and other processes are designated as the source node and the destination nodes, respectively, the problem of determining respective first contaminated message of the error-contaminated processes is similar to the well-known *single-source-all-destination* problem [7]. The *single-source-all-destination* problem is how to find all the shortest paths from a given source node to all destination nodes. This is based on the following observations:

- A contaminated message is either directly or indirectly propagated by the error-occurred process. This can be conceived that a contaminated message introduces a contaminated path from the error-occurred process. For example, in Fig. 1, message m_2 introduces a contaminated path $\langle m_1, m_2 \rangle$ from process P_1.
- The first contaminated message of an error-contaminated process is the message with the lowest reception time among all the process's contaminated messages. This observation is equivalent to saying that the first contaminated message of an error-contaminated process is on the contaminated path with the lowest weight among all of that process's contaminated paths. Here, the weight of a contaminated path is defined as the weight of its last edge. For example, in Fig. 1, the first contaminated message of process P_3 is on the contaminated path $\langle m_1, m_2 \rangle$, and the weight of this contaminated path is t_5.

However, the algorithm for the *single-source-all-destinations* problem can not be directly applied to solve the problem of the final determination stage. It needs to be slightly modified, as shown in Fig. 2.

4. Evaluation

In this section, we will evaluate the overhead incurred by the proposed approach. The overhead incurred is measured as: performance degradation incurred and amount of the address space required by the state preservation. The advantages of the proposed approach is also described in this section.

4.1. Overhead

The copy-on-write checkpointing is used to assist the state preservation. The process state (CPU state and memory state) is incrementally saved in the second address space. The performance degradation incurred by the state preservation contains two parts:

- Time to copy the CPU states to the second address space in main memory.
- Time to incrementally copy the memory state to the second address space in main memory.

The first part takes a fixed period of time. The second part is dependent on how many write operations are issued during the preservation period, since the write operations in this period are required to copy the old data. This part can be overlapped with the process execution. The performance degradation is mainly dominated in the time to save a CPU state.

The amount of the address space required by the state preservation can be represented as the total space

```
Procedure Recovery_of_Error-Contaminated_Processes$(G, p)$
/* $G$ is the weighted directed graph formed by all error-suspected messages,
   $p$ is the identifier of the error-occurred process */
p.contamination_ time $\leftarrow t_N - R_{\max} - D_{\max}$ (the earliest time of error occurrence,
                                   $t_N$ is the reception time of the Error-Notification message);
$S \leftarrow \{p\}$; $T \leftarrow \{\}$;
while (True) {
    min_weight $\leftarrow \infty$;
    $(v1_{\text{next}}, v2_{\text{next}}) \leftarrow$ null
    for all $v_1 \in S$ do {
        examine all the outgoing edges of node $v_1$ and then choose one outgoing edge
        $(v_1, v_2)$ with the following three properties:
            (1) minimum weight among the outgoing edges from node $v_1$
            (2) $v_2 \notin S$
                /* the next error-contaminated process can not already exist in set $S$ */
            (3) the weight of $(v_1, v_2)$ must be greater than $v_1$.contaminated-time;
                /* the contaminated message must be sent after the contaminated point */
        if min_weight $>$ the weight of $(v_1, v_2)$ {
            min_weight $\leftarrow$ the weight of $(v_1, v_2)$;
            $(v1_{\text{next}}, v2_{\text{next}}) \leftarrow (v_1, v_2)$ };
        /* next contaminated message is the message with the smallest reception time among all
            contaminated messages sent from $S$ */
    };
    if $(v1_{\text{next}}, v2_{\text{next}})$ is null then
        exit while;
    else {
            $v_2$.contamination_time $\leftarrow$ the weight of $(v1_{\text{next}}, v2_{\text{next}})$;
            $S \leftarrow S \cup \{v_2\}$;
            $T = T \cup \{(v1_{\text{next}}, v2_{\text{next}})\}$;
};   /* end while */
$T$ contains the sets of edges corresponding to the first contaminated messages of error-contaminated processes;
end;
```

Fig. 2. The procedure to determine error-contaminated processes and their recovery points.

required by preserving the CPU states and the memory states. Whenever a receive instruction is issued, a block is allocated to preserve the current CPU state. Based on Definition 4, we know that the process state with respect to a receive instruction needs to be preserved for a constant period $D_{\max} + R_{\max}$. Each allocated block can be reused after the time $D_{\max} + R_{\max}$ elapses. Therefore, the maximum amount CPU_{space}, of the space for preserving CPU states is dependent on the maximum number of receive instructions issued during the preservation period. The memory state is incrementally preserved. If a block is allocated to keep the old data of a memory block at time t, it can be used to preserve another written memory block at time $t + D_{\max} + R_{\max}$. The maximum amount $Memory_{\text{space}}$ of the space for preserving memory states is also determined based on the maximum number of write operations executed during the preservation period. The total space required by the state preservation is:

$$\frac{D_{\max} + R_{\max}}{I_c \times c_t} \times (\text{size of a CPU state}) + \frac{D_{\max} + R_{\max}}{I_c \times c_t} \times (\text{size of a memory block}), \qquad (1)$$

where c_t is cycle time, and I_c is average number of clock cycles per instruction.

4.2. Advantages

Compared to the approaches of [4,5], our approach has the following advantages.

- The recovery point of each error-contaminated process is an optimal location since the recovery point is at the point of receiving the first contaminated message. Unlike the approaches [4,5], each error-contaminated process is recovered from its most valid checkpoint. The most valid checkpoint of an error-contaminated process may be also far from its current execution point. The rollback distance can be reduced by the proposed approach.
- The error-contaminated processes are recovered by using the process states preserved in the main memory. In the previous approaches [4,5], the error-contaminated processes are recovered by loading their most valid checkpoint from stable storage. This significantly saves the recovery time.

5. Conclusions

In this paper, a state preservation scheme has been presented to handle the problem of error propagation. In addition, a well-known *single-source-all-destination* graph algorithm is also employed to find the optimal recovery points of error-contaminated processes. Based on the proposed approach, the error-contaminated process can be recovered from its first contaminated point instead of the most recent checkpoint. The stable storage access is also not involved in the recovery. Thus, the recovery overhead can be significantly reduced. The evaluation indicates that the performance degradation incurred and the amount of the address space required by the state preservation are dependent on the maximum detection latency.

References

[1] R. Koo, S. Toueg, Checkpointing and rollback recovery for distributed systems, IEEE Trans. Software Engrg. SE-13 (1) (1987) 23–31.

[2] E.N. Elnozahy, W. Zwaenepoel, Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit, IEEE Trans. Comput. 41 (5) (1990) 526–531.

[3] J.L. Kim, T. Park, An efficient protocol for checkpointing recovery in distributed systems, IEEE Trans. Parallel Distrib. Systems 4 (8) (1993) 955–960.

[4] L.M. Silva, J.G. Silva, Global checkpointing for distributed programs, in: Proc. IEEE Symp. on Reliable Distributed Systems, 1992, pp. 155–162.

[5] P. Krishna, N.H. Vaidya, D.K. Pradhan, Recovery in multicomputers with finite error detection latency, in: Proc. 24th Internat. Symp. on Fault-Tolerant Computing, 1994, pp. 155–162.

[6] K.L. Jeffrey, F. Naughton, J.S. Plank, Low-latency, concurrent checkpointing for parallel programs, IEEE Trans. Parallel Distrib. Systems 5 (8) (1994) 874–879.

[7] U. Manber, Introduction to Algorithms A Creative Approach, Addison-Wesley, Reading, MA, 1989.

[8] B. Janssens, W.K. Fuchs, Relaxing consistency in recoverable distributed shared memory, in: Proc. 23rd Internat. Symp. on Fault-Tolerant Computing, 1993, pp. 155–163.