プログラミング 2−18
（1995. 8. 25）

メモリマップに基づく永続オブジェクト管理のための トランザクション機構

原 政博　山本 耕平　上原 敬太郎
宮澤 元　猪原 茂和　益田 隆司
東京大学 大学院 理学系研究科 情報科学専攻

近年はグループウェアなどの分散協調作業環境が重要になってきている。分散CADシステムや電子会議システムでは、永続オブジェクトの一貫性を保つためや永続オブジェクトをネットワークを介して効率よく操作するためのトランザクション機構が重要な役割を果たしている。そこで、本研究では永続オブジェクトを仮想記憶にマップする手法を用いて、分散環境で効率のよいトランザクション機構を提案する。優先順位を2段階でつけることによって飢餓状態をなくし、アクセス予測情報を用いることで不要なアボートをなくすリアルタイム性を持つトランザクション機構を構築する。

Memory-map Based Transaction System for Persistent Objects

Masahiro Hara, Kouhei Yamamoto, Keitaro Uehara,
Hajime Miyazawa, Shigekazu Inohara and Takashi Masuda
The Department of Information Science,
Graduate School of Science, the University of Tokyo

Recently distributed cooperative working environments like group software become important. The transaction systems play important roles in distributed CAD system and Electric conference system, because of preserving consistency of persistent objects and accessing persistent objects on network efficiently. Then we propose the effective transaction system on distributed computing environments by mapping persistent objects to virtual memory. Two stage priorities remove starvation. Read-write estimates remove unnecessary aborts. We construct the new transaction system by adding the essence of real-time.

1 はじめに

近年のネットワークの整備に伴って、コンピュータをネットワークで結ぶ分散環境が整ってきた。その分散環境を用いて、遠隔地にいても同時に作業のできる協調作業が発達してきている。例えば、分散CADシステムや電子会議システムなどがある。

そのような協調作業環境では、共有する永続オブジェクト(二次記憶上にあるデータ)の効率的な管理と一貫性を保つためのトランザクション機構が重要な役割を果たすことになる。

そこで、本研究では、分散協調作業環境で永続オブジェクトを効率よく管理するためのトランザクション機構および並行性制御機構を考える。

分散CADなどのアプリケーションでは、共有データが複雑なデータ構造を取ることが多い。この複雑なデータ構造を持つデータを分散環境で効率よく管理する方法が必要であると考えられる。

分散環境でのトランザクションでは、永続オブジェクトの要求や書き戻しの度に、ネットワークをデータが行き来するため、なるべく通信の回数を少なくする必要がある。そのために、トランザクションを管理する単位を大きくして、一度に必要なデータをある程度まとめて送ることができると、ネットワークをデータが行き来する回数が減ることになる。

そこで、ページサイズ(数KB)を一つのブロックとして管理するページサーバアーキテクチャ[6]を使用する。これらの例として、O_2, EXODUS, ObjectStore, Eosなどが既に研究されている。またそれと同時に、永続オブジェクトを仮想記憶にマップするメモリマップに基づく永続オブジェクト管理システムを用いる。

永続オブジェクトを仮想記憶に直接マップするとき、二次記憶上のイメージと仮想記憶上のイメージとを全く同じにすることができる。そのため、メモリで用いるような任意のデータ構造を用いることが可能となる。また、永続オブジェクトは常に同一アドレスにマップされるため、treeやlistといったポインタ操作を伴うデータ構造を用いることもできる。

分散CADや電子会議システムでは、ユーザの書き込みが、即座に永続オブジェクトに反映して、他のユーザに見えるようになることが望ましい。つまり、あるデッドラインまでにトランザクションが終るようにしたい。そこで、トランザクション機構にリアルタイム性のエッセンスを導入する。そこでは、デッドラインの情報と、どの永続オブジェクトをアクセスするのかという予測情報を用いて、なるべく短い一定の時間でコミットできるようにする。

以下、2節で本研究で用いる並行性制御機構の概要とその分散化について述べる。また、3節でリアルタイム性のエッセンスの導入について述べる。そして、4節で実験結果を述べる。

2 分散環境でのOptimistic Method

並行性制御機構には、Two Phase Lock (2PL)やTime-stamp Ordering (TO)といったものが知られている。しかし、これらの並行性制御機構はページ単位アクセスの分散環境では十分な高速性が得られない可能性がある。

例えば、2PLは分散環境では、デッドロックが起きた時に分散デッドロック検出をしなければならず、これに対するオーバーヘッドは極めて大きい。

また、TOでは、タイムスタンプを分散環境で統一的に扱うためのコストが大きくなる。さらに、トランザクションの管理の単位がページサイズと比較的大きいため、衝突するようなアクセスの可能性が高くなる。そうすると、あるトランザクションがアボートしたとき、他のトランザクションも道連れにしてしまうCascading Abortの危険性が大きい。

そこで、本研究では分散環境でオーバーヘッドの小さいOptimistic Method (OPT)に注目してトランザクション機構を構築していく。

2.1 バージョン化とタイムスタンプ管理

本提案のポイントは2つある。

1. バージョン化することで、読み出し専用のトランザクションをアボートさせないようにして、並列性を大きくする。
2. タイムスタンプを分散管理することで、タイムスタンプアクセスによるボトルネックをなくす。

従来のOPT[3]では、読み出し専用トランザクションまでもアボートしてしまうため、並列性が

小さい。また、永続オブジェクトの一貫性が壊れたときにアボートする必要があるが、アボートしてしまうまでにプログラムが異常終了する可能性がある。というのは、永続オブジェクトを仮想記憶に直接マップするため、永続オブジェクトに対してポインタ操作が可能となっている。そうすると、このポインタアクセスによる不当なメモリアクセスを起こす可能性があるからである。例えば、並行に実行されている2つのトランザクション T_1 と T_2 があるとする。まず、T_1 が書き込み $W_1(x)$ を行い、同時にポインタの書換え $W_1(y)$ を行ったとする。そのとき、T_2 は、T_1 によって書き換えられたページを読み出す操作 $R_2(y)$ を行うこともある (図1)。この場合、T_2 は、今まで見ていたバージョンと異なるバージョンを見ることになるので、当然アボートするはずである。しかし、従来のOPTでは validation phase までこのことはわからない。そのため、T_2 は、この異なるバージョンのページを見たままトランザクションが進行していく。そのとき、T_2 がこのポインタをたどってしまうと、このポインタは本来ならば T_1 が書き換えを行った新しいバージョンを指し示すはずなのに、T_2 が見ている古いバージョンを指し示してしまうことになる。そうすると、誤ったポインタアクセスによって不当なメモリアクセスを起こしてしまうことになる。

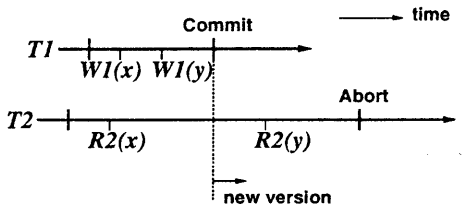

図 1: 衝突するトランザクション

したがって、まずは永続オブジェクトをバージョン化することによって、上記のような不当なメモリアクセスをすることをなくす。また、serializability を保つようにして、読み出し専用トランザクションはアボートしないようにする。

バージョン化した時、例えば、2つのトランザクション T_1, T_2 が (R: 読み出し, W: 書き込み, C: コミット要求)、

$$T_1 : W_1(x), W_1(y), C_1$$
$$T_2 : R_2(x), R_2(y), C_2$$

というアクセスしたとき、たとえ

$$R_2(x), W_1(x), W_1(y), C_1, R_2(y). C_2$$

という順番になっても、serializability を

$$\underbrace{R_2(x), R_2(y), C_2}_{\text{old version}}, \underbrace{W_1(x), W_1(y), C_1}_{\text{new version}}$$

つまり、$T_2 \rightarrow T_1$ と保つことができるので、読み出し専用トランザクションは必ずコミット可能になる。

また、分散システムで問題になるのが、バージョンを管理するためのタイムスタンプを取り扱うことである。

一般には、タイムスタンプを管理するために、専用のタイムスタンプサーバを用いることが多い。しかし、トランザクション処理においては、短い時間で処理できるようなトランザクションを何回も行う。その場合、トランザクションを行う度にタイムスタンプを獲得するので、何度もタイムスタンプサーバーにアクセスしなければならない。すると、そのタイムスタンプサーバーへのアクセスがボトルネックになり、トランザクションの効率が低下してしまう。

そこで、本研究では、各ホストのリアルタイムクロックを用いて、分散的にタイムスタンプを管理する手法をとる。ネットワーク上をメッセージが送られてくるときに、メッセージに含まれるタイムスタンプとリアルタイムクロックから得られたタイムスタンプを比較して、タイムスタンプが過去に戻るという矛盾した状態になる場合に、リアルタイムクロックを補正するという手法を用いる。

2.2 システムの概要

このトランザクションシステムでは、図2のように、Transaction Manager (TM) と Storage Server (SS) の二つで、トランザクション機構を提供する。TMはトランザクションの管理とクライアントとの通信を行う。SSは二次記憶上の永続オブジェクトを管理する。

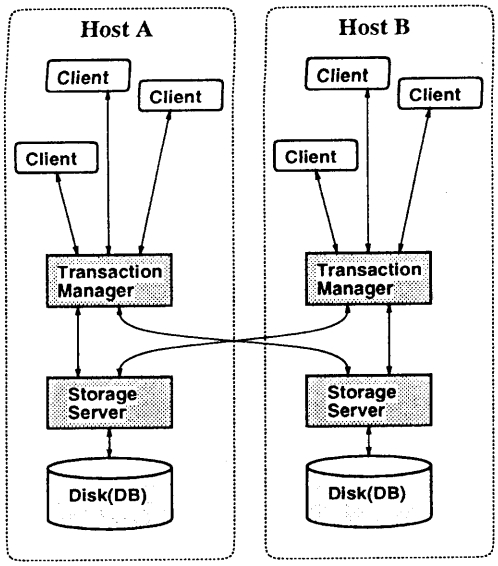

図 2: システムの概要

トランザクションに与えられるタイムスタンプには2種類ある。トランザクションを開始したときに与えられるタイムスタンプ (view-TS) と validation phase で与えられるタイムスタンプ (commit-TS) である。永続オブジェクトの各ページに対して、view-TS はどのバージョンを読むのかに使われ、commit-TS は書き込んだバージョンを表すのに使われる。また、各ページの古いバージョンをそのタイムスタンプと共に格納している。そして、各ページにはそれを参照した最も大きい view-TS の値 (max-view-TS) が保持されている。

2.3 アルゴリズム

OPT では、トランザクションは read phase, validation phase, write phase の3つの phase に分けられる。read phase では、2次記憶上の永続オブジェクトをメモリ上に読み込み、それに対して read や write などのアクセスを行う。validation phase では、このトランザクションのアクセスが、他のトランザクションの書き込みと衝突が起こっていないかを検出する。write phase では、コミットかアボートかを決定し、コミットならメモリ上の永続オブジェクトを2次記憶上に書き戻す。

以下、本研究で用いられる並行性制御機構の具体的なアルゴリズムを記述する。

- Read phase
  1. クライアントがトランザクションを開始して、はじめて永続オブジェクトにアクセスすると、現在のタイムスタンプ (view-TS) がトランザクションに与えられる。
  2. 永続オブジェクトのまだマップされていないページにアクセスすると、TM は SS に view-TS と共にページ要求を送る。
  3. SS は view-TS より古いバージョンの中で最も大きなタイムスタンプを持つページを返す。それを、TM はクライアントの仮想記憶に読み込む。このとき、そのページの max-view-TS が view-TS より小さかったら、max-view-TS を view-TS の値と同じにする。
  4. 永続オブジェクトのマップされたページに書き込みが起こると、ページはコピーされ、すべての書き込みはそのコピー上で行われる。
- Validation phase
  1. トランザクションに現在のタイムスタンプ (commit-TS) が与えられる。
  2. 永続オブジェクトに書き込みがなければ、バリデーション成功とみなす。
  3. 永続オブジェクトに書き込みがあれば、TM は view-TS と commit-TS とページのアクセス情報と共に "バリデーション開始" 要求を各 SS に送る。このとき、書き込みが起こったページの内容も伴う。
  4. SS は view-TS が各ページの最新バージョンのタイムスタンプよりも大きいかどうか調べる。もし、そのタイムスタンプが view-TS よりも大きければ、別のトランザクションが既に書き込みを行っていたことを示すので、TM に "バリデーション失敗" メッセージを返す。そうでなければ、 "バリデーション成功"

メッセージを返す。このとき、タイムスタンプを矛盾のないようにするために、recommended-commit-TS をトランザクションのアクセスした全てのページの max-view-TS と現在の SS のリアルタイムクロックの中で最も大きな値にする。

5. TM がトランザクションに関係ある全ての SS から“バリデーション成功”メッセージを得るとバリデーションは成功し、そうでなければ、バリデーションは失敗に終わる。そして、commit-TS と SS から返ってきた recommended-commit-TS の中で、最も大きなタイムスタンプの値を新しい commit-TS とする。

- Write phase

1. バリデーションが成功したのなら、TM は新しい commit-TS とともに“コミット”メッセージを各 SS に送り、失敗なら“アボート”メッセージを送る。
2. SS が“コミット”メッセージを受け取ると、書き込みが起こったページは commit-TS をタイムスタンプとする新しいバージョンとなり、全てのトランザクションに見えるようになる。このとき、SS のあるホストのリアルタイムクロックが commit-TS よりも小さければ、リアルタイムクロックを commit-TS の値に補正する。
3. SS が“アボート”メッセージを受け取ると、書き込みが起こったページは捨てられる。

2.4 本トランザクション機構の性質

このアルゴリズムでは、実際にどのページをアクセスしたのかという情報は、validation phase になって、はじめて使われる。また、コミットかアボートかということも、validation phase に入った順 (commit-TS の順) に決められる。つまり、validation phase に先に入ったトランザクションの処理が優先されることになる。

例えば、書き込みを行う短いトランザクションが多くあり、その最中に書き込みを行う長いトランザクションが現れる場合を考える。短いトランザクションの方が、先に validation phase に入るために優先されてしまう。そこで、長いトランザクションが短いトランザクションと衝突を起こすようなアクセスをしてしまうと、この長いトランザクションはアボートしてしまうことになる。このとき、長いトランザクションは restart するが、書き込みを行う短いトランザクションが多いために、この長いトランザクションは何回もアボートすることになり、なかなかコミットできないという starvation 状態に陥ってしまう可能性がある。

このような状況では、分散 CAD や電子会議システムのように、なるべく短い一定の時間で、永続オブジェクトへの書き込みを反映して欲しいというシステムに対して、妥当な時間で反映してくれない可能性がある。そこで、ある程度のリアルタイム性を導入することによって、長いトランザクションでも、少ない回数のアボートでコミットできるようにする。

3 リアルタイム性の付加

分散協調作業環境では、ときとしてリアルタイム性が重要になってくる。

例えば、分散 CAD システムでは、一つのトランザクションがデッドラインまでに終わるということがある程度わかっていれば、それを見込んだ永続オブジェクトの取り扱いができる。また、電子会議システムでは、ユーザの書き込みがなるべく短い一定の時間で他のユーザの表示系に反映することが望ましい。

そこで、本研究では完全なリアルタイム性を要求するのではなく、リアルタイムのエッセンスを導入することで、starvation のないトランザクション処理を達成することにする。

3.1 リアルタイムトランザクション

リアルタイムシステムでは、デッドラインの扱い方に Hard Real-time と Soft Real-time の 2 種類がある。

- Hard Real-time: デッドラインまでに処理を終えていなければならない。
- Soft Real-time: デッドラインまでに終えておく必要はないが、最大限の努力をする必要がある。

一般に、トランザクション処理では、処理にかかる時間の予測が難しいため、Soft Real-time が用いられることが多い。

また、リアルタイムシステムでは、このデッドラインを基にして優先順位をつけ、優先順位の高いものから優先的に処理を行っていく。

このデッドラインを基にした優先順位の付け方には、主として次に挙げる二つがある。

- Earliest Deadline: デッドラインが近い方の優先順位を高くする。
- Least Slack-time: Slack-time (デッドラインまでの時間から、残りの処理に要する時間を引いたもの) が小さい方の優先順位を高くする。

トランザクション処理では、処理にかかる時間の予測が難しいために、Earliest Deadline 法が使われることが多い。

この場合、書き込みを行う短いトランザクションが多く、その最中に書き込みを行う時間のかかる長いトランザクションがあったとき、この長いトランザクションは、なかなかコミットできずに、デッドラインまで達する可能性がある。つまり、書き込みを行う長いトランザクションは starvation に陥る可能性が高くなる。

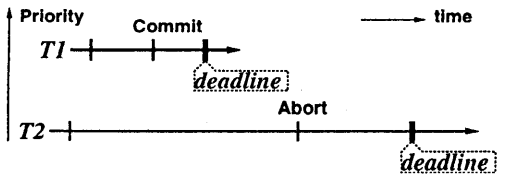

図 3: リアルタイムトランザクション

例えば、図3のような場合を考える。つまり、短いトランザクションと長いトランザクションが並列に実行されている場合、一般に、短いトランザクション T_1 はデッドラインを早めに設定してしまう。一方、長いトランザクション T_2 はデッドラインまでの時間を十分にとる必要がある。すると、Earliest Deadline 法では、この短いトランザクションの方が優先順位が高くなることになる。したがって、短いトランザクション T_1 の方が優先的に実行される。しかし、長いトランザクション T_2 がこの T_1 に衝突するような書き込みをした場合、T_2 はアボートすることになる。そこで、この長いトランザクション T_2 は restart する。書き込みを行う短いトランザクションが多い環境では、その短いトランザクションの優先順位が高くなりやすい。そうすると、長いトランザクションが何度もアボートすることになり、デッドラインに間に合わない可能性が高くなる。つまり、starvation が生じてしまう。

3.2 優先順位二段階割り当て法

そこで、長いトランザクションの starvation を解消するために、優先順位を二段階で割り当てる。

- Primary Priority: デッドラインに関係なく割り当てられる優先順位である。他のトランザクションによってアボートさせられて restart したときに、この優先順位が高くなる。
- Secondary Priority: Primary Priority が等しいときに用いられる優先順位である。これは、デッドラインが近いときにこの優先順位が高くなる。

こうすることで、図4のように、書き込みを行う短いトランザクションが多数あるときに書き込みを行う長いトランザクションが来ても、starvation を起こすことなくコミット可能になる。

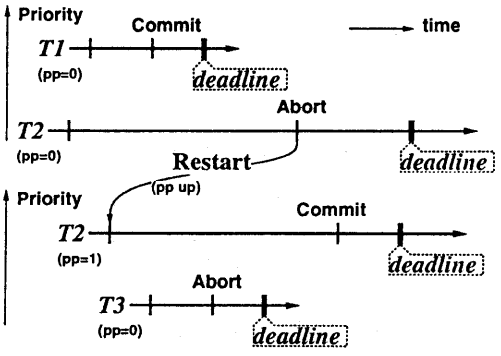

図 4: 優先順位 2 段階法

しかし、このままでは、Primary Priority の高い長いトランザクションを待つ間に、Primary

Priority の低い短いトランザクションはデッドラインに達してしまう。Soft Real-time であっても、デッドラインを大幅に遅れてまでも待つという訳にはいかない。そのため、デッドラインをある程度過ぎてしまったものは、アボートすることになる。その結果、短いトランザクションのアボート率が増加してしまい、効率がよくない。

3.3 予測情報の付加

本トランザクション機構は、ページ単位でトランザクションを管理するので、どのページをアクセスするのかという「アクセス予測情報」を、トランザクション開始のときに付加をする。

こうすることにより、Primary Priority が低くてもデッドラインの早いトランザクションがコミットできるようになる。

予測情報を付加することによって、Primary Priority の低いトランザクションはそのアクセスが、Primary Priority の高いトランザクションの予測情報に衝突していなければ、十分コミット可能になる。

逆に、Primary Priority.の高いトランザクションにとっては、アクセスがその予測情報内であれば十分コミット可能である。しかし、予測情報以外にまでアクセスしてしまった場合は、Primary Priority が低くてもデッドラインが早かったために既にコミットしているというトランザクションが存在する可能性がある。したがって、この Primary Priority の高いトランザクションでもアボートする可能性がある。

そこで、必要十分な予測情報が得られれば、よりアボートが少なくなると考えられる。

4 実験

ここでは、

1. Earliest Deadline 法 (ED)
2. 優先順位 2 段階法 (2S)
3. 優先順位 2 段階法 + アクセス予測情報 (2S+E)

3 種類で実験および比較を行う。

今回は、プロトタイプ段階であるため、まだ、優先順位を用いたスケジューリングを行っていない。現段階では、優先順位は validation phase でのみ用いている。

ここでは、3 人のユーザが各 100 回のトランザクションを連続で行うという仮定で実験を行う。トランザクションはほとんどが書き込みを行うトランザクションであり、短いトランザクション (デッドラインまでの時間 1sec) が 90%、長いトランザクション (デッドラインまでの時間 12sec) が 10% という割合で計測した。

全体で 64 ページある永続オブジェクトのうち、短いトランザクションはランダムに 4 ページ以下のアクセスを行い、長いトランザクションはランダムに 20-30 ページ程度のアクセスを行う。

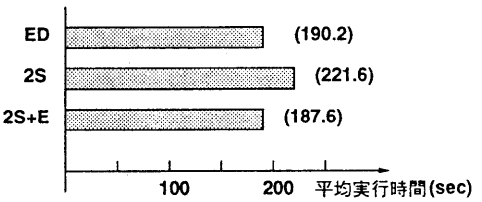

図 5: 100 トランザクションの実行時間

手法	starvation	deadline miss
ED	0.9	11.3
2S	0.0	18.8
2S+E	0.0	16.7

表 1: 100 トランザクションあたりの starvation と deadline miss の数

その結果、各ユーザのトランザクション実行時間は、図 5のようになった。表 1からわかるように 2S にすることで starvation をなくすことができた。しかし、逆に starvation をなくすことに重点を置き過ぎたために、デッドラインに間に合わないトランザクションが数多く出てしまうこととなった。このデッドラインに間に合わなかったトランザクションのほとんどが、Primary Priority の上がった長いトランザクションと衝突をするような書き込みを行った短いトランザクションであった。また、予測情報を導入した 2S+E では、優先順位だけによる不必要なアボートの数を減らすことができたので、並行性を向上させることができた。しかし、それでも、長いトランザクションの Primary Priority が上がったときに、短いト

ランザクションがデッドラインに間に合わないという問題も抱えている。

5 おわりに

分散環境で効率のよい並行性制御を行うために、Optimistic Method を用いた新しい並行性制御機構を考案した。

- バージョン化をすることで、不当なメモリアクセスによるプログラムの異常終了がなくなった。また、読み出し専用トランザクションのアボートがなくなり、並行性が向上した。
- タイムスタンプをリアルタイムクロックを用いて分散管理することにより、タイムスタンプ獲得によるボトルネックを排除した。
- リアルタイムのエッセンスを導入し優先順位を2段階でつけることによって、従来起こりがちだった長いトランザクションが starvation を起こすということはなくなり、実行効率も上がった。それだけでは、優先順位の低いトランザクションが不必要にアボートしてしまうので、それをなくすためにどのページをアクセスするかという予測情報を付加した。
- 逆に、長いトランザクションの優先順位が上がったときに、デッドラインに間に合わない短いトランザクションの数が増えてしまう結果となった。

そこで、将来はデッドラインに間に合わない短いトランザクションの数を減らすように、優先順位をより効果的に用いたスケジューリング方法を考えていきたい。

謝辞

本研究の一部は、情報処理振興事業協会「独創的情報技術育成事業」の一環として行われたものです。

参考文献

[1] 原 政博, 山本 耕平, 上原 敬太郎, 宮澤 元, 猪原 茂和, 益田 隆司: “メモリマップに基づく永続オブジェクト管理のためのトランザクション機構”, 情報処理学会第 50 回全国大会, pp. 4-161–162, 1995.

[2] Inohara, S., Y. Shigehata, K. Uehara. H. Miyazawa, K. Yamamoto, and T. Masuda. “Page-Based Optimistic Concurrency Control for Memory-Mapped Persistent Object Systems,” in *Proc. of the 28th Hawaii Int'l Conf. on System Sciences* vol. II, pp. 645–654, Jan. 1995.

[3] Kung, H. T., and J. T. Robinson, “On Optimistic Methods for Concurrency Control,” *ACM Trans. on Database Systems*, vol. 6, pp. 213–226, Jun. 1981.

[4] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb, “The ObjectStore Database System,” *Communications of the ACM*, vol. 6, pp. 34–49, Oct. 1991.

[5] Lausen, G., “Formal Aspects of Optimistic Concurrency Control in Multiple Version Database System,” *Information Systems*, vol. 8, pp. 291–301, Feb. 1983.

[6] Özsu, M. T., U. Dayel, and P. Valduriez, *Distributed Object Management*. San Metro: Morgan Kaufmann Publishers, 1994.

[7] Prädel, U., G. Shalageter, and R. Unland. “Redesign of Optimistic Methods: Improving Performance and applicability,” in *Proc. of the 2nd Int'l Conf. on Data Engineering*. pp. 466–473, IEEE Computer Society Press. Aug. 1986.