공간데이터를 변경하는 모바일 트랜잭션의 변경 전파 회복 기법

A Recovery Scheme of Mobile Transaction Based on Updates Propagation for Updating Spatial Data

김동현*, 강주호**, 홍봉희*
Dong-Hyun Kim, Ju-Ho Kang, Bong-Hee Hong

요 약 공간 객체를 변경하기 위한 모바일 트랜잭션은 단절 상태에서 지역 데이터를 변경하는 긴 트랜잭션이다. 모바일 트랜잭션이 회복할 때 단절 상태로 인하여 회복 시점보다 먼저 완료된 다른 트랜잭션의 쓰기 집합을 읽을 수 없기 때문에 회복된 트랜잭션이 충돌할 수 있다. 그러나 만약 회복된 긴 트랜잭션을 변경 충돌 때문에 철회하면 회복된 데이터를 포함한 모든 변경 데이터를 취소해야 하기 때문에 장애가 발생한 모바일 트랜잭션의 회복 기법으로 적합하지 않다.

이 논문에서는 회복된 모바일 트랜잭션이 변경 충돌로 인해 철회되는 것을 줄이기 위하여 다른 트랜잭션의 쓰기 집합에서 외래충돌가능객체를 검색하는 회복 기법을 제안한다. 외래충돌가능객체는 회복 시점보다 먼저 완료된 다른 트랜잭션이 변경한 객체로서 회복된 트랜잭션이 변경한 객체와 충돌 가능한 객체이다. 제안한 기법은 회복할 때 외래충돌가능객체를 최근 검사점 상태의 읽기 집합과 함께 읽기 때문에 회복된 트랜잭션이 변경 충돌이 발생하지 않도록 객체를 재변경할 수 있다.

ABSTRACT Mobile transactions updating spatial objects are long transactions that update local objects of mobile clients during disconnection. Since a recovered transaction cannot read the write sets of other transactions committed before the recovery due to disconnection, the recovered transaction may conflicts with them. However, aborting of the recovered long transaction leads to the cancellation of all updates including the recovered updates. It is definitely unsuitable to cancel the recovered updates due to the conflicts.

In this paper, we propose the recovery scheme to retrieve foreign conflictive objects from the write sets of other transactions for reducing aborting of a recovered transaction. The foreign conflictive objects are part of the data committed by other transactions and may conflict with the objects updated by the recovered transaction. In the scheme, since the recovered transaction can read both the foreign conflictive objects and the recently checkpoint read set, it is possible to reupdate properly the potentially conflicted objects.

주요어 : 트랜잭션 회복, GIS, 공간데이터베이스, 모바일 트랜잭션
Key word : transaction recovery, GIS, spatial database, mobile transaction

1. 서 론

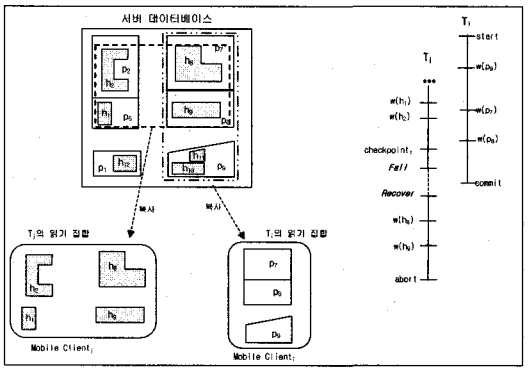
무선 통신과 모바일 클라이언트가 발달함에 따라 사용자가 현장에서 직접 지형지물을 확인하며 공간 데이터를 변경할 수 있다. 사용자는 서버에 위치한 공간 데이터베이스로부터 무선 통신을 통하여 공간 데이터를 모바일 클라이언트에 다운로드받은 후 지형지물을 확인하면서 해당 데이터를 변경한다. 변경 작업이 완료되면 변경된 데이터를 무선 통신을 이용하여 서버의 데이터베이스에 병합한다[1][2][3][4]. 모바일 클라이언트를 이용하여 공간 데이터를 변경하는 기법은 최신의 변경 내용을 모든 사용자들이 신속히 공유하는 장점을 제공한다.

모바일 클라이언트에서 공간 데이터를 변경하기 위한 모바일 트랜잭션(mobile transaction)은 크게 두 가지 특징을 가진다. 첫 번째로 모바일 트랜잭션은 서

* 부산대학교 컴퓨터 및 정보통신 연구소
** 부산대학교 컴퓨터공학과
{pusrover, jooh078, bhhong}@ pusan.ac.kr

버와 단절 상태에서 대부분의 변경 연산을 수행한다. 비록 모바일 클라이언트가 무선망을 이용할 수 있지만 무선망은 유선망에 비하여 불안정하고 자주 단절되는 특성이 있기 때문이다[1][2][3][4]. 단절 기간 동안 모바일 트랜잭션은 다른 트랜잭션의 쓰기 집합을 알 수 없다. 둘째로 모바일 트랜잭션의 사용자가 현장에서 지형지물을 측정하며 공간 데이터를 변경하기 때문에 모바일 트랜잭션은 수 십분 또는 수 시간 동안 수행된다. 따라서 공간 데이터 변경을 위한 모바일 트랜잭션은 단절 상태에서 사용자 대화 방식으로 수행되는 긴 트랜잭션(Long Transaction)이다[1][4][5].

장애가 발생한 트랜잭션을 회복하기 위하여 일반적으로 로그 기반의 검사점 (checkpoint)[6]을 이용한다. 검사점은 회복에 필요한 정보를 로그에 저장하는 연산으로 메모리에 저장된 로그레코드들을 안정저장장치의 로그에 저장하고 검사점 로그레코드를 추가한다. 검사점을 이용한 회복 기법은 크게 두 가지로 분류된다. 첫 번째는 유선망을 기반으로 한 동기적검사점 기법[7][8]이다. 이 기법에서 서버는 주기적인 검사점 수행 시 모든 트랜잭션들의 로그레코드들을 동기적으로 서버의 로그에 저장하고 트랜잭션에 장애가 발생하면 로그를 이용하여 최근의 검사점 시점으로 트랜잭션을 회복한다. 그러나 단절된 모바일 트랜잭션의 로그레코드를 동기적으로 서버에 저장할 수 없는 단점이 있다. 두 번째는 무선망을 기반으로 한 비조정검사점 기법[9]이다. 이 기법은 모바일 트랜잭션의 단절 상태를 고려하여 로그레코드를 모바일 클라이언트의 지역 저장장치에 주기적으로 저장하는 검사점을 수행한다. 트랜잭션에 장애가 발생하면 모바일 클라이언트의 로그를 이용하여 최근의 검사점 시점으로 회복한다.



〈그림 1〉 공간 객체를 동시 변경하는 모바일 트랜잭션 회복의 예

〈그림 1〉은 비조정검사점 기법에 따라 중첩 지역의 공간 객체를 동시 변경하는 모바일 트랜잭션에 장애가 발생한 후 회복한 예를 보여준다. 모바일 트랜잭션 T_j와 T_i가 모바일 클라이언트의 작업 영역으로 복사한 각각의 읽기 집합은 {h_1, h_2, h_8, h_9}과 {p_7, p_8, p_9}이다. T_j는 단절 상태에서 h_1과 h_2를 변경한 후 첫 번째 검사점을 수행한다. 첫 번째 검사점은 h_1과 h_2의 로그 레코드를 모바일 클라이언트의 지역저장장치에 저장하고 검사점 로그레코드를 추가한다. T_j가 검사점을 수행한 후에 장애가 발생한다고 가정하자. T_i는 p_7, p_8 그리고 p_9를 변경하고 T_j의 회복 시점보다 먼저 완료하였다. Tj는 첫 번째 검사점 로그레코드까지 검색된 h_1과 h_2의 로그를 이용하여 최근의 검사점 시점으로 회복한 후에 h_8과 h_9을 변경한다.

그러나 무선망 기반의 비조정검사점 기법을 이용하여 동시 수행되는 모바일 트랜잭션을 회복할 때 다음과 같은 문제가 발생한다. 만약 장애로부터 회복된 트랜잭션 T_j가 회복 시점보다 먼저 완료된 T_i와 충돌하면 T_j는 철회(abort)되고 회복된 데이터를 포함한 모든 변경 데이터는 취소된다. 예를 들어 〈그림 1〉에서 보듯이 T_j와 T_i는 중첩 지역의 h_9과 p_8을 동시 변경하며 h_9은 p_8과 내부공간관련성(inside spatial relationship)을 가지고 있다. 단절 상태인 T_j는 T_i의 쓰기 집합을 알 수 없기 때문에 동시 변경된 h9와 p8 간에 비일관적인 공간 관련성이 생성될 수 있다[1][4]. 따라서 만약 T_j가 완료 연산 수행 시 T_i와 충돌하면 T_j는 철회되고 회복된 h_1과 h_2를 포함한 T_j의 쓰기 집합은 취소된다. 변경 작업을 마치기 위해서는 T_j를 재시작하고 모든 연산을 반복하는 고비용이 요구된다.

이 논문의 기본 아이디어는 복구된 트랜잭션이 회복 시점보다 먼저 완료된 다른 트랜잭션의 쓰기 집합에 속하는 객체를 최근 검사점 상태의 읽기 집합과 함께 읽는 것이다. 이를 위하여 먼저 모바일 트랜잭션 T_j의 최근 검사점 상태 읽기 집합을 생성하는 강제 로깅 기법을 제시한다. 강제 로깅 기법에서 T_j는 주기적으로 서버에 강제 접속하여 T_j의 로그레코드들을 서버의 안정저장장치에 저장한다. 만약 T_j에 장애가 발생하면 서버에서 검색된 T_j의 로그를 이용하여 최근 검사점 상태의 읽기 집합을 생성한다. 그리고 회복 연산을 수행할 때 먼저 완료된 다른 트랜잭션의 쓰기 집합에 속하는 외래충돌가능객체(foreign conflictive objects)를 검색하는 회복 기법을 제안한다. 외래충돌가능객체는 장애가 발생한 T_j가 회복하기 전에 완료된 T_i의 쓰기 집합에 속하는 객체로서 T_j가 변경한 객체와 변경

충돌이 발생할 수 있는 객체이다. 예를 들어 〈그림 1〉의 p_8은 T_j가 변경한 h_9과 충돌할 수 있는 외래충돌가능객체이다. T_j의 회복 연산 수행 시 회복관리자는 T_j의 회복 시점보다 먼저 완료된 T_i의 쓰기 집합으로부터 T_j의 외래충돌가능객체를 검색한다. 그리고 최근 검사점 상태의 읽기 집합과 외래충돌가능객체를 함께 T_j의 작업 영역으로 전송한다. T_j는 외래충돌가능객체를 최근 검사점 상태의 읽기 집합과 함께 읽어서 디스플레이하기 때문에 사용자는 T_j를 완료하기 전에 T_i와 충돌 가능한 객체를 변경할 수 있다.

장애가 발생한 T_j의 회복 연산을 수행할 때 외래충돌가능객체를 검색하여 최근 검사점 상태의 읽기 집합과 함께 읽는 이유는 공간 객체를 변경하는 트랜잭션 T_j가 사용자에 의해 수행되는 긴 트랜잭션이기 때문이다. 장애가 발생한 긴 트랜잭션 T_j를 회복시켰지만 만약 다른 트랜잭션과의 변경 충돌 때문에 철회시킨다면 매우 비효율적이며 취소된 공간 데이터의 변경을 완료하기 위하여 고비용이 요구된다.

이 논문의 구성은 다음과 같다. 2장에서 관련 연구를 기술하고 3장에서는 회복된 트랜잭션의 변경 충돌로 인한 철회 문제를 기술한다. 4장에서 강제 로깅 기법에 대하여 기술하고 5장에서 외래충돌가능객체를 이용한 회복 기법을 제시한다. 6장에서 외래충돌가능객체 기반의 회복 기법을 이용한 회복 시스템의 설계 및 구현을 기술하고 7장에서 결론을 기술한다.

2. 관련 연구

무선망을 기반으로 수행되는 모바일 트랜잭션이 실패할 때 트랜잭션 회복을 위한 연구는 크게 두 가지 측면에서 기존 연구가 수행되었다.

첫 번째는 트랜잭션의 단절 상태를 고려하여 모바일 클라이언트의 지역저장장치를 이용하는 방법이다. 비조정검사점(Uncoordinated Checkpoint) 기법에서는 모바일 트랜잭션이 단절 상태 하에서 수행되는 기간 동안 모바일 클라이언트 내의 임시저장장치에 주기적으로 로그레코드를 저장한다[9]. 만약 트랜잭션 수행 시 장애가 발생하면 모바일 클라이언트의 임시저장장치에 저장된 로그와 서버에 저장된 로그를 병합하여 일관된 상태로 회복한다. 그러나 이 기법에서는 모바일 클라이언트 자체에 발생하는 심각한 장애를 고려하지 않는다. 따라서 단절된 상태하에서 만약 모바일 클라이언트의 심각한 장애로 인하여 임시저장장치의 로그를 모두 손실하면 최근 검사점 상태로 회복할 수 없는 문제가 있다.

두 번째 방법은 유선망 기반의 동기적검사점 기법을 응용하는 방법이다. 통신유도검사점(Communication-induced Checkpoint) 기법은 모바일 분산 환경에서 동기적검사점 기법의 문제를 해결하기 위해 타이머를 이용한다[10]. 즉, 각 모바일 트랜잭션의 타이머를 동기화하여 검사점 수행 시점을 일치화한다. 각 모바일 트랜잭션이 시작될 때 트랜잭션 내에 검사점 수행 타이머를 설정하고, 모바일 클라이언트 간의 교환 메시지에 타이머 동기화 정보를 포함하여 전송한다. 그러나 이 기법은 타이머 동기화를 위해 모바일 클라이언트 간에 불필요한 메시지 교환이 발생하는 문제가 있다. 또한 모바일 트랜잭션이 단절되었을 때 타이머 동기화 메시지를 받지 못하기 때문에 전체 모바일 트랜잭션의 검사점 수행 시점이 동기화되지 않는 문제가 있다.

그러나 기존의 모바일 트랜잭션 회복 기법[9][10]은 트랜잭션에 장애가 발생할 때 저장된 로그를 이용하여 최근 검사점 상태로의 회복을 위한 기법만을 기술하고 있으며 모바일 트랜잭션이 회복할 때 발생할 수 있는 변경 충돌에 관한 기존의 연구는 없다. 만약 기존의 기법을 모바일 트랜잭션의 회복에 적용할 때 만약 장애로부터 회복된 트랜잭션 T_j가 회복 시점보다 먼저 완료된 T_i와 충돌하면 T_j는 철회(abort)되고 회복된 데이터를 포함한 모든 변경 데이터는 취소되는 문제가 있다. 이러한 문제를 해결하기 위하여 이 논문에서는 트랜잭션 T_j가 회복할 때 회복 시점보다 먼저 완료된 다른 트랜잭션 T_i의 쓰기 집합에 속하는 객체를 최근 검사점 상태의 읽기 집합과 함께 읽는다.

트랜잭션들이 동시 수행되고 있는 상태하에서 트랜잭션의 회복으로 인한 데이터 비일관성을 다룬 연구로는 엄격한 2단계 잠금 기법(strict 2-phase locking)을 이용한 기법이 있다[6]. 이 기법은 읽기 잠금과 쓰기 잠금을 이용하여 트랜잭션이 완료 혹은 철회될 때까지 다른 트랜잭션이 중복 영역의 공간 객체를 변경하는 것을 금지한다. 따라서 트랜잭션의 완료 시 변경 충돌로 인해 회복된 트랜잭션이 철회될 필요가 없다. 그러나 이 기법은 데이터 일관성을 유지하기 위하여 잠금을 사용하기 때문에 단절 상태하에서 수행되는 모바일 트랜잭션에 적합하지 않다. 특히 긴 트랜잭션인 공간 데이터 변경 작업에서는 동시 수행중인 다른 트랜잭션이 중첩 영역의 다른 공간 객체에 대한 쓰기 잠금을 획득하기까지 오랜 시간을 기다려야 하는 문제가 발생한다.

3. 문제 정의

모바일 트랜잭션은 대부분의 수정 기간 동안 서버와

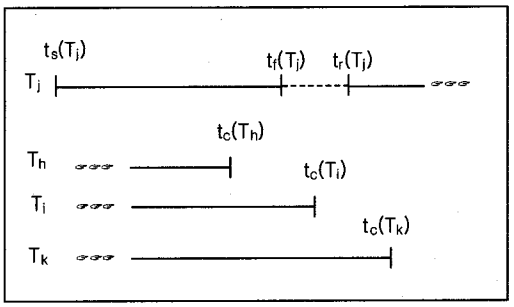
단절되는 특성을 가지고 있기 때문에 트랜잭션의 시작 시 서버 데이터베이스의 일부를 모바일 클라이언트의 임시저장장치에 복사한다. 그리고 단절 기간동안 임시저장장치의 지역 데이터를 독립적으로 변경한다. 따라서 모바일 트랜잭션의 수행 방법으로 낙관적 기법의 하나인 검증기반프로토콜(validation based protocol)이 적합하다[1][2][11].

검증기반프로토콜에 따라 모바일 트랜잭션 T_j는 크게 세 단계로 수행된다. 읽기 단계에서 T_j는 지역저장장치의 공간 객체를 변경한다. 모든 객체에 대한 변경을 마치면 T_j는 완료를 위하여 검증 단계에서 T_j보다 먼저 완료된 T_i와의 충돌 여부를 검사한다. 만약 변경 충돌이 없으면 쓰기 단계에서 T_j의 쓰기 집합을 서버의 공간 데이터베이스로 병합한다.

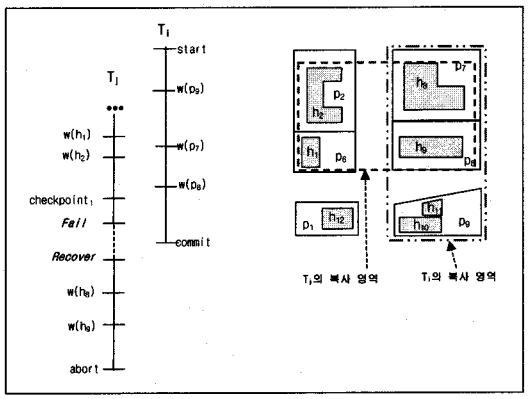
만약 읽기 단계에서 T_j에 장애가 발생하면 T_j는 최근 검사점 상태로 회복할 수 있어야 한다. 모바일 트랜잭션의 단절상태를 고려할 때 일반적으로 비조정검사점 기법이 적합하다. 트랜잭션 수행 중에 T_j는 비조정검사점 기법에 따라 주기적인 검사점을 실행한다. T_j의 k번째 검사점을 checkpointk(T_j)라 하자. k번째 검사점 연산은 $checkpoint_{k-1}(T_j)$부터 $checkpoint_k(T_j)$까지의 로그레코드를 지역저장장치에 저장한다. T_j에 장애가 발생한 시점을 $t_f(T_j)$라 하고 T_j가 복구된 시점을 $t_r(T_j)$라 하자. 만약 $t_f(T_j)$에 장애가 발생하면 회복매니저는 지역저장장치의 로그에서 $MAX(checkpoint_k(T_j))\langle t_f(T_j)$인 $checkpoint_k(T_j)$까지의 로그레코드를 검색한다. 그리고 T_j의 읽기 집합에 검색된 로그레코드를 순차적으로 적용하여 $checkpoint_k(T_j)$ 상태의 읽기 집합과 쓰기 집합을 회복한 후 $t_r(T_j)$에 T_j를 다시 시작한다.

T_j의 시작과 완료 시점을 각각 $t_s(T_j)$와 $t_c(T_j)$라 할 때 $t_r(T_j)$에 회복된 트랜잭션 T_j와 동시 수행된 트랜잭션은 〈그림 2〉와 같이 크게 세 가지가 있다.

첫 번째는 〈그림 2〉에서 보듯이 $t_s(T_j) < t_c(T_h) < t_f(T_j)$인 T_h이다. 그리고 두 번째는 $t_f(T_j) < t_c(T_i) < t_r(T_j)$인 T_i이다. T_h와 T_i의 쓰기 집합에 속하는 객체는 T_j가 회복할 때 읽을 수 있어야 한다. 왜냐하면 T_j의 회복 시 T_h, T_i와 T_j간의 직렬성 순서(Serializability Order)는 $T_h \rightarrow T_j$, $T_i \rightarrow T_j$이기 때문이다. 그러나 T_j는 단절되어 있기 때문에 Th와 T_i의 쓰기 집합을 알 수 없다. 회복된 T_j의 쓰기 집합에 속하는 객체 o_j와 $t_s(T_j) \langle t_c(T_h) \langle t_f(T_j)$ 또는 $t_f(T_j) < t_c(T_i) \langle t_r(T_j)$인 T_h 또는 T_i의 쓰기 집합에 속하는 객체 o_i가 있다고 하자. $o_j.G$를 o_j의 기하 영역이라 할 때 $o_j.oid = o_i.oid$ 또는 $o_j.G \cap o_i.G \neq \emptyset$이면 T_j는 T_h 또는 T_i와 충돌한다[1][4]. 검증 단계에서 Tj가 Ti와 충돌하면 데이터 일관성을 유지하기 위하여 T_j는 철회되고 회복된 데이터를 포함한 T_j의 모든 쓰기 집합은 취소된다. 따라서 공간 객체에 대한 변경을 마치기 위해서는 철회된 T_j를 재시작하고 변경 연산을 다시 수행해야 하는 고비용이 요구된다.



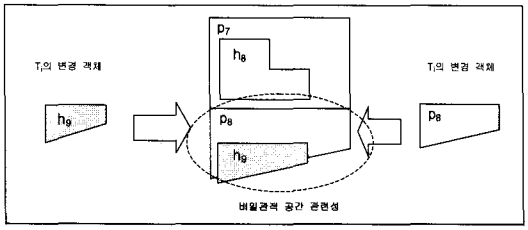
〈그림 2〉 장애가 발생한 트랜잭션과 동시 수행된 트랜잭션의 예



〈그림 3〉 중첩된 영역의 공간 객체를 동시 변경하는 두 트랜잭션 T_i와 T_j

〈그림 3〉은 T_i와 T_j가 각각 필지 객체와 건물 객체를 동시 변경하는 예이다. T_i는 필지 객체 p_7, p_8 그리고 p_9을 변경하고 T_j는 건물 객체 h_1, h_2, h_8 그리고 h_9를 변경한다. 〈그림 3〉에서 보듯이 T_j는 $checkpoint_1(T_j)$에 h_1과 h_2의 로그레코드를 지역저장장치에 저장한다. $checkpoint_1(T_j) \langle t_f(T_j)$인 $t_f(T_j)$에 심각한 장애가 발생하고 $t_f(T_j) \langle t_c(T_i) \langle t_r(T_j)$라고 가정하자. 단절 상태인 T_j는 T_i의 쓰기 집합을 알 수 없기 때문에 중첩지역에서 변경된 p_7과 p_8의 변경된 공

간데이터를 읽을 수 없다. 따라서 T_j는 h_1과 h_2의 로그레코드만을 순차적으로 적용하여 $t_r(T_j)$에 회복한다. 그리고 h_8과 h_9를 변경하였다. 그러나, 예를 들어 h_9과 p_8은 inside 공간 관련성을 가지고 있고 T_j가 p_8의 변경을 알 수 없기 때문에 동시 변경할 경우에 〈그림 4〉와 같이 h_9과 p_8간에 비일관적인 공간 관련성이 생성될 수 있다. 공간 데이터베이스의 일관성을 유지하기 위하여 T_j는 철회되고 회복된 객체인 h_1과 h_2를 포함한 T_j의 모든 변경 객체들은 취소된다.



〈그림 4〉 동시 변경으로 인한 공간 데이터 비일관성

이 논문에서는 $t_s(T_j)$ 〈 $t_c(T_i)$ 〈 $t_r(T_j)$인 T_i의 쓰기 집합에 속하는 객체 중 T_j의 쓰기 집합에 속하는 객체와 충돌 가능한 객체를 외래충돌가능객체(foreign conflictive object)라 한다.

또 한가지 고려해야 할 점은 심각한 장애(hard failure)로 인한 로그레코드 손실 문제이다. 비조정검사점 기법은 모바일 클라이언트의 단절 상태를 고려하여 검사점 연산 수행 시 로그레코드를 모바일 클라이언트의 지역저장장치에 저장한다. 그러나 모바일 클라이언트의 지역저장장치는 외부의 충격에 약하고 안정저장장치가 아니기 때문에 시스템 오류와 같은 심각한 장애가 발생하면 지역저장장치의 모든 데이터가 손실된다[17]. 따라서 회복에 필요한 로그레코드를 지역저장장치에 저장할 때 만약 심각한 장애가 발생하면 로그레코드들도 손실되기 때문에 최근 검사점 상태로 회복할 수 없다.

4. 강제 로깅 기법

모바일 트랜잭션 Tj를 장애로부터 회복하기 위해서는 먼저 최근 검사점 상태의 읽기 집합을 만드는 것이 필요하다. 이 장에서는 최근 검사점 상태의 읽기 집합을 생성하기 위하여 T_j의 로그레코드를 서버의 안정저장장치에 강제로 저장하는 강제 로깅 기법에 대하여 기술한다.

4.1 강제 로깅의 주기 및 수행 방법

비조정검사점 기법을 이용할 때 만약 모바일 클라이언트에 심각한 장애가 발생하면 회복에 필요한 로그레코드도 손실되기 때문에 최근 검사점 상태의 읽기 집합을 회복할 수 없다. 따라서 로그레코드를 서버의 안정저장장치에 저장할 필요가 있다. 이를 위하여 모바일 트랜잭션이 수행 도중에 강제적으로 서버에 연결을 설정하고 서버의 안정저장장치에 회복에 필요한 로그레코드들을 저장한다.

강제 로깅의 주기는 모바일 클라이언트의 로그레코드카운트(Log Record Count: LRC)와 최대로그레코드카운트(Max Log Record Count: MAX_LRC)를 이용하여 결정한다. 로그레코드카운트는 모바일 트랜잭션 시작 시에 0으로 초기화되고 변경 연산의 수행으로 인해 로그레코드가 생성될 때마다 1씩 증가한다. 로그레코드카운트가 미리 설정된 최대로그레코드카운트가 되었을 때 트랜잭션은 강제 로깅 연산을 수행한 후 로그레코드카운트를 0으로 재설정한다.

모바일 클라이언트측에서의 강제 로깅 알고리즘은 〈그림 5〉(가)와 같다. 두 개의 로그레코드카운트에 의해 강제 로깅 수행이 결정되면 모바일 클라이언트의 임시저장장치에 저장되어 있는 로그를 역방향으로 검색하여 강제 로깅의 시작 위치를 결정한다. k번째 강제 로깅의 시작 위치는 k-1번째 검사점 로그레코드 혹은 트랜잭션 시작 로그레코드이다. 강제 로깅의 시작 위치가 결정되면 시작 위치에서부터 로그레코드를 순방향으로 추출하여 로그레코드 리스트를 생성한다. 생성된 로그레코드 리스트는 트랜잭션 식별자와 함께 강제 로깅 요청 메시지인 request-forced-logging(t_id, log-list)를 구성하여 서버에 전송된다.

모바일 트랜잭션의 강제 로깅 요청 메시지를 받은 서버측의 강제 로깅 수행 알고리즘은 〈그림 5〉(나)와 같다. 강제 로깅 요청을 받은 서버는 메시지에 포함된 로그레코드 리스트를 자신의 안정저장장치에 저장한다. 저장이 성공적으로 수행되면 강제 로깅 완료 메시지를 전송하여 모바일 트랜잭션에게 응답한 후에 검사점 로그레코드를 안정저장장치에 저장한다. 그러나 만약 로그에 오류가 있거나 또는 다른 이유로 강제 로깅을 실패하면 모바일 트랜잭션에게 강제 로깅 실패 메시지를 전송한다.

4.2 트랜잭션 로그와 변경 로그

모바일 트랜잭션의 회복을 위한 로그는 크게 트랜잭션 로그(transaction log)와 변경 로그(update log)로 분류된다. 트랜잭션 로그는 트랜잭션의 연산을 기

```
log-list          //서버에 저장될 로그 리스트
log_record        //로그 레코드
forced_logging_start  // 강제 로깅을 시작할 로그 레코드의 위치

Procedure MC_Forced_Logging( t_id, LOG )
Begin
    log_list ← ø;
    log_record ← ø;

    //강제 로깅을 시작 위치 검색
    current_position ← end position of LOG
    WHILE scan the LOG backward
        log_record ← read a log record from LOG;
        IF log_record = <t_id start> OR <t_id checkpoint> THEN
            forced_logging_start ← current_position;
            EXIT LOOP
        END IF
        current_position ← current_position - 1;
    END LOOP
    //강제 로깅할 로그 리스트 작성
    WHILE scan the LOG forward from current position
        log_record ← read a log record from LOG;
        log_list ← add log_record;
    END LOOP
    //강제 로깅 수행
    send a message "request-forced-logging(t_id, log_list)";
    //강제 로깅에 성공하면 강제 로깅 성공 로그 레코드 남김
    IF response = reply-forced-logging( SUCCEED ) THEN
        append <t_id, checkpoint> to LOG;
    END IF
End
```

(가) 모바일 클라이언트 측의 강제 로깅 알고리즘

```
global_log  // global log list of a server

Procedure SVR_Forced_Logging( t_id, log_list )
Begin
    log_record ← ø;

    // 로그레코드 기정
    logging ← FAILURE;

    global_log ← read global log from the stable storage of a server;
    WHILE log_record IS NOT end of log_list
        log_record ← read a log record from log_list;
        append log_record to GLOBAL_LOG;
    END LOOP
    write global_log into the stable storage of a server;

    //검사점 레코드 추가
    logging ← SUCCEED;
    append <t_id, checkpoint> to GLOBAL_LOG;

    IF logging = SUCCEED THEN
        //강제 로깅에 성공하면 이를 해당 MC에 통보함
        send a message "reply-output( SUCCEED )" to the mobile client;
    ELSE
        send a message "reply-forced-logging( FAILED )" to the mobile
        client;
    END IF
End
```

(나) 서버 측의 강제 로깅 알고리즘

〈그림 5〉 강제 로깅 수행 알고리즘

록하는 로그로서 〈표 1〉과 같이 트랜잭션 시작 로그레코드, 검사점 로그레코드, 그리고 트랜잭션 완료 로그레코드이다. 트랜잭션 시작 로그레코드는 모바일 트랜잭션이 시작될 때 기록된다. 검사점 로그레코드는 모바일 트랜잭션의 강제 로깅이 성공적으로 수행될 때마다 로그에 추가된다. 트랜잭션 완료 로그레코드는 완료를 요청한 트랜잭션의 쓰기 집합이 검증 단계에서 먼저 완료된 다른 트랜잭션의 쓰기 집합과 충돌하지 않고 서버의 공간 데이터베이스로 병합될 때 기록된다. t_id(transaction identifier)는 트랜잭션의 식별자이다.

〈표 1〉 트랜잭션 로그레코드

트랜잭션 로그레코드 종류	구 조
트랜잭션 시작 로그레코드	(t_id, start)
검사점 로그레코드	(t_id, checkpoint)
트랜잭션 완료 로그레코드	(t_id, commit)

변경 로그는 공간 객체에 대한 변경 연산의 내용을 기록하는 로그이다. 기존의 변경 로그레코드는 변경 연산의 종류를 고려하지 않고 동일한 구조로 구성되었다. 그러나 모바일 트랜잭션은 무선 네트워크를 이용하여 변경 로그레코드들을 서버에 전송해야 하기 때문에 변경 로그레코드의 크기를 줄일 필요가 있다.

변경 로그레코드는 〈표 2〉와 같이 변경 연산인 삽입, 삭제 그리고 수정 연산에 따라 유형이 분류된다. 삽입 로그레코드는 공간 객체가 삽입되었을 때 삽입된 공간 객체의 객체 유형과 기하 속성으로 구성된다. 삭제 로그레코드는 공간 객체가 삭제되었을 때 삭제된 공간 객체의 식별자로만 구성된다. 수정 로그레코드는 공간 객체가 수정되었을 때 생성되며 기하 속성의 수정 정보를 포함한다.

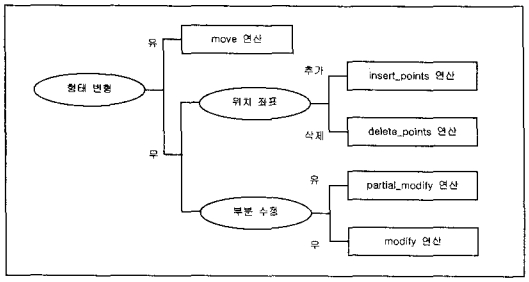
수정 로그레코드는 수정 연산의 결과인 기하 형태 변형의 유/무, 위치 좌표의 추가/삭제, 그리고 부분 수정의 유/무에 따라 〈그림 6〉과 같이 다시 세분화 될 수 있다.

이동(move) 연산은 기하 형태(shape)가 변형되지 않고 공간 좌표 체계상에서 객체의 위치만 수정하는 연산이다. 〈표 2〉의 이동 연산을 위한 로그레코드는 공간 객체의 기하 속성의 이동분만을 저장한다. 따라서 많은 위치 좌표로 구성된 객체의 이동 연산을 기록하기 위하여 수정된 위치 좌표 값들을 모두 저장하지

않고 이동분, 즉 한 좌표 크기의 정보만 저장하기 때문에 로그레코드의 크기를 감소시킬 수 있다.

〈표 2〉 변경 로그레코드

변경 로그레코드 종류	구 조
삽입 로그레코드	(t_id, insert, obj_type, new_points {(x, y)})
삭제 로그레코드	(t_id, delete, oid)
수정 로그레코드	(t_id, move, oid, delta (Δx, Δy)) (t_id, insert_points, oid, (seq, num), delta {(x, y)}) (t_id, delete_points, oid, (seq, num)) (t_id, partial_modify, oid, (seq, num), delta{(x, y)}) (t_id, modify, oid, new_points {(x, y)})



〈그림 6〉 수정 로그레코드의 세분화

점 삽입(insert_points) 연산은 공간 객체의 기하 속성인 좌표 값 집합의 특정 순서에 새로운 위치 좌표 값을 추가하는 연산이다. 점 삽입 연산을 위한 로그레코드는 좌표 값 집합에서의 순서와 삽입된 위치 좌표 값들의 개수, 그리고 삽입된 값들을 저장한다. 점 삭제(delete_points) 연산은 공간 객체의 좌표 값 집합에서 특정 순서에 있는 위치 좌표 값들을 삭제하는 연산이다. 따라서 점 삭제 연산을 위한 로그레코드는 삭제된 위치 좌표 값들에 대한 좌표 값 집합에서의 순서와 개수를 저장한다.

부분 수정(partial_modify) 연산은 공간 객체의 좌표 값 집합에서 특정 순서에 있는 위치의 좌표 값들을 일부분 수정하는 연산이다. 부분 수정 연산을 위한 로그레코드는 수정된 위치 좌표 값들의 순서와 개수, 그리고 부분 수정된 점들의 좌표 값을 저장한다. 만약 수행된 수정 연산이 앞에서 기술한 모든 연산의 분류에 속하지 않으면 수정된 객체의 모든 좌표 값을 저장한다.

4.3 최근 강제 로깅 상태의 읽기 집합

모바일 트랜잭션 T_j를 장애로부터 회복하기 위하여 먼저 T_j의 변경 로그를 이용하여 T_j의 초기 읽기 집합으로부터 최근 검사점 상태의 읽기 집합을 생성하는 것이 필요하다.

CopiedRegion(T_j)를 T_j의 시작 시 T_j가 모바일 클라이언트의 작업 영역으로 복사한 공간 데이터의 복사영역이라 하고 $o_j.G$를 객체 o_j의 기하 속성이라 할 때 T_i의 읽기 연산에 의한 초기 읽기 집합, Initial_RS(T_j),은 다음과 같이 정의한다.

정의 1: For$\exists o_j$, Initial_RS(T_j) =
$\{o_j | \text{CopiedRegion}(T_j).G \cap o_j.G \neq \varnothing r(T_j) \rightarrow oj\}$
where $1 \leq j \leq k$

T_j의 변경 로그를 적용하는 방법은 크게 연산의 수행 장소에 따라 두 가지가 있다. 첫 번째는 공간 데이터베이스에서 CopiedRegion(T_j)를 이용하여 검색한 Initial_RS(T_j)과 변경 로그를 모바일 클라이언트의 작업 영역으로 전송하여 클라이언트의 작업 영역에서 최근 검사점 상태의 읽기 집합을 생성하는 방법이다. 두 번째는 서버에서 Initial_RS(T_j)에 로그의 변경 객체를 순서대로 병합하여 최근 검사점 상태의 읽기 집합을 생성하는 방법이다. 무선망은 유선망에 비해 대역폭이 좁고 통신비용이 많이 들며 모바일 클라이언트는 서버나 기존의 클라이언트에 비해 계산 능력이 떨어지고 저장 공간이 적다. 따라서 클라이언트로 전송할 데이터 양을 줄이고 클라이언트 연산을 최소화하기 위하여 두 번째 방법이 적합하다.

〈그림 7〉은 공간 데이터베이스로부터 CopiedRegion(T_j)를 이용하여 검색한 초기 읽기 집합에 변경 로그의 객체를 순서대로 적용하여 최근 강제 검사점 상태의 읽기 집합을 생성하는 알고리즘을 보여준다. 〈그림 7〉의 알고리즘은 크게 서버의 로그에서 T_j의 시작 위치를 결정하는 단계와 로그의 객체를 순서대로 병합하는 단계로 구성된다. 첫 번째 단계에서 트랜잭션 매니저는 서버에 저장되어 있는 전역 로그를 역방향으로 스캔하여 시작 위치를 결정한다. 시작 위치는 〈T_j, start〉 로그레코드이다. 두 번째 단계에서 로그를 정방향으로

스캔하면서 T_j의 식별자를 이용하여 T_j의 로그들을 검색한다. 만약 T_j의 로그레코드가 검색되면 로그레코드의 객체를 초기 읽기 집합에 병합한다. 즉, 만약 로그레코드의 객체와 초기 읽기 집합의 객체가 같다면 초기 읽기 집합의 객체를 삭제하고 로그레코드의 객체를 병합한다.

```
Procedure Apply_log (Tj, Initial_Read_Set)
Begin
    log_record ← a;              //한 로그 레코드
    log_obj ← a;                 //로그레코드의 변경 객체
    flag ← false;        //의존 객체 집합의 한 변경 객체와의 동일 여부

    Recently_Logged_Read_Set ← Initial_Read_Set;

    WHILE scan GLOBAL_LOG backward
        log_record ← read LOG;
        IF log_record = <Tj start> THEN
            set start position on GLOBAL_LOG;
            EXIT LOOP
        END IF
    END LOOP
    WHILE scan GLOBAL_LOG forward UNTIL end of GLOBAL_LOG
        IF log_record.t_id = Tj THEN
                log_obj ← read a object from log_record;
                FOR all oi in Recently_Logged_Read_Set
                        IF oi.oid = log_obj.oid THEN
                        delete oj from Recently_Logged_Read_Set;
                Recently_Logged_Read_Set ← log_obj;
            END IF
            flag ← false;
        END IF
    END LOOP
END
```

〈그림 7〉 최근 로깅 시점 상태 읽기 집합 생성 알고리즘

T_j의 로그에 속하는 객체의 집합을 Log_Objects(T_j)라 할 때 〈그림 7〉의 알고리즘에 따라 최근 검사점 상태의 읽기 집합은 다음과 같이 정의된다.

정의 2: $\text{Recently\_Logged\_RS}(T_j) = (\text{Initial\_RS}(T_j) - (\text{Initial\_RS}(T_j) \cap \text{Log\_Objects}(T_j))) \cup \text{Log\_Objects}(T_j)$

5. 외래충돌가능객체를 이용한 회복 기법

이 장에서는 모바일 트랜잭션 Tj가 회복할 때 회복 시점보다 먼저 완료된 다른 트랜잭션의 쓰기 집합에서 외래충돌가능객체를 검색하는 회복 기법에 대하여 기술한다.

5.1 외래충돌가능객체

CopiedRegion(T_j)를 T_j의 복사영역이라 하고 o_i.G를 객체 o_i의 기하 속성이라 할 때 T_j의 쓰기 연산에 의해 변경된 객체들의 집합을 쓰기 집합, WS(T_j),이라 하며 다음과 같이 정의한다.

정의 3: $\text{For} \exists o_j,\ \text{WS}(T_j) = \{o_i | \text{CopiedRegion}(T_j).G \cap o_i.G \neq \emptyset \wedge w(T_j) \rightarrow o_i\}$ where $1 \leq i \leq k$

T_j에 장애가 발생하여 회복 작업을 수행할 때, 회복 관리자는 T_j의 회복 시점 전에 완료한 트랜잭션들의 쓰기 집합에서 의존 객체를 검색하기 위하여 쓰기 집합의 트랜잭션을 먼저 결정해야 한다. 이러한 트랜잭션을 T_j의 충돌가능트랜잭션(potentially conflictive transaction)이라 한다.

T_j의 충돌가능트랜잭션은 다음 두 조건을 만족하는 트랜잭션이다. 첫 번째는 충돌가능트랜잭션은 반드시 T_j의 복사영역과 중첩되는 영역의 객체를 변경해야 한다. 서로 떨어진 복사영역의 객체를 변경하는 트랜잭션들간은 서로 충돌하지 않기 때문이다. 두 번째는 충돌가능트랜잭션은 T_j의 시작과 회복사이에 완료되어야 한다. 왜냐하면 장애가 발생한 T_j가 회복할 때 충돌가능트랜잭션의 변경 객체를 읽어야 하기 때문이다. 두 조건을 이용하여 T_j의 충돌가능트랜잭션, Potentially_Conflictive_Transactions(T_j),은 다음과 같이 정의될 수 있다.

정의 4: $\text{For} \exists T_i,\ \text{Potentially\_Conflictive\_Transactions}(T_j) = \{T_i | t_s(T_j) \langle t_c(T_i) \langle t_r(T_j) \text{CopiedRegion}(T_j).G \cap \text{CopiedRegion}(T_i).G \neq \emptyset\}$ where $i \neq j$

T_j의 외래충돌가능객체는 충돌가능트랜잭션의 쓰기 집합에 속하는 객체들 중에서 T_j가 변경한 객체들과 충돌할 가능성이 있는 객체이다. 만약 충돌가능트랜잭션이 변경한 객체 o_i가 T_j의 변경 영역과 Disjoint이외의 공간 관련성을 가지면 o_i와 T_j가 변경한 객체 oj 간에 변경 충돌이 발생할 수 있다[1]. T_j의 외래충돌가능객체는 다음과 같이 정의된다.

정의 5: $\text{Foreign\_Conflictive\_Objects}(T_j) = \{o_i | \text{CopiedRegion}(T_j).G \cap o_i.G \neq \emptyset \wedge o_i \in \text{WS}(T_i)\}$ where $T_i \in \text{Potentially\_Conflictive\_Transactions}(T_j)$ and $1 \leq k \leq n$

〈그림 8〉은 T_j의 외래충돌가능객체를 검색하기 위

한 알고리즘이다. 먼저 정의 4에 따라 T_j의 시작 타임스탬프와 회복 타임스탬프 사이에 완료되고 T_j의 복사 영역과 중첩된 영역을 변경한 충돌가능트랜잭션을 검색한다. 검색된 충돌가능트랜잭션의 리스트를 이용하여 각 충돌가능트랜잭션의 쓰기 집합에 속하는 객체들 중에서 정의 5에 따라 T_j의 변경 영역과 Disjoint이외의 공간 관련성을 가지는 객체를 검색한다.

```
Procedure search_foreign_conflictive_object_set( Tj )
Begin
    potentially_conflictive_transactions ← ø;        //의존 트랜잭션 집합
    foreign_conflictive_objects ← ø;                 //의존 객체 집합
    FTIR ← ø;                                        //트랜잭션 정보 테이블
    TIR ← get transaction information of Tj;         //장애 발생한 트랜잭션의 정보

    //의존 트랜잭션 검색
    FOR all Ti where Ti ≠ Tj
        FTIR ← read the transaction information of Ti from the Catalog;
        IF TIR.ts < FTIR.tc < TIR.tf AND FTIR.CopiedRegion.G ∩ TIR.CopiedRegion.G ≠ ø
          THEN
            potentially_conflictive_transactions ← add TIR;
        END IF
    END LOOP
    //의존 객체 검색
    FOR each Ti where Ti ∈ potentially_conflictive_transactions
        FTIR ← read the transaction information of Ti;
        FOR each oi where oi ∈ FTIR.WS
            IF TIR.CopiedRegion.G ∩ oi.G ≠ ø THEN
                foreign_conflictive_objects ← add oi;
            END IF
        END LOOP
    END LOOP
End
```

〈그림 8〉 외래충돌가능객체 검색 알고리즘

5.2 모바일 트랜잭션 회복 알고리즘

회복된 트랜잭션 T_j의 사용자가 T_j를 완료하기 전에 충돌가능트랜잭션 T_i와 충돌 가능한 객체를 재변경하기 위해서는 T_i의 변경 객체인 외래충돌가능객체를 볼 수 있어야 한다. 이를 위해서 회복된 T_j의 읽기 집합에 외래충돌가능객체를 포함시킨다. 즉, 회복된 트랜잭션 T_j의 읽기 집합은 최근 검사점 상태의 읽기 집합과 T_j의 외래충돌가능객체로 구성된다. 따라서 회복된 T_j의 읽기 집합, Recovered_RS(T_j),은 다음과 같이 정의된다.

정의 6: Recovered_RS(T_j) = Recently_Logged_RS(T_j) ∪ Foreign_Conflictive_Objects(T_j) where $1 \le j \le k$

모바일 트랜잭션에 장애가 발생했을 때 회복 연산의 수행 알고리즘은 다음과 같다. 〈그림 9〉는 서버 측의 회복 연산 알고리즘을 보여준다. 먼저 서버의 회복 매니저는 모바일 트랜잭션의 식별자를 이용하여 회복 연산에 필요한 초기 읽기 집합을 검색한다. 트랜잭션의 로그를 이용하여 최근 검사점 상태의 읽기 집합을 생성한 후에 트랜잭션 식별자를 이용하여 해당 트랜잭션의 외래충돌가능객체들을 검색한다. 외래충돌가능객체의 검색이 완료되면 장애가 발생한 클라이언트로 트랜잭션 식별자, 최근 검사점 상태의 읽기 집합, 외래충돌가능객체들을 반환한다.

```
Procedure SVR_Recovery_Procedure( mc_id );
Begin
    Initial_Read_Set ← ø   // 초기 읽기 집합

    //회복 정보 검색
    t_id ← read Catalog with mc_id;
    ts, tc ← read Catalog with t_id;
    TIR ← read Catalog with t_id;
    Initial_Read_Set ← read DB with t_id, FTIR.CopiedRegion.geometry

    //최근 강제 로깅 상태 읽기 집합 생성
    Recovered_Read_Set ← Apply_Log( t_id, Initial_Read_Set );

    //피충돌가능객체 검색
    Foreign_Conflictive_Objects ← search_committed_conflictive_object_set( t_id );

    //회복 응답
    send a message reply_recovery(t_id, Recovered_Read_Set, committed_conflictive_objects)
    to mobile client;
End
```

〈그림 9〉 서버 측의 회복 연산 알고리즘

〈그림 10〉은 모바일 클라이언트측의 회복 연산 알고리즘을 보여준다. 모바일 클라이언트의 회복 매니저는 서버로 회복 요청을 보낸 후 서버로부터 최근 검사점 상태의 읽기 집합과 외래충돌가능객체 그리고 트랜잭션 식별자를 포함한 응답 메시지를 기다린다. 메시지가 도착하면 트랜잭션을 다시 시작하고 최근 검사점 상태의 읽기 집합과 외래충돌가능객체를 읽어서 화면에 디스플레이한다. 그리고 외래충돌가능객체를 표시한다.

```
Procedure MC_Recovery_Procedure( mc_id );
BEGIN
    //서버에 회복 요청
    send a message "request_reovery( mc_id )" to server;

    wait for message "reply_recovery(t_id, Recovered_Read_Set,
    committed_conflictive_objects )";
    IF response "reply_recovery" IS EXIST THEN
        restart transaction with t_id;
        display Recovered_Read_Set;
        highlight committed_conflictive_objects;
    END IF
END
```

〈그림 10〉 모바일 클라이언트측의 회복 연산 알고리즘

6. 회복 시스템의 설계 및 구현

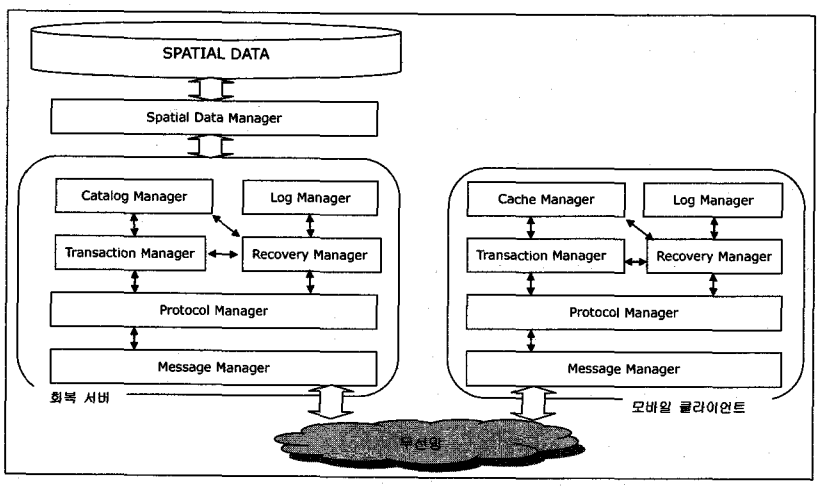
6.1 시스템 구조

〈그림 11〉은 이 논문에서 제시한 강제 로깅 방법과 외래충돌가능객체 기반의 회복 기법을 지원하는 회복 프로토타입 시스템의 시스템 구조도를 보여준다. 서버는 리눅스 시스템을 기반으로 한 공간 데이터 서버 상에서 수행되며 모바일 클라이언트는 Compaq Ipaq의 Windows CE 3.0을 기반으로 수행된다. 안정저장장치로의 공간 데이터 및 회복 데이터 접근을 위한 공간 데이터 서버는 사이버맵 데이터 서버를 이용한다.

회복 프로토타입 시스템에서 강제 로깅 기법과 외래충돌가능객체를 반영하는 회복 기법이 현실화될 수 있도록 설계하고 구현하는 것이 가장 중요하다. 〈그림 11〉의 시스템 구조에서 강제 로깅 기법과 외래충돌가능객체 기반의 회복 기법은 다음과 같이 수행된다.

외래충돌가능객체 기반의 회복 기법: 모바일 클라이언트에서 수행되는 모바일 트랜잭션에 장애가 발생하면 클라이언트의 프로토콜 관리자는 트랜잭션 식별자와 함께 회복 요청 메시지를 서버로 전송한다. 서버의 프로토콜 관리자는 회복 요청 메시지를 받은 후 트랜잭션 식별자를 이용하여 카탈로그 관리자로부터 초기 읽기 집합, 복사영역, 그리고 타임 스탬프 정보를 검색하고 복사영역과 타임 스탬프를 이용하여 회복 관리자에게 회복 요청을 한다. 회복 관리자는 트랜잭션 식별자와 복사영역, 타임 스탬프를 이용하여 로그 관리자에게 해당 로그의 검색을 요청한다. 로그 검색이 완료된 후에 회복 관리자는 초기 읽기 집합에 변경 로그를 순차적으로 적용하여 최근 검사점 상태 읽기 집합을 생성한 후 외래충돌가능객체들을 검색한다. 그리고 외래충돌가능객체와 최근 검사점 상태 읽기 집합을 프로토콜 관리자에게 보낸다. 프로토콜 관리자는 트랜잭션 식별자, 최근 검사점 상태 읽기 집합, 그리고 외래충돌가능객체들로 회복 응답 메시지를 구성하여 메시지 관리자를 통해 장애가 발생한 클라이언트에게 전송한다. 서버로부터 회복에 필요한 데이터를 전송받은 모바일 클라이언트의 프로토콜 관리자는 최근 검사점 상태 읽기 집합과 외래충돌가능객체들을 캐쉬 관리자에게 보내고 회복 관리자에게 회복 요청을 한다. 회복 관리자는 트랜잭션 식별자를 이용하여 트랜잭션 관리자에게 트랜잭션을 회복되었음을 알린다.

강제 로깅 기법: 모바일 클라이언트의 회복 관리자는 공간 객체가 변경될 때마다 로그레코드카운트를 1씩 증가시킨다. 로그레코드카운트가 최대로그레코드카운트와 같아지면 회복 관리자는 로그 관리자로부터 서버의 안정저장장치에 저장할 로그레코드를 검색하고 프로토콜 관리자에게 강제 로깅을 요청한다. 프로토콜 관리자는 메시지 관리자를 통해 서버에 연결을 설정한 후에 로그레코드와 트랜잭션 식별자로 강제 로깅 요청 메시지를 구성하여 서버에 전송한다. 메시지를 받은 서버의 프로토콜 관리자는 회복 관리자에게 트랜잭션 식별자와 로그레코드를 보내어 강제 로깅하도록 한다. 회복 관리자는 로그레코드를 로그 관리자를 통해 안정저장장치에 저장한 후 프로토콜 관리자에게 강제 로깅이 성공적으로 수행되었음을 클라이언트에게 응답한다.



〈그림 11〉 모바일 트랜잭션의 회복 프로토타입 시스템 구조도

6.2 구현 예

이 절에서는 건물 객체를 변경하는 두 모바일 트랜잭션에 장애가 발생하였을 때 강제 로깅한 로그레코드와 외래충돌가능객체를 이용하여 회복하는 예를 보여준다. 최대로그카운트는 3으로 설정하였다.

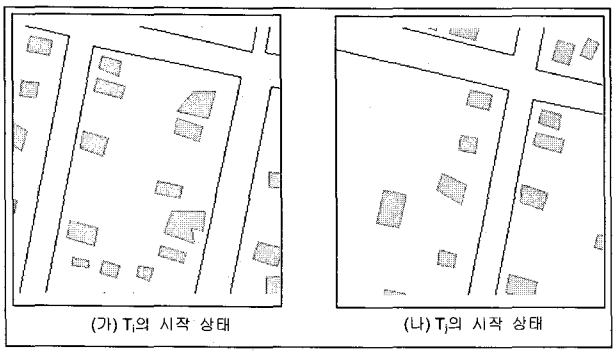
〈그림 12〉는 두 모바일 트랜잭션 T_i와 T_j의 시작 상태를 보여준다. 트랜잭션은 $t_s(T_j)\langle tc(Ti)\langle t_r(T_j)$의 순으로 설정하였다. 〈그림 12〉에서 보듯이 T_i와 T_j가 변경하고자 하는 영역은 중첩되어 있다.

〈그림 13〉은 T_i가 공간 객체를 변경하는 예를 보여준다. 〈그림 13〉에서 보듯이 T_i는 두 개의 건물 객체를 변경한 후에 충돌하는 트랜잭션이 없기 때문에 성공적으로 완료되었다.

〈그림 14〉는 T_j가 공간 객체를 변경하는 예를 보여준다.〈그림 14〉에서 보듯이 T_j는 세 개의 건물 객체를 변경하였다.

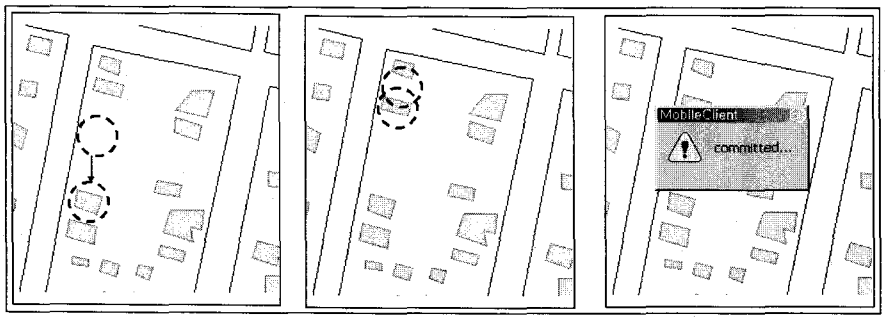
〈그림 15〉은 T_j가 강제 로깅을 수행한 후에 장애가 발생한 예를 보여준다. T_j의 최대로그카운트가 3으로 설정되어 있고 〈그림 14〉처럼 세 개의 건물 객체를 변경하였기 때문에 T_j는 〈그림 15〉(가)에서 보듯이 강제 로깅을 수행하고 변경 로그를 서버의 안정저장장치에 저장한다. 강제 로깅을 수행한 후에 T_j는 〈그림 15〉(나)처럼 한 개의 건물 객체를 더 변경하였으나 〈그림 15〉(다)처럼 장애가 발생하였다.

장애가 발생한 T_j는 회복 연산을 시작하고 서버에 회복 요청을 한다. $t_s(T_j)\langle t_c(T_i)\langle t_f(T_j)$이고 $CopiedRegion(T_j).G \cap CopiedRegion(T_i).G \neq \varnothing$이기 때문에 T_i는 충돌가능트랜잭션이 된다. 서버의 회복 관리자는 먼저 T_j의 로그레코드들을 초기 읽기 집합에 적용하여 최근 검사점 상태 읽기 집합을 생성한다. T_i의 변경 객체들 중에서 외래충돌가능객체를 검색한 후에 서버는 T_j의 클라이언트에게 외래충돌가능객체와 최근 검사점 상태 읽기 집합을 전송하고 클라이언트의 회복 관리자는 〈그림 16〉(가)처럼 T_j를 다시 시작한다. 이 때 외래충돌가능객체와 T_j가 변경한 객체 간에 변경 충돌이 발생할 수 있기 때문에 T_j는 변경 충돌이 해소될 수 있도록 〈그림 16〉(나)처럼 충돌이 발생할 수 있는 객체를 재변경한다. 모든 객체에 대한 변경을 완료한 T_j는 T_i와의 변경 충돌을 이미 해소하였기 때문에 〈그림 16〉(다)처럼 성공적으로 완료된다.

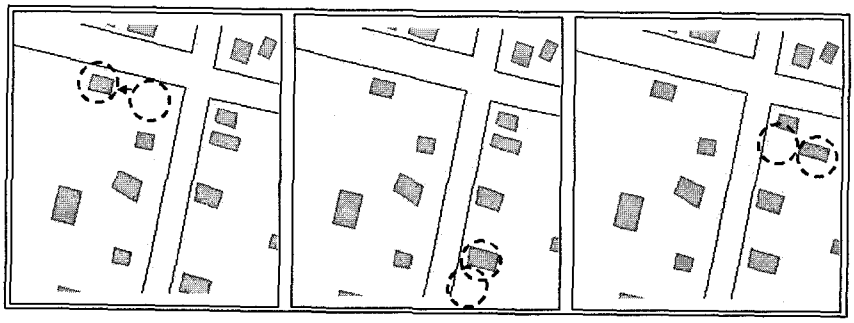


(가) T_i의 시작 상태 (나) T_j의 시작 상태

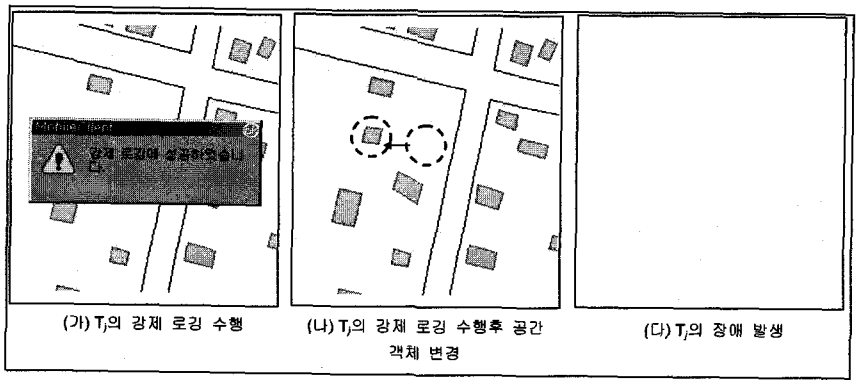
〈그림 12〉 두 모바일 트랜잭션 T_i와 T_j의 시작 상태



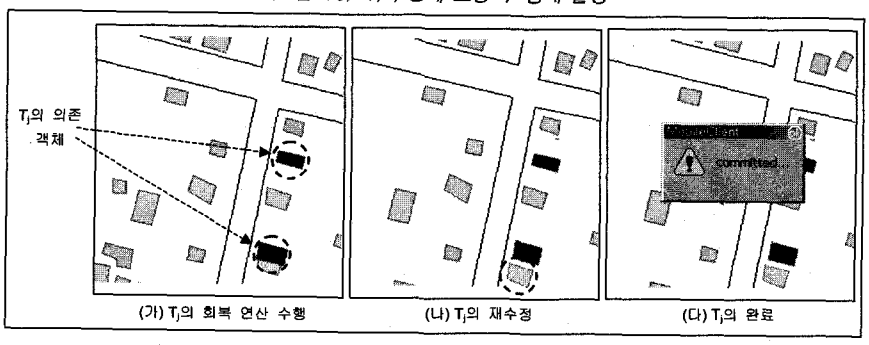
〈그림 13〉 T_i의 공간 객체 변경 후 완료



〈그림 14〉 T_i의 공간 객체 변경



〈그림 15〉 T_i의 강제 로깅 후 장애 발생



〈그림 16〉 T_i의 회복 후 완료

7. 결론

모바일 클라이언트에서 공간 데이터를 변경하는 모바일 트랜잭션은 단절 상태하에서 사용자 대화 방식으로 수행되는 긴 트랜잭션이다. 장애로부터 모바일 트랜잭션이 회복할 때 만약 회복 시점보다 먼저 완료된 다른 트랜잭션이 중첩 지역에서 변경한 객체가 있으면 해당 객체를 읽을 필요가 있다. 그러나 단절된 모바일 트랜잭션은 다른 트랜잭션의 쓰기 집합을 알 수 없고 자신의 로그레코드만을 이용하여 회복하기 때문에 동시 변경된 객체간에 비일관적인 공간관련성이 생성될 수 있다. 만약 회복된 긴 트랜잭션이 변경 충돌로 인해 철회되면 회복된 데이터를 포함한 모든 변경 데이터는 취소되기 때문에 변경 작업을 완료하기 위해서 트랜잭션을 재시작하고 모든 변경 연산을 다시 수행해야 하는 고비용이 요구된다.

이 논문의 목표는 모바일 트랜잭션을 회복할 때 동시 수행된 다른 트랜잭션의 쓰기 집합에 속하는 객체

들을 읽음으로써 충돌 가능한 객체를 재변경하는 것이다. 이를 위하여 먼저 완료된 다른 트랜잭션의 쓰기 집합에서 외래충돌가능객체를 검색하는 회복 기법을 제시하였다. 외래충돌가능객체는 회복 시점보다 먼저 완료된 다른 트랜잭션에 의해 변경된 객체로써 회복된 트랜잭션의 변경 객체와 충돌할 가능성이 있는 객체이다. 회복된 트랜잭션이 최근 검사점 상태의 읽기 집합과 함께 외래충돌가능객체를 읽기 때문에 사용자는 충돌이 해소될 수 있도록 객체를 재변경할 수 있다. 또한 최근 검사점 상태의 읽기 집합을 생성하기 위하여 강제 로깅 기법을 제시하였다. 강제 로깅 기법은 주기적으로 모바일 트랜잭션이 서버에 강제로 접속하여 서버의 안정저장장치에 로그레코드를 저장한다.

이 논문에서 제안한 회복 기법은 먼저 완료된 트랜잭션의 쓰기 집합에서 검색된 외래충돌가능객체를 회복된 트랜잭션의 읽기 집합에 포함하기 때문에 회복된 트랜잭션을 완료하기 전에 트랜잭션의 변경 충돌을 해소할 수 있다. 따라서 완료 연산 수행 시 변경 충돌로 인한 긴 트랜잭션의 철회를 줄일 수 있는 장점이 있다.

제안한 회복 기법은 모바일 클라이언트-서버 환경에서 모바일 클라이언트의 시스템 장애만을 고려하였다. 향후 연구로는 모바일 클라이언트의 트랜잭션 오류, 미디어 장애 등을 고려한 모바일 트랜잭션 회복 기법의 연구가 필요하다.

참고 문헌

[1] 김동현, 홍봉희, "모바일 컴퓨팅 환경을 위한 재수행 트랜잭션 모델의 설계 및 구현", 정보과학회 논문지, Vol. 30, No. 2, 2003, pp. 176-196

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

[3] Field Solutions Technical White Paper, Tadpole Technology, Inc.

[4] Dong Hyun Kim, Bong Hee Hong, Byunggu Yu, Eun Suk Hong, "Validation-Based Reprocessing Scheme for Updating Spatial Data in Mobile Computing Environments", Int. Conf. on advanced information networking and application, 2003, pp.211-214.

[5] H. F. Korth, G. D. Speegle, "Long-Duration Transaction in Software Design Projects", Proc. of Int. Conf. on Data Engineering, 1990, pp. 568-574.

[6] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Database System Concepts, 1996

[7] Franklin, M., Zwilling, M., Tan, C.K., Carey, M., DeWitt, D., "Crash Recovery in Client-Server EXDOUS", Proc. of ACM SIGMOD, 1992.

[8] C. Mohan, Inderpal Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", Proc. of ACM SIGMOD, 1994, pp.55-66.

[9] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems", ACM Transactions on Computer Systems, Vol. 3, No. 3, 1985, pp. 204-226.

[10] N. Neves and W. K. Fuchs "Adaptive Recovery for Mobile Environments", Communications of the ACM, Vol. 40, No.1, 1997.

[11] James J Kistler, M Satyanarayanan, "Disconnected Operation in the CODA file system", ACM Transactions on Computer Systems, Vol. 10, No.1, 1992, pp. 3-25

[12] Gail E. Kaiser, "Cooperative Transactions for Multiuser Environments", Modern Database Systems, 1995, pp.409-433.

[13] Gail E. Kaiser, "Cooperative Transactions for Multiuser Environments", Modern Database Systems, 1995, pp.409-433.

[14] A. Reuter, Transaction Processing, 1993.

[15] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, Patrick O'Neil, "A Critique of ANSI SQL Isolation Levels", Proc. of ACM SIGMOD, 1995.

[16] Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha, "The Dangers of Replicationand a Solution", Proc. of ACM SIGMOD, 1996, pp. 173-182.

[17] Sanjay Kumar Madria, Bharat Bhargava, "A Transaction Model for Mobile Computing", Int. Database Engineering and Application Symposium, 1998, pp.92-102.

김동현

1995년 부산대학교 컴퓨터공학과 졸업(공학사)

1997년 부산대학교 대학원 컴퓨터공학과(공학석사)

2003년 부산대학교 대학원 컴퓨터공학과(공학박사)

관심분야: 지리정보시스템, 모바일 GIS, 모바일 트랜잭션, 이동체 색인

강주호

2001년 부경대학교 전자컴퓨터정보통신공학부 졸업(공학사)

2003년 부산대학교 대학원 컴퓨터공학과(공학석사)

홍봉희

1982년 서울대학교 컴퓨터공학과 졸업(공학사)

1984년 서울대학교 대학원 컴퓨터공학과(공학석사)

1988년 서울대학교 대학원 컴퓨터공학과(공학박사)

1987년 4월~현재 부산대학교 공과대학 컴퓨터공학과 교수

관심분야: 데이터베이스, 공간 데이타베이스, 이동체 데이타베이스, 이동체 색인, 트랜잭션