Technical Note

Dealing With Concurrent Modifications Using REP++*toolkit* for .NET

Author: R&D Department

Publication date: August 28, 2006

Revision date: May 2010

Dealing With Concurrent Modifications Using REP++toolkit for .NET

Overview

In multi-user systems, one frequent problem concerns the concurrent modification of data. When more than one user modifies the same data in a database, integrity problems can arise. In some cases, the problem originates at the operational procedures level. In others, the information system is the culprit. There are several strategies used to prevent this issue.

This article describes how to implement a timestamp strategy to solve the concurrent modification problem using the **DataChangedDetector** component of REP++*toolkit* for .NET. This component automatically detects if the data read by one user was modified by another since it was first read from the database.

Concurrent modification challenge

Integrity problems due to concurrent modification of data occur in some cases at the level of the information system itself, but in most cases at the level of operational procedures. For instance, two users might receive a different address modification request for the same employee: whether the modifications are done simultaneously or within minutes does not change the fact that the user that saves the record last overwrites all previous modifications of the employee's record. Those issues of poorly designed workflow must be resolved at the operational level. The solution at the system level is often to do nothing but to ensure the integrity of the database.

Here is an example of concurrent modification at the system level: a user receives a shipment of 5 items of a product. The user selects the product in the system, and notes that there are currently 10 items in stock. The user adds the 5 new items and saves the new total of 15 items. Meanwhile, a second user receives a shipment of 2 items for the same product and requests the same record before the first user saves the transaction in the database. The second user also sees 10 items in stock, adds the 2 items and saves the record. Depending on who saves the modifications last, the inventory might show 12 or 15 items instead of the 17 items that really are available. This problem would not have occurred if the second user had read the record after the first user had saved the modifications.

Strategies for preventing problems derived from concurrent modification

There are several strategies used to solve this concurrency problem.

1. Prevent the direct editing of data: modify instead the information with an adjustment of the current value applied through an update. In the example above, the adjustment to the in-stock quantity could be done using a command of type **Update prod set qty=qty+delta**.

2. Lock, in the database, the data requested by a user, from the moment it is read until it is saved.
3. Detect, before saving the modifications, that the data read and modified by one user was modified by another since it was first read.

The first solution is common sense, from a database design point of view, and is generally used for the kind of problem described here. By using an approach where the current data is revised instead of being directly modified by a user, it is possible to know what modification was performed and why; the database's intrinsic mechanism of transaction processing is sufficient to eliminate the problematic occurrences.

The second solution causes severe performance degradation by locking the database for a relatively long period while the user processes the data. It is therefore only used as a last resort.

The third solution requires a timestamp of the last modification performed on the record. When a user requests, modifies and then saves a record, the timestamp of the local record is compared to that of the database record: if they differ, it indicates that the record was modified. The transaction is then cancelled and the user notified. The timestamp is interesting when the first solution cannot be applied or implemented easily. The drawback is that it acts *a posteriori*: if the transaction fails, the user must read the record again and reapply the modifications. However, it presents no significant performance degradation since it locks the record only long enough to compare the timestamps and then go ahead with the transaction or cancel it.

Preventing concurrent modifications with REP++toolkit

REP++*toolkit* can help you prevent concurrent modification issues using the timestamp approach. The **DataChangedDetector** component, included with **RepPP.Toolkit**, implements this solution. To use it, you simply need to create an instance of the component and attach it to a **RowsetTree**. One of its nodes must contain a field whose attributes **Automatic Date Stamping at Insertion** and **Automatic Date Stamping at Update** are set. At execution, the **DataChangedDetector** component will intercept the events related to the **RowsetTree**, i.e. to the transaction. If a user modifies the data and tries to save it, the component checks if the date of the field with the automatic date stamping has changed since it was read from the database. If so, the component triggers the **DataChanged** event and cancels automatically the transaction. You can then send a message notifying the user that the modifications could not be saved. If however the dates are identical, the transaction is completed.

The code below shows how to use the **DataChangedDetector** component with a program generated with the assistant. The bold typeface indicates the code related to the **DataChangedDetector** component.

```
private RepPP.Toolkit.DataChangedDetector m_dataChangedDetector;

public override void AttachChildEditor(Control ctlFather) {
    base.AttachChildEditor(ctlFather);
    m_rowsetTreeCLIENT = RowsetTree as TestSyncAccess.RTClient;
    m_nodeHandlerCLIENT = Handler;
    toolbarCLIENT.NodeHandler = m_nodeHandlerCLIENT;
    m_nodeHandlerADDRESS = new RepPP.Framework.Window.nodeHandler(m_nodeHandlerCLIENT,
                                RowsetTreeDefNode.RowsetTreeDef.FindNode("ADDRESS"),
                                true);
    toolbarADDRESS.NodeHandler = m_nodeHandlerADDRESS;
    if (m_dataChangedDetector == null) {
        // Creates the component. The m_nodeHandlerClient.RowsetTreeDefNode specifies
        // the RowsetTreeDefNode containing the automatic datetime field to check.
        m_dataChangedDetector =
         new RepPP.Toolkit.DataChangedDetector(m_nodeHandlerCLIENT.RowsetTreeDefNode,
                                               null);
        m_dataChangedDetector.DataChanged +=
         new RepPP.DbBeforeUpdRowsetTreeEventHandler(m_dataChangedDetector_DataChanged);
```

```
    }
    m_dataChangedDetector.RowsetTree = m_rowsetTreeCLIENT;
}


protected override void Dispose(bool bDisposing) {
    if (bDisposing) {
        if (components != null) {
            components.Dispose();
        }
        if (m nodeHandlerADDRESS != null) {
            m_nodeHandlerADDRESS.Dispose();
            m_nodeHandlerADDRESS = null;
        }
        if (m_dataChangedDetector != null) {
            m_dataChangedDetector.DataChanged -= new
              RepPP.DbBeforeUpdRowsetTreeEventHandler(m_dataChangedDetector_DataChanged);
            m_dataChangedDetector.Dispose();
        }
    }
    base.Dispose(bDisposing);
}

private void m_dataChangedDetector_DataChanged(object sender, RepPP.RowsetTreeEventArgs
e) {
    MessageBox.Show("The data has been changed by another user. Cannot save the client");
}
```