データベースミラーリング（DBM）利用時のアプリケーション設計・実装の注意点

マイクロソフト株式会社

DBM 利用時のアプリケーション設計・実装の注意点 Agenda

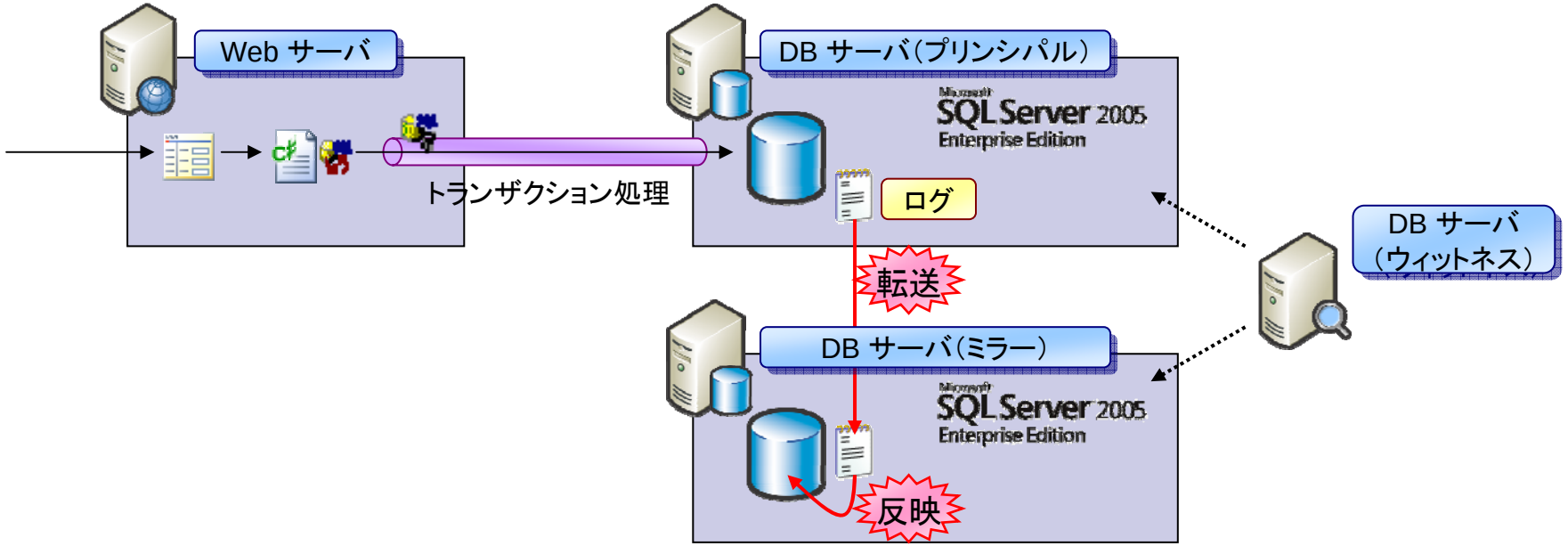
- DBM を利用する際の、アプリケーション設計・実装に関する注意点と制限事項について
  - データベースミラーリング（DBM）の仕様上の制限事項
  - 典型的なアプリケーション設計シナリオにおける DBM の利用
    - パターン 1. 単一データベースマニュアルトランザクション
    - パターン 2. クロスデータベースマニュアルトランザクション
    - パターン 3. 分散型 DTC トランザクション
    - パターン 4. ローカル型 DTC トランザクション
  - その他

DBM 利用時のアプリケーション設計・実装の注意点 Agenda

- 本資料は以下の KB に関する技術詳細解説資料です
  - 「データベースミラーリングをクロスデータベース トランザクション または分散トランザクションと併用する場合の問題」
    - http://support.microsoft.com/kb/926150/ja
  - "Database Mirroring and Cross-Database Transactions"
    - http://msdn2.microsoft.com/en-us/library/ms366279.aspx

DBM の仕様による制限事項

- DBM は、ログファイルの転送により動作します
  - プリンシパルデータベースからミラーデータベースへトランザクションログを転送することで、二つのデータベース上のデータを同期します
  - 内部動作の仕様上、システム構成やアプリケーション設計シナリオによっては DBM の利用に際して注意が必要です



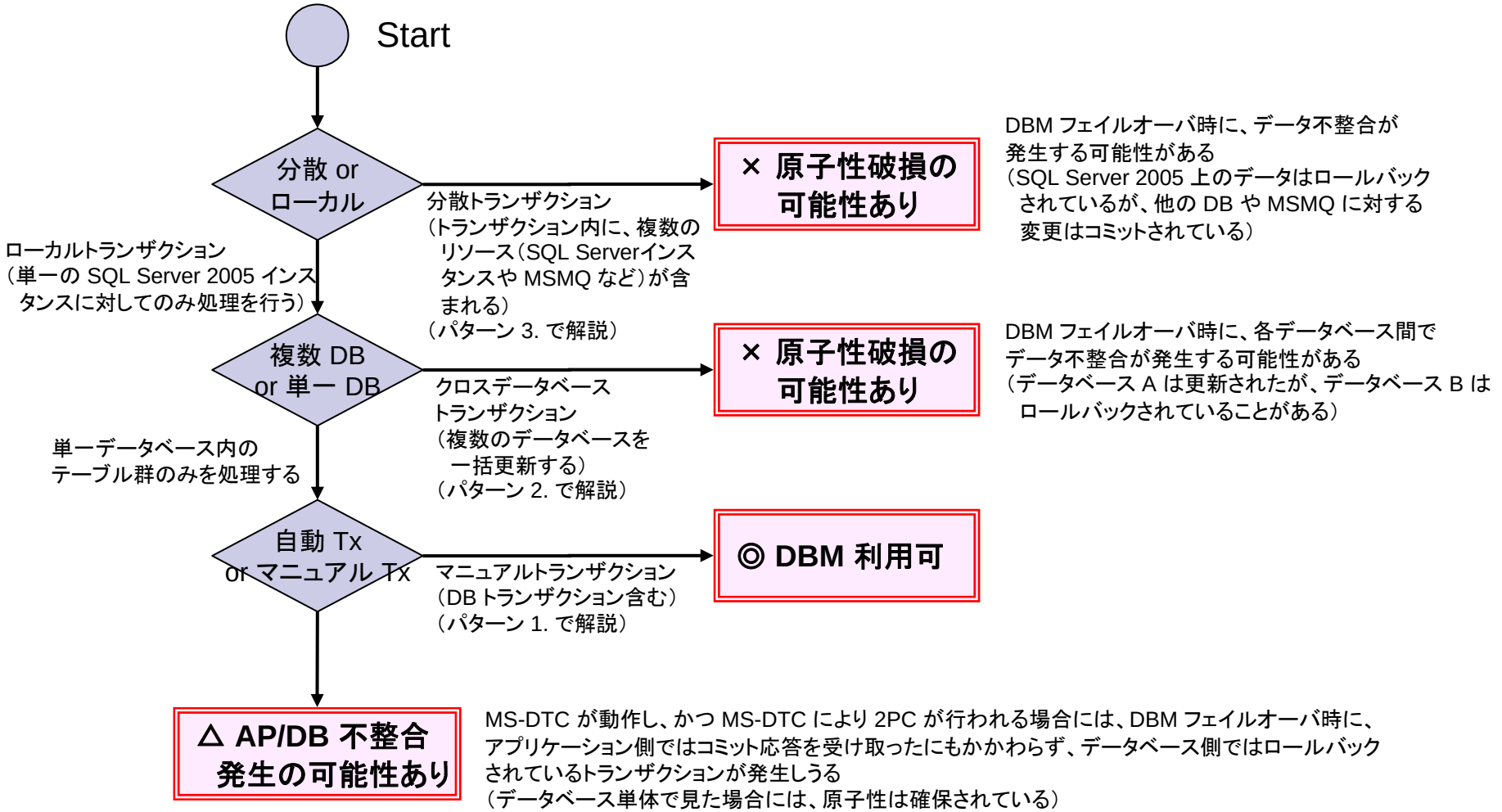
DBM の仕様による制限事項
ー仕様による主な設計制限事項

- DBM の仕様上の特性から、以下のような場合には DBM の利用に注意が必要な場合があります（→ 次ページ）
  - クロスデータベーストランザクション
  - 2PC を行う DTC トランザクション

    ※ これらの場合、トランザクション原子性などが失われることがあります

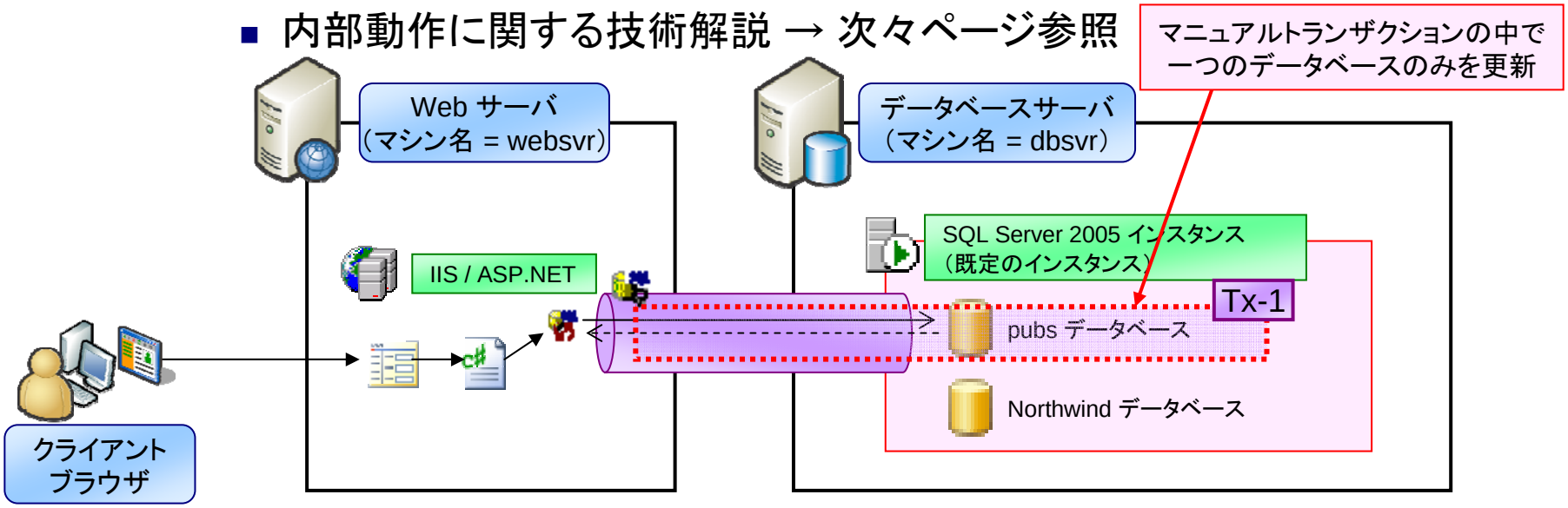
    ※ DBM 利用に問題があると判断された場合でも、DBM の代わりにマイクロソフトクラスタサービス（MSCS）を利用することで問題を解決できます
- 以降のスライドでは、以下の 4 つの代表的なアプリケーションシナリオを取り上げ、DBM 利用時の注意点を解説します
  - パターン 1. 単一データベースマニュアルトランザクション
  - パターン 2. クロスデータベースマニュアルトランザクション
  - パターン 3. 分散型 DTC トランザクション
  - パターン 4. ローカル型 DTC トランザクション

DBM の仕様による制限事項
ーDBM 利用に関する確認チャート



1. 単一データベースマニュアルトランザクション

- 単一データベースに対してマニュアルトランザクションでデータ更新を行うアプリケーションでは、DBM が利用できます
  - 具体例）pubs データベースに対して、マニュアルトランザクションでデータ更新を行うアプリケーション
    - このようなアプリケーションでは、DBM が利用できます
    - サンプルコード → 次ページ参照
    - 内部動作に関する技術解説 → 次々ページ参照



C#

```
// 接続文字列 : "Data Source=dbsvr;Initial Catalog=pubs;Persist Security Info=True;User ID=sa;Password=p&sswOrd"
SqlConnection sqlcon = new SqlConnection(ConfigurationManager.ConnectionStrings["pubsConnectionString"].ConnectionString);
SqlCommand sqlcmd1 = new SqlCommand("SELECT price FROM titles WITH (UPDLOCK) WHERE title_id=@title_id", sqlcon);
SqlCommand sqlcmd2 = new SqlCommand("UPDATE titles SET price = @val WHERE title_id=@title_id", sqlcon);
string title_id = "PS2106";
sqlcmd1.Parameters.AddWithValue("@title_id", title_id);
sqlcmd2.Parameters.AddWithValue("@title_id", title_id);

try
{
    sqlcon.Open();
    SqlTransaction sqltx = sqlcon.BeginTransaction(IsolationLevel.Serializable);
    sqlcmd1.Transaction = sqltx;
    sqlcmd2.Transaction = sqltx;
    SqlDataReader sqldr = sqlcmd1.ExecuteReader();
    if (sqldr.Read() == false)
    {
        sqldr.Close();
        sqltx.Rollback();
        lblResult.Text = "当該IDは存在しません。";
        return;
    }
    Decimal val = (Decimal)sqldr["price"];
    sqldr.Close();
    val = val + 1;
    sqlcmd2.Parameters.AddWithValue("@val", val);
    sqlcmd2.ExecuteNonQuery();
    sqltx.Commit();
    lblResult.Text = "データの価格を $1.00 上げました。";
}
finally
{
    sqlcon.Close();
}
```

① コネクションオープン

② トランザクション開始
（BEGIN TRANSACTION 送信）

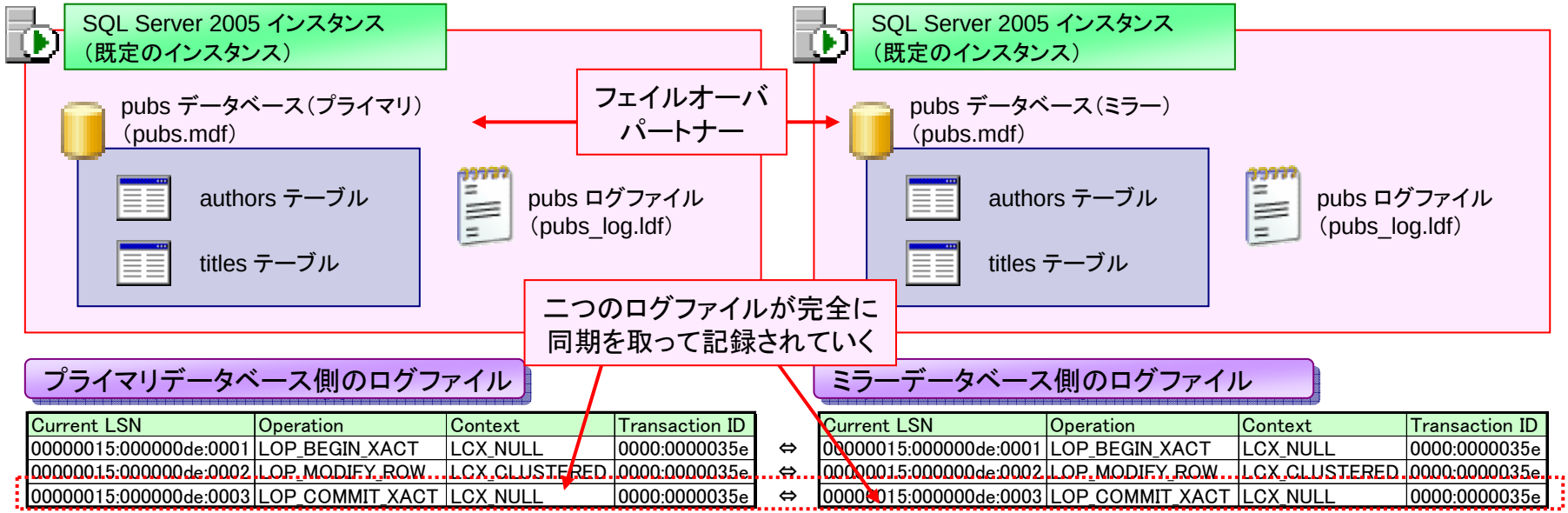
③ 各 SQL 文をトランザクションに
参加させる

④ トランザクションコミット
（COMMIT TRANSACTION 送信）

⑤ コネクションクローズ

1. 単一データベースマニュアルトランザクション ーDBM の内部動作

- DBM は、プライマリデータベースとミラーデータベース間でのログファイルの同期更新により機能しています
  - 動作モード＝同期（高度な保護）の場合、プライマリ／ミラー間でログファイルが完全に同期を取って記録されていきます
  - これにより、2 つのデータベースの完全なデータ同期が図られます



Current LSN	Operation	Context	Transaction ID
00000015:000000de:0001	LOP_BEGIN_XACT	LCX_NULL	0000:0000035e
00000015:000000de:0002	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:0000035e
00000015:000000de:0003	LOP_COMMIT_XACT	LCX_NULL	0000:0000035e

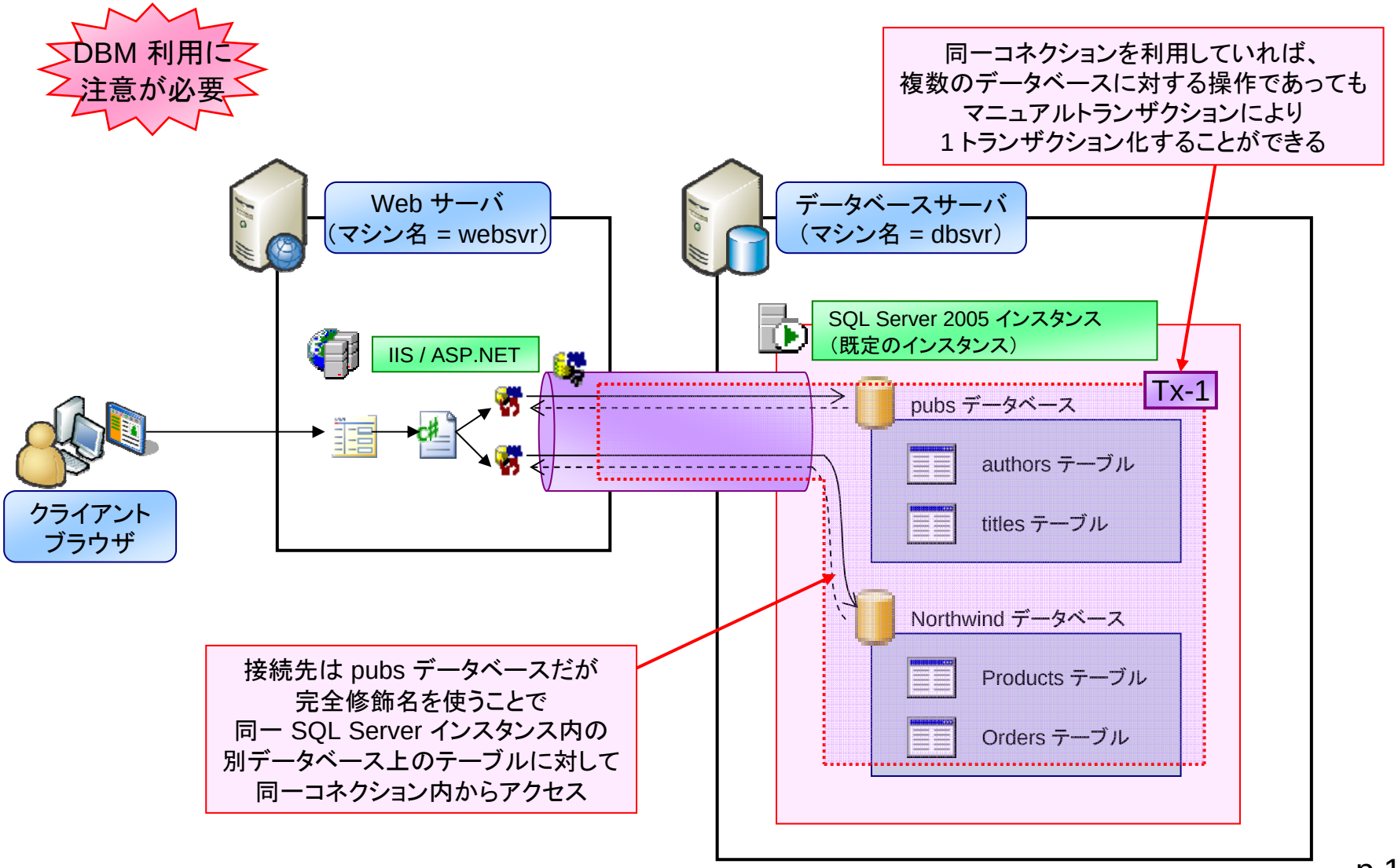
⇔

Current LSN	Operation	Context	Transaction ID
00000015:000000de:0001	LOP_BEGIN_XACT	LCX_NULL	0000:0000035e
00000015:000000de:0002	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:0000035e
00000015:000000de:0003	LOP_COMMIT_XACT	LCX_NULL	0000:0000035e

2. クロスデータベースマニュアルトランザクション

- 同一インスタンス内の複数データベースに対してデータ更新を行うアプリケーションでは、DBM 利用に注意が必要です
  - SQL Server 2005 では、同一インスタンス内に複数のデータベースを持つことができます
  - さらにテーブル名として完全修飾名を利用すると、別データベース上のテーブル上のデータをマニュアルトランザクションで更新できます
    - 完全修飾名＝[Northwind].[dbo].[Products]
    - 完全修飾名を利用すると、コネクションの接続先以外のデータベースを、同一マニュアルトランザクション内で処理することができます
  - しかし上述したようなマニュアルトランザクションを利用した場合は、DBM 利用時にトランザクションの原子性が失われる場合があります
    - DBM はデータベース単位でのフェイルオーバのみをサポートします（＝インスタンス単位のフェイルオーバをサポートしません）
    - このため、フェイルオーバ時に、データベース間でのデータ整合性が崩れる場合があります

2. クロスデータベースマニュアルトランザクション



C#

```
// 接続文字列 : "Data Source=dbsvr;Initial Catalog=pubs;Persist Security Info=True;User ID=sa;Password=p&sswOrd"
SqlConnection sqlcon = new SqlConnection(ConfigurationManager.ConnectionStrings["pubsConnectionString"].ConnectionString);
SqlCommand sqlcmd1 = new SqlCommand("UPDATE titles SET price = price + 1 WHERE title_id=@title_id", sqlcon);
SqlCommand sqlcmd2 = new SqlCommand("UPDATE [Northwind].[dbo].[Products] SET UnitPrice = UnitPrice + 1 WHERE
ProductID=@ProductID", sqlcon);
string title_id = "PS2106";
int ProductID = 1;
sqlcmd1.Parameters.AddWithValue("@title_id", title_id);
sqlcmd2.Parameters.AddWithValue("@ProductID", ProductID);

try
{
    sqlcon.Open();
    SqlTransaction sqltx = sqlcon.BeginTransaction(IsolationLevel.Serializable);
    sqlcmd1.Transaction = sqltx;
    sqlcmd2.Transaction = sqltx;

    int affectedRows1 = sqlcmd1.ExecuteNonQuery();
    int affectedRows2 = sqlcmd2.ExecuteNonQuery();

    if (affectedRows1 != 1 || affectedRows2 != 1)
    {
        sqltx.Rollback();
        lblResult.Text = "少なくとも一方のデータが存在していません。";
        return;
    }
    sqltx.Commit();
    lblResult.Text = "両方のデータの価格を $1.00 ずつ上げました。";
}
finally
{
    sqlcon.Close();
}
```

完全修飾名を利用すると
同一コネクション内で
別データベース内のテーブルを
操作できる

マニュアルトランザクションにより
トランザクション制御できる
（クロスデータベーストランザクション）

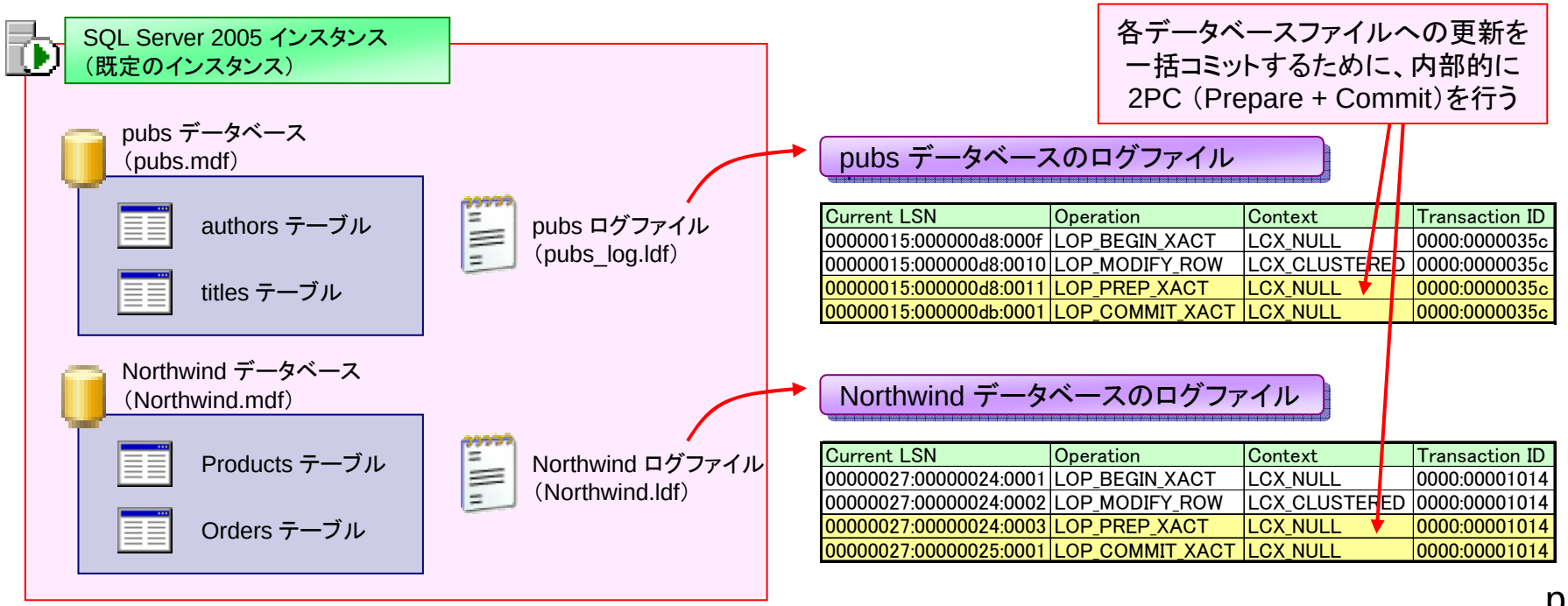
DBM 利用に
注意が必要

2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- クロスデータベースマニュアルトランザクションへの DBM 適用時に注意が必要なのは、以下の仕様上の理由によります
  - 詳細 → 次ページ以降のスライドで解説
  - クロスデータベーストランザクションは内部的に 2PC で処理されます
    - ログファイル上にて、Prepare / Commit レコードが立ちます
  - DBM を利用しない場合には、in-doubt トランザクションを正しく解決することができます
    - サービスがダウンした場合でも、再起動時に in-doubt トランザクションを正しく解決することができます
  - しかし、DBM を利用する場合、未コミットのトランザクションは一律でロールバックされます
    - ログレコードに Commit レコードが立っていないトランザクションは、すべてロールバックされます
  - このため、フェイルオーバのタイミングによってはデータベース間でのデータ不整合が発生する場合があります

2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- 詳細な動作原理の説明①（DBM を利用しない場合）
  - クロスデータベーストランザクションは内部的に 2PC で処理されます
    - データベースのログファイルは、データベース単位に作成されます
    - このため、pubs データベースと Northwind データベースに対するクロスデータベーストランザクションでは、内部的に 2PC 処理が行われます

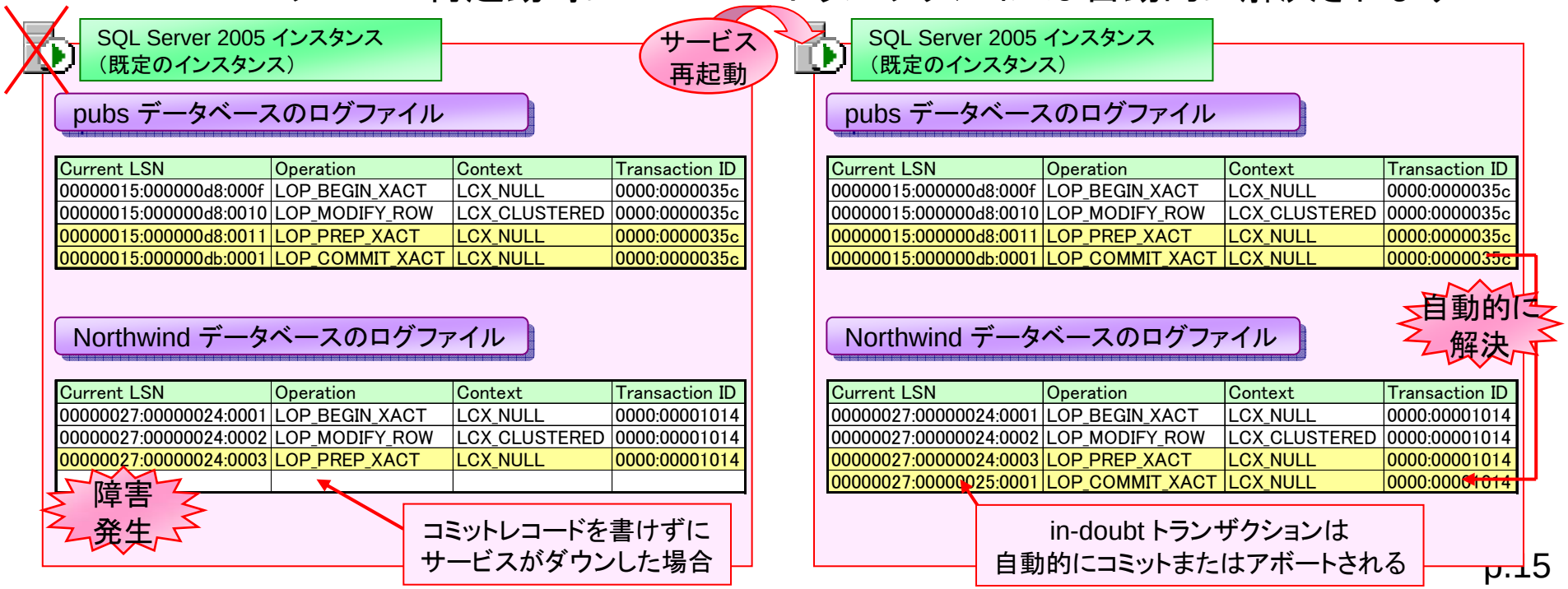


Current LSN	Operation	Context	Transaction ID
00000015:000000d8:000f	LOP_BEGIN_XACT	LCX_NULL	0000:0000035c
00000015:000000d8:0010	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:0000035c
00000015:000000d8:0011	LOP_PREP_XACT	LCX_NULL	0000:0000035c
00000015:000000db:0001	LOP_COMMIT_XACT	LCX_NULL	0000:0000035c

Current LSN	Operation	Context	Transaction ID
00000027:00000024:0001	LOP_BEGIN_XACT	LCX_NULL	0000:00001014
00000027:00000024:0002	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:00001014
00000027:00000024:0003	LOP_PREP_XACT	LCX_NULL	0000:00001014
00000027:00000025:0001	LOP_COMMIT_XACT	LCX_NULL	0000:00001014

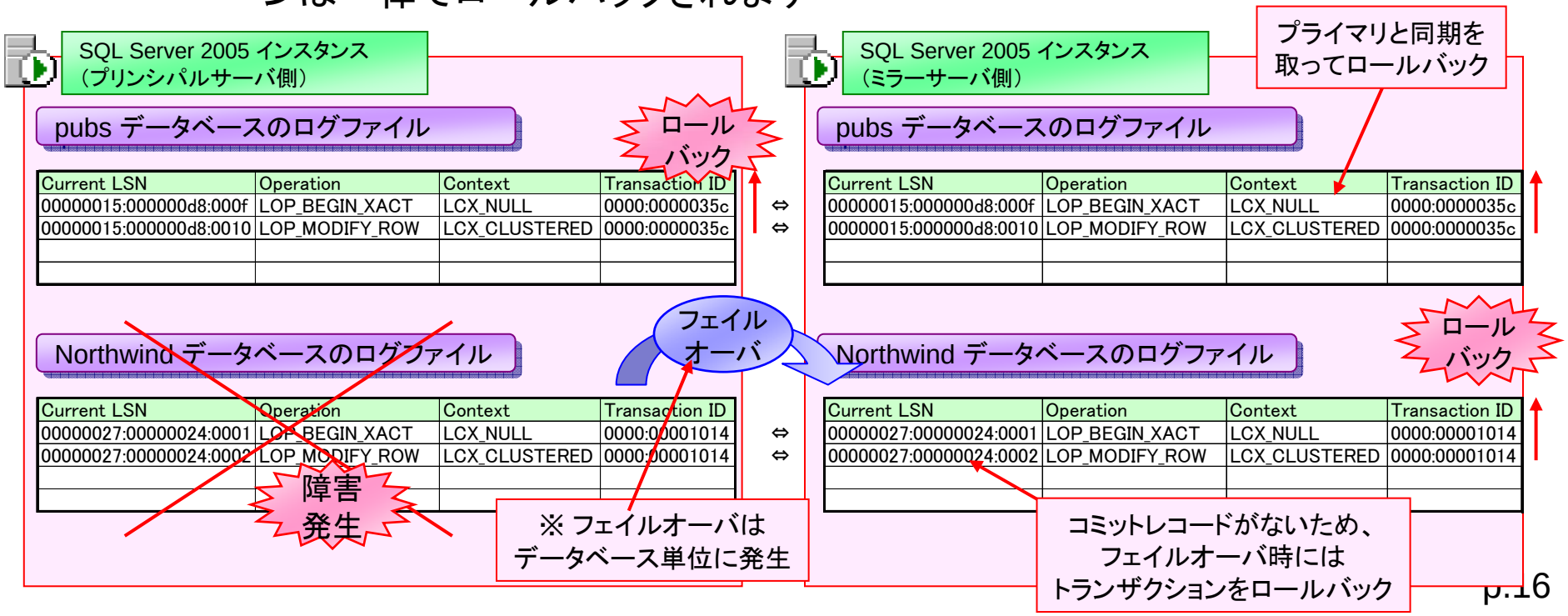
2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- 詳細な動作原理の説明②（DBM を利用しない場合）
  - Commit レコードを書けずに SQL Server インスタンスが異常終了しても、再起動時に in-doubt トランザクションは自動的に解決されます
    - in-doubt トランザクション＝ Prepared 状態に滞留してしまっているもの
    - サービス再起動時に in-doubt トランザクションは自動的に解決されます



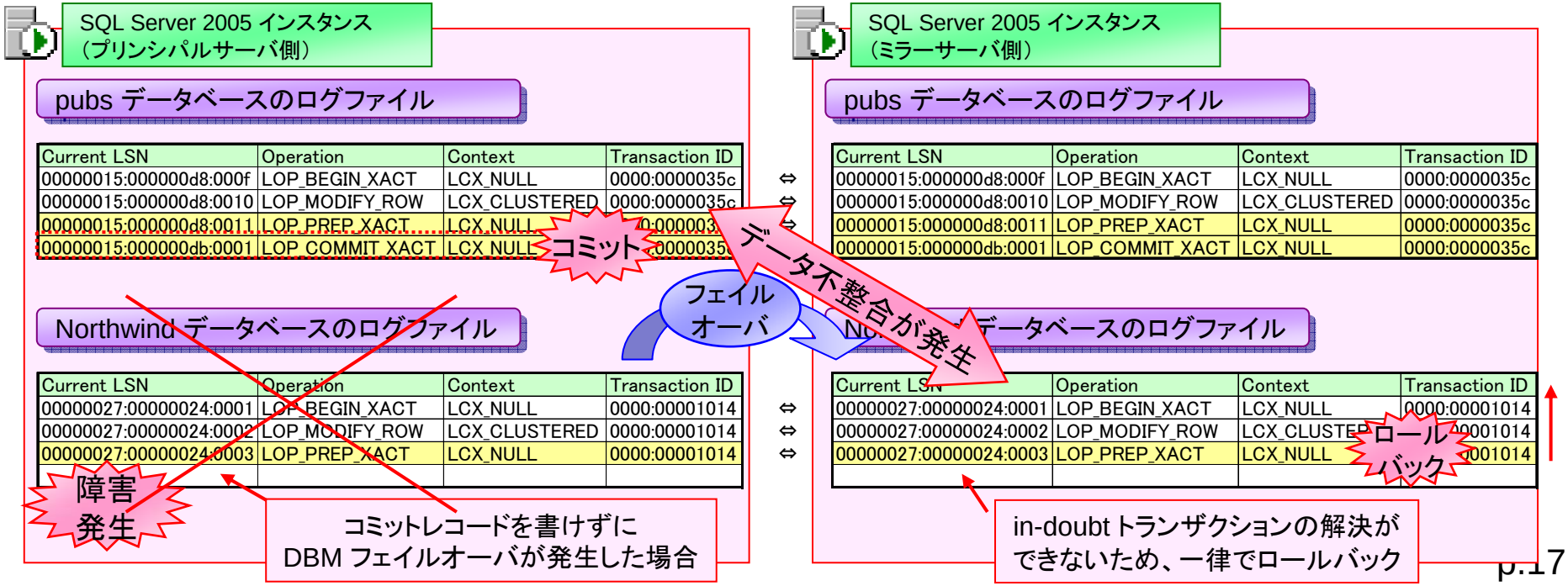
2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- 詳細な動作原理の説明③（DBM を利用する場合）
  - DBM では、フェイルオーバ時にログ上にコミットレコードが立っていないトランザクションは、一律でロールバックする仕様になっています
    - 単一データベースを前提とした設計であるため、未コミットトランザクションは一律でロールバックされます



2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- 詳細な動作原理の説明④（DBM を利用する場合）
  - DBM では in-doubt 状態でフェイルオーバしたトランザクションの自動解決ができないため、一律でロールバックを行います
    - この結果、一部のデータベースでは更新がコミットされ、一部のデータベースでは更新がロールバックされる事態が発生することがあります



2. クロスデータベースマニュアルトランザクション ー本パターンでの DBM 利用に注意が必要な理由

- まとめ
  - クロスデータベースマニュアルトランザクションを行った場合、DBM フェイルオーバの発生タイミングによっては以下の事象が発生します
    - 一部のデータベースではトランザクションがコミットされるが、一部のデータベースではトランザクションがロールバックされる
    - この結果、クロスデータベーストランザクションの原子性（Atomicity）が確保されない場合があります

    ※（参考）アプリケーション側では例外情報を受け取ることができるため、障害が発生したことを認識することは可能です

  ※ ログファイル上に Prepare / Commit レコードが立つ場合には、同様の問題が発生する場合があります
    - 同様の問題＝トランザクションとしてはコミットされているにもかかわらず、フェイルオーバ時に in-doubt トランザクションをロールバックしてしまう
    - このため、ログファイル上で in-doubt 状態が発生するようなトランザクション処理に対して DBM を適用する際には注意が必要になります

3. 分散型 DTC トランザクション

※（注意）分散型トランザクション ≠ DTC トランザクション
（トランザクション処理が MS-DTC により制御されていたとしても、操作しているリソースが単一である場合には、分散型トランザクションではありません。）

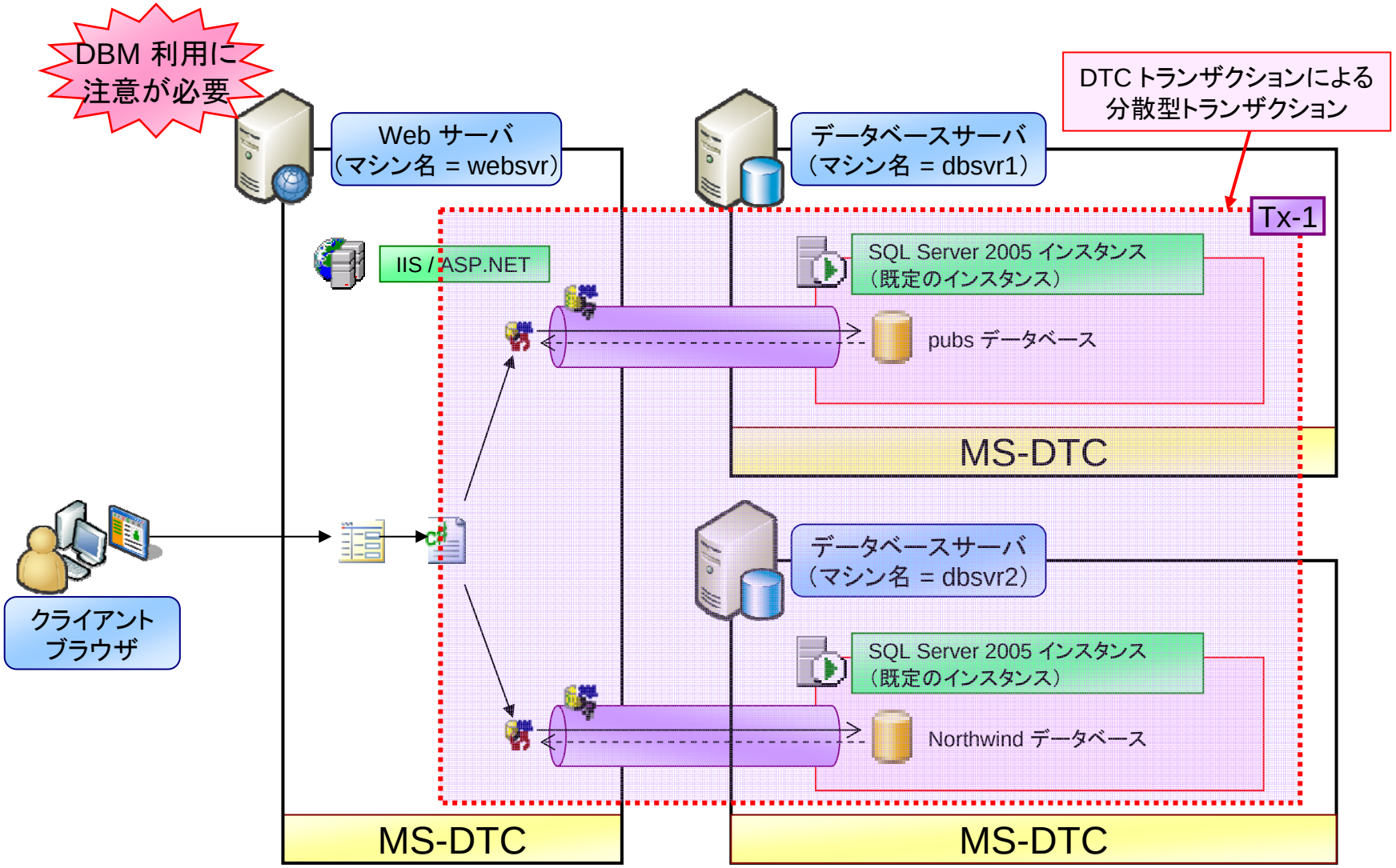
- 複数のデータベースが関与するような分散型トランザクション処理では、DTC トランザクションが必要になります
  - 分散型トランザクションとは、以下のようなものを指します
    - 1 つのトランザクション内において、複数のリソースを更新する処理
      - 例① 複数の SQL Server 2005 サーバ上のデータを同時に更新
      - 例② 同一サーバマシン内の異なるインスタンス上のデータを同時に更新
      - 例③ SQL Server 2005 上のデータと他社 DB 上のデータを同時に更新
      - 例④ SQL Server 2005 上のデータと MSMQ 上のデータを同時に更新
      - 例⑤ リンクサーバ機能を利用して他データベース上のデータを同時に更新
    - このような分散型トランザクション処理を行う場合には、MS-DTC によるトランザクション制御（DTC トランザクション）が必要になります

3. 分散型 DTC トランザクション

- 分散型 DTC トランザクションを利用する場合には、SQL Server 2005 の DBM 利用に注意が必要です
  - 分散型 DTC トランザクションでは 2PC 制御が行われるため、SQL Server 2005 のログ上に Prepared / Commit レコードが発生します
  - フェイルオーバのタイミングによっては、データベース間でのデータ不整合が発生する場合があります
    - in-doubt 状態でのフェイルオーバが発生した場合、データベース間でのデータ不整合が発生する場合があります

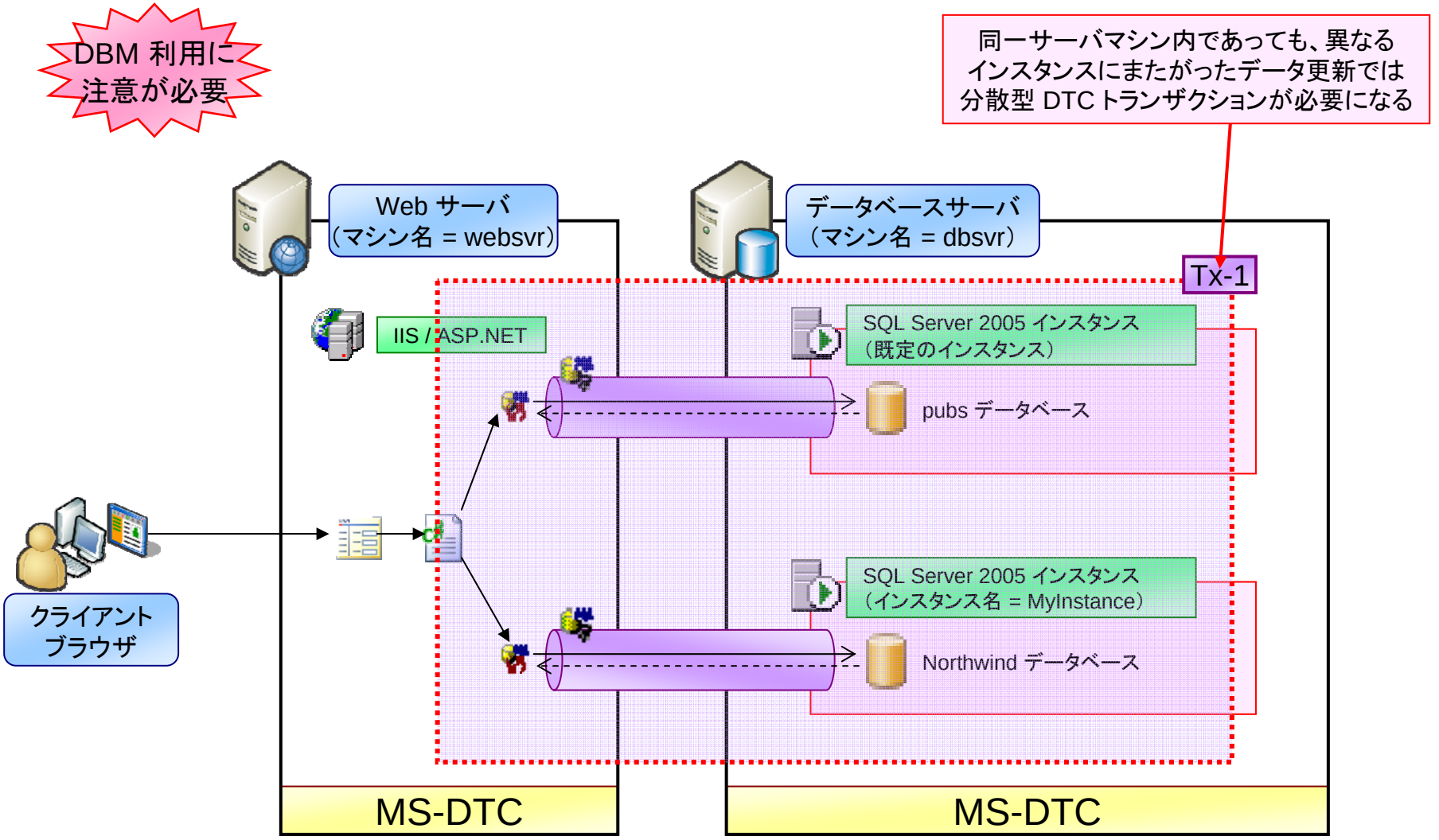
3. 分散型 DTC トランザクション

ー例① 複数の SQL Server 2005 サーバ上のデータを同時に更新



3. 分散型 DTC トランザクション

ー例② 同一サーバマシン内の異なるインスタンス上のデータを同時に更新



3. 分散型 DTC トランザクション

ー例① 複数の SQL Server 2005 サーバ上のデータを同時に更新

System.Transactions により
DTC トランザクションを利用

C#

```
// 接続文字列① : "Data Source=dbsvr1;Initial Catalog=pubs;Persist Security Info=True;User ID=sa;Password=p&ssw0rd"
// 接続文字列② : "Data Source=dbsvr2;Initial Catalog=Northwind;Persist Security Info=True;User ID=sa;Password=p&ssw0rd"
string connectionString1 = "Data Source=nakama90;Initial Catalog=pubs;Persist Security Info=True;User
ID=sa;Password=p&ssw0rd";
string connectionString2 = "Data Source=nakama78;Initial Catalog=Northwind;Persist Security Info=True;User
ID=sa;Password=p&ssw0rd";
```

Tx-1

```
using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew))
{
    SqlConnection sqlcon1 = new SqlConnection(connectionString1);
    SqlCommand sqlcmd1 = new SqlCommand("UPDATE titles SET price = price + 1 WHERE title_id=@title_id", sqlcon1);
    string title_id = "PS2106";
    sqlcmd1.Parameters.AddWithValue("@title_id", title_id);
    try
    {
        sqlcon1.Open();
        int affectedRows1 = sqlcmd1.ExecuteNonQuery();
        if (affectedRows1 != 1)
        {
            lblResult.Text = "titles テーブル上にデータがありません。";
            return;
        }
    }
    finally
    {
        sqlcon1.Close();
    }

    // 次ページへ続く
```

SQL Server
インスタンス①

DBM 利用に
注意が必要

3. 分散型 DTC トランザクション

ー例① 複数の SQL Server 2005 サーバ上のデータを同時に更新

C#

```
SqlConnection sqlcon2 = new SqlConnection(connectionString2);
SqlCommand sqlcmd2 = new SqlCommand("UPDATE Products SET UnitPrice = UnitPrice + 1 WHERE ProductID=@ProductID", sqlcon2);
int ProductID = 1;
sqlcmd2.Parameters.AddWithValue("@ProductID", ProductID);
try
{
    sqlcon2.Open();
    int affectedRows2 = sqlcmd2.ExecuteNonQuery();
    if (affectedRows2 != 1)
    {
        lblResult.Text = "Products テーブル上にデータがありません。";
        return;
    }
}
finally
{
    sqlcon2.Close();
}

scope.Complete();
}
```

SQL Server
インスタンス②

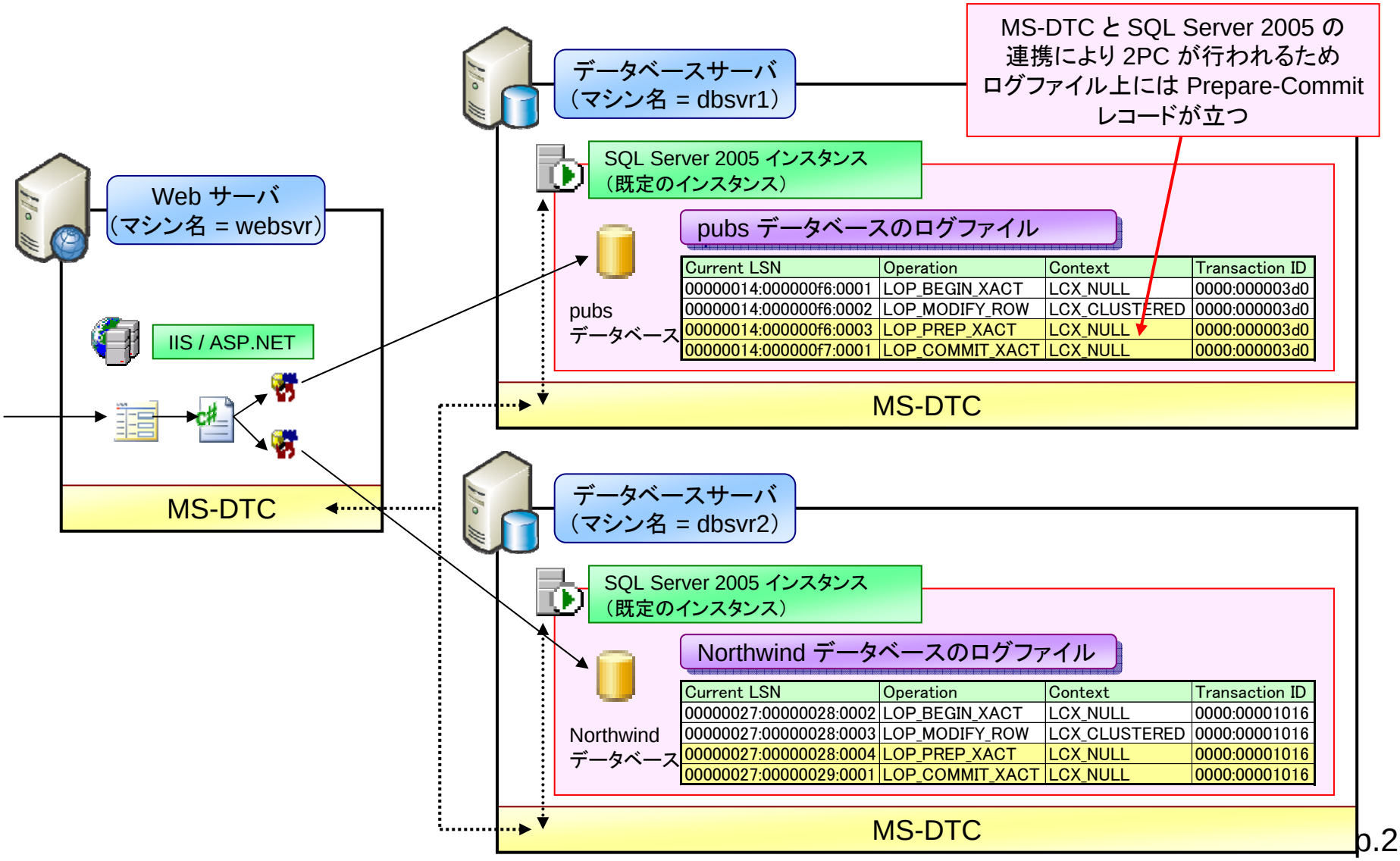
Tx-1

```
lblResult.Text = "データを更新しました。";
```

DBM 利用に
注意が必要

3. 分散型 DTC トランザクション

ー例① 複数の SQL Server 2005 サーバ上のデータを同時に更新



3. 分散型 DTC トランザクション

- まとめ
  - 分散型 DTC トランザクションを利用する場合には、DBM の利用に注意が必要です
    - 分散型 DTC トランザクションでは、MS-DTC から各リソースマネージャに対して 2PC 要求が行われます
    - この際、SQL Server 2005 は MS-DTC からの 2PC 要求を処理するために、ログレコード上に Prepare / Commit レコードを立てます
    - in-doubt 状態のトランザクションがフェイルオーバすると一律ロールバックされるため、データ不整合が発生する場合があります

  ※ この問題は、DTC トランザクションの作成方法に依存しません
    - DTC トランザクションを制御する方法は大別して 3 通りあります
      - System.Transactions, ServiceDomain, ServicedComponent（後述）
    - どの方法を利用しても DBM の利用には注意が必要です
      - 分散型トランザクションである限り、MS-DTC は 2PC を行います
      - 2PC が行われる場合、Prepare / Commit レコードが立ちます
      - この結果、フェイルオーバタイミングによって不整合が起こる場合が生じます p.26

4. ローカル型 DTC トランザクション

- DTC トランザクション利用時でも、操作対象リソースが単一の場合には、DBM が問題なく適用できることもあります
  - DTC トランザクションを利用しても、操作対象リソースが唯一である場合には、ログに Prepare / Commit が立たない場合があります
    - この場合には、DBM を利用することができる
    - ただし複数の条件があるため、以下の解説を正しく理解した上でご利用ください
  - これについて、以下の手順で解説します
    - TransactionScope + TableAdapter の組み合わせパターン
    - 上述のパターンで注意が必要になる理由（MS-DTC 非同期応答動作）
    - 回避策① コネクション変更による LCT の利用
    - 回避策② ServiceDomain の利用による SPC 最適化
    - 回避策③ セントラル MS-DTC サーバの利用による SPC 最適化
    - （非回避策）④ アプリケーションログによる運用対処

4. ローカル型 DTC トランザクション
－TransactionScope + TableAdapter の組み合わせパターン

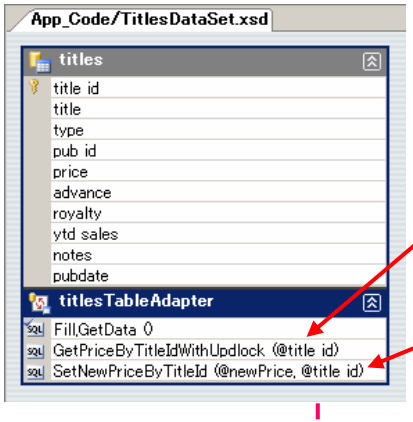
- Visual Studio 2005 でアプリケーションを開発する場合には、以下のパターンを利用することが多い
  - データアクセスコンポーネントの開発 → テーブルアダプタを利用
  - トランザクション制御 → 自動トランザクションを利用

  ※ 開発例 → 次ページ参照

- このパターンをそのまま適用した場合には、DBM の利用に注意が必要になります
  - 仮に、トランザクションスコープ内でアクセスするデータベースが単一であったとしても、DBM の利用には注意が必要
    - アプリケーションではコミットしたと認識したにもかかわらず、データベース側ではトランザクションがロールバックされている場合があります

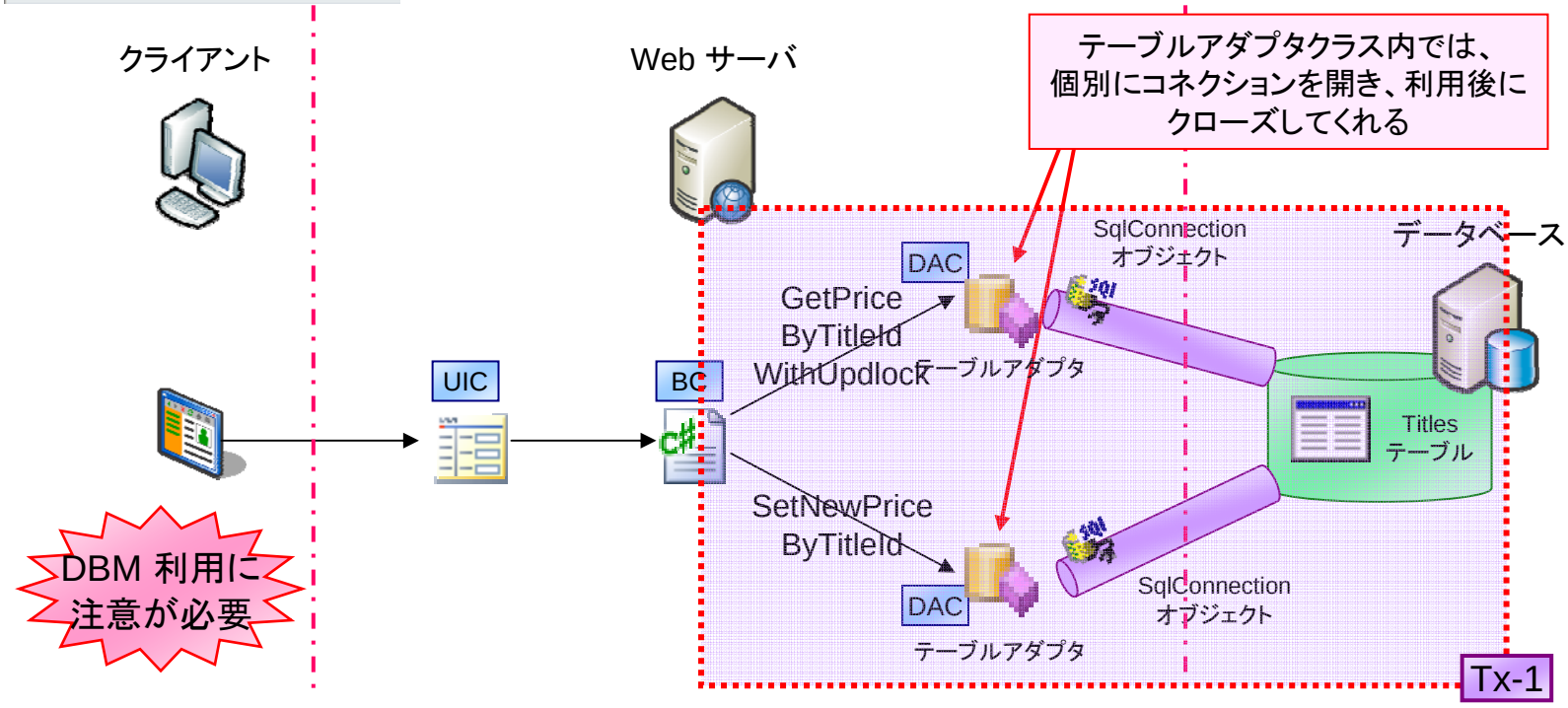
  ※ このケースで発生する問題と、その問題が発生する理由は、前述の 1.～3. とは異なります（→ 後述）

4. ローカル型 DTC トランザクション
ーTransactionScope + TableAdapter の組み合わせパターン



① 単一データ値の取得を行う SELECT 文
SELECT price FROM titles **WITH (UPDLOCK)**
WHERE title_id=@title_id

② UPDATE 文
UPDATE titles SET price = @newPrice
WHERE title_id=@title_id



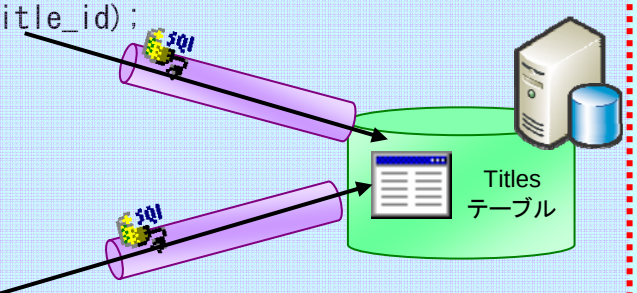
4. ローカル型 DTC トランザクション
－TransactionScope + TableAdapter の組み合わせパターン

C#

```
string title_id = "PS2106";
TitlesDataSetTableAdapters.titlesTableAdapter ta = new TitlesDataSetTableAdapters.titlesTableAdapter();
```

Tx-1

```
using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew))
{
    decimal? price = (decimal?)ta.GetPriceByTitleIdWithUpdlock(title_id);

    if (price == null)
    {
        lblResult.Text = "データがありません。";
        return;
    }

    decimal newPrice = price.Value + 1;
    ta.SetNewPriceByTitleId(newPrice, title_id);

    scope.Complete();
}
```

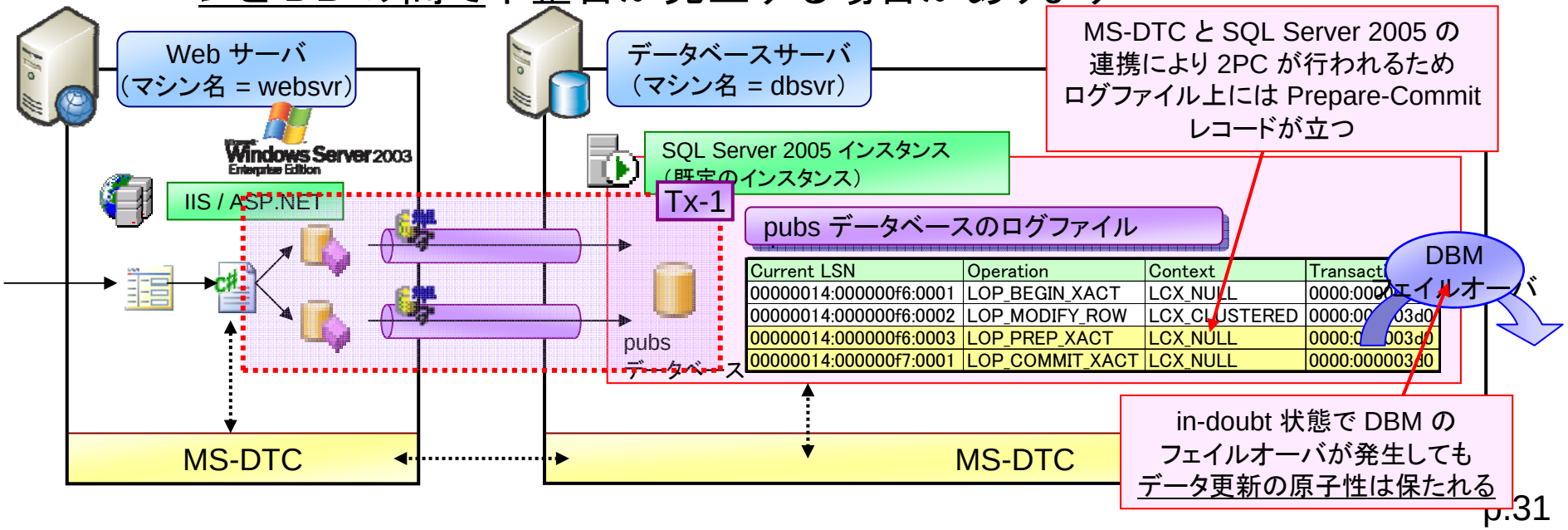



```
lblResult.Text = "価格を $1.00 上げました。";
```

DBM 利用に
注意が必要

4. ローカル型 DTC トランザクション
－TransactionScope + TableAdapter の組み合わせパターン

- パターン 3. と異なりログファイルが単一であるため、DBM フェイルオーバが発生してもデータ不整合は発生しません
  - in-doubt 状態でデータベースがフェイルオーバしても、データベースが単一であればデータベース間でのデータ不整合は発生しません
  - しかし、MS-DTC のコミットシーケンスの仕様により、アプリケーションと DB の間で不整合が発生する場合があります



4. ローカル型 DTC トランザクション
－TransactionScope + TableAdapter の組み合わせパターン

- アプリ／DB 間での不整合が発生するケース
  - 万が一、以下のような状況が発生した場合
    - MS-DTC がコミットを決定した後、SQL Server 2005 に非同期でコミット要求を発行し、アプリケーションに対してコミット応答を返します
    - この直後に SQL Server 2005 側で障害が発生し、DBM フェイルオーバが発生すると、当該トランザクションはロールバックされます

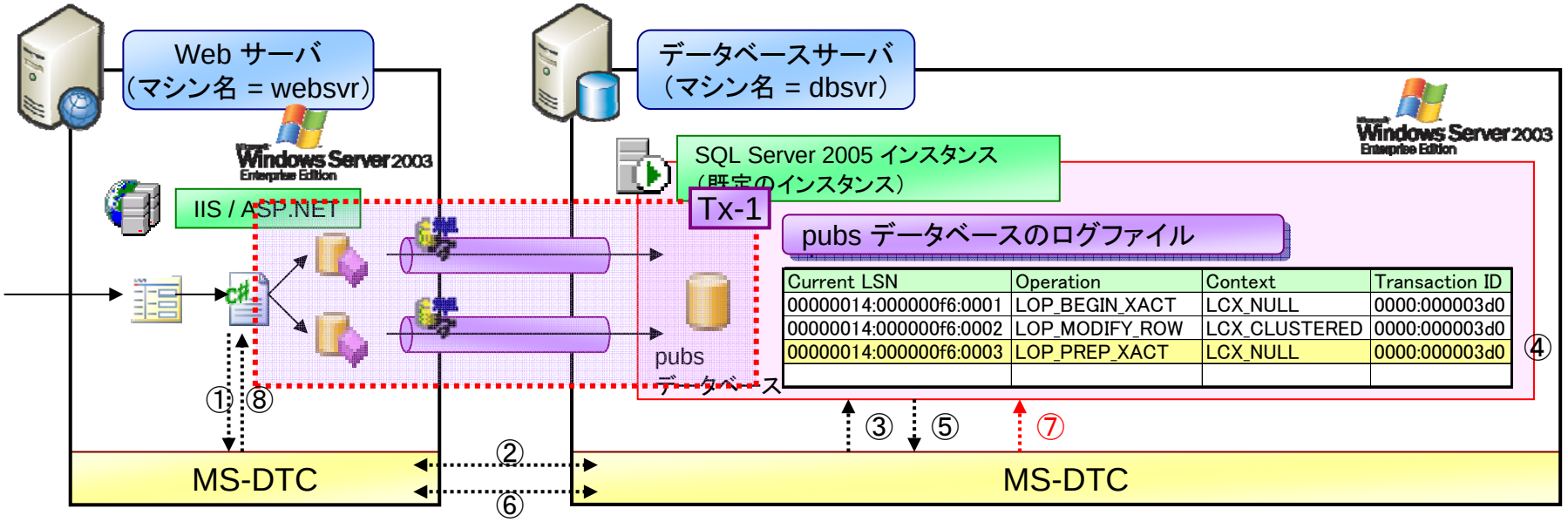
    ※ 詳細な処理シーケンス → 次ページ参照
  - このような場合、次のような状況となります
    - トランザクションの原子性は保たれる
      - データベースのみを見た場合には、データ不整合は発生しません
    - しかし、アプリケーションとの間の認識ズレが発生します
      - アプリケーション側はトランザクションコミットと判定しているにもかかわらず、データベース側ではトランザクションがロールバックされています
  - このため、DBM の利用に注意が必要になります

4. ローカル型 DTC トラ
ーTransactionScope + Tal

※（参考）この問題は MSCS では発生しません。
アプリには MS-DTC のログ上にコミット／アボートが記録された上で応答が返されますが、in-doubt 状態でフェイルオーバが発生しても、MS-DTC のログは保護されているため、それを元にして、アプリに対して返した答えと同じようにコミット／アボートができます。
しかし DBM では、フェイルオーバ時に MS-DTC ログを参照して in-doubt 状態を解決する、という動作をすることができません。

【詳細な動作シーケンス】

① TransactionScope オブジェクトから MS-DTC に対してコミット要求が発行される
② 各サーバの MS-DTC 間で連携が行われる
③ MS-DTC からリソースマネージャ（SQL Server 2005）に対して Prepare 要求が発行される
④ SQL Server 2005 側では、ログ上に Prepared レコードを立てる
⑤ MS-DTC に対して Prepared 応答を返す
⑥ MS-DTC は各リソースマネージャからの通知結果を元に、コミット／アボートを決定する
⑦ コミットを決定した場合、MS-DTC は各リソースマネージャに対して非同期でコミット要求を発行する
⑧ MS-DTC はコミット完了応答を待たずにアプリケーションに対して 2PC コミット完了応答を返す



4. ローカル型 DTC トランザクション ーDBM 対応アプリケーションとするための方法

- 前述のようなアプリケーションでも、以下のような対策を行うと、不整合問題が発生する可能性をなくすことができます
  - 回避策① コネクション変更による LCT の利用
  - 回避策② ServiceDomain の利用による SPC 最適化
  - 回避策③ セントラル MS-DTC サーバによる SPC 最適化
  - （非回避策）④ アプリケーションログによる運用対処

※ 要コード修正

- 以下に順を追って解説します

※（注意）以下に述べる回避策は、トランザクションスコープ内で操作するデータベースが単一の場合に限られることに注意してください

- 接続先が一つの SQL Server 2005 インスタンスでも、スコープ内で操作するデータベースが複数ある場合は、前述の 2. と同じ状況となります
- （SQL Server 2005 インスタンス内部でクロスデータベーストランザクションとなってしまうため）

4. ローカル型 DTC トランザクション ー回避策① コネクション変更による LCT の利用

- TableAdapter が利用するコネクションを揃えて事前にオープンしておくことで、MS-DTC の動作を抑えることができます
  - この方法を使うと、TransactionScope オブジェクトによる自動トランザクションはマニュアルトランザクションにより処理されます
  - これにより、DBM を利用することができるようになります

- コード修正例 → 次ページ、次々ページ参照

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

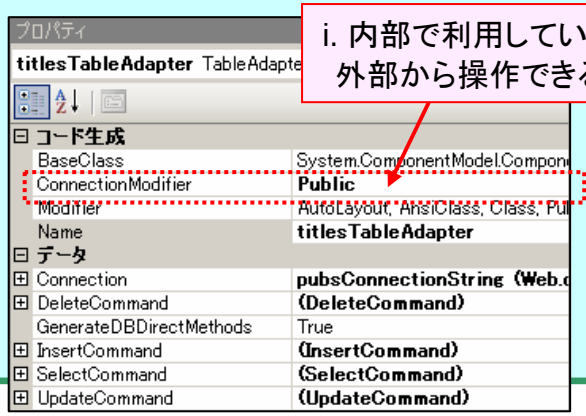
C#

```
string title_id = "PS2106";
TitlesDataSetTableAdapters.titlesTableAdapter ta = new TitlesDataSetTableAdapters.titlesTableAdapter();

using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew))
{
    try
    {
        ta.Connection.Open();

        decimal? price = (decimal?)ta.GetPriceByTitleIdWithUpdlock(title_id);
        if (price == null)
        {
            lblResult.Text = "データがありません。";
            return;
        }

        decimal newPrice = price.Value + 1;
        ta.SetNewPriceByTitleId(newPrice, title_id);
        scope.Complete();
    }
    finally
    {
        ta.Connection.Close();
    }
}
lblResult.Text = "価格を $1.00 上げました。";
```

ii. 事前に TableAdapter が持つ接続を開いておくことで、内部的な自動 Open() / Close() 処理を防止する

TableAdapter 上のメソッドを呼び出しても
接続の Open() / Close() は行われないため
同一接続が利用し続けられ、
結果として昇格が発生しなくなる
（＝マニュアルトランザクションと同等の処理になる）

i. 内部で利用しているコネクションを外部から操作できるようにしておく



4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

C#

```
string au_id = "172-32-1176";
string title_id = "PS2106";
AuthorsDataSetTableAdapters.authorsTableAdapter ata = new
AuthorsDataSetTableAdapters.authorsTableAdapter();
TitlesDataSetTableAdapters.titlesTableAdapter tta = new TitlesDataSetTableAdapters.titlesTableAdapter();

using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew))
{
    SqlConnection sqlcon = ata.Connection = tta.Connection;
    try
    {
        sqlcon.Open();

        ata.SetNewContractByAuId(true, au_id);
        tta.SetNewPriceByTitleId(25, title_id);

        scope.Complete();
    }
    finally
    {
        sqlcon.Close();
    }
}

lblResult.Text = "データを更新しました。";
```

複数の TableAdapter を利用する場合は
内部で利用するコネクションオブジェクトを
揃えてからオープンする
（既定では、TableAdapter オブジェクトの
インスタンスごとに異なる接続
オブジェクトインスタンスが利用される）

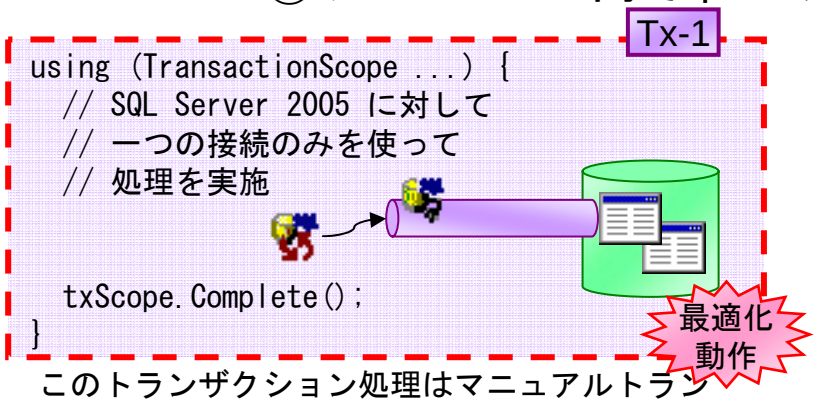
4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

- （参考）なぜこの方法により MS-DTC の動作を抑えられるのかに関しては、以下のポイントを理解する必要があります
  - A. System.Transactions の昇格動作と LCT
  - B. TableAdapter の内部仕様

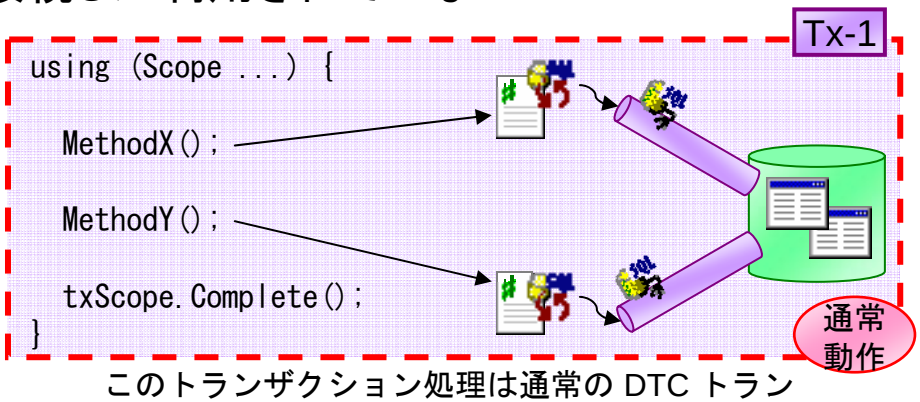
- 以下に、順を追って解説します

4. ローカル型 DTC トランザクション ー回避策① コネクション変更による LCT の利用

- A. System.Transactions の昇格動作と LCT
  - SQL Server 2005 に対する最初の SQL 処理は、後から昇格可能な LCT（Lightweight Committable Transaction）により処理されます
    - スコープ内で SQL Server 2005 に対する単一接続しか利用していない場合には、マニュアルトランザクションと同様に処理されます
  - このため、以下の 2 つの条件が満たされる場合には、Transaction Scope オブジェクトを利用しても MS-DTC が動作しません
    - ① 利用するデータベースサーバが SQL Server 2005
    - ② かつスコープ内で単一の接続しか利用されていない



このトランザクション処理はマニュアルトランザクションと同等に処理される（MS-DTC は動作せず）



このトランザクション処理は通常の DTC トランザクション処理となる（MS-DTC が動作する）

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

- A. System.Transactions の昇格動作と LCT（続き）
  - ただし、この昇格機能には以下の制限があります
    - 同一 Connection オブジェクトを利用する場合でも、TransactionScope 内で Open() / Close() を繰り返すと DTC トランザクションに昇格します
    - このため、昇格防止のためには単一接続を開きっぱなしで利用する必要があります

C#

```
SqlConnection sqlcon = new SqlConnection("...");
SqlCommand sqlcmd1 = new SqlCommand("...", sqlcon);
SqlCommand sqlcmd2 = new SqlCommand("...", sqlcon);

using (TransactionScope scope =
 new TransactionScope(TransactionScopeOption.RequiresNew))
{
    sqlcon.Open();

    sqlcmd1.ExecuteNonQuery();
    sqlcmd2.ExecuteNonQuery();

    sqlcon.Close();

    scope.Complete()
}
```

Tx-1

LCT により
マニュアルトランザクション
同様に処理される

C#

```
SqlConnection sqlcon = new SqlConnection("...");
SqlCommand sqlcmd1 = new SqlCommand("...", sqlcon);
SqlCommand sqlcmd2 = new SqlCommand("...", sqlcon);

using (TransactionScope scope =
 new TransactionScope(TransactionScopeOption.RequiresNew))
{
    sqlcon.Open();
    sqlcmd1.ExecuteNonQuery();
    sqlcon.Close();

    sqlcon.Open();
    sqlcmd2.ExecuteNonQuery();
    sqlcon.Close();

    scope.Complete();
}
```

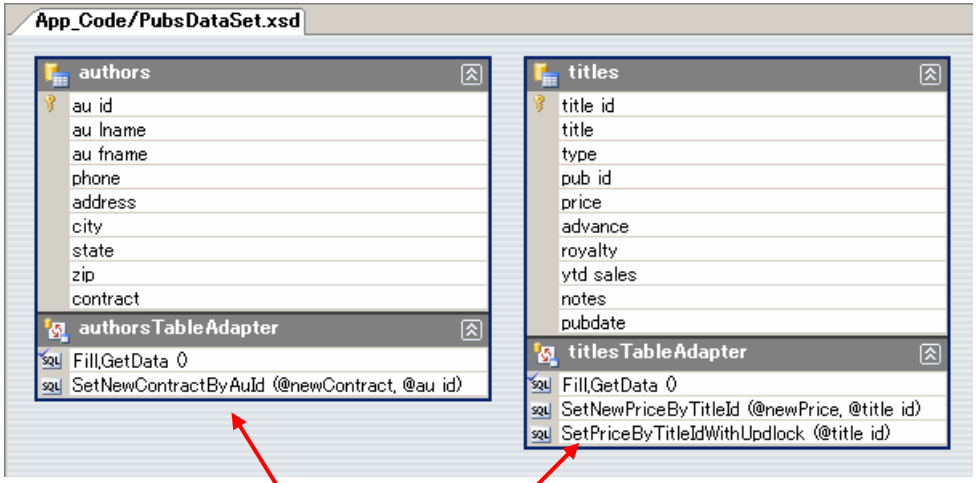
Tx-1

DTC
昇格

DBM 利用に
注意が必要

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

- B. TableAdapter の内部仕様
  - TableAdapter オブジェクトごとに Connection オブジェクトを持つ
  - メソッド呼び出しのつど接続を開き、クエリを実行後、クローズする
  - しかし、事前に接続が開かれていた場合には、そのままクエリを実行し、接続をクローズすることはしない（→ 次ページ参照）



同一 .xsd ファイル上に定義されている場合でも
TableAdapter ごとに Connection オブジェクトを持つ

自動生成された TableAdapter オブジェクト上のコード例

C#

```
[System.Diagnostics.DebuggerNonUserCodeAttribute()]
[System.ComponentModel.Design.HelpKeywordAttribute("vs.data.TableAdapter")]
[System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update,
false)]
public virtual int SetNewContractByAuId(bool newContract, string au_id) {
    System.Data.SqlClient.SqlCommand command = this.CommandCollection[1];
    command.Parameters[0].Value = ((bool)(newContract));
    if ((au_id == null)) {
        throw new System.ArgumentNullException("au_id");
    }
    else {
        command.Parameters[1].Value = ((string)(au_id));
    }
    System.Data.ConnectionState previousConnectionState = command.Connection.State;
    if (((command.Connection.State & System.Data.ConnectionState.Open)
                != System.Data.ConnectionState.Open)) {
        command.Connection.Open();
    }
    int returnValue;
    try {
        returnValue = command.ExecuteNonQuery();
    }
    finally {
        if ((previousConnectionState == System.Data.ConnectionState.Closed)) {
            command.Connection.Close();
        }
    }
    return returnValue;
}
```

事前にオープンされていなかった場合に限り
接続を開き、終了後に接続を閉じる

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

- B. TableAdapter の内部仕様（続き）
  - このため、TransactionScope 内で TableAdapter を利用する場合、
    - 既定の動作を利用してメソッドを複数回呼び出すと、TransactionScope の昇格が発生し、DTC トランザクションとして処理されます
    - しかし事前に手作業で TableAdapter オブジェクト内の接続を開いておくと複数回メソッド呼び出しを行っても昇格が発生しません

C#

```
titlesTableAdapter ta = new titlesTableAdapter();

using (TransactionScope scope =
 new TransactionScope(TransactionScopeOption.RequiresNew))
{
    ta.GetPriceByTitleIdWithUpdlock(title_id);
    ta.SetNewPriceByTitleId(newPrice, title_id);
    scope.Complete();
}
```

DTC 昇格

DBM 利用に注意が必要

C#

```
titlesTableAdapter ta = new titlesTableAdapter();

using (TransactionScope scope =
 new TransactionScope(TransactionScopeOption.RequiresNew))
{
    try
    {
        ta.Connection.Open();
        ta.GetPriceByTitleIdWithUpdlock(title_id);
        ta.SetNewPriceByTitleId(newPrice, title_id);
        scope.Complete();
    }
    finally
    {
        ta.Connection.Close();
    }
}
```

事前に接続を開いておく

LCT により マニュアルトランザクション 同様に処理される

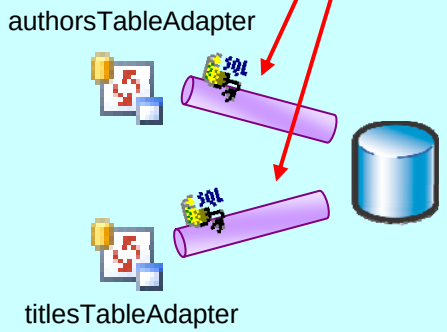
4. ローカル型 DTC トランザクション ー回避策① コネクション変更による LCT の利用

- B. TableAdapter の内部仕様（続き）
  - 複数の TableAdapter オブジェクトを利用する場合、オブジェクトごとに Connection を保有しているため、揃える必要があります
    - Open() する前に各 TableAdapter オブジェクトを操作し、実質的に一本の接続を開きっぱなしで利用するように調整してください

C#

```
string au_id = "172-32-1176";
string title_id = "PS2106";
authorsTableAdapter ata = new authorsTableAdapter();
titlesTableAdapter tta = new titlesTableAdapter();
using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew)) {
    SqlConnection sqlcon = ata.Connection = tta.Connection;
    try {
        sqlcon.Open();
        ata.SetNewContractByAuId(true, au_id);
        tta.SetNewPriceByTitleId(25, title_id);
        scope.Complete();
    }
    finally {
        sqlcon.Close();
    }
}
```

既定では TableAdapter ごとに異なる接続を利用してしまうため、これを揃えてからオープンする



4. ローカル型 DTC トランザクション ー回避策① コネクション変更による LCT の利用

- 実際の利用時には、以下の点にもご注意ください
  - System.Transactions の昇格の防止
    - 既定状態で利用していると、気づかずに DTC トランザクションに昇格するようなアプリケーションを開発してしまう恐れがあります
    - TransactionScope オブジェクトの昇格防止のためには、イベントハンドラを活用し、昇格時に例外を発生させるようにします（コード参照）
  - 実装コードの簡素化
    - using ブロック内で手動で接続を Open() / Close() する方法は大変です
    - ラッパークラスを作成すると、実装コードを簡素化できます

C#

```
// 以下のコードを global.asax の Application_Start() イベントハンドラなどに組み込む

TransactionManager.DistributedTransactionStarted += new TransactionStartedEventHandler(
    delegate(object sender, TransactionEventArgs e)
    {
        throw new ApplicationException("昇格（DTCトランザクションの利用）は認められていません。");
    }
);
```

4. ローカル型 DTC トランザクション ー回避策① コネクション変更による LCT の利用

- 実装コードの簡素化の方法（例）
  - TransactionScope オブジェクトと TableAdapter クラス群をラップしたクラスを作成して利用します
    - RequiredNew の TransactionScope を自動的に作成
    - TableAdapter クラス群の接続を自動的に揃えた上で、接続をオープン
  - これにより、実装コードを以下のように簡略化することができます

C#

```
string au_id = "172-32-1176";
string title_id = "PS2106";

AuthorsDataSetTableAdapters.authorsTableAdapter ata = new AuthorsDataSetTableAdapters.authorsTableAdapter();
TitlesDataSetTableAdapters.titlesTableAdapter tta = new TitlesDataSetTableAdapters.titlesTableAdapter();

using (TableAdapterTransactionScope scope = new TableAdapterTransactionScope(ata, tta))
{
    ata.SetNewContractByAuId(false, au_id);
    tta.SetNewPriceByTitleId(28, title_id);

    scope.Complete();
}
lblResult.Text = "データを更新しました。";
```

自動的に接続を揃えてオープンする

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

- 実装コードの簡素化の方法(例)(ラッパークラス)

C#

```
public class TableAdapterTransactionScope : IDisposable {
    // 昇格防止
    static TableAdapterTransactionScope() {
        TransactionManager.DistributedTransactionStarted += new TransactionStartedEventHandler(
            delegate(object sender, TransactionEventArgs e) {
                throw new ApplicationException("トランザクションの昇格（DTCトランザクションの利用）は認
められていません。");
            }
        );
    }
    private object[] _tableAdapters = null;
    private TransactionScope _scope = null;
    private IDbConnection _connection = null;
    public TableAdapterTransactionScope(params object[] tableAdapters) {
        // テーブルアダプタが利用するコネクションを統一する
        if (tableAdapters == null || tableAdapters.Length == 0) throw new
ArgumentNullException("tableAdapters");
        _tableAdapters = tableAdapters;
        SetSameConnectionToTableAdapters();
        // RequiredNew トランザクションを開始する
        _scope = new TransactionScope(TransactionScopeOption.RequiresNew);
        // コネクションを自力でオープンする
        _connection.Open();
    }
```

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

C#

```
    private void SetSameConnectionToTableAdapters() {
        foreach (object tableAdapter in _tableAdapters) {
            // Connection プロパティから接続を取り出す
            Type t = tableAdapter.GetType();
            PropertyInfo pi = t.GetProperty("Connection", BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance | BindingFlags.Static | BindingFlags.DeclaredOnly);
            MethodInfo mi1 = pi.GetGetMethod(true);
            IDbConnection dbc = (IDbConnection)mi1.Invoke(tableAdapter, null);
            if (_connection == null) {
                _connection = dbc;
            } else {
                // 既存の接続と接続文字列が同一か否かを確認
                if (_connection.ConnectionString != dbc.ConnectionString) {
                    throw new ApplicationException("テーブルアダプタ間の接続文字列が一致しません。同一デ
ータベースに対するテーブルアダプタを利用してください。");
                }
                // 一致している場合には、コネクションを揃える
                MethodInfo mi2 = pi.GetSetMethod(true);
                mi2.Invoke(tableAdapter, new object[] { _connection });
            }
        }
    }
```

TableAdapter 構成ウィザードは
既定では接続を Public としていないため
リフレクションで非公開プロパティを
操作する

4. ローカル型 DTC トランザクション
ー回避策① コネクション変更による LCT の利用

C#

```
    public void Complete() {
        _scope.Complete();
    }

    void IDisposable.Dispose() {
        // コネクションをクローズする
        if (_connection != null) {
            try {
                _connection.Close();
            }
            catch {
            }
        }

        if (_scope != null) {
            try {
                _scope.Dispose();
            }
            catch {
            }
        }
    }
}
```

4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- .NET Framework 1.1 の ServiceDomain 機能を利用すると、TransactionScope に似たクラスを作成できます
  - Windows Server 2003, Windows XP SP2 以降でのみサポート
  - この機能をラップしたクラスを作成すると、TransactionScope と同様なコードにより DTC トランザクションを作成することができます
  - TransactionScope クラスと異なり、昇格機能を持ちません（＝最初から DTC トランザクションとして処理されます）
- このクラスによる DTC トランザクションを利用すると、SPC 最適化により不整合発生の可能性なく DBM を利用できます
  - MS-DTC は、スコープ内に存在するリソースが単一である場合には、SPC 最適化（Single Phase Commit 最適化）を行います
  - これにより、DBM を利用できるようになります
  - 実装方法 → 次ページ参照
  - SPC 最適化、内部動作に関する解説 → 後述

4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- 実装方法
  - System.EnterpriseServices 名前空間下の ServiceConfig クラスをラップしたクラス（ServiceDomainTransactionScope クラス）を作成
    - トランザクションオプションを指定するようにもできますが、ここでは常に RequiredNew を利用するものとしました
  - 上述により作成したクラスを、System.Transactions 名前空間の TransactionScope クラスの代わりに利用するようコード修正します

C#

```
string title_id = "PS2106";
TitlesDataSetTableAdapters.titlesTableAdapter ta = new TitlesDataSetTableAdapters.titlesTableAdapter();
using (ServiceDomainTransactionScope scope = new ServiceDomainTransactionScope()) {
    decimal? price = (decimal?)ta.GetPriceByTitleIdWithUpdlock(title_id);
    if (price == null) {
        lblResult.Text = "データがありません。";
        return;
    }
    decimal newPrice = price.Value + 1;
    ta.SetNewPriceByTitleId(newPrice, title_id);
    scope.Complete();
}
```

Tx-1

System.Transactions.TransactionScope クラスではなく、
ServiceDomain をラップしたクラスによって
自動トランザクションのスコープを作成する

TransactionScope クラスの代わりに利用する ServiceDomain ラッパクラス

C#

```
using System;
using System.Runtime.InteropServices;
using System.EnterpriseServices;

// To Create the RequiresNew Transaction Scope by ServiceDomain
public class ServiceDomainTransactionScope : IDisposable
{
    public ServiceDomainTransactionScope()
    {
        if (OSVersionCheckUtil.CheckCanUseServiceDomain() == false)
        {
            throw new ApplicationException("You can't use this feature on this machine. Please use Windows XP SP2 or Windows
Server 2003 or later.");
        }

        ServiceConfig cfg = new ServiceConfig();
        cfg.Transaction = TransactionOption.RequiresNew;
        cfg.TrackingEnabled = true;
        cfg.TrackingAppName = System.Reflection.Assembly.GetExecutingAssembly().FullName;
        cfg.TrackingComponentName = this.GetType().FullName;
        ServiceDomain.Enter(cfg);
        ContextUtil.SetAbort();

        // DTC トランザクションを物理的に開始(このコードがないと Windows XP SP2 の場合は 2PC になってしまう)
        Guid txId = ContextUtil.TransactionId;
}

    public void Complete()
    {
        ContextUtil.SetComplete();
    }

    public void Dispose()
    {
        ServiceDomain.Leave();
    }
}
```

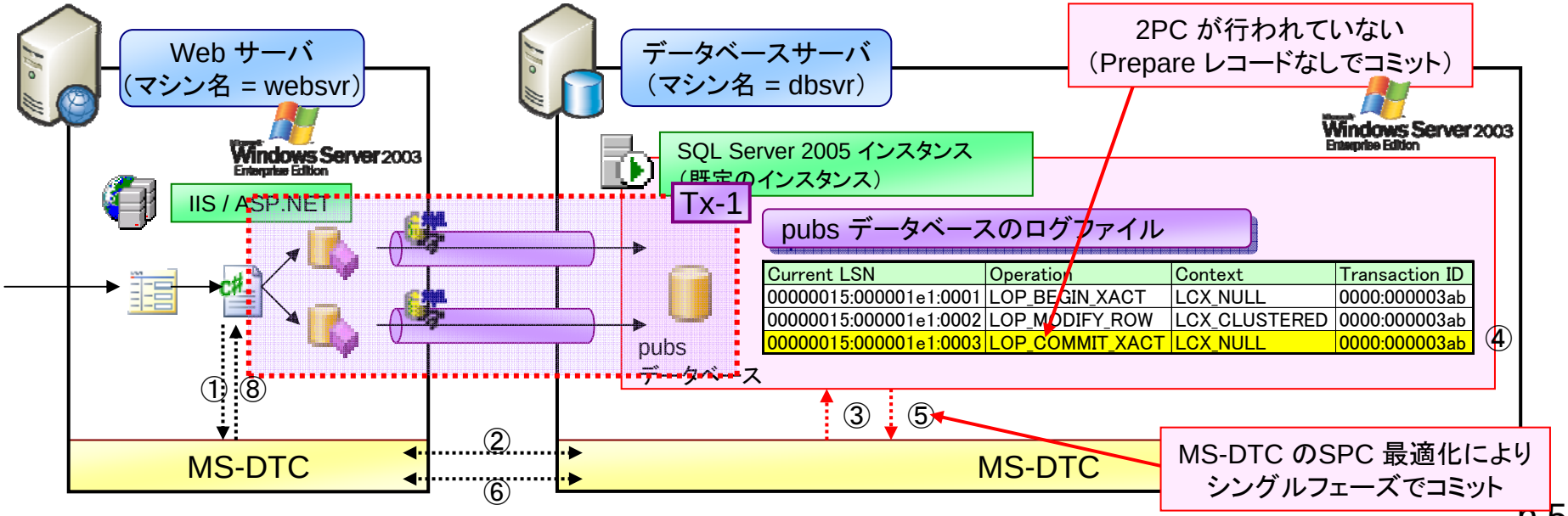
※ この機能は Windows XP SP2, Windows Server 2003 以降でのみ利用可能

C#

```
internal class OSVersionCheckUtil
{
    // How To Determine the Operating System Service Pack Level in Visual C# .NET
    // http://support.microsoft.com/default.aspx?scid=kb;EN-US;304721
    [StructLayout(LayoutKind.Sequential)]
    private struct OSVERSIONINFO
    {
        public int dwOSVersionInfoSize;
        public int dwMajorVersion;
        public int dwMinorVersion;
        public int dwBuildNumber;
        public int dwPlatformId;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
        public string szCSDVersion;
    }

    [DllImport("kernel32.Dll")]
    private static extern short GetVersionEx(ref OSVERSIONINFO o);
    private static bool _checked = false;
    private static bool _result = false;
    internal static bool CheckCanUseServiceDomain()
    {
        if (_checked) return _result;
        OSVERSIONINFO os = new OSVERSIONINFO();
        os.dwOSVersionInfoSize = Marshal.SizeOf(typeof(OSVERSIONINFO));
        GetVersionEx(ref os);

        // Windows XP SP2, Windows Server 2003, or later.
        Version v = new Version(os.dwPlatformId, os.dwMajorVersion, os.dwMinorVersion, os.dwBuildNumber);
        bool ret = (v >= new Version(2, 5, 1, 2600));

        // Result Cache
        _checked = true;
        _result = ret;

        return ret;
    }
}
```

4. ローカル型 DTC トランザクション
ー回避策② ServiceDomain による SPC 最適化

- （昇格した）TransactionScope クラスも、ServiceDomain も、内部的には DTC トランザクションになります
  - どちらもトランザクションスコープのコミット時には、MS-DTC との連係動作が行われます
  - しかし ServiceDomain を利用した場合、SPC 最適化が行われます



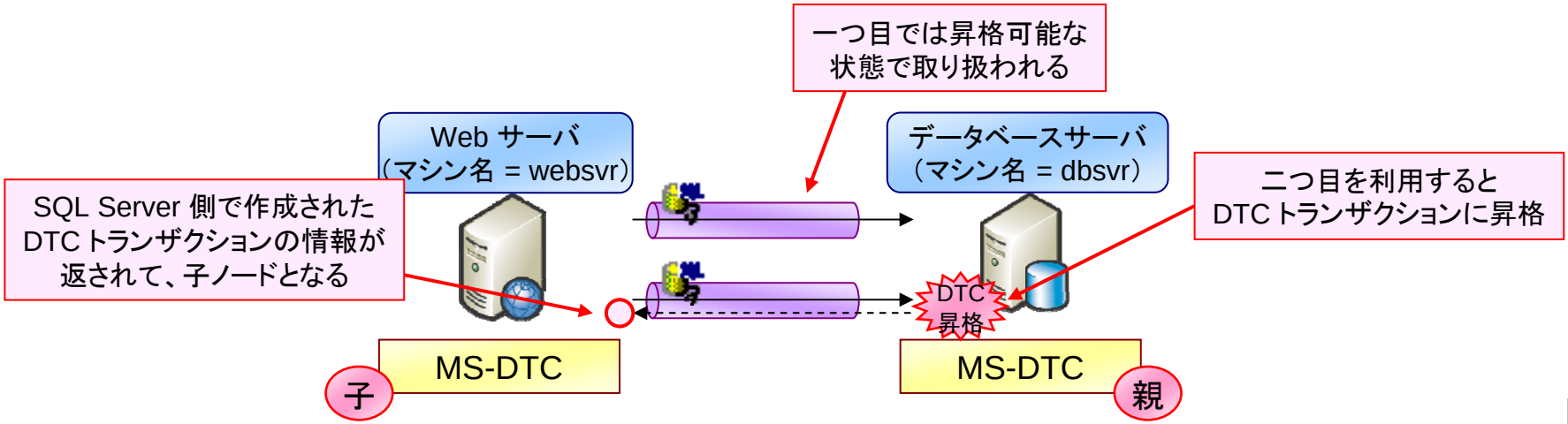
4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- （参考）なぜこの方法により SPC 最適化が行われるのかは、以下のポイントを理解する必要があります
  - A. TransactionScope クラスによる昇格時の MS-DTC の挙動
  - B. ServiceDomain 利用時の MS-DTC の挙動

- 以下に順を追って解説します

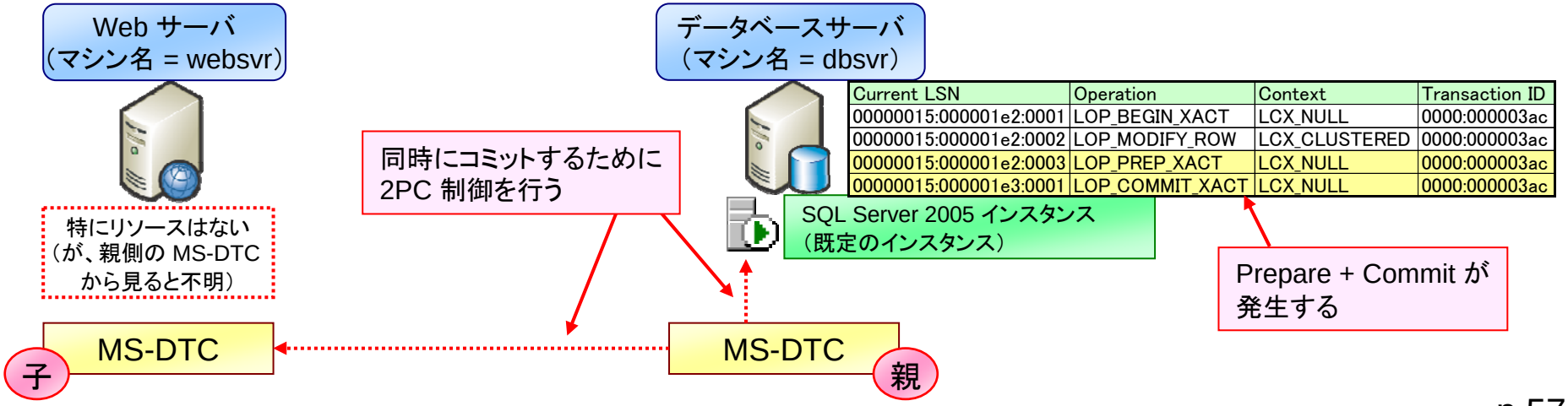
4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- A. TransactionScope クラスによる昇格時の MS-DTC 挙動
  - TransactionScope 内で 2 回以上 SqlConnection をオープンすると、SQL Server 2005 側で昇格が発生します
    - System.Transactions と System.Data がサポートする PSPE（Promotable Single Phase Enlistment）機能により実現されています
    - PSPE の仕様により、昇格時には SQL Server 2005 ノード側が分散トランザクションの親ノードに、Web サーバ側が子ノードになります
    - この場合には、必ず 2PC が行われます（→ 次ページ）



4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- A. TransactionScope クラスによる昇格時の MS-DTC 挙動
  - SQL Server 2005 側の MS-DTC が親ノードとなった場合、DTC トランザクションコミット時には、必ず 2PC が行われます
    - 親ノードの MS-DTC は、SQL Server 2005 インスタンスと、Web サーバの MS-DTC が管理するリソースを同時にコミットする必要があります
    - このため、実際に DTC トランザクションに関与するリソースは単一であるにもかかわらず、MS-DTC は 2PC 制御を行います
    - この結果、DBM の利用には注意が必要になります



Current LSN	Operation	Context	Transaction ID
00000015:000001e2:0001	LOP_BEGIN_XACT	LCX_NULL	0000:000003ac
00000015:000001e2:0002	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:000003ac
00000015:000001e2:0003	LOP_PREP_XACT	LCX_NULL	0000:000003ac
00000015:000001e3:0001	LOP_COMMIT_XACT	LCX_NULL	0000:000003ac

Web サーバ側が子ノード扱い
DB サーバ側(親ノード)からの指示により
2PC 要求を受ける

[TransactionScope Web 側 DTC ログ (nakama61, Windows Server 2003 R2)]

pid	tid	time	seq	eventid	tx_guid	message
1644	3276	06/06/2006-14:42:11.753	13	TRANSACTION_PROPOGATED_FROM_PARENT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	transaction propagated from parent node 'NAKAMA78', Description = ''
1644	3276	06/06/2006-14:42:11.803	14	RECEIVED_PREPARE_FROM_PARENT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	child node received prepare request from parent node 'NAKAMA78'
1644	3276	06/06/2006-14:42:11.803	15	VOTING_COMMIT_TO_PARENT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	child node votes commit to parent node 'NAKAMA78'
1644	3276	06/06/2006-14:42:11.813	16	RECEIVED_COMMIT_FROM_PARENT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	child node received commit request from parent node 'NAKAMA78'
1644	3276	06/06/2006-14:42:11.813	17	ACKNOWLEDGING_COMMIT_TO_PARENT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	child node achnowledging the delivery of commit request from parent node 'NAKAMA78'

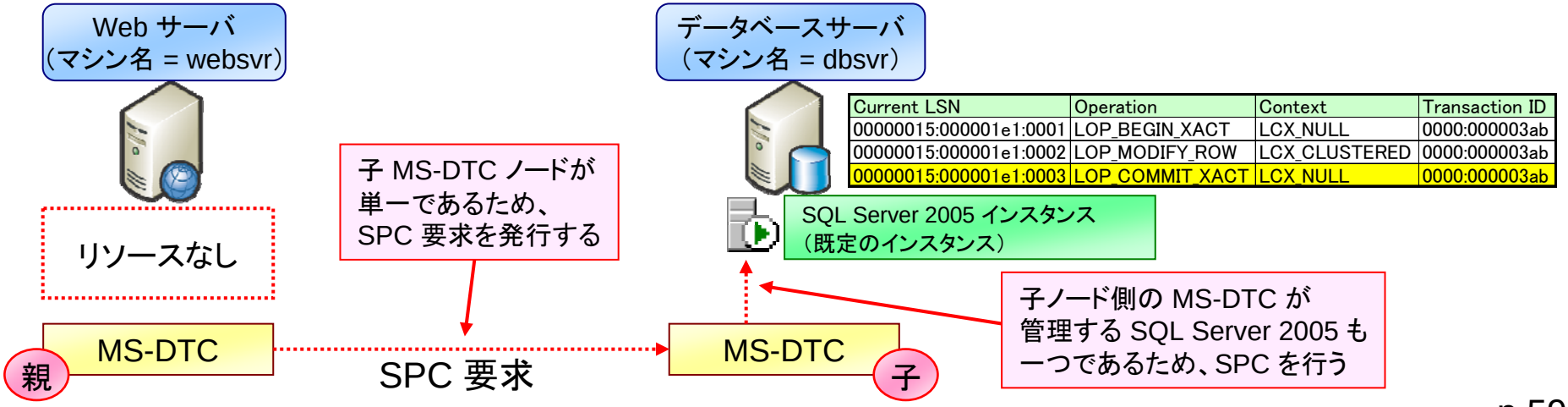
DB サーバ側が親ノードとなって 2PC を
行い、トランザクションをコミットする

[TransactionScope DB 側 DTC ログ (nakama78, Windows Server 2003 R2)]

pid	tid	time	seq	eventid	tx_guid	message
1064	1908	06/06/2006-14:42:19.926	146	TRANSACTION_BEGUN	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	transaction got begun, description : 'user_transaction'
1064	1992	06/06/2006-14:42:19.926	147	RM_ENLISTED_IN_TRANSACTION	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	resource manager #1002 enlisted as transaction enlistment #1. RM guid = 'd63da867-4286-49b1-b25e-3e38aa70f985'
1064	3432	06/06/2006-14:42:20.067	148	TRANSACTION_PROPOGATED_TO_CHILD_NODE	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	transaction propagated to 'NAKAMA61' as transaction child node #1
1064	1124	06/06/2006-14:42:21.108	149	RECEIVED_COMMIT_REQUEST_FROM_BEGINNER	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	received request to commit the transaction from beginner
1064	1124	06/06/2006-14:42:21.108	150	RM_ISSUED_PREPARE	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	prepare request issued to resource manager #1002 for transaction enlistment #1
1064	1124	06/06/2006-14:42:21.108	151	CHILD_NODE_ISSUED_PREPARE	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	prepare request issued to transaction child node #1 'NAKAMA61'
1064	1908	06/06/2006-14:42:21.108	152	RM_VOTED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	resource manager #1002 voted commit for transaction enlistment #1
1064	3432	06/06/2006-14:42:21.108	153	CHILD_NODE_VOTED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	transaction child node #1 'NAKAMA61' voted commit
1064	2548	06/06/2006-14:42:21.118	154	TRANSACTION_COMMITTED	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	transaction has got committed
1064	2548	06/06/2006-14:42:21.118	155	RM_ISSUED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	commit request issued to resource manager #1002 for transaction enlistment #1
1064	2548	06/06/2006-14:42:21.118	156	CHILD_NODE_ISSUED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	commit request issued to transaction child node #1 'NAKAMA61'
1064	2548	06/06/2006-14:42:21.118	157	RM_ACKNOWLEDGED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	received acknowledgement of commit request from the resource manager #1002 for transaction enlistment #1
1064	3432	06/06/2006-14:42:21.118	158	CHILD_NODE_ACKNOWLEDGED_COMMIT	2b71f8b7-a24a-48dd-abcb-6e8c558e6323	received acknowledgement of commit request from transaction child node #1 'NAKAMA61'

4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- B. ServiceDomain 利用時の MS-DTC の挙動
  - ServiceDomain を利用すると、Web サーバ側で MS-DTC の物理トランザクションを開始することができます
  - このようにすると、DTC トランザクションコミット時にリソースが単一であることを適切に認識できるため、SPC 最適化が可能になります
  - MS-DTC が 2PC を行わなければ、AP/DB 間でコミット認識のズレが発生することはなくなり、DBM が特に問題なく適用できます



4. ローカル型 DTC トランザクション ー回避策② ServiceDomain による SPC 最適化

- Web サーバで DTC トランザクションを開始した際の MS-DTC ログ
  - 下記の MS-DTC ログから分かるように、SPC 最適化が行われる場合には、2PC は発生していません

※ 2PC が発生していないことは、SQL Server 2005 側のログからも確認できます

アプリからコミット要求を受け取る

[ServiceDomainScope Web 側 DTC ログ (nakama61, Windows Server 2003 R2)]

pid	tid	time	seq	eventid	tx_guid	message
1644	1696	06/06/2006-14:31:23.168	6	TRANSACTION_BEGUN	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	transaction got begun, description : 'MCS.Japan.Utilities.Transaction.Service'
1644	3276	06/06/2006-14:31:25.371	7	TRANSACTION_PROPOGATED_TO_CHILD_NODE	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	transaction propagated to 'NAKAMA78' as transaction child node #1
1644	1696	06/06/2006-14:31:25.822	8	RECEIVED_COMMIT_REQUEST_FROM_BEGINNER	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	received request to commit the transaction from beginner
1644	3276	06/06/2006-14:31:25.822	9	TRANSACTION_COMMITTED	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	transaction has got committed

子ノードに SPC 発行

親ノードから SPC フラグつきで Prepare 要求（＝コミット要求）を受け取る

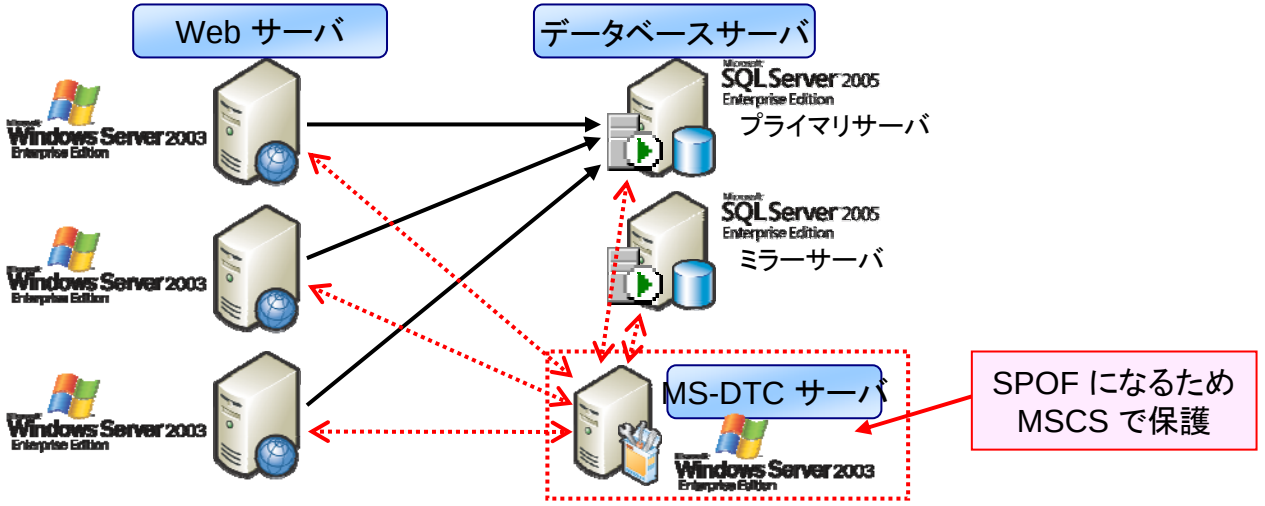
[ServiceDomainScope DB 側 DTC ログ (nakama78, Windows Server 2003 R2)]

pid	tid	time	seq	eventid	tx_guid	message
1064	3432	06/06/2006-14:31:37.080	139	TRANSACTION_PROPOGATED_FROM_PARENT	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	transaction propagated from parent node 'NAKAMA61', Description = ''
1064	1908	06/06/2006-14:31:38.082	140	RM_ENLISTED_IN_TRANSACTION	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	resource manager #1002 enlisted as transaction enlistment #1. RM guid = 'd63da367-4286-49b1-b25e-3e38aa70f985'
1064	3432	06/06/2006-14:31:38.482	141	RECEIVED_PREPARE_FROM_PARENT	b5cc3b53-4bcf-4683-8bd7-2f6e10bffbe0	child node received prepare request from parent node 'NAKAMA61'

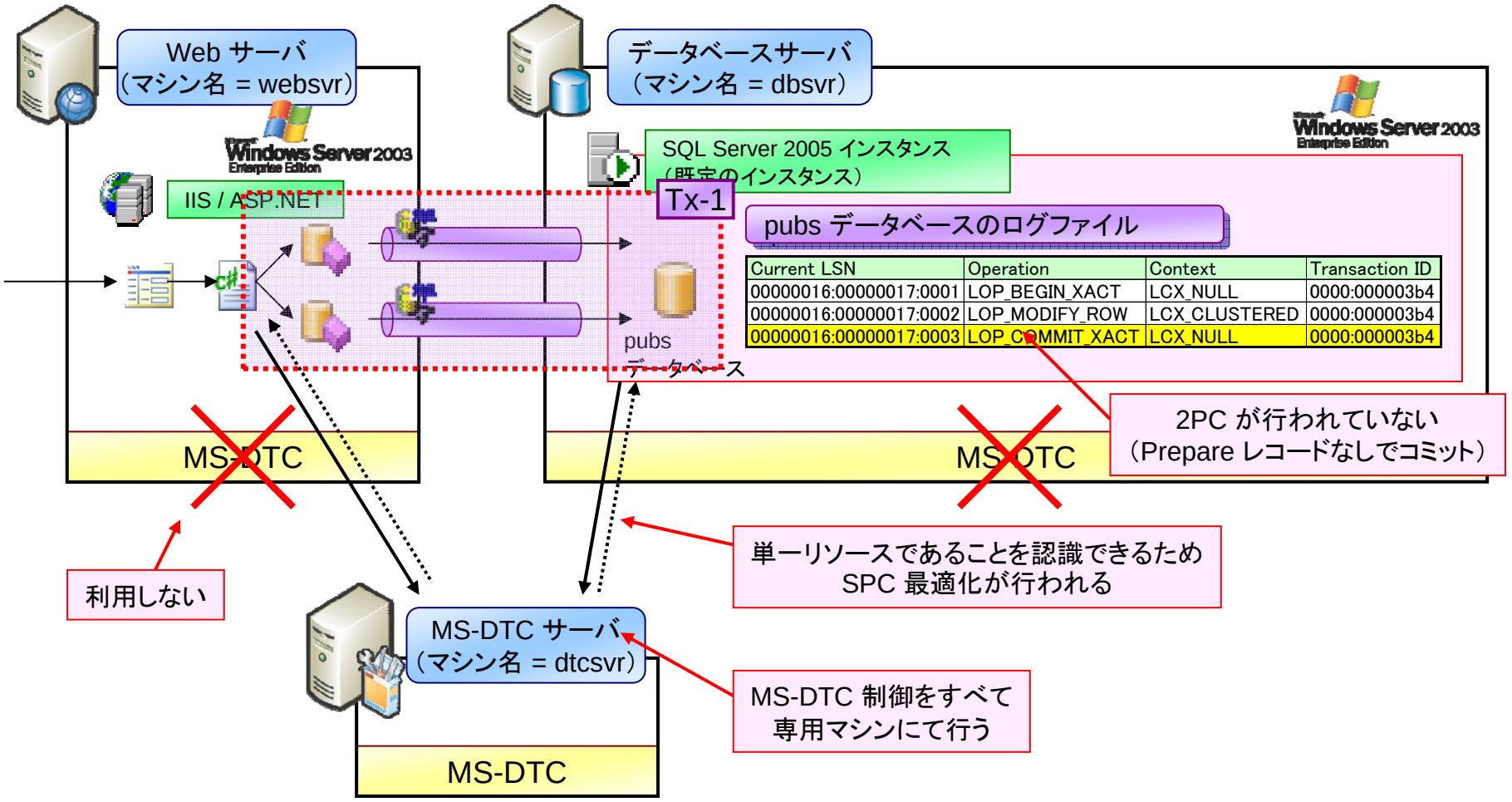
※ コミットレコードはログに記録されず

4. ローカル型 DTC トランザクション
ー回避策③ セントラル MS-DTC サーバ SPC 最適化

- アプリケーションコードに一切手を加えられない場合は、MS-DTC 専用サーバを用意して利用する方法があります
  - 全 Web、DB サーバが単一 MS-DTC サーバを利用するように構成
    - SPOF になるため、MSCS で保護することを推奨します
    - （保護対象リソースは MS-DTC のみ、フェイルオーバは比較的高速）
  - この構成の場合には SPC 最適化が可能なため、AP/DB 間での不整合の発生の可能性なく DBM を利用することができます



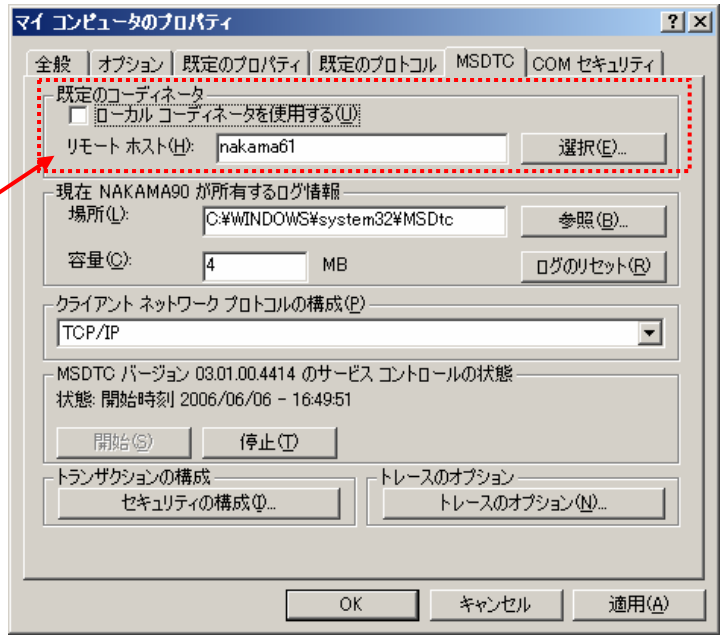
4. ローカル型 DTC トランザクション
ー回避策③ セントラル MS-DTC サーバ SPC 最適化



4. ローカル型 DTC トランザクション
ー回避策③ セントラル MS-DTC サーバ SPC 最適化

- リモート MS-DTC の指定方法について
  - コンポーネントサービス画面の MS-DTC プロパティを設定します
    - ローカルコーディネータではなく、リモートマシンを指定します
  - 設定を変更したら、必ずコンピュータを再起動してください

※ どの MS-DTC を利用するかに関しては、MS-DTC を利用するプロセスにキャッシュされるため



リモート MS-DTC の利用

4. ローカル型 DTC トランザクション ー回避策③ セントラル MS-DTC サーバ SPC 最適化

- リモート MS-DTC を利用する構成にすると、すべての作業がリモート MS-DTC 側に委譲されます
  - Web サーバや DB サーバ上の MS-DTC は利用されなくなります

[TransactionScope Web 側 DTC ログ (nakama90, Windows XP)]

pid	tid	time	seq	eventid	tx_guid	message
ログなし						

[TransactionScope DB 側 DTC ログ (nakama78, Windows Server 2003 R2)]

pid	tid	time	seq	eventid	tx_guid	message
ログなし						

[TransactionScope DTC 側 DTC ログ (nakama61, Windows Server 2003 R2)]

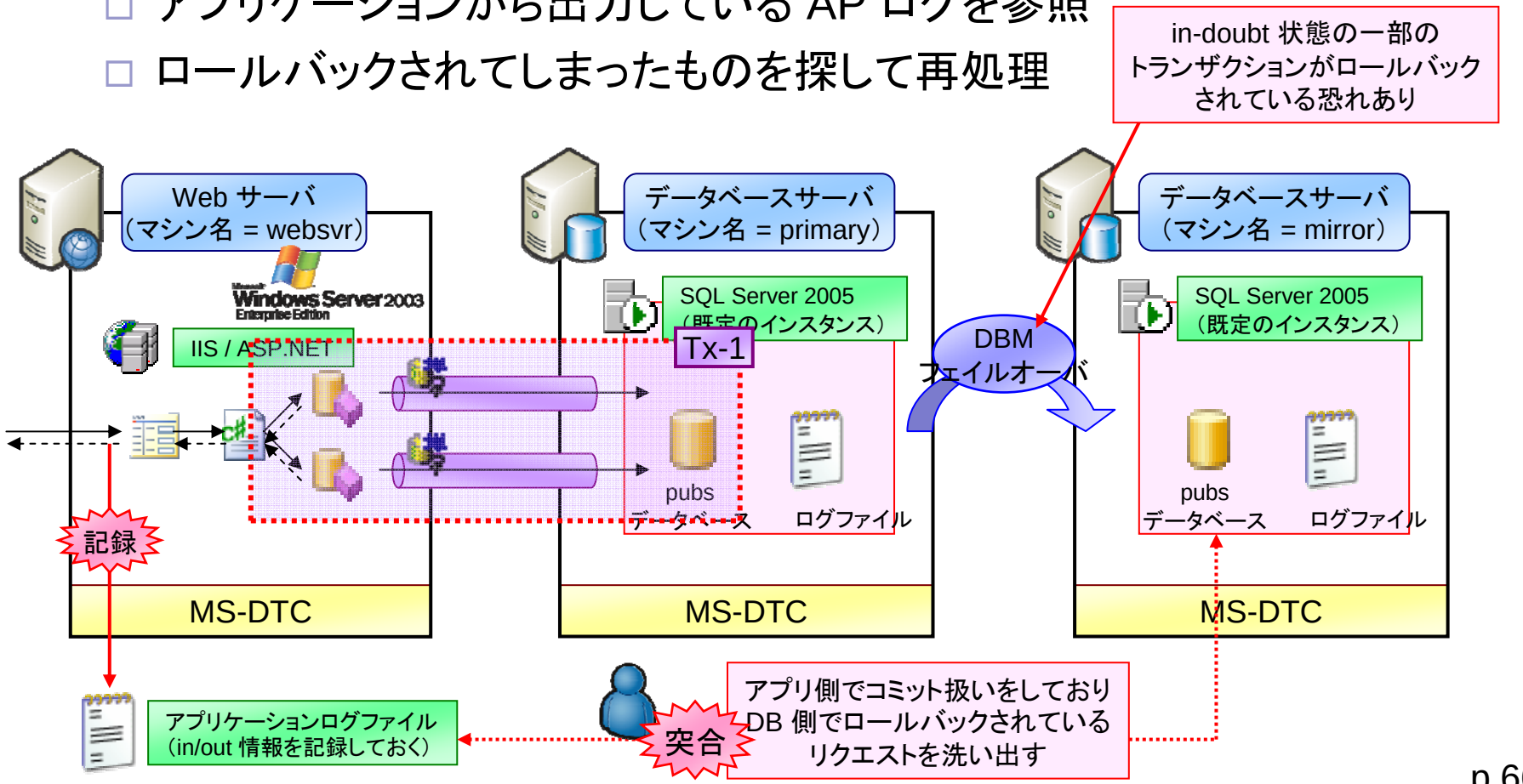
pid	tid	time	seq	eventid	tx_guid	message
1644	3612	06/06/2006-16:55:34.788	46	TRANSACTION_BEGUN	3038d540-6e82-4252-8fb6-e992298c7625	transaction got begun, description : 'user_transaction'
1644	3612	06/06/2006-16:55:34.798	47	RM_ENLISTED_IN_TRANSACTION	3038d540-6e82-4252-8fb6-e992298c7625	resource manager #1003 enlisted as transaction enlistment #1. RM guid = 'd63da867-4286-49b1-b25e-3e38aa70f985'
1644	3612	06/06/2006-16:55:42.680	48	RECEIVED_COMMIT_REQUEST_FROM_BEGINNER	3038d540-6e82-4252-8fb6-e992298c7625	received request to commit the transaction from beginner
1644	3612	06/06/2006-16:55:42.780	49	TRANSACTION_COMMITTED	3038d540-6e82-4252-8fb6-e992298c7625	transaction has got committed

4. ローカル型 DTC トランザクション ー(非回避策)④ アプリケーションログによる運用対処

- ①～③の方法は AP/DB 間での不整合発生を完全に防止するものですが、運用で対処する方法も考えられます
  - クロスデータベーストランザクションの場合と異なり、ローカル型 DTC トランザクションではデータ自体の破損(不整合)は発生しません
    - 発生する問題は、AP 側ではコミットと意識しているにもかかわらず、DB 上ではロールバックされているトランザクションが存在しうることです
    - しかしこれは、DBM のフェイルオーバ時に非同期コミット処理中であった in-doubt トランザクションでしか発生しません(＝比較的稀です)
  - このため、DBM フェイルオーバ時にアプリケーションログと DB 上のデータを突合せて確認する(運用対処)という方法も考えられます
    - アプリケーション側では、処理の in/out 時にアプリケーションログを出力していることが一般的です
    - DBM フェイルオーバ発生時に AP ログと DB 上のデータを突合し、誤ってロールバックされたトランザクションを再投入する方法も考えられます

4. ローカル型 DTC トランザクション
ー（非回避策）④ アプリケーションログによる運用対処

- DBM フェイルオーバ時の運用対処の内容について
  - アプリケーションから出力している AP ログを参照
  - ロールバックされてしまったものを探して再処理



4. ローカル型 DTC トランザクション ー(非回避策)④ アプリケーションログによる運用対処

- この④の対応方法も以下の理由から現実的と考えられます
  - 主な理由は以下の通りです
    - ローカル型 DTC トランザクションでは更新処理の原子性は保たれます
      - クロスデータベーストランザクションと異なり、データ不整合は発生しません
      - ロールバックされたトランザクションの再投入で対処可能なこともあります
    - DBM フェイルオーバは頻繁に発生するものではありません
      - AP／DB 間での不整合発生は望ましくありませんが、非常に稀なケースでしか発生しません
      - レアケースの対策に大量のコストをかけるのは現実的ではありません
    - Web サーバの AP ログと DB データの突合せはよく行われます
      - このようなケースに限らず、Web サーバが障害を起こした場合には、AP ログと DB データの突合せが必要になることはよくあります
  - この方法の利用可否は、生じる問題の大きさによって違ってきます
    - 例）株式取引システム → 注文データのロストはあってはならない事態
    - 例）イントラネットの情報系小規模アプリ → 再投入でも対応可能

4. ローカル型 DTC トランザクション
ー3 通りの回避策の比較とまとめ

- 前述の 3 通りの回避策は、それぞれにトレードオフがある
  - 主に、コード修正量と性能との間にトレードオフがあります
    - コードが修正できる場合には、LCT を利用するように修正する方法をお奨めします
    - 3rd party 製品など、コードが修正できない場合にはセントラル MS-DTC サーバの採用を検討してください
  - 新規開発アプリケーションでは設計時点から DBM への対応方法を検討してください
    - DBM はアプリケーション開発に対する注意事項があります
    - このため開発初期から DBM 適用を考慮した開発を行ってください

回避策	内部処理	コード修正量	性能
① コネクション変更による LCT の利用	LCT	中	◎
② ServiceDomain の利用による SPC 最適化	SPC 最適化された DTC Tx	小	○
③ セントラル MS-DTC サーバによる SPC 最適化	リモート通信、SPC 最適化された DTC Tx	なし	△

その他

- 技術的なキーポイントのまとめ
  - DBM（データベースミラーリング）
    - クロスデータベーストランザクションを利用した場合は DBM フェイルオーバ時にデータベース間でのデータ不整合が発生する場合があります
      - DBM フェイルオーバでは in-doubt 状態の処理を一律でロールバックします
      - このため、トランザクション処理の原子性が失われる場合があります
    - 単一データベースに閉じたトランザクション処理をしている限りは、データベース内でのデータ不整合が発生することはありません
  - MS-DTC
    - MS-DTC が 2PC を行った場合、DBM フェイルオーバ時にアプリケーションとデータベース間でコミット認識のズレが発生することがあります
      - MS-DTC は Commit Phase において、非同期処理を行います
      - このため、アプリケーション側では MS-DTC からコミット応答を受け取ったにもかかわらず、データベース側ではロールバックされている場合があります
    - MS-DTC が SPC 最適化を行った場合は、認識のズレは発生しません

その他

- System.Transactions（TransactionScope）
  - 同一スコープ内で SQL Server 2005 に対して 1 つの接続を開き続けて利用する場合には、LCT トランザクションとなります
    - マニュアルトランザクションと同様に処理され、MS-DTC は動作しません
  - 同一スコープ内で 2 回以上接続を開くと、DTC トランザクションに昇格します
    - 同一 Connection を Open() → Close() → Open() としても昇格します
    - 昇格時には、DB サーバ側の MS-DTC が親ノードとなります
- TableAdapter オブジェクト
  - 既定では、各メソッドは接続のオープン／クローズを自動で行います
    - 事前にオープンしていた場合には、接続の自動オープン／クローズはしない
  - Connection の Modifier を public にすると、外部から Connection オブジェクトを操作できるようになる

- 上記の挙動の組み合わせにより、DBM 利用時の注意要否が変わってきます

その他

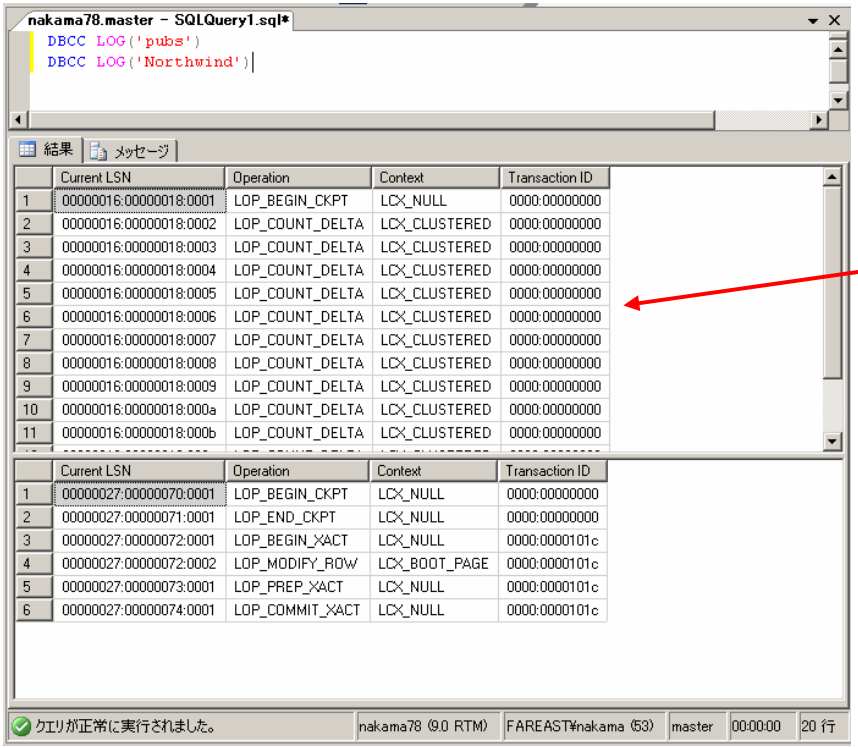
- 今後登場する予定の新技術への影響について
  - System.Transactions によるトランザクション制御は、.NET Framework 2.0 以降での標準的なトランザクション制御モデルです
    - .NET Framework 3.0 などのほとんどの技術は、この System.Transactions によってトランザクション制御される予定です
      - LINQ（Language Integrated Query）
      - WF（Windows Workflow Foundation）
      - WCF（Windows Communication Foundation）など
  - DBM を利用する場合、今後提供予定のテクノロジを利用した際に制限が発生する可能性があることにご注意ください
  - このような条件下で高可用性が必要なシステムを開発する場合には、マイクロソフトクラスタサービス（MSCS）の適用をご検討ください

その他

- DBM 利用時に注意が必要か否かを確認する簡単な方法
  - SQL Server 2005 のデータベースログファイル上の Prepared レコードの有無を確認すると注意が必要か否かを容易に判定できます
    - 以下の 2 つのいずれの場合も、SQL Server 2005 データベースログファイル上には Prepared レコードが立ちます
      - クロスデータベーストランザクションの場合 → 利用時に注意が必要
      - 2PC を行う DTC トランザクションの場合 → 利用時に注意が必要
  - ただし、SQL Server 2005 データベースエンジンの最適化処理により、Prepared レコードが立たない場合もあるためご注意ください
    - 例）UPDATE クエリで更新したデータが既存のデータと同一だった場合には、物理的なデータ更新を行わないような最適化が行われます
    - このため、ログファイルを安易に確認して判断することは危険であるため、十分に注意を払って確認を行うようにしてください

その他

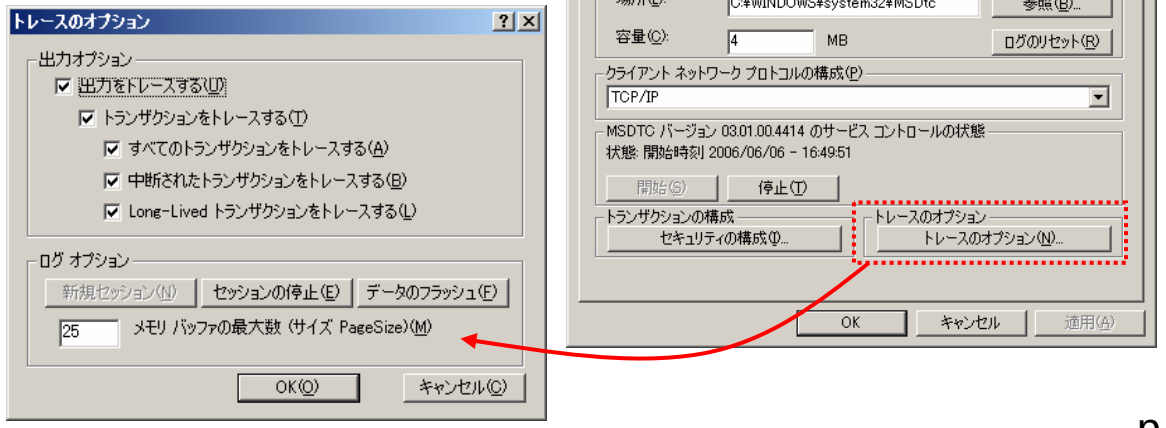
- SQL Server 2005 のログファイルの確認方法
  - SQL Server Management Studio を利用します
  - DBCC LOG('databasename') コマンドにより確認します



ログファイル上のレコードが
見やすい形で表示される

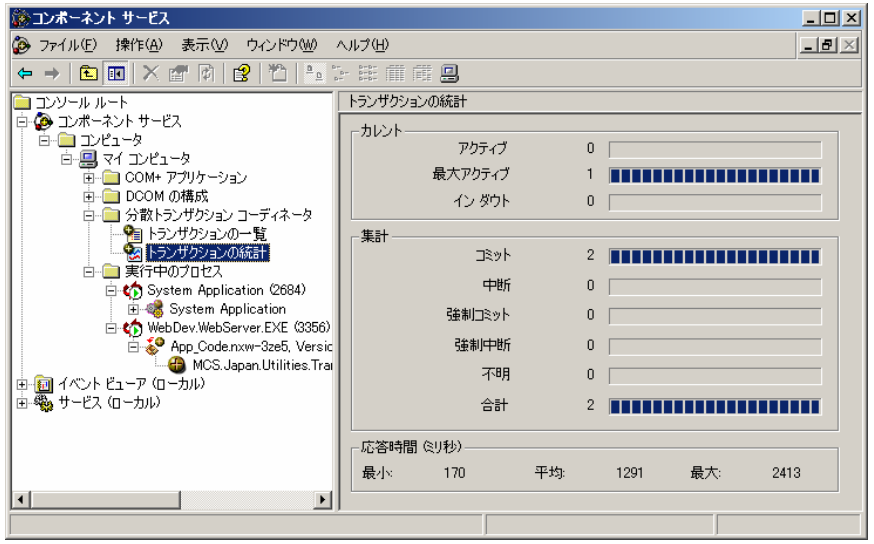
その他

- MS-DTC のトレースログの確認方法
  - 以下の手順により、トレースログを解析することができます
    - 1. トレースログ出力ツールの入手
      - TraceFmt.exe ツールを入手（Windows XP, Windows Server 2003 サポートツールに含まれます）
    - 2. MS-DTC トレース出力を有効化
      - コンポーネントサービスのプロパティで設定
      - 既定では C:¥WINDOWS¥system32¥MsDtc¥Trace に出力



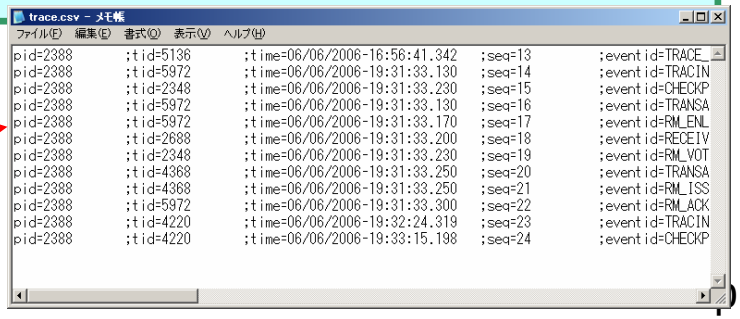
その他

- 3. アプリケーションを動作させます
  - コンポーネントサービス画面を利用して、MS-DTC の動作をモニタリングしながらアプリケーションを動作させると分かりやすいです
  - C:¥WINDOWS¥system32¥MsDtc¥Trace にログが出力されます
- 4. トレースログをフラッシュし、csv 形式に変換
  - コマンドラインからログをフラッシュします
  - 入手したツールと用意されているバッチファイルにより csv に変換できます



コマンドライン

```
C:¥WINDOWS¥system32¥MsDtc¥Trace>logman update
MSDTC_TRACE_SESSION -fd -ets
コマンドは、正しく完了しました。

C:¥WINDOWS¥system32¥MsDtc¥Trace>msdtcvtr -tracelog
C:¥WINDOWS¥system32¥MsDtc¥Trace¥dtctrace.log
```



その他

- OS による MS-DTC の内部動作の違いについて
  - COM+ ランタイムの IContextTransacitonInfo の実装有無や内部動作により、DTC トランザクションのスコープの挙動が変化します
  - このため、検証テスト時には少なくとも以下の OS での動作検証が必要になります
    - Windows XP SP2
    - Windows 2000
    - Windows 2003 SP1
    - Windows 2003 R2
    - Windows Server 2003 Post-Service Pack 1 COM+ 1.5 HotFix Rollup Package 8（http://support.microsoft.com/kb/912818/）

- 本書に記載した情報は、本書各項目に関する発行日現在の Microsoft の見解を表明するものです。Microsoftは絶えず変化する市場に対応しなければならないため、ここに記載した情報に対していかなる責務を負うものではなく、提示された情報の信憑性については保証できません。
- 本書は情報提供のみを目的としています。 Microsoft は、明示的または暗示的を問わず、本書にいかなる保証も与えるものではありません。
- すべての当該著作権法を遵守することはお客様の責務です。Microsoftの書面による明確な許可なく、本書の如何なる部分についても、転載や検索システムへの格納または挿入を行うことは、どのような形式または手段（電子的、機械的、複写、レコーディング、その他）、および目的であっても禁じられています。これらは著作権保護された権利を制限するものではありません。
- Microsoftは、本書の内容を保護する特許、特許出願書、商標、著作権、またはその他の知的財産権を保有する場合があります。Microsoftから書面によるライセンス契約が明確に供給される場合を除いて、本書の提供はこれらの特許、商標、著作権、またはその他の知的財産へのライセンスを与えるものではありません。
-
- Microsoft, Windows は、Microsoft Corporation の米国およびその他の国における登録商標または商標です。
- その他、記載されている会社名および製品名は、一般に各社の商標です。