인과적 메시지 로깅에서 확장성 지원 방법
(How To Support Scalability in Causal Message Logging)

김 기 범 † 황 종 선 †† 유 헌 창 ††† 손 진 곤 †††† 정 순 영 †††††
(Kibom Kim) (Chung-Sun Hwang) (HeonChang Yu) (JinGon Shon) (SoonYoung Jung)

요 약 인과적 메시지 로깅기법은 프로세스의 파손 결함을 포용하는 분산 시스템을 작은 비용으로 구축할 수 있는 기법이다. 기존의 인과적 메시지 로깅기법은 결함 포용 시스템 내에 고정된 수의 프로세스가 존재한다는 기본가정을 갖고 있다. 이러한 가정은 새로운 프로세스의 추가 혹은 현존하는 프로세스의 소멸 시 모든 프로세스들이 자신의 자료구조를 변경해야 한다. 그러나, 우리는 각각의 프로세스가 모든 프로세스에 대한 목록을 유지하는 것이 아니라 통신을 수행하는 프로세스에 대한 목록을 유지하도록 한다. 이러한 방법은 결함 포용시스템을 여러 다른 스케일에서도 동작하도록 하여 준다. 이러한 기법을 이용하여, 우리는 현존하는 인과적 메시지 로깅 기법에 적용할 수 있는 새로운 회복 알고리즘을 개발하였다. 제안하는 알고리즘은 1) 회복리더를 필요로 하지 않는 분산화된 기법이고, 2) 회복이 진행되는 동안 정상프로세스의 실행을 막지 않는 비방해기법이고, 3) 여러 프로세스의 동시 결함도 포용할 수 있는 기법이다. 기존의 인과적 메시지 로깅기법에서는 위의 성질 중 하나 이상을 만족시키지 못했다.

Abstract The causal message logging is a low-cost technique of building a distributed system that can tolerate process crash failures. Previous research in causal message logging protocol assumes that the number of processes in a fault-tolerant system is fixed. This assumption makes all processes modify their data structures when a new process is added or an existing process terminates. However, the proposed approach in this paper allows to each process retain identifiers of only the communicating processes instead of all processes. This mechanism enables the fault-tolerant system to operate at many different scales. Using this mechanism, we develop a new algorithm that can be adapted for recovery in existing causal message logging protocols. Our recovery algorithm is 1) a distributed technique which does not require recovery leader, 2) a nonblocking protocol which does not force live processes to block while recovery is in progress, and 3) a novel mechanism which can tolerate failures of an arbitrary number of processes. Earlier causal message logging protocols lack one or more of the above properties.

1. 서 론

분산시스템에서 신뢰성 향상을 위하여 결함 포용(fault tolerance)에 대한 많은 연구가 진행되고 있다. 결함을 극복하기 위하여, 각각의 프로세스는 정상 실행 시 주기적으로 자신의 상태를 안전한 저장 장소에 저장한다. 만약 한 프로세스가 결함이 발생하면, 새로운 프로세스가 생성되고, 타당한 저장된 상태가 주어지고, 계산을 재개하게 된다. 이때, 저장된 프로세스의 상태를 검사점(checkpoint)이라 한다. 이러한 결함 포용 방법을 복귀 회복(rollback recovery)이라 하고, 자원을 중복시키는 기법과 더불어 널리 이용되어 지고 있는 기법이다.

검사점을 이용하는 방법에는 크게 두 가지가 있다. 첫째, 동기적 검사점 기법은 프로세스가 검사점을 취할 때 다른 프로세스의 상태와 일관성(consistency)이 존재

· 본 연구는 한국과학재단 핵심전문연구(과제번호 : 981-0924-126-2)인 "분산시스템에서 비동기적 회복기법의 개발"의 지원으로 수행되었음.
† 학생회원 : 고려대학교 컴퓨터학과
kibom@disys.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과 교수
hwang@disys.korea.ac.kr
††† 정 회 원 : 고려대학교 컴퓨터교육과 교수
yuhc@comedu.korea.ac.kr
†††† 종신회원 : 한국방송통신대학교 컴퓨터과학과 교수
jgshon@mail.knou.ac.kr
††††† 비 회 원 : (주) 에코 연구개발실장
jsy@eco.co.kr
논문접수 : 1999년 2월 25일
심사완료 : 2000년 3월 9일

하도록 하는 방법으로서 검사점을 취할 때마다 동기화를 위한 대기가 발생함으로 인하여 성능 저하를 초래한다[1,2,3]. 둘째, 비동기적 검사점기법은 프로세스들의 검사점을 비동기적으로 취하는 방법으로서, 각각의 프로세스가 취한 검사점들 사이의 전역 일관성이 존재하지 않을 가능성이 높다. 이로 인하여 복귀의 도미노 현상을 야기한다. 그래서 비동기적 검사점 기법은 독립적으로 사용되지 않으며 메시지 로깅(message logging) 기법과 함께 사용된다[4,5]. 메시지 로깅 기법은 정상 수행 시 추가적으로 프로세스간 통신에서 발생하는 메시지를 저장하는 방법으로서, 프로세스의 결함 발생으로 인하여 잃어버리는 정보의 양은 검사점만 사용하는 기법에 비하여 적다. 메시지 로깅 기법은 부분 결정적 수행 모델(piecewise deterministic execution model; PWD 모델)의 가정에 근간을 두고 있다. PWD 모델에서, 각각의 프로세스의 수행은 일련의 결정적 상태 구간으로 이루어져 있고 각각의 구간들은 비결정적 사건의 발생에 의하여 시작된다[6,7,8].

메시지 로깅 프로토콜은 메시지에 대한 정보를 언제, 어디에 저장하는가에 따라서 크게 3가지 방법으로 나뉜다[6,9]. 첫째 비관적(pessimistic) 메시지 로깅 기법에서는 프로세스가 메시지를 보내기 전에 받은 메시지에 대한 정보를 안전한 저장 장치에 저장한다[7,10]. 이로 인하여 전역 일관성을 위배하는 일은 없지만 동기적 로깅으로 인하여 프로세스의 성능을 저하시킨다. 둘째, 낙관적(optimistic) 메시지 로깅 기법에서는 메시지에 대한 정보를 프로세스의 수행과 비동기적으로 저장한다[11,12,13]. 이 기법은 정상 수행 시 성능 향상의 대가로, 결함 발생 후 전역 일관성을 충족시켜주기 위하여 정상인 프로세스가 복귀를 수행하여야 한다. 셋째, 인과적(causal) 메시지 로깅은 비관적 메시지 로깅과 낙관적 메시지 로깅의 장점만을 결합시킨 방법으로서, 메시지에 대한 정보를 여러 프로세스에 중복시킴으로 인하여, 안전한 저장 장치에 동기적으로 메시지에 대한 정보를 저장할 필요가 없고, 프로세스의 결함으로 인하여 정상 프로세스의 복귀를 강요하지 않는다[4,9,14]. 그러나, 지금까지 제안된 인과적 메시지 로깅 기법은 결함 발생시 정상인 프로세스의 수행을 멈추게 하거나[4,9], 회복 리더(recovery leader)를 필요로 하는 중앙 집중형 회복 방법이 제시되었다[14].

기존 메시지 로깅 기법들은 분산시스템에 고정된 수의 프로세스가 존재한다는 가정 하에 개발되었다[4,6,8]. 이러한 가정은 분산 시스템의 중요한 성질인 확장성(scalability)을 지원해 줄 수 없다. 또한, 각각의 프로세스는 모든 프로세스와 통신을 수행한다라는 가정을 가지고 있다. 이러한 두 가정 사항은 크게 세 가지 문제점을 안고 있다. 첫째, 프로세스가 새로이 추가되어 질 때, 시스템 내의 모든 프로세스들은 안전한 저장장치에 유지하고 있는 다른 프로세스에 대한 식별자를 갱신해 주어야 한다. 둘째, 임의의 프로세스가 종료하고자 할 때, 시스템 내의 다른 모든 프로세스들은 종료하고자 하는 프로세스의 식별자를 제거하여야 한다. 셋째, 비록 하나의 프로세스가 결함이 발생하였다 하더라도, 모든 프로세스에게 도움을 요청하는 메시지를 보내야 한다.

이 논문에서는 기존 메시지 로깅기법에서의 두 가지 가정 사항을 제거하고, 세 가지 문제점을 해결하고자 한다. 즉, 각각의 프로세스는 자신과 통신하는 프로세스에 대한 목록을 개별적으로 유지한다. 이러한 방법은 새로운 프로세스가 추가되어 질 때 다른 프로세스에게는 전혀 영향을 미치지 않는다. 단지 최초로 통신을 수행하게 될 때, 자신의 목록과 통신 대상 프로세스의 목록만 갱신시켜 주면 된다. 또한 프로세스가 종료하고자 할 때, 자신과 통신을 수행했던 프로세스에게만 목록 갱신을 요청하면 된다. 그리고, 결함 발생 시 오직 자신과 통신을 수행했던 프로세스에게만 도움을 요청하는 메시지를 보내게 된다. 이러한 메카니즘을 기존 메시지 로깅 기법 중 인과적 메시지 로깅 기법에 적용하여, 인과적 메시지 로깅의 현존하는 단점을 해결하고자 한다. 즉, 결함 발생 시 정상인 프로세스의 수행을 멈추게 하지 않고, 회복을 위한 리더를 필요로 하지 않는 분산 회복 기법을 제안한다. 뿐만 아니라, 제안하는 회복 알고리즘은 여러 프로세스의 동시 다발적인 결함도 포용하도록 한다.

본 논문에서는 우선 2장에서 메시지 로깅 프로토콜에서 가정하는 기본적인 시스템 모델에 대하여 기술하고, 3장에서는 확장성을 지원하기 위한 방법을 살펴보고, 4장에서는 확장성을 지원하는 새로운 메시지 로깅 프로토콜을 제안한다. 5장에서는 제안하는 메시지 로깅 프로토콜의 정당성을 증명하고, 6장에서는 제안된 메시지 로깅 프로토콜의 최적화 방안에 대하여 살펴보고, 7장에서는 관련된 연구와 비교하고, 끝으로 8장에서 결론을 맺는다.

2. 시스템 모델

이 논문에서는 여러 지역에 퍼져 있는 다수의 프로세스들을 갖는 분산시스템을 기본 대상으로 한다. 프로세스의 수는 고정되어 있지 않고, 프로세스들은 메시지 교환을 통해서만 통신을 수행하고, 공유하고 있는 메모리

는 없다. 시스템은 비동기적이다. 즉, 프로세스들간의 상대적 속도에 대한 제한이 없고, 메시지 전송 지연에 대한 한계가 없고, 전역 시간 자원(global time source)을 갖고 있지 않다. 통신 채널은 선입선출(First-In-First-Out; FIFO) 방식으로서, 프로세스 P_1이 메시지 m_1과 m_2를 차례로 P_2에게 보냈다면, P_2는 m_1을 받기 전에 m_2를 받지 않는다는 것이다. 프로세스들은 실패-정지(fail-stop) 결함 모델에 따라서 독립적으로 결함이 발생할 수 있다. 실패-정지 결함 모델에서 프로세스는 결함이 발생하면 수행을 멈추게 되고, 결함 발생 사실을 궁극적으로 모든 살아 있는 프로세스들이 탐지해 낼 수 있다. 프로세스들의 수행은 PWD 모델의 가정에 따른다. 각각의 프로세스들은 독립적으로 자신의 상태를 안전한 저장장치에 저장한다. 즉, 비동기적으로 검사점을 취하게 된다.

어플리케이션 프로세스들은 송신(send) 사건과 수신(receive 혹은 deliver)[1] 사건, 지역(local) 사건을 포함한 사건들을 수행한다. 이 논문에서는 하나의 송신 사건은 오직 하나의 목적지를 가진다고 가정한다. 수신사건은 받은 메시지 m에 대하여 하나의 수신 순서 번호(receive sequence number; RSN)를 할당하게 되는데, 이 번호는 m을 받은 순서를 표시하는 것이다. 여기서, 우리는 메시지 m에 대한 RSN을 $m.rsn$을 표기한다. 따라서, 만약 프로세스 P_i가 m을 수신하였고 $m.rsn=j$ 라면, 메시지 m은 P_i가 j 번째 수신한 메시지이다. 메시지를 수신한 프로세스를 $m.dest$로 표기하고, $m.source$를 해당 메시지를 송신한 프로세스, $m.ssn$은 $m.source$가 메시지를 송신하면서 유일하게 할당한 식별자 번호이다. 튜플 <$m.source$, $m.ssn$, $m.dest$, $m.rsn$, $m.data$>는 메시지 m을 다른 메시지와 구분 되도록 한다. 이 튜플을 인과적 메시지 로깅에서 "결정자(determinant)"라 부른다. 결함을 극복하는 데 있어서 튜플의 원소 중 $m.data$는 생략되어 질 수 있는 데, 이는 $m.source$에서 다시 생성가능하기 때문이다.

두 개의 프로세스 P_i와 P_j에 대한 상태를 S_i와 S_j라 할 때($P_i \neq P_j$), P_i가 S_i까지 P_j로부터 수신한 모든 메시지는 P_j가 S_j까지 P_i에게 송신한 메시지이고, P_j가 S_j까지 P_i로부터 수신한 모든 메시지는 P_i가 S_i까지 P_j에게 송신한 메시지일 때, 상호 일관성(mutually consistent)을 만족한다고 한다[6,9]. 하나의 프로세스로부터 하나의 상태씩을 추출한 전역 상태에 대하여, 각각의 상태들의 상호 일관성을 만족하면, 전역 일관성(global consistency)을 만족한다.

3. 확장성 문제

분산시스템에서 제공되어야 할 중요한 특성으로 자원공유(resource sharing)와 개방성(openness), 동시성(concurrency), 결함 포용, 확장성(scalability), 투명성(transparency)이 있다[15,16]. 이들 특성들에 대하여 개별적으로 활발한 연구가 진행되어 왔다. 이 논문에서는 결함 포용을 제공하면서 동시에 확장성도 지원해 줄 수 있는 방법에 초점을 맞추고 있다. 확장성을 지원하는 시스템은 기본적으로 프로세스의 추가 혹은 소멸에 대하여 기존의 알고리즘이나 시스템에 변경을 가하지 않고도 수정할 수 있는 시스템이다.[2] 확장성을 지원하기 위한 알고리즘은 다음과 같은 4가지 성질을 만족하여야 한다[16].

- S1 어떠한 프로세스도 시스템 상태에 관한 완전한 정보를 갖고 있지 않다.
- S2 각각의 프로세스는 오직 지역 정보를 바탕으로 결정(decision)을 행한다.
- S3 한 프로세스의 결함이 알고리즘을 무효화(ruin)하지 않는다.
- S4 전역 시계가 존재한다는 가정사항이 없다.

비동기적 검사점 기법에 기반을 둔 메시지 로깅에서는 일반적으로 전역 시계에 대한 가정을 하지 않음으로 S4를 지원한다. 또한 결함 포용에 초점을 둔 알고리즘이기 때문에 S3는 기본적으로 지원한다. 그러나, 기존의 메시지 로깅 프로토콜에서는 S1 및 S2 성질을 타당하게 지원하고 있지 않다. 지금까지 조사한 바에 의하면, 기존의 모든 메시지 로깅 기법들은 모든 프로세스에 대한 식별자를 알고 있다는 가정 하에서 시스템이 개발되었다. 즉, N개의 프로세스가 존재한다는 시스템 모델에 근간을 두고 있다. 이로 인하여, 프로세스(프로세서)의 수행을 시작하기 전에 결함 포용을 지원하는 계층(혹은 운영체제)에서는 시스템내의 관련된 모든 프로세스에 대한 정보를 수집하여야 한다. 또한, 시스템의 수행 도중

1) receive 사건은 결함 포용 시스템에서 메시지를 받는 사건으로, deliver 사건은 결함 포용 시스템에서 어플리케이션 프로세스로 메시지를 전달하는 사건으로 구분하는 논문도 있으나, 본 논문에서는 결함 포용 시스템에서 받은 어플리케이션 메시지는 바로 어플리케이션으로 전달한다고 가정한다.

2) 본 논문에서는, 병렬처리시스템에서의 프로세서의 수의 증가에 따른 성능향상은 논외로 하고, 분산시스템에서의 프로세스의 추가·삭제를 허용하는 확장성에 초점을 맞추고 있다.

에 N의 변화에 대해서 현존하는 시스템에서는 모든 프로세스들의 자신의 자료구조를 갱신해 주어야만 하는 문제점을 가지고 있다.

이러한 문제점을 해결하기 위한 방법으로, 프로세스에 대한 목록을 중앙집중형으로 유지하여, 프로세스의 생성, 소멸 시 목록을 관리하는 서버에 기반한 방법을 고려할 수 있다. 이 방법은 서버에게 너무 높은 신뢰성을 요구하고 목록 유지의 일관성을 위한 추가적인 오버헤드를 유발시킨다. 즉, 각각의 어플리케이션 프로세스에 결함포용을 제공하는 계층에서는 서로 같은 프로세스에 대한 목록을 유지해야만 한다. 따라서, 결함 포용 계층에서 서로 다른 프로세스 목록을 유지하는 보다 효율적인 방법이 필요하다. 이를 위해서는 기존 연구에서 가정한 모든 프로세스들이 같은 프로세스 목록을 유지한다는 가정을 제거해야만 한다.

이 논문에서는 결함 포용 시스템에서 확장성을 가장 자연스럽게 지원해 주기 위하여, "자신과 통신을 수행하는 프로세스에 대한 목록"을 유지하도록 함으로써 해결해 주고자 한다. 이러한 프로세스 목록 유지 방법은 결함 포용시스템의 중요한 성질인 어플리케이션에 대한 투명성과 확장성을 만족한다. 왜냐하면, 분산 어플리케이션 프로세스의 수행 초기에 결함 포용 시스템은 어플리케이션 프로세스가 통신을 수행해야 하는 모든 프로세스에 대한 목록을 전혀 모르고 있어도 무관하기 때문이다. 즉, 어플리케이션 프로세스에서 통신을 수행하고자 할 때 결함 포용시스템에서 해당 어플리케이션 프로세스에 대한 목록을 유지하도록 하면 된다. 또한 새로운 프로세스가 추가될 때 현존하는 결함 포용시스템에 영향을 미치지 않게 된다. 즉, 현존하는 프로세스들은 추가된 프로세스와 통신을 수행하는 시점에서 자신이 유지하는 프로세스의 목록만 갱신하면 된다. 그리고, 프로세스가 소멸하고자 할 때, 자신과 통신을 수행했던 프로세스에게(즉, 자신의 유지하고 있는 프로세스에 대한 목록) 목록을 변경하라고 메시지를 보내주면 된다.

제안하는 확장성 지원 방법은 S1과 S2 성질을 지원한다. 왜냐하면, 분산 어플리케이션의 수행 초기에서부터 수행을 종료하는 시간까지 시스템내의 어떠한 프로세스도 시스템내의 모든 프로세스를 파악하지 않아도 되기 때문이다. 또한, 프로세스의 삽입 및 제거에 대한 결정은 개별 프로세스가 독자적으로 결정하고 작업을 수행할 수 있기 때문이다.

4. 확장가능한 인과적 메시지 로깅 프로토콜

이 장에서는 확장성을 지원하면서 기존 인과적 메시지 로깅기법이 갖고 있는 문제점을 해결하기 위한 새로운 메시지 로깅 프로토콜을 제안한다. 메시지 로깅 프로토콜은 일반적으로 정상 수행시의 회복을 위한 정보 축적 과정과 결함 발생 후 회복을 위한 과정으로 나뉜다.

4.1 정상 수행

제안하는 프로토콜은 정상 수행 시 기존 인과적 메시지 로깅을 확장한 형태로서, 메시지에 대한 정보를 중복시키는 방법은 기존 기법을 그대로 이용한다. 즉, 각각의 프로세스는 메시지를 보내면서 자신이 유지하고 있는 결정자에 대한 정보를 피기백(piggyback)해서 보낸다. 결정자에 대한 정보는 다섯 개의 원소를 갖는 튜플 자체이거나, 그래프 형태를 이용한다. 결정자를 수신한 프로세스는 이를 휘발성 메모리에 저장한다. 결정자들은 메시지의 흐름에 따라서 계속 인과적으로 피기백되는 것이 아니라 각각의 기법에 따라 피기백되는 결정자들의 수를 줄일 수 있다.

인과적 메시지 로깅에 확장성을 지원하기 위하여 새로운 자료구조, 즉, *CSet*이 필요하다. 기존 기법에서 시스템 내의 각각의 프로세스들은 다른 모든 프로세스들에 대한 식별자를 가지고 있다고 가정하였다. 이 가정을 대신하여, 제안하는 기법에서는 각각의 프로세스들은 개별적으로 다른 프로세스에 대한 목록인 *CSet*을 유지한다. 프로세스 P_i가 유지하는 $CSet_i$는 다음과 같다.

[CSet$_i$] $CSet_i$는 집합으로서, 초기 값은 공집합이고, 원소는 P_i를 포함하지 않는 프로세스에 대한 식별자로서, 안전한 저장장치에 저장된다.

각각의 프로세스가 모든 프로세스 식별자를 유지하는 기존 인과적 메시지 로깅기법에서는 메시지 송·수신시 해당 프로세스에 대한 식별자는 반드시 갖고 있다는 기본 가정을 가지고 있다. 그러나, 제안하는 기법에서 임의의 프로세스는 메시지를 송·수신할 때 해당 프로세스가 *CSet*의 원소 여부를 먼저 검사한다. 즉, *CSet*을 기존 인과적 메시지 로깅에 적용하기 위해서는 메시지

```
When a process P_i send(receive) a message to(from) P_j
    if ( P_j ∉ CSet_i) {
        CSet_i = CSet_i ∪ {P_j};
        logging CSet_i to stable storage;
    }
```

그림 1 인과적 메시지 로깅에 *CSet*을 적용한 알고리즘

의 송·수신을 처리하는 알고리즘에 그림 1과 같은 전처리 알고리즘이 추가된다.

임의의 프로세스가 종료하고자 한다면, 모든 프로세스에게 종료를 알리는 것이 아니라, 오직 자신과 통신을 수행했던 프로세스들 즉, *CSet*에 속한 프로세스들에게 Termination 메시지를 보낸다. 또한 Termination 메시지를 받은 프로세스는 *CSet* 목록에서 종료 프로세스의 식별자를 제거한다. 이때 종료한 프로세스는 다른 프로세스의 결함에 대한 정보를 제공해 줄 수 없고, 종료직전까지 수행한 내용은 보존되어야 한다. 이 문제는 출력완료(output commit)문제와 같다. 이를 해결하기 위하여, 종료하는 프로세스는 통신했던 프로세스로부터 받은 결정자 정보를 되돌려 주어야 한다. 종료하는 프로세스로부터 결정자 정보를 받은 프로세스는 이들 결정자 정보를 안전한 저장장치에 저장한다. 프로세스의 종료와 관련된 알고리즘은 그림 2와 같다.

When a process P_i want to terminate,
  for each process(P_j) in $CSet_i$
    Send *Termination*(P_i, *determinats which received from* P_j) message to P_j;
  Wait until receive all *OkTerm* messages;
  Terminate the execution;

When a process P_j receive *Termination*(P_i, *Ds*) message,
  Save the determinants(*Ds*) on stable storage;
  Send *OkTerm* message to P_i;

그림 2 프로세스의 종료 지원 알고리즘

정상 수행 시 프로토콜의 동작에 대하여 그림 3을 통하여 살펴보자. 우선 프로세스 P_2 관점에서 살펴보면, 프로세스 P_1으로부터 메시지 m_3을 받을 때, P_1을 $CSet_2$에 추가하고, 프로세스 P_3에게 메시지 m_4를 보내면서, P_3를 $CSet_2$에 추가하고, 이후에 메시지에 대해서 (m_8,m_{11},m_{12},m_{14},m_{15})는 $CSet_2$의 변경이 없다. $CSet_2$에 대한 변경은 안전한 저장장소에 접근하는 것을 말한다. 프로세스 P_5는 프로세스 P_4로부터 메시지 m_2를 받으면서 $CSet_5$에 P_4를 추가하고, m_5를 보낼 때, P_4가 이미 $CSet_5$의 원소이므로 추가하지 않는다. 그리고, P_5가 종료를 하고자 하는 경우 자신의 $CSet_5$ 원소인 P_4에게 종료를 알리는 *Termination* 메시지와 P_4로부터 인과적 메시지 로깅에 의하여 받은 메시지 m_1과 m_2에 대한 결정자 정보를 전달한다. 이를 전달받은 P_4는 결정자 정보를 안전한 저장장치에 저장하고 *OkTerm* 메시지를 P_5에게 보낸다. *OkTerm* 메시지를 받은 P_5는 수행을 종료하게 된다. 프로세스 P_6는 다른 프로세스들이 수행을 하고 있는 중간에 생성된 프로세스임에도 불구하고, 다른 프로세스에게 영향을 미치지 않고 오직 해당 프로세스가 $CSet_6$에 원소인가만 검사해 주면 되기 때문에 P_2에 대한 설명과 같다.

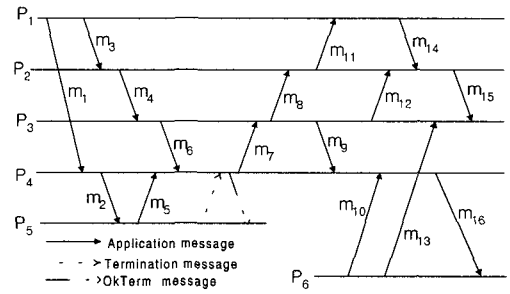

그림 3 정상 수행시 제안하는 프로토콜의 수행예제

4.2 회복 수행

이 절에서는 서론에서 언급한 기존 인과적 메시지 로깅이 갖고 있는 회복 시 비효율적인 문제점을 해결하기 위한 회복 알고리즘을 제안한다. 프로세스가 결함이 발생하면, 정상 수행시 저장해 둔 검사점 정보가 주어진 새로운 프로세스가 생성된다[3]. 회복 프로세스에게 주어진 검사점은 다른 프로세스와 비동기적으로 저장했기 때문에 전역 일관성을 위배할 수 있다. 따라서, 회복 프로세스는 결함으로 인해 잃어버린 메시지 정보(즉, 결정자들에 대한 정보)를 수집하여야 한다. 이를 위해서 기존 방법에서는 모든 프로세스에게 결정자를 요구하는 메시지를 보냈지만, 제안하는 방법에서는 오직 자신과 통신을 수행했던 프로세스들에게 결정자를 요구한다. 다른 프로세스로부터 받은 결정자를 바탕으로 재수행을 하면 일관된 전역 상태로 회복하게 된다.

여러 개의 프로세스의 결함을 극복해 주기 위해서, Alvisi는 인과적 순서(causal order)를 이용하였고[6,9], Elnozahy는 정상 프로세스들의 수행을 멈추게 하거나[4], 회복 서버를 통한 중앙집중형 방법을 이용하였다[14]. 그러나, 제안하는 기법에서는 프로세스 P_i의 결함에 대하여, $HSet_i$라는 다음과 같은 자료구조를 이용한다.

3) 분산시스템의 Naming 기법에 의하여, 결함이 발생한 프로세스와 회복을 위하여 새롭게 생성되는 프로세스의 식별자는 같다.

[HSet$_i$] $HSet_i$는 집합으로서, 초기 값은 $\{P_i\}$이고, 원소는 프로세스 P_i에 대한 결정자를 요구받은 결함이 발생한 프로세스의 식별자이다.

제안하는 회복 알고리즘은 결함 발생 프로세스에 의한 결정자를 요구하는 알고리즘과 회복을 돕는 다른 프로세스들의 알고리즘으로 나뉜다. 결함이 발생한 프로세스를 P_i라하자. P_i는 검사점으로부터 프로세스의 상태와 자료구조를 회복하고 나서, $HSet_i$를 초기화한다. 결정자 정보를 획득하기 위하여 $CSet_i$에 속하는 프로세스에게 *Help*메시지를 보내게 된다. *Help*메시지에는 결함 발생 프로세스의 식별자인 P_i, $HSet_i$를 내용으로 갖고 있다. 만약 P_i로부터 *Help* 메시지를 받은 프로세스 P_h가 정상인 경우에는 P_i의 결정자 정보를 갖는 *OrderInfo* 메시지를 P_i에게 전달해 준다.

만약 P_h가 결함이 발생하여 회복 진행 중에 프로세스 P_j부터 P_i를 위한 *Help* 메시지를 받은 경우는 다음과 같다. 이때, P_i와 P_j는 동일 프로세스일 수 있다. 우선, $HSet_i$에 포함되지 않는 경우, P_h는 결함 발생 전에 P_i의 결정자 정보를 다른 프로세스에게 전달했을 가능성이 있으므로, 자신의 통신 집합인 $CSet_h$에 속하는 모든 프로세스에게 P_i의 결정자를 요구하는 메시지를 보내게 된다. $CSet_h$에 속하는 프로세스로부터 P_i의 결정자를 수집한 P_h는 자신이 갖고 있는 P_i의 결정자 정보와 수집한 결정자 정보를 갖는 *OrderInfo* 메시지를 P_j에게 전달한다. 반면, P_h가 $HSet_i$의 원소인 경우는 P_h가 결함이 발생한 프로세스 P_i의 결정자를 수집하기 위한 일련의 작업을 수행하고 있거나 완료하였음을 나타낸다. 이때 작업을 수행하고 있는 경우는 P_h가 P_j로부터 P_i를 위한 *Help* 메시지를 받고 처리하는 중에 P_k ($\neq P_j$)로부터 P_i를 위한 *Help* 메시지를 받은 경우이다. 이런 경우는 오직 P_h와 P_i, P_j, P_k 모두 결함이 발생한 경우이고, P_h가 획득한 P_i의 결정자 정보는 P_j를 통하여 P_i에게 전달되므로 P_k에게는 결정자 정보를 전달할 필요가 없다. 또한 작업을 완료한 경우는 P_j로부터 수신한 *Help* 메시지내의 H_i에 P_h가 포함된 것으로서, 이미 P_i를 위한 결정자 정보 수집 및 전달을 완료하였음을 나타낸다. 따라서, P_h가 $HSet_i$의 원소인 경우는 결정자를 전달해 줄 필요가 없기 때문에 결정자가 없다는 *OrderInfo* 메시지를 P_j에게 전달한다.

자신과 통신을 수행했던 모든 프로세스로부터 *OrderInfo* 메시지를 통한 결정자 정보를 수집한 P_i는 이들 정보를 이용하여 재수행하면 일관된 전역상태가 된다. 프로세스가 결함으로부터 회복을 위한 알고리즘은 그림 4에 기술되어 있고, 회복을 돕는 다른 프로세스들의 알고리즘은 그림 5에 기술되어 있다.

S_i ← the latest checkpointed state from stable storage ;
Restore data structures including $CSet_i$;
$HSet_i$ = { P_i } ;
Send *Help*(P_i, $HSet_i$) message to each process in $CSet_i$;
Gather determinants from *OrderInfo*(P_i, *determinants*) messages ;
Replay the execution from determinants and recover to a consistent state ;

그림 4 프로세스 Pi의 결함 회복 알고리즘

```
//Ph receives Help(Pi, Hi) message from Pj
IF ( Ph's state == live)
      Send OrderInfo(Pi, determinants of Pi) message to
       Pj ;
ELSE {
     HSeti = HSeti ∪ Hi ;
     IF ( Ph ∉ HSeti ) {
           HSeti = HSeti ∪ { Ph };
           send  Help(Pi, HSeti) message
                 to each element in CSetj - HSeti;
           After getting replies from the processes
             which receive Help message,
           send  OrderInfo(Pi, determinants of Pi)
           message to Pj ;
       }
     ELSE
           Send OrderInfo(Pi, no determinant of Pi)
           message to Pj ;
     HSeti = {} ;
}
```

그림 5 프로세스 P_h의 *Help* 메시지 처리 알고리즘

제안하는 알고리즘을 그림 6을 가지고 살펴보면 다음과 같다. 프로세스 P_3는 결함이 발생하면, 정상 수행 시 검사점을 저장한 C_1 상태로 회복하게 되고, 다른 프로세스와의 일관성을 유지시켜 주기 위하여 자신의 통신 집합인 프로세스들 즉, $CSet_3$의 원소(P_1, P_2, P_4)들에게 자신으로부터 받은 결정자를 요구하는 *Help* 메시지를 보낸다. *Help* 메시지를 받은 프로세스들은 P_3로부터 받은 결정자 정보를 검색하여, P_3에게 전달해 주어야 한

다. 이때, 본 예제에서는 프로세스 P_2가 결함이 발생하였다. P_2의 결함 발생으로 인하여, 메시지 m_2와 m_3에 대한 정보를 P_3는 알 수 없게 된다. 그러나, 타임아웃(timeout) 메카니즘에 의하여, 프로세스 P_2가 회복을 시작하였을 때, P_3로부터 다시 *Help* 메시지를 받게 되고, 자신의 현재 회복 상태에 있으므로 $CSet_2$에 속하는 프로세스(P_1)에게 P_3에 대한 결정자를 요구하는 *Help* 메시지를 보내게 된다. P_1으로부터 P_3에 대한 결정자를 받은 P_2는 이를 P_3에게 *OrderInfo* 메시지로 전달하여 준다. 그럼, P_3는 자신의 통신 집합에 있는 모든 프로세스로부터 받은 *OrderInfo* 메시지를 바탕으로 재수행하여 일관된 전역상태로 회복하게 된다.

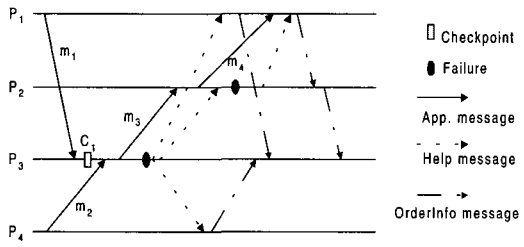

그림 6 회복 알고리즘 예제

5. 증 명

이 장에서는 4장에서 제시한 회복 알고리즘이 일관된 전역상태로 회복한다는 정당성을 증명하고, 제안한 알고리즘이 확장성을 지원함으로 인하여 추가적인 오버헤드가 발생하지 않음을 증명한다.

보조정리 1. 결함이 발생하지 않은 정상인 프로세스는 다른 프로세스의 결함에도 불구하고 복귀를 수행하지 않는다.

증명 [6,9]의 증명에 따르면, 인과적 메시지 로깅에 기반한 방법은 정상 프로세스의 복귀가 없다. 제안하는 방법은 정상 수행시 인과적 메시지 로깅 정책을 따르므로, 정상 프로세스의 복귀는 발생하지 않는다.□

보조정리 2. 정상인 프로세스는 어플리케이션 메시지를 처리하는 데 지연(delay)이 없다.

증명 다른 모든 프로세스가 정상인 경우에 어플리케이션 메시지를 처리하기 위한 제약사항이 없으므로 지연이 없다. 만일 다른 프로세스가 결함이 발생하여 회복과정에 있다고 가정하여 보자. 결함 발생 프로세스로부터 직접 받은 어플리케이션 메시지는 FIFO 가정에 의하여 결함 발생 전 혹은 결함으로부터 회복을 마친 후 보내진 메시지이다. 또한 처리한 메시지에 대해서는 결함 발생프로세스에게 *OrderInfo* 메시지를 통하여 결정자를 전달해 주기 때문에 결함 발생 프로세스는 전달받은 결정자를 바탕으로 재수행을 수행한다. 따라서, 제안된 기법에서는 전달받은 어플리케이션 메시지는 무조건 처리함으로 인하여 다른 프로세스의 회복과정으로 인한 기다림은 발생하지 않는다.□

정리 1. 결함이 발생한 프로세스는 제안된 회복 알고리즘을 이용하면, 일관된 전역 상태로 회복한다.

증명 결함이 발생한 프로세스를 P_f라 하자. 만약, $CSet_f$의 모든 원소가 정상 상태로 *OrderInfo* 메시지를 P_f에게 전달해 준다면 인과적 메시지 로깅의 기본 로깅 정책에 의하여 일관된 전역 상태로 회복된다. 반면, $CSet_f$의 원소 중 하나 이상의 프로세스가 결함이 발생한 경우를 고려해 보자. 프로세스 P_c를 $CSet_f$의 원소이면서 결함이 발생한 프로세스라 가정하자. 결함이 발생한 P_c는 P_f의 결정자 정보 중 일부를 상실할 수 있다. P_c가 상실한 결정자 정보를 M이라고 하자. P_c가 M을 다른 프로세스로 피기백하지 않았다면, P_c와 P_f가 서로 갖고 있지 않은 결정자 정보이므로 일관성을 위배하지 않는다. 만약 P_c가 M을 다른 프로세스에게 피기백했다면 분명 $CSet_c$의 원소인 프로세스가 받았을 것이다. 따라서 제안하는 알고리즘에서는 $CSet_c$에 해당하는 프로세스들에게 M을 요구하는 *Help*메시지를 보낸다. 이때, $CSet_c$에 속하는 모든 프로세스가 정상이라면 P_c는 M을 구할 수 있고 이를 P_f에게 보내면 된다. 그러나, $CSet_c$에 속하는 프로세스가 결함이 발생한 경우에는 P_c가 수행하는 오퍼레이션과 같다. 이때, *Help* 메시지를 계속해서 보내게 되는 데, $HSet_f$를 유지함으로 인하여, $HSet_f$에 속하지 않는 프로세스에게만 보낸다. 즉, 결함이 발생한 프로세스가 더 존재하면 할수록 *Help* 메시지의 수는 줄어든다. 따라서 P_c는 유한시간 내에 M을 구할 수 있다. 결국, P_c는 P_f에게 M을 전달하게 된다. 따라서 P_f는 모든 $CSet_f$의 원소로부터 결정자를 전달받게 되고 이를 바탕으로 일관된 전역상태를 회복하게 된다. 또한, 중간에 종료하는 프로세스가 발생하는 경우에도, 종료하는 프로세스는 자신의 갖고 있는 결정자를 결함이 발생한 프로세스에게 전달해주고 종료하기 때

문에 문제가 발생하지 않는다.□

정리 2. 제안한 결함 회복 알고리즘은 확장성 지원을 위한 추가적인 오버헤드를 유발하지 않는다.

증명 결함 발생프로세스의 회복시간(T_r)은 크게 세 가지로 나뉜다. 즉, 안전한 저장 공간에 저장된 내용을 새로 생성된 프로세스의 상태로 만드는 시간(T_1)과 재수행을 위해 필요한 결정자 정보를 획득하는 시간(T_2), 결정자 정보를 바탕으로 재수행하는 시간(T_3)이다[6,14]. 기존에 제안된 인과적 메시지 로깅의 회복 기법들과 비교해 볼때 제안한 결함 회복 알고리즘의 다른 점은 오직 T_2이다. 우선, 결함이 발생한 프로세스를 P_c, P_f로, 시스템내의 전체 프로세스의 집합을 N이라고 가정하고, T_2를 결함 발생 가능성 즉, 한 개의 프로세스 결함과 두 개 이상의 프로세스 결함으로 나누어 살펴보면 다음과 같다.

(1) 한 개의 프로세스(P_c) 결함

P_c는 N에 속하는 프로세스들과 통신을 수행함으로, $CSet_c$는 그 어떤 경우든 N의 부분집합이 된다. 따라서, 제안한 알고리즘은 N에 속한 모든 프로세스의 응답을 기다리지 않고, 오직 $CSet_c$에 속한 프로세스의 응답만 받으면 됨으로 T_2를 위한 부가적인 오버헤드는 전혀 없다.

(2) 두 개 이상의 프로세스 결함

비록 많은 프로세스가 결함이 발생한 상황이지만 P_c가 통신을 수행한 프로세스들이 모두 정상이면 ($\forall x, x \in CSet_c \rightarrow state(x) = Live$), 제안한 결함 회복 알고리즘은 단일 프로세스의 결함과 같은 오버헤드를 유발한다. 이는 브로드캐스팅보다 훨씬 저비용의 멀티캐스팅을 이용하기 때문이다. 반면, P_c가 P_f에게 *Help* 메시지를 보낸 경우를 고려해 보자. 제안한 알고리즘에서 P_f는 P_c를 제외한 $CSet_f$의 모든 프로세스에게 P_c를 위한 *Help* 메시지를 보내고 응답을 받고 다시 P_c에게 전달한다. 이때, P_f가 보내는 *Help* 메시지들이 시스템의 일관성을 유지하기 위한 필수적인 메시지라면 확장성을 지원하기 위한 추가적인 오버헤드가 아니다. 이를 위하여 그림 6의 상황을 고려해 보자. P_c에 해당하는 P_3는 P_f에 대응되는 P_2에게 *Help* 메시지를 보내고 P_2는 P_1에게 P_3를 위한 *Help* 메시지를 보냄으로써 메시지 m_4에 의하여 피기백된 m_2 결정자 정보를 획득하여 P_3에게 보낸다. 만약 P_3가 P_1으로부터 결정자가 존재하지 않는다는 메시지만을 가지고 회복을 수행하는 경우 m_4는 고아 메시지가 됨으로 인하여 시스템의 일관성을 잃게 된다. 즉, 인과적 순서(Causal Order)를 가정하지 않고 FIFO 만을 가정한 시스템이기 때문에 FIFO 순서에 의한 결정자 정보 획득이 요구된다. 따라서, P_f에서 이용되는 *Help* 메시지는 시스템의 일관성 유지를 위해 반드시 필요한 것으로서 확장성과는 무관하다.

따라서, 제안한 회복 알고리즘은 동시다발적인 프로세스의 결함에도 확장성 지원을 위한 오버헤드는 없다. □

6. 최적화

이 장에서는 4장에서 제안한 프로토콜의 성능 향상을 도모하기 위한 방법에 대하여 정상 수행과 회복 수행으로 나누어 기술한다.

정상 수행 시의 오버헤드는 안전한 저장장치에 유지하고 있는 *CSet*이다. 이로 인한 정상 수행 시 발생하는 오버헤드를 줄이는 방법으로, 결함포용을 제공하는 계층(layer)에서는 분산시스템내의 다른 모든 프로세스(프로세서)들과 통신을 바탕으로 동종의 어플리케이션을 수행하는 프로세스들의 목록을 조사하여 이들 목록을 어플리케이션 프로세스가 수행을 시작하는 시점에 미리 안전한 저장장소에 *CSet*으로 저장해 두는 방법이다. 이 방법은 어플리케이션을 수행하기 전에 작업 준비 시간에 대한 제약이 없는 경우에 활용될 수 있다. 개선된 방법으로, 어플리케이션 프로세스에서 통신을 수행하게 되는 프로세스들의 목록을 미리 추출 가능하다면, 결함 포용을 제공하는 계층이 이들 목록을 어플리케이션 프로세스가 수행을 시작하는 시점에 안전한 저장 장치에 저장해 두면 오버헤드를 줄일 수 있다. 어떤 방법을 이용하든, 프로세스의 생성과 소멸은 동적으로 발생하는 것이므로 기본적인 오버헤드는 분산 어플리케이션에 있어서 고유한 것이다.

제안된 회복 알고리즘은 정상 수행 시 메시지 정보를 중복시키는 형태가 인과적 로깅 정책을 사용하는 일반적인 방법에 기반하였다. 만약, 정상 수행 시 메시지 정보의 중복 방법이 그래프 형태로 유지된다면, 결함 발생 프로세스는 자신의 다른 프로세스들에게 도움을 요청하는 *Help* 메시지를 보내면서 안전한 저장장치로부터 회복된 자료구조 중 수신 순서 번호 *rsn*의 가장 큰 값을 피기백해서 보낸다. 또한, 결정자 자체를 중복시키는 방법에서는 회복된 자료구조 중 메시지를 보내면서 순서를 붙인 송신 순서 번호 *ssn* 을 피기백 해서 보낸다.

이 메시지를 받은 프로세스는 결함 발생 프로세스에게 필요한 결정자 정보가 작아지게 되고, 이로 인해 회복 속도의 향상을 기할 수 있다.

7. 관련 연구

비동기적 검사점과 메시지 로깅을 이용한 복귀 회복 기법은 프로세스의 결함으로부터 회복하는 데 널리 이용되어지고 있는 기법이다. 지금까지 저자들이 조사한 바에 의하면, 이들 연구들은 프로세스의 생성 및 소멸에 대한 고려가 전혀 없다. 또한 자신과 통신을 수행한 프로세스를 구별하지 않기 때문에 많은 비용이 소요된다고 알려진 브로드캐스팅을 회복을 위한 기본 메카니즘으로 요구한다. 그러나, 제안하는 기법에서는 정상 수행 시 확장성을 지원하고, 결함 발생 시 브로드캐스팅 메시지를 제거하였다. 이를 위하여 정상 수행 시 *CSet*이라는 자료구조를 안전한 저장장치에 유지하여 자신과 통신을 수행하는 프로세스에 대한 목록을 유지하도록 하였다. 또한 현재의 빠른 통신 환경에 가장 적합하다고 알려진 인과적 메시지 로깅 정책을 사용하였고, 회복 수행 시 발생하는 기존 인과적 메시지 로깅 프로토콜의 단점을 해결한 회복 알고리즘을 제안하였다.

인과적 메시지 로깅 프로토콜의 대표적인 예로 FBL [9] 과 Manetho[4]가 있다. 이들 기법은 결함으로부터 회복을 수행할 때 정상의 프로세스들의 진행을 멈추어야만 하는 문제점을 가지고 있다. 즉, 정상인 프로세스는 어플리케이션 메시지에 대한 처리를 결함 발생 프로세스가 회복을 다 수행한 이후에야 가능하다. 이러한 인과적 메시지 로깅의 문제점을 제안하는 기법에서의 *CSet*을 정상 수행 시 이용하고, 결함이 발생하고 나서는 제안하는 회복 알고리즘을 이용하면 해결할 수 있다. 제안하는 기법의 이러한 장점은 정상 수행 시 *CSet*을 안전한 저장장치에 유지하는 추가적인 비용이다. 그러나, 이 비용은 프로세스의 동적인 생성 소멸을 지원해주는 확장성 입장에서 볼 때 기본적인 비용이다.

Elnozahy는 기존 인과적 메시지 로깅 프로토콜의 회복을 개선하기 위한 알고리즘을 제시하였다[14]. Elnozahy가 제안한 알고리즘은 회복리더를 선출하여 이 회복리더가 모든 회복작업에 필요한 결정자 정보들을 수집하고, 필요한 프로세스에게 나누어 주는 방법을 취하고 있다. 이러한 기법은 회복 수행시 정상인 프로세스의 진행을 허용하여 준다. 반면, 회복 리더를 선출하기 위한 선출(election) 알고리즘이 필요하고, 결함이 발생한 프로세스는 회복 리더가 정보를 제공해 줄 때까지 무작정 기다려야 하며, 신뢰성있는 브로드캐스팅 메카니즘을 요구한다. 또한 확장성에 대한 고려도 없다.

지금까지 기술한 인과적 메시지 로깅 기법들과 제안된 기법을 항목별로 비교하면 표 1과 같다.

표 1 기존 인과적 메시지로깅 기법과의 비교표

	FBL	Manetho	Eln95	Ours
Non-blocking of normal process	NO	NO	YES	YES
Handle Multiple process failure	NO	YES	YES	YES
Distributed Recovery by failed process	YES	YES	NO	YES
Recovery without Broadcasting	NO	NO	NO	YES
Support Scalability	NO	NO	NO	YES

8. 결 론

이 논문에서 확장성을 인과적 메시지 로깅에 지원하기 위한 방법을 제시하였다. 정상 수행 시 각각의 프로세스들의 자신과 통신을 수행하는 프로세스에 대한 목록을 동적으로 유지하도록 함으로써 프로세스의 생성 소멸을 자연스럽게 해결해 주었다. 제안하는 방법은 비동기적 검사점 기법과 인과적 메시지 로깅 정책을 이용하여 결함에 대비하도록 하였다. 기존 인과적 메시지 로깅이 갖고 있는 단점인 결함 발생 후 정상 프로세스의 블록킹 문제를 *HSet*을 이용하여 해결하였다. 즉, 정상인 프로세스는 어플리케이션 메시지를 받아서 처리하기만 하면 된다. 분산 어플리케이션은 많은 형태의 통신 패턴을 가지고 있다[17,18]. 기존 연구들은 각각의 프로세스들이 모든 프로세스와 통신을 수행한다는 일반적인 하나의 패턴에 대하여 연구되었다. 그러나, 많은 어플리케이션에서는 이웃한 몇 개의 프로세스와 계속적인 통신을 수행한다. 이러한 어플리케이션인 경우, 제안하는 기법은 정상 수행 시 초기 수행을 제외하고는 추가적인 안전한 저장장치에 대한 접근이 필요 없다. 또한 제안하는 기법은 여러 프로세스의 결함이 발생하더라도, 개별적인 지역 정보에 근간을 둔 회복을 수행함으로 인하여 해당 결함 발생 프로세스와 통신을 수행했던 프로세스가 정상이라면 하나의 프로세스의 결함과 같다. 제안하는 방법을 낙관적 메시지 로깅에 적용하기 위한 연구가 현재 진행 중이다.

참 고 문 헌

[1] K. M. Chandy and L. Lamport, "Distributed snapshots:Determining global states of distributed systems," *ACM Trans. on Computer Systems*, pp.63-75, Vol. 3, No. 1, Feb. 1985.

[2] J. L. Kim and T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," IEEE Trans. on Parallel and Distributed Systems, pp.955-960, 1993.

[3] R. Koo and S.Toueg, "Checkpointing and rollback recovery for distributed systems," IEEE Trans. on Software Engineering, pp.21-31, 19987

[4] E.N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. on Computers*, pp.526-531, 1992.

[5] R. B. Strom and S. Yemeni, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems*, pp.204-226, 1985.

[6] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal and Optimal," *IEEE Trans. on Software Engineering*, pp.149-159, 1998.

[7] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *In Proc. Conf. on Fault-Tolerant Computing Systems*, pp. 14-19, 1987.

[8] S. Venkatesan, T. T. Y. Juang, and S. Alagar. "Optimistic Crash Recovery without Changing Application Messages," *IEEE Trans. on Parallel and Distributed Systems*, pp. 263-271, 1997.

[9] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and Orphan-Free Message Logging Protocols," *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pp.145-154, June 1993.

[10] K. Kim, "Message Interval Based Crash Recovery Reducing Recovery Overhead," MD thesis, Korea University Department of Computer Science and Engineering, 1996.

[11] O.P. Damani and V.K. Grag, "How to Recover Efficiently and Asynchronously when Optimism Fails," *Proceedings of the 16th Internationl Conference on Distributed Computing Systems*, pp.108-115, 1996.

[12] D.B. Johnson and W. Zwaenepol, "Recoevery in distributed systems using optimistic message logging and checkpointing," *Journal of Algorithms*, pp.462-491, 1990.

[13] D.B. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," *Proceedings of 12th Symposium on Reliable Distributed Systems*, Oct. 1993.

[14] E.N. Elnozahy, "On the relevance of communication costs of rollback-recovery protocols," *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp.74-79, August 1995.

[15] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems Concepts and Design*, 2nd Ed., Addison-Wesley, 1996.

[16] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, 1995.

[17] G.R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys*, pp.49-90, 1991.

[18] S. Rao, L. Alvsi, and H. M. Vin, "The Cost of Recovery in Message Logging Protocols," *Proceedings of 17th International Symposium on Reliable Distributed Systems*, 1998.

김 기 범

1994년 제주대학교 공과대학 정보공학과 졸업. 1996년 고려대학교 대학원 컴퓨터학과 졸업(이학석사). 1996년 ~ 현재 고려대학교 대학원 컴퓨터학과 박사과정. 관심분야는 분산시스템, 결함포용시스템, 이동컴퓨팅시스템, 운영체제

황 종 선

1978년 Univ. of Georgia, Statistics and Computer Science 박사. 1978년 South Carolina Lander 주립대학교 조교수. 1981년 한국표준연구소 전자계산실 실장. 1995년 한국정보과학회 회장. 1982년 ~ 현재 고려대학교 컴퓨터학과 교수. 1996년 ~ 현재 고려대학교 컴퓨터과학기술원 원장. 관심분야는 알고리즘, 분산 시스템, 결함포용시스템, 데이터베이스

유 헌 창

1989년 고려대학교 이과대학 컴퓨터학과 졸업. 1991년 고려대학교 대학원 컴퓨터학과 졸업(이학석사). 1994년 고려대학교 대학원 컴퓨터학과 졸업(이학박사). 1995년 ~ 1997년 서경대학교 이공대학 컴퓨터공학과 조교수. 1998년 ~ 현재 고려대학교 사범대학 컴퓨터교육과 조교수. 관심분야는 분산 시스템, 이동 컴퓨팅 시스템, 결함 포용 시스템, 웹기반교육

손 진 곤

1984년 고려대학교 이과대학 수학과(이학사). 1988년 고려대학교 대학원 전산학전공(이학석사). 1991년 고려대학교 대학원 전산학전공(이학박사). 1991년 ~ 1995년 한국경영과학회 정보기술연구회 운영위원. 1995년 ~ 1996년 한국정보과학회 학회지 편집위원. 1997년 ~ 1998년 SUNY at Stony brook (Visiting Professor). 1991년 ~ 현재 한국 방송통신대학교 컴퓨터과학과 부교수. 관심분야는 분산시스템, 컴퓨터통신망, Petri nets, 컴퓨터 보안

정 순 영

1990년 고려대학교 이과대학 컴퓨터학과 졸업. 1992년 고려대학교 대학원 컴퓨터학과 졸업(이학석사). 1997년 고려대학교 대학원 컴퓨터학과 졸업(이학박사). 1997년 ~ 현재 (주)ECO 연구개발실장. 관심분야는 분산 시스템, 데이터베이스, 결함포용 시스템