모바일 컴퓨팅 환경을 위한 재수행 트랜잭션 모델의 설계 및 구현

(Design and Implementation of a Reprocessing Transaction Model for Mobile Computing Environments)

김 동 현 † 홍 봉 희 ††
(Dong-Hyun Kim) (Bong-Hee Hong)

요 약 공간 객체를 수정하는 모바일 트랜잭션은 단절 상태에서 지역 데이타를 독립적으로 수정할 수 있는 긴 트랜잭션이다. 모바일 트랜잭션의 수행에 적합하다고 알려진 검증 기반 프로토콜은 데이타 일관성을 유지하기 위하여 충돌된 트랜잭션을 철회시킨다. 그러나 긴 트랜잭션을 철회하면 모든 작업 결과를 취소해야 하기 때문에 철회 방법은 모바일 트랜잭션 수행에 적합하지 않다.

이 논문에서는 충돌된 모바일 트랜잭션을 철회하지 않고 충돌을 해소하기 위하여 재수행 트랜잭션 모델을 제안한다. 그리고 재수행 트랜잭션을 지원하는 트랜잭션 서버를 설계하고 모바일 현장 시스템의 프로토타입을 구현하였다. 재수행 트랜잭션은 외래 충돌 객체를 이용하여 충돌 객체만을 재수정하는 트랜잭션으로 완료를 요청한 트랜잭션의 충돌에 의해 시작되는 서브 트랜잭션이다. 그리고 재수행 트랜잭션의 기아 문제를 줄이기 위하여 쓰기 집합의 객체들 중 비충돌 객체는 다른 트랜잭션에게 노출시키는 점진적 재수행 기법을 제시한다.

키워드 : 모바일 GIS, 모바일 트랜잭션, 재수행, 변경 충돌 해소

Abstract Mobile transactions updating spatial objects are long transactions that can update independently local objects of mobile clients during disconnection. Validation based protocols that are well known to be appropriate for mobile transactions are required to abort a conflicted transaction in order to keep data consistent. Since abortion leads to cancel of all of the updates, it is not desirable to use the abortion for resolving conflicts of mobile transactions.

In this paper, we propose a reprocessing transaction model to resolve the update conflicts between mobile transactions without aborting them. We also design a mobile transaction server to support the reprocessing transaction and build a prototype of a mobile field system. The reprocessing transaction is a subtransaction of a newly committed mobile transaction and re-executes only conflicted objects with foreign conflicted objects. We also introduce a progressive reprocessing scheme to expose non-conflicted objects of the mobile transaction to other transactions in order to reduce starvation of reprocessing transactions.

Key words : mobile GIS, mobile transaction, reprocessing, update conflict resolution

1. 서 론

무선 모바일 클라이언트 기술이 발전함에 따라 GPS 등의 측량 장치를 통하여 현장에서 공간 데이타를 바로 수정할 수 있는 모바일 현장 시스템(mobile field system)이 가능하게 되었다. 시설물(상하수도, 가스) 관리업자, 현장 조사원 등의 사용자는 그림 1과 같이 모바일 현장 시스템을 이용하여 서버로부터 다운로드 받은 공간 데이타를 현장의 모바일 클라이언트에서 수정한다. 그리고 무선 네트워크를 통하여 수정된 공간 데이타를 바로 서버에 반영함으로써 다른 사용자들이 항상 현장의 최신 데이타를 공유할 수 있도록 한다.

무선 네트워크는 유선 네트워크에 비하여 불안정하고

† 학생회원 : 부산대학교 컴퓨터공학과
pusrover@pusan.ac.kr
†† 종신회원 : 부산대학교 컴퓨터 및 정보통신 연구소
bhhong@pusan.ac.kr
논문접수 : 2002년 8월 26일
심사완료 : 2002년 12월 28일

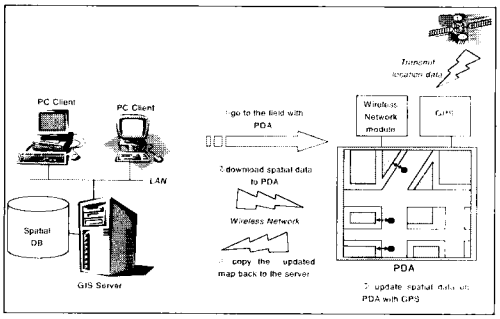


그림 1 모바일 클라이언트를 이용한 지도 수정의 예

자주 단절(disconnect)되는 특성을 가지고 있기 때문에 모바일 트랜잭션은 서버와 항상 연결되어 있는 강 연결 상태(strongly connected state)가 아니다. 모바일 트랜잭션은 대부분의 수정 기간 동안 서버와 단절되어 있고 완료 작업 수행 등의 필요한 경우에만 서버와 연결하는 약 연결 상태(weakly connected state)이다. 따라서 공간 데이타 수정을 위한 모바일 트랜잭션은 일반적인 트랜잭션과는 달리 단절 상태(disconnected state)에서도 모바일 클라이언트의 지역 데이타를 수정할 수 있는 긴 트랜잭션(long transaction)이다[1].

특정 지역의 공간 객체들을 수정하는 작업은 논리적 전역 트랜잭션(global transaction)이며 모바일 트랜잭션은 이 전역 트랜잭션의 서브 트랜잭션(subtransaction)이다. 전역 트랜잭션은 각 공간 객체들을 수정하는 모든 모바일 트랜잭션들의 집합으로 정의된다. 비록 현장 수정 작업은 전역 트랜잭션의 관리를 받지만 각 모바일 트랜잭션의 시작, 끝 그리고 수정 영역 등은 정해진 범위 안에서 작업자가 자체적으로 결정한다. 따라서 다수의 모바일 트랜잭션이 중첩된 지역의 공간 객체를 동시에 수정할 수도 있다. 그림 2는 특정 지역에 위치하고 있는

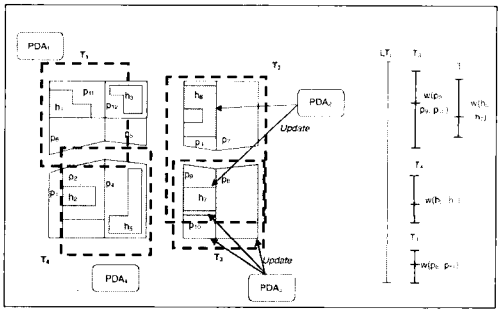


그림 2 동일 지역을 동시 수정하고 있는 모바일 트랜잭션의 예

지번과 건물 객체들의 좌표 데이타를 네 개의 모바일 트랜잭션이 수정하고 있는 예를 보여준다. 그림 2에서 보듯이 모바일 트랜잭션 T_2와 T_3는 중첩된 지역의 공간 객체인 p_9과 h_7을 동시에 수정한다. 따라서 p_9과 h_7간에 비정상적인 공간 관련성이 생성될 수 있기 때문에 T_2와 T_3는 서로 충돌한다.

모바일 트랜잭션의 충돌을 해소하기 위한 기법은 크게 세가지로 분류할 수 있다. 첫 번째는 검증 기법에 기반한 낙관적 접근 방법이다[2,3,4,5]. 이 방법은 두 트랜잭션이 충돌하면 나중에 완료를 요청한 트랜잭션을 철회함으로써 트랜잭션간의 충돌을 해소한다. 그러나 긴 트랜잭션이 철회되면 오랜 시간동안 작업한 모든 수정 결과가 취소되는 문제가 있다. 두 번째는 잠금을 이용한 비관적 접근 방법으로 나중에 잠금을 요청한 트랜잭션은 요청한 잠금이 설정될 때까지 대기한다[1]. 그러나 잠금을 설정한 트랜잭션이 긴 트랜잭션이면 오랜 시간동안 대기해야 하는 문제가 있다. 세 번째는 협동 작업 기법을 이용한 방법이다[6]. 모바일 클라이언트는 서버에 연결할 때마다 단절 기간동안의 히스토리를 서버의 히스토리와 병합해서 변경 충돌을 검사한다. 그러나 이 방법은 서버에 연결할 때마다 히스토리들을 병합해야 하는 문제가 있다.

일반적으로 모바일 트랜잭션을 처리하기 위한 기법으로 검증 기반 프로토콜이 적합하다. 그러나 단절 상태에서 공간 데이타를 수정하는 모바일 트랜잭션간의 충돌을 해소하기 위하여 단순히 철회 방법 또는 대기 방법을 적용하기 어려운 것은 분명하다. 이 논문에서는 트랜잭션간의 충돌을 해소하기 위하여 재수행 트랜잭션 모델(reprocessing transaction model)을 제안한다. 그리고 재수행 트랜잭션을 지원하는 트랜잭션 서버를 설계하고 모바일 현장 시스템의 프로토타입을 구현하였다. 이 논문의 기본 아이디어는 긴 시간동안 수행된 트랜잭션을 철회하지 않고 충돌된 객체에 대한 연산만을 재수행하여 충돌을 해소하는 것이다. 트랜잭션 서버는 완료를 요청한 트랜잭션의 쓰기 집합과 이미 완료된 다른 트랜잭션의 쓰기 집합들을 비교하여 변경 충돌 검사를 수행한다. 만약 충돌 검사가 실패하면 충돌 객체(conflicted object)들을 검색하고 재수행 트랜잭션을 시작한다. 재수행 트랜잭션은 외래 충돌 객체를 이용하여 충돌 객체만을 재수정하는 트랜잭션으로 완료를 요청한 트랜잭션의 서브 트랜잭션이다. 외래 충돌 객체(foreign conflicted object)는 이미 완료된 트랜잭션의 쓰기 집합에 속하는 객체로 트랜잭션의 충돌을 유발하는 객체이다.

그러나 재수행 트랜잭션은 모든 충돌 객체의 충돌이 해

소될 때까지 연속적으로 수행되기 때문에 최악의 경우에 트랜잭션이 계속 완료되지 않는 트랜잭션 기아(starvation) 문제가 발생한다. 기아 문제를 줄이기 위하여 이 논문에서 점진적 재수행 기법(progressive reprocessing scheme)을 제시한다. 점진적 재수행 기법은 충돌로 인하여 k+1번째의 재수행 트랜잭션이 시작되기 전에 쓰기 집합의 객체들 중 충돌되지 않는 비충돌 객체는 서버에 저장하고 다른 트랜잭션에게 노출시킨다. 그리고 k번째 트랜잭션의 충돌 객체만을 재수행한다.

충돌된 트랜잭션을 철회하지 않고 재수행 트랜잭션 기법을 사용하는 이유는 다음과 같다. 첫째, 모바일 트랜잭션들은 전역 트랜잭션에 의하여 조정되기 때문에 모바일 트랜잭션들의 쓰기 집합간에 서로 중첩될 가능성은 매우 낮다. 만약 중첩이 되어서 충돌하더라도 전체 쓰기 집합에서 충돌 객체들의 수는 매우 적다. 둘째, 공간 데이타를 수정하는 대부분의 작업들은 사용자 상호 연산에 의해 수행된다. 만약 적은 수의 충돌 객체로 인하여 모든 수정 내용이 취소되면 작업을 완료하기 위하여 모든 객체에 대한 긴 수정 작업을 반복해야 하는 고비용이 요구된다.

이 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 기술하고 3장에서 검증 기반 프로토콜을 모바일 트랜잭션에 적용할 때 문제점을 정의한다. 4장에서 직접 충돌 및 간접 충돌을 검사하는 확장 검증 조건을 제시하고 5장에서 충돌 해소를 위한 재수행 트랜잭션을 기술한다. 그리고 6장에서 점진적 재수행 기법을 제시하고 7장에서 재수행 트랜잭션이 보장하는 일관성 단계에 대하여 기술한다. 8장에서는 재수행 트랜잭션을 지원하는 모바일 현장 시스템의 설계 및 구현 결과를 기술하고 마지막으로 9장에서 결론 및 향후 연구를 기술한다.

2. 관련 연구

변경 충돌 해소를 위한 첫 번째 방법은 낙관적 접근 방법이다. Mobisnap[2]은 트랜잭션을 정의할 때 충돌 검사를 위하여 사용자 의도를 기술한 전조건(precondition)과 후조건(post condition)을 지원한다. 모든 조건을 만족하지 못할 경우 트랜잭션은 철회된다. [4]에서는 다중 버전 기법을 사용하였다. 단절 기간동안 모바일 트랜잭션은 지역 저장 장치의 스냅샷(snapshot)에 대하여 수정 작업을 수행한다. 그리고 완료를 요청할 때 서버는 조정 작업(reconciliation process)을 수행한다. 조정 작업은 충돌 검사, 충돌 해소 그리고 직렬화로 구성된 통합 과정이다. 만약 조정 작업이 실패하면 충돌된 트랜잭션을 철회하고 쓰기 집합의 결과를 취소한다.

2단계 트랜잭션 모델(two-tier transaction model)은 [5]에서 제안되었다. 모바일 노드가 기본 노드에 접속할 때 기본 트랜잭션(base transaction)이 전송된 임시 트랜잭션(tentative transaction)의 모든 연산을 재수행한다. 만약 기본 트랜잭션의 결과가 수용 요건(acceptance criteria)을 만족하지 못하면 관련된 트랜잭션은 철회된다. 수용 요건은 트랜잭션의 결과가 만족해야 하는 조건들이다. 모든 연산을 재수행하는 부하를 줄이기 위하여 히스토리 병합 기법(history merge scheme)이 [3]에서 제안되었다. 서버는 기본 히스토리와 임시 히스토리를 이용하여 선행 그래프(precedence graph)를 생성한다. 만약 선행 그래프에서 사이클이 인식되면 충돌을 해소하기 위하여 사이클을 풀 수 있는 트랜잭션의 히스토리를 선택한 후에 해당 트랜잭션을 철회한다.

그러나 낙관적 접근 방법을 사용한 대부분의 기법은 충돌을 해소하기 위하여 충돌된 트랜잭션을 철회하는 방법을 사용하는 문제가 있다. 만약 이러한 기법들을 공간 객체를 수정하는 모바일 트랜잭션에 적용하면 충돌된 트랜잭션은 철회되고 긴 시간동안 작업한 모든 결과는 취소되어야만 한다. 그러나 모든 작업 내용의 취소는 분명히 모바일 트랜잭션에 적합하지 않다.

두 번째 방법은 비관적 접근 방법이다. 잠금을 사용한 모바일 트랜잭션 모델이 [1]에서 제안되었다. [1]에서는 객체를 수정하기 위하여 전읽기(preread)와 전쓰기(prewrite) 잠금을 사용하였다. 만약 요청한 잠금이 객체에 이미 설정되어 있는 잠금과 충돌하면 모바일 트랜잭션은 잠금을 획득할 때까지 대기한다. 전읽기 잠금과 전쓰기 잠금은 기존의 읽기, 쓰기 잠금과 충돌하지 않기 때문에 완료 연산 수행중의 데이타 가용성을 향상했다. 그러나 수정 작업 중에 사용하는 전쓰기 연산과 전읽기 연산은 서로 충돌하기 때문에 오랜 시간동안 대기해야 하는 문제가 있다.

충돌 해소를 위한 세 번째 방법은 협동 작업 기법을 사용하는 것이다. CoAct모델[6]에서 모바일 클라이언트의 히스토리와 서버의 히스토리를 병합하기 위한 알고리즘이 제안되었다. 충돌을 감지하기 위하여 서버는 병합된 히스토리를 역으로 스캔하면서 충돌하는 연산들을 검색하고 대체 병합 히스토리(alternative merged history)들을 생성한다. 대체 병합 히스토리는 충돌하는 두 개의 연산 중 하나의 연산만을 포함한 히스토리이다. 서버는 선택된 히스토리의 연산만을 수행하기 때문에 충돌은 해소된다. 그러나 충돌을 해소하기 위하여 이미 완료된 트랜잭션들의 결과들을 동의 없이 취소할 수 있기 때문에 트랜잭션의 원자성과 지속성을 보장할 수 없는 문제점이 있다.

3. 문제 정의

모바일 트랜잭션은 대부분의 수정 기간동안 서버와 단절되는 특성을 가진다[4]. 따라서 단절되기 전에 서버 데이타베이스의 일부를 지역 저장 장치에 캐싱한다. 그리고 단절 기간동안 캐싱된 데이타에 대하여 독립적으로 수정작업을 수행한다. 모바일 트랜잭션 T_i의 캐싱 영역을 CachedRegion(T_i)라 하자. T_i의 쓰기 집합과 읽기 집합은 다음과 같이 정의된다.

정의 1 : 공간 객체 o_i의 기하 영역(spatial extent)을 $o_i.G$라 하자. T_i의 ***읽기 집합 RS(T_i)***는 CachedRegion (T_i)에 포함되고 읽기 연산 r의 결과인 객체들의 집합이다. 그리고 ***쓰기 집합 WS(T_i)***는 CachedRegion(T_i)에 포함되고 쓰기 연산 w의 결과인 객체들의 집합이다.

- $RS(T_i) = \{o_{ik} \mid CachedRegion(T_i).G \cap o_{ik}.G \neq \emptyset \wedge r(T_i) \rightarrow o_{ik}\}$ where $1 \leq k \leq n$.
- $WS(T_i) = \{o_{ik} \mid CachedRegion(T_i).G \cap o_{ik}.G \neq \emptyset \wedge w(T_i) \rightarrow o_{ik}\}$ where $1 \leq k \leq n$.

MBR을 T_i가 캐싱한 객체 oik의 집합에 대한 최소 경계 사각형이라 할 때 T_i의 캐싱 영역 CachedRegion (T_i)는 MBR(RS(T_i))이다.

기존의 검증 기반 프로토콜[3,4,5]은 검증 단계에서 완료를 요청한 T_i와 이미 성공적으로 완료된 트랜잭션간에 충돌 여부를 검사한다. 만약에 start(T_i) < commit(T_j) < commit(T_i)인 트랜잭션 T_j에 대하여 $WS(T_j) \cap RS(T_i) \neq \emptyset$ 또는 $WS(T_j) \cap WS(T_i) \neq \emptyset$이면 T_i는 T_j와 충돌한다. 만약에 T_i가 T_j와 충돌하면 데이타 일관성을 보장하기 위하여 T_i는 철회되고 $WS(T_i)$의 모든 값들은 취소된다.

그러나 기존의 검증 기반 프로토콜을 이용하여 공간 객체 수정을 위한 모바일 트랜잭션을 수행하면 다음과 같은 문제점이 있다. 첫째는 충돌된 트랜잭션이 철회되면 수정 작업을 완료하기 위한 비용이 매우 크다. 공간 객체 수정 작업은 사용자 상호 작업이기 때문에 T_i는 긴 트랜잭션이다. 만약 T_i의 충돌 검사가 실패하면 모든 T_i의 결과 값들은 취소된다. 따라서 수정하고자 하는 공간 객체들에 대한 작업을 완료하기 위해서는 T_i를 재시작하고 모든 객체에 대한 긴 수정 작업을 반복해야 한다. 그러므로 철회 방법은 공간 객체를 수정하는 모바일 트랜잭션의 충돌을 해소하기 위한 방법으로 적합하지 않다.

둘째는 검증 조건의 하나인 $WS(T_j) \cap RS(T_i) \neq \emptyset$은 매우 엄격한 조건이기 때문에 빈번한 트랜잭션 철회를 유발할 수 있다. 공간 객체 수정 작업은 사용자에 의해 수행되기 때문에 T_i는 수정하고자 하는 객체들의 주변 객체들도 읽어야 한다. 즉, T_i의 읽기 집합인 $RS(T_i)$의 대부분 객체는 단순히 화면에 디스플레이하기 위하여 읽히는 반면에 소수의 객체만이 수정된다. 따라서 읽기 집합의 크기가 쓰기 집합에 비하여 매우 크기 때문에 빈번한 철회가 발생한다. 특히 CachedRegion(T_i).G $\cap$ CachedRegion(T_j). $G \neq \emptyset$ 이면 T_i는 철회될 가능성이 매우 높아진다.

두 모바일 트랜잭션 T_1과 T_2가 각각의 모바일 클라이언트에서 지번 데이타를 수정하고 있다고 가정해보자. 그림 3에서 보듯이 지번 객체를 수정하기 위하여 T_1은 10개의 객체를 읽고 T_2는 p_{15}를 읽어서 화면에 디스플레이한다. 그림 4는 T_1과 T_2가 공간 객체를 수정하는 과정을 보여준다. 그림 4에서 보듯이 start(T_1) < start(T_2) < commit(T_2) < commit(T_1)의 순이기 때문에 T_1은 T_2에 대하여 충돌 검사를 수행한다. 그러나 $RS(T_1) \cap WS(T_2) = p_{15}$이기 때문에 T_1은 강제적으로 철회되고 오랜 시간동안 작업한 p_2, p_{10}, p_{17} 그리고 p_{15}의 수정 값도 취소된다. 그러나 그림 4에서 보듯이 T_1과 T_2는 완전히 다른 공간 객체들을 수정하였기 때문에 실제 변경 충돌이 발생하지 않는다. 만약 T_1이 완료되어 쓰기 집합이 서버에 병합되더라도 서버의 데이타 일관성은 유지되기 때문에 T_1이 취소될 필요는 없다.

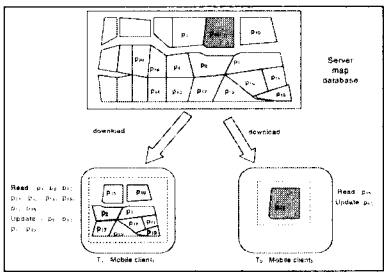


그림 3 지번을 수정하는 두 모바일 트랜잭션의 예

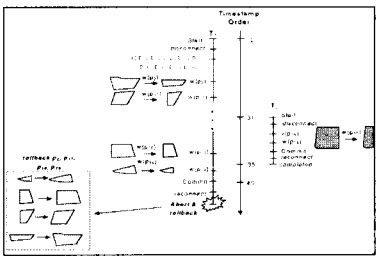


그림 4 지번 객체를 동시 수정하고 있는 T_1과 T_2의 쓰기 집합의 예

또 한가지 고려해야 할 점은 기하 영역이 겹치는 서로 다른 공간 객체간의 일관성 유지 문제이다. 두 공간 객체 o_m과 o_n이 분리(disjoint) 관계를 제외한 공간 관련성을 가지고 있을 때 o_m의 기하 영역과 o_n의 기하 영역을 동시 수정하면 비정상적인 공간 관련성이 생성될 수 있다. 즉, $o_m \in WS(T_i)$이고 $on \in WS(T_j)$일 때 $o_m \neq o_n$이고 $o_m.G \cap on.G \neq \emptyset$이면 o_m과 o_n의 간접 충돌 여부를 검사해야 한다. 만약 간접 충돌이 검사되지 않으면 데이타 일관성을 보장할 수 없다. 그러나 기존 프로토콜[3,4,5]의 검증 조건은 간접 충돌을 검사할 수 없다.

예를 들어 다수의 지번 객체를 수정하는 T_1과 건물 객체를 수정하는 T_2를 가정해보자. 그림 5와 같이 T_1은 p_1, p_4, p_8 그리고 p_{11}을 수정하며 T_2는 h_5를 수정하였다. $WS(T_2)$은 $RS(T_1)$은 물론 $WS(T_1)$과도 전혀 교차하지 않기 때문에 T_1은 성공적으로 완료된다. 그러나 p_4의 수정 결과를 그대로 병합하면 그림 6에서 보듯이 h_5와 p_4 간에 비정상적인 공간 관련성이 생성된다. 따라서 그림 6과 같이 동시 수정한 공간 객체들간에 간접 충돌하면 재수정하여 충돌을 해소할 필요가 있다.

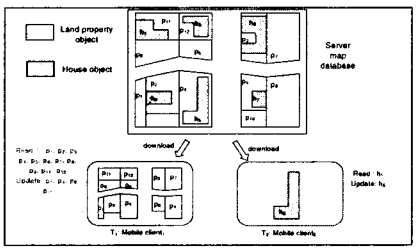


그림 5 다수의 지번 객체를 수정하는 T_1과 건물 객체를 수정하는 T_2의 예

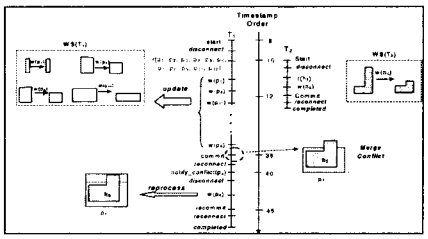


그림 6 간접 충돌된 지번 객체를 재수정하는 T_1의 예

그림 6의 p_4와 h_5간의 변경 충돌을 검사하기 위하여 공간 객체를 수정하는 모바일 트랜잭션간의 간접 충돌을 다음과 같이 정의한다.

정의 2 : $start(T_i) < commit(T_i) < commit(T_j)$인 두 개의 쓰기 집합 $WS(T_i)$ 와 $WS(T_j)$가 있다고 하자. $o_m \in WS(Ti)$이고 $o_n \in WS(Tj)$인 두 객체 o_m과 o_n이 있을 때 $o_m \neq o_n$ 이고 $o_m.G \cap o_n.G \neq \emptyset$ 이면 o_m과 o_n은 ***간접 충돌(indirectly conflict)***한다.

이 논문은 다음의 두 가지 이유 때문에 기존의 검증 기반 기법을 확장한다. 첫째, 충돌된 긴 트랜잭션을 철회하면 해당 작업을 완료하기 위하여 고비용이 요구된다. 또한 엄격한 조건으로 인해 빈번한 트랜잭션 철회가 발생할 수 있다. 따라서 철회 방법이 아닌 확장된 방법이 필요하다. 둘째, 기존의 검증 조건은 공간 객체간의 간접 충돌을 검사할 수 없기 때문에 공간 객체간의 일관성을 유지하기 위하여 검증 조건을 확장해야 한다. 다음 장에서 확장 검증 조건을 기술한다.

4. 확장 검증 조건

이 장에서는 모바일 트랜잭션간의 변경 충돌을 검사할 때 공간 객체간의 직접 충돌과 간접 충돌을 검사하기 위한 확장 검증 조건과 충돌 객체에 대하여 기술한다.

4.1 검증 리스트

모바일 트랜잭션 T_i의 충돌 검사 대상인 트랜잭션들을 결정할 때 단절된 트랜잭션의 쓰기 집합은 알 수 없기 때문에 후위 지향 검증 기법(backward oriented validation scheme)을 사용한다. 이 기법 하에서 T_i는 동시 수행한 트랜잭션 중 성공적으로 완료된 트랜잭션 T_j와의 충돌을 검사한다. T_j를 T_i의 인터리빙 트랜잭션이라 하고 정의 3과 같이 정의한다. 그리고 T_i의 모든 인터리빙 트랜잭션들의 리스트를 검증 리스트(validation list)라 하고 validation_list(T_i)로 표기한다.

정의 3 : 두 모바일 트랜잭션 T_i와 T_j가 있을 때 $start(T_i) < commit(T_j) < commit(T_i)$이면 T_j는 T_i의 ***인터리빙 트랜잭션(interleaving transaction)***이다.

비록 T_j가 T_i의 인터리빙 트랜잭션이라 해도 T_i의 캐슁 영역인 CachedRegion(T_i)가 CachedRegion(T_j)와 교차하지 않는다면 T_i와 T_j는 전혀 다른 지역의 공간 데이타를 수정하기 때문에 T_i와 T_j는 서로 충돌하지 않는다. 따라서 이러한 경우에 T_i는 T_j에 대하여 충돌 검사를 수행할 필요가 없다. 그러나 만약 T_j가 T_i의 인터리빙 트랜잭션이고 또한 캐슁 영역이 교차하면 T_i는 T_j와 충돌할 수 있다. 캐슁 영역을 이용하여 T_i의 중첩 트랜잭션을 다음과 같이 정의한다.

정의 4 : 두 모바일 트랜잭션 T_i와 T_j가 있을 때 CachedRegion(T_i).G ∩ CachedRegion(T_j).G $\neq \emptyset$ 이면 T_j는 T_i의 중첩 트랜잭션(overlapped transaction)이다.

정의 5 : 두 모바일 트랜잭션 T_i와 T_j가 있을 때 T_j

가 T_i의 인터리빙 트랜잭션이고 동시에 중첩 트랜잭션이면 T_i는 T_j와 ***충돌 가능(potentially conflict)***하다.

정의 4와 5를 이용하면 T_i의 충돌 검사를 위한 트랜잭션의 수를 줄여서 효과적으로 충돌 검사를 수행할 수 있다. 공간 데이타를 수정하는 모바일 트랜잭션 T_i의 검증 리스트는 다음과 같이 재정의한다 : validation_list(T_i) = {T_j | start(T_i) < commit(T_j) < commit(T_i) Cached Region(T_i).G ∩ CachedRegion(T_j).G ≠ ∅}.

4.2 충돌 객체

그림 6의 예에서 보았듯이 트랜잭션들의 쓰기 집합들이 서로 다른 집합이더라도 독립적인 동시 수정은 객체간의 올바른 공간 관련성을 보장할 수 없다. 따라서 공간 객체간의 간접 충돌을 검사하기 위하여 다음과 같이 공간 관련성을 위한 검증 조건을 포함한 확장 검증 조건을 정의한다.

정의 6 : 공간 데이타를 수정하는 임의의 모바일 트랜잭션 T_i와 T_j가 있을 때 T_i가 다음의 두 조건 중 하나를 만족하면 T_i의 ***확장 검증 조건***은 실패한다.

1. 직접 충돌 조건 : (start(T_i) < commit(T_j) < commit(T_i)) ∧ (WS(T_i) ∩ WS(Tj) ≠ ∅).

2. 간접 충돌 조건 : $o_m \in WS(T_i)$, $o_n \in WS(T_j)$인 공간 객체 o_m과 o_n이 있을 때, (start(T_i) < commit(T_j) < commit(T_i)) ∧ (WS(T_i) ∩ WS(T_j) = ∅) ∧ (o_m.G ∩ o_n.G ≠ ∅).

확장 검증 조건에 따라 T_i의 충돌 객체의 집합은 다음과 같이 정의된다.

정의 7 : 두개의 쓰기 집합 WS(T_i)와 WS(T_j)가 있을 때 T_i의 충돌 객체인 ***conflicted_objects(T_i)***는 WS(T_i)와 WS(T_j)에 교집합에 속하는 공유 객체이거나 또는 WS(T_i)의 객체이면서 동시에 WS(T_j)에 속하는 임의의 객체와 공간적으로 교차하는 객체이다. 단, T_j는 validation_list(T_i)의 트랜잭션이다.

• conflicted_objects(T_i) = {o_m | o_m.oid = ∃o_n.oid ∨ o_m.G ∩ ∃o_n.G ≠ ∅} where $o_m \in WS(T_i)$, $o_n \in WS(T_j)$.

그림 7은 T_i의 충돌 객체를 검색하기 위한 알고리즘을 보여준다. 첫 번째 단계는 객체간의 충돌 검사를 위하여 T_i의 검증 리스트를 만든다. T_j가 T_i의 인터리빙 트랜잭션인 동시에 중첩 트랜잭션이면 검증 리스트에 T_j를 추가한다. 두 번째 단계는 충돌 객체를 검색하는 단계이다. 확장 검증 조건을 이용하여 WS(T_i)의 모든 객체를 WS(T_j)의 모든 객체와 비교한다. 만약 WS(T_i)의 객체인 o_m이 WS(T_j)의 임의의 객체와 직접 충돌하거나 또는 간접 충돌하면 o_m을 T_i의 충돌 객체 집합에 추가한다.

```
oi.Gold : an old extent of oi
oi.Gnew : a new extent of oi
PROCEDUPE extended_validation_test
BEGIN
read start(Ti), commit(Ti), CachedRegion(Ti)
// 1st step: obtain validation_list(Ti)
FOR each Tj
BEGIN
     read commit(Tj), CachedRegion(Tj)
     IF (CachedRegion(Tj).G ∩ CachedRegion(Ti).G ≠ ∅)
          and (start(Ti) < commit(Tj) < commit(Ti))
     THEN validation_list(Ti) ← Tj
END
//2nd step: detect conflicted_objects of Ti
valiadation ← valid
conflicted_objects(Ti) ← ∅
read WS(Ti)
FOR each Tj in validation_list(Ti)
BEGIN
     read WS(Tj)
     FOR each oi in WS(Ti)
     BEGIN
          // compare between all objects of two write sets
          FOR each oj in WS(Tj)
               IF( (oi.oid = oj.oid) or (oi.Gold ∩ oj.Gold ≠ ∅)
               or (oi.Gnew ∩ oj.Gnew ≠ ∅)or( oi.Gnew ∩ oj.Gold ≠ ∅ )
               THEN exit inner FOR loop
          validation ← conflcit
          conflicted_objects(Ti) ← oi
     END
END
store conflicted_objects(Ti)
return validation
END
```

그림 7 트랜잭션 T_i의 충돌 객체를 검색하기 위한 알고리즘

5. 재수행 트랜잭션

이 장에서는 충돌 객체를 재수정하기 위한 재수행 트랜잭션에 대하여 기술한다.

5.1 외래 충돌 객체

충돌 객체를 위한 재수행 작업은 사용자 상호 연산에 의해 수행되기 때문에 트랜잭션 T_i의 충돌을 올바르게 해소하기 위해서는 다른 트랜잭션에 의해 동시 수정된 객체들 중 충돌 객체와 관련된 객체를 T_i의 사용자가 볼 수 있어야 한다. 이러한 객체들을 T_i의 외래 충돌 객체라 하고 다음과 같이 정의한다.

정의 8 : start(T_i) < commit(T_j) < commit(T_i)인 conflicted_objects(T_i)와 WS(T_j)가 있을 때 T_i의 ***외래 충돌 객체***인 ***foreign_conflicted_objects(T_i)***는 conflicted_objects(T_i)에 속하는 객체와 간접 충돌하는 WS(T_j)의 객체이다.

• foreign_conflicted_objects(T_i) = {o_{jk} | o_{im}.G ∩ o_{jk}.G ≠ ∅ } where $o_{im} \in$ conflicted_objects(T_i) ∧ $o_{jk} \in WS(T_j)$.

그림 8은 T_1의 충돌 객체와 외래 충돌 객체의 예를 보여준다. T_1과 T_2의 쓰기 집합은 각각 {p_1, p_4, p_8, p_{11}}과 {h_5}이다. T_1은 T_2의 쓰기 집합에 대하여 충돌 검사를 수행하며 그 결과로 p_4는 h_5와 간접 충돌한다. 따라서

T_1의 충돌 객체는 $\{p_4\}$이고 외래 충돌 객체는 $\{h_5\}$이다.

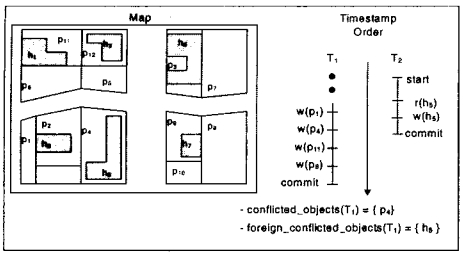


그림 8 T_1의 충돌 객체와 외래 충돌 객체의 예

5.2 재수행 트랜잭션

재수행 작업은 쓰기 연산에 의해 이루어지기 때문에 충돌 해소 작업도 하나의 단위로 완료될 필요가 있다. T_i의 충돌 객체들을 위한 재수행 작업을 재수행 트랜잭션 RT_i라 한다. RT_i는 사용자 상호 연산에 의해 충돌 공간 객체를 수정하며 T_i의 충돌에 의해 시작되는 T_i의 서브 트랜잭션이다.

T_i의 충돌 객체를 사용자가 재수정하기 위하여 RT_i는 T_i의 외래 충돌 객체를 읽어서 디스플레이해야 한다. 따라서 RT_i는 두 종류의 읽기 집합을 가진다. 하나는 T_i의 충돌 객체들의 집합이고 다른 하나는 T_i의 외래 충돌 객체들의 집합이다. RT_i는 충돌 해소를 위한 트랜잭션이기 때문에 일반적인 모바일 트랜잭션과는 달리 충돌 객체만을 재수정할 수 있고 그 외의 공간 객체는 수정할 수 없다. 그러므로 $WS(RT_i)$는 T_i의 충돌 객체 집합인 conflicted_objects(T_i)이다.

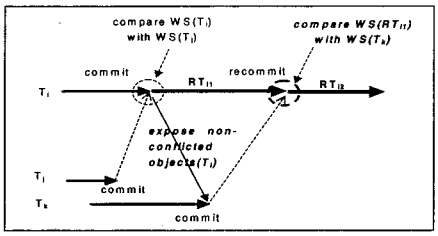


그림 9 재수행 트랜잭션 RT_{il}의 재수행

재수행 트랜잭션은 모든 충돌 객체의 충돌이 해소될 때까지 연속적으로 시작되어 수행된다. 그리고 재완료할 때 재수행 기간동안에 완료된 다른 트랜잭션과 충돌 검사를 수행한다. 만약 변경 충돌이 발생하면 두 번째 재수행 트랜잭션을 시작한다. 트랜잭션 T_i의 k번째 재수행 트랜잭션을 RT_{ik}라 한다. 예를 들어 그림 9에서 보여주듯이 RT_{il}의 수행 중에 새로운 트랜잭션 T_k가 완료되었다고 가정해보자. 만약 RT_{il}이 T_k와 충돌하면 RT_{il}은 T_k와의 충돌 객체들을 재수행하기 위해 두 번째 재수행 트랜잭션인 RT_{i2}를 시작한다. RT_{il}의 충돌 객체는 T_i의 충돌 객체 집합과 T_k의 쓰기 집합간의 검사를 통하여 결정된다. 그리고 RT_{il}의 외래 충돌 객체들은 RT_{il}과 간접 충돌하는 T_k의 쓰기 집합의 부분 집합이다.

재수행 트랜잭션의 시작은 상위 트랜잭션의 충돌 여부에 의존적이다. 만약 k번째 재수행 트랜잭션이 성공적으로 완료되면 k+1번째 재수행 트랜잭션은 시작되지 않는다. 이러한 측면에서 재수행 트랜잭션은 보상 트랜잭션(compensating transaction)의 시작과 유사하다. 그러나 재수행 트랜잭션은 다음과 같이 분명히 틀리다. 첫째, 재수행 트랜잭션은 충돌을 해소해서 모바일 트랜잭션을 성공적으로 완료하기 위한 서브 트랜잭션이다. 그러나 보상 트랜잭션은 상위 트랜잭션이 철회하는 경우에 데이타베이스의 상태를 수행 이전 상태로 만들기 위한 서브 트랜잭션이다. 둘째, 재수행 트랜잭션은 쓰기 집합의 객체들 중 충돌 객체만을 재수행한다. 그러나 보상 트랜잭션은 쓰기 집합의 모든 객체에 대하여 수행된다.

연속적인 재수행 트랜잭션 수행은 반복적으로 수행될 수도 있다. 그러나 만약 T_i의 재수행 트랜잭션들이 수행 기간동안 완료되는 다른 트랜잭션들과 계속 충돌되는 경우 트랜잭션 기아 문제가 발생할 수 있다. 기아 문제를 줄이기 위하여 이 논문에서 재수행 트랜잭션은 점진적으로 쓰기 집합 객체 중 비충돌 객체들을 다른 트랜잭션들에게 노출한다. 이 문제는 6장에서 자세히 기술한다.

5.3 추정 완료 트랜잭션에 대한 충돌 검사

트랜잭션 T_i의 모든 재수행 트랜잭션이 성공적으로 완료되기 전까지 T_i는 완전히 완료된 것이 아니다. 재수행 트랜잭션이 수행중인 모바일 트랜잭션의 상태를 추정 완료 상태(presumed committed state)라 한다. 추정 완료 상태는 완료 상태와 부분 완료 상태의 중간 단계로 T_i의 모든 충돌 객체의 충돌이 완전히 해소되면 완료될 수 있는 조건부 완료 상태이다.

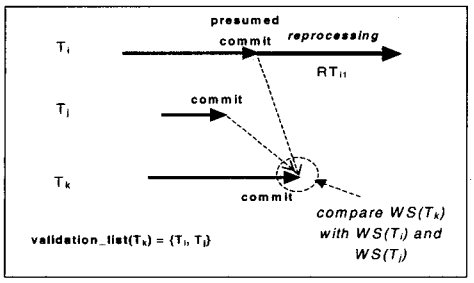


그림 10 새로 완료를 요청한 T_k가 충돌 검사를 수행해야 하는 트랜잭션의 예

만약 새로 완료를 요청한 트랜잭션 T_k가 추정 완료 상태의 T_i와 충돌 가능하면 T_i는 T_k의 검증 리스트에 포함된다. 그리고 T_k는 T_i의 쓰기 집합 객체에 대하여 충돌 검사를 수행한다. 즉, 그림 10의 예에서 보듯이 T_k는 완료된 T_j의 쓰기 집합뿐만 아니라 추정 완료 상태의 T_i의 쓰기 집합에 대해서도 충돌 검사를 수행한다. 그리고 만약 T_k가 재수행 트랜잭션이 현재 작업중인 객체와 충돌하면 T_k의 재수행 트랜잭션을 시작한다.

6. 재수행 트랜잭션의 재완료

이 장에서 재수행 트랜잭션 기아 현상을 줄이기 위한 점진적 재수행 기법에 대하여 기술한다.

6.1 점진적 재수행

k번째 재수행 트랜잭션 RT_{ik}의 수행 기간동안 새로운 모바일 트랜잭션 T_k 또는 다른 재수행 트랜잭션 RT_{jk}가 완료할 수도 있다. RT_{ik}의 재완료를 처리하기 위한 방법은 크게 두 가지가 있다. 첫 번째 방법은 반복적 재수행 방법으로 RT_{ik}가 재완료를 요청할 때 T_i의 최초 충돌 객체들을 이용하여 새로 완료한 모바일 트랜잭션 T_j의 쓰기 집합과 반복적으로 충돌 검사를 수행하는 방법이다. 만약 RT_{ik}가 T_j와 충돌하면 다음 k+1번째 재수행 트랜잭션인 $RT_{i(k+1)}$을 시작한다 그리고 $RT_{i(k+1)}$는 동일한 T_i의 최초 충돌 객체들을 재수행한다.

예를 들어 그림 11에서 보듯이 RT_{11}이 재완료할 때 RT_{11}이 T_1의 최초 충돌 객체들을 이용하여 T_4에 대하여 충돌 검사를 수행한다고 가정해보자. 그림 12에서 보듯이 RT_{11}은 T_1의 충돌 객체들을 재수행하였기 때문에 RT_{11}의 쓰기 집합은 {A, B, C}이다. T_4의 쓰기 집합은 T_1의 충돌 객체 집합 중 하나인 {B}와도 교차하기 때문에 $WS(RT_{11}) \cap WS(T_4) = \{B\}$이다. 따라서 RT_{12}는 T_1의 초기 충돌 객체들인 {A, B, C}를 다시 재수행한다.

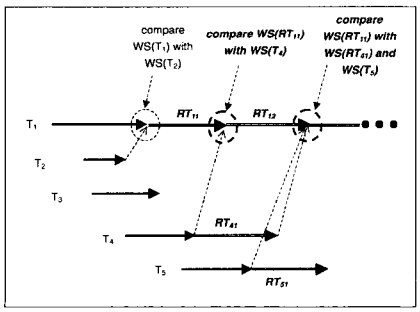


그림 11 반복적 재수행의 예

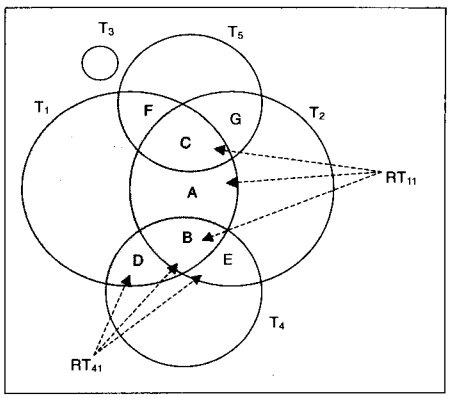


그림 12 T_i와 인터리빙하고 동시에 중첩되는 트랜잭션들의 쓰기 집합들간의 교차 상태의 예

그러나 반복적 재수행 기법은 재수행 트랜잭션 기아 문제가 발생할 수 있다. 연속되는 재수행 기간동안에 동일한 초기 충돌 객체 집합이 계속 k번째 재수행 트랜잭션의 쓰기 집합이 되기 때문에 k-1번째 재수행 트랜잭션의 재완료시 충돌 객체가 아니더라도 k번째 트랜잭션의 재완료시 다시 충돌 객체가 될 수 있다.

재수행 트랜잭션의 기아 문제를 줄이기 위하여 이 논문에서는 두 번째 방법인 점진적 재수행 기법을 사용한다. 점진적 재수행 기법은 재수행 트랜잭션이 완전히 완료되지 않았더라도 쓰기 집합의 객체들 중에 비 충돌 객체들은 완료시켜서 다른 트랜잭션들의 접근을 허용한다. 그리고 다음 번째의 재수행 트랜잭션은 이전 트랜잭션의 충돌 객체만을 재수행한다. RT_{ik}가 새로 완료된 T_j와 충돌하면 $RT_{i(k+1)}$이 RT_{ik}의 충돌 객체들을 재수행하기 위하여 생성된다. RT_{ik}의 충돌 객체들은 $RT_{i(k-1)}$의 충돌 객체들의 부분 집합이며 $RT_{i(k+1)}$은 T_j와 RT_{ik}간의 충돌 객체만을 재수정한다. $RT_{i(k+1)}$을 시작하기 전에 RT_{ik}는 비충돌 객체들을 완료하고 다른 트랜잭션들에게 노출시킨다. 재수행 트랜잭션이 충돌 객체들을 점진적으로 재완료하기 때문에 재수행하기 위한 충돌 객체들은 전 단계의 재수행 트랜잭션에 비하여 점차적으로 줄어들게 된다. 따라서 T_i의 모든 충돌 객체들은 최종적으로 재수행되어 완료될 수 있다.

예를 들어 그림 13에서 보듯이 RT_{11}이 재완료 작업을 끝마칠 때 RT_{11}은 비충돌 객체인 {A, C}를 다른 트랜잭션인 RT_{41}과 T_5에게 노출시킨다. 그리고 RT_{12}는 RT_{11}의 쓰기 집합의 부분 집합이며 또한 $RT_{i0} = T_i$의 충돌 객체들의 일부인 {B}만을 재수행한다. 따라서

RT_{12}의 쓰기 집합은 {B}이며 RT_{12}는 더 이상 T_5와 충돌하지 않는다.

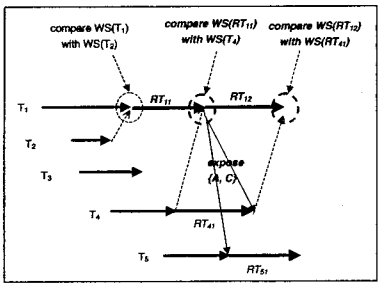


그림 13 점진적 재수행의 예

6.2 수행중인 트랜잭션에 대한 재완료 처리

k번째 재수행 트랜잭션 RT_{ik}가 재완료할 때 현재 재수행중인 트랜잭션에 대하여 변경 충돌 검사를 수행해야 한다. 예를 들어 그림 14에서 보듯이 RT_{11}이 재완료할 때 RT_{41}은 T_4의 충돌 객체를 첫 번째로 재수행하고 있는 중이다. RT_{11}은 RT_{41}이 재수행하고 있는 객체와 충돌하기 때문에 RT_{11}은 $WS(RT_{11}) \cap WS(T_4) = \{B\}$를 재수정하기 위하여 RT_{12}를 시작한다.

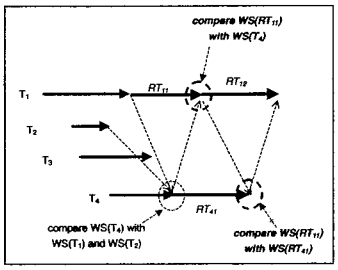


그림 14 재완료시 고려해야 할 경우의 예

재수행 트랜잭션 RT_{ik}가 재완료할 때 RT_{ik}의 검증 리스트에 포함되는 트랜잭션의 종류는 크게 2가지가 있다. 첫 번째는 추정 완료 상태이거나 또는 이미 완료된 모바일 트랜잭션 T_j이다. 비록 T_j가 추정 완료 상태이더라도 RT_{ik}는 자신의 쓰기 집합과 T_j의 쓰기 집합을 비교한다. $start(RT_{ik}) < commit(T_j) < commit(RT_{ik})$인 트랜잭션 RT_{ik}와 T_j가 있을 때, 만약 $WS(T_j) \cap conflicted\_objects(RT_{i(k-1)}) \neq \phi$ 이고 $CachedRegion(T_j).G \cap CachedRegion(RT_{i(k-1)}).G \neq \phi$이면 T_i의 k번째 재수행 트랜잭션 RT_{ik}는 T_j와 충돌한다. RT_{ik}와 T_j가 충돌하면 $RT_{i(k+1)}$을 시작한다.

예를 들어 그림 12와 14에서 보듯이 $WS(T_4)$ n $WS(T_1) = \{B, D\}$ 이고 $WS(T_4) \cap WS(T_2) = \{E\}$이기 때문에 재수행 트랜잭션 RT_{41}이 충돌 객체들 {B, D, E}를 재수행하기 위하여 시작된다. 특히 $WS(T_4) \cap conflicted_objects(T_1) = \{B\}$이다. 따라서 RT_{11}이 충돌 검사를 수행할 때 $WS(T_4) \cap WS(RT_{11}) = \{B\}$이기 때문에 RT_{11}은 T_4와 충돌한다. T_4와의 충돌을 해소하기 위하여 두 번째 재수행 트랜잭션인 RT_{12}가 시작된다.

두 번째는 T_j의 l번째 재수행 트랜잭션인 RT_{jl}이다. $start(RT_{ik}) < commit(RT_{jl}) < commit(RT_{ik})$인 트랜잭션 RT_{ik}와 RT_{jl}이 있을 때, 만약 $WS(RT_{jl}) \cap conflicted_objects(RT_{i(k-1)}) \neq \phi$이고 $CachedRegion(RT_{jl}).G \cap CachedRegion(RT_{i(k-1)}).G$ 이면 RT_{ik}는 T_j의 l번째 재수행 트랜잭션 RT_{jl}와 자동적으로 재충돌한다. 이 경우에 재완료를 요청한 RT_{ik}는 RT_{jl}의 수정 결과를 받아들일 것인지 결정해야 한다. 만약 RT_{ik}가 RT_{jl}의 수정 결과를 받아들여서 RT_{ik}를 완료하고 쓰기 집합을 서버에 병합한다면 $RT_{i(k-1)}$의 충돌은 해소되고 최상위 트랜잭션인 $RT_{i0} = T_i$도 최종적으로 완료된다. 그러나 RT_{ik}가 받아들이지 않는다면 $RT_{i(k+1)}$이 시작되고 재충돌한 충돌 객체는 다음 재수행 트랜잭션인 $RT_{j(l+1)}$ 또는 $RT_{i(k+1)}$에 의하여 해소된다.

예를 들어 그림 14에서 보듯이 RT_{41}이 재완료할 때 RT_{11}에 대하여 변경 충돌 검사를 수행한다. $WS(T_4) \cap WS(RT_{11}) = \{B\}$이기 때문에 RT_{11}의 재수행 트랜잭션인 RT_{12}는 {B}를 재수정하기 위하여 수행중이다. RT_{41}의 쓰기 집합은 T_1과 T_4간의 충돌 객체들인 두 개의 부분 집합으로 구성된다. 하나는 RT_{11}과 관련이 없는 {D, E}이고 다른 하나는 RT_{11}의 쓰기 집합과 교차하는 {B}이다. {B}에 대한 재 충돌은 RT_{41}이 재완료하는 시점에서 RT_{11}의 결과를 받아들여서 최종적으로 해소될 수 있다. 그러므로 RT_{41}이 RT_{11}의 결과를 받아들인다면 RT_{12}의 재수행 작업을 쓸모없는 일이 된다. 이 부분은 재수행 트랜잭션 모델이 극복해야 될 단점이다.

7. 재수행 트랜잭션의 일관성 단계

트랜잭션의 직렬화 상태를 나타내는 일관성 단계(consistency level)는 크게 4가지 단계로 구성된다. 단계 0는 비완료 데이타 읽기(uncommitted data read) 단계로 최저 하위 단계이다. 단계 1은 완료 데이타 읽기(committed data read) 단계로 트랜잭션이 비완료 데이타 읽기 오류를 방지하고 완료된 데이타만 읽을 수 있도록 하는 단계이다. 단계 2는 반복 읽기(repeatable

read) 단계로 비일관적 분석(inconsistent analysis) 오류를 방지하고 트랜잭션 내에서 반복하여 읽기 연산을 수행하는 경우에 동일한 값을 읽을 수 있도록 보장하는 단계이다. 그리고 단계 3은 직렬화(serializable) 단계로 팬텀(phantom) 오류를 방지하는 단계이다. 하위 단계에서 발생하는 오류들은 상위 단계에서 방지되기 때문에 상위 단계일수록 좀 더 엄격하다.

확장 검증 조건하에서 모바일 트랜잭션 및 재수행 트랜잭션들은 단절된 클라이언트에서 수행되기 때문에 수행 기간 동안 서버 및 다른 모바일 클라이언트와 물리적으로 단절되어 있다. 트랜잭션의 읽기 연산은 트랜잭션의 시작 시 전송되어 캐싱된 지역 데이타에 대하여 수행되며 쓰기 연산의 결과는 완료되기 전까지 지역 저장 장치 또는 임시 버퍼에서 유지된다. 즉, 정의 1에 따라 재수행 트랜잭션 RTik의 RS(RT_{ik})는 CachedRegion(RT_{ik})에 속하는 객체들의 집합으로 RS(RT_{ik})를 구성하는 객체 중 동시 수행된 트랜잭션 T_j 결과인 foreign_conflicted_objects($RT_{i(k-1)}$)는 T_j의 완료된 객체로만 구성된다. 또한 비완료 데이타인 conflicted_objects($RT_{i(k-1)}$)은 임시 버퍼에 저장되고 RT_{ik}가 재완료할 때까지 다른 트랜잭션에게 노출되지 않는다. 따라서 트랜잭션의 비완료 데이타는 물리적으로 다른 트랜잭션이 접근할 수 없고 읽기 집합의 모든 객체는 완료된 데이타로만 구성되기 때문에 재수행 트랜잭션은 완료 데이타 읽기 단계를 보장한다.

비일관적 분석 오류는 참조 무결성(referential integrity)을 유지하지 못하는 경우에 발생하는 오류이다. 공간 데이타 측면에서 공간 객체간의 참조 무결성은 일관된 공간 관련성 유지이다. 확장 검증 조건인 정의 6의 간접 충돌 조건은 동시 수정된 두 공간 객체간의 공간 관련성 오류를 검사한다. 그리고 정의 8에 의해 RT_{ik}의 foreign_conflicted_objects(RT_{ik})는 conflicted_objects (RT_{ik})의 객체와 간접 충돌하는 객체들의 집합이다. 확장 검증 조건에 의해 구성된 conflicted_objects(RT_{ik})는 점진적 재수행 기법에 의하여 외부에 노출되지 않고 다음 단계인 $RT_{i(k+1)}$에서 foreign_conflicted_objects(RT_{ik})를 이용하여 재수행된다. $RT_{i(k+1)}$를 통하여 conflicted_objects(RT_{ik})의 비일관성은 해소되며 $RT_{i(k+1)}$의 재완료시 충돌하지 않으면 WS($RT_{i(k+1)}$)로써 안정 저장장치에 저장된다. 따라서 재수행 트랜잭션은 비일관적 분석 오류 발생시 다음 단계의 재수행 트랜잭션에서 일관된 데이타로 변경하여 저장하기 때문에 반복 읽기 단계를 보장한다.

재수행 트랜잭션 수행시 팬텀 오류는 발생할 수 없다. 팬텀 오류는 트랜잭션의 수행 중에 다른 트랜잭션의 삽입 연산에 의해 발생되는 오류이다. 그러나 수행 기간 동안 재수행 트랜잭션 RT_{ik}는 물리적으로 단절되어 있기 때문에 RS(RT_{ik})는 재수행 트랜잭션의 시작 시 서버로부터 전송된 객체들로만 구성되며 수행 중에는 변하지 않는다. 즉 수행중인 RT_{ik}는 T_j가 삽입한 WS(T_j)의 객체에 접근할 수 없다. 따라서 재수행 트랜잭션은 직렬화 단계를 보장한다.

8. 구 현

8.1 시스템 구조

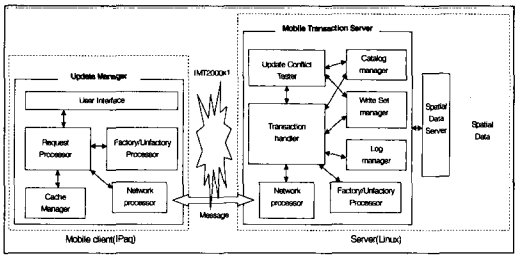


그림 15 모바일 현장 시스템 프로토타입 시스템 구조

그림 15은 이 논문에서 제시한 재수행 트랜잭션 모델을 지원하는 모바일 현장 시스템 프로토타입의 구현 환경과 시스템 구조를 보여준다. 트랜잭션 서버는 리눅스 시스템을 기반으로 한 공간 데이타 서버상에서 수행되며 모바일 클라이언트는 Windows CE 3.0을 기반으로 한 Compaq Ipaq상에서 수행된다. 트랜잭션 서버는 IMT2000-1x 무선 네트워크를 통하여 각 모바일 클라이언트와 필요한 데이타 및 메시지 교환을 수행한다. 서버에서 영구 저장 장치로의 공간 데이타 입출력을 위한 공간 데이타 서버는 사이버맵 데이타 서버이다.

그림 15에서 보듯이 서버측의 트랜잭션 서버는 네트워크 처리 모듈, 팩토리 처리 모듈, 트랜잭션 처리기, 변경 충돌 검사기, 카탈로그 매니저 그리고 쓰기 집합 매니저로 구성된다. 그리고 모바일 클라이언트에서 트랜잭션 수행을 위한 모듈은 사용자 인터페이스, 캐쉬 매니저, 요구 처리기, 팩토리 처리기, 네트워크 처리기로 구성된다. 주요 모듈의 기능은 다음과 같다.

- 트랜잭션 처리기 : 모바일 클라이언트에서 수행되고 있는 모바일 트랜잭션을 관리하고 모바일 클라이언트에서 보내온 트랜잭션 명령을 수행한다.
- 변경 충돌 검사기 : 모바일 트랜잭션이 완료를 요청했을 때 확장 검증 조건을 이용하여 전송된 쓰기 집합과 이미 완료된 트랜잭션들의 쓰기 집합들간의

충돌 여부를 검사한다. 또한 충돌이 발생할 때 충돌 객체와 외래 충돌 객체 집합을 구성한다.

- 카탈로그 매니저 : 모바일 트랜잭션의 충돌 검사를 위하여 트랜잭션의 시작, 완료 타임스탬프 및 캐싱 영역 등을 관리한다.
- 쓰기 집합 매니저 : 새로 완료를 요청할 트랜잭션의 충돌 검사를 위하여 이미 완료된 트랜잭션들의 쓰기 집합들을 저장하고 관리한다.
- 캐쉬 매니저 : 모바일 클라이언트의 지역 저장 장치에 캐싱된 지역 데이타 등을 관리한다. 또한 모바일 트랜잭션 수행 중에 발생한 쓰기 집합의 객체들을 유지, 관리한다.

이러한 수행 모듈을 이용하여 재수행 트랜잭션과 점진적 재수행 기법은 다음과 같이 수행된다.

- 재수행 트랜잭션 : 모바일 트랜잭션의 요구 처리기가 사용자로부터 모바일 트랜잭션의 완료 또는 재수행 트랜잭션의 재완료를 요청 받으면 완료될 트랜잭션의 쓰기 집합을 구성하여 서버 측의 트랜잭션 처리기로 전송한다. 트랜잭션 처리기는 서버 측에서 완료 연산의 수행을 조정하는 역할이며 변경 충돌 검사기를 통하여 전송 받은 쓰기 집합의 충돌 여부를 검사한다. 이를 위해 충돌 검사기는 카탈로그 매니저로부터 이미 완료된 다른 트랜잭션의 메타 정보를 검색하고 쓰기 집합 매니저를 통하여 비교할 쓰기 집합을 읽는다. 완료된 트랜잭션의 메타 정보를 이용하여 인터리빙 트랜잭션이며 동시에 중첩 트랜잭션인 검사 대상 트랜잭션들을 선택한 후에 대상 트랜잭션의 쓰기 집합과 완료 요청한 트랜잭션의 쓰기 집합을 비교한다. 만약 충돌이 감지되면 충돌 검사기는 쓰기 집합으로부터 충돌 객체와 외래 충돌 객체들을 검색한다. 최종적으로 트랜잭션 처리기는 다음 단계의 재수행 트랜잭션을 시작하고 구성된 충돌 객체와 외래 충돌 객체를 모바일 클라이언트에게 전송한다.
- 점진적 재수행 기법: 만약 충돌 객체 집합이 구성되면 트랜잭션 처리기는 충돌된 트랜잭션의 쓰기 집합으로부터 비충돌 객체들의 집합을 구성한다. 다음 단계의 재수행 트랜잭션을 시작하기 위하여 충돌 객체와 외래 충돌 객체를 모바일 클라이언트에게 전송한 후에 트랜잭션 처리기는 쓰기 집합의 임시 버퍼에 저장되어 있는 비충돌 객체들을 공간 데이타 서버를 통하여 안정 저장 장치에 쓴다. 그리고 다른 트랜잭션의 접근을 허용한다. 트랜잭션 처리기는 충돌 객체는 버퍼에 계속 유지하면서 다른 트랜잭션의 접근을 허용하지 않기 때문에 쓰기 집합의 비충돌 객체들만 외부에 노출된다.

8.2 구현 예

이 절에서는 지번 데이타와 건물 데이타를 수정하는 모바일 트랜잭션들이 충돌할 때 재수행 트랜잭션을 통하여 변경 충돌을 해소하는 예를 보여준다.

그림 16 모바일 트랜잭션 T_1, T_2 그리고 T_3의 초기 화면

그림 16는 인접 지역의 공간 데이타를 수정하는 세 모바일 트랜잭션의 초기 화면을 보여준다. 트랜잭션은 commit(T_2) < commit(T_1) < commit(T_3) < commit(RT_{11})의 순이다. 그림 17에서 보듯이 T_1은 두 개의 건물 객체를 수정하고(a) T_2는 하나의 도로 객체를 수정한다(b). T_2가 완료된 후 T_1이 완료를 요청할 때 수정한 건물 객체들은 T_2가 수정한 도로 객체와 접합(meet) 관계가 된다. 따라서 서버는 T_1의 충돌 객체와 외래 충돌 객체를 검색하여 T_1의 클라이언트로 보내고 재수행 트랜잭션인 RT_{11}을 시작한다.

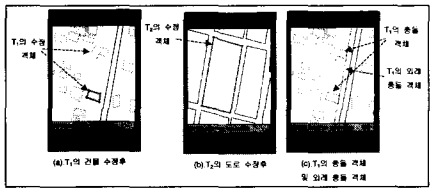


그림 17 T_1과 T_2간의 충돌

그림 17(c)에서 보듯이 RT_{11}은 서버로부터 전송 받은 충돌 객체와 외래 충돌 객체를 화면에 디스플레이한다. 그림 18(a)에서 RT_{11}의 사용자는 충돌을 해소하기 위하여 충돌 객체를 재수정하여 재완료를 요청한다. RT_{11}의 재수행 기간 중에 그림 18(b)와 같이 T_3가 건물을 수정하고 완료하였다. 그림 18(c)에서 보듯이 RT_{11}이 재완료를 요청한 객체와 T_3의 객체가 재충돌하기 때문에 서버는 RT_{11}의 충돌 객체와 외래 충돌 객체를 모바일 클라이언트로 전송하고 RT_{12}를 시작한다.

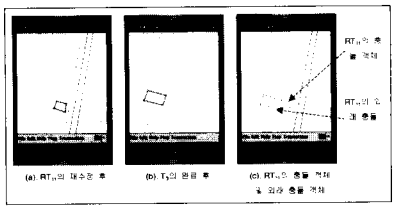


그림 18 RT_{11}의 재완료

그림 19(a)에서 보듯이 RT_{11}에서 재수정한 객체들 중 비 충돌 객체는 서버에 저장되고 다른 트랜잭션인 T_1에게 노출된다. RT_{12}의 사용자는 재 충돌한 객체를 재수정한 후에 재완료를 할 때 더 이상 다른 트랜잭션과 충돌하기 않기 때문에 그림 19(b)와 19(c)에서 보듯이 RT_{12}는 완료되고 최종적으로 T_1은 완료된다. 그림 19(c)는 공간 데이타베이스의 최종 상태를 보여준다.

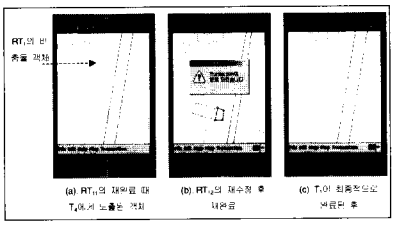


그림 19 RT_{12}의 재완료

9. 결 론

무선 네트워크는 자주 단절되는 특성을 가지고 있기 때문에 모바일 트랜잭션은 대부분의 수정 기간동안 단절되어 있고 필요한 경우에만 서버와 연결하는 약 연결 상태를 유지한다. 이러한 특성 때문에 단절 상태에서 객체를 수정할 수 있는 검증 기반 기법을 일반적으로 사용한다. 그러나 공간 객체를 수정하는 모바일 트랜잭션은 긴 트랜잭션이기 때문에 충돌된 트랜잭션이 강제 철회되면 공간 객체들에 대한 작업을 마치기 위해서 모든 수정 작업을 다시 수행해야 하는 고비용이 요구된다. 특히 기존의 검증 조건이 공간 객체를 수정하는 트랜잭션에 적용하기에 엄격하기 때문에 충돌 검사를 수행할 때 다수의 트랜잭션 철회가 발생한다.

충돌 검사가 실패할 때 트랜잭션을 대기 또는 철회하지 않기 위하여 이 논문에서 재수행 트랜잭션 모델을 제시하였다. 그리고 재수행 트랜잭션 모델을 지원하는 트랜잭션 서버를 설계하고 모바일 현장 시스템 프로토타입을 제안하였다. 재수행 트랜잭션은 충돌 객체만을 재수정하여 충돌을 해소한다. 올바른 충돌 해소를 위하여 재수행 트랜잭션은 외래 충돌 객체를 사용한다. 외래 충돌 객체는 이미 완료된 트랜잭션의 쓰기 집합에 속하는 객체로 충돌 객체와 간접 충돌하는 객체이다. 재수행 트랜잭션은 모든 충돌 객체의 충돌이 해소될 때까지 연속적으로 시작되어 수행되기 때문에 반복적으로 수행될 때 트랜잭션 기아 문제가 발생한다. 재수행 트랜잭션의 기아 문제를 줄이기 위하여 점진적 재수행 기법을 제시하였다. 점진적 재수행 기법은 k+1번째의 재수행 트랜잭션이 시작되기 전에 비충돌 객체는 서버에 부분적으로 완료하고 다른 트랜잭션의 접근을 허용한다. 그리고 k번째 트랜잭션의 충돌 객체만을 재수행한다.

충돌 객체가 쓰기 집합에서 차지하는 비율이 낮고 현장에서의 작업 시간이 길수록 이 논문에서 제시한 재수행 트랜잭션 기법이 기존의 철회 기법에 비하여 효율적이다. 향후 연구로는 불필요한 재수행 트랜잭션을 줄이기 위한 기법과 쓰기 집합의 충돌 객체가 매우 많은 경우를 위한 연구가 필요하다.

참 고 문 헌

[1] Sanjay Kumar Madria, Bharat Bhargava: A Transaction Model for Mobile Computing, Int. Database Engineering and Application Symposium (1998), 92-102.

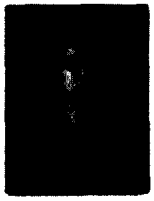
[2] Nuno Preguica, Carlos Baquero: Mobile Transaction Management in Mobisnap, Advances in Databases and Information Systems(2000), 379-386.

[3] Peng Liu, Paul Ammann, Sushil Jajodia: Incorporating Transaction Semantics to Reduce Reprocessing Overhead in Replicated Mobile Data Applications, Int. Conf. on Distributed Computing Systems(1999).

[4] Shirish Hemant Phatak, B. R. Badrinath: Multiversion Reconciliation for Mobile Databases, Int. Conf. on Data Engineering(1999).

[5] Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha: The Dangers of Replication and a Solution, Proc. of the 1996 ACM SIGMOD(1996), 173-182.

[6] Justus Klingemann, Thomas Tesch, Jurgen Wasch: Enabling Cooperation among Disconnected Mobile Users, Conf. on Cooperative Information Systems(1997), 36-45.



김 동 현
1995년 부산대학교 컴퓨터공학과(학사)
199년 부산대학교 컴퓨터공학과(석사).
1998년 ~ 현재 부산대학교 컴퓨터공학과 작사과정 재학중. 관심분야는 지리정보시스템, 모바일 GIS, 모바일 트랜잭션, 이동체 색인

홍 봉 희
정보과학회논문지 : 데이타베이스
제 30 권 제 1 호 참조