マルチメディア通信と分散処理 66−18
（１９９４．７．８）

RSC: グループウェアのためのオブジェクト共有機構

岡野 裕之
日本アイ・ビー・エム (株) 東京基礎研究所

あらまし　WYSIWIS 型グループウェアの構築を容易とする、オブジェクト共有機構 RSC を設計し、そこで必要となるソースコード・トランスレータを実現した。RSC は C++で記述されたイベント駆動型のプログラムを対象とし、インスタンスオブジェクトの共有機構を提供する。また、オブジェクトに対する書込みロック、読出しロックの機能を提供し、オブジェクトの一貫性を保証する。このとき、トランザクションをイベント処理に対応づけること、ソースコード・トランスレータでロックを自動的に行うことで、ユーザの負担を少なくした。さらに、インタラクディブなユーザインタフェースに対応するため、楽観的なロック機構を用意した。

RSC : An Object-sharing Mechanism for Groupware

Hiroyuki Okano
okanoh@trl.ibm.co.jp
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242 JAPAN

Abstract. This paper describes the design of Remote Sharing C++ (RSC), a new object sharing mechanism for constructing WYSIWIS groupware and the implemention of its source code translator. Unlike most groupware construction toolkits, RSC provides a sharing mechanism for C++ instance objects that can be used to connect general event-driven style programs and provides locking mechanisms that do not block interactive userinterface.

1 はじめに

多人数で文書を作成してそれを同時に検閲・校正するなどの、同期協調活動を支援するグループウェアがある。このようなグループウェアの構築方法について、記述の容易性、既存のアプリケーションプログラムの再利用性など、いろいろな観点から研究がなされている [1][2][3][4]。

グループウェアの実現において、一般性を損ない易い部分が次の 2 点である。これらのために一般的な解決は困難で、個々に独自の方法で実現されているのが現状である。

(1) ポインタを含む複雑なオブジェクトの共有機構
(2) 共有するオブジェクトに対するロック機構

これに対し筆者は、プロセス間でオブジェクトを自動的に共有するための機構を提供し、さらに共有オブジェクトに対するロックを自動的に行う機構を提供することで、グループウェアの記述を容易にするための研究を行っている。このオブジェクト共有機構を RSC (Remote Sharing C[1]) と呼ぶ。

RSC は、オブジェクトサーバとソースコード・トランスレータ、およびランタイムルーチン・ジェネレータから構成される。オブジェクトサーバは、永続的なオブジェクトを蓄積するものではなく、グループウェアのためのオブジェクトキャッシュ、およびロックマネージャとして機能する。ソースコード・トランスレータは、C++のプログラムを変換し、共有オブジェクトへの参照・変更を検出するための機能を組み込む。また、ランタイムルーチン・ジェネレータは、変換後のプログラムが呼び出すランタイムルーチンと、そこで参照される共有オブジェクトのスキーマ (構造の記述) を生成する。

RSC を用いてグループウェアを記述することで、ユーザがプロセス間通信のプロトコルを記述する必要がなくなり、一般性の高いプログラムとすることができる。また、共有するオブジェクトが同種のものである限り、異種のプログラムが 1 つの協調作業に参加することが可能となり、それらのプログラムを独立に開発することが可能となる。本論文は、RSC の設計およびソースコード・トランスレータの実現について述べる。

2 RSC の目的と設計方針

2.1 RSC の目的

RSC の目的は、ヘテロな環境で動作するヘテロなアプリケーションプログラムを容易に結合することである。対象とするアプリケーションはおもにグループウェアで、共有オブジェクトを加工するサーバのようなものも考える。

結合を容易にする方針は、手続きの呼び合いをなくし、オブジェクト共有だけでプロセス間通信を抽象化する、ということである。

リモートプロシジャーコールなどの明示的なプロセス間通信を不要とするために、ロックおよび同期処理をオブジェクトへの操作として抽象化する。これによってプロセス間通信のプロトコルをユーザが記述する必要がなくなる。したがって、オブジェクトのインタフェースをあらかじめ定義することにより、通信に参加するプログラムを独立に設計・開発することができるようになる。

2.2 グループウェアの性質

共有オブジェクトを操作するグループウェアの性質を述べる。共有オブジェクトとは、例えばテキストや図形などのデータである。また、同じ共有オブジェクトを操作するグループウェアにも、テキストエディタ、校正支援エディタ、作図エディタなど、さまざまなものがあると仮定する。

- What You See Is What I See (WYSIWIS)

  操作対象となるオブジェクトを共有するだけでなく、他のユーザ[2]の視点、注目している位置の情報も共有する必要がある。

- ロックによってブロックされない画面表示

  オブジェクトを共有してその一貫性を保つには、オブジェクトに対するロックが必要となる。一方、グループウェアはインタラクティブなユーザインタフェースを持つため、オブジェクトの内容を画面に表示しなければならない。このとき、ロックによって表示がブロックされてはいけない。

- ユーザインタフェースに依存した情報の管理

  画面上の座標などのユーザインタフェースに依存した情報を、各オブジェクトに対応させて管理できると効率がよい。これを実現するには、共有オブジェクトの中に、各参加者でローカルに有効なメンバを記述できる必要がある。

- ユーザの操作に依存したインタラクション時間

  オブジェクトの変更を開始してから終了するまでをインタラクションと呼ぶとすると、このインタラクションの時間はユーザの操作に依存する。例えば作図エディタでは、図形プリミティブをマウスで選択し、ボタンを押したままドラッグして、別の場所でボタンを放すといった操作がある。この場合ボタンを押している期間がインタラクションになるが、その時間はユーザの操作に依存する。

[1] RSC の C は、C++, Collaboration, Cooperative work, Communication ... を指す。

[2] 「ユーザ」はグループウェアの利用者である。これに対し「参加者」はグループウェアのプログラムを指す。

2.3 設計の前提

2.3.1 グループウェアの構造

RSC が対象とする参加者プログラムは、イベント駆動型のプログラムであると仮定する。例えば、X ツールキットを使ったプログラムなら、図1のようなループ構造がある。

```
while (TRUE) {
  XEvent event;
  XtNextEvent(&event);          callback1
  XtDispatchEvent(&event); ↔    callback2
}                                ...
```

図 1: イベント駆動型のプログラム

1 つのイベントを処理する単位、この例で言うと callback 関数の実行を、RSC ではトランザクションとみなす。また、複数のトランザクションにまたがってオブジェクトを変更する操作を、RSC ではインタラクションと呼ぶ。インタラクションの実行時間は非常に長く、あらかじめ予測できないとする。

2.3.2 システム環境

対象とする OS やネットワーク環境について、次のような前提をおく。

- ヘテロな実行環境を対象とする。
  グループウェアを実行するマシンは LAN で結ばれていて、プロセッサのアーキテクチャや OS は異種のものとする。
- メッセージによるプロセス間通信を行う。
  分散共有メモリは使用しない。なぜなら、ヘテロな環境に対応できない、オブジェクトのすべてを共有してしまう[3]という問題があるからである。

2.3.3 ロックの必要性

複数のプログラムでオブジェクトを共有し、その consistency を管理することは、データベースにおいてデータの integrity を保証することと等価な問題を含んでいる。つまり、正しいプログラムが書けることを保証するためには、オブジェクトに対する書込みロック、読出しロックを用意する必要がある。

例えば、整数のオブジェクト A, B, C, D がある時に、B と C を加えて A と D に代入したいとする。

(1) A = B + C
(2) D = A

このとき、B を参照した後で C が書き換えられてしまうと、A には不完全な値を代入することになる。したがって、誰からも書き込まれないように B と C を読出しロックしなければならない。

また、(1) の実行と (2) の実行の間で、誰かが A と D を読み出してしまうと、その A と D の関係は不完全である。したがって、誰からも読み出されないように A と D を書込みロックしなければならない。

取得したロックは、処理が終了するまで持ち続け、処理が終了した時点で解放する。ロックの取得と解放が処理の前後に分れることからこれを 2 相ロックと言い、この処理単位をトランザクションと言う。

[3]つまり、各参加者でローカルに有効な、共有されないオブジェクトメンバ (ローカルメンバ) を実現できない。

2.4 設計方針

上記のようなグループウェアの性質、設計の前提をもとにして、次のような方針で設計を行った。

- C++オブジェクトのグラフ構造を対象とする。
  マークアップ付きテキストと図形が混在する文書を、1 つの典型的な対象として仮定する。
- ロック機構を用意する。
  ロックに関するプログラマの負担はなるべく少なくする。また、他の参加者のインタラクションによって、画面表示がブロックされないようにする。
- 視点情報を共有する機能を用意する。
  共有オブジェクトに対するユーザの視点や、画面の位置などの視点情報を共有する。WYSIWIS の実現に必要な機能である。

3 設計

3.1 共有オブジェクト

参加者間で共有するオブジェクトは、C++のインスタンスオブジェクトである。特定の名前のスーパクラスを継承するクラスを共有クラスと呼び、共有クラスのインスタンスを共有オブジェクトとする。この共有オブジェクトの中で、`public` で宣言されたメンバが共有対象である。`private` や `protected` で宣言されたメンバは、そのオブジェクトを生成したプロセスだけで使用可能なローカルメンバとなる (図2)。

```
class  class-name:
       public  rscshare(session-name)  {
       public:      共有メンバ
       private:     ローカルメンバ
       protected:   ローカルメンバ
       };
```

図 2: 共有クラスの定義

3.2 ロックとトランザクション

共有オブジェクトの一貫性を保つために、書込みロック、読出しロックを用意する。このとき、RSC におけるプログラミングスタイルが従来のものと大きく異なることは避けたい。そうなっては、既存のプログラムを RSC を使ってグループウェア化することが困難になる。これに対し、イベント処理型のプログラムに対象を絞り、1 回のイベント処理をトランザクションと考えることにより、ユーザがトランザクションの区間を明示的に指定することを不要とする。

また、ソースコード・トランスレータによってプログラムを変換することにより、ユーザがオブジェクトへの読出しロック、書込みロックを指定することを不要とする。変換されたプログラムには、共有オブジェクトへの参照・変更を検出する機能を持たせ、ロック操作とオブジェクトの一貫性管理を自動的に行う。

ロックの単位をオブジェクト単位あるいはクラス単位のように小さくすると、ロックを取得するためのオーバヘッドが無視できなくなるため、いくつかの共有クラスをまとめてセッションと呼び、セッションのインスタンスをロックの単位とした(図3)。

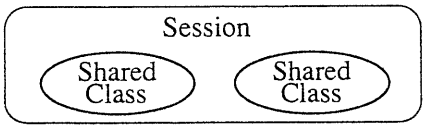

図3: セッションと共有クラス

C++のプログラム中から共有オブジェクトを参照・変更すると、自動的にそのオブジェクトに対応するセッションインスタンスがロックされる。さらに、各共有オブジェクトはvalidやinvalidなどの状態を持っており、オブジェクトへの操作が行なわれるたびにこの状態が参照され、必要な処理が行なわれる。

例えば、共有オブジェクトへのポインタpがある時に、右辺値のp->mを評価すると、pが含まれるセッションインスタンスを読出しロックしたあと、pがvalidかどうかを検査し、invalidならオブジェクトの実体をコピーしてくる。

左辺値のp->mを評価すると、pが含まれるセッションインスタンスを書込みロックしたあと、pがinvalidならオブジェクトの実体をコピーしてくる。

デッドロックになった場合は、トランザクション中に変更されたオブジェクトをすべてinvalidateして、イベント待ちの状態に戻る。デッドロックの検出は、後述するRSCサーバが行う。

3.3 楽観的ロックとインタラクション

インタラクションとは、複数のトランザクション(イベント処理)にまたがった処理である。インタラクションをとおして変更するオブジェクトには、楽観的なロックをかける。楽観的なロックとは、他の参加者をブロックしないロックで、コミットの際に成功だったか失敗だったかが判断できるというものである。

楽観的なロックでオブジェクトを変更するには、p->mの代わりにp->>mを式の左辺に記述する。これはRSCで拡張された演算子である[4]。演算子->による書込みロックが悲観的でセッションインスタンス単位であったのに対し、->>によるロックは楽観的でオブジェクト単位である。また、->による変更のコミットがトランザクション単位であるのに対し、->>による変更のコミットはインタラクション単位である。

インタラクションは、サービス関数[5]の1つである`rsccommit(`*flag*`)`によって終了する。*flag*によって、指定したオブジェクトだけに関するインタラクションを終了するのか、セッション内のすべてのインタラクションを終了するのかを指定する。

3.4 RSCサーバとリアクタ

RSCサーバは、参加者とセッションインスタンスの管理を行うと同時に、共有オブジェクトのキャッシュ、ロックの管理などを行う。デッドロックを検出し、参加者のトランザクションをアボートするのもRSCサーバの役割である。

また、各参加者にリンクされるリアクタが、RSCサーバからのメッセージに応答して適当な処理を行う。リアクタは、イベント処理ルーチンの1つとして位置づけられ、参加者プログラム(イベント処理)に対してノンプリエンプティブに実行される。

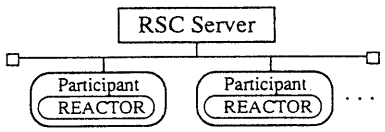

図4: RSCサーバと参加者プログラム

3.5 共有オブジェクトの管理

3.5.1 管理情報

共有クラスのスーパクラス(`rscshare`)では、次のようなメンバを定義する。これらの情報はリアクタやランタイムルーチンから参照される。

- オブジェクトID
- オブジェクトの状態
- メンバ定義表へのポインタ
- セッション構造体へのポインタ
- 演算子のオーバロード関数
- サービス関数
- 視点情報フラグ
- 楽観的ロックに関するフラグ

オブジェクトID、オブジェクトの状態については後述する。メンバ定義表(スキーマ)は、共有メンバの種類や先頭からのオフセット値などを格納した表である。また、セッション構造体には、セッションインスタンスの識別子などを格納する。これらの構造体の生成は、ランタイムルーチン・ジェネレータが行う。

演算子のオーバロード関数は、ソースコード・トランスレータが出力する演算子に対応するもので、共有オブジェクトの参照・変更を検出するものである。詳細は実現の章で述べる。

[4]RSCで拡張した言語仕様はこの演算子(->>)だけである。

[5]共有クラスのスーパクラスが定義するメンバ関数。

3.5.2 オブジェクト ID

オブジェクト ID は、ポインタと同じ語長で管理され、最下位ビットを常に 1 とすることで、有効なオブジェクトへのポインタと区別する (図 5)。

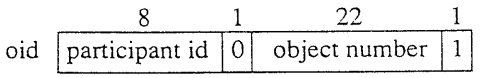

図 5: オブジェクト ID

オブジェクト ID には、オブジェクトを生成した参加者の ID が含まれる。参加者 ID はシステムでユニークなので、オブジェクト ID の発行を参加者でローカルに行うことができる。

3.5.3 共有オブジェクトの生成と削除

共有オブジェクトの生成・削除は、通常のオブジェクトと同様に `new`, `delete` で行う。ただし `new` の直後で、サービス関数 `rscroot(`*class-name*`)` あるいは `rscinit(`*class-name*`)` を呼び出す約束とする。

`rscroot()` と `rscinit()` は、管理情報の初期化を行う。さらに `rscroot()` は、参加者間でルートポインタを受け渡す機能を持ち、次のように動作する。

(1) 共有クラス C の共有ルートオブジェクトが登録されていれば、その ID をオブジェクト ID とする。
(2) 登録されていなければ新しい ID を生成し、それを C の共有ルートオブジェクトとして登録する。

3.5.4 共有オブジェクトの状態遷移

共有オブジェクトへのポインタがある時、その内容は∅か、キャッシュへの有効なポインタか、オブジェクト ID のどれかである。共有オブジェクトのキャッシュの状態には次のものがある。

invalid　キャッシュの内容が古くて、そのままでは参照できない状態。invalid 以外の状態を valid という。オブジェクトを valid にするとは、オブジェクトの実体を RSC サーバに問い合わせてキャッシュすることである。

clean　キャッシュの内容が新しく、そのまま参照できる状態で、かつローカルな変更を加えていないもの。

dirty　キャッシュの内容が新しく、そのまま参照できる状態で、かつローカルな変更を加えたもの。トランザクションの終了時には、dirty なオブジェクトに対応する他の参加者上のキャッシュを invalidate し、そのキャッシュを clean に遷移する。

private-dirty　楽観的ロックによって書込みを行っている状態。楽観的ロックが成功しているかどうかのフラグは、管理情報として保存する。

3.6 視点情報の共有

RSC が提供すべき視点情報は、プログラム中から RSC が自動的に抽出できるものである。そうでなければ、ユーザが共有オブジェクトとして実現できる。そこで、演算子-->>によって変更されているオブジェクトが注目されていると仮定する。

インタラクションが開始されると、->>で変更されたオブジェクトに関して、すべての参加者の視点情報フラグをセットする。視点情報フラグは、キャッシュが invalidate することでリセットされる。参加者プログラムは、このフラグを参照することで、インタラクション中のオブジェクトをハイライトで表示するなどの処理を行うことができる。

この機能によって、relaxed-WYSIWIS のグループウェアを実現できる。厳格な WYSIWIS を実現するには、ユーザの責任で視点情報を共有オブジェクトに書き込み、画面表示のたびにそれを参照すればよい。

4 ソースコード・トランスレータの実現

4.1 目的と方針

次のことを、ユーザにできるだけ負担をかけない方法で実現したい。

(1) 共有オブジェクトの参照・変更を検出すること。
(2) 共有オブジェクトの生成・削除を検出すること。
(3) 共有オブジェクトに、メンバ定義表へのポインタ、セッション構造体へのポインタを持たせること。

(1) は、式を変換してオーバロードした単項演算子を埋め込むことで実現する。(2) は、`new`, `delete` をオーバロードすることで実現できる。(3) は、トランスレーションで行うことも可能だが、なるべく処理系を RSC に依存させたくないので、`new` の後に `rscroot()` あるいは `rscinit()` を指定してもらうことにした。表への参照宣言やポインタの初期化は、これらのマクロ定義の中で行う。

4.2 変換の仕様

トランスレータを起動する時の引数として、共有対象となる (つまり変換の対象となる) クラスのルートクラスの名前を指定する。トランスレータは、指定された名前の親を持っているかどうかで、あるクラスが共有クラスであることを認識する。

式を読み込むとそれを逆ポーランド式に直し、それを評価しながら各項が右辺値なのか左辺値なのかなどのフラグを立てる。次に入力と同じ順序でトークンを出力する。このとき、共有クラスを型に持つ項について、次のような変換を行う。

共有クラス C があって、`C c; C* p;` がある場合:

左辺値の c.m を (*+c).m に変換する。
右辺値の c.m を (*-c).m に変換する。
左辺値の p->m を (+*p)->m に変換する。
右辺値の p->m を (-*p)->m に変換する。
左辺値の p->>m を (!*p)->m に変換する。
右辺値の p->>m を (-*p)->m に変換する。

変換後の式に埋め込まれる、単項の演算子 +, -, ! は、共有クラスのスーパクラスである rscshare がオーバロードするもので、rscshare* 型を返す関数として定義されている。

左辺値とは変更される項のことであり、見かけ上=の右側にあっても ++(p->m) などは左辺値である。また、右辺値とは参照される項のことであり、見かけ上=の左側にあっても (p->m, foo) の中の p->m などは右辺値である。また、共有クラスのメンバのアドレス取出し、および共有クラス型から別の型へのキャストは左辺値として扱われる。

上の例では、共有クラス C を型に持つ項 c, p はいずれも 1 つのトークンだが、これらは任意の式であってもよい。式が 2 つ以上のトークンの場合、(+*(p+1))->m のように式をカッコで囲む。

4.3 特別な機能

トランスレータは、_Ø から始まるいくつかのキーワードを認識して、特別な機能を提供する。

_Øgobble
次の "{" までトークンを読み飛ばす。

_Øtypeof **識別子**
識別子の型名に置き換わる。

_Øcat トークン$_1$ トークン$_2$
2 つのトークンを連結したトークンに置き換わる。

これらのキーワードを利用して、rscshare(), rscroot() などのマクロを定義する。たとえば、_Øgobble を使えば、親クラス名の位置に記述された rscshare(*session-name*) から、{ } の中に定義を差し込むことができる。また、_Øtypeof と_Øcat を使ってメンバ定義表の名前を生成することができ、メンバ定義表の extern 宣言をした後、メソッドの引数にそのポインタを渡すことなどが可能である。

4.4 実現の結果

C++の機能である演算子のオーバロードを利用したことと、特別な機能を持ったキーワードを定義することで、RSC に依存しないソースコード・トランスレータを実現できた。

実現したトランスレータは、型情報を出力する機能も持っており、これを読み込むことでランタイムルーチン・ジェネレータがスキーマの生成を行う。この型情報を、RSC 以外のオブジェクトデータベースのために使用することも可能である。

5 議論

グループウェアの構築として提案されているものの多くは、操作の単位でプログラムを抽象化し、操作を放送することでコンテキストの共有を実現する。

GROVE[1] は、放送した操作を前後して受信した場合に、変換操作で補正することでロックを不要としている。DistEdit[2] は、エディタにおける文字列の挿入・削除・置換をプリミティブとしており、ある時点で 1 人のユーザだけしか書込みできないように制限する。文献 [3][4] も同様に操作を単位としている。

これらはそれぞれ利点を持つが、いずれも操作によってプログラムを抽象化しており、一般的なグループウェアに対応できるものとは言えない。これに対し RSC は、プログラム中のオブジェクトそのものを共有の対象とする。したがって、プログラムを記述する際の自由度は非常に高い。

これらが提供するロック機能も、一般性のあるものとは言えない。GROVE における変換操作は複雑で、操作数がごく少数の場合にしか適用できない。また、DistEdit のような制限は好ましくない。これに対し RSC は、データベースと同様の手法で共有オブジェクトの一貫性を保証し、長いトランザクションに対応するために楽観的なロックも用意した。

6 おわりに

一般的な WYSIWIS 型グループウェアの記述を容易にするオブジェクト共有機構を設計し、そこで必要となるソースコード・トランスレータを実現した。

現在ランタイムルーチン・ジェネレータと RSC サーバの実現に取りかかっている。これらの実現が完了した後は、RSC を用いて分散テキストエディタを実現し、RSC の評価を行う予定である。

参考文献

[1] C. A. Ellis and S. J. Gibbs "Concurrency Control in Groupware Systems," Proceedings of ACM SIGMOD '89, pp. 399-407.

[2] M. J. Knister and A. Prakash: "DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors," Proceedings of CSCW '90, pp. 343-355.

[3] J. R. Rhyne and C. G. Wolf: "Tools for Supporting the Collaborative Process," Proceedings of UIST '92, pp. 161-170.

[4] A. Karsenty, C. Tronche and M. Beaudouin-Lafon: "GroupDesign: Shared Editing in a Heterogeneous Environment," Computing Systems (USENIX '93), Vol. 6, No. 2, Spr. 1993, pp. 167-195.