MTS기반 트랜잭션 시스템의 안전성 검증

윤일철[+] ○ 김형호[+] 배두환[+]
[+]한국과학기술원 전산학과

Reliability Verification on MTS-based Transaction System

Yoon, Il-Chul[+] ○ Kim, Hyung-Ho[+] Bae, Doo-Hwan[+]
[+]Dept. of Computer Science, KAIST

요 약

컴포넌트 기반 소프트웨어 개발(CBD)은 점차 대중화되어 가고 있는 소프트웨어 개발 방법이며, 최근에는 여러 미들웨어들이 서버 컴포넌트를 이용한 CBD를 지원하고 있다. 본 논문에서는 Microsoft Transaction Server(MTS)를 이용하여 서버 컴포넌트를 기반으로 안전한 트랜잭션 시스템을 디자인하고 검증하는 방법을 제시한다. 위의 목적을 위해 우선 MTS를 이용한 트랜잭션 시스템을 정형화하고, 정형화된 프레임워크를 이용해, 디자인한 시스템이 명세를 만족하는지 검증해 볼 것이다. 또, 위의 프레임워크를 위해 MSDN에 있는 은행 시스템을 예로 들어서 그 시스템이 만족해야 하는 명세를 위배한다는 것을 보이고, 명세를 위배하지 않도록 시스템의 설계를 변경할 것이다.

우리는 MTS에서 컴포넌트가 트랜잭션에 참여하는 방식이 시스템의 확장성과 안전성에 영향을 줄 수 있다고 생각하며, 그렇기 때문에, 우리가 제시한 프레임워크가 트랜잭션 시스템의 안전성과 확장성을 평가하는데 있어 유용할 것으로 기대한다.

1. 서론

컴포넌트 기반 소프트웨어 개발은 현재 부상중인 소프트웨어 개발 방법이다. 최근에는 Enterprise JavaBeans(EJB)나 Microsoft Transaction Server(MTS) 같은 상용 미들웨어들이 서버 컴포넌트들을 이용한 CBD를 지원하기 시작했다. MTS[2]나 EJB[6]는 각각 다른 회사에서 개발되었지만 서버 컴포넌트의 보안이나 트랜잭션 관리의 문제들에 대한 비슷한 해답을 제시하고 있다. 본 논문에서는 MTS에서의 트랜잭션 관리에 중점을 두고 MTS를 이용해 설계한 컴포넌트 기반 트랜잭션 시스템의 안전성을 검증하는 방법을 소개한다.

트랜잭션[1]의 개념은 메임프레임 구조에서부터 근래의 클라이언트/서버 구조나, n-tier구조에 이르기까지 데이타의 무결성을 위한 중요한 기술로 인식되고 있다. 여러 커맨드들을 개념적으로 하나의 커맨드처럼 다루어, 원자성(atomicity)를 보장하기 때문에, 트랜잭션 기술은 데이터의 일관성을 유지하면서, 동시성이 있는 요구들을 효율적으로 다루는 데 유용하다.

어떤 트랜잭션이 수행될 때, 그 트랜잭션은 다른 트랜잭션으로부터 자신이 사용하는 시스템 자원을 보호하기 위해 락을 걸고 그 자원을 필요로 하는 다른 트랜잭션들은 그 자원을 사용하고 있는 트랜잭션이 락을 해제할 때까지 기다리게 된다. 그러므로, 락을 얼마나 오랫동안 걸고 있을 것인가의 문제는 효율적인 시스템의 구축을 위해 중요한 이슈가 된다. 트랜잭션 경계가 너무 넓어서 시스템 자원을 너무 점유하는 경우에는 비효율적이고, 확장성이 떨어지는 시스템이 될 가능성이 있는 반면에, 너무 트랜잭션 경계가 좁을 경우에는 시스템이 관리하는 데이터베이스의 무결성이 깨져서 트랜잭션 시스템의 안전성이 낮아질 수 있다. 그러므로, 확장성과 안전성을 주의깊게 고려하여 트랜잭션 경계를 정할 필요가 있다. 트랜잭션의 안전성에 관한 연구는 70년대부터 주로 데이터베이스 영역에서 연구되어 왔는데, 데이터베이스의 무결성을 보장하기 위한 safe transaction에 대한 연구는 본 논문에서 추구하는 시스템 명세의 만족에 비해 좁은 범위를 다루고 있지만, 결국 시스템의 요구사항을 만족시키려는 시도라는 측면에서 상통하는 부분이 있다. 특히, Qian과 Waldinger[4]는 그들이 개발한 트랜잭션 로직을 이용하여 트랜잭션을 기술함으로서 어떤 트랜잭션이 데이터베이스의 무결성을 유지하는지 검증할 수 있음을 보여주었다. 그 후, Sheard와 Stemple[5]는 safe transaction을 컴파일시에 자동으로 검증하는 툴을 개발하였다. 우리의 연구는 로직을 이용하여 데이터베이스의 무결성을 검증하려는 위와 같은 연구들과 방향을 같이 하지만, 컴포넌트들이 트랜잭션에 참여할 때, 그 트랜잭션이 시스템의 명세를 만족하는지를 검증한다는 점에서 차이가 있다.

MTS에서는 컴포넌트에 트랜잭션 속성을 선언하는 방식으로 트랜잭션 경계를 정한다. 그러므로, 컴포넌트들이 조합되는 방식과, 각 컴포넌트들이 가지는 트랜잭션 속성에 따라서 트랜잭션 경계는 달라질 수 있다. 예를 들어, 어떤 컴포넌트의 트랜잭션 속성이 tx_new로 설정되어 있다면 그 컴포넌트가 생성될 때 새로운 트랜잭션이 하나 만들어진다는 것을 의미한다.

컴포넌트 시스템에서 트랜잭션을 고려하여 시스템을 설계하기 위해서, 우리가 개발한 프레임워크에서는 로직을 이용한 정형화된 시스템의 명세와 설계 방법을 제안하고, 만들어진 시스템 설계를 가지고 그 설계가 MTS의 트랜잭션을 이용하여 구현될 경우, 명세를 만족하는지 확인할 수 있는 알고리즘을 제시할 것이다. 본 논문에서 제안한 프레임워크는 기본적으로 MTS를 가정하고 있으나, EJB같은 미들웨어 기술에도 적용해 볼 수 있을 것으로 생각한다.

2. MTS의 개요

MTS는 DCOM에 바탕을 둔 트랜잭션 관리기능을 지원하는 미들웨어로, MTS 실행부, 리소스 관리자, Microsoft 분산 트랜잭션 관리자(MSDTC), MTS 탐색기로 구성된다[2]. 간략히 구성요소들에 대해 살펴보면 아래와 같다.

MTS 실행부는 런타임에 생성되는 MTS객체들에 대해 쓰레드와 트랜잭션 컨텍스트를 관리해주는 백그라운드 프로세스이다. 리소스 관리자는 데이터베이스와 같은 지속성을 가지는 자원을 관리해주는 구성요소로서 커밋이나 롤백같은 작업을 하위레벨에서 처리하며, 트랜잭션의 요건인 ACID를 만족시키기 위하여 MSDTC와 연동하여 2단계 커밋 프로토콜을 이용해 작동한다. MSDTC는 올바른 트랜잭션 처리를 위해 어떤 트랜잭션에 등록된 리소스 관리자들을 감독하는 역할을 한다. 또한, 어떤 트랜잭션이 커밋될 필요가 있을 경우, 2단계 커밋을 이용하여 등록된 리소스 관리자들 간에 atomic 트랜잭션을 보장한다. MTS 탐색기는 윈도우 시스템에서 작

동하는 그래픽 환경을 제공하는 관리툴이다. 이 툴을 이용하여, 컴포넌트를 MTS에 등록하고 관리하는 작업을 할 수 있고, 각 컴포넌트의 트랜잭션 속성을 설정할 수 있도록 되어 있다.

성공적으로 트랜잭션을 수행하기 위해서는 위의 구성요소간에 긴밀한 협조가 이루어져야 하는데, 그림 1에서 간략히 MTS가 어떻게 작동하는가를 UML의 sequence diagram을 통해 보여주고 있다.

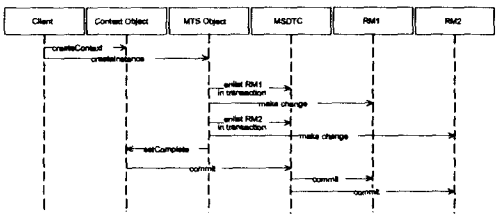

그림 1: MTS 작동 메커니즘

위에서도 언급했듯이, 각 컴포넌트는 트랜잭션 속성을 가진다. 그림 2는 MTS의 컴포넌트가 가질 수 있는 트랜잭션 속성에 대해 설명하고 있다. MTS컴포넌트는 생성되면서 설정되어 있는 트랜잭션 속성에 따라서 다르게 작동한다.

keyword	Description
tx_new	생성된 컴포넌트는 자신을 생성시킨 컴포넌트의 트랜잭션 존재 유무에 상관없이 새로운 트랜잭션 컨텍스트를 생성한다.
tx_req	생성된 컴포넌트는 자신을 생성시킨 컴포넌트의 트랜잭션에 참여해 작동한다. 부모 컴포넌트가 트랜잭션을 가지고 있지 않은 경우에는 새로운 트랜잭션을 생성한다.
tx_sup	생성된 컴포넌트는 컴자신을 생성시킨 포넌트의 트랜잭션에 참여해 작동한다. 부모 컴포넌트가 트랜잭션을 가지고 있지 않은 경우에는 트랜잭션없이 수행된다.
tx_not	생성된 컴포넌트는 자신을 생성시킨 컴포넌트의 트랜잭션과 관계없이 트랜잭션이 참여하지 않고 수행된다.

그림 2: 컴포넌트의 트랜잭션 속성

3. 컴포넌트의 설계, 검증 프레임워크

이 섹션에서는 본 논문에서 제안한 시스템 설계, 검증 프레임워크에 관해 설명한다. 우선 시스템의 failure를 배제하고 시스템을 설계한 후에 트랜잭션을 고려하였을 때 failure에 대해 시스템이 어떤 결과를 보여줄 수 있는지 살펴볼 것이다.

트랜잭션이 처리된다는 것은 시스템이 어떤 상태에서 다른 상태로 전이된다는 것을 의미한다. 우리는 트랜잭션 처리에 관련된 시스템 명세를 표현하기 위해서 램포트가 action을 표현하기 위해 제안한 표기법[3]을 사용한다. 예를 들어서 $x' = x + 1$는 어떤 action이고, x는 변수, x'은 1만큼 증가된 x를 의미한다. 본 논문에서 action은 composite와 primitive action으로 구분할 수 있는데, primitive action은 더 작은 action들로 분리될 수 없는 action을 의미하고, 그 외의 action은 모두 composite action이다. composite action을 다루기 위해 invocation개념을, 그리고, action들의 순서있는 수행을 표현하기 위해 sequence개념을 프레임워크에 추가하였다.

invocation은 'component.methodname'으로 표기되는데, 궁극적으로는 여러 action으로 구성된다. 예를 들어 Transfer.transfer(a,b,amt)는 Credit.credit(a, amt);Debit.debit(b, amt)로 치환된다.

sequence는 sequence 연산자 ;로 action들을 묶어서 순서를 가지고 수행되는 action들을 표현한다. 하지만, 만일 서로 다른 변수에 접근하는 두 action이 sequence로 연결되어 있을 경우 그 두 action은 순서에 무관하며, 교집합으로 표현될 수 있다. 즉, A와 B가 전혀 다른 변수들에 접근한다면, $A;B \equiv B;A \equiv A \wedge B$라고 기술할 수 있다.

우리는 시스템이 만족해야 하는 명세를 primitive action의 sequence들로 기술하고, 위에서 설명한 방법으로 각 컴포넌트를 디자인할 것이다. 다음은 예제로 삼은 은행 시스템의 '예금이체'에 관한 명세이다. a 계좌에서 b 계좌로 amt만큼의 금액을 이체하는 과정에서 이체과정의 성공유무에 관계없이 영수증 번호를 1 증가시키도록 명세가 기술되었다. 하지만, 이체과정의 성공에도 불구하고 영수증 번호가 증가되지 않는 경우는 명세에서 제외되었다.

$$\begin{aligned} & ((a' = a - \text{amt}) \wedge (b' = b + \text{amt}) \wedge (R' = R + 1)) \\ \vee\ & ((a' = a) \wedge (b' = b) \wedge (R' = R + 1)) \\ \vee\ & ((a' = a) \wedge (b' = b) \wedge (R' = R)) \end{aligned}$$

각 컴포넌트의 설계는 위에서 언급한대로 action, invocation, sequence을 이용해서 하도록 한다. 다음은 우리가 설계한 은행시스템의 각 컴포넌트의 설계이다.

component	method	behavior
Account	credit(a, amt)	Credit.credit(a, amt); RcptPtr.RcptPrint()
Account	debit(a, amt)	Debit.debit(a,amt); RcptPtr.RcptPrint()
Account	traasfer(a, b, amt)	Tranfer.transfer(a, b amt); RcptPtr.RcptPrint()
Credit	credit(a, amt)	$a' = a + \text{amt}$
Debit	debit(a, amt)	$a' = a - \text{amt}$
Transfer	traasfer(a,b, amt)	Credit.debit(a, amt); Debit.credit(b, amt)
RcptPtr	RcptPrint()	$R' = R + 1$

그림 3: 예제 시스템의 구성 컴포넌트 설계

그림 3의 설계에 따라서 시스템은 아래의 그림 4과 같은 동작을 보일 것이다. 그림 4에서 볼 수 있듯이 시스템이 제대로 동작한다면, 원래 명세에

$$\begin{aligned} & \text{Account.transfer}(a, b, \text{amt}) \\ \equiv\ & \{\text{by the invocation}\} \\ & \text{Tranfer.transfer}(a, b, \text{amt}); \text{RcptPtr.RcptPrint}() \\ \equiv\ & \{\text{by the sequence}\} \\ & \text{Debit.debit}(a, \text{amt}); \text{Credit.credit}(b, \text{amt}); \\ & \text{RcptPtr.RcptPrint}() \\ \equiv\ & \{\text{by the invocation, three times}\} \\ & a' = a - \text{amt}; b' = b + \text{amt}; R' = R + 1 \\ \equiv\ & \{\text{by the property of independent sequence actions}\} \\ & a' = a - \text{amt} \wedge b' = b + \text{amt} \wedge R' = R + 1 \\ \rightarrow\ & \{\text{logical implication}\} \\ & (a' = a - \text{amt} \wedge b' = b + \text{amt} \wedge R' = R + 1) \\ & \vee (a' = a \wedge b' = b \wedge R' = R + 1) \\ & \vee (a' = a \wedge b' = b \wedge R' = R) \end{aligned}$$

그림 4: 설계에 따른 결과 추론과정

서 정의한 결과를 얻을 수 있다. 하지만, 현실적으로 분산환경에서 여러 리소스 관리자가 트랜잭션에 참여하고 있는 경우, 위와 같은 이상적인 경우만 발생하는 것은 아니다. 즉, failure가 발생할 가능성이 있다. 그렇기 때문에, 앞서 언급했듯이 시스템의 확장성과 안정성을 고려한 트랜잭션이 사용되어야 한다.

그림 5는 MTS에 등록된 컴포넌트의 트랜잭션 속성과 각 컴포넌트의 설계가 명세를 만족하는지를 검증하는 알고리즘이다. 알고리즘에서 #은 #

이 있는 시점에서 failure가 발생했음을 나타내고, #뒤에 발생한 action은 실제로는 발생할 수 없는 action임을 의미한다. 알고리즘에서 τ는 시스템 자원에 변화가 없었음을 나타내기 위해 사용되었다. 즉, $a;\tau$는 $\tau;a$, 그리고, a와 같은 의미이다. 또한, 알고리즘에서 append(A,B), drop(A), rollback(A), commit(A), unfold(a), type(a)를 사용하고 있는데, 알고리즘을 보다 잘 이해하기 위해 각각에 대해 간략히 설명하면 다음과 같다.

```
1   fun boundary (b, a)
2   case a of
3      primitive: return {#, a} //an atomic primitive action
4      sequence(a1; a2):
5         let
6            var fst = boundary(b, a1);
7            var snd = boundary(b, a2)
8         in
9            if b = null
10           then //there is no transaction boundary
11              return drop(append(fst, snd))
12           else
13              return rollback(append(fst, snd))
14        end
15     invocation:
16        if type(a) = tx_new ∨ (type(a) = tx_req ∧ b = null )
17        then //new transaction boundary is created
18           return commit(boundary(new_Tx() :: b, unfold(a)))
19        else
20           return boundary(b, unfold(a))
21  end
```

그림 5: 설계에서 결과를 추론하는 알고리즘

append(A,B)는 두 집합 A,B를 받아서 가능한 sequence를 만들어내는 함수이다. 즉, append$(\{a,b\},\{c,d\})$의 결과는 $\{a;c,a;d,b;c,b;d\}$가 된다. drop(A)와 rollback(A)는 결과를 간략화하기 위해 사용되는 함수이다. drop(A)는 A의 원소들 가운데 #이 포함된 원소에 대해 #뒤에 연결된 action들을 제거하는 함수로 트랜잭션 경계가 없이 시스템 자원이 변경된 경우를 처리하는 데 사용된다. rollback(A)는 drop(A)과 유사하지만, 트랜잭션 경계가 존재하는 경우에 사용되는데, #앞에 존재하는 action들이라도, #이 있는 시점에 커밋되지 않은 action들을 결과에서 제거한다. commit(A)는 집합 A의 결과 앞에 "c:" 태그를 붙이는 함수이다. 이 태그는 집합 A의 결과가 이미 커밋되었음을 나타낸다. 집합 A의 결과에 #이 포함되어 있었다면, 그것은 결과적으로 어떤 변화도 없었음을 의미하므로, τ로 변경된다. unfold(a)는 컴포넌트 설계 테이블에서 a를 찾아서 테이블에 정의된 sequence로 바꾸는 역할을 한다. 예를 들면, Credit.credit$(a$,amt$)$는 $a'=a$+amt로 변경되게 된다. 마지막으로 type(a)는 a가 invocation일 때 불려지는데, a에 기술된 컴포넌트의 트랜잭션 속성을 리턴하는 함수이다. 참고로, 알고리즘에서 tx_not은 컴포넌트에 설정하지 않는다고 가정하였다. 그림 6은 시스템을 구성하는 컴포넌트들에 트랜잭션 속성을 설정한 예이다.

component	transactional property
Account	tx_new
Credit	tx_req
Debit	tx_req
Transfer	tx_req
RcptPtr	tx_new

그림 6: 예제 시스템 구성 컴포넌트의 트랜잭션 속성설정 예

위에서 구성한 시스템 설계와 알고리즘, 그리고 트랜잭션 속성을 이용하여 시스템을 작동시키면, 시스템은 다음과 같은 결과를 보이게 된다.

{ τ,
c:($a'=a$-amt;$b'=b$+amt);τ,
c:($a'=a$-amt;$b'=b$+amt);c:($R'=R+1$) }

그리고, 유도된 결과집합에서 c:($a'=a$-amt;$b'=b$+amt);τ는 원래 명세에서 의도하지 않았던 결과임을 알 수 있다.

올바른 결과를 위해서 아래와 같이 시스템 설계를 약간 변경하면 올바른 결과가 나오는 것을 확인할 수 있다.

RcptPtr.RcptPrint(); if $r=$ success then Transfer.transfer$(a,b,$ amt$)$

위와 같은 시스템에서는 아래와 같은 결과가 도출된다.

{ c:($R'=R+1$),
c:($R'=R+1$);c:($a'=a$-amt;$b'=b$+amt) }

위에서처럼 논문에서 제안한 프레임워크를 사용하여 각 컴포넌트를 설계하고, 알고리즘을 적용하면, 설계한 시스템이 명세를 만족하는지 검증해볼 수 있다.

4. 결론

본 논문에서는 MTS를 사용하는 안전한 컴포넌트 기반 트랜잭션 시스템을 명세하고, 설계할 수 있는 정형화된 방법과, 설계한 시스템의 동작을 명세와 비교하여 검증할 수 있는 프레임워크를 제안하였다.

MTS를 이용하여 트랜잭션 시스템을 설계하는 경우에는 각 컴포넌트의 트랜잭션 속성 설정에 의해 생성된 트랜잭션 경계가 시스템의 확장성과 안전성에 큰 영향을 미칠 수 있다. 그러므로, 본 논문에서 제안된 프레임워크가 시스템의 확장성과 안전성을 고려한 트랜잭션 시스템을 설계하는데 유용할 것으로 기대한다.

참고 문헌

[1] Jim. Gray, Andreas Reuter. *"Transaction Processing : Concepts and techniques"*. Morgan Kaufmann Publishers, 1993.

[2] Rodney Limprecht. *"Microsoft Transaction Server"*. Proceeding of COMPCON. (San Jose, CA, February 1997)

[3] Leslie Lamport. *"The Temporal Logic of Actions"*. ACM Transaction on Programming Languages and Systems. VOL 16. 1994.

[4] Xiaolei Qian, Richard Waldinger. *"Transaction Logic for Database Specification"*. Proceedings of the conference on Management of data. Chicago. 1988.

[5] T. Sheard, D. Stemple. *"Automatic verification of database transaction safety"*. ACM Transaction on Database Systems. Vol.14, No.3, 1989.

[6] Vlada Matena, Mark Hapner. *"Enterprise JavaBeans ^{TM} Specification. v1.1"*. Sun Microsystems, Inc. 1999

[7] *"MSDN Online Library"*. http://msnd.microsoft.com/isapi/msdnlib.idc?theURL=library/psdk/mts20sp1/mtxpg10vb_57dx.htm. 2000