データベースシステム 122－64
(2000. 7. 28)

分散独立アクティブデータベース上での ワークフロー用トランザクションモデル

井川 智崇 tikawa@de.cs.titech.ac.jp
横田 治夫 yokota@de.cs.titech.ac.jp

東京工業大学
大学院 情報理工学研究科 計算工学専攻
〒152-8552 東京都目黒区大岡山 2-12-1

あらまし 組織の中で複数の部門が協調して作業を行う場合に、ネットワークで接続されたコンピュータによって部門間の処理の流れをサポートするものとしてワークフローがあげられる。我々はワークフローを分散した独立アクティブデータベースシステム間のメッセージ通信によって実現する方式を提案してきた。今回我々はこのシステムを用い、発火するルールがその時の緒条件によって変化ようなモデルを管理する方法について、保持し続ける必要のなくなった親トランザクションを取り除く枠組みも含めて説明する。さらにシステム障害が発生した場合のリカバリについて、トランザクションの親子関係をどの様にしてシステム内に再構築するのかについても説明する。

キーワード ワークフロー、アクティブデータベース、入れ子トランザクション、補償トランザクション

A Transaction Model for Workflows on Distributed Independent Active Databases

Tomotaka Ikawa tikawa@de.cs.titech.ac.jp
Haruo Yokota yokota@de.cs.titech.ac.jp

Graduate School of Information Science and Engineering,
Tokyo Institute of Technology
2-12-1 Oookayama, Meguro,
Tokyo 152-8552, Japan

Abstract

A workflow system can be seen a system supporting cooperative processes among network connected computers placed in separate sections of an organization. We have proposed a mechanism for the workflow system constructing a nest transaction spread over distributed independent active databases. In this paper, we report a model to manage distributed subtransactions fired by rules depend on dynamic conditions. We also propose a mechanism to disconnect committed parent transactions from the subtransactions, and method to reconstruct relationship between parent and child transactions dynamically during system recovery.

key words workflow, active database, nested transaction, compensating transaction

1 はじめに

組織内の複数の部門あるいは複数の組織が協調して業務を遂行する際、その業務の流れをサポートするものとしてワークフローがある。ワークフローを実現する手段としてはメールを用いたエージェントシステムや集中サーバーを用いたシステムなど多数の提案が成されている。その中で我々はワークフローを分散した独立アクティブデータベースシステム間のメッセージ通信によって実現する方式を提案しており、我々はこれを DIADEM (Distributed Independent Active Database with subtransaction Exporting Mechanism) と呼んでいる。

イベントによって発火するルールがその時の緒条件によって変化する場合、発火したルールの親子関係を何らかの形で保持しておかなければ補償を行う事が出来ない場合がある。この様な動的に変化するルールに対しては、木構造を用いてルールの親子関係を保持する事により対応することが可能であると考えられる。

今回我々は、我々の提案している DIADEM を用い、上述した様な動的に発火するルールが変化するモデルの管理方法について、保持し続ける必要のなくなったトランザクションの部分木を取り除く枠組みも含めて説明する。さらにシステム障害が発生した場合のリカバリにおいて、この木構造をどの様にしてシステム内に再構築するのかについても説明する。

2 ワークフローにおけるトランザクションモデル

ワークフローをトランザクションで管理する場合、そのトランザクションというのはしばしば長時間にわたって処理が継続されうる。これはワークフローにおいては多くのデータベースオブジェクトにアクセスしたり、膨大な計算を処理したり、そしてユーザーからの入力待ちを行ったりするためである。その結果として、ほとんどの場合においてこれは深刻な問題を示すことになる。

伝統的な ACID トランザクションモデルでは、トランザクションを atomic に実行するためにオブジェクトをロックする。そのため、他のトランザクションはそのオブジェクトのロックが開放されるまで待たされることがある。トランザクションが長大になればそれだけ他のトランザクションが待たされる確率や時間も増加する。そのためにシステムのスループットが低く押さえられると同時に、利用者の利便性も大幅に損なわれることになる。

しかしながらワークフローの場合には、伝統的な ACID トランザクションにおける制約を緩和させることによってこれらの問題に対処する事が可能である。多くのモデルでは以下の様な形態を用いてこれに対処している。

- トランザクションはサブトランザクションの集まりとして構成されており、それぞれのサブトランザクションは伝統的なトランザクションの ACID 性を満たしている。
- トランザクション全体では isolation は満たされない。これはワークフローにおいては isolation があまり意味のある制約ではないという認識によるものである。サブトランザクションの単位でコミットもしくはアボートが行われるので、トランザクションの途中結果 (すなわちサブトランザクションによってもたらされる結果) は外部のトランザクションから見えることとなる。
- トランザクション全体の atomicity を保持するために、個々のサブトランザクションに対して補償トランザクションが提供されている。これは意味的には undo と同等の操作を実現するものであり、サブトランザクションの処理結果を打ち消す処理を実行する事によってデータベースを元の状態に戻すものである。

ワークフローを管理するトランザクションモデルには様々な手法が提案されている。これらのモデルは上述の要求を満たしながらもそれぞれトランザクションの管理方法や実装方法は異なっている。以下に代表的なモデルの特徴を示す。

2.1 SAGAS

saga[2] においてはトランザクションモデルはフラットで、トップレベル saga と単純なトランザクションと言う 2 レベルのネスティングのみが許可されている。外部にトランザクションの途中結果を見せ、それを打ち消すために補償トランザクションを導入した最初のモデルが saga である。

障害によって saga が停止した時のリカバリの手段としては backward recovery と forward recovery (そして mixed backward/forward recovery) がある。backward recovery は補償トランザクションを用いて実行され、また forward recovery は savepoint を用いて実行されるものである。ただし、forward recovery はあらゆるシチュエーションで利用可能と言うわけではない。

尚、このモデルの拡張として、並列トランザクションのために拡張された parallel sagas や、入れ子トランザクションを用いた nested sagas 等がある。

2.2 ConTract

ConTract[3][4] では、内部的にはイベント指向のフローマネジメントを実装している。*step* の終了後 event の発火は condition によって制御される。condition の評価が真である場合それに引き続く *step* が発火する。しかしこれらは内部言語として隠蔽されており、ユーザーにはより高級なプログラミング言語が提供されている。サブトランザクションはスクリプトのレベルで取り扱われている。

補償はユーザーが明示的に要求する場合にのみ行われる。システムによるリカバリやコンフリクトの解消と言った暗黙的な目的のためには用いられない。

ConTract では、*Context* と呼ばれる、アプリケーションの状態を定義するプライベートなデータのセットを用いて forward recovery を行う。ConTract では、過去の値を上書きするのではなく、更新される context の要素それぞれに対して新しいバージョンを生成する。これにより context データベースは ConTract の完全な履歴を保持することとなる。

2.3 OPMS

OPMS[5] ではプロセスはその構造とコントロールの範囲を基にしたアクティビティの階層によって表されている。OPMS は ECA ルールによる動的なプロセスの修正をサポートしている。また、**two-phase remedy** と呼ばれる階層的な障害ハンドリングメカニズムを開発している。これは障害時にトランザクションの木構造の論理的な undo の root (LUR) を定めることにより部分的なロールバックを許可し、そしてリスタートとロールフォワードを許可するものである。さらに、OPMS はプロセスが自己修復を行う、すなわち障害が発生した祖先達を部分的にロールバックする、そしてその後に "rolling forward" を行う。OPMS のプロセスモデルは *multilevel activity nesting* と *activity/block coupling* に特徴づけられる。

さらに、hyper-activity はリモートサイトにて実行されるサブプロセスのプロキシとして取り扱われる事が可能である、これは分散ワークフロー環境において上記のアドバンテージを拡張するものである。

3 DIADEM のトランザクションモデル

我々の提案している DIADEM はアクティブデータベースをベースとし、相互にネットワーク通信を行うことにより、全体が協調してトランザクションの管理を行うシステムである。サブトランザクションは ECA ルールに基づいて処理され互いに入れ子状に木構造を形成し、全体として 1 つのトランザクションとなる。

ここで取り扱うワークフローモデルでは、ネットワークによって繋げられた複数の部門によって処理が進む事が前提となっている。このため、ネットワーク上の独立した複数のシステムにまたがって入れ子トランザクションが存在するような機構を用意する必要がある。この場合の入れ子トランザクションのサブトランザクションはネットワーク上の別のアクティブデータベースのトランザクションとして生成される事になる。なお、ここでは簡単化のため、サブトランザクション間の発火後のインターラクションは考えない。

3.1 アクティブデータベースと木構造

アクティブデータベースとは、旧来の受動的なデータベースとは対照的に、データベースが定められたルールを能動的に起動し実行するデータベースの事である [1]。アクティブデータベースには ECA (Event-Condition-Action) と呼ばれるモデルがあり、このモデルに従ってルールが起動/実行される。すなわちルールは、

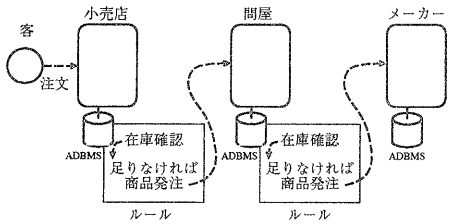

図 1: サプライチェーンにおけるルールとワークフローの関係

- **Event:** ルールが起動される原因となるイベント
- **Condition:** 起動したルールを実行するかどうかの評価
- **Action:** 処理の実行

というモデルに従い、起動され、評価されそして実行される。さらに実行された処理によって新たなイベントが発生し、別のルールが起動し実行される事もある。このような処理の流れにより、ルールの実行、すなわちサブトランザクション、が互いに入れ子状になり全体として 1 つのトランザクションを形成している。こうすることで、イベントの発生を補償する場合に正しくサブトランザクションを補償することが可能となる。

また、 DIADEM では相互にネットワーク通信を行うことで、複数のアクティブデータベースが協調してトランザクションを処理する。各サイトは自サイトにおいて処理が行われる、トランザクションの断片に対してのみ責任を持つ。このような枠組みは、前出の saga、 ConTract、 OPMS 等では扱うことができないものである。

3.2 サプライチェーンの例

例を用いて説明しよう。図 1 における、小売店、問屋、メーカーにおけるサプライチェーンについて見ていく事とする。それぞれの部署では DIADEM が稼働しているものとする。まずお客が小売店である商品を購入したとする。すると小売店では在庫の確認が行われ、在庫が底をつきそうであると判断したならば問屋に商品の発注が行われる。商品の注文を受けた問屋でも同様に在庫の確認が行われ、必要ならばメーカーに商品を発注するように要請する。この様に、それぞれの部署は独立して処理を行うが、システム全体として 1 つのトランザクションを管理すると言うのが DIADEM のトランザクションモデルの特徴である。

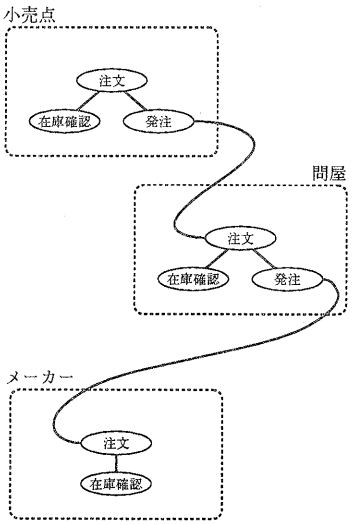

図 2: 図 1 のワークフローのトランザクションの木構造

DIADEM ではその時の様々な条件によって発火するルールや実行されるアクションが異なってくるのが特徴である。例えば先の図 1 においては、小売店に十分な数の在庫があるならば問屋に商品を発注する必要はないし、残り少なくなってきたならば問屋に発注を行うだろう。これはトランザクションの木構造が動的に変化することを意味している。

3.3 補償トランザクション

あるトランザクションを補償する際、そのトランザクションによって生じたイベントによって発火したトランザクションも全て補償する必要がある。しかしデータベースレベルではそれぞれのトランザクションは独立して処理されているので、どれが補償の対象となるのか知ることができない。このことから各トランザクションの対応関係を保存する必要があり、 DIADEM ではこれを木構造を用いて保存している。前出のサプライチェーンの例で言うと、客が注文をキャンセルした場合に、小売店から問屋への、さらには問屋からメーカーへの注文を正しくキャンセルするために木構造 (図 2)

を利用する。

逆に、我々はイベントによって発火したサブトランザクションが親トランザクションをロールバックしたり補償したりする事はないと考えている。サブトランザクションはそれぞれデータベース内では独立したトランザクションとして取り扱われる。さらに、それぞれのトランザクションは自分自身のロールバックや補償に関してのみ責任を持つのが普通である。例えば小売店が問屋に発注を行った後、問屋が小売店の発注を取り消す必要はない。

3.4 部分木の独立化

木構造による対応関係の保存に関してさらに言えば、ロールバックや補償はもはや行われないと分かっているトランザクションまで保持し続ける必要はない。

例えば図3において斜線部の部分木が既にコミット状態にあるとすると、前述した様に親トランザクションに対してロールバックや補償は行われない事から、その下の部分木によって斜線部が補償されることはないことが分かる。またシステム障害が発生したとしてもデータベースレベルでは斜線部は既にコミット状態にあり、リカバリはその下のサブツリーに関してのみ行われる。すなわち斜線部の木構造を保持し続ける必要性は全くないという事が分かる。そのため DIADEM では、この様な保存し続ける必要のないトランザクションは木構造から除去する事にしている。

前出のサプライチェーンの例で言うと、客が小売店のトランザクションをコミットすると、問屋の注文トランザクションは下にメーカーのサブトランザクションを持つ、独立したトランザクションとなる。

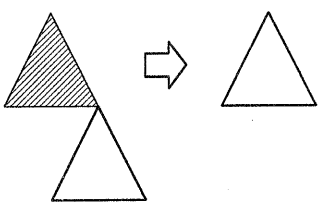

図 3: 不必要な木構造の除去

4 DIADEM における障害ハンドリング

DIADEM は前章にて示した様々なモデルとは異なり、他のサイトに独立して存在する DIADEM と通信を行う事によってシステム全体でトランザクションを処理する。そのためシステム障害が発生した場合にはこれを考慮したリカバリ手法を用いる必要がある。

例として 前述の例と同様にトランザクションが小売店、問屋、メーカーにまたがって処理されている場合について考えよう。図4は DIADEM における木構造がどの様に構築されているかを表している。図中の a_n は action を表しており、実際のデータベース処理を行うものである。c_n は補償トランザクションを表しており、r_n はルールを管理するためのオブジェクトで、イベントによって発火したトランザクションを持っている。また、t_n は a_n、c_n、そして r_n の3つ組を保持するものであり、e_n はネットワークで繋がれた別の DIADEM 上で管理されるルールへのリンクを表すものである。

今、メーカーにおいて DIADEM がシステム障害を起こしたとする、問屋におけるこのトランザクション (e_2) は、メーカーの DIADEM が復旧して処理を再開するまで待ち続ける事になる。メーカーの DIADEM はログを用いてトランザクションの木構造を再構成し、問屋との通信を回復する事によってシステム障害から回復する。回復後は引き続きトランザクション処理を続行する事が可能である。また、小売店や問屋ではシステム障害発生から回復するまでの時間待たされることになるが、これによって小売店や問屋における DIADEM の利便性が損なわれる事にはならない。と言うのもそれぞれの部署は互いに独立して稼働しているため、外部の DIADEM にて実行されるトランザクションに対してロックを行う必要は無いからである。

それぞれのサイトにてトランザクション全体のログを保持する必要はない。それぞれのサイトは自分自身が受け持っているトランザクションの断片に対してのみログを保持すれば良い。ログにはアクティブデータベースとして必要となる情報に加え、どのサイトの DIADEM とリンクしているのかという情報が必要となる。

ログには以下の情報が記述される。

- *begin-transaction*: トランザクションの生成時

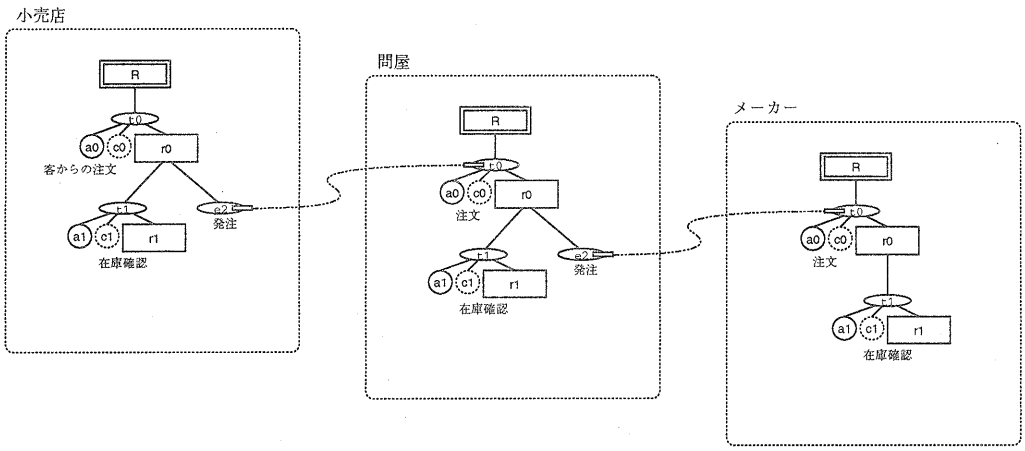

図 4: トランザクションの木構造

にログに記述される。この時、親トランザクションのID、自分自身のトランザクションID、そしてこのトランザクションを実行するのは自分自身なのかそれとも外部の DIADEM なのか、外部で実行されるならばどこで実行されるのか、さらに実行する action の SQL 文と補償トランザクションの SQL 文といった情報が同時にログに記述される。

- *end-transaction*: トランザクションの終了時に記述される。この時トランザクション ID が同時にログに記述される。

- *begin-event, end-event*: トランザクションによって生じるイベント検出の開始と終了を表わす。この時トランザクション ID が同時にログに記述される。

- *begin-action, end-action*: トランザクションの action の実行の開始と終了を表わす。この時トランザクション ID が同時にログに記述される。

- *begin-compensation, end-compensation*: トランザクションの action に対する補償トランザクションの開始と終了を表わす。この時トランザクション ID が同時にログに記述される。

トランザクションの生成は、実際の処理内容を表す action と補償トランザクションそしてイベント検出部を生成し、トランザクションの対応関係を木構造に保持し、ログに *begin-transaction* を記述することによって行われる。実際にデータベースに対して処理を行うのは action 部であり、この処理の開始時には *begin-action* がログに記述される。

action の実行が完了してコミットする時、ログには *end-action* が記述される。action がコミットした後も補償される可能性が残っている間はトランザクションは終了しない。トランザクションは、もはや補償やロールバックが起こらないとみなされた時点で終了し、*end-transaction* がログに記述される。システム障害からのリカバリの際に、ログ内に *end-transaction* がある場合、このトランザクションに関しての操作は全て完了しており、システムは何もする必要はない。

システム障害からのリカバリの際に *end-event* がログに記述されていない場合、イベントの検出が完了する前にシステム障害が発生したことが分かる。その時はイベントの検出をもう一度やり直さなければならない

同様にシステム障害からのリカバリの際に *end-compensation* がログに記述されていない場合、補償トランザクションの実行は完了していないので、まず補償トランザクションをロールバックし、再度補償トランザクションを実行する。

5 まとめ

本稿では、分散した独立アクティブデータベースシステム間のメッセージ通信によるワークフローの管理方法を示し、イベント発生時の条件により発火するルールが変化するワークフローモデルを、親子関係を木構造として表現することで管理する方法を示した。この様に表現される木構造においてはロールバックや補償がもはや行われないと分かっているトランザクションまで保持し続ける必要はないので、この様な不必要な部分木を削除するための方法を示した。さらにシステム障害発生時のリカバリに関し、この木構造がどの様にして再構築されるのかについて、ロギングを用いた木構造の修復方法を示し、システムのリカバリ手法についての説明を行った。

ConTract はサブトランザクションの親子関係をシステム内に保持している点で我々のモデルと共通しているが、 ConTract の場合ワークフローのルールをスクリプトとして記述しているため、親子関係はあらかじめ決められた、静的なものであると考えることができる。対して DIADEM の場合これは動的に変化する。

OPMS においては、アクティビティをリモートサイトで実行されるサブプロセスのプロキシとして取り扱うことで分散ワークフロー環境に拡張することが可能であるとされている。この場合 OPMS はトランザクションは 1 つのサイトにて集中管理するものである。 DIADEM ではトランザクションは各々のサイトにて独立して管理される。

DIADEM におけるトランザクションの管理モデルは、「自分が実行しているトランザクションの断片のみに対して責任を持つ」というものである。つまりトランザクションは集中管理されておらず、それぞれの DIADEM によって独立して管理され、システム全体として論理的なトランザクションを構成するというものである。このため、どこかでシステム障害が発生したとしてもシステム全体が停止することはないので、システムのスループットの低下が押さえられると同時に、利用者の利便性あまり低下しないと考えられる。

現在、 java を用いて実験システム [8][9] を構築しているが、障害回復の機能等はこれから実現する予定である。さらに、分散したサブトランザクション間のインタラクションはないものと仮定しているが、そのようなインタラクションについても今後考察する必要がある。今後このモデルを詳細化すると共に、 Java による実験システムに不足している機能を追加しモデルの確認を行う予定である。

参考文献

[1] J. Widom and S. Ceri. "Active Database Systems: Triggers and Rules For Advanced Database Processing", Morgan Kaufmann Publishers,Inc., San Francisco, CA, 1996.

[2] Hector Garcia-Molina, Kenneth Salem. "SAGAS", ACM SIGMOD, 1987.

[3] Helmut Wächter, Andreas Reuter. "The ConTract Model", Database Transaction Models for Advanced Applications, chap. 7, 1992.

[4] Andreas Reuter, Friedemann Schwenkreis. "ConTracts – A Low-Level Mechanism for Building General-Purpose Workflow Management-Systems", Data Engineering Bulletin 18(1), 1995.

[5] Qiming Chen, Umesh Dayal. "A Transactional Nested Process Management System", International Conference on Data Engeneering, 1996

[6] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, C. Mohan. "Advanced Transaction Models in Workflow Contexts", 12th ICDE, 1996.

[7] 速水 治夫 , 坂口 俊昭 , 渋谷 亮一. "ここまで来たワークフロー管理システム: (3) ワークフロー製品の実際", 情報処理 39 巻 12 号, pages 1258-1263, Dec 1998.

[8] 井川 智崇 , 宮崎 純 , 横田 治夫. "独立アクティブデータベース間の入れ子トランザクションの実現", 第 10 回 データ工学ワークショップ, May 1999.

[9] 井川 智崇 , 宮崎 純 , 横田 治夫. "Java のスレッドを用いた分散入れ子トランザクションの実現", 情報処理学会研究会報告, データベースシステム DBS-119-60, pp.357-362, 1999.