비동기적 검사점 기록을 고려한 저 비용 인과적 메시지 로깅 기반 회복 알고리즘

안 진 호[†] · 방 승 준[††]

요 약

인과적 메시지 로깅을 위한 기존 회복 알고리즘들에 비해, Elnozahy가 제안한 회복 알고리즘은 안전한 저장소 접근횟수를 매우 줄이고, 회복과정을 수행하는 동안 살아있는 프로세스들이 자신의 계산을 계속해서 수행할 수 있도록 한다. 그러나, 인과적 메시지 로깅 기법이 비동기적 검사점 기록 기법과 함께 사용된다면, 동시적 고장들이 발생하는 경우 이 알고리즘 수행 후 전체 시스템 상태가 일관적이지 못하게 될 수 있다. 본 논문에서는 이러한 일관적이지 못한 경우들을 보여주고, 이러한 문제점을 해결하는 인과적 메시지 로깅을 위한 저 비용의 회복 알고리즘을 제안한다. 시스템 일관성을 보장하기 위해, 이 알고리즘은 회복 리더가 모든 살아있는 프로세스들뿐만 아니라 다른 회복 프로세스들로부터 회복정보를 얻을 수 있도록 한다. 또한, 제안된 알고리즘은 Elnozahy 회복 알고리즘에 비해 어떠한 부가적인 메시지도 요구하지 않으며, 메시지 피기백에 의해 발생되는 제안된 알고리즘의 부가적인 비용이 매우 낮다. 이를 입증하기 위해, 시뮬레이션 결과는 제안된 알고리즘이 Elnozahy 알고리즘에 비해 회복정보 수집시간을 단지 1.0%~2.1% 정도로 증가시킴을 보여준다.

키워드 : 분산 시스템, 결함포용성, 검사점 기록, 메시지 로깅, 회복 알고리즘

Low-Cost Causal Message Logging based Recovery Algorithm Considering Asynchronous Checkpointing

Jin-Ho Ahn[†] · Seong-Jun Bang[††]

ABSTRACT

Compared with the previous recovery algorithms for causal message logging, Elnozahy's recovery algorithm considerably reduces the number of stable storage accesses and enables live processes to execute their computations continuously while performing its recovery procedure. However, if causal message logging is used with asynchronous checkpointing, the state of the system may be inconsistent after having executed this algorithm in case of concurrent failures. In this paper, we show these inconsistent cases and propose a low-cost recovery algorithm for causal messge logging to solve the problem. To ensure the system consistency, this algorithm allows the recovery leader to obtain recovery information from not only the live processes, but also the other recovering processes. Also, the proposed algorithm requires no extra message compared with Elnozahy's one and its additional overhead incurred by message piggybacking is significantly low. To demonstrate this, simulation results show that the first only increases about 1.0%~2.1% of the recovery information collection time compared with the latter.

Key Words : Distributed System, Fault-Tolerance, Checkpointing, Message Logging, Recovery Algorithm

1. 서 론

로그 기반 복귀회복(log-based rollback recovery)은 분산 시스템을 위해 결함포용성(fault-tolerance)을 제공하는 잘 알려진 기법이다[7]. 이러한 메시지 로깅 프로토콜들은 비관적(pessimistic)[4, 11, 15], 낙관적(optimistic)[13, 14], 인과적(causal)[1, 8]이라는 세 가지 종류의 기법들로 나누어진다. 이 세 기법들 중 인과적 메시지 로깅 기법은 다음과 같은 장점들을 가지고 있다[3]. 먼저, 각 프로세스는 고아 프로세스(orphan process)를 발생시키지 않기 위해 자신의 휘발성 저장소에 있는 로그 정보를 모든 송신 메시지에 피기백(piggyback)한다. 따라서, 이 기법은 어떠한 살아있는 프로세스도 복귀시키지 않으면서, 각 고장난 프로세스가 안전한 저장소에 저장된 자신의 최신 검사점 기록 상태까지만 복귀하도록 한다. 또한, 이 기법은 의존성 있는 프로세스들(dependent processes)의 휘발성 저장소를 이용함으로써 동시적인 프로세스 고장을 해결할 수 있다. 두 번째로, 각 프

† 종신회원 : 경기대학교 정보과학부 전자계산학과 조교수
†† 준 회 원 : 경기대학교 정보과학부 전자계산학과 석사과정
논문접수 : 2006년 3월 2일, 심사완료 : 2006년 10월 9일

로세스는 메시지의 로그정보를 안전한 저장소에 비동기적으로 저장한다.

이러한 바람직한 특성들을 가진 대표적인 세 개의 인과적 메시지 로깅 프로토콜들이 있는데, 동시적 프로세스 고장 발생시 다음과 같은 문제점들을 가지고 있다. 먼저, 가족기반(family-based) 메시지 로깅 프로토콜[1]은 회복 시 살아있는 프로세스들이 계속적으로 계산(computation)을 수행하지 못하도록 한다. 이러한 바람직하지 못한 속성은 프로세스 회복 시 높은 비용을 발생시켜 시스템 성능을 매우 저하시킬 수 있다. Manetho의 회복알고리즘[8]은 각 살아있는 프로세스가 회복 프로세스로부터 회복 메시지를 수신한 후 회복정보를 안전한 저장소에 쓰도록 요구한다. 또한, 이 알고리즘은 각 살아있는 프로세스의 상태가 회복 프로세스의 상태와 일관적이지 못하게 될 수 있는 모든 수신된 메시지를 회복과정이 완료될 때까지 애플리케이션에게 전달하지 않는다. 마지막으로 Elnozahy는 안전한 저장소 접근 횟수를 줄이고, 살아있는 프로세스가 동시적 고장이 발생한 경우에도 계속적으로 자신의 계산을 수행할 수 있도록 하는 회복 알고리즘을 제안하였다[6]. 그러나, 이 알고리즘은 비동기적 검사점(asynchronous checkpointing) 기법[5]과 결합되는 경우, 프로세스들이 동시에 고장 난다면 시스템의 일관성을 보장하지 못할 수 있다.

본 논문에서는 이러한 Elnozahy 알고리즘의 일관적이지 못한 경우들을 살펴본 후, 회복 리더가 모든 살아있는 프로세스들뿐만 아니라 다른 회복 프로세스들로부터 회복정보를 얻을 수 있도록 함으로써 어떠한 검사점 기법과 결합하는 경우에도 시스템 일관성을 보장하는 회복 알고리즘을 제안한다. 또한, 이 알고리즘은 Elnozahy 알고리즘에 비해 어떠한 부가적인 메시지도 요구하지 않도록 설계되었다.

본 논문의 이후 구성은 다음과 같다. 먼저, 2절에서는 관련연구를 소개하고, 3절에서는 본 논문에서 가정하는 분산 시스템 모델을 기술한다. 4절에서는 기존 회복 알고리즘의 일관성 문제점을 살펴본 후, 이러한 문제점을 해결하는 새로운 회복 알고리즘을 제안하고 그 알고리즘의 정당성을 증명한다. 마지막으로 5절과 6절에서는 각각 성능평가 및 결론을 제시한다.

2. 관련 연구

Alvisi가 제안한 FBML 메시지 로깅 프로토콜[1]은 회복 후 살아있는 프로세스들이 고아 프로세스가 되지 않도록, 회복과정 수행 시 살아있는 프로세스들의 수행을 대기(block)시킨다. 이러한 특성으로 인해 FBML 프로토콜은 프로세스 고장이 발생하는 경우, 애플리케이션의 수행시간을 매우 증가시킬 수 있다.

Manetho 프로토콜[8]은 동시적 고장이 발생한 경우, 살아있는 프로세스들이 시스템 일관성을 깨뜨릴 수 있다고 판단되는 모든 애플리케이션 메시지들을 전달하지 않고 버퍼에 저장시키고, 회복과정 중 안전한 저장소에 대한 동기적 로깅을 요구한다.

Elnozahy가 제안한 회복 알고리즘[6]은 앞에서 언급한 Manetho에 비해 조금 높은 통신비용을 요구하는 반면, 안전한 저장소에 대한 접근을 매우 줄이고 여러 개의 프로세스들이 고장 난다 하더라도 살아있는 프로세스의 현재 수행이 계속적으로 진행될 수 있도록 한다. 그러나, 인과적 메시지 로깅 기법이 비동기적 검사점 기록 기법[5]과 함께 사용되는 경우, 이 알고리즘은 특정한 경우들에 대해 전체 시스템의 일관성을 보장하지 못한다.

Mitchell과 Garg는 모든 프로세스가 회복 응답상태(Recovery Reply State)를 생성시키는 대기없는 분산형 회복 알고리즘을 제안하였다[10]. 여기서, 회복 응답상태란 모든 프로세스의 지역상태가 동일한 재현벡터(incarnation vector)를 가지기 위한 전역상태를 말한다. 이 알고리즘에서는 각 회복 프로세스가 회복 프로세스들 간의 어떠한 대표자 선출 과정 없이 자율적으로 자신의 회복과정을 수행한다. 그러나, 이 알고리즘은 회복과정이 종료되지 못할 수 있다[9].

3. 시스템 모델

본 논문에서 가정하는 비동기적 분산 시스템에서의 계산 N은 호스트에서 수행하는 n(n>0)개의 프로세스들로 구성된다. 프로세스들은 어떠한 전역 메모리와 전역 클락(global clock)을 가지고 있지 않다. 각 프로세스는 독립적인 속도로 수행하고, 메시지 전송지연시간이 유한하지만 임의적이다. 통신 네트워크는 분할되지 않고, 모든 프로세스가 항상 접근할 수 있는 안전한 저장소가 있다고 가정한다[8]. 또한, 프로세스는 고장이 발생하면 자신의 휘발성 저장소에 있는 자신의 상태를 잃어버리고, 수행을 멈추는 고장-멈춤 모델(fail-stop model)을 따른다고 가정한다[12]. 각 프로세스의 수행은 부분 결정적(piecewise deterministic)이다[7]. 즉, 이 모델에서는 한 프로세스 수행 중 어떠한 시점에서도 그 프로세스의 상태구간(state interval)[13]은 하나의 비결정적 이벤트(non-deterministic event)에 의해 결정된다. 본 논문에서의 비결정적 이벤트는 하나의 프로세스가 메시지를 수신하는 경우 그 메시지를 해당 애플리케이션에게 전달해주는 이벤트(delivery event)이다. 프로세스 p의 k번째 상태구간 si_p^k(k>0)는 p의 k번째 수신한 메시지 m의 전달 이벤트 dev_p^k(m)에 의해 시작된다. 따라서, p의 초기 상태(initial state) s_p^0와 그 프로세스의 수행 시 발생해왔던 모든 비결정적 이벤트들이 주어진다면, 그 프로세스의 상태는 유일하게 결정된다. 또한, 정상수행시 발생한 프로세스의 이벤트들은 Lamport의 전후 관계(happen-before relation)를 사용하여 순서화한다[7]. p의 상태 $s_p^i=\{si_p^0, si_p^1, ..., si_p^i\}$를 si_p^i까지의 모든 상태구간들의 순서집합이라 한다. p가 s_p^i까지 수행하는 동안 q로부터 수신하여 전달된 모든 메시지들에 대해 q가 s_q^j까지 수행하는 동안 그 메시지들을 송신했다는 정보가 반영되었고 q가 s_q^j까지 수행하는 동안 p로부터 수신하여 전달된 모든 메시지들에 대해 p가 s_p^i까지 수행하는 동안 그 메시지

들을 송신했다는 정보가 반영되었다면, s_p^i와 s_q^j(p≠q)는 상호 일관적(mutually consistent)이라고 말한다. 또한 시스템의 전역상태(global state)가 모든 프로세스들로부터 각각 하나의 상태로 구성된다고 할 때, 그 상태들 중 임의의 두 상태들이 상호 일관적이라면 그 시스템의 전역상태는 일관적이라고 말한다[5].

dev_p^k(m)의 결정자가 안전한 저장소에 저장되거나 메시지 m을 수신한 후에 p가 검사점을 취한다면, s_p^k는 안정적(stable)이다라고 한다[7]. 만약, p가 향후 고장이 발생하더라도 자신의 수행상태를 si_p^k까지 재연할 수 있다면, s_p^k는 회복가능(recoverable)하다라고 한다[7]. 한 메시지의 결정자(determinant)는 메시지 송신자(sid), 수신자(rid), 송신번호(ssn)와 수신번호(rsn)로 구성된다. 본 논문에서 메시지 m의 결정자와 p의 휘발성 저장소에 유지되는 결정자들의 집합을 각각 det(m)과 d-set_p로 나타낸다. sri(m)은 메시지 m의 수신자가 고장 난 경우 m을 수신자에게 재전송하기 위한 송신자의 저장소에 유지되는 m의 회복정보를 나타낸다. sri(m)은 m의 rid, ssn과 전달된 데이터(data)로 구성된다. s-log_p는 p의 휘발성 저장소에 유지되는 송신한 메시지의 회복정보 집합이다.

4. 제안하는 회복 알고리즘

4.1 Elnozahy 회복 알고리즘의 일관성 문제점

기존의 회복 알고리즘들 중 Elnozahy의 알고리즘(Elno)[6]은 살아있는 프로세스들이 회복 시 그들의 계산과정을 계속적으로 수행하도록 하고, 다른 알고리즘들 [1, 8]에 비해 안전한 저장소에 대한 접근 횟수를 줄이도록 한다. 그러나, 인과적 메시지 로깅이 비동기적 검사점 기록 기법과 결합되는 경우, Elno는 동시적으로 프로세스들이 고장 난다면 시스템의 상태가 일관적이지 않도록 만들 수 있다. 따라서, 본 절에서는 어떠한 경우 이러한 일관성 문제가 발생하는지에 대해 알아보고, 일관된 회복을 수행하는 회복 알고리즘을 제안하고자 한다. Elno는 다음과 같이 4 단계로 구성된다:

(단계 1) 모든 회복 프로세스는 자신의 가장 최근 검사점으로 복구하고, 재현번호(incarnation number)를 1만큼 증가시킨다. 여기서, 재현번호는 각 프로세스가 고장으로부터 회복할 때마다 1만큼 증가되는 정수이다. 그리고, 그 프로세스는 다른 프로세스들로부터 그들이 현재 살아있는지 회복 중인지에 관한 정보를 얻는다.

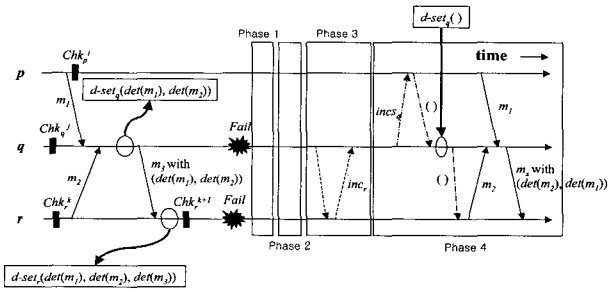
(단계 2) 회복 프로세스들은 자신들 중 하나의 회복 리더를 선출한다.

(단계 3) 선출된 리더는 모든 회복 프로세스로부터 각각의 현재 재현번호를 수집하고 그 번호를 incs라는 하나의 벡터에 저장한다.

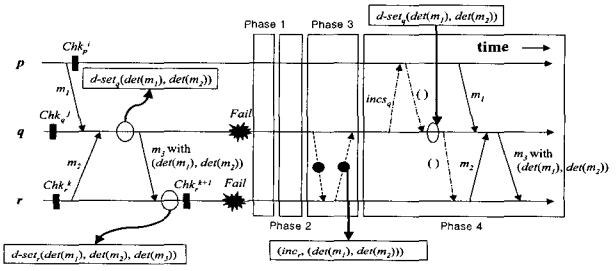
(단계 4) 리더는 모든 살아있는 프로세스에게 회복 프로세스들에게 필요한 결정자들을 요구하는 회복 메시지를 단계 3에서 수집한 incs를 포함하여 송신한다. 각 살아있는 프로세스가 리더로부터 요구 메시지를 수신할 때, 그 프로세스는 요구 메시지에 포함된 incs를 사용하여 자신의 incs를 갱신한다. 그 후, 그 프로세스는 회복 리더에게 자신이 가지고 있는 결정자를 제공한다. 만약, 각각의 살아있는 프로세스가 회복 프로세스의 고장 전에 송신된 애플리케이션 메시지를 수신한다면, 그 살아있는 프로세스는 자신의 incs를 사용함으로써 그 메시지가 일관성을 깨뜨릴 수 있다는 사실을 즉각적으로 알 수 있다. 이는 살아있는 프로세스가 회복 리더에게 결정자를 제공한 후, 시스템이 일관되지 않은 상태가 되도록 하는 어떠한 애플리케이션 메시지도 받아들이지 않는다는 것을 보장한다. 만약, 살아있는 프로세스가 회복 리더에게 결정자를 전달하기 전에 고장난다면, 그 리더는 자신의 incs를 갱신한 후 단계 4의 첫 부분부터 재시작한다. 이러한 과정은 회복 리더가 회복 시에 동시적 고장이 발생하더라도 일관된 결정자들의 집합을 얻을 수 있도록 보장한다. 회복 리더는 모든 살아있는 프로세스로부터 결정자들을 얻은 후, 그 결정자들을 다른 회복 프로세스들에게 제공함으로써, 모든 회복 프로세스가 일관된 상태로 회복한다. 만약, 회복 시 리더가 고장 난다면, 회복 프로세스들 중에 새로운 리더가 선출되어, 단계 2부터 단계 4까지 수행한다.

단계 3에서, 회복 리더는 단지 살아있는 프로세스들로부터만 회복 프로세스를 위한 결정자들을 수집한다. 따라서, 리더는 다른 회복 프로세스들이 가장 최근의 검사점을 취하기 전에 수신된 메시지에 피기백된 어떠한 결정자도 얻을 수 없다. 이러한 상황은 그 메시지 송신 프로세스의 회복된 상태가 수신 프로세스의 상태와 일관적이지 않도록 할 수 있다.

(그림 1)은 Elno의 일관성 문제를 예를 통해 설명해준다. 이 그림에서 프로세스 q는 자신의 지역 검사점 Chk_q^j을 취하고 p와 r로부터 메시지 m_1과 m_2 순서로 수신한 후, 메시지 m_3을 r에게 송신한다. r은 q로부터 메시지 m_3을 수신하



(그림 1) Elno의 일관적인 못한 수행 예



(그림 2) 제안한 회복 알고리즘의 일관적인 회복과정 수행 예

고 자신의 지역 검사점 $Chkr^{k+1}$을 취한다. 여기서, q와 r이 (그림 1)에서와 같이 동시에 고장났다고 가정하자. Elno에서 q와 r은 단계 1을 수행하고 단계 2에서 q가 회복 리더로써 선출된다. 단계 3에서 q는 r의 재현번호 inc_r을 얻는다. 단계 4에서 q는 살아있는 프로세스 p로부터 q와 r의 회복에 필요한 결정자를 요구한다. 그러나, p는 q와 r의 회복에 필요한 어떠한 결정자도 가지고 있지 않기 때문에, q는 p로부터 어떠한 결정자도 얻지 못한다. 이 경우, q는 m_1과 m_2의 rsn에 대한 정보를 가지고 있지 못하고 분산시스템에서 메시지 전송지연시간이 일정하지 않기 때문에, 단계 4에서 메시지 m_2과 m_1 순서로 재연할 수 있다. 그러므로 q는 단계 4에서 m_3가 아닌 m_x을 발생시킬 수 있다. 그러나, 프로세스 r은 q로부터 메시지 m_3을 수신한 후에 저장된 자신의 가장 최근의 검사점 $Chkr^{k+1}$로부터 재시작한다. 따라서, q와 r이 Elno를 각각 수행한 후에 q의 회복 상태는 r의 회복 상태와 일관적이지 못하다.

4.2 기본개념

Elno의 일관성 문제를 해결하기 위해 본 논문에서는 일관적이고 효율적인 회복 알고리즘을 제안한다. 제안하는 알고리즘은 Elno에 비해 부가적인 메시지 없이 회복 리더가 모든 회복 프로세스의 회복에 필요한 결정자를 살아있는 프로세스뿐만 아니라 다른 회복 프로세스들로부터도 수집하도록 한다. 이 알고리즘은 Elno와 같이 4 단계로 구성된다. 제안하는 알고리즘의 단계 1과 2는 Elno와 유사하다. 그러나, 단계 3에서 회복 리더는 모든 회복 프로세스의 재현번호와 그 회복 프로세스가 자신의 d-set에 유지하고 있는 결정자를 수집한다. 이렇게 수집된 각 재현번호들은 회복 리더의 incs에 저장되고 결정자들은 리더의 d-set에 저장된다. 이 알고리즘의 단계 4는 Elno과 유사하다.

본 논문에서 제안하는 회복 알고리즘에서 일관적인 회복을 수행하는 방법을 설명하기 위해 (그림 2)의 예를 보기로 하자. q와 r이 이 그림에서와 같이 동시적으로 고장난 후에, 그들은 각각 단계 1을 수행하고 단계 2에서 회복 리더를 선출한다. 이 경우 q가 회복 리더로써 선출된다. 따라서, 단계 3에서 q는 r로부터 r의 재현번호 inc_r과 $d\text{-}set_r$에 있는 결정자 ($det(m_1)$, $det(m_2)$)를 수집한다. 단계 4에서 q는 살아있는 프로세스 p로부터 q와 r을 위한 결정자를 요구한다. 이 경우, p는 $d\text{-}set_p$에 어떠한 결정자도 존재하지 않으므로 q에게 결정자를 제공할 수 없다. 그러나, q는 $d\text{-}set_q$에 메시지 m_1과 m_2의 결정자를 유지하고 있으므로 q는 단계 4에서 메시지 m_1과 m_2 순서로 재연할 수 있다. 따라서, q와 r이 이 알고리즘을 각각 수행한 후, q의 회복된 상태는 r의 회복된 상태와 일관적이다. 단계 3에서 회복 리더를 제외한 모든 회복 프로세스는 응답메시지에 자신의 재현번호와 결정자들을 포함시키고 그 메시지를 회복 리더에게 송신하기 때문에, 제안하는 알고리즘은 Elno에 비해 어떠한 부가적인 메시지도 요구하지 않는다.

본 논문에서 제안된 회복 알고리즘에서 프로세스 p를 위한 데이터 구조와 프로시저들은 (그림 3)과 (그림 4)에 주

[프로세스 p를 위한 데이터 구조]

-$incs_p$: 시스템 내 모든 프로세스의 재현번호를 저장하는 벡터. 벡터의 각 원소는 0로 초기화된다. p는 송신하는 메시지마다 p가 수신하는 각 메시지에 $incs_p[p]$를 피기백한다. 따라서, p가 q로부터 메시지를 수신할 때, p는 그 메시지에 피기백된 q의 재현번호가 $incs_p[q]$보다 작다면 그 메시지를 제거한다. 그렇지 않으면, p는 그 메시지를 받아들인다.

-$procstatus_p$: 시스템 내 모든 프로세스의 상태정보를 저장하는 벡터. 각 프로세스가 살아있다면 그 프로세스의 상태정보 값이 'L'이고, 회복중이면 'R'이다. 벡터의 각 원소는 'L'로 초기화된다.

[프로세스 p를 위한 프로시저]

[Procedure-1] 고장난 프로세스 p가 회복과정을 수행하는 프로시저

```
get its latest checkpoint, d-set_p and inc_p from the stable storage;
restore its state using the latest checkpoint;
inc_p ← inc_p + 1;
save inc_p into the stable storage;
incs_p[p] ← inc_p;
send Recovery_Msg() to the other processes;
leader_fail ← false;
do {
   (i, procstatus_p) ← ELECT_LEADER(p, 'R');
   if (p = i) then LEADER();
    else leader_fail ← RECOVERING(i);
} while(leader_fail);
replay the messages received beyond the latest checkpoint by using d-set_p;
```

(그림 3) 제안된 알고리즘에서 모든 프로세스 p를 위한 데이터 구조 및 회복 프로시저

[Procedure-2] 회복 리더 프로세스 p가 전체 회복과정을 조정하는 프로시저

```
dets_p ← ψ;
for all q ∈ N st ((p ≠ q) ∧ (procstatus_p[q] = 'R')) do {
    send Req_Inc_Dets() to q;
    receive Rep_Inc_Dets(inc_q, dets_q) from q;
    dets_p ← dets_p ∪ dets_q;
    incs_p[q] ← inc_q;
}
T: d-set_p ← ψ;
for all q ∈ N st (procstatus_p[q]) = 'L') do {
    send Req_Dets(incs_p) to q;
    receive Rep_Dets(status_q, inc_q, dets_q) from q;
    incs_p[q] ← inc_q;
    if (status_q = 'L') then d-set_p ← d-set_p ∪ dets_q;
    else dets_p ← dets_p ∪ dets_q;
    procstatus_p[q] = 'R';
    goto T;
}
d-set_p ← d-set_p ∪ dets_p;
for all q ∈ N st (procstatus_p[q] = 'R') do {
    temp_dets ← ψ;
    for all e ∈ d-set_p st (e.rid = q) do
        temp_dets ← temp_dets ∪ {e};
    send Deliver_Incs_Dets(incs_p, temp_dets) to q;
}
```

[Procedure-3] 회복 리더 이외의 회복 프로세스가 수행하는 프로시저

```
while(true) {
    receive m from q;
    if ((m is of the form Recovery_Msg()) ∧ q = leader) then return true;
    else if (m is of the form Req_Inc_Dets()) then
        send Rep_Inc_Dets(inc_p, d-set_p) to q;
    else if (m is of the form Req_Dets(incs_q)) then
        send Rep_Dets('R', inc_p, d-set_p) to q;
    else if (m is of the form Deliver_Incs_Dets(incs_q, dets_q)) then {
        d-set_p ← d-set_p ∪ dets_q;  incs_p ← incs_q;  return false;
    }
}
```

[Procedure-4] 살아있는 프로세스 p가 메시지 $Req_Dets(incs_p)$를 수신하는 경우 수행하는 프로시저

```
for all k ∈ N st (p ≠ k) do
    incs_p[k] ← max(incs_p[k], incs_q[k]);
send Rep_Dets('L', inc_p, d-set_p) to q;
```

(그림 4) 제안된 알고리즘에서 회복 시 수행되는 프로세스 p를 위한 프로시저

어진다. 프로시저 [*Procedure-1*]은 각 프로세스가 고장난 후 회복 시 수행되는 전체적인 프로시저이다. Elect_Leader()는 모든 회복 프로세스가 그들 중에 회복 리더를 선출하고 시스템에 있는 각 프로세스가 살아있는지 회복 중인지를 알도록 한다. 제안된 알고리즘에서 회복 리더는 프로시저 [*Procedure-2*]를 수행하고, 그 외 회복 프로세스는 [*Procedure-3*]을 수행한다. [*Procedure-2*]에서 리더 p는 모든 회복 프로세스 q의 재현번호와 결정자를 q에게 메시지 Req_Inc_Dets()를 송신하고 메시지 Rep_Inc_Dets(inc_q, $dets_q$)를 수신함으로써 얻는다. 그 후 p는 모든 살아있는 프로세스 r에게 메시지 Req_Dets($incs_p$)를 송신하고 메시지 Rep_Dets($status_r$, inc_r, $dets_r$)를 수신함으로써 결정자를 수집한다. r이 p로부터 회복 메시지 Req_Dets($incs_p$)를 수신할 때, r은 프로시저 [*Procedure-4*]를 수행함으로써 자신의 d-set을 p에게 제공한다. r이 p에게 응답하기 전에 고장난다면, p는 $dets_p$와 $procstatus_p[r]$를 갱신한 후 라벨 T로부터 재수행한다. p가 모든 살아있는 프로세스로부터 결정자들을 수집한 후, p는 다른 회복 프로세스 q에게 Deliver_Incs_Dets($incs_p$, temp_dets)를 송신함으로써 결정자를 q에게 제공한다. 그 후에, p와 q는 [*Procedure-3*]에서 그들의 결정자들을 이용함으로써 정상수행시 가장 최근의 검사점 이후에 수신된 메시지들을 재연하다. p가 회복 시 고장 난다면, q는 Elect_Leader()를 재수행하고 그 다음 과정으로 간다.

4.3 정당성 증명

본 절에서는 정리 1과 2를 통해 제안하는 회복 알고리즘의 안전성(safety)과 생존성(liveness)을 증명하고자 한다.

[보조정리 1] 인과적 메시지 로깅이 비동기적 검사점 기록 기법과 결합된다면, 회복 프로세스들간에 선출된 회복리더가 살아있는 프로세스들뿐만 아니라 다른 회복 프로세스로부터 결정자들을 수집해야만 $f(1 \le f \le n)$개의 동시적 고장시에도 시스템은 전역적으로 일관된 회복과정을 수행할 수 있다.

[증명] 모순법을 사용하여 보조정리 1을 증명하고자 한다. 회복리더가 살아있는 프로세스들뿐만 아니라 다른 회복 프로세스로부터 결정자들을 수집하더라도 일관된 회복을 수행할 수 없다고 가정하자. C를 시스템 N에서 고장난 모든 프로세스들의 집합이고, 메시지 m1은 메시지 m에 의존하고, $q(q \in C)$는 m의 수신 프로세스이고, $r(r \in N)$은 m_1의 수신 프로세스라 하자. 다음과 같은 두 가지 경우를 고려해야 한다.

(경우 1) det(m)이 m1에 피기백되었다.

이 경우 다음과 같은 두 경우를 고려해야 한다:

(경우 1.1) $r \in (N-C)$.

이 경우 다음과 같은 두 경우를 고려해야 한다:

(경우 1.1.1) r이 m1을 수신한 후, 회복 리더로부터 결정자들을 요구하기 위한 회복 메시지를 수신한다.

이 경우, det(m)이 $d\text{-}set_r$에 저장되어있기 때문에 r은 회복 리더에게 det(m)을 제공할 수 있다. 따라서, r은 결코 고아 프로세스가 되지 않는다.

(경우 1.1.2) r이 m_1을 수신하기 전, 회복 리더로부터 결정자들을 요구하기 위한 회복 메시지를 수신한다.

r이 회복 메시지를 수신할 때, r은 그 메시지에 포함된 재현벡터를 사용하여 자신의 재현벡터인 $incs_r$을 가장 최근 값들로 갱신한다. 그러나, 이러한 경우 r은 회복 리더에게 det(m)을 제공할 수 없다. 그 후 r이 m_1을 수신할 때, $incs_r$에서 m_1의 송신 프로세스의 재현번호가 m_1에 피기백되는 재현번호보다 크다면, q는 det(m)을 얻어서 m을 재연할 수 없기 때문에 r은 m_1을 제거한다. 따라서, r의 현재 상태는 q의 회복된 상태와 일관적이다. 그러므로, r은 결코 고아 프로세스가 되지 않는다.

(경우 1.2) $r \in C$.

이 경우 다음과 같은 두 경우를 고려해야 한다:

(경우 1.2.1) r이 고장나기 전, m_1을 수신하고 자신의 가장 최근 검사점을 취했다.

만약 r이 회복 리더로부터 결정자를 요구하기 위한 회복 메시지를 수신한다면, r의 가장 최근 검사점이 det(m)을 포함하기 때문에, r은 det(m)를 리더에게 전달할 수 있다. 따라서, r은 결코 고아 프로세스가 되지 않는다.

(경우 1.2.2) r이 고장나기 전, 자신의 가장 최근 검사점을 취하고, m_1을 수신하였다.

만약 r이 회복 리더로부터 결정자를 요구하기 위한 회복 메시지를 수신한다면, r은 det(m)를 리더에게 전혀 전달할 수 없다. 그러나, r의 가장 최근 검사점이 det(m)을 포함하지 않기 때문에, r은 결코 고아 프로세스가 되지 않는다.

(경우 2) det(m)이 m1에 피기백되지 않았다.

이 경우 r은 det(m)을 회복 리더에게 제공할 수 없다. 그러나, 인과적 메시지 로깅의 속성에 의해 f개의 동시적 고장이 발생하더라도 det(m)을 유지하고 있는 f+1개의 프로세스들 중 최소한 하나의 프로세스는 결코 고장나지 않는다. 따라서, 그 프로세스는 회복 리더에게 det(m)을 항상 제공할 수 있다. 그러므로, r은 결코 고아 프로세스가 되지 않는다.

따라서, 모든 경우에 일관적인 회복이 가능하다. 이는 앞에서의 가정에 위배된다.

[정리 1] 제안된 회복 알고리즘은 f개의 동시적 고장이 발생하더라도 시스템이 전역적 일관된 상태로 회복하도록 한다.

[증명] 보조정리 1에 의해 f개의 동시적 고장이 발생하더라도 전역적으로 일관된 회복과정을 수행하기 위해서는, 회복 프로세스들 중에 선출된 회복 리더는 살아있는 프로세스들뿐만 아니라 다른 회복 프로세스들로부터 결정자를 수집해야 한다. 제안된 회복 알고리즘은 그 규칙을 따른다. 따라서, 이 알고리즘은 f개의 동시적 고장이 발생하더라도 시스템이 전역적 일관된 상태로 회복하도록 한다.

[정리 2] 제안된 회복 알고리즘은 유한한 시간 내에 종료한다.

[증명] 회복 리더가 임의의 살아있는 프로세스로부터 필요한 결정자들을 수집하기 전에 그 살아있는 프로세스가 고장 나

는 경우, 회복 리더는 현재까지 수집한 결정자들을 제거하고 다시 다른 프로세스로부터 결정자들을 재 수집한다. 시스템 모델의 가정에 의해 최대한으로 f개의 동시적 고장이 발생할 수 있으므로, 회복 리더는 기껏해야 f번 제안된 회복 알고리즘을 재수행하면 된다. 부가적으로, 이 알고리즘은 어떠한 살아있는 프로세스도 대기시키지 않고, 리더를 제외한 회복 프로세스들은 리더가 회복에 필요한 모든 결정자들을 수집하는 과정이 종료될 때까지 대기한다. 따라서, 본 알고리즘은 유한한 시간 내에 종료한다.

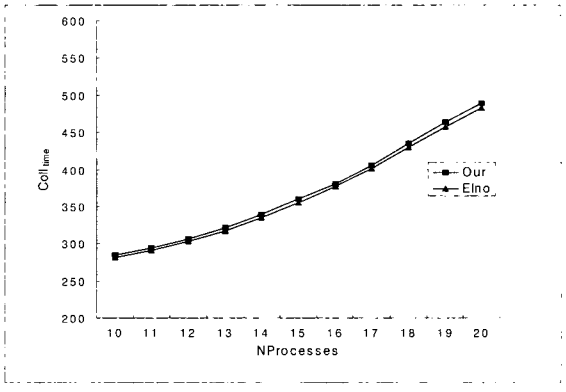
5. 성능평가

본 절에서는 메시지 전달 기반 시뮬레이션 언어인 PARSEC[2]을 사용하여 본 논문에서 제안한 회복 알고리즘(Our)과 기존의 Elnozahy가 제안한 회복 알고리즘(Elno)에 대한 성능을 비교하고자 한다. 시뮬레이션되는 시스템은 16개의 호스트들로 구성되고, 시뮬레이션의 편의를 위해 각 호스트에는 하나의 프로세스만 수행하고 모든 프로세스들이 동시에 생성되고 그 수행이 동시에 종료된다. 일반적으로 한 회복 프로세스의 회복 시간(R_{time})은 메시지 로깅 기법에 기반한 회복 알고리즘들의 성능 평가를 위해 사용된다. 인과적 메시지 로깅인 경우, 한 회복 프로세스의 회복 시간 R_{time}은 다음과 같이 세 가지 시간들 $Ckpt_{time}$, $Coll_{time}$, $Repl_{time}$로 구성된다. 먼저, $Ckpt_{time}$는 고장난 프로세스가 가장 최근의 검사점을 이용하여 고장 전 정상적인 상태로 복구하는데 걸리는 시간이다. 두 번째로, $Coll_{time}$는 다른 프로세스들로부터 자신의 회복을 위한 모든 결정자들을 얻는데 걸리는 시간이고, 마지막으로 $Repl_{time}$은 수집된 결정자들을 이용하여 회복 프로세스를 정상적인 상태로 복귀하는데 걸리는 시간이다. 평가하는 두 회복 알고리즘들은 모두 인과적 메시지 로깅에 기반하였기 때문에, 두 알고리즘의 $Ckpt_{time}$과 $Repl_{time}$는 각각 동일하다. 따라서, 본 시뮬레이션에서 비교하는 두 회복 알고리즘들의 성능 평가 기준은 $Coll_{time}$이기 때문에, 시뮬레이션에서 $Ckpt_{time}$=1000 time units, $Repl_{time}$=500 time units로 설정한다. 또한, 프로세스들 간의 메시지 전송 지연시간 L_{time}는 10 time units으로 설정한다. 마지막으로 각 프로세스에서의 검사점 기록 간격 CInterval은 평균 360000 time units으로 지수분포를 따른다.

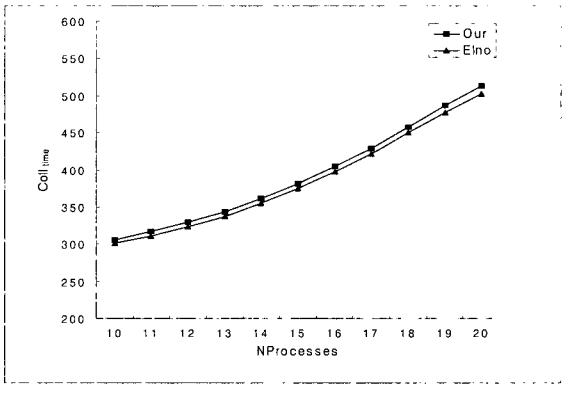
$Coll_{time}$에 영향을 미치는 주요 매개 변수들로는 NCurrFailures와 NProcesses이 있다. NCurrFailures는 인과적 메시지 로깅 기반 회복 알고리즘에서 프로세스들의 동시적 고장 발생 수이고, NProcesses는 전체 프로세스들의 수이다. 일반적으로, NCurrFailures가 증가함에 따라, 회복 프로세스들이 회복에 필요한 결정자들을 수집하는 과정이 복잡해지고 $Coll_{time}$이 증가하게 된다. 또한, NProcesses가 증가함에 따라, 각 회복 프로세스가 필요한 결정자들을 수집하기 위해 통신해야 하는 프로세스들의 수가 증가하여 $Coll_{time}$이 증가하게 된다. <표 1>은 앞에서 언급한 매개 변수들의 설정범위를 보여준다. 시뮬레이션에서 사용되는 분산 애플리케이션은 프로세스들이 서로 메시지를 주고받으면서 수행하는데, 각 프로세스는 수신한 메시지를 처리하고 임의의 다른 프로세스들에게 메시지를 전송한다. 두 회복 알고리즘들에 대한 공정한 평가를 위해 시뮬레이션되는 애플리케이션의 수행 중에 고장이 발생하는 위치가 동일하도록 설정한다.

〈표 1〉 시뮬레이션 매개 변수 및 설정범위

매개 변수	설정범위
NCurrFailures	2, 3
NProcesses	10-20



(그림 5) NProcesses vs. $Coll_{time}$ (NCurrFailures = 2)



(그림 6) NProcesses vs. $Coll_{time}$ (NCurrFailures = 3)

(그림 5)와 (그림 6)는 각각 동시에 고장나는 프로세스 수 즉, NCurrFailures가 2와 3일 때, 전체 프로세스 수 NProcesses의 변화에 따른 두 회복 알고리즘들의 $Coll_{time}$을 보여준다. 두 그림에서와 같이 두 알고리즘들은 모두 NProcesses가 증가함에 따라 $Coll_{time}$이 증가함을 알 수 있다. 이는 NProcesses가 증가함에 따라, 회복 프로세스들이 회복에 필요한 결정자들을 얻기 위해 통신하는 살아있는 프로세스들의 수가 증가하기 때문이다. 또한, (그림 5)에 비해 (그림 6)에서 두 알고리즘의 $Coll_{time}$이 보다 높음을 알 수 있다. 이는 NCurrFailures가 증가함에 따라, 회복 프로세스들 간에 회복 리더를 선출하는데 걸리는 시간이 길어지고, 회복 프로세스들에게 필요한 결정자량이 많아짐으로 회복 리더가 살아있는 프로세스들로부터 그 회복정보를 얻어오는

〈표 2〉 검사점 기록 기법에 따른 두 알고리즘의 일관적 회복 보장성

종류 \ 알고리즘	Our 알고리즘	Elno 알고리즘
동기적 검사점 기록	일관적 회복 보장	일관적 회복 보장
비동기적 검사점 기록	일관적 회복 보장	일관적 회복 불가능

데 걸리는 시간이 길어지기 때문이다.

한편, Our의 $Coll_{time}$이 Elno의 $Coll_{time}$에 비해 높지만, 그 차이는 매우 적다. 이러한 차이가 발생하는 이유로는 4절에서 (그림 1)와 (그림 2)를 통해 설명한 바와 같이 회복단계 3에서 Elno 알고리즘과 달리 Our 알고리즘은 회복 리더가 각 회복 프로세스 d-set에 유지하고 있는 결정자들을 메시지 피기백에 의해 부가적으로 얻기 때문이다. (그림 5)와 (그림 6)에서는 두 알고리즘의 $Coll_{time}$의 차이가 각각 1.0%~1.3%와 1.6%~2.1% 정도이다. 그러나, Elno 알고리즘에서 회복 리더가 살아있는 프로세스들로부터만 회복 프로세스를 위한 결정자들을 얻음으로써, 그 리더가 다른 회복 프로세스들이 가장 최근의 검사점을 취하기 전에 수신된 메시지에 피기백된 어떠한 결정자도 얻을 수 없다. 따라서, (그림 1)과 같은 경우가 발생한다면, 고장난 프로세스들의 회복 후 상태가 서로 일관적이지 못할 수 있다. 그러므로, 인과적 메시지 로깅 기법이 비동기적 검사점 기법과 결합하는 경우 시스템 일관성을 보장하는 프로세스 회복을 수행하기 위해서는, Our 회복 알고리즘에서의 이러한 부가적인 과정이 필수적이다. 또한, 이 시뮬레이션을 통해 Elno 알고리즘에 비해 제안한 Our 알고리즘의 부가적인 비용이 매우 적음을 알 수 있다. 마지막으로 <표 2>는 이러한 두 회복 알고리즘들이 결합되는 검사점 기록 기법의 종류에 따른 일관적 회복 보장 제공여부를 보여준다.

6. 결 론

본 논문에서는 인과적 메시지 로깅이 비동기적 검사점 기록 기법과 결합되는 경우, Elnozahy 회복 알고리즘은 동시적 고장 발생시 시스템의 상태를 일관적이지 못한 상태가 되도록 할 수 있다는 것을 제시하였다. 또한, 이러한 문제점을 해결하기 위해 본 논문에서는 회복 리더가 살아있는 프로세스뿐만 아니라 다른 회복 프로세스들에게도 결정자들을 얻도록 하는 일관적인 회복 알고리즘을 제안한다. 이 회복 알고리즘은 Elnozahy 회복 알고리즘에 비해 어떠한 부가적인 메시지도 요구하지 않으며, 본 논문에서 수행한 시뮬레이션에서 제안된 알고리즘의 피기백에 의한 부가적인 비용이 매우 적음을 보여준다.

참 고 문 헌

[1] L. Alvisi, B. Hoppe and K. Marzullo, "Nonblocking and Orphan-Free Message Logging Protocols," In Proc. of the 23th Symposium on Fault-Tolerant Computing, pp.145-154, 1993.

[2] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin and H. Y. Song, "Parsec: A Parallel Simulation Environ- ments for Complex Systems," IEEE Computer, pp.77-85, 1998.

[3] K. Bhatia, K. Marzullo and L. Alvisi, "Scalable Causal Message Logging for Wide-Area Environments," Con- currency and Computation: Practice and Experience, Vol.15, No.3, pp.873-889, August, 2003.

[4] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier and F. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," In Proc. of the 15th International Conference on High Performance Networking and Computing(SC2003), November, 2003.

[5] 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, 1985.

[6] E. N. Elnozahy, "On the Relevance of Communication Costs of Rollback Recovery Protocols," In Proc. the 15th ACM Symposium on Principles of Distributed Computing, pp. 74-79, 1995,

[7] E. N. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, Vol.34, No.3, pp.375-408, 2002.

[8] 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, pp.526-531, 1992.

[9] B. Lee, T. Park, H. Y. Yeom and Y. Cho, "On the impossibility of non-blocking consistent causal recovery," IEICE Transcations on Information Systems, Vol. E83-D, No.2, pp.291-294, 2000.

[10] J. R. Mitchell and V. Garg, "A non-blocking recovery algorithm for causal message logging," In Proc. of the 17th Symposium on Reliable Ditributed Systems, pp.3-9, 1998.

[11] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," In Proc. of the 9th International Symposium on Operating System Principles, pp.100-109, 1983.

[12] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant distributed computing systems," ACM Transactions on Computer Systems, Vol.1, pp.222-238, 1985.

[13] R. B. Strom and S. Yemeni, "Optimistic recovery in distributed systems," ACM Transactions on Computer Systems, Vol.3, pp.204-226, 1985.

[14] S. Venkatesan, "Optimistic crash recovery without changing application messages," IEEE Transactions on Parallel and Distributed Systems, Vol.8, pp.263-271, 1997.

[15] B. Yao, K. -F. Ssu and W. K. Fuchs, "Message Logging in Mobile Computing," In Proc. of the 29th International Symposium on Fault-Tolerant Computing, pp.14-19, 1999.

안 진 호

e-mail : jhahn@kyonggi.ac.kr
1997년 고려대학교 컴퓨터학과(이학학사)
1999년 고려대학교 컴퓨터학과(이학석사)
2003년 고려대학교 컴퓨터학과(이학박사)
2003년~현재 경기대학교 정보과학부 전자계산학과 조교수
관심분야 : 결함포용 분산시스템, 이동에이전트 시스템, 그룹통신, p2p 컴퓨팅

방 승 준

e-mail : ppangas@hotmail.com
2006년 경기대학교 전자계산학과(이학학사)
현 재 경기대학교 전자계산학과 석사과정
관심분야 : 결함포용 분산시스템, 그룹통신, p2p 컴퓨팅