DBpia

論文 95-6-6-6

분산 화일시스템에서 액션을 기반으로한 일치성 제어기법

正會員 李 栽 完*, 周 洙 鍾**

A Consistency Control Mechanism Based on the Action in Distributed file Systems

Jae Wan Lee*, Su Jong Joo** Regular Members

이 논문은 1993년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음

要 約

본 논문에서는 분산 화일시스템에서 효율적인 동시성제어를 통한 일치성유지 기법을 제시한다. 분산 시스템에서 사용되는 데이타 영역을 데이타의 중요성에 따라 고장이 발생할 경우에도 항상 일치성을 유지하는 atomic 데이타 영역과 화일시스템에서 프로세스 처리의 형태와 같이 정상적으로 수행할 경우에만 일치성을 유지하는 non-atomic 데이타 영역으로 구성한다. 또한 프로그램 내에서 수행되는 액션의 구조를 수행 특성에 따라 직렬 액션, 밀착 액션, 프로세스 액션으로 구분한다. 자원의 효율적인 활용과 분산 시스템의 성능향상을 위하여 액션의 특성에 따라 적절한 동시성제어 및 일치성유지 기법을 제시한다.

ABSTRACT

This paper presents the mechanism on the preservance of consistency through an efficient concurrency control in distributed file systems. According to the importance of data, the data sets of the distributed systems are composed of the atomic data region which always preserves consistency in spite of failure and the non-atomic data region which preserves consistency only in case of completing processing successfully. Three new action structures in program are proposed - serial action, glued action, and process action. We show that the efficiency in utilizing the resource and the performance of the distributed system are achieved by providing the proper concurrency control and consistency management according to the structrues of actions.

*군산대학교 정보통신공학과
**원광대학교 컴퓨터공학과
論文番號 : 94362-1216
接受日字 : 1994年 12月 16日

www.dbpia.co.kr

Ⅰ. 서 론

컴퓨터 및 통신 기술의 발달에 따라 응용 분야의 확장으로 인한 정보 서비스 및 광역화 처리는 단일 시스템 처리의 한계를 벗어나 분산 환경 체제의 연구가 불가피하게 되었다. 분산 시스템에서 자원 공유를 통한 자원의 이용성(availability) 향상을 위해서, 자원은 여러 노드에 걸쳐 중복 할당되며 지역적으로 분산된다. 이러한 분산 환경에서 자원을 분산 프로세스들이 동시에 접근하려할 때 프로세스들 간에 동시성 제어를 통한 자원의 일치성 유지의 문제가 대두되며, 이를 위해서 분산 시스템에서는 트랜잭션(transaction) 처리의 개념을 사용한다.

트랜잭션 기법은 시스템의 일치성과 신뢰성을 유지하기 위해서 데이타 베이스 시스템에서 널리 연구되어 왔으며 이러한 개념은 분산 운영체제에서 점차 확대되어 연구되고 있다[5,6]. 지금까지 대부분의 연구는 하나의 트랜잭션을 평형(flat)구조로 처리하고 있으며, 로킹(locking)방법을 사용하는 동시성제어에서는 액션의 수행이 완료되면 소유하고 있는 모든 자원을 해제하고 수행을 종료한다. 그러나 이러한 평형구조의 수행방법에서는 프로세스의 수행이 장시간을 요구하는 경우에 자원을 독점하는 시간이 너무 길어지며, 따라서 그 자원을 사용하기 위하여 다른 트랜잭션들이 대기하는 시간이 길어져 자원의 효율적인 사용이나 동시성제어에 많은 문제점을 야기하게 된다. 또한 장시간을 요구하는 트랜잭션이 취소될 경우에 수행한 모든 작업들을 취소시켜야하기 때문에 그에 대한 손실이 너무 크다[2,3,11]. 이러한 문제는 하나의 트랜잭션이 직렬화가능성의 이론에 바탕을 둔 평형구조로 처리되기 때문에 발생하며 제한된 동시수행만을 제공한다.

따라서 일치성을 유지하면서 동시수행을 높이고 수행비용을 감소시키기 위해서 트랜잭션의 특성에 따라 처리기법을 달리하는 방법들이 연구되고 있다. Argus시스템[7]은 트랜잭션이 비원자성(non-atomic)의 데이타를 접근할 수 있도록 하여 액션간에 상호작용을 할 수 있도록 하였다. Avalon/Camelot 시스템[8]은 사용자가 복구를 위한 완료(commit)과정을 스스로 작성할 수 있도록 하였다. 분산 운영체제인 Cloud[9]에서는 트랜잭션의 처리 방법과 프로세스 처리 방법을 혼합하여 사용함으로써 일치성 및 동시성 제어의 효과를 높이는 방법을 제시하였다. Locus시스템[4,10]은 프로세스와 트랜잭션이 동시에 존재할 수 있도록 하였다. 트랜잭션은 non-atomic 데이타를 접근하여 록(lock)할 수 있으며, 이 록은 프로세스와 트랜잭션에서 같은 적용을 받는다. 이는 트랜잭션이 직렬성은 보장하지만 같은 데이타에 접근하는 프로세스에게는 동시수행에 너무 많은 제약을 가한다.

본 연구에서는 분산시스템에서 데이타의 일치성을 효율적으로 관리할 수 있는 기법을 제시 하고자 한다. cloud 시스템[9]에서와 같이 트랜잭션 처리 방법과 프로세스 처리 방법을 분리하지 않고 혼합하여 사용한다. 분산 시스템에서 사용되는 데이타 영역을 중요성에 따라 고장이 발생할 경우에도 항상 일치성을 유지하는 atomic 데이타 영역과 프로세스 처리의 형태와 같이 정상적으로 수행할 경우에만 일치성을 유지하는 non-atomic 데이다 영역으로 구성하였다.

또한 프로그램 내에서 독자적인 작업수행의 최소단위를 액션이라 정의하고, 수행되는 액션의 구조를 수행 특성에 따라 직렬 액션, 밀착 액션, 프로세스 액션으로 나누어 각 액션의 종류별로 로킹(locking)방법에 의한 동시성 제어 기법을 제시한다. 직렬 액션과 밀착 액션은 수행중에 발생한 데이타를 버전을 통해 관리함으로써 액션의 수행중에 수정된 데이타는 고장의 발생시에도 항상 데이타의 일치성을 유지하도록 하였으며, 프로세스 액션은 일반 프로세스 처리 기법을 따르는 액션으로 정상적으로 수행을 완료했을 경우에만 일치성을 보장하도록 하였다. 또한 각 데이타 영역에 대한 접근 규칙을 정의함으로써 트랜잭션의 수행중에 고장이 발생할 경우에 atomic 데이타 영역에 대해서는 항상 일치성이 보장되도록 하였다.

Ⅱ. 액션의 정의 및 구조

액션(action)이란 프로그램내에서 독자적인 작업수행의 최소 단위를 말한다. 프로그램은 여러개의 수행 액션으로 구성되며, 일반적으로 그림 1과 같은 중포 구조(nested structure)를 갖는다[1,12]. 그림 1에서 3개의 액션 A, B, C 에서 A를 최상위 액션(top_level action)이라 하며 액션 B와 C는 A에 내포되어 있고 B는 C보다 먼저 수행되어야 한다는 것을 의미한다.

www.dbpia.co.kr

2.1. 액션의 정의

2.1.1 데이타 영역에 대한 정의

〔정의 1〕: 오류가 발생할 경우에도 항상 일치성이 유지되는 영역을 atomic 데이타 영역이라 한다. 이 영역에서 오류가 발생하였을 경우에는 트랜잭션 처리기법에 따라 시스템에서 복구한다.

〔정의 2〕: 프로세스가 오류없이 정상적으로 수행될 경우에는 일치성이 유지되지만 수행중에 오류가 발생하면 일치성이 유지되지 않는 영역을 non-atomic 데이타 영역이라 한다. 이 영역은 일반 프로세스 수행방식으로 수행되며 오류가 발생하면 시스템에서 복구하지 않고 사용자가 복구를 담당하는 영역이다.

분산시스템의 데이타 영역은 데이타의 중요성 및 특성에 따라 시스템에서 항상 엄격한 일치성을 유지 시켜야 하는 atomic 데이타 영역과 사용자가 오류에 대한 복구 및 동시성제어를 담당하는 non-atomic 영역으로 구성된다.

2.1.2 액션에 대한 정의

〔정의 3〕: 트랜잭션의 일반적인 형태로서 일치성을 유지하며 내포된 액션간에 직렬적으로 수행되는 액션을 직렬 액션(serial action)이라 한다.

〔정의 4〕: 두 개의 액션 A_i와 A_j에서 A_i는 A_j보다 먼저 수행되고 수행결과는 일치성을 유지하며, R_i와 R_j를 각각 A_i와 A_j의 자원 집합이라 할때 다음 조건을 만족하는 액션을 밀착 액션(glued action)이라 한다.

① $(R_i \supseteq R_j)$

② A_i의 수행이 끝나는 시점과 A_j의 수행이 시작되는 시점 사이에 R_j의 자원은 다른 액션에 의해 변경될 수 없다.

밀착 액션은 직렬 액션의 특별한 경우라 할 수있다.

〔정의 5〕: 프로세스 액션(process action)이란 오류 발생시 일치성을 유지시키지 못하는 액션으로 정상적으로 수행을 완료했을 경우에만 일치성을 유지하는 액션이다. 이 액션은 화일 시스템의 일반 프로세스 처리 기법에 따라 오류가 복구 되고 일치성이 유지된다.

2.2 액션의 구조

본 절에서는 프로그램에서 흔히 볼 수 있는 3가지의 구조를 제시하고 로킹 기법으로 동시성 제어를 수행할 때 액션간의 관계를 기술한다.

2.2.1 일치성 액션

일치성 액션(CP Action : Consistency Preserving Action)은 시스템에서 동시성 제어 및 복구를 담당하며 트랜잭션의 처리기법을 따르는 액션을 말한다. 액션이 수행될 때마다 생성되는 수정자료에 대해서는 영속적인 자료에 대해 직접 영향을 미치는 것이 아니고 쉐도우 버전을 생성하여 수행한다. 액션의 수행이 성공적으로 끝나면 수정자료는 메모리 상의 기초 버전(base version)이나 안전 버전(stable version)에 완료된다. 일치성 액션은 직렬 액션(serial action)과 밀착 액션(glued action)으로 구성된다.

① 직렬 액션

트랜잭션의 일반적인 형태로서 내포된 액션간에 직렬적으로 수행된다. 액션간에 다른 액션이 끼어들어 수행됨으로써 발생하는 불일치를 방지하기 위해서 하나의 액션에서 인접한 다른 액션으로 록이 이전되는 기법이 필요하다.

그림 2에서 B의 수행이 끝났을 때, B가 보유한 록을 A에 넘겨주며 C와 같이 내포된 액션에서 이를 사용한다.

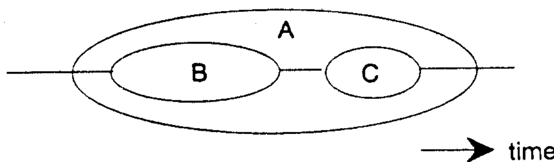


그림 1. 중포 프로그램의 구조
Fig. 1. Structure of nested program

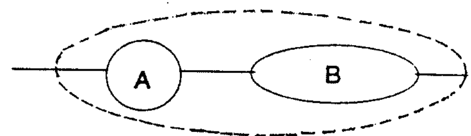


그림 2. 직렬 액션
Fig. 2. Serial action

www.dbpia.co.kr

② 밀착 액션

이는 직렬 액션 중 특별한 경우로서 장시간을 요구하는 액션의 경우에 동시 수행의 효과를 높이기 위해 구분하였다. 장시간의 수행을 요하는 트랜잭션이 완료될 때까지 록을 보유하면 동시수행에 막대한 지장을 주기 때문이다. 동시 수행의 효과를 높이는 한 가지 방법으로는 록을 조기에 해제하여 데이타의 활용성을 높이는 것이다.

조기에 록을 해제하였을 경우에 발생하는 문제는 액션이 취소되면 그 액션이 수행한 결과를 접근한 다른 액션이 연쇄적으로 취소(cascading abort)되어야 한다는 점이다. 밀착 액션은 이러한 연쇄적인 취소가 일어나지 않는 액션의 경우를 말한다. 두 개의 액션 A와 B로 구성된 구조에서 A는 B에서 필요한 데이타를 읽는 액션이며 B는 장시간 수행을 요하는 액션일 경우를 예로 들어보자. 액션 A의 기능은 데이타 집합 $P = \{O_1, O_2, \ldots\ldots, O_n\}$를 검색하여 P의 부분집합인 $Q = \{O_i, \ldots\ldots, O_k,\ 1 \le i \le k \le n\}$를 추출한다. 이 부분 집합은 액션 B로 전달되고 액션 B는 Q를 이용하여 장시간 수행한다. Q의 데이타는 A의 끝에서 B가 시작되는 시점까지 다른 액션에 의해서 변화되어서는 안된다. 또한 P에 관한 A의 효과는 B가 취소될 때 같이 복구가 되어서는 안된다.

마지막의 조건으로부터 액션을 최상위 액션으로 내포시킬 필요성이 없어진다. 즉 그림 3(a)와 같이 A와 B를 최상위 액션으로 수행시킬 수도 있지만 Q에서의 자료들이 A의 끝에서 B의 시작까지 변화되어서는 안된다는 조건을 만족시키지 못한다. 또 다른 방법으로는 그림 3(b)와 같이 직렬 액션으로 A와 B를 내포시키는 것이다. 이 방법은 P의 집합에서 Q의 집합을 제외한(P - Q) 데이타의 read와 write 록이 B가 끝날 때까지 해제되지 않기 때문에 다른 프로세스들이 사용할 수없으며 B에서는 이러한 자료를 요구하지 않기 때문에 분명히 바람직한 방법이 아니다. 이러한 경우에는 그림 4와 같이 A를 B에 밀착시키고 해당 자료의 록을 A에서 B로 이전시키고 나머지 록은 A가 완료될때 해제시킨다. 밀착시킨다는 것은 다른 액션들이 록을 취득할 수 없도록 록을 A에서 B로 이전한다는 것을 의미한다.

A가 보유한 모든 록이 B에 이전될 경우에 밀착액션은 직렬 액션과 같다. 즉 직렬 액션은 밀착 액션의 특별한 경우로 볼 수있다.

2.2.2 프로세스 액션

프로세스 액션은 일반 프로세스 처리의 방법을 따르는 액션으로 동시성 제어에 대한 로킹 방법을 액션의 프로그래머가 스스로 정의하며 이에따라 일치성이 보장된다. 프로세스 액션은 수정자료에 대해 쉐도우 버전을 생성하지 아니하며 곧바로 영속적인 버전에 수정한다. 그러므로 정상적으로 수행을 완료하였을 경우에는 일치성이 보장되지만 수행중에 고장이 발생하였을 경우에는 시스템에서 데이타의 일치성을 보장하지 않는다.

프로세스 액션은 일치성 액션의 경우와 같이 시스템에서 제공하는 로킹방법을 사용하지 않고 사용자들이 정의하는 일반 프로세스의 기법을 따르기 때문에 자원의 록에 따른 대기의 시간이 줄어들어 동시수행의 효과를 높일 수 있다.

수행중에 고장이 발생할 경우 사용자가 쉽게 수정할

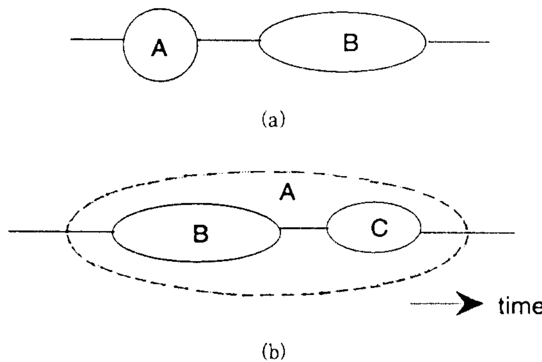


그림 3. 밀착 액션의 부적절한 수행
Fig. 3. Unappropriate execution of glued action

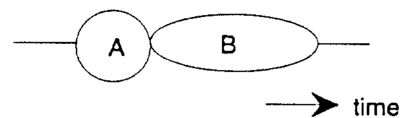


그림 4. 밀착 액션
Fig. 4. Glued action

www.dbpia.co.kr

수 있으면서 긴급을 요하지 않는 자료들 즉, 임시 화일, 메세지 등의 문자화일, 이름 서버(name server)등을 접근하는 액션들은 프로세스 액션의 예가 될 수 있다.

프로그래머는 액션의 앞에 액션의 종류를 명시하며 이 액션의 종류에 따라 시스템이 수행하는 동시성 제어 방법이 결정된다. 일치성 액션인 직렬 액션이나 밀착 액션은 트랜잭션 처리 방법에 따라 동시성 제어가 수행되며, 프로세스 액션은 사용자가 정의한 동시성 제어 기법에 따라 동시성 제어가 수행된다.

Ⅲ. 액션의 관리

3.1 액션의 생성

프로그램은 액션의 호출에 따라 수행중에 새로운 액션을 생성할 수 있다. 새롭게 생성된 액션을 자(Child) 액션이라 하며, 자 액션을 생성한 액션을 부모(Parent) 액션이라 한다. 부모 액션은 자 액션의 상태(제거, 지연, 재시작)를 관리한다.

최상위 액션은 수행중에 발생한 수정분을 영속적인 상태로 완료하지만 중포된 액션은 부모 액션에 수정분을 완료한다.

3.2 로킹 규칙

일치성을 필요로하는 액션은 2단계 로킹 규칙에 따라 수행한다. 일치성 중포 액션의 로킹 규칙은 Moss[1]가 제안한 로킹 규칙과 유사하다. 액션의 생성은 공동의 최상위 액션을 가진 트리 구조의 일치성 액션을 생성한다. 이러한 액션 트리 구조를 일치성 액션 트리 구조라 한다. 일치성 액션 트리 구조의 로킹 규칙은 다음과 같다.

① 데이타를 록한 모든 액션이 read 록을 가지든지 또는 write 록을 소유한 액션이 요청하는 액션의 ancester이면 read 록을 취득한다.

② 데이타에 대해 write 록을 소유한 모든 액션이 요청하는 액션의 ancester 이면 write 록을 소유할 수 있다.

③ 자 액션의 수행이 완료되지 아니하면 액션은 영속적인 상태로 완료하지 못한다.

④ 자 액션이 소유한 록은 완료나 취소시 부모 액션으로 이전한다.

⑤ 부모 액션이 취소되면 그에 속한 모든 자 액션은 취소된다.

3.3 버전 관리

중포 액션의 로킹 규칙은 완료에 이르지 못한 수정자료를 버전 스택(Version Stack)의 개념을 이용하여 완료(commit)한다. 전체 스택은 휘발성(volatile)의 메모리에 존재하며 안전 저장소에 있는 안전 버전에 근거를 둔다.

버전 스택의 버전은

0, 1, 2, ...n-1, n 이다

0 : 영속적인 버전

n : 최상위 버전

일치성 액션의 버전 생성은 액션 트리 구조에서 액션의 중포되는 수준에 준한다. 이러한 버전 스택의 개념으로 일치성 액션은 쉐도우 버전에서 작동할 수 있도록 하고 액션 단위로 버전을 완료 또는 취소시킨다.

깊이 d인 중포 액션 T가 전에 접근하지 않은 자료에 대해서 write 록을 얻으면 그 자료에 대한 새로운 버전이 생성되며 버전 스택의 최상위값을 초기화시킨다. n이 버전 스택의 깊이이고, $d - n > 1$ 이면 $d - n - 1$개의 새로운 버전이 생성되고, $n + 1$, $n + 2$, ...$d - 1$의 버전으로 버전 스택에 올려진다.

액션은 수행하지 않은 자 액션이 없을 때만 영속적인 상태를 수정할 수 있다는 규칙과 로킹 규칙으로 인하여, 액션당 버전 스택의 각 스택 계층에는 단지 하나의 버전만이 존재한다. 즉, 스택은 항상 선형의 스택이 된다.

일치성 액션이 완료될 때, 수정된 자료의 모든 항목이 완료되든지 취소된다. 깊이가 n인 중포 액션은 버전 스택의 최상위 버전을 버전 스택의 그 다음 최근 버전에 복사하고 최상위 버전을 삭제한다. 즉, 중포 액션은 수정자료를 부모 액션의 중간 버전에 완료된다. 깊이가 1일때, 최상위의 중포 액션은 영속적인 버전에 수정자료를 완료한다.

3.4 액션의 상호작용

프로세스 액션은 중간 버전을 생성하지 않고, 항상 안전 버전에 직접 수정함으로 수정본은 즉시 다른 액션에게 안전 뷰(Stable view)를 제공한다. 이는 일반 프로세스 처리 형태의 일치성을 유지한다. 중간 버전을 생성하지도 아니하며 로킹도 일반 프로세스 처리 기법에서의 경우처럼 사용자가 정의한 것에 따라 수행한다. 그러므로 동시성의 효과를 증대시킨다는 관점에서는 효율적이

www.dbpia.co.kr

나 사용자의 동시성 제어기법에 따라 자료의 일치성을 보장하지 못하는 문제가 발생할 수 있다.

일치성 액션의 트리 구조는 그림 5와 같이 순간 뷰(instantanious view)에서 동작하여 안전 뷰로 변화시킴으로써 시스템에서 일치성을 보장해 준다. 그러므로

일치성 액션들만이 수행되면 트랜잭션의 직렬성은 보장된다 atomic 데이타 영역에 대해서는 항상 엄격한 일치성이 보장되어야 한다. 이를 위해 각 액션이 데이타 영역을 접근하는 규칙은 다음과 같다.

① non-atomic 데이타 영역은 일치성 액션과 프로세스 액션 모두 read/write 접근이 가능하다.

② atomic 데이타 영역은 일치성 액션의 경우에는 read/write 접근이 가능하나 프로세스 액션에서는 read만 가능하다.

정리 1: 액션이 접근 규칙을 따르면 atomic 데이타 영역은 항상 일치성이 보장된다.

증명 : 규칙 ②로 인하여 프로세스 액션은 atomic 데이타 영역을 수정할 수 없으며, atomic 데이타 영역의 수정은 일치성 액션에 의해서만 가능하다. 그러므로 atomic 데이타 영역은 오류가 발생할 경우에도 항상 일치성이 보장된다.

정리 2: 프로그램 내에 일치성 액션과 프로세스 액션이 공존할 경우, 수행후 데이타의 일치성은 보장된다.

증 명 : 일치성 액션과 프로세스 액션의 상호작용에서 발생할 수 있는 경우는 다음의 3가지 이다.

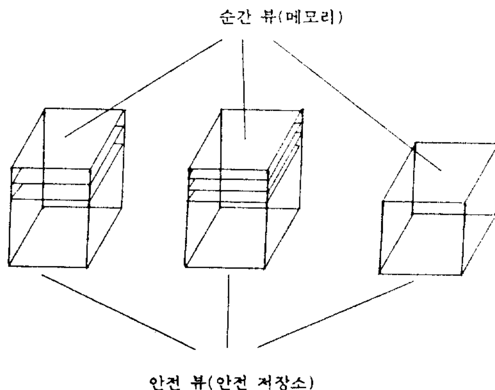


그림 5. 순간 뷰와 안전 뷰와의 관계
Fig. 5. Relation of instantanious view and stable view

① 일치성 액션과 프로세스 액션 모두 정상적으로 수행을 완료한 경우 프로세스 액션이 정상적으로 수행을 마쳤을 경우에는 데이타의 일치성을 보장한다는 정의에 따라 일치성을 보장한다.

② 일치성 액션만 수행된 상태에서 프로그램이 중단된 경우 수행중에 생성한 버전이 삭제되기 때문에 atomic 데이타 영역의 데이타는 수행전의 상태로 복구되고 일치성이 유지된다.

③ 일부분의 일치성 액션과 프로세스 액션이 실행된 상태에서 프로그램이 중단된 경우

ⓐ atomic 데이타 영역 : 일치성 액션은 임시 버전에서 수행하였기 때문에 프로그램이 취소될 경우 임시버전을 삭제함으로써 일치성이 유지된다. 프로세스 액션은 데이타 접근 규칙에 따라 이 영역을 수정하지 못함으로 일치성이 유지된다.

ⓑ non-atomic 데이타 영역 : 이 영역은 사용자가 복구한다는 정의에 따라 프로세스 처리 관점에서 일치성이 유지된다.

프로세스 액션을 생성할때 프로그래머는 프로세스 액션 내의 일치성 유지 및 프로그램 내의 일치성 액션과의 관계를 고려하여 생성하여야 한다. 프로그램 내에서의 일치성이 중요한 요인이 되지 않는 주소 서버와 같은 경우에는 일치성 액션이 취소되어도 복구가 요구되지는 않는다. 또한 지역적으로 사용되는 화일처럼 사용자가 복구할 수 있는 경우나 문자 화일 등을 처리할 경우에는 분산 시스템의 성능을 고려하여 프로세스 액션으로 구성한다.

프로세스 액션을 적절하게 활용함으로써 분산 시스템 전체의 자원의 이용성을 향상시킬 수 있으며, 분산 프로세스들 간에 최대한의 동시수행을 제공함으로써 분산 시스템의 전체적인 성능을 향상시킬 수 있다.

Ⅳ. 분산 시스템에서 일치성 제어기법 설계

분산 시스템의 각 노드는 각자의 자원을 관리하며 프로세스가 원격 노드(remote node)에 있는 자원을 접근하려면 원격 프로세스에 요청하여 그 작업을 수행하고 결과를 받아온다. 이러한 작업들은 원격 프로시듀어 호출(remote procedure call)에 의하여 수행된다. 분산 공유 메모리 시스템은 분산 시스템에서 모든 노드들의 메모리를 분산 프로세스들이 공유할 수 있는 하나의 논

www.dbpia.co.kr

리적인 단위로 통합한 가상주소공간(virtual address space)을 제공한다. 따라서 분산 공유 메모리 시스템은 분산 프로세스가 이 가상 주소 공간을 지역(local) 메모리로 사용하는 방법과 같이 수행할 수 있도록 한 또 다른 기법이다[13,14]. 분산 공유 메모리 시스템은 다음과 같은 특징을 갖는다.

① 분산 공유 메모리 시스템에서는 프로그램들이 광역 주소공간(global address space)에 존재하기 때문에 원격 프로세스를 호출할 때, 운영체제에서는 노드의 영역이 서로 다를지라도 사용자들에게는 투명성(transparency)을 제공해 준다.

② 프로세스와 데이타는 영속적인 가상 주소 공간(virtual address space)을 형성한다.

각 노드는 주 메모리의 일부를 캐쉬로 사용하여 공유 메모리 공간의 자료를 접근한다. 한 노드의 프로세스가 자기 노드의 기억장치(주 기억장치 또는 보조 기억장치)에 없는 데이타를 접근할 경우에는 네트워크 페이지 부재(fault)가 생하며, 해당 페이지는 원격 노드로부터 가져와서 수행된다. 일반적으로 프로세스들의 수행은 지역성(locality)을 나타내기 때문에 원격 메모리에서 가져온 페이지는 지역 메모리에 접근하는 시간과 같은 효과를 가져올 수 있다. 이를 지원하는 하부의 가상 메모리 시스템이 이러한 공유 메모리의 작업 과정을 수행한다.

4.1 일치성 제어 및 복구 모델

액션이 수행 중에 수정한 모든 페이지는 시스템 고장이나 오류가 발생하였을 때 복구가 가능하여야 한다. 복구를 위해서 일치성 액션이 수정한 모든 페이지에 대해서는 액션별로 새로운 버전을 생성한다. 공유 메모리 시스템에서 새로운 버전을 생성하는 기법으로는 액션이 수정한 페이지만을 지역 저장소에 저장하는 쉐도우 페이지(shadow page)기법을 사용한다. 액션은 여러 노드에 있는 페이지에 걸쳐 수행되기 때문에 2단계 완료 프로토콜(2 phase commit protocol)을 사용한다.

쉐도우 페이지는 프로그램 관리자의 완료 로그로 사용된다. 수정된 페이지만을 기록함으로써 2단계 완료 프로토콜의 1단계인 준비 단계에서 새로운 버전 준비를 위한 저장 영역을 절약할 수있다. 1단계 precommit의 결과는 세그먼트의 쉐도우 버전이며 원래의 세그먼트 헤더는 쉐도우 버전을 가리킨다.

2단계 절차에서 쉐도우 페이지는 저장 영역에 할당되어 영속적인 버전이 된다. 액션이 취소되면 현 세그먼트 헤더는 원래의 세그먼트 헤더로 바뀌어 지고, 쉐도우 버전으로 할당된 페이지들은 할당이 취소된다.

액션의 취소로 인해 액션이 재시작될 때는 3.3절에서 언급한 바와 같이 버전을 액션별로 관리함으로써 취소된 액션으로부터 재시작한다. 따라서 한 액션의 취소로 인해 전 프로그램이 취소되는 문제점을 방지한다.

분산 공유 메모리 환경에서 프로그램의 동시성 제어 및 오류 복구에 관한 설계 구조는 그림 6과 같다.

1) 버전 관리기(Version Manager)

주 기억장치와 보조 기억장치 사이의 버전 이동, 보조 기억장치에서의 안전한 자료관리를 담당한다. 이는 임시 버전의 생성, 완료, 취소, 그리고 임시 버전의 디스크 전송 및 디스크로부터 최근 버전 이동 등을 포함 한다.

(1) 분산 공유 메모리(Distributed Shared Memory) 시스템 공유 메모리 내에서의 일관성을 유지시키는 역할을 한다. 프로세스들은 해당되는 모든 노드들이 마치 지역 노드에서 수행되고 있는 것처럼 수행되며 세그먼트에 대한 버전의 생성, 완료, 취소 등을 수행한다.

(2) 복구 관리기(Recovery Manager)

복구 관리는 페이지 가져오기나 복구 요청을 페이지의 위치에 따라 분산공유 메모리 관리자나 지역 복구 관리 시스템에 요청하여 수행한다.

2) 분산 록 관리기(Distributed Lock Manager)

분산 공유 메모리의 일관성 유지 및 데이타의 일치성을 유지하기 위해서 분산 시스템의 로킹을 담당한다. 분산 록 관리기는 다음과 같은 록 테이블을 유지한다.

segment.page#	lock_mode	copyset	lock_owner

이 테이블 엔트리에서 lock_mode는 해당 페이지의 록 상태를 나타내며 read, write, none 중의 하나이다. none은 페이지에 록이 없는 경우를 뜻한다. copyset은 복사본을 가진 노드의 집합을 나타내며, lock_owner는 록을 소유한 프로그램의 번호를 기록한다. copyset은 페이지의 무효화 등의 일관성 유지를 위한 정보를 유지한다.

www.dbpia.co.kr

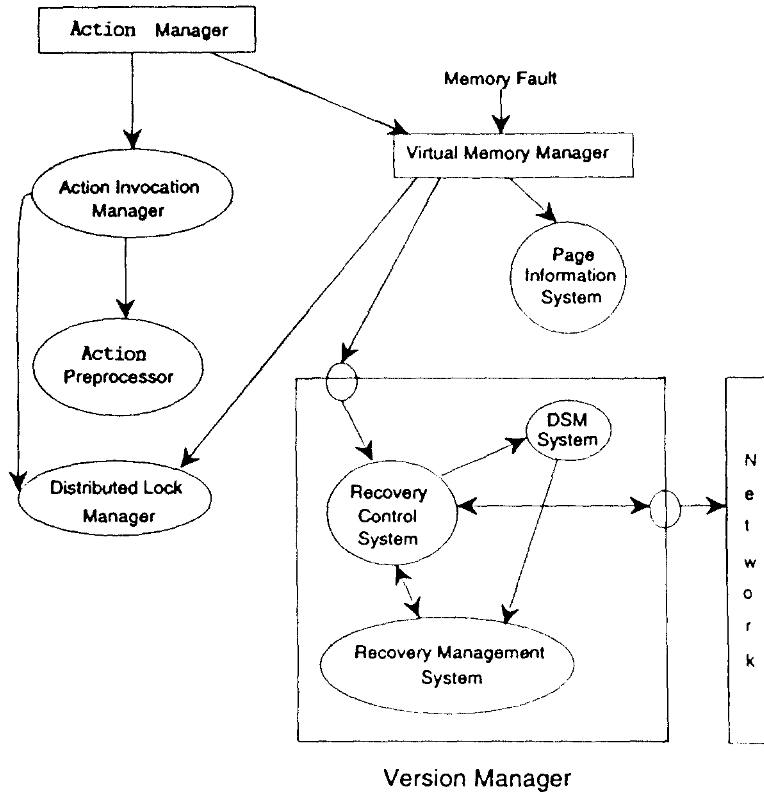


그림 6. 일치성제어 및 복구 모델
Fig. 6. Model of consistency control and recovery

분산 메모리 관리자는 지역 로킹 방법만으로는 부족하기 때문에 록의 요청 및 록의 정보는 광역적으로 유지하며 분산 록 관리기에의해 수행된다. 알고리즘은 그림 7과 같다.

3)액션 전 처리기 (Action Preprocessor)

액션의 호출에서 밀착 액션이 호출되면 밀착 액션에서 사용하는 영속적인 자료와 전에 수행된 액션에서 수행할 밀착 액션에 영향을 준 영속적인 자료를 검사한다. 이들 자료 항목은 록이 해제되어서는 안되며 이 자료 항목들의 목록을 록 목록(lock_list)이라 한다. 프로그램에서 사용한 모든 데이타 항목 중에서 록 목록에 없는 데이타 항목은 록을 해제시키는 대상이 되며, 프로그램의 수행 중에 이들 데이타의 록을 해제하기 위해 록 해제목록(unlock_list)에 저장 한다.

즉 unlock_list = R_P - { lock_list } 이다.

R_P : 프로그램에서 사용한 모든 데이타 항목의 집합.

또한 록을 해제할 페이지 목록(page_list)인

Unlock_pagelist = page_list of R_P - page_list of {lock_list} 이다.

4)액션 호출 관리기(Action Invocation Manager)

액션 호출 관리기는 각 액션이 호출될 때마다 액션의 종류를 검사한다. 일치성 액션의 수행이 끝나면 그 액션이 최상위 액션일 경우에는 영속적인 버전에 완료하고 그렇지 않으면 부모 액션에 완료한다.

프로세스 액션이 호출되면 일반 프로세스 처리기법에 따라 수행된다. 따라서 쉐도우 버전이 생성되지 않고 프

www.dbpia.co.kr

```
DLM(page,lock_mode,PID)
/* lock request */
case(lock_mode)
Read:
        if(lock(page.read.PID) == nil) then return(0)
        if(Info[page].lock_mode == none)
          Reconstructinfo(page,read,PID)
        else
          Info[page].lock_owner = Info[page].lock_owner U PID
Write:
        if(lock(page.write.PID) == nil) then return(0)
        if(Info[page].copyset == nil)
          Reconstructinfo(page,write,PID)
        else
          Invalidate(page,info[page].copyset)
          Info[page].copyset = { }
          Reconstructinfo(page,write,PID)
end_case

/* unlock request */
case(exist_lock_mode)
   Read:
        Info[page].lock_owner = Info[page].lock_owner - PID
        if(Info[page].lock_mode == nil) then
        Info[page].lock_mode = none
Write:
        Infor[page].lock_mode = none
end_case
```

그림 7. 분산 록 관리기의 알고리즘
Fig. 7. The algorithm of distributed lock manager

로세스의 수행의 경우와 같이 사용자가 정의한 동시성 제어 기법을 따른다.

일치성 액션이 호출되면 버전 관리기를 호출한다. 버전생성을 위해 쉐도우 세그먼트 헤더가 생성되며, 이 쉐도우 세그먼트 헤더는 어느 다른 프로그램에서도 접근할 수 없도록 보호된다.

밀착 액션이 호출되면 액션 전 처리기에서 작성한 록 해제목록을 가져와서 록 해제목록에 있는 데이타의 readset에 대해서는 해당 페이지의 록을 해제한다. 각 액션은 3.4절에서 정의한 atomic 데이타 영역과 non-atomic 데이타 영역에 대한 접근 규칙을 따라 수행한다. 각 데이타 영역은 정리 3.4절의 정리 2에 따라 일치성이 유지된다.

Ⅴ. 분석 및 성능평가

분산 공유 메모리 시스템 환경에서 기존의 트랜잭션의 처리 기법과 제시한 기법과의 자원 활용도를 중심으로 성능을 비교하였다.

두 기법 간의 성능 평가를 위한 요인으로는 록에 의한 대기 시간, CPU 실행시간, 버전 관리에 소요되는 시간을 중심으로 시뮬레이션 하였다. 시뮬레이션 방법은 사건 중심(event-driven)방식을 사용하였으며, 주요 사건으로는 프로그램의 발생, CPU 서비스, 자원 요구,

www.dbpia.co.kr

원격 자료 전송, 자원 처리, 버전 관리, 페이지록 및 해제 등이다.

5.1 시뮬레이션 가정 및 매개 변수

시뮬레이션을 수행하기 위해 설정한 가정은 다음과 같다.

- 프로그램의 도착시간 간격은 지수분포를 한다.
- 각 액션의 자원 요구 횟수는 포아슨 분포를 한다.
- 각 액션의 요구 자원수는 포아슨 분포를 한다.
- 페이지의 전송시간은 일정하다.
- 프로그램이 요구한 자원 번호는 균등분포를 한다.
- 하나의 페이지를 접근한 후 처리기가 수행하는 처리 시간은 지수분포를 한다.

프로그램의 수행에 필요한 각 항목의 단위 시간은 Cloud 시스템에서 측정한 수치를 이용하였으며 표 1과 같다[10].

시뮬레이션 실행시 매개변수로는 프로그램의 도착율과 프로그램내의 액션 종류별 구성 비율을 택했다.

5.2 시뮬레이션 결과

5.2.1 프로세스 액션의 구성비율을 변화시킨 경우

다른 모든 변인들은 고정시키고 프로그램에서 프로세스액션의 구성 비율을 0% 에서 100% 까지 변화시키면서 프로그램의 평균 반환시간을 비교하였으며 그 결과는 그림 8과 같다.

동일한 부하에서 프로세스 액션의 구성 비율이 증가할수록 프로그램의 평균 반환시간이 감소하는 것을 알 수 있다. 반환시간의 감소에 대한 대부분이 록 대기시간 및 버전 처리시간의 감소로 나타났다. 이는 프로세스의 처리기법에서 알 수 있는 바와 같이 버전의 생성을 하지 않고 데이타에 대한 록 및 록의 해제가 트랜잭션 처리기법과 다르기 때문에 나타난다. 일반적으로 프로세스 방식에서 록은 해당 명령이 끝나면 즉시 해제하는 경우와 전 프로세스가 끝날 때 해제하는 경우로 나누어 볼 수있다.

본 시뮬레이션에서는 편의상 중간을 택하여 프로세스 액션은 하나의 액션이 끝나면 록을 해제하는 것으로 시뮬레이션하였다. 프로세스 액션을 사용함으로써 공유 자원에 대한 대기 시간이 감소하여 자원에 대한 활용성이 증가되고, 버전처리에 소비되는 시간이 없어지게 되어 결과적으로 프로그램의 평균 반환시간이 감소하게 된다.

5.2.2 밀착 액션의 구성비율을 변화 시킨 경우

밀착 액션은 프로그램에서 맨 끝에 위치하며, 하나의 프로그램에서 오직 한 개만 존재하든지 또는 존재하지 않는다. 그러므로 밀착 액션의 구성 비율은 모든 프로그램에서 밀착액션을 포함하고 있는 프로그램의 갯수가 된다.

프로세스 액션의 구성 비율을 0%로 하고, 밀착 액션의 구성 비율을 0%에서 100%까지 변화시키면서 프로그램의 평균 반환시간을 비교하였으며 그 결과는 그림 9와 같다

표 1. 항목당 단위시간
Table 1. Unit time per item

항　　목	단위 시간
페이지당 원격 접근	16
지역 페이지 접근	1.5
페이지당 버전 생성	12
원격 페이지 lock	6
지역 페이지 lock	0.5

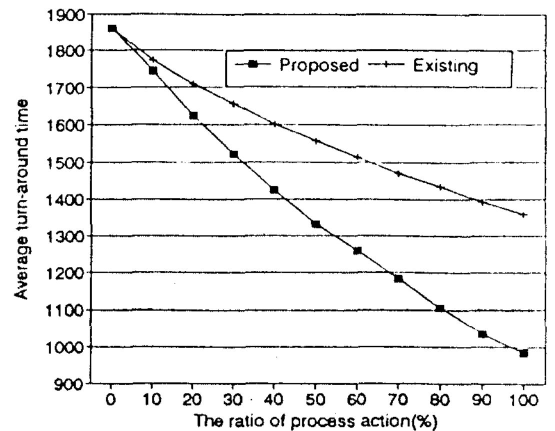


그림 8. 프로세스 액션의 비율에 따른 반환시간 비교
Fig. 8. The comparision of turn-arround time by the ratio of process action(%)

www.dbpia.co.kr

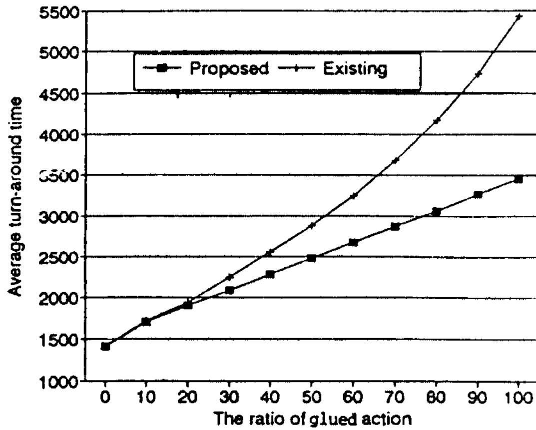


그림 9. 밀착 액션의 비율에 따른 반환시간 비교
Fig. 9. The comparison of turn arround time by the ratio of glued action

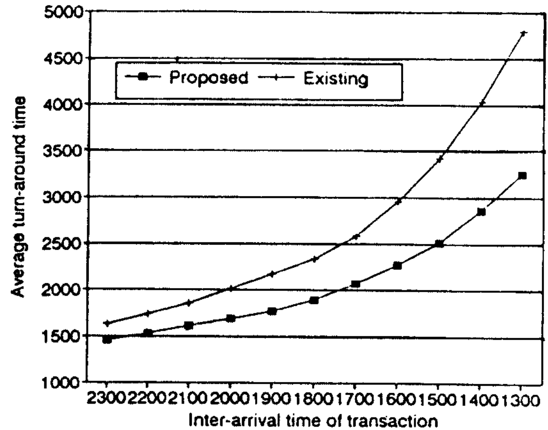


그림 10. 시스템 부하에 따른 반환시간 비교
Fig. 10. The comparison of turn arround time by system load

하나의 프로그램 내에서 액션의 수를 고정시키고 분산 공유 메모리 시스템에 있는 모든 프로그램에서 밀착 액션이 차지하는 비율을 변화시키었다. 밀착 액션의 특성에서 수행시간은 일치성 액션의 수행시간 보다 긴 지수 분포를 하므로 밀착 액션의 구성 비율이 증가할수록 평균 반환시간은 증가한다.

기존의 기법과 제안한 기법간에 밀착액션의 구성비율이 증가할수록 평균 반환시간의 차이가 커지는 것을 알 수 있는데 이는 제안한 기법이 기존의 기법보다 페이지 접근시 록 해제를 기다리는 대기시간이 적기 때문에 발생한다.

5.2.3 시스템의 부하를 변화시킨 경우

중앙처리기에 부과되는 부하가 높아짐에 따라 프로그램의 수행시간은 길어진다. 분산 시스템의 부하가 증가할 때 두 기법간에 시스템의 성능을 대비하여 보기 위하여 액션 종류의 비율을 일치성 액션을 60%로 고정시키고 이중 밀착 액션은 총 프로그램 중 1/2만이 포함하며, 액션의 구성중 마지막에 단지 한 개의 밀착 액션만 갖는다. 프로세스 액션은 40%로 고정시켰다. 프로그램의 도착 시간 간격을 변화시켜 시스템의 부하를 변화시켰을 때 두 기법간의 프로그램 반환시간을 비교 분석한 결과를 그림 10에 나타내었다.

분석결과 부하가 증가 함에 따라 기존의 방법이 제시한 방법보다에 더 민감하게 증가하는 경향을 알 수 있었고, 두 기법 간에 프로그램의 반환시간의 차이가 부하가 증가함에 따라 커지는 것을 알 수 있었다.

Ⅵ. 결 론

본 연구에서는 분산 시스템에서 데이타의 일치성을 유지시키면서 프로세스간의 동시수행 능력을 향상시켜 자원을 효율적으로 활용함으로써 분산 시스템의 성능을 향상시키는 기법을 제시하였다. 분산 시스템의 데이타 영역을 고장이 발생할 경우에도 항상 일치성을 유지시켜야 하는 atomic 데이타 영역과 프로세스 처리의 형태와 같이 정상적으로 수행할 경우에만 일치성을 유지하는 non-atomic 데이타영역으로 구성하였다.

프로그램 내에서 수행되는 액션의 구조를 수행 특성에 따라 직렬 액션, 밀착 액션, 프로세스 액션으로 나누고 각 액션의 종류별로 로킹(locking) 방법에 의한 동시성 제어 기법을 제시하였다. 또한 각 데이타 영역에 대한 접근 규칙을 정의함으로써 프로그램의 수행중에 고장이 발생할 경우에 atomic 데이타 영역에 대해서는 항상 일치성이 보장되도록 하였다.

2단계 로킹에 기초를 둔 기존의 트랜잭션 기법과 제

www.dbpia.co.kr

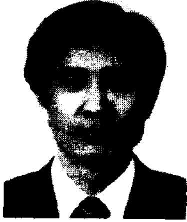
시한 기법을 시뮬레이션을 통해 비교함으로써 시스템의 성능을 평가하였다. 시뮬레이션 결과 프로세스 액션의 비율이 증가함에 따라 제안한 기법의 평균 반환시간은 기존의 기법보다 감소하는 추세가 커짐을 알 수 있었으며, 프로세스 액션의 비율이 40%일때 제안한 기법이 11%정도의 향상을 가져왔다. 이는 버전 관리 및 록의 대기 시간이 감소하기 때문임을 알 수 있었다.

프로세스 액션 및 밀착 액션의 비율이 증가함에 따른 비교 결과, 평균 반환시간(average turnaround time)이 감소하는 경향은 프로세스 액션만이 존재할 경우보다 커짐을 보였다. 프로그램내에 직렬 액션과 프로세스 액션의 비율이 각각 60%와 40%이고, 밀착 액션을 포함하는 비율이 50%이며, 분산 시스템의 부하가 45%일때 평균 반환시간은 제시한 기법이 기존의 기법보다 20%정도 감소함을 보였다. 이는 밀착 액션에서도 록의 대기 시간이 감소하였기 때문이다. 또한 시스템 부하의 증가에 따른 분석 결과 부하가 증가할수록 평균 반환 시간의 증가 추세는 기존의 기법보다 제시한 기법이 낮아짐을 알 수 있었다.

참고문헌

1. J.E.B.Moss, "Nested Transactions:an approach to reliable distributed computing," Ph.D. Thesis 260, Massachusetts Institute of Technology, Cambridge, MA, Apr., 1981.
2. M.H.Nodine and S.B.Zdonik, "Coorperative Transaction Hierachies: A Transaction Model to Support Design Application," In Proc. of 16th VLDB conference, pp.83-94, 1990.
3. A.Skarra, "Localized correctness specifications for cooporating transactions in an object-oriented database," SIGPLAN Notices, Vol. 24, No. 4, Apr., 1989.
4. B.Walker,G.Popek,R.English,C.Kline and G.Thiel, "The LOCUS Distributed Operating System," In Proc. of the 9th ACM Symposium on Operating systems principles, Bretton Woods, New Hampshire, pp.49-70, 1983.
5. A.Goscinski, Distributed Operating Systems - The Logical Design -, Addison-Wesley Pub. Co., 1991.
6. B.Liskov and R.Scheifler,, "Guardians and Actions: Linguistic Suport for Robust,Distributed Programs," ACM Transactions on Programming Languages and systems, 5(3), pp.381-404, Jul., 1983.
7. B. Liskov, M. Day, M. Herlihy ,P. Johnson, G. Leavens, R. Scheifler, and W.Weihl, Argus Reference Manual, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Mar., 1987.
8. M.P.Herlihy and J.M.Wing, "Avalon:Language support for reliable distributed systems," In Proc. of the 17th international symposium on fault-tolerent computing, Jul., 1987.
9. R.C.Chen and P.Dasgupta, "Implementing Consistency control Mechanism in the Cloudes Distributed Operating System," In Proc. of the 11th International conference on Distributed computing systems, pp.10-17, 1991.
10. G.Popek and B.J.Walker, *The LOCUS Distributed System Architecture*, the MIT Press, 1985.
11. C.Pu,G.Kaiser and N.Huchinson, "Split-Transactions for Open-Ended Activities," In Proc. of 14th VLDB Conf., Los Angeles, pp.26-37, Sep., 1988.
12. B.CARTIEL, "A model of concurrency control in nested transactions system," JACM, Vol. 36, No. 2, pp.230-269, Apr., 1989.
13. U.Ramachandran,M.Yousef and A.Khalidi, "An Implementation of Distributed Shared Memory," In Proc. of the USENIX Conference, pp.21-38, Oct., 1989.
14. K.Li and P.Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM Trans. Computing System, Vol. 7, No. 4, pp.321-359, Nov., 1989.

www.dbpia.co.kr



李 栽 完(Jae Wan Lee) 정회원

1957년 9월 4일생
1984년 : 중앙대학교 전자계산학과 졸업(공학사)
1987년 : 중앙대학교 대학원 전자계산학과 졸업(공학석사)
1992년 : 중앙대학교 대학원 전자계산학과 졸업(공학박사)
1992년 9월~현재 : 군산대학교 정보통신공학과 조교수
※주관심 분야 : 분산시스템, 운영체제, 컴퓨터 네트워크

周 洙 鍾(Su Jong Joo) 정회원

1957년 9월 3일생
1986년 : 원광대학교 전자계산학과 졸업(공학사)
1987년 : 중앙대학교 대학원 전자계산학과 졸업(공학석사)
1992년 : 중앙대학교 대학원 전자계산학과 졸업(공학박사)
1993년 7월~1994년 8월 : 미국 University of Massachusetts at Amherst, 전기 및 컴퓨터공학과에서 박사후과정
1990년 9월~현재 : 원광대학교 컴퓨터공학과 조교수
※주관심 분야 : 분산시스템, 운영체제, 실시간 시스템, 시스템 최적화

www.dbpia.co.kr