RPC Reduction

Microsoft Dynamics AX 2009

RunBase Patterns

White Paper

This white paper introduces patterns for addressing Remote Procedure Call (RPC) induced performance problems that are encountered when you refactor the *RunBase* class to run on both the client and server tiers.

Date: September 9, 2008

http://www.microsoft.com/dynamics/ax

Table of Contents

Spacing paragraph

Introduction

During development of Microsoft Dynamics AX 2009, the product's global performance team defined a general pattern with the goal of reducing Remote Procedure Calls (RPCs) across the application.

The general pattern is as follows:

1. Change the *RunBase*-derivative class to be "RunOn = Called from".
2. Change any caller menu items to be "RunOn = Server".
3. Change the *main* method to be "RunOn = Called from" by setting its modifier to include the keywords "*client server"*.

Of course, in programming things are rarely as easy as they seem, and this was no exception, as many issues were encountered while implementing the general pattern. This white paper outlines these issues and the patterns that can be used to work around them.

Issues

Issue #1: The #CurrentList macro is used for both *SysLastValue* and Pack/Unpack serialization.

In most *SysPackable* implementations, the #CurrentList macro defines a serialization list for storing both a user's last entered values (*SysLastValue*) and for moving a serialized version of the class across application tiers. This overlap in functionality begins to fail when the serialization list includes items that should not be persisted by *SysLastValue*. In order to work around this, you should implement the pattern in Appendix A, Listing A.1 (taken from the class *LedgerJournalSave*) if you are extending *RunBase*. Note that the *pack* and *unpack* method implementations are unchanged, so those methods are not shown.

If support for inheritance is not required, then you can remove the *unpackSysLastValues* and *packSysLastValues* methods and change the *getLast* and *saveLast* methods as shown in Appendix A, Listing A.2.

Important: If you are extending *RunBaseBatch* instead of *RunBase,* use the pattern that is listed in Appendix A, Listing A.3. Note the extra calls to `xSysLastValue::saveLast(this.batchInfo())` and `xSysLastValue::getLast(this.batchInfo())` in *saveLast* and *getLast*. If inheritance support is not required by your *RunBaseBatch*-derivative class, use the pattern shown in Appendix A, Listing A.2 with the added calls in *saveLast* and *getLast* as highlighted in Appendix A, Listing A.3.

Issue #2: Non-packable objects are created on the dialog, which runs on the client, but are used on the *RunBase-derivative* class, which runs on the server.

In one case, we encountered a *RunBase*-derivative with a *\classes\NumberSeq* class field. This seemed fine until we discovered that the *NumberSeq* class was being constructed on the *client* tier—instead of the server tier—by way of the prompted dialog. That is, although the field was being initialized on the client, it was not being passed back to the server tier.

In order to work around this cross-tier data sharing issue, we implemented the *SysPackable* interface on the *NumberSeq* class so that it could be packed and unpacked across tiers.

The implementation of *SysPackable* may not always be possible because not all classes are capable of being fully serialized. However, if the dependent class is fully serializable, you should implement the *SysPackable* interface, perhaps by leveraging the patterns from Issue #1 (in this list).

Issue #3: Client interaction is being performed in the *main* method.

We encountered one case where the RPC count actually increased after we moved the class to the performance team's pattern.

We determined that this increase occurred because the *main* method was performing large amounts of client interaction through *Args* (which is constructed on the client) and *FormDatasource* objects (*refresh*, and so on).

You can fix this increase in RPC in one of two ways:

- Change the *main* method to run on the client, and then create a static method that calls *run* on the server.
- Leave the *main* as called from, and create static client methods to strip values from *Args* and perform operations, such as *FormDatasource.refresh*.

For examples of each option, see Appendix A, Listing A.4 and Listing A.5; Listing A.5 shows an improved version of Listing A.4 with fewer RPCs.

Issue #4: Significant server interaction is being performed on the dialog or in the *dialog* method.

We found a few *RunBase*-derivative classes that were performing database access in their *dialog* method or in the dialog that they had presented themselves.

To check for database access, use the Trace Parser tool or manual investigation to see whether the *dialog* method is spawning a custom dialog through a form or if it is making method calls, which perform database access. If any of the aforementioned behavior is discovered, you should move the identified code into *static server* methods, with results passed back by using containers or some other serialized (packed) list structure. Note that returning a handle to a reference type that is created on a different tier will result in RPCs, and this behavior drives the requirement of using a container, which is passed as a value type, or a packed list type, which can be deserialized on the calling tier.

Issue #5: The *RunBase*-derivative class needs to load data from a file that exists on the client, but the class will be running on the server. The *RunBase*-derivative class needs to load data from a user-specified file that exists on the user's client machine, but the class' *run* method will be executing on the server.

When a user specifies the path to a file within a dialog that is running on the client, you cannot assume that the file is available on the server as the path is relative to the client machine.

One way to work around this accessibility issue is to load the file by using a static client method, with the data returned to the server tier in a container or a serialized (packed) list structure. Be sure not to return a reference type, or you will encounter RPCs. See Appendix A, Listing A.6 for an example of doing this for a client-stored file that contains a list of integers. Note that any packable structure that stores your data could be used instead of a *List*.

Important: If the class loading the file is a *RunBaseBatch* derivative, the file will need to be loaded in advance of executing the *run* method, because the batch system will not have access to the client at that time. Furthermore, if the file is very large, you should consider forcing the path to be a Universal

Naming Convention (UNC) path (a network path) that the server has access to so that the file can be accessed directly, without the need of persisting it to the database at any point during the batch process.

Appendix A

```
public class LedgerJournalSave extends RunBase
{
    // The following fields can be directly  persisted to a
    // container because they are primitive data types.
    LedgerJournalNameId         ledgerJournalName;
    LedgerJournalId             ledgerJournalId;
    LedgerJournalNameId         toLedgerJournalName;
    LedgerJournalId             toLedgerJournalId;
    Name                        toName;

    // The following fields cannot be directly persisted to a
    // container because they are object types.
    LedgerJournalPeriodicCopy   journalSave;

    DialogField                 dialogToJournalNum;
    DialogField                 dialogToJournalName;
    DialogField                 dialogToName;

    // This list defines which fields will be persisted and
    // restored by way of the system's SysLastValue functionality.
    #DEFINE.SysLastValuesCurrentVersion(100)
    #LOCALMACRO.SysLastValuesList
        LedgerJournalName,
        LedgerJournalId
    #ENDMACRO

    #DEFINE.SysLastValuesForkedFromVersion(1) // Defines which version of CurrentList 
that
                                              // SysLastValuesList forked from.

    // This list defines how the object is serialized and unserialized
    // as it is sent across the wire. This list should contain all
    // nonobject types that are defined on this class declaration.
    #DEFINE.CurrentVersion(2)
    #LOCALMACRO.CurrentList
        ledgerJournalName,
        ledgerJournalId,
        toLedgerJournalName,
        toLedgerJournalId,
        toName
    #ENDMACRO

    #DEFINE.Version1(1)
    #LOCALMACRO.CurrentListV1
        LedgerJournalName,
        LedgerJournalId
    #ENDMACRO
}

/// <summary>
/// Retrieves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void getLast()
{
    container packedValues;
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    // super();

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    getLastCalled   = true;
    inGetSaveLast   = true;

    // Restore the pertinent values from the SysLastValue table.
    packedValues = xSysLastValue::getValue(this.lastValueDataAreaId(),
                                           this.lastValueUserId(),
                                           this.lastValueType(),
                                           this.lastValueElementName(),
                                           this.lastValueDesignName());

    this.unpackSysLastValues(packedValues);

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast = false;
}

/// <summary>
/// Saves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void saveLast()
{
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    //super();

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast   = true;

    // Persist the pertinent values to the SysLastValue table.
    xSysLastValue::putValue(this.packSysLastValues(),
                            this.lastValueDataAreaId(),
                            this.lastValueUserId(),
                            this.lastValueType(),
                            this.lastValueElementName(),
                            this.lastValueDesignName());

    // The following fields must be maintained to mimic the functionality of the
overriden
```

```
    // method.
    inGetSaveLast = false;
}

/// <summary>
/// Unpacks the object for <c>SysLastValue</c> framework to support persisting user input.
/// </summary>
/// <param name="_packedValues">
/// A packed instance of <c>LedgerJournalSave</c>.
/// </param>
/// <remarks>
/// This methods brings support for inheritance to the <c>SysLastValue</c> implementation
on this class. Derivative classes should override this method and
/// provide their own implementation with an unpack list such as:
///     [#SysLastValuesList, baseClassPackedValues] = _packedValues;
///     super(baseClassPackedValues);
/// </remarks>
public void unpackSysLastValues(container _packedValues)
{
    Version version = RunBase::getVersion(_packedValues);
    ;

    switch (version)
    {
        case #SysLastValuesCurrentVersion:
        {
            [version, #SysLastValuesList] = _packedValues;

            break;
        }
        case #SysLastValuesForkedFromVersion:
        {
            // This pack list came from the forked version of the
            // CurrentList list.
            [version, #CurrentListV1] = _packedValues;

            break;
        }
    }
}
/// <summary>
/// Packs the object for the <c>SysLastValue</c> framework to support persisting user
input.
/// </summary>
/// <returns>
/// A container storing the list of values that are specified by the
<c>SysLastValuesList</c> macro.
/// </returns>
/// <remarks>
/// This methods brings support for inheritance to the <c>SysLastValue</c> implementation
on this class. Derivative classes should override this method and
/// provide their own implementation with a pack list such as "[#SysLastValuesList,
super()]".
/// </remarks>
public container packSysLastValues()
{
    ;

    return [#SysLastValuesCurrentVersion, #SysLastValuesList];
}
```
Listing A.1—Implementation of split serialization and *SysLastValue* list for *RunBase*-derivative classes

```
/// <summary>
/// Retrieves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void getLast()
{
    Version version;
    container packedValues;
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    // super();

       // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    getLastCalled   = true;
    inGetSaveLast   = true;

    // Restore the pertinent values from the SysLastValue table.
    packedValues  = xSysLastValue::getValue(this.lastValueDataAreaId(),
                                            this.lastValueUserId(),
                                            this.lastValueType(),
                                            this.lastValueElementName(),
                                            this.lastValueDesignName());

    version = RunBase::getVersion(packedValues);

    switch (version)
    {
        case #SysLastValuesCurrentVersion:
        {
            [version, #SysLastValuesList] = packedValues;

            break;
        }
        case #SysLastValuesForkedFromVersion:
        {
            // This pack list came from the forked version of the
            // CurrentList list.
            [version, #CurrentListV1] = packedValues;

            break;
        }
    }

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast = false;
}

/// <summary>
/// Saves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void saveLast()
{
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    //super();

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast   = true;


    // Persist the pertinent values to the SysLastValue table.
    xSysLastValue::putValue([#SysLastValuesCurrentVersion, #SysLastValuesList],
                            this.lastValueDataAreaId(),
                            this.lastValueUserId(),
                            this.lastValueType(),
                            this.lastValueElementName(),
                            this.lastValueDesignName());

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast = false;
}
```

Listing A.2—*getLast* and *saveLast* without inheritance support for *RunBase*-derivative classes

```
public class MyClass extends RunBaseBatch
{
    // The following fields can be directly  persisted to a
    // container because they are primitive data types.
    // Primitives

    // The following fields cannot be directly persisted to a
    // container because they are object types.
    // Object

    // This list defines which fields will be persisted and
    // restored by way of the system's SysLastValue functionality.
    #DEFINE.SysLastValuesCurrentVersion(100)
    #LOCALMACRO.SysLastValuesList
        // Primitives
    #ENDMACRO

    #DEFINE.SysLastValuesForkedFromVersion(1) // Defines what which version of
CurrentList that
                                              // SysLastValuesList forked from.

    // This list defines how the object is serialized and unserialized
    // as it is sent across the wire. This list should contain all
    // nonobject types that are defined on this class declaration.
    #DEFINE.CurrentVersion(2)
    #LOCALMACRO.CurrentList
        // Serialization list.
    #ENDMACRO

    #DEFINE.Version1(1)
    #LOCALMACRO.CurrentListV1
        // Old serialization list
    #ENDMACRO
}

/// <summary>
/// Retrieves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void getLast()
{
    container packedValues;
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    // super();

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    getLastCalled   = true;
    inGetSaveLast   = true;

    // Restore the pertinent values from the SysLastValue table.
    packedValues = xSysLastValue::getValue(this.lastValueDataAreaId(),
                                                this.lastValueUserId(),
                                                this.lastValueType(),
                                                this.lastValueElementName(),
                                                this.lastValueDesignName());

    this.unpackSysLastValues(packedValues);

    // This is a RunBaseBatch derivative class. Manually restore the user's
    // settings on the batch tab because super() is not being called.
    xSysLastValue::getLast(this.batchInfo());


    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast = false;
}

/// <summary>
/// Saves the <c>SysLastValue</c> record for this user and object.
/// </summary>
/// <remarks>
/// This is a customization of the standard SysLastValues functionality
/// and it is required to support the split implementation of serializing
/// (by way of pack and unpack) and persisting user input (SysLastValue).
/// </remarks>
public void saveLast()
{
    ;

    // Do not make the call to super because the #CurrentList and #SysLastValues
    // lists are different, requiring that we have specialized logic for the
    // SysLastValues implementation.
    //super();

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast   = true;

    // Persist the pertinent values to the SysLastValue table.
    xSysLastValue::putValue(this.packSysLastValues(),
                            this.lastValueDataAreaId(),
                            this.lastValueUserId(),
                            this.lastValueType(),
                            this.lastValueElementName(),
                            this.lastValueDesignName());

    // This is a RunBaseBatch derivative class. Manually save the user's
    // settings on the batch tab because super() is not being called.
    xSysLastValue::saveLast(this.batchInfo());

    // The following fields must be maintained to mimic the functionality of the
overriden
    // method.
    inGetSaveLast = false;
}

/// <summary>
/// Unpacks the object for <c>SysLastValue</c> framework to support persisting user
input.
```

```
/// </summary>
/// <param name="_packedValues">
/// A packed instance of <c>LedgerJournalSave</c>.
/// </param>
/// <remarks>
/// This methods brings support for inheritance to the <c>SysLastValue</c>
implementation on this class. Derivative classes should override this method and
/// provide their own implementation with an unpack list such as:
///     [#SysLastValuesList, baseClassPackedValues] = _packedValues;
///     super(baseClassPackedValues);
/// </remarks>
public void unpackSysLastValues(container _packedValues)
{
    Version version = RunBase::getVersion(_packedValues);
    ;

    switch (version)
    {
        case #SysLastValuesCurrentVersion:
        {
            [version, #SysLastValuesList] = _packedValues;

            break;
        }
        case #SysLastValuesForkedFromVersion:
        {
            // This pack list came from the forked version of the
            // CurrentList list.
            [version, #CurrentListV1] = _packedValues;

            break;
        }
    }
}
/// <summary>
/// Packs the object for the <c>SysLastValue</c> framework to support persisting user
input.
/// </summary>
/// <returns>
/// A container storing the list of values that are specified by the
<c>SysLastValuesList</c> macro.
/// </returns>
/// <remarks>
/// This methods brings support for inheritance to the <c>SysLastValue</c>
implementation on this class. Derivative classes should override this method and
/// provide their own implementation with a pack list such as "[#SysLastValuesList,
super()]".
/// </remarks>
public container packSysLastValues()
{
    ;

    return [#SysLastValuesCurrentVersion, #SysLastValuesList];
}
```

Listing A.3—Implementation of split serialization and *SysLastValue* list for *RunBaseBatch*-derivative classes

```
public client static void main(Args args)
{
    RunBaseClass runBaseClass;
    SomeTable someTable;
    ;

    ←Execution Starts Here On The Client
    // Any call to args would cause a jump back to the client
    // if we were running this code on the server.
    if (args && args.caller())
    {
        someTable = args.record();
    }

    // The construct method *must* be set to run on server.
    runBaseClass = RunBaseClass::construct();

    if (runBaseClass.prompt())
    {
       // Because the object was constructed on the server, the run method will
        // execute on that tier.
        runBaseClass.run();
    }

    // This would be a very expensive call if this code were running on the server.
    if (someTable.isFormDatasource())
    {
        someTable.datasource().refresh();
    }
}
```
Listing A.4—*main* method deviated to “RunOn=Client”

```
public client server static void main(Args args)
{
    RunBaseClass runBaseClass;
    SomeTable someTable;
    ;

    ←Execution Starts Here On The Server (menu item set to "RunOn=Server")
    someTable = RunBaseClass::getTableBufferFromArgs();

    // The construct method should either be set to run on server "RunOn=Server"  or
"RunOn=called from".
    runBaseClass = RunBaseClass::construct();

    if (runBaseClass.prompt())
    {
         // Because the object was constructed on the server, the run method will
execute on that tier.
        runBaseClass.run();
    }

    RunBaseClass::refreshFormDatasource(someTable);
}

protected client static SomeTable getTableBufferFromArgs(Args _args)
{
    SomeTable someTable;
    ;

    // Any call to args would cause a jump back to the client if we were running this
    // code on the server.
    if (args && args.caller())
    {
        someTable = args.record();
    }

    return someTable;
}


protected client static void refreshFormDatasource(SomeTable _someTable)
{
    ;

    // This would be a very expensive call if this code were running on the server.
    if (_someTable.isFormDatasource())
    {
        _someTable.datasource().refresh();
    }
}
```
Listing A.5—*main* method leveraging helper methods to perform client interaction

```
public class MyClass extends RunBase
{
    str fileFullPath; // Stores the file's full path.
    // ...Snip...
}

public void run()
{
    List loadedFile;
    ;

    ←Execution Starts Here On The Server

    // Load the file on the server.
    loadedFile = List::create(MyClass::loadFileOnClient(fileFullPath));

    // The list now exists on the server so we can
    // code against it all day from the server
    // and never encounter a RPC.
}

private static client container loadFileOnClient(str _fileName)
{
    List loadedFile = new List(Types::Integer);
    ;

    // Load the file while executing on the client
    // so that the file path makes sense, as it
    // was likely relative to the client user's computer
    // and not the AOS.

    // Load the file into the list...
    // ... Snip ...

    // Return a packed version of the list.
    return loadedFile.pack();
}
```

Listing A.6—Loading a file on the client with the *run* method executing on the server

Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll Free 1-888-477-7989

Worldwide +1-701-281-6500

www.microsoft.com/dynamics