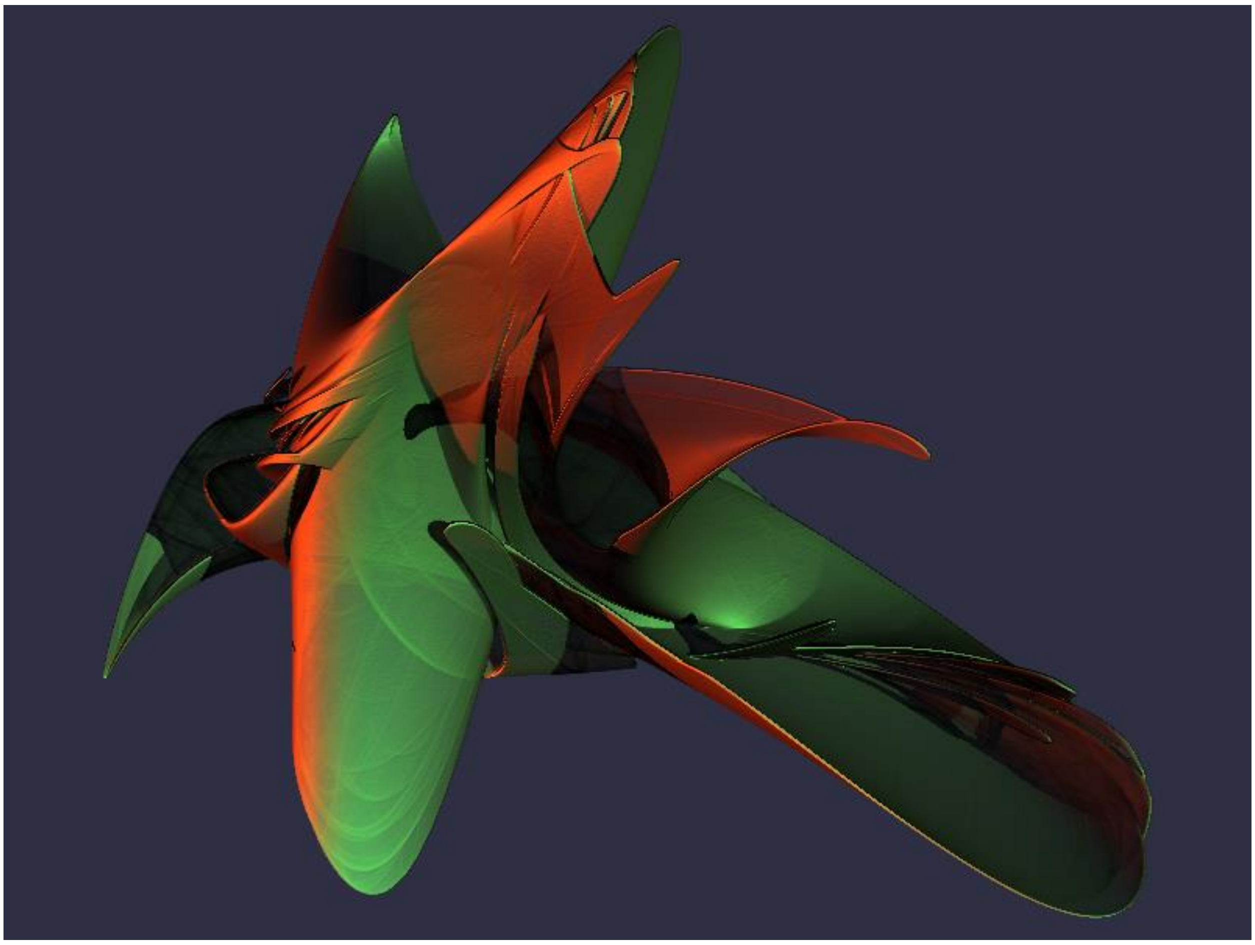
DX Tools

Application Design Guide 3



Author: Ian Tree
Owner: HMNL b.v.
Customer: Public
Status: QE
Date: 10/03/2015 15:46
Version: 3.14.0

Disposition: Open Source

Document Usage

Revision History

Date of this revision: 10/03/2015 15:46	Date of next revision *None*

Revision Number	Revision Date	Summary of Changes	Changes marked
0.1	24/01/12	Initial Base Version	No
3.12.0	03/04/12	QE Version	No
3.14.0	10/03/15	Updated for x64 support	No

Acknowlegements

Frontpiece Design was produced by the chaoscope application.

CHAOSCOPE.org
3D strange attractor rendering software

Contents

1. Introduction to DX Applications

The Domino eXplorer (DX) was developed as a means for facilitating the rapid development of tools to be used in projects that involve high volumes of data transformation. DX has been, and continues to be developed for use across a wide range of Domino versions and platforms. The reference platforms are Domino 9.0.x on Windows Server 2003 R2 (32 and 64 bit) and Red Hat Linux 6.6. DX is also used as a research tool to investigate various aspects of Autonomic Systems, in particular Autonomic Throughput Optimisation.

Standardised utilities have also been built around some of the functional DX classes, these are published as “DX Tools” and can save time by providing off-the-shelf processing to be incorporated into complex transformations that need high throughput rates.

DX is **NOT** a framework (we hate frameworks), instead it provides a grab bag of classes that can be assembled in different designs to provide high throughput processing of Notes databases. There are certainly some constraints imposed by the relationships between different objects and contracts imposed by the API, these have been kept as minimal as possible.

This document provides an insight into how DX applications can be put together. The document is based around a description of building two applications, the first is a single threaded applications intended to be run from the command line that updates database ACLs. The second application is the Database Copier tool (QCopy) that is a multi-threaded Server Add-In task. As the structure and design of these applications is explored generic issues of DX application design, build, deployment and tuning are discussed.

2. Designing and Coding DX Applications

This section describes some of the guiding principles involved in designing and coding applications that use the Domino eXplorer (DX).

However do remember ………

“Rules are for the guidance of wise men and obedience of fools.” *Douglas Bader*.

2.1 C++ Style

Workmanlike; a premium is put on portability, readability and maintainability rather than slickness. Coding stays close to ANSI C with objects but this is not enforced as a rigid standard. Quality Engineering reviews do sometimes eliminate some of the “uglier” coding constructs, however, when applying updates to the code base the “if it isn’t broken then don’t fix it” dictate applies.

Historically the code base has evolved over a long period of time and on different platforms and so there is some variability in coding styles evident in the implementation.

2.1.1 Function Boilerplate

The implementation of functions should adopt the following style rules.

Rule #1: All resources allocated or opened in a function, with the exception of those returned by the function, should be freed or closed within the same function in which they were allocated or opened.

Rule #2: Resources that are required by multiple functions that are invoked sequentially should be allocated or opened and freed or closed in a higher level function and passed as parameters.

Rule #3: Check the validity of passed parameters at the top of a function and exit immediately if any out-of-envelope condition is detected.

Rule #4: Progressively allocate or open all resources needed within a function at the top of the function (after parameter safety checks), any failure to acquire a resource should trigger the close or free of any resources acquired up to the failure point and an immediate return to the invoking function.

Rule #5: Favour linear processing steps with early function exits in place of deep nesting in the implementation of the mainline code of the function.

Rule #6: Implement polymorphic functions as mapping functions between each polymorphic form and a common base function.

2.2 C API versus C++ API

The first reason that the Notes C API is used in preference to the Notes C++ API is largely historical, in the “old” days the C++ API was a somewhat flaky implementation prone to leaks, exceptions and other oddities and therefore the Notes C API was the only industrial strength option available.

There are two other reasons that continue to favour the use of the Notes C API over the C++ API. The Notes C API offers additional capabilities over those exposed by the C++ API which would lead to the DX implementation being a heterogeneous mix of both APIs. Use of the C++ API would impose a base containment pattern that would influence the Object Model design of the DX Kernels and DX applications in a way that would not be optimal for high throughput systems.

The DX implementation continues to use the Notes C API, the reference version of DX uses the R8.5 release of the API however applications have been built using levels back to the R5.0 API without problems. The R8.5 DHANDLE is implemented by the DX headers if a release of the API is used that does not implement it natively.

2.3 ASCII versus UNICODE

For historical and cross-platform compatibility reasons the Domino eXplorer code used the ASCII coding set. There is no fundamental reason at this time why the kernel and applications could not be migrated to a UNICODE model however there a minimal benefits from this and therefore ASCII remains the de-facto implementation standard.

2.4 32 Bit versus 64 Bit

For historical and cross-platform compatibility reasons the Domino eXplorer code uses a 32 Bit implementation model. Again porting to a 64 Bit model would be possible however lack of demand and relatively small benefits dictate that a 32 Bit implementation remains the reference model.

2.5 Object Model Design

The DX kernel implementation has been designed to support the development of sustainable high throughput applications. Experience has shown that the most successful approach to the Object Model is to follow the processing model rather than a more data oriented approach.

2.6 Standard DX Header Files

The following describe the contents of the standard generic include header files for DX, refer to the “DX Class Catalogue” for object specific include header files.

- **PlatBase.h** – This header file includes C Run Time and OS specific headers needed by DX applications certain cross-platform mapping macros are implemented in addition to debugging definitions. This should be the first include file in any module.
- **NotesBase**.**h** – This header file includes the essential and the most commonly used Notes API include files it also makes some cross version compatibility definitions. This header file should be included after the PlatBase.h header.
- **DXGlobals.h** – This header files contains definitions used by application code and the DX kernel code, it should be included after the NotesBase.h haeder.

2.7 Project Directory Structure

The default Domino eXplorer development environment follows the Visual Studio paradigm of a “Solution” directory that contains multiple “Project” directories with a single application directory per application, this paradigm is followed on both Windows and Linux development environments.

The standard DX header and code files are **NOT** designed to be added to the default include search directories on either environment. The package expects to find a directory called “DXCommon” as a project level directory in each solution directory that will be used to build DX applications, these should be included in applications by relative re-direction. An include statement for the PlatBase.h header file that is located in the “Platform” sub-directory of the DXCommon package would be coded as follows.

```
//  Platform Includes
#include     "../DXCommon/Platform/PlatBase.h" //  Basic platform includes
```

Although a copy of the DXCommon package could be physically placed in each solution directory it would be more usual to place the package in a shared location on the development workstation/server and then create a symbolic link in each of the solution directories.

Windows:

The DXCommon kernel is supplied as a zipped archive (.zip). The contents of the archive should be unpacked to either the <solution directory>\DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>\DXCommon directory.

As an example.

Unpack the DXCommon kernel into a directory “c:\usr\include\DXCommon-3.12.0” and then create the symbolic link from within the solution directory using the following command.

```
mklink /D DXCommon “c:\usr\include\DXcommon-3.12.0”
```

Linux:

The DXCommon kernel is supplied as a gzipped archive (.tar.gz). The contents of the archive should be unpacked to either the <solution directory>/DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>/DXCommon directory.

File ownership and access settings should be adjusted according to your local policies.

As an example.

Unpack the DXCommon kernel into a directory “/usr/include/DXCommon-3.12.0” and then create the symbolic link from within the solution directory using the following command.

```
ln -s /usr/include/DXCommon-3.12.0 DXCommon
```

This deployment model allows different levels of the DXCommon package to be used in different solutions without reconfiguring the development environment.

2.8 Reference Platforms

DXTools and the DXCommon kernel are portable across multiple platforms that support the Notes API. However there are a limited set of reference environments on which they are regularly built and regression tested.

Windows:

Build Environment:

Microsoft Visual Studio 2005/2008/2010/2011

Version 8.0.50727.867 (vsvista.050727-8600)

Running on any supported windows workstation.

Note: Backward compatibility tests are done with Visual Studio 2003 as that is the officially supported development platform for the Notes API.

Notes API Version 9.0.

Execution Environment:

Windows Standard Server 2008 R2 (32 bit and 64 bit).

Domino Server 9.0.1 FP1.

Note: Execution environments from Domino 6.5.x through 9.0.x are regularly used.

Linux:

Build Environment:

Gcc Version: 4.1.2 for i386-redhat-linux.

Running on Redhat Linux 6.6.

Notes API Version 9.0

Execution Environment:

Redhat Linux 6.6

Domino Server 9.0.1 FP1.

Note: Execution environments from Domino 7.0.x through 9.0.x are regularly used.

3. A Single Threaded Command Processor

This section examines the design and coding of a single threaded command processor that is intended to run on workstations. The sample application that will be discussed in this section is the ACLMorph (ACL Transformer) application.

```
ACL Transformer

This utility command processor will update the ACL of all datases within the
specified source so that they conform to the specification of the supplied
XML ACL Pattern.

USAGE:

ACLMorph ScopeServer ScopeDb ACLPatternURL [-V|-T[:Area]|-D[:Area]] [-E][-S][-N]

ScopeServer   - The Server Name of the target | Domain | *
ScopeDb             - The Database Name of the target | Directory | *
ACLPatternURL- The URL of the ACL Pattern XML document
[-V]|[-T[:Area]]|[-D:Area] - Set logging level to Verbose, Trace (optionally the area
to trace) or Debug.
-S     - Optional - skips updating the ACL even if changes were made
-N     - Optional - do not recurse into sub-directories
```

3.1 Task 1: Populate the RunSettings Object

The normal method of populating the RunSettings object is to extend the RunSettings class with a custom (AppRunSettings) class that will derive application settings that may change from run to run and populate those at the same time as populating the base RunSettings members. This processing is not compulsory so long as this phase yields a correctly populated RunSettings object.

The command line arguments are passed to the constructor of the AppRunSettings class, the constructor will set any default values and parse the command line arguments to populate members in the base and extending class.

Some of the parameters define the configuration of the application, these would be set as default values in the application code. The ACLMorph application is a standalone application and does not use a repository database and does not keep an application log, all logging messages are output to the console. The following default settings are made in the AppRunSettings class to specify this configuration.

```
RunningAsAddin = FALSE;            //  Not Running as server Addin Task
NoRepository = TRUE;               //  Do not allow use of the repository
NoAppLog = TRUE;                   //  Do not allow Application Event Logging
EchoLog = TRUE;                    //  Force Echo to the console
CreateRepository = FALSE;          //  Do not Create the repository
```

Once the AppRunSettings object is created the mainline code should check two switches in the object to determine if the application should proceed.

```
//  Validate and build the Run Settings for this execution
arsLocal = new AppRunSettings(argc, argv);
if (!arsLocal->AllowExecution)       //  Exit silently if just showing usage
{
       return APPRC_NOERROR;
}
```

```
if (!arsLocal->IsValid)        //  Abort if validation failed
{
      std::cout << MSG_ACM0030S << std::endl;
      return APPRC_FATAL;
}
```

The AllowExecution flag is set to FALSE if the parameters were valid but indicated that the switches on the command line (-?) indicated that the application usage messages should be shown and no execution atrtempted. In this case the application just silently terminates, the console will show the application usage messages.

The IsValid is set to FALSE if the application parameters were invalid or any other condition prevented the valid instantiation of the AppRunSettings object. The application terminates with an error message showing that the application could not be started.

Populating the RunSettings object is completed by setting the application name, short title and version in the appropriate members.

```
//  Set the identification in the Run Settings
strcpy_s(arsLocal->APPName, MAXAPPNAME, APP_NAME);
strcpy_s(arsLocal->APPTitle, MAXAPPTITLE, APP_TITLE);
strcpy_s(arsLocal->APPVer, MAXAPPVERSION, APP_VERSION);
```

The APP_NAME, APP_TITLE and APP_VERSION symbolic values are defined in the application header file.

It should be noted that there is not a permanent application log available at this stage of processing, even if the application intended to use one so all output is directed to STDOUT.

Although this application has been designed to execute from a command line on a user workstation there is nothing to prevent it from being run on a server using a “Load” command from the local or remote server console or from a program document.

3.2 Task 2: Initialising the Run Time

The AppRunSettings object is passed to the constructor of the single threaded run time (ExecEnvironment).

Initialising the Run Time will accomplish the following tasks.

- Initialise the Notes Runtime
- If the application is using a repository database this will be opened and the DBHANDLE made available
- Logging will be initialised and directed to the appropriate destination(s)
- A default elapsed timer will be initialised

In the case of the ACLMorph application there is no repository used and the only destination for logging is the command line console.

After creating the run time object applications should test the IsInitialised member to make sure that the run time was properly initialised as indicated by a value of TRUE for the member.

```
//  Initialise the runtime environment
xeLocal = new ExecEnvironment(arsLocal, argc, argv);
```

```
if (!xeLocal->IsInitialised)      //  Abort if runtime failed to initialise
{
       std::cout << MSG_ACM0030S << std::endl;
       delete arsLocal;
       return APPRC_FATAL;
}
```

Assuming that the run time initialised successfully then from this point on in the applications the logging interface can be used and Notes API calls can be made. If a repository is used by the application then the database would be open and available for use from this point.

Once initialised the ACLMorph application reports on the build version of the run time engine.

```
//  Report the Engine Build Information
xeLocal->GetBuildInfo_s(szBuildID, MAX_BUILD);
sprintf_s(szMsg, MAX_MSG, MSG_ACM0032I, szBuildID);
xeLocal->LogVerbose(szMsg);
```

The message will only be reported to the console if the application is running in Verbose, Trace or Debug logging levels. The output sent to the console is as follows.

```
ACM0032I: Using Runtime Version: DXCommon ST Runtime 3.12.0 (build: 105).
```

3.3 Logging

If a permanent logging destination is specified for the application then this can either be (default) the database that is specified as the repository database or the current log.nsf workstation or server Notes Log. Logging messages may be echoed to a command line console or a Domino server console.

Logging messages are written in the same form as the standard logging performed by Notes log messages are written to a series of “Events” documents that will display in the “Miscellaneous Events” view in a standard log or an equivalent view in a repository database. When designing a repository database the views and form for displaying these logging messages can be copied from a standard Note Log template.

The logging functions in the kernel generate a value for the “Server” item in the “Events” documents in the form “<app title>(<Server or User Common Name>)”, this creates a unique category in the Miscellaneous Events view for each instance of a DX application.

Messages that are generated and logged by the kernel, functional DX objects and the standard DX tools are always prefixed by a message ID. The message ID consists of 3 alpha characters identifying the source of the message e.g. “ACM” for the ACLMorph application layer. This is followed by four numeric characters that uniquely identify the message and finally a single alphabetic character that identifies the severity level of the message with the following meanings.

- T – A Trace message
- I – An Informational message
- W – A Warning message
- E – An Error message
- S – A Severe Error message

The convention of prefixing all logged messages with an identifier makes it easier to locate specific events in log files and easier to locate specific areas of code in an application where messages are being generated.

3.4 Task 3: Parse the ACL Rules XML document

The application constructs an ACLRuleSetParser object and then uses this object into an ACLRuleSet object.

```
//  Create the parser
rsParser = new DXACLRuleSetParser(xeLocal, 0);
//  Parse the ACL rule set
rsRules = rsParser->parseTheseRules(arsLocal->szACLRuleSet, 0);
```

If the XML document contains a valid definition for an ACLRuleSet than a pointer to the object is returned, if an object cannot be created then the parser returns NULL.

If the logging mode of the application is Verbose (or higher) then messages are issued by the parsing process.

```
DXR7004I: Loading resource source
'http://betamax.lan/DCF/Depot.nsf/Payloads/ACL.T01/$File/ACL-T01.xml' of type 3.
DXR7009I: Resource source
'http://betamax.lan/DCF/Depot.nsf/Payloads/ACL.T01/$File/ACL-T01.xml' has been loaded,
Length = 304 bytes.
DXR7108I: XML: <?xml version='1.0' encoding='utf-8'?>.
DXR7108I: XML: <!-- Add a generic server entry -->.
DXR7108I: XML: <ACLRuleSet>.
DXR7108I: XML: <CompulsoryEntries>.
DXR7108I: XML: <ACLRule Name="*/SERVER/ACME" Type=ServerGroup Level=Manager>.
DXR7108I: XML: <Option Type=NoDeleteDocs Set=Off/>.
DXR7108I: XML: <Role Name=* Assign=Yes/>.
DXR7108I: XML: </ACLRule>.
DXR7108I: XML: </CompulsoryEntries>.
DXR7108I: XML: </ACLRuleSet>.
DXR7130I: The rule set has been loaded from
'http://betamax.lan/DCF/Depot.nsf/Payloads/ACL.T01/$File/ACL-T01.xml'.
```

3.5 Task 4: Create a DbACLMorpher

The DbACLMorpher class is an application defined class that extends the DXReporter class, the DXReporter allows application code to be invoked for, most commonly, every database that is identified in the scope of a search performed by the Domino eXplorer. This is the mechanism that allows application code to be executed for a single database or for an arbitrary collection of databases.

```
//  Construct a new DbACLMorpher and condition it for execution with this rule set
amLocal = new DbACLMorpher(xeLocal);
amLocal->acsCurrent = rsRules;    //  Bind to the current rule set
if (arsLocal->bSkipUpdates) amLocal->SkipUpdates = TRUE;    //  Set Skip updates
```

Once constructed the DbACLMorpher is conditioned with the ACL Rule Set created in the previous step as well as setting a control flag that was derived from the command line, the flag indicates to the processing that any changes made to ACLs are not to be saved.

3.6 Task 5: Create a DominoExplorer

The DominoExplorer class provides the central engine that permits application code to be exercised against every database in an arbitrary collection. Using the Domino eXplorer cuts down considerably on the amount of custom application code that needs to be provided to action transformations on potentially large sets of databases.

```
//  Now construct a Domino Explorer to drive the whole process
dxLocal = new DominoExplorer(xeLocal, 0);
```

Once the object is constructed the application should test the IsStarted member for TRUE to ensure that the object has been correctly initialised.

3.7 Task 6: Create a Request and Invoke the Explorer

The application next creates a DXRequest object that describes the scope of the collection of databases that is to be processed.

```
//  Construct an explorer request from the run settings
dxrTop = new DXRequest();
dxrTop->RecurseDirectories = arsLocal->bNoRecursion;
//  Set the Database ACL Morpher in the request
dxrTop->ScannersNeeded = SCANNER_DATABASE;     //  Exit is for the D/B Level
dxrTop->Reporters = DXREP_DATABASE;            //  Reporter is for D/B Exits
dxrTop->dxrDatabase = amLocal;                 //  Set the object address

//  Attempt to parse the request scope - if valid then execute the request
if (dxrTop->ParseRequest(xeLocal, arsLocal->szRQScope1, arsLocal->szRQScope2))
```

The last step in the code shown above parses the two parameters supplied on the command line into a scope that identifies the collection of databases that are to be processed, the parse process returns TRUE if the scope could be determined from the passed parameters. If the parser returns FALSE then an explorer search should not be attempted with the supplied settings.

Once the request has been successfully created it is passed to the explorer to execute.

```
//  Pass the request to the Domino Explorer to process
if (!dxLocal->ProcessRequest(dxrTop, 0))
```

The call to ProcessRequest will return TRUE if the request was executed successfully and FALSE if the request failed or was only partially completed.

The log messages shown below show the messages issued by the explorer when running in verbose (or higher) logging mode. The example below is shown for a scan which has a scope of a single database alone.

```
DXR0015I: Database/Directory 'CN=Betamax/OU=SERVER/O=ACME!!ACL\TDB01.nsf' has been successfully opened.
DXR0074I: Remote open on server 'Betamax/SERVER/ACME' R8.5.2 FP:0 HF:0 Build:379 took 4509 ms (Latency: 0 = 30 + -30 ms.).
DXR1404I: Starting scan of database 'TDB01.nsf' in 'ACL'.
DXR1405I: Completed scan of database 'TDB01.nsf' in 'ACL'.
DXR1013I: The current request was processed successfully.
DXR1023I: 0 Servers were processed and 0 were dropped for this request.
DXR1024I: 0 Directories were processed and 0 were dropped for this request.
DXR1025I: 1 Databases were processed and 0 were dropped for this request.
```

The DXR0074I message in the logging above is the standard message shown by the kernel whenever a database is successfully opened. It shows where the database was opened, what version of Notes/Domino it is being served by, how long the open process took in milliseconds and the network latency of the connection over which it was opened.

3.8 Domino eXplorer Scope

The scope for a Domino eXplorer scan is specified by providing two null terminated character strings. The first of the strings specifies the server scope the second specifies the database scope. In addition a BOOL switch value is supplied indicating if scans can recurse into sub-directories.

The server scope can specify any of the following values.

- The abbreviated name of a server – only that server will be scanned.
- An empty string or the value “Local” - only the server or workstation on which the application is running will be scanned.
- @<domain name> - all servers that can be found that are in the specified Domino Domain will be scanned.
- @<domain name pattern> - the pattern can contain the “*” and/or “?” wildcard characters, any servers that can be found where the domain name matches the supplied will be scanned.
- An abbreviated server name pattern - the pattern can contain the “*” and/or “?” wildcard characters, any servers that can be found where the server abbreviated name matches the supplied pattern will be scanned.
- * - all servers that can be found will be scanned.

The database scope can specify any of the following.

- The path of a Notes database relative to the Notes Data Directory – this database will be scanned on all servers within the server scope.
- The name of a directory relative to the Notes Data Directory – all databases in the specified directory will be scanned on all servers within the server scope. All databases in sub-directories will also be scanned **ONLY** if the recursion control switch is set to TRUE.
- * - all databases on all servers within the server scope will be scanned.
- The path of a Notes database relative to the Notes Data Directory containing the “*” and/or “?” wildcard characters – any database matching the pattern on all servers within the server scope will be scanned.
- The name of s directory relative to the Notes Data Directory containing the “*” and/or “?” wildcard characters – any directories that match the pattern on all servers within the server scope will be scanned, including all sub-directories ONLY if the recursion control switch is set to TRUE.

In normal usage the server scope is supplied as a server name, unqualified or partially qualified specification for the server scope can cause very large quantities of processing. If the server scope is not fully qualified then scanning begins with the current or home server and every server visited is checked to see if it has server documents that identify servers that match the scope.

3.9 Task 7: Clean Up and Terminate the Application

Once the processing has been completed the application code should destroy the objects that have been created and then terminate the application. The run time and the associated RunSettings should be the last objects that are disposed of, this ensures that the logging interface is available for messages generated during the termination process.

```
//  Clean up the processing object
if (dxrTop != NULL) delete dxrTop;       //  Explorer Request
if (dxLocal != NULL) delete dxLocal;     //  Explorer
if (amLocal != NULL) delete amLocal;     //  DbACLMorpher
if (rsRules != NULL) delete rsRules;     //  ACL Rule Set
delete rsParser;                         //  ACL Rule Set Parser


//  Processing is completed - shut down, clean up and exit.
if (!xeLocal->Close())
{
       std::cout << MSG_ACM0031S << std::endl;
       return APPRC_FATAL;
}

//  Cleanup the local objects
delete xeLocal;             //  Run Time
delete arsLocal;            //  RunSettings


//  Terminate the application
return APPRC_NOERROR;
```

3.10 Active Code in the DbACLMorpher

In this application the active custom code is contained in the ReportOnThisDatabase interface in the DbACLMorpher class. The code is not large or complex mostly relying on code paths that are available in the DX functional objects being used. The ReportOnThisDatabase interface is invoked passing a handle to the database that is to be processed, the code firstly reads the ACL from the database.

```
//  Read the ACL of the current database
stAPIRC = NSFDbReadACL(hdbEntity, &hACL);
```

The call is a native Notes API call, so the returned status must be checked and any appropriate error processing performed if the status indicates a problem.

```
if (ERR(stAPIRC) != NOERROR)
{
       sprintf_s(szMsg, MAX_MSG, MSG_ACM0134E, dxrRQ->szDirectory, dxrRQ->szDatabase,
dxrRQ->szServer);
       xeLocal->LogMessage(szMsg, iThreadID);
       xeLocal->GetAPIMessage(stAPIRC, szMsg);
       xeLocal->LogMessage(szMsg, iThreadID);
}
```

In the case of a problem being indicated from the call to read the ACL, messages are issued and no further processing is attempted on this database ACL. The code above shows the use of the GetAPIMessage call in the kernel to obtain a formatted description of the error code.

The next step in the application code is to create a clone of the ACL Rule Set that is to be applied on this database. The step is not strictly necessary in this application context but is there for compatibility with use in multi-threaded applications. The DXACLRuleSet contains stateful information that is used in the execution of a request and therefore calls into the DXACLRuleSet are not re-entrant, hence the need to clone the object for use on each database.

```
//  Create a local clone of the Rule Set
arsClone = new DXACLRuleSet(xeLocal, iThreadID);
acsCurrent->clone(arsClone, iThreadID);
```

The cloned instance of the ACL Rule Set is then used to process against the database ACL and coerce changes to the ACL so that it conforms to the specification in the XML document provided to the application.

```
//  Coerce the ACL to conform to the Rule Set
if (arsClone->coerce(hACL, iThreadID))
```

The call to coerce will return a BOOL with value TRUE if the ACL has been updated by this action, the changes are only made to the in-memory ACL and the application needs to save the ACL back to the database if these changes are to be made permanent.

```
// Update the ACL - unless skipping
if (!SkipUpdates)
{
       //  Save the updated ACL
       stAPIRC = NSFDbStoreACL(hdbEntity, hACL, 0L, 0);
….
}
```

The return status from the Notes API call needs to be checked as usual.

All that remains now is to clean up and exit from the ReportOnThisDatabase interface.

```
delete arsClone;
//  Free the memory associated with the ACL
OSMemFree(hACL);
dxrRQ->ReturnCode = RETURN_NOERROR;
return TRUE;
```

A sample of the logging generated by the coerce processing running with a logging level of verbose or higher is shown below.

```
DXR7239I: ACL entry '*/SERVER/ACME' has been added.
DXR7240I: The ACL has had 1 updates applied and should be saved.
ACM0133I: The ACL has been updated in 'ACL\TDB01.nsf' on 'Betamax/SERVER/ACME'.
```

3.11 Summary

In the sample application a small amount of custom code is used to “stitch” together different DX elements to deliver a powerful application. The generic design pattern that is exposed can be used to mix DX elements and custom processing in an identical manner.

Build custom processing into an “engine” style class that is capable of performing the processing on a single database. Then construct a class that extends the DXReporter class and use the ReportOnThisDatabase interface to invoke the custom processing for each database that is passed to the interface.

The mainline code would follow the same pattern as the sample application, it largely consists of constructing the necessary objects initialising them as needed and then invoking the Domino eXplorer to drive the process against all databases within the scope for the current invocation of the application.

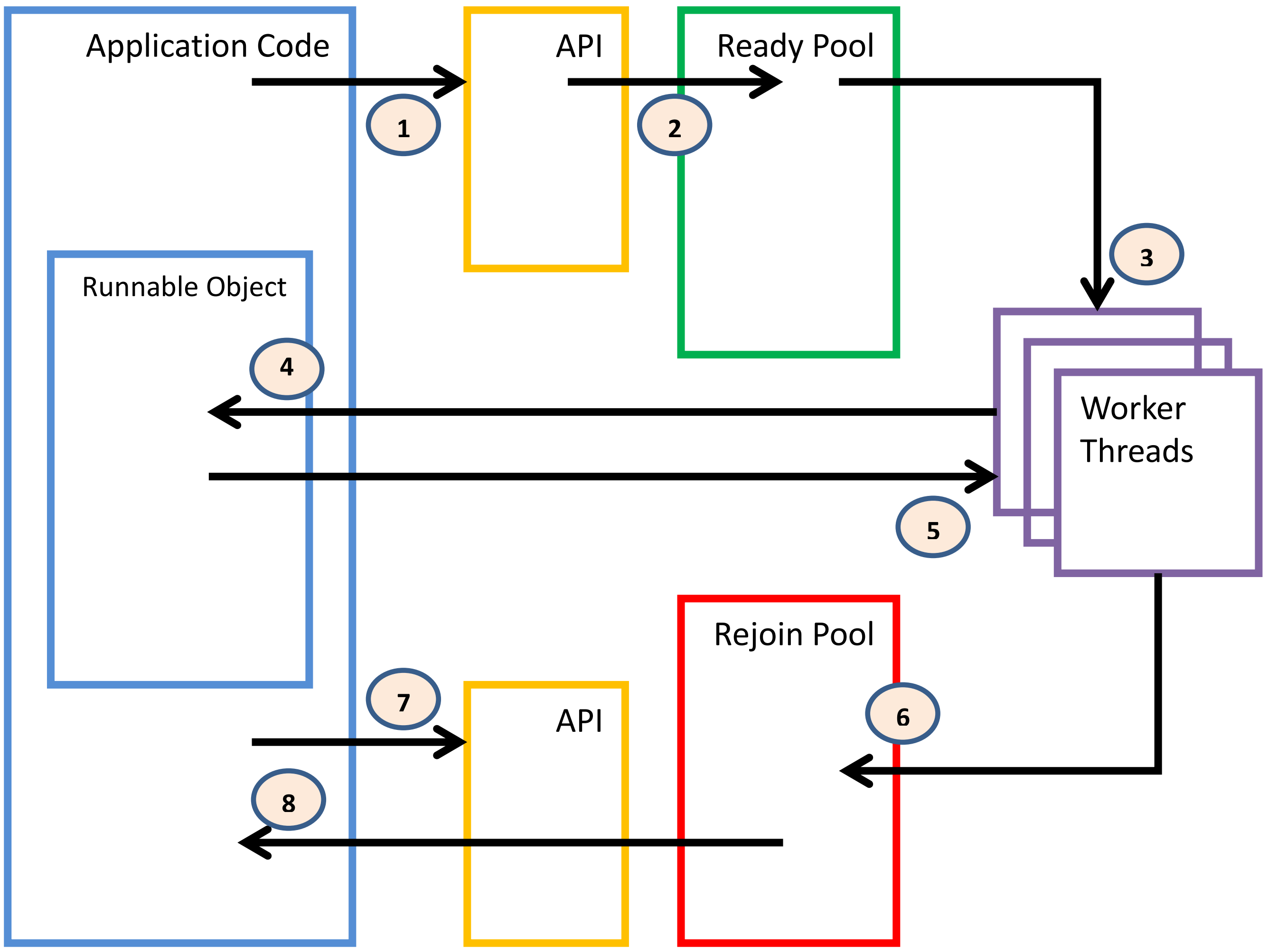
4. The DX Threading Model

This section of the document presents the key aspects of the threading model that a developer using the API should be familiar with and is presented here before looking at the implementation of a multi-threaded application in the next chapter.

4.1 Introduction

The DX threading model has been designed to present application developers with a simple architecture that is easy to design for and an API that is simple to use. The application interface is based on a message passing interface (MPI). Applications create request objects that describe some work that must be performed asynchronously and post these request to the kernel for execution. The kernel manages a pool of threads, the threads in the pool are homogeneous and can execute any request. The kernel will dispatch requests for execution by one of the worker threads in the thread pool. Once a request has completed the state change can be detected in the application code by a polling mechanism that is invoked through the API.

4.2 The Request Lifecycle



1

The application creates a request object and populates with the information needed to execute a chunk of work. The application the calls the “PostARequest” function in the run time API, passing the address of the request object and the address of the “Runnable” object that is to execute the request.

2

The run time API takes the information passed by the application code in the “PostARequest” call and stores it in the “Ready Pool” where it is available to be executed.

3

The kernel code monitors the pool of worker threads and as soon as one is available to run work it will locate the most appropriate request that is waiting in the “Ready Pool” and will dispatch it to the available worker thread for execution.

4

The worker thread will invoke the “ExecuteThisRequest” interface on the Runnable object to have the application code service the request. The application code will indicate the success or otherwise and return any needed information in the request object that was passed to it.

5

When processing or the request is completed the application code returns to the Worker Thread.

6

The Worker Thread stores the information in the “Rejoin Pool” and signals the kernel that it is available for processing work again.

7

The application code calls the “GetRejoinRequest” function in the run time API to poll the “Rejoin Pool” to see if request have completed processing.

8

If a request has been completed then the API will return the address of the completed request object. The application code then processes any returned information and disposes of the returned request object.

4.3 The Request Owner

Calls to the “PostARequest” and “GetRejoinRequest” functions in the run time API take a parameter of “Request Owner” this parameter is an arbitrary address encoded as a void pointer (void *). This parameter provides a mechanism for grouping bunch of requests together and localising the code that will process these grouped requests.

The kind of processing that the DX kernel was designed for often break down into a hierarchic pattern for parallel execution, one request will generate a number of sub-requests and each sub-request will, in turn, create a number of sub-sub-requests and so on. The owner mechanism can be used here to reflect the hierarchic workload, in this case the Owner for each request is set to the address of the parent request in the hierarchy. When polling for completed requests using the “GetRejoinRequest” the address of a parent request is specified as the owner and the return will signal when every sub-request that belongs to that parent has completed and therefore the parent processing can be completed.

4.4 Request Priority

Calls to the “PostARequest” functions in the run time accept a parameter that specifies the “Priority” of the request. The priority is specified as an arbitrary integer value with larger numbers being a higher (more urgent) priority. The priority is used by the kernel to determine which of the requests available in the “Ready Pool” will be the next to be dispatched to an available thread.

As a general rule workloads that follow the hierarchic model described in the section above should post requests at higher priorities the lower they are in the hierarchy.

The kernel also implements an optional, request priority ageing mechanism. When ageing is in effect then requests that reside in the “Ready Pool” have their priority increased at regular intervals. This mechanism is intended to prevent requests becoming stale while waiting to be executed and tying up resources while not contributing to throughput rates.

4.5 Constrained Multi Lane Scheduling

As pointed out in the earlier sections the kernel treats all worker threads as equals, any request can be dispatched to any worker thread that is available to process work. When a single request is being processed by a worker thread all processing for that request must be completed before the thread becomes available to process other requests, including the execution and rejoin processing of any sub-requests that are posted. There is a fundamental exposure from this model, it is possible for all threads to fill up with “higher level” requests leaving no worker threads available to execute the lower level requests that have been posted. In this scenario processing will simply grind to a halt with all worker threads waiting for sub-requests to complete, which they never will, or waiting for space to become available in the “Ready Pool” so that more sub-requests can be posted.

The kernel solves this thread exhaustion problem by providing a different scheduling mode “Constrained Multi Lane Scheduling” (CMLS). The CMLS mode is selected by setting the TPOOL_MODE_CMLS bit in the TPSchedMode member of the ThreadMnagerPolicy object that is used to configure the multi-threading kernel.

When running in CMLS mode the kernel still regards all worker threads as being equal and able to execute any request however it limits (constrains) the number of threads that can be concurrently executing requests from different levels in the workload hierarchy. CMLS identifies four arbitrary levels of request hierarchy. Level or Lane 0 defines service requests these requests would normally be running for the duration of the application. Level or Lane 1 defines requests that will themselves generate any number of what the application would recognise as unit transactions, these are referred to as feeder transactions. Level or Lane 2 defines unit transactions and Level or Lane 3 defines sub-requests or requests that will perform the work of a part of a transaction. The Level or Lane for an individual request is identified to the kernel by setting the appropriate bits in the Attributes flag that is passed in the call to “PostARequest”.

Constraints may be applied as a percentage of the threads in the thread pool that can be executing requests from levels 0, 1 and 2. These constraints are applied by setting the appropriate members in the ThreadManagerPolicy that used to initialise the kernel. The constraints not only apply to the worker threads but also to the number of requests in that level that can be in the “Ready Pool” at any point in time. The protocols for the CMLS implementation allow a worker thread or a “Ready Pool” entry to be used from the requests level or from a resource that is available from any higher level.

Supposing that there is an application that will execute with 10 worker threads in the thread pool and 100 entries in the “Ready Pool”, the application has configured the CMLS limits as Lane 0 is set to 0% (i.e. we will not be executing any of these requests, Lane 1 is set to 20% and Lane 2 is also set to 20%. In this example there could be a maximum of 20 Lane 1 requests in the ready queue at any point in time and there could be a maximum of 2 Lane 1 requests executing concurrently. There could also be a maximum of 40 Lane 2 requests in the “Ready Pool” at any point in time, assuming that there were no Lane 1 requests in the pool at that time and there could be a maximum of 4 Lane 2 requests executing concurrently, also assuming that no Lane 1 requests were executing at that time. Lane 3 requests are always unconstrained and can occupy all of the available slots in the “Ready Pool” and can be concurrently executing requests in every thread in the thread pool.

It should be noted that the priority mechanisms described in the previous section remain in effect when the CMLS scheduling mode is engaged.

The original analogy used in the design of the CMLS facility was to view the worker threads as separate lanes on a motorway and to view the different Levels as vehicle types with 0 being large articulated lorries, 1 being lorries, 2 being vans and 3 being cars and motorbikes. Signals above the motorway restrict vehicle types to only using assigned lanes. When entering the motorway, if the assigned lanes for your vehicle type are full then you have to wait. The analogy can still be useful but does introduce some false assumptions about how CMLS works. The main failing is that threads are not assigned to handle particular CMLS levels, individual threads can be used for any request but CMLS will prevent the total current work profile from exceeding any of the prescribed constraints.

4.6 Error Detection, Localisation and Percolation

Error handling in mullti-threaded applications can be complex if not clearly thought through in the application design. While not prescriptive the following pattern is suggested for applications that are implementing the Domino eXplorer multi-threaded kernel and functional objects.

Top-Level requests are issued a “permit to execute” this is normally implemented as a BOOL and carried in the top-level request object. Sub-requests all carry the address of the top-level request, before starting the execution of any sub-request or issuing a new sub-request the “permit to execute” is checked and if the permit has been revoked then a sub-request is not executed but is marked as failed, new requests are not dispatched and the object destroyed.

While processing a sub-request any error or out-of-envelope condition should be detected and reported as soon as possible. The processing of the sub-request must determine if it will fix the error or tolerate the error in which case it continues to process. If the processing of the sub-request determines that it is unable to recover or tolerate an error then it should revoke the permit to execute in the top level request by setting it to FALSE.

All processing paths that have issued sub-requests should recover them from the rejoin pool, even if the permit to execute has been revoked and marking the parent request as having failed. This paradigm will result in the percolation of the error, eventually to the top-level request.

4.7 The Design of Runnable Classes

Although there are no real constraints imposed by the kernel for the design of Runnable objects apart from the fact that they need to inherit from the “Runnable” class and implement the “ExecuteThisRequest” interface, there are a few simple guidelines that should be followed to ensure a successful implementation pattern.

There should never be a need to instantiate more than a single instance of any runnable class, no matter how many threads are being run in the thread pool. Some developers assume that there is some kind of affinity between the Runnable object and a particular thread in the pool, this is not the case there is no such affinity.

Any variable data used in processing a request should only be held in either local automatic storage or in “Transaction Storage” i.e. members in the request object. These variables should **NEVER** be stored in members in the Runnable object. The selection of Automatic or Transaction storage is determined by the lifecycle of the data in the variable. If the data is to be used across multiple asynchronous request dispatches then the variable should be stored in Transaction storage, if the data is only to be used for the processing of a single request then it is probably more appropriate to use Automatic storage.

The kernel does not have any provision for “Thread Local Storage” i.e. memory that is reserved for use by a single thread. The kernel does however provide API functions for using one resource, database handles, on a per thread basis. All other Domino resources can be used from multiple threads, compiled formulas can only be used by one thread at a time but are more appropriately handled in Transaction storage rather than dedicated to a particular thread or by serialising access to the compiled formula.

4.8 Thread Synchronisation

This section examines a number of aspects of Thread Synchronisation, specifically how they apply to DX applications and the DX kernel.

4.8.1 What Is It and Why Do We Need It?

Thread Synchronisation is the implementation and use of mechanisms in an application that permit safe serial access to resources in a multi-threaded application. Failure to serialise access to some application resources, usually data structures, can result in a failure mode referred to as a “Data Race” or “Race Condition” in this failure mode lack of data coherence between the code in different threads of an application results in an indeterminate state of the application.

The following pseudo-code example will illustrate a “Race Condition” and the resulting coherence failure in part of an application.

The area of the application that we will examine is designed to create a forward linked list of objects (“Things”), the analysis will look at the execution of this process by two threads. In the example code the HeadOfList variable (Thing *) is a globally addressable variable that is initialised to NULL and the “Thing” class implements a member called “NextThing” that is the foward pointer that forms the linked list also initialised to NULL on creation of an instance of the class, all other variables can be assumed to be local automatic variables within the scope of the code being examined.

The Code.

```
MyThing = new Thing();          //  Create a new Thing object

//  Maintain the linked list
//  Determine if the head of the list has already been allocated
If (HeadOfList == NULL) HeadOfList = MyThing;      //  Add the object as the head of
the list
Else
{
        //  Locate the last object on the chain
        ExistingThing = HeadOfList;
        While (ExistingThing->NextThing != NULL) ExistingThing = ExistingThing-
>NextThing;
```

```
        //  Add the new object to the end of the chain
        ExistingThing->NextThing = MyThing;
}
```

We will now look at two instances of execution by two threads (**RED** and **BLUE**) each one will add two objects to the linked list, in the first instance the execution is successful and in the second the execution fails due to a race condition.

An Example: Successful Execution

```
MyThing = new Thing();    // #1
If (HeadOfList == NULL)   // true
HeadOfList = MyThing;
MyThing = new Thing();    // #2
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
MyThing = new Thing();    // #3
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
MyThing = new Thing();    // #4
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
```

The linked list has been correctly formed as shown below.

```
HeadOfList -> #1 -> #2 -> #3 -> #4 -> NULL
```

An Example: Failure – Race Condition

```
MyThing = new Thing();    // #1
If (HeadOfList == NULL)   // true
HeadOfList = MyThing;
```

```
MyThing = new Thing();    // #2
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
MyThing = new Thing();    // #3
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
MyThing = new Thing();    // #4
If (HeadOfList == NULL)   // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
ExistingThing->NextThing = MyThing;
```

The linked list is now incorrectly formed as shown below.

`HeadOfList -> #1 -> #2 -> #3 -> NULL`

The `#4` object is now orphaned, also to make matters worse in this case there is nothing obviously wrong with the execution, the error would not be detectable unless we checked for memory leaks at program termination or there were other checks and balances that would surface the problem then it could go unnoticed and simply result in incorrect output. The example above only illustrates a single failure mode that results from a race condition, there are many other modes that can have mire catastrophic results such as use-after-free, loops, data overwrites and many more.

4.8.2 Fix the Issue with a Lock

One approach to fixing the issue is to introduce a lock (semaphore, mutex, critical section) this mechanism introduces a state of the vulnerable data “locked” where it can only be accessed by the thread that currently “owns” the lock. This would then provide the means for us to serialise access to the forward chain of the linked list and through that enforce coherence of the data. For the moment we will treat the “lock” as an opaque mechanism, we will look at the implementation later.

We introduce two new constructs to the code a LockTheList() and UnlockTheList() call, the first asserts ownership of the linked list and blocks until the lock can be asserted the second releases ownership of the lock.

The Code.

```
MyThing = new Thing();          //  Create a new Thing object

//  Maintain the linked list
//  Acquire The Lock
LockTheList();
//  Determine if the head of the list has already been allocated
```

```
If (HeadOfList == NULL) HeadOfList = MyThing;      //  Add the object as the head of
the list
Else
{
       //  Locate the last object on the chain
       ExistingThing = HeadOfList;
       While (ExistingThing->NextThing != NULL) ExistingThing = ExistingThing-
>NextThing;
       //  Add the new object to the end of the chain
       ExistingThing->NextThing = MyThing;
}
//  Release The Lock
UnlockTheList();
```

We now look at the failing execution sequence again with the additional code in place. The code change that we have introduced here is a coarse-grained lock that is a lock that ensures safety of the resource it is protecting with a wide ranging safety net that has a quite high risk of contention. There are other more fine-grained implementations that would involve some code changes that would reduce the contention.

An Example: Success – Race Condition is Avoided

```
MyThing = new Thing();   // #1
LockTheList();  //  Locked
If (HeadOfList == NULL)  // true
HeadOfList = MyThing;
UnlockTheList(); //  Unlocked
MyThing = new Thing();   // #2
LockTheList();  //  Locked
If (HeadOfList == NULL)  // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
UnlockTheList(); //  Unlocked
MyThing = new Thing();   // #3
LockTheList();  //  Locked
If (HeadOfList == NULL)  // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
MyThing = new Thing();   // #4
LockTheList();  //  List is still locked – wait for it
ExistingThing->NextThing = MyThing;
UnlockTheList(); //  Unlocked
LockTheList();  //  Locked
If (HeadOfList == NULL)  // false
ExistingThing = HeadOfList;
While (ExistingThing->NextThing != NULL)
```

```
ExistingThing = ExistingThing->NextThing;
While (ExistingThing->NextThing != NULL)
ExistingThing->NextThing = MyThing;
UnlockTheList(); //  Unlocked
```

The linked list has been correctly formed as shown below.

```
HeadOfList -> #1 -> #2 -> #3 -> #4 -> NULL
```

4.8.3 Implementing Locks

We will now look at the implementation of the locking mechanism that we introduced to fix the race condition in our code example.

The first implementation to be considered is a lightweight implementation that uses a shared memory variable to communicate the lock state between the different threads.

Locking Code.

```
Volatile BOOL ListLock;           // Shared memory Lock Variable
.
.
void LockTheList()
{
While(ListLock == TRUE) Wait;     //  Wait for the lock to become free
ListLock = TRUE;
Return;
}

Void UnlockTheList()
{
ListLock = FALSE;
Return;
}
```

The code above raises a few important question, first of all what is that volatile keyword doing on the declaration of the shared memory lock variable? The volatile keyword tells the compiler to always read the value of the variable from memory and never cache it's value or an intermediate value that depends on the variable in registers. This is not an option that defeats optimisation it modifies the core "register assignment" function in the compiler for all references to any variable that is defined with the keyword.

The second question is what does this "Wait" thing do? There are two types of locks that can be implemented they differ in their implementation of the "Wait" process. The first type are Spin locks, this type of lock does nothing for the "Wait" it merely loops back to check the value of the lock again, hence it just keeps spinning the CPU. Spin locks are designed to be used in situations that experience very short state transitions such as within the operating system or in real-time systems. The alternative implementation pattern to the Spin lock is the Yield lock. The Yield lock implements a mechanism for suspending the operation of the thread at the point where "Wait" is needed and allows the operating system to immediately start executing a different thread.

The more eagle-eyed reader will have already noticed a rather more important question, do we not have the same possibility of a race condition in the code that asserts the lock? Indeed we do, the following execution map will show the race with two threads (**RED** and **BLUE**).

```
ListLock = FALSE;           // Initial State is unlocked
```

```
While(ListLock == TRUE)

While(ListLock == TRUE)

ListLock = TRUE;

Return;
ListLock = TRUE;

Return;
```

Both threads now continue executing in the belief that they hold the lock exclusively which is not the contract that we have relied on to make the linked list processing safe and we are back to having the potential for the same race condition that we identified initially.

The core of the problem is that the test of the ListLock variable (while(ListLock == TRUE)) that determines that the variable is in a suitable state for it to be changed (i.e. FALSE) and the following instruction setting it to be TRUE (ListLock = TRUE) are not “atomic” the execution of the instructions can be interrupted between the test and the set instruction which opens up the possibility of a race condition. The examples here illustrate the problem at the level of “C” code, the problem becomes much more of an issue when we look at the level of the machine instructions that the compiler generates and we throw into the mix multi-processors and multicore processors and CPU architectures that allow out-of-order execution of instructions along with memory caching. Modern CPUs provide primitive operations at the hardware level that allow the correct implementation of true “atomic” operations that include the “Test and Set” primitive used in our flawed example. Developers would not normally introduce code that directly implement these hardware methods to achieve true atomicity instead they would rely on higher level functions provided by the language, third-party libraries or the operating system to implement heavyweight locks and counters etc. These heavyweight implementations will themselves use lower level functions that eventually rely on using the hardware methods to achieve correct atomic operations.

So we could fix our flawed example by for instance, on the Windows platform creation of an OS Mutex (CreateMutex) to use as the lock and then using the OS provided assertion (WaitForSingleObject) and release (ReleaseMutex) functions to enforce the correct contract for the lock mechanism suggested in the example. However, there is no such thing as a free lunch, the use of these heavyweight synchronisation mechanisms have a cost. Even at the hardware layer there are several Intel papers that indicate that the penalties of using the mechanisms can have significant impacts on system performance and behaviour, and suggesting that where appropriate lightweight alternatives can be correctly implemented then those are preferred. Indeed use of the heavyweight locking mechanisms may display multiple wait states and side effects that can give rise to “emergent behaviours” that make it extremely difficult to tune and optimise some multi-threaded applications.

4.8.4 Alternatives to Heavyweight Locking

In this section we will examine some alternatives to using heavyweight locking mechanisms to achieve thread synchronisation.

4.8.4.1 Option #1: Avoid Synchronisation

The very best option is to design your application so that it is race free i.e. does not require the use of any of the synchronisation mechanisms that are vulnerable to race conditions. We will examine one such example implementation for our flawed example.

We change the structure of our application so that there is an additional thread (**PURPLE**) that is responsible for the maintenance of the linked list, the **RED** and **BLUE** threads remain responsible for the creation of Things that will be assembled into the forward linked list. The only thing that we have to be careful about is to make sure that the mechanism that we implement to communicate the availability of a new Thing to be added to the list is not vulnerable to race conditions.

New Code Example.

```
#define RED 0

#define BLUE 1
```

```
Thing * HeadOfList = NULL;                  // Head of the linked List
Volatile Thing * NewThings[2] = {NULL, NULL};     // Per Thread Array of new Things


MyThing = new Thing();              //  Create a new Thing object
// Post the new object to the list maintainer thread
While(NewThings[RED or BLUE} != NULL) Wait;    //  Wait for available to post
NewThings[MyThread} = MyThing;    // Post the request
Loop;                               // Process all Things


// List Maintenance
If (NewThing[0] != NULL)
{
NextThing = NewThing[RED];
NewThing[RED] = NULL;
}
Else
{
NextThing = NewThing[BLUE];
NewThing[BLUE] = NULL;
}
//  Maintain the linked list
//  Determine if the head of the list has already been allocated
If (HeadOfList == NULL) HeadOfList = NextThing;      //  Add the object as the head of
the list
Else
{
       //  Locate the last object on the chain
       ExistingThing = HeadOfList;
       While (ExistingThing->NextThing != NULL) ExistingThing = ExistingThing-
>NextThing;
       //  Add the new object to the end of the chain
       ExistingThing->NextThing = NextThing;
}
Loop;                               // Process all Things
```

The first section of code would be executed continuously by the **RED** and **BLUE** threads until there are no more Things to be constructed. The second section of code is continuously executed by the **PURPLE** thread until some control signal (not shown) indicates that there are no more Things to be processed.

At first glance it might be assumed that we have a race vulnerability on the NewThing array, however that is not the case it is perfectly safe. Only the **RED** thread can set the [0] element from NULL to a non-NULL value and only the **PURPLE** thread can set the [0] element from a non-NULL value to NULL. This bistable implementation uses a form of Thread Local Storage (TLS) for the communication between threads, this is simple to achieve requiring only that the code executing in a thread needs to be aware of which thread it is executing in, which is a trivial problem.

While being safe the implementation above may not show optimal performance, if the average processing time of creating a Thing in the **RED** and **BLUE** threads is much shorter than the time-slice that each thread gets and is shorter than the average time taken for the **PURPLE** thread to maintain the linked list then there can be an awful lot of waiting time in the system. This can be addressed be having the **RED** and **BLUE** threads create short chains of Things and then post the address of the first Thing on the short chain for addition to the master chain by the **PURPLE** thread.

The NewThing array provides a per-thread semaphore that is used to co-ordinate the producer and consumer threads. This use of the term semaphore extends the railway analogy from which it was initially derive to encompass the naval analogy where a semaphore (flag) was used to signal between ships. Note that the **RED** and **BLUE** threads do not have to wait (block) after they have posted their Things to the **PURPLE** thread for adding to the linked list, they only need to wait (block) if they come to post a new Thing and the **PURPLE** thread has not yet captured and cleared their respective semaphore.

One of the nice things about this kind of model is that we can easily collect profiling data and use it to adjust the number of producer threads and the multiplexing of Things that are posted to the consumer thread. This type of mechanism is used for different purposes, where it is appropriate, in the DX kernel, the eventual intention in the DX implementation is to make the profiling and tuning adjustments autonomic i.e. self-adjusting.

The example presented here is only one possible way of avoiding the thread synchronisation issues that can lead to race conditions, designers and developers should look carefully at proposed implementation patterns to develop appropriate solutions to individual cases.

4.8.4.2 Option #2: Tolerating Races

Another possible approach to avoiding the use of heavyweight locks to achieve thread synchronisation is to use lightweight locks, with the knowledge that they can sometime fail and tolerate the failures or resulting race conditions. This may be possible and even appropriate in certain circumstances.

The first thing that should be considered is the probability of a failure in a particular implementation and workload which will therefore determine the likely frequency of the race condition. If the threads are executing pieces of work that have variable size and are large enough that multiple time slices are required for each piece of work then the probability of a race failure is reduced. Experiments have shown that lightly loaded servers executing a lightly loaded application have an increased probability of seeing a collision while using lightweight lock mechanisms. The second aspect to consider is what is the effect of a failure? If the failure can be detected before any lack of application coherence occurs then there is no danger because the failing operation can be backed out and re-driven when the collision is detected. If the application fails catastrophically when a collision occurs then that may also be safe providing that it happens only infrequently, an application re-run once a month may be a good trade-off against a loss of 10% of throughput through the use of safer synchronisation mechanisms.

The DX implementation of the lightweight lock/semaphore/mutex mechanism is a little more sophisticated than the simple example introduced earlier. The following code shows the implementation used in the ObjectCache object used by some DX Tools.

```
volatile UINT CacheMutex;          //  Cache Mutex


void ObjectCache::lockTheCache(int iThreadID)
{
        DWORD           dwCWaits = 0;        //  Wait count
        DWORD           dwCWaitQuanta = 0;   //  Wait Qanta
        while (CacheMutex != iThreadID)
        {
                while (CacheMutex != CACHE_MUTEX_FREE)
                {
                        dwCWaits = 1;
                        dwCWaitQuanta++;
                        Sleep(MUTEX_WAIT_MILLIS);
                }
                CacheMutex = iThreadID;
        }
        dwWaits += dwCWaits;
        dwWaitQuanta += dwCWaitQuanta;
        return;
}


void ObjectCache::unlockTheCache(int iThreadID)
{
        if (CacheMutex == iThreadID) CacheMutex = CACHE_MUTEX_FREE;
        return;
}
```

The advantage of this design is that it allows a sequence of code to acquire the mutex and later in the execution path call the lock function again which will detect a collision and force an additional wait state before re-applying that mutex. The mechanism can still fail however the failure rate is very small and applications that use the cache can check any object that is returned from the cache and ensure that it was the object that was requested, if it does not get the correct object it can re-drive the request.

4.8.4.3 Option #3: Lightweight Mutexes with Atomic Locks

An alternative is provided by using the lightweight lock construct shown in option 2 but fixing the race conditions on the lock by using Atomic operations to assert the lock (acquire the mutex). This is the approach taken by the DX Kernel in situations where there is no lock free (option 1) solution available. The kernel provides a set of APIs for implementing these mutexes. However race tolerant processing remains a valid option to be applied in situations where it is warranted.

4.9 Request Sizing

The DX threading model has been designed to handle large workload tasks with heavy I/O requirements (network and disk), high memory occupancy and moderate CPU processing, we use the term “Heavy Lift Computing” (HLC) for these types of workload. The model is absolutely **NOT** suitable for the implementation of “High Performance Computing” (HPC) applications.

To ensure that applications fit the “Heavy Lift” paradigm it is important to design the lowest level sub-requests used in the application so that they do not contain too small a quantum of the total workload. There are no definitive rules to determine what is the optimal size and characteristics of the lowest level sub-requests, determining this is a part of the application tuning process. The most successful approach has been to identity the smallest sensible unit of processing at the lowest level of functional decomposition and then to make the lowest level sub-request capable of processing a variable number of these base functional quanta. Tuning of the application consist of changing the number of threads in the pool and varying the number of base functional quanta in the lowest level sub-requests, alongside eliminating bottlenecks and resource contention.

The “Database Copier” (DbCopier) engine implements a good method for dealing with the sizing of the lowest level requests. The functional quantum in the copier is a request to copy a single note from the source database to the target database, the engine determines a value for how many quanta will be combined into a sub-request by computation using size of the source database and the number of documents to be copied. Databases that have many small documents will dispatch sub-requests with more document copy operations than when copying databases with fewer larger documents. The Database Copier also implements a mechanism for scaling the copy operations per request count by a specified factor, this allows for rapid tuning of an implementation.

It has also been noted that if the functional quantum in an application has long wait times associated with it, such as disk I/O to very slow devices or more usually network I/O over “long fat pipes” then these benefit, from running more threads with a smaller size of sub-request.

5. A Multi-Threaded Server Add-In Task

This section examines the design and coding of a multi-threaded Server Add-In task. The sample application examined is the QCopy application.

```
Q(ueue)Copy


This utility server add-in task will make replica or non-replica copies of
notes databases across networks at light speed.The utility uses multi-
threading for the database I/O and can run up to 100 threads at the
same time. The databases to be copied and the parameters to use for each copy
operation are read from a queue in a control database.

USAGE:

Load QCopy ControlDatabase [-V|-T[:Area]|-D[:Area]] [-E][-M:nnn] [-X:nn][=IniFilename]

[ControlDatabase]   - The name of the control database, relative to the server's Data
Directory
[-V]|[-T[:Area]]|[-D[:Area]] - Set logging level to Verbose, Trace (optionall the area
to trace) or Debug.
-M:nnn  - Optional - Switch that sets the number of threads to use in each copy
operation
-X:nn  - Optional - Switch that determines the max number of concurrent transactions.
-E     - Optional - Switch that echoes the logging messages to the server console
```

5.1 Task 1: Populate the RunSettings Object

The normal method of populating the RunSettings object is to extend the RunSettings class with a custom (AppRunSettings) class that will derive application settings that may change from run to run and populate those at the same time as populating the base RunSettings members. This processing is not compulsory so long as this phase yields a correctly populated RunSettings object.

The command line arguments are passed to the constructor of the AppRunSettings class, the constructor will set any default values and parse the command line arguments to populate members in the base and extending class.

Some of the parameters define the configuration of the application, these would be set as default values in the application code. The QCopy application is a Server Add-In task that uses a repository database and can run multiple instances on the same server. The following default settings are made in the AppRunSettings class to specify this configuration.

```
RunningAsAddin = TRUE;            //  Running as server Addin Task
AllowMultipleAddins = TRUE;       //  Allow multiple addins
NoRepository = FALSE;             //  Allow use of the repository
NoAppLog = FALSE;                 //  Allow Application Event Logging
EchoLog = FALSE;                  //  Do not Echo to the console
NeedsMQ = TRUE;                   //  Processor needs an MQ
CreateRepository = FALSE;  //  Do not Create the repository
CreateOnDemand = FALSE;           //  Do not ceate on demand
```

Once the AppRunSettings object is created the mainline code should check two switches in the object to determine if the application should proceed.

```
//  Validate and build the Run Settings for this execution
arsLocal = new AppRunSettings(argc, argv);
if (!arsLocal->AllowExecution)    //  Exit silently if just showing usage
{
       delete arsLocal;
       return APPRC_NOERROR;
}

if (!arsLocal->IsValid)                  //  Abort if validation failed
{
       std::cout << MSG_QCP0101S << std::endl;
       delete arsLocal;
       return APPRC_FATAL;
}
```

The AllowExecution flag is set to FALSE if the parameters were valid but indicated that the switches on the command line (-?) indicated that the application usage messages should be shown and no execution atrtempted. In this case the application just silently terminates, the console will show the application usage messages.

The IsValid is set to FALSE if the application parameters were invalid or any other condition prevented the valid instantiation of the AppRunSettings object. The application terminates with an error message showing that the application could not be started.

Populating the RunSettings object is completed by setting the application name, short title and version in the appropriate members.

```
//  Set the identification in the Run Settings
strcpy_s(arsLocal->APPName, MAXAPPNAME, APP_NAME);
strcpy_s(arsLocal->APPTitle, MAXAPPTITLE, APP_TITLE);
strcpy_s(arsLocal->APPVer, MAXAPPVERSION, APP_VERSION);
```

The APP_NAME, APP_TITLE and APP_VERSION symbolic values are defined in the application header file.

It should be noted that there is not a permanent application log available at this stage of processing, even if the application intended to use one so all output is directed to STDOUT.

5.2 Task 2: Set the Thread Manager Policies

An object of the ThreadManagerPolicy class contains information used by the thread manager to configure the multi-threaded runtime system. An application can configure an object of this class and use it in the creation of the runtime system to influence many settings and constraints that are used by the runtime system. Only members that should be set by the application are described here, other members of the class are intended for internal use by the kernel.

Refer to the “KernelAPI Reference” document for details of the different settings that can be made in the ThreadManagerPolicy class.

```
//  Create and initialise a Thread Manager Policy
tmpInit = new ThreadManagerPolicy();
tmpInit->TPoolPolicy = TPOOL_POLICY_OBEYMAXMIN | TPOOL_POLICY_PRESTARTTARGET;
tmpInit->MaxThreads = arsLocal->ThreadCount;   //  Max Thread Count
tmpInit->MinThreads = arsLocal->ThreadCount;   //  Min Thread Count
tmpInit->TargetThreads = arsLocal->ThreadCount;       //  Initial target Thread Count
//  Pending RQE Pool size
tmpInit->PendingRQECapacity = PCAP_PERTHREAD * arsLocal->ThreadCount;
//  Rejoin RQE Pool size
tmpInit->RejoinRQECapacity = RCAP_PERTHREAD * arsLocal->ThreadCount;
tmpInit->AsyLogPoolEntries = 200; //  Log pool size is 200 entries
```

```
//  Set the priority policy
tmpInit->PriorityPolicy = PRIO_POLICY_AGERQS | PRIO_POLICY_PREFBOOST;

//  Set CMLS Mode and the CMLS contraints
tmpInit->TPSchedMode = TPOOL_MODE_CMLS;
tmpInit->MaxPctL1Threads = 20;    //  20% allocated to Feeders
tmpInit->MaxPctL2Threads = 20;    //  20% allocated to UOWs
```

The count of threads is determined from the RunSettings which sets a default value that can be overridden by a command line argument.
The PCAP_PERTHREAD and RCAP_PERTHREAD symbolic values allocate capacity in the pools to 20 requests per thread.

See the previous chapter for a discussion of Constrained Multi-Lane Scheduling (CMLS).

5.3 Task 3: Initialising the Run Time

The AppRunSettings and ThreadmanagerPolicy objects are passed to the constructor of the multi-threaded run time (MTExecutive).

Initialising the Run Time will accomplish the following tasks.

- Initialise the memory pools and threads that provide the multi-threaded kernel
- Initialise the Thread Pool with the requested number of worker threads
- Implement the threading policies
- Initialise the Notes Runtime
- If the application is using a repository database this will be opened and the DBHANDLE made available
- Logging will be initialised and directed to the appropriate destination(s)
- A default elapsed timer will be initialised

After the run time is initialised the IsInitialised and MTFunctionsAvailable flags are checked to make sure that all of the expected kernel functionality is now available.

```
//  Initialise the runtime environment
xeLocal = new MTExecutive(arsLocal, tmpInit, argc, argv);
if (!xeLocal->IsInitialised)      //  Abort if runtime failed to initialise
{
       std::cout << MSG_QCP0101S << std::endl;
       delete tmpInit;
       delete xeLocal;
       delete arsLocal;
       return APPRC_FATAL;
}

//  Also check the the Thread Manager was initialised properly
if (!xeLocal->MTFunctionsAvailable)
{
       std::cout << MSG_QCP0101S << std::endl;
       delete tmpInit;
       delete xeLocal;
       delete arsLocal;
```

```
        return APPRC_FATAL;
}
```

5.4 Task 4: Construct the Processing Engines

The QCopy application uses two different engines to process transactions. The first engine is the QCopier that performs copying of single databases. The second engine is the QCFeeder, this performs the task of mapping transactions for multiple database collections into single database copy transactions that can be processed by the QCopier.

```
//  Construct the QCopier copy engine
qcLocal = new QCopier(xeLocal, 0);
//  Set the multiplexor scaling factor
qcLocal->MultiplexScale = arsLocal->MultiplexScale;

//  Construct the Feeder Transaction processor
qfLocal = new QCFeeder(xeLocal, 0);
```

5.4.1 The DbCopier Engine

The DbCopier is a typical example of the implementation of a major processing component in a DX application, although it does implement a wider variety of processing options that would be encountered in a less generic processor.

The engine can be invoked synchronously or asynchronously to execute a top-level request to copy a single database, it creates multiple lower level requests to be executed asynchronously by itself to achieve the requested copy operation.

The first phase of processing of a top-level request constructs the target database and copies all of the data and design notes from the source database to the target database. After constructing the target database the DbCopier posts two asynchronous requests for execution. The first request is to copy all of the design notes in the source database and the second is to copy all of the data notes in the source database. The copy of the design notes process will build a table of all of the design notes in the source database and add to it any profile documents in the source database it then generates a number of asynchronous requests to copy a small subset of all the design notes. The number of notes that will be copied by each request is determined dynamically and is set between 5 and 100 notes per request. The copy data notes process is almost identical to the copy design process, it creates a table of all of the data notes in the source database and creates asynchronous requests to copy a small subset of these notes, again between 5 and 100 notes are copied per request. The copy notes process copies the number of notes provided in the request, if a note being copied is found to be a Folder Design note then the ID of the note is added to a table for processing in a later phase of execution. If the copying of an individual note fails then depending on the type of failure it may be re-tried by the DbCopier copy notes process or the failures may be tolerated depending on the failure mode and the fault tolerance settings that are in effect. If any errors are not fixed by a retry or tolerated then the copy notes process will signal that further copying should be quenched by withdrawing the Run Permit in the top-level request.

The DbCopier will wait until all of the copy requests have completed successfully before moving on to the next phase of processing.

The next phase of processing will perform any Folder maintenance and Full Text Index (FTI) maintenance that are needed. Assuming that both Folder and FTI maintenance are needed then one asynchronous request is created for each. The Folder maintenance request will use the table of Folder design notes that was created earlier to generate asynchronous request each for the maintenance of an individual folder.

The DbCopier will wait until all of the Folder and FTI maintenance requests are completed before moving on to the next phase of processing.

The following phases of the DbCopier processing are optional and are performed in sequence.

- Refresh or Replace the design of the target database.
- Run a specified “conditioning” agent in the target database.
- Build any views in the target database.
- Perform a one-time replication with the source database.

5.4.2 The QCFeeder Engine

The QCFeeder engine asynchronously processes transactions that specify copying collections of databases, it creates an individual copy transaction for each database in the collection and serialises these transactions to the ready queue in the repository.

The QCFeeder creates a Domino eXplorer singleton object that it will use to expand collections to single databases and a QCFeedProcessor singleton object that implements the DXReporter interface and does the generation of individual copy transactions for each database discovered by the Domino eXplorer.

Each transaction that is received by the QCFeeder is passed to the Domino eXplorer to scan. The eXplorer invokes the ReportOnThisDatabase interface implemented by the QCFeedProcessor class for each database that is found during the scan. The ReportOnThisDatabase interface will use information from the original “Feeder” transaction and the name of the source database found by the eXplorer to construct a new QCopy transaction which is then written to the transaction queue in the repository ready to be executed.

5.5 Task 5: Construct the Transaction Handler and Queue

The transaction handler is the core part of the application responsible for pumping work into the application form the repository database transaction queues and for pumping the results of processing back to the transactions in the repository.

The base TransactionHandler class is extended to form the AppTransactionHandler class, the extensions provide the transaction handler to interact with the physical implementation of the queues in the repository database.

```
//  Construct the Application Transaction Handler
thLocal = new AppTransactionHandler(xeLocal, 0);
thLocal->qcMaster = qcLocal;      //  Set the copy engine in the transaction handler
thLocal->qfMaster = qfLocal;      //  Set the feeder engine in the transaction handler
```

The AppTransactionHandler code is also responsible for determining if transactions are to be executed by the copy engine (QCopier) or they are “feeder” transactions and should be executed by the feeder transaction processor (QCFeeder), hence the address of the two engines need to be set in the AppTransactionHandler.

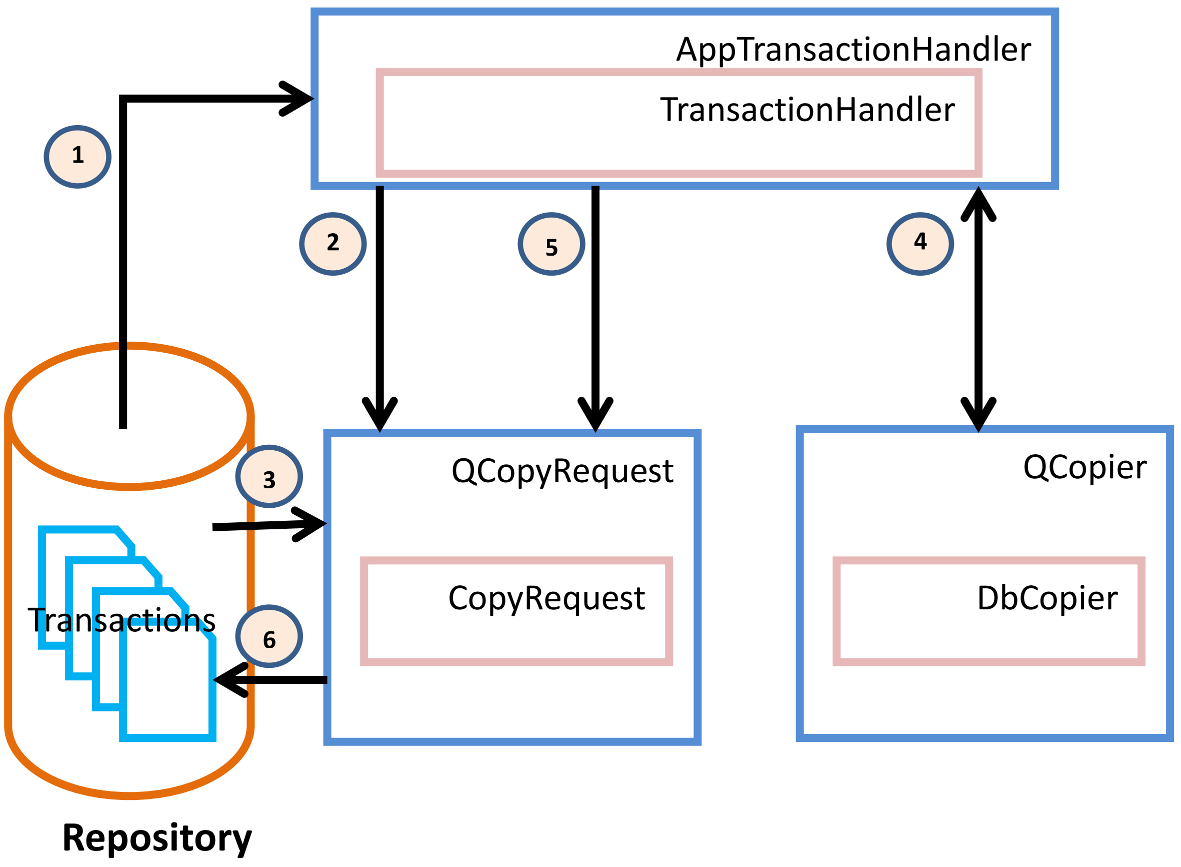
A TransactionQueue object describes the physical implementation of the transaction queues in the repository database. Later in the applicatioin the object is posted to the multi-threaded kernel for aynchronous execution by the transaction handler.

```
//  Now construct a new Transaction Queue descriptor for the transaction queue
tqQCopy = new TransactionQueue();
//  Set the D/B handle of the repository as the queue database
tqQCopy->hdbQueue = xeLocal->hdbRepository;
//  Set the Queue Name
strcpy_s(tqQCopy->szQName, MAX_ELEMENT, QCQNAME);
//  Define the views to use for the sub-queues
strcpy_s(tqQCopy->szReadyQName, MAX_ELEMENT, QUEUE_NEW_TRANS_VIEW);
strcpy_s(tqQCopy->szInProgressQName, MAX_ELEMENT, QUEUE_IP_TRANS_VIEW);
strcpy_s(tqQCopy->szDelayedQName, MAX_ELEMENT, QUEUE_DELAYED_TRANS_VIEW);
strcpy_s(tqQCopy->szSchedQName, MAX_ELEMENT, QUEUE_SCHED_TRANS_VIEW);
```

```
//  Set the options that control the queue behaviours
tqQCopy->wQueueProtocols = QPFLAG_DEFAULT | QPFLAG_MONITOR_SCHED;  //  Default
Behaviour for startup & delay
tqQCopy->MaxConcurrent = arsLocal->MaxCTX;     //  Set initial Max Concurrency
tqQCopy->MinDelay = 15 * 60;             //  Delay Queue for minimum of 15 mins
tqQCopy->DelayCycleSecs = 10;            //  10 seconds per dispatch loop
```

5.6 Transaction Flow

The following diagram shows the normal flow path for transaction execution.



1

The transaction handler detects that there are transactions available for processing in the queues in the repository database.

2

The transaction handler creates a new QCopyRequest.

3

The QCopyRequest populates itself with the information from the transaction document detected by the transaction handler.

4

The transaction handler posts the request for asynchronous execution and monitors for completion of the request.

5

The transaction handler notifies the QCopyRequest that it should serialise the processing results back to the transaction document.

6

The QCopyRequest writes processing status, transaction log information and statistics back to the transaction document in the repository database.

The QCopier class extends the DbCopier class to implement the ProxyLogMessage interface this allows it to capture log messages from the DbCopier and record the messages in the individual requests so that they can later be written back to the appropriate transaction document.

5.7 Task 6: Construct the Application Command Handler

The AppCommandHandler class extends the base CommandHandler class to provide additional commands that are application specific or to modify or extend commands that are implemented by the standard command handler. In the case of the QCopy application it extends the STATUS command and the ABORT, STOP, SUSPEND and RESUME commands and implements a MAXTRANS command that allows the number of transactions that can be executed simultaneously to be varied.

```
//  Contruct the Application Command Handler and attach it to the Thread Monitor
achLocal = new AppCommandHandler(xeLocal);
achLocal->tqCurrent = tqQCopy;    //  Set the Queue address in the handler
achLocal->thCurrent = thLocal;    //  Set the Queue handler address
xeLocal->AttachCommandHandler(achLocal);
```

The final statement above activates the application command handler, from this point on it will respond to any commands detected on the message queue.

Multiple instances of the QCopy Server Add-In Task are allowed to run on the same server at the same time. The name of the Message Queue (MQ) used for each instance is determined dynamically during start up. The first or only instance will have a queue named “QCOPY1” the second “QCOPY2” and so on.

5.8 Task 7: Initiate Transaction Processing

The TransactionQueue that was constructed earlier is now posted to the multi-threaded kernel for asynchronous execution by the transaction handler. The transaction handler will continue to monitor the transaction queues for any work to do, it continues running until it is told to stop, usually in response to a signal from the main application code path.

```
//  Dispatch the Transaction Queue for execution
xeLocal->PostARequest(PXR_WAITIF_BUSY | PXR_APP_WAIT,
                    achLocal,
                    thLocal,
                    tqQCopy,
                    RQATTR_REJOIN | RQATTR_CMLSL0,
                    10,
                    0);
```

The QCopy application sets the multi-threaded kernel to run in Constrained Multi-Lane Scheduling mode, the transaction to run the transaction handler is flagged as a Level (Lane) 0 transaction i.e. a service transaction that will run until the application shuts down (RQATTR_CMLSL0).

From this point on the application will execute any transactions that appear on the Ready Queue in the repository database.

5.9 Task 8: Monitor for Application Completion

The main line of the application code should now sit in a loop looking for signals raised by the command handler that indicate that the application should shut down.

```
//  Monitor the command handler and respond to any requested changes in state
bProcessorShouldQuit = FALSE;
AddInSetStatusText(ADDIN_STATUS_RUNNING);
while (!bProcessorShouldQuit)
{
//  Check if any terminal condition signals have been posted in the Command Handler
        if (achLocal->State & CH_STATE_QUIT)
        {
                //  Detach the command handler
                xeLocal->AttachCommandHandler(NULL);
                AddInSetStatusText(ADDIN_STATUS_TERM);
                xeLocal->LogMessage(MSG_QCP0107I);
                AddInLogMessageText(MSG_QCP0107I, NOERROR);
                bProcessorShouldQuit = TRUE;
        }
        else
        {
                if (achLocal->State & (CH_STATE_STOP | CH_STATE_ABORT))
                {
                        //  Detach the command handler
                        xeLocal->AttachCommandHandler(NULL);
                        AddInSetStatusText(ADDIN_STATUS_TERM);
                        xeLocal->LogMessage(MSG_QCP0108I);
                        AddInLogMessageText(MSG_QCP0108I, NOERROR);
                        bProcessorShouldQuit = TRUE;
                }
                else xeLocal->AppWait(TIMEOUT_EVENT_WAIT);     //  Sleep
        }
}
```

Note that the only states that cause a response in the code are QUIT, STOP or ABORT any other state signals are ignored at this level and only result in the main application thread going to sleep.

5.10 Task 9: Terminate the Transaction Handler

After receiving a signal that the application should shut down the first thing to do is to signal to the transaction handler that it should drain and shut down.

```
//  Wait for the transaction queue to drain before shutting down
tqQCopy->LocalPermit = FALSE;
```

The permit in the transaction queue is monitored continuously by the transaction handler, as soon as it is revoked the handler will monitor for the completion of any transactions that are currently executing and once these have completed it will shut down. The main application thread should wait in a loop until the transaction queue rejoins the main processing.

```
while (iRejoinRC != RJR_NONE_EXIST)
{
        if (iRejoinRC != RJR_RETURNED) xeLocal->AppWait(PXR_BUSY_WAIT);
        iRejoinRC = xeLocal->GetRejoinRequest(achLocal, (void **) &tqReturned, 0);
}
```

5.11 Task 10: Clean up and Terminate the Application

Once the processing has been completed the application code should destroy the objects that have been created and then terminate the application. The run time and the associated RunSettings should be the last objects that are disposed of, this ensures that the logging interface is available for messages generated during the termination process.

```
//  Dispose of the inactive components
if (achLocal != NULL) delete achLocal;
if (tqQCopy != NULL) delete tqQCopy;
if (thLocal != NULL) delete thLocal;
if (qcLocal != NULL) delete qcLocal;
if (qfLocal != NULL) delete qfLocal;

//  Processing is completed - shut down, clean up and exit.
if (!xeLocal->Close())
{
        std::cout << MSG_QCP0103S << std::endl;
        //  Cleanup the local objects
        delete xeLocal;
        delete arsLocal;
        return APPRC_FATAL;
}

//  Cleanup the local objects
delete xeLocal;
delete arsLocal;
return APPRC_NOERROR;
```

5.12 Active Code in the Application Transaction Handler

This section describes the code that is implemented in the key interfaces of the class that extends the TransactionHandler.

5.12.1 MarshallTransaction Interface

The MarshallTransaction interface is invoked whenever the transaction handler detects that there is a new transaction document available for execution in the repository. The job of the interface is to create a request object from the transaction document in the database and pass this back to the transaction handler with information about what should be done with the request.

The first step is to remove any transaction level log that is present in the transaction document.

```
//  Remove the old transaction history log from the transaction
stAPIRC = NSFItemDelete(hnTX, TRANSDOC_HISTORY, (WORD) strlen(TRANSDOC_HISTORY));
```

The next step is to flag the transaction document with a start of processing timestamp.

```
//  Set the copy start time timestamp
OSCurrentTIMEDATE(&tdCurrent);
stAPIRC = NSFItemSetTime(hnTX, TRANSDOC_STARTTIME, &tdCurrent);
```

Next create the new QCopyRequest object and provide it with a text list that will be used to hold any transaction log entries.

```
//  Create a New QCopy Request
qcrNew = new QCopyRequest();

//  Create a new Text List to hold the Transaction Log History
stAPIRC = ListAllocate(0, 0, FALSE, &qcrNew->hTXLog, &pList, &wRetLen);


//  UnLock the memory for the list
OSUnlockObject(qcrNew->hTXLog);
```

Now the contents of the request object can be populated from the information in the transaction document. The request object itself contains the code needed to find the information in the transaction document.

```
//  Populate the request from the transaction document
if (!qcrNew->populate(nidTX, hnTX))
```

The QCopy application contains code to perform checks against the source and target servers for a copy request and if either of these servers is not available either because it is down or is too busy then it can delay attempting to execute the transaction until later.

```
//  Check the availability of the source and target servers
if (checkAvailability(qcrNew, iThreadID) != STATE_EXEC_REQUEST)
{
       //  The transaction should be delayed
       dwDelayed++;
       delete qcrNew;
       return TX_DISPOSITION_DELAY;
}
```

The QCopyRequest is not passed directly to the processing engines instead a PartCopyRequest object is passed that contains a pointer to the QCopyRequest. This is done to allow the processing engines to present an isomorphic asynchronous execution entry point.

```
//  Set up the rest of the information required by the interface
qcrNew->PermitRun = TRUE;                //  Set the Run Permit

//  The Copy Request is sent as the parent to a low level copy request
pcrSend = new PartCopyRequest(qcrNew);   //  Create the low level request
pcrSend->CopyAction = COPY_DATABASE;     //  Top level request type
*txObject = pcrSend;                     //  Address of the request object
```

The code now determines which processing engine should process the current request, the database copy engine or the feeder transaction engine. The parameters that need to be set to execute the request are set as appropriate and the code returns to transaction handler to dispatch the request.

```
//  Determine if this is a copy transaction or a feeder transaction and dispatch
accordingly
```

```
if ((!xeLocal->IsThisADatabase(qcrNew->szSourceDatabase)) || (HasWildcards(qcrNew-
>szSourceDatabase)))
{
        //  Feeder transaction
        *xxObject = qfMaster;             //  Processed by the QCFeeder engine
        *Attrs = RQATTR_REJOIN | RQATTR_CMLSL1;
        *Priority = 20;
}
else
{
        //  Copy transaction
        *xxObject = qcMaster;             //  Processed by the QCopier engine
        *Attrs = RQATTR_REJOIN | RQATTR_CMLSL2;
        *Priority = 100;
}
return TX_DISPOSITION_NORMAL;
```

If any error or out-of-envelope condition is detected in the interface then it can return a signal to cause the transaction handler to mark the transaction as being in error.

```
sprintf_s(szMsg, MAX_MSG, MSG_QCP0303E, nidTX);
LogTransactionMessage(szMsg, qcrNew->hTXLog, iThreadID);
//  Append the history log to the transaction
AddHistory(qcrNew->hTXLog, hnTX, iThreadID);
delete qcrNew;
return TX_DISPOSITION_ERROR;
```

5.12.2 *SerializeTransaction Interface*

The SerializeTransaction interface is the counterpoint of the MarshallTransaction interface, the interface is invoked whenever a request completes so that the resulting status, log and any other information can be written back to the appropriate transaction document.

The first step is to open the transaction document.

```
//  Open the transaction note
stAPIRC = NSFNoteOpen(hdbQueue, qcrOld->nidRequest, 0, &hnTransaction);
```

Next the transaction is stamped with a completion timestamp.

```
//  Set the copy finish time timestamp
OSCurrentTIMEDATE(&tdCurrent);
stAPIRC = NSFItemSetTime(hnTransaction, TRANSDOC_FINISHTIME, &tdCurrent);
```

If the transaction completed successfully then the processing statistics are written to the transaction document and the appropriate status updates are performed and the transaction log is added to the document.

```
if (qcrOld->ReturnCode == RETURN_NOERROR)
{
        //  Transaction completed ok
        //  Write the stats to the request document
        qcrOld->writeStats(hnTransaction);

        //  Mark the transaction as completed
        MarkTransactionCompleted(qcrOld->nidRequest, hnTransaction, tqCurrent,
iThreadID);
        dwCompleted++;
}
```

```
//  Append the history log to the transaction
AddHistory(qcrOld->hTXLog, hnTransaction, iThreadID);

//  Dispose of the transaction object
delete qcrOld;
```

If the transaction was not successful then the code must determine if it can be re-tried or it should be marked as a permanent error. When a transaction is re-tried the original transaction is saved with all of the processing information and a new transaction is generated as a clone of the original. There is a limit on the number of times that a transaction can be re-tried, if this count is exceeded then the transaction is marked as a permanent error. When a transaction is to be re-tried then is written back to the delayed transaction queue for later execution.

```
//  Mark the transaction as retried
MarkTransactionRetried(qcrOld->nidRequest, hnTransaction, tqCurrent, iThreadID);
dwRetried++;
//  Create a new transaction document for the retry request
hdbLocalRep = xeLocal->GetMappedDBH(xeLocal->hdbRepository, iThreadID);
stAPIRC = NSFNoteCreate(hdbLocalRep, &hnNewTX);
//  Increment the retry count
qcrOld->RetryCount++;

//  Seralize the transaction data to the new request document
if (!qcrOld->serialize(0, hnNewTX, xeLocal))

//  Get the NoteID
NSFNoteGetInfo(hnNewTX, _NOTE_ID, &qcrOld->nidRequest);

//  Set the Urgent & Approved flags
NSFItemSetText(hnNewTX, TRANSDOC_URGENT_FLAG, "0", MAXWORD);
NSFItemSetText(hnNewTX, TRANSDOC_APPROVED_FLAG, "Yes", MAXWORD);

//  Mark the new transaction as delayed
MarkTransactionDelayed(qcrOld->nidRequest, hnNewTX, FALSE, tqCurrent, iThreadID);
dwDelayed++;

//  Save and close the new transaction
NSFNoteUpdate(hnNewTX, UPDATE_FORCE);
NSFNoteClose(hnNewTX);
```

5.12.3 MarkTransaction<status> Interfaces

These interfaces are used to mark transactions with the appropriate status values that are used to determine their disposition on the various different transaction queues. The base TransactionHandler class provides default implementations for these interfaces, these may be overridden in any inheriting class to implement different transaction processing models.

```
void virtual MarkTransactionDelayed(NOTEID nidTX, NOTEHANDLE hnTX, BOOL bInProgress,
TransactionQueue *tqCurrent, int iThreadID)

void virtual MarkTransactionReady(NOTEID nidTX, NOTEHANDLE hnTX, BOOL bInProgress,
TransactionQueue *tqCurrent, int iThreadID)

void virtual MarkTransactionError(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionInProgress(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionCompleted(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)

void virtual MarkTransactionRetried(NOTEID nidTX, NOTEHANDLE hnTX, TransactionQueue
*tqCurrent, int iThreadID)
```

The following describes the normal transaction life cycles managed by the transaction handler.

Transactions may be marked for once-off execution at a particular point in time or for repeated execution at particular time intervals. These transactions have a status value of “SCHEDULED” and reside on the Schedule Queue. When the execution time is reached or the scheduled interval expires then these transactions are copied to the Ready Queue.

The Ready Queue is the queue monitored by the transaction handle to find work that is ready to be executed. These transactions have a status value of “NEW”. When transactions are marshalled for execution they are moved to the In Progress Queue. If the transaction should not be processed at the current time, for whatever reason then it is marked with a status value of “DELAYED” and moved to the Delayed Queue.

Transactions on the In Progress Queue are considered to be currently executing and have a status value of “INPROGRESS”.

Once a transaction has completed processing it will be passed to the SerializeTransaction interface to determine the disposition and move it to the appropriate queue. If the transaction completed successfully it will be stamped with a status value of “COMPLETED” and moved to the Completed Queue. If the transaction failed then the destination is determined by the transaction retry settings for the current queue and transaction. If the transaction can be retried and the retry limit has not been exhausted then a new copy of the transaction is marked with the “DELAYED” status value and moved to the Delayed Queue, the original transaction is stamped with the “RETRIED” status value and moved to the Delayed Queue for later execution. If the transaction has failed and does not support retries or the retry limit has been exhausted then it will be marked with the “ERROR” status value and moved to the Error Queue.

The Delayed Queue is scanned occasionally and if a transaction has been on that queue for long enough then it is marked with the “NEW” status value and returned to the Ready Queue.

When a transaction handle starts to process a queue then, depending on settings, it may scan the In Progress Queue and process any transactions as if they had failed.

5.13 Active Code in the QCopier

The QCopier class extends the DbCopier class to implement the ProxyLogMessage interface this allows it to capture log messages from the DbCopier and record the messages in the individual requests so that they can later be written back to the appropriate transaction document.

The DbCopier calls the ProxyLogMessage interface for selected log messages to allow extending classes to capture the log message before writing them to the standard run time log. In the case of the QCopy application these messages are added to the Text List that is provided in the QCopyRequest by the application transaction handler. There are size limits to a Text List so the interface implements a check for an overflow of the Text List and if detected it sets a switch that causes the DbCopier code to no longer write messages about folder processing to the interface, folder processing messages can be numerous if the database being copied contains many folders.

As a last step in the process the passed message is written to the standard run time log interface.

5.14 Active Code in the Application Command Handler

Code in the CustomCommandHandler interface is used to implement custom commands or extend/replace system commands that would be handled by the base CommandHandler class. In this section we describe the processing for the STATUS command and the MAXTRANS command.

All commands, both system and custom are passed to the interface as an encoded value. The selection of the appropriate processing is performed in a switch construct. The default action is to return the passed command for processing by the base class implementation.

```
switch(wCommand)
{
case MQ_COMMAND_xxxx:
       //  Processing for the selected command
              break;
default:
       //  Pass the command on to the system command processor
       wNextCommand = wCommand;   //  Pass the command on to the system processor
       break;
}
return wNextCommand;
```

The STATUS command implementation displays additional status messages and then passes the command to the base class for it to display the standard output.

```
case MQ_COMMAND_STATUS:
       //  This is the system status command - before it is issued we show some custom 
messages
       if (tqCurrent == NULL) strcpy_s(szMsg, MAX_MSG, MSG_QCP0202E);
       else sprintf_s(szMsg, MAX_MSG, MSG_QCP0201I, tqCurrent->dwStarted, tqCurrent-
>dwCompleted, tqCurrent->dwStarted - tqCurrent->dwCompleted, tqCurrent-
>MaxConcurrent);
       xeLocal->LogMessage(szMsg, iThreadID);
       AddInLogMessageText(szMsg, NOERROR);
       sprintf_s(szMsg, MAX_MSG, MSG_QCP0208I, thCurrent->dwCompleted, thCurrent-
>dwError, thCurrent->dwRetried, thCurrent->dwDelayed);
       xeLocal->LogMessage(szMsg, iThreadID);
       AddInLogMessageText(szMsg, NOERROR);
       //  Inform if the queue is suspended
       if (tqCurrent->QueueIsSuspended)
       {
              sprintf_s(szMsg, MAX_MSG, MSG_QCP0205I, tqCurrent->szQName);
              xeLocal->LogMessage(szMsg, iThreadID);
              AddInLogMessageText(szMsg, NOERROR);
       }
       wNextCommand = wCommand;   //  Pass the command on to the system processor
       break;
```

Note that the messages are written to the console (AddInLogMessageText) as well as to the run time log.

The MAXTRANS command alters settings in the transaction queue and displays the new setting. The command is identified as being consumed by the CustomCommandHandler interface.

```
case MQ_COMMAND_MAXTRANS:
       //  Change the max number of concurrent transaction that are being handled by 
the Queue
       iNewMaxTrans = atoi(szOptions);   //  Get the new value to be set
       if (iNewMaxTrans == 0)
       {
              sprintf_s(szMsg, MAX_MSG, MSG_QCP0206E, tqCurrent->szQName);
              xeLocal->LogMessage(szMsg, iThreadID);
              AddInLogMessageText(szMsg, NOERROR);
       }
       else
       {
              iOldMaxTrans = tqCurrent->MaxConcurrent;
              tqCurrent->MaxConcurrent = iNewMaxTrans;
              sprintf_s(szMsg, MAX_MSG, MSG_QCP0207I, tqCurrent->szQName, iOldMaxTrans, 
iNewMaxTrans);
              xeLocal->LogMessage(szMsg, iThreadID);
```

```
                AddInLogMessageText(szMsg, NOERROR);
        }
        wNextCommand = MQ_COMMAND_NULL;
        break;
```

5.15 Summary

The design pattern is essentially simple based on two or optionally three core classes assembled to operate together.

Construct a processing engine that implements the Runnable interface, the processing engine should be capable of responding to a request object by carrying out the required work by breaking it down into smaller units of work and executing them asynchronously.

Construct an “Injector” that is capable of discovering work that needs to be done and injecting request objects into the processing engine and recovering the results. The Injector model in this case is the transaction handler however it can be any one (or more) of the following.

- A Transaction Queue
- A timer driven request generator
- An external event driven request generator
- Environmental monitor that triggers requests
- Requests generated from information passed on the command line

Decouple the processing engine from the injector by extending the processing engine class and the processing request class to carry any information needed by the work injector. This decoupling makes the processing engine more flexible and capable of implementation in additional application configurations.

Optionally construct a command processing class to handle any commands, these commands can trigger the generation of work requests.

In the main line code of the application create singleton objects of the three classes and asynchronously dispatch a request to initiate execution of the injector, then wait until a terminal state is detected.

Complex applications can be built up that contain multiple injectors and multiple, possibly chained processing engines.

6. Debug Builds

This section describes some of the additional features that are activated in DX applications when compiled and linked with DEBUG settings in effect.

6.1 Logging

In addition to the ability to raise the execution logging level of a DX application when it is invoked to Tracing (-T) or Debugging (-D) level, a capability that is present in both Debug and Release builds of applications some DX modules contain additional debugging messages that only included in Debug builds. These additional messages are always written to STDOUT, they are unnumbered but are always prefixed by the literal “DEBUG:” in the message string. These development phase messages are no longer removed from DX code modules during Quality Engineering reviews.

6.2 The Debug Helper Class

This class is only implemented for Windows Platforms and only exposes functionality in Debug builds.

The Helper class is used to establish an object that provides additional diagnostic capabilities to the runtime environment. Objects of this class should only be constructed in DEBUG build configurations of an application.

The class provides services for monitoring application memory usage and producing Core Dumps on demand.

6.2.1 Reporting on Memory Usage

Calls to the ReportMemoryUsage generate a report in the current log and optionally on the console of the current allocation of memory by the application, the call also reports on the difference in memory allocation since last reported and since the Helper was created.

The functions report on the memory usage by the application in the C runtime heap.

Sample Output:

```
Current(3) Working Set size: 30704 Kb, +328 Kb since last measured, +4580 Kb since first measured, Peak: 30704 Kb.
Current(3) Paged Pool use: 1252 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 1252 Kb.
Current(3) Non-Paged Pool use: 10 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 10 Kb.
Current(3) Normal Objects on the Heap: 28 , 0  since last measured, +7  since first measured, Peak: 28.
Current(3) Normal Objects Allocation: 124 Kb, 0 Kb since last measured, +2 Kb since first measured, Peak: 124 Kb.
Current(3) Client Objects on the Heap: 0 , 0  since last measured, 0  since first measured, Peak: 0.
Current(3) Client Objects Allocation: 0 Kb, 0 Kb since last measured, 0 Kb since first measured, Peak: 0 Kb.
```

The report indicates the sequence number of memory reports “Current(3)” indicates the third time that the reporting method has been called. Each line of output references a different memory statistic and shows the current, delta since last reported, delta since first reported and the peak measurement of the particular statistic. The statistics of particular focus for programmers are the count and size of “Normal Objects” on the heap, steady increases in these values would indicate a leak of C++ objects from within the application.

6.2.2 Creating Memory Dumps

The CreateMemoryDump function can be called at any point in an application to create a “minidump” file of the application process including heap memory. The memory dump files are created in the

"IBM_TECHNICAL_SUPPORT" sub-directory in the Notes data directory. The minidump files can be loaded into Visual Studio for contextual analysis or in the Windows Debugger (WinDbg).

The dump files are created with a standard name format:

DXDump-<appname>YYYYMMDD-HHMMSS.dmp.

Where <appname> is the name of the application and YYYYMMDD-HHMMSS is the timestamp that the dump was created.

6.3 Memory Leak Detection

Debug compilations of DX applications also enables the C runtime memory leak tracing protocols. At any point in a program a call can be made to the C Runtime "CheckMemoryLeaks()" (in fact this is a DX implemented macro that invokes the actual CRT entry point: `_CrtDumpMemoryLeaks()`) method and this will report on each object that is allocated on the Heap giving the source file and line number where it was allocated as well as the size of the object. A call to CheckMemoryLeaks should be made immediately before an application terminates this will show any objects that remain allocated and provides an excellent means of detecting and fixing leaks caused by failing to delete C++ objects or failing to free memory allocations.

There is a good deal of information published on the internet on the topic of using the debug heap in C++ applications unfortunately a good deal of it is either misleading or downright incorrect.

The Microsoft debugging heap implements debugging traps that can be used to intercept all allocations and de-allocations of memory on the heap, in addition it provides certain default behaviours. Memory allocations on the heap are extended to include sentinel memory before and after each allocated chunk of heap memory and the allocated chunks are filled with particular memory patterns when they are allocated and freed, these measures can detect certain memory overwrites on the heap. Memory applications on the heap are also supplemented with trace information indicating in what code module and at what line number they were allocated, calls to the `_CrtDumpMemoryLeaks()` write a list of this trace infoirmation for every allocation on the heap. One problem with using the stock functions in a C++ application is that all allocations on the heap that arre made during the creation of a C++ object report that the allocation was made by code in the "new" operator in the standard C++ run time, which is not much good for debugging leaks (unless you implement code to capture and analyse code stacks during allocations). However you can fix this shortcoming with a limited number of definitions in your application header files, in DX applications this is done for you in the PlatBase.h header file these modifications are position sensitive so should always be in the first header file that is included. The changes make the trace information for heap allocations for the creation of C++ objects report the module and line number where the "new" operator is used and hence the output from `_CrtDumpMemoryLeaks()` calls now provides more useful information without having to incorporate additional code.

The following shows the definitions from the PlatBase.h header file that enables this feature.

```
#ifdef _DEBUG
#define _CRTDBG_MAP_ALLOC  //  Have file & line recorded in memory allocations
#include <stdlib.h>        //  Standard functions
#include <crtdbg.h>        //  Include the runtime debugging macros
#endif
```

You should include all necessary C/C++ language includes at this point.

```
#ifdef _DEBUG
#define DEBUG_NEW new(_NORMAL_BLOCK, __FILE__, __LINE__)
#define new DEBUG_NEW
#endif
```

These statements switch the existing "new" operator form to use the debugging alternative.

The section has so far only discussed the detection of memory leaks on the C/C++ heap, there is also the issue of leaks of Domino resources and/or memory. Previous releases of the DX kernel code used to include Domino resource tracking mechanisms similar to the Heap memory tracking described above, however over time we have seen that following our coding style guides had so greatly reduced the frequency of Domino resource leaks that it was no longer worthwhile maintaining this facility, so it has been dropped. In place of run time resource tracking we now include testing in our regression tests that specifically look for Domino resource leakages by employing the following protocol.

- Prepare the server or workstation to execute the test.
- Shut down the server or workstation.
- Re-start the server or workstation.
- Take an NSD of the server or workstation.
- Run the test, while the application is running take another NSD.
- Take another NSD of the server or workstation.
- Analyse the last NSD for any suspicious allocations that were made by the application under test, if found check back in the first NSD to confirm that the allocation was not there before the application was run.

6.4 Instrumentation

The DX kernel contains an instrumentation package that records certain data from events in the kernel, the facility can be activated by performing a build with the DXIP symbolic defined (-DDXIP) this setting is always asserted in Debug builds.

In a DX kernel that is built with the instrumentation active the kernel collects records and periodically writes these records to an instrumentation recording file. The recording files are created in the "IBM_TECHNICAL_SUPPORT" sub-directory in the Notes data directory.

The recording files are created with a standard name format:

DXDump-<appname>YYYYMMDD-HHMMSS.ipr.

Where <appname> is the name of the application and YYYYMMDD-HHMMSS is the timestamp that the recording was created.

The DXIPA application can be used to display information from an Instrumentation Package Recording (.ipr) file, see "Debugging Tools" below.

The current kernel ships with only a single recording record type defined and collected. These records are collected at the point where a request completes execution in one of the worker threads in the threadpool. The record contains the elapsed time that the worker thread was idle before the request arrived, this is referred to as the "dwell time". The record also contains the elapsed time that the request was executing in the thread, this is referred to as the "mill time" and the dispatch priority of the request.

6.5 Debugging Tools

This section describes the debugging tools specifically developed for DX applications that are publicly available.

6.5.1 DXTell

When initially testing a new or updated Server Add-In Task the first thing that you need to do is to prepare a test server to do the testing on. While it is extremely easy to install a new Domino server it can be time consuming to prepare that environment just for initial "kick the tyres" testing. Also if your code causes a

PANIC or other fatal condition you may have to wade your way through a rather large NSD to locate an obvious (when you have seen it) but trivial problem. Well, there is a better way.

Little Known Fact #247

All of the Server API calls and functionality that is required to support a Server Add-In task is also available in the Notes Workstation code.

Testing Methodology

Install the Server Add-In task in your Notes Executable directory. Have any local databases that are needed by your Add-In available on the client. Shut down your Notes Client, this minimises the volume of extraneous clutter that will appear in any NSD that you might have. Open two DOS Prompt windows. In the first window you launch your server Add-In by typing what you would type as parameters to the Load command if you were running your Add-In task on a server and your Add-In will start and, hopefully, begin it's normal processing. Should a fault occur then you will be offered the choice of debugging the problem with any of your installed debuggers. The second DOS prompt window that you opened is for the cunning bit. When running your Add-In on a server you control your Add-In task through the use of "Tell" command entered through the server console. From the second window you can use the DXTell program to pass commands to your Add-In in the same way that you would on a server.

e.g. DXTell MyAddin stop

This will pass a "stop" command to the Add-In task, assuming that it has created a Message Queue (MQ) with a name of "MyAddin" and that the Add-In task is correctly monitoring that Message Queue (MQ).

The second DOS Prompt Window can also be useful for running NSD commands. If you suspect that your program is looping you can use the "nsd -kill" or "nsd -dumpandkill" commands to terminate or terminate and dump your application. You can also use the "nsd -monitor" command to start an interactive NSD session, this can be particularly useful to inspect the stacks of your application at regular intervals to verify correct execution or to aid with problem diagnosis. Take note of the pid of your application, this will be reported when nsd attaches to it, then periodically issue the "DUMP 0x<your pid>" command (the pid is reported in hex so it is vital to prefix it with "0x").

When you have finished you should issue the "DETACH" command to detach nsd from all of the processes and then issue the "QUIT" command to shut nsd down.

6.5.2 DXIPA

The DXIPA tool is provided to extract readable output from an Instrumentation Package Recording (.ipr) file. The application can list records in a recording stream, produce frequency tables in text or histogram form according to the command line options specified. All outputs from DXIPA are written to the standard output file (STDOUT).

Syntax:

```
DXIPA <ipr file name> [-D][-S][-C]
```

The –D switch produces the output in list (dump) format, each record is annotated from the stream in the order that they appear. A short sample is shown below with an explanation of the output.

```
DX Instrumentation Package Analyser (DXIPA) Version: 1.0.0 build: 02 analysing:
/home/domino/cyclotron/IBM_TECHNICAL_SUPPORT/DXIP-QCopy20111229-124517.ipr
+0102584, TID=9, Evt=1, Prio=10, Dwell=0, Mill=126.
+0102576, TID=8, Evt=1, Prio=10, Dwell=0, Mill=230.
+0102580, TID=7, Evt=1, Prio=15, Dwell=0, Mill=442.
+0102710, TID=9, Evt=2, Prio=15, Dwell=21, Mill=366.
+0103022, TID=7, Evt=2, Prio=11, Dwell=21, Mill=90.
+0102806, TID=8, Evt=2, Prio=10, Dwell=23, Mill=350.
+0103097, TID=9, Evt=3, Prio=11, Dwell=21, Mill=141.
+0102598, TID=10, Evt=1, Prio=15, Dwell=0, Mill=660.
+0103133, TID=7, Evt=3, Prio=11, Dwell=21, Mill=151.
+0102604, TID=3, Evt=2, Prio=15, Dwell=21, Mill=702.
```

Each line in the output shows that values collected from a single instrumentation record. The first column `+0102584` shows the elapsed time in milliseconds since the application was started that the record was collected. The second column `TID=9` shows the identity of the worker thread from which the record was collected. The third column `Evt=1` shows the relative event number on the thread this allows sequencing of events from a single thread. The fourth column `Prio=10` shows the priority at which the request was dispatched to the worker thread. The fifth column `Dwell=0` shows the number of milliseconds that the worker thread was idle before processing the current request. The sixth column `Mill=126` shows the elapsed time in milliseconds that the current request took to complete execution.

The –S switch tells the analyser application to collate statistics on the stream in the recording file. A sample is shown below with explanation.

```
Mill Times.

 3250 - 3299: 1.
 3200 - 3249: 1.
 3150 - 3199: 0.
 3100 - 3149: 0.
 3050 - 3099: 1.
 3000 - 3049: 1.
 2950 - 2999: 2.
 2900 - 2949: 0.
 2850 - 2899: 5.
 2800 - 2849: 3.
 2750 - 2799: 0.
 2700 - 2749: 1.
 2650 - 2699: 2.
 2600 - 2649: 2.
 2550 - 2599: 3.
 2500 - 2549: 4.
 2450 - 2499: 3.
 2400 - 2449: 5.
 2350 - 2399: 3.
 2300 - 2349: 4.
 2250 - 2299: 0.
 2200 - 2249: 1.
 2150 - 2199: 2.
 2100 - 2149: 4.
 2050 - 2099: 6.
 2000 - 2049: 8.
 1950 - 1999: 4.
 1900 - 1949: 2.
 1850 - 1899: 7.
 1800 - 1849: 4.
 1750 - 1799: 7.
```

```
1700 - 1749: 1.
1650 - 1699: 2.
1600 - 1649: 4.
1550 - 1599: 7.
1500 - 1549: 5.
1450 - 1499: 5.
1400 - 1449: 5.
1350 - 1399: 3.
1300 - 1349: 2.
1250 - 1299: 7.
1200 - 1249: 5.
1150 - 1199: 4.
1100 - 1149: 7.
1050 - 1099: 8.
1000 - 1049: 13.
0950 - 0999: 9.
0900 - 0949: 15.
0850 - 0899: 16.
0800 - 0849: 7.
0750 - 0799: 16.
0700 - 0749: 19.
0650 - 0699: 23.
0600 - 0649: 27.
0550 - 0599: 35.
0500 - 0549: 37.
0450 - 0499: 60.
0400 - 0449: 63.
0350 - 0399: 78.
0300 - 0349: 95.
0250 - 0299: 100.
0200 - 0249: 134.
0150 - 0199: 125.
0100 - 0149: 110.
0050 - 0099: 32.

Low outliers: 23, 24, 25, 23, 24, 23.
High outliers: 79674, 95846, 243545.
```

The first column `3250 - 3299:` shows the bounds of the interval of times that were counted in the bucket being displayed, in this case from 3,250 milliseconds up to 3,299 milliseconds inclusive. The second column shows the frequency with which timings fell within the bucket being listed, in this case 1.

The two lines at the bottom of the table show any times that were excluded from the table because they were considered too low or too high.

The –C switch (Chart) will output the statistics as shown in the tables in the form of a frequency histogram.

```
150 -|
     |
     |
     |   *
     |   *
125 -|  **
     |  **
     |  **
     | ***
     | ***
100 -| ****
     | *****
     | *****
     | *****
     | ******
075 -| ******
     | ******
     | *******
     | ********
     | ********
050 -| ********
     | ********
     | *********
     |***********
     |************
025 -|*************
     |*************** *
     |*************** ** *
     |**********************  *     *   * *  **
     |******************************************** ********* ** ***  **
      |---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
      0050      0550      1050      1550      2050      2550      3050      3550      4050      4550      5050
```

The Y axis shows the frequency and the X axis shows the timing value.

7. Building DX Applications

7.1 Reference Platforms

DXTools and the DXCommon kernel are portable across multiple platforms that support the Notes API. However there are a limited set of reference environments on which they are regularly built and regression tested.

Windows:

Build Environment:

Microsoft Visual Studio 2005/2010/2011/2013

Version 8.0.50727.867 (vsvista.050727-8600)

Running on any supported windows workstation.

Note: Backward compatibility tests are done with Visual Studio 2003 as that is the officially supported development platform for the Notes API.

Notes API Version 9.0.

Execution Environment:

Windows Standard Server 2008 R2 (32 bit and 64 bit).

Domino Server 9.0.1 FP1

Note: Execution environments from Domino 6.5.x through 9.0.x are regularly used.

Linux:

Build Environment:

Gcc Version: 4.1.2 for i386-redhat-linux.

Running on Redhat Linux 6.6

Notes API Version 9.0

Execution Environment:

Redhat Linux 6.6

Domino Server 9.0.1 FP1

Note: Execution environments from Domino 7.0.x through 9.0.x are regularly used.

Note: Some modules depend on the “curl” package for HTTP interaction this is standard in RHEL6 however other distributions may require installation of the “curl” RPM.

7.2 Notes API Installation

For both Windows and Linux DXTools assumes that the Notes API is installed in the default configuration specified in the API documentation. Include and Library directory should be added to search paths as indicated in the API documentation.

7.3 Project Directory Structure

The default Domino eXplorer development environment follows the Visual Studio paradigm of a “Solution” directory that contains multiple “Project” directories with a single application directory per application, this paradigm is followed on both Windows and Linux development environments.

The standard DX header and code files are **NOT** designed to be added to the default include search directories on either environment. The package expects to find a directory called “DXCommon” as a project level directory in each solution directory that will be used to build DX applications, these should be included in applications by relative re-direction. An include statement for the PlatBase.h header file that is located in the “Platform” sub-directory of the DXCommon package would be coded as follows.

```
//  Platform Includes
#include      "../DXCommon/Platform/PlatBase.h" //  Basic platform includes
```

Although a copy of the DXCommon package could be physically placed in each solution directory it would be more usual to place the package in a shared location on the development workstation/server and then create a symbolic link in each of the solution directories.

Windows:

The DXCommon kernel is supplied as a zipped archive (.zip). The contents of the archive should be unpacked to either the <solution directory>\DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>\DXCommon directory.

As an example.

Unpack the DXCommon kernel into a directory “c:\usr\include\DXCommon-3.12.0” and then create the symbolic link from within the solution directory using the following command.

```
mklink /D DXCommon “c:\usr\include\DXcommon-3.12.0”
```

Linux:

The DXCommon kernel is supplied as a gzipped archive (.tar.gz). The contents of the archive should be unpacked to either the <solution directory>/DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>/DXCommon directory.

File ownership and access settings should be adjusted according to your local policies.

As an example.

Unpack the DXCommon kernel into a directory “/usr/include/DXCommon-3.12.0” and then create the symbolic link from within the solution directory using the following command.

```
ln -s /usr/include/DXCommon-3.12.0 DXCommon
```

This deployment model allows different levels of the DXCommon package to be used in different solutions without reconfiguring the development environment.

7.4 Installing the DXCommon Kernel Sources

Windows:

The DXCommon kernel is supplied as a zipped archive (.zip). The contents of the archive should be unpacked to either the <solution directory>\DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>\DXCommon directory.

As an example.

Unpack the DXCommon kernel into a directory “c:\usr\include\DXCommon-3.12.0” and then create the symbolic link from within the solution directory using the following command.

```
mklink /D DXCommon “c:\usr\include\DXcommon-3.12.0”
```

Linux:

The DXCommon kernel is supplied as a gzipped archive (.tar.gz). The contents of the archive should be unpacked to either the <solution directory>/DXCommon directory or unpacked to a directory that will then be used as the base for a symbolic link from the <solution directory>/DXCommon directory.

File ownership and access settings should be adjusted according to your local policies.

As an example.

Unpack the DXCommon kernel into a directory "/usr/include/DXCommon-3.12.0" and then create the symbolic link from within the solution directory using the following command.

```
ln -s /usr/include/DXCommon-3.12.0 DXCommon
```

7.5 Installing the Application Sources

Windows:

The DX Tools application sources are supplied as a zipped archive (.zip). Create an empty project called <application name> in the <solution directory>. Then unpack the contents of archive into the project directory and add each of the source and header files to the project. Add any needed kernel header and code files to the project (the file are in the <solution directory>\DXCommon directory and sub-directories).

Linux:

The DX Tools application sources are supplied as a gzipped archive (.tar.gz). Create the <application name> project directory within the <solution directory> unpack the contents of the archive into that directory. Review the supplied (minimal) make file and edit it to reflect any local conventions.

File ownership and access settings should be adjusted according to your local policies.

7.6 Build Settings

Windows 32 bit:

The following non-default settings should then be made to the project settings. Any other settings should not prevent a successful build.

Section/Entry	Release Setting	Debug Setting
General		
Character Set	"Not Set"	"Not Set"
C/C++		
Preprocessor		
Preprocessor Definitions	WIN32;NDEBUG;_CONSOLE;W32	WIN32;_DEBUG;_CONSOLE;W32
Code Generation		
Runtime Library	Multi-threaded (/MT)	Multi-threaded Debug DLL (/MTd)
Struct Member Alignment	1 Byte (/Zp1)	1 Byte (/Zp1)
Command Line		
Additional Options	/Oy-	/Oy-
Linker		
Input		
Additional Dependencies	notes.lib	notes.lib Dbghelp.lib Psapi.lib
For Resource Loader	winhttp.lib	winhttp.lib
For NE Resolver	Wldap32.lib	Wldap32.lib

Windows 64 bit:

Section/Entry	Release Setting	Debug Setting
General		
Character Set	Not Set	Not Set
C/C++		
Preprocessor		
Preprocessor Definitions	WIN32;NDEBUG;_CONSOLE;NT;W32;W;W64;ND64;_AMD64_;ND64SERVER	WIN32;_DEBUG;_CONSOLE;NT;W32;W;W64;ND64;_AMD64_;ND64SERVER
Code Generation		
Runtime Library	Multi-threaded (/MT)	Multi-threaded Debug (/MTd)
Struct Member Alignment	1 Byte (/Zp1)	**Default**
Command Line		
Additional Options	/Oy-	/Oy-
Linker		
Input		
Additional Dependencies	notes.lib	notes.lib Dbghelp.lib Psapi.lib

Notes:

Static linking of the runtime is used as since the advent of Side-By-Side (SXS) assembly of applications it is increasingly common to find server environments that do not have the latest C/C++ Runtime manifests installed.

/Zp1 packing is a Notes API requirement as all Notes API structures are packed and not padded or member aligned. **IMPORTANT**: Do not set this option for Windows x64 (64 bit) builds, Notes uses the default structure packing on Windows 64 bit.

/Oy- is an important setting, without it the compiler will use the Frame Pointer as a general purpose register rather than pointing to the current frame, this will cause any NSD dump to be complete garbage and make debugging virtually impossible.

The additional libraries for the debug settings Dbghelp.lib and Psapi.lib are used to enable additional debug capabilities such as memory leak detection that are provided by DXCommon kernel modules.

Applications that use the DXResourceLoader class require the Windows HTTP library (winhttp.lib).

Applications that use the NEResolver class require the Windows LDAP library (Wldap32.lib).

Linux:

A minimal and environment independent make file is supplied in the source distribution of any DXTool application. The contents of the make file should be reviewed and any changes made to reflect local conventions or environment. The following shows the necessary settings for building any application that uses DX.

```
CC = g++
CCOPTS = -c -march=i486
NOTESDIR = $(LOTUS)/notes/latest/linux
LINKOPTS = -o <appname>
DEFINES = -DUNIX -DLINUX -DHANDLE_IS_32BITS
INCDIR = $(LOTUS)/notesapi/include
LIBS = -lnotes -lm -lnsl -lpthread -lc -lresolv -ldl
```

Compile:

```
$(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) <sources>
```

Link:

```
$(CC) $(LINKOPTS) <objects> -L$(NOTESDIR) -Wl,-rpath-link $(NOTESDIR) $(LIBS)
```

For applications that use the DXResourceLoader class add the `-lcurl` library to the `LIBS =` setting.

There is no specific build provided for a debug version of the application as the helper functions used for additional debugging are for the Windows platform only. Sorry! There is an enhancement request to have a linux specific debug helper added to the build.

NOTE: the output file name for the executable (specified in the LINKOPTS setting) should be folded to lowercase if the application is to be run on a server.

8. Deploying DX Applications

Windows:

Select “Build” and then Build <application name>.

Copy the resulting executable (<application name>.exe) and the associated Program Debug Database (<application name>.pdb) to the Notes Executable directory on the server or workstation where you want to run the application.

Linux:

From the <project> project directory issue the “make <application name>” command.

Copy the resulting executable (<application name>) to the Notes Executable directory where you want to run the application. According to your local security policies and Domino install you may need to have an administrator copy the executable and possibly change ownership of the executable.

The default ownership and attributes indicated by the Notes API documentation are as follows.

chown server <application name>

chgrp notes <application name>

chmod 2555 <application name>

8.1 Deploying an Application Specific Database

The installation of the application specific databases provided with DX Tools is done from a “Virtual Template” that is available on the internet, this section assumes that you have available and installed the Remote Database Create (**Windows:**
RDBCreate.exe **Linux:** rdbcreate) tool. If you do not have this tool then download the DXTool source for RDBCreate and build it. Refer to the “Using RDBCreate” manual.

8.1.1 Install the Database

From a command window go to the Notes Executable directory where RDBCreate exists enter the following command.

RDBCreate <server name> <database name> <templateURL> -V

Where:

<server name> is the abbreviated name of the server on which you want to install the control database.

<database name> is the name of the control database relative to the notes data directory.

<templateURL> is the URL for the Virtual Template you wish to install.

NOTE: RDBCreate can also be run directly on a server using the load command use the following command syntax.

load RDBCreate <server name>|Local <database name> <templateURL> -V

Specify “Local” for the <server name> to create the database on the same server where the command is being run.

9. Tuning DX Applications

The DXCommon kernel used in the DX Tools applications are the product of a research project that is investigating methods of autonomic optimisation in multi-threaded applications. There are many internal parameters that can be manipulated at the source level in order to effect changes in performance profile. The distributed version of the applications only allows the manipulation of the number of threads in the thread pool at run time (usually the –M:nnn> parameter. Try increasing the number of threads in use by small increments, throughput should increase to match the increase in the number of threads until it reaches a flat top. If increasing the number of threads shows no increase in throughput then it may be that the application is being limited by too small sizing of the per request workload, refer to the discussion on this topic in the “Threading Model” section of this document.

9.1 Long Fat Pipes

The term “Long Fat Pipes” refers to network connections that have plenty of capacity but a high round-trip time, this type of connection is often encountered with intercontinental connections. Performance investigations into TCP/IP on “long Fat Pipes” show that applications typically have problems filling the pipe and therefore suffer from slow throughput as turnarounds on the connection lead to high idle times during data transfers across the connection. A DX application can be started with more threads than the default to help alleviate this slow throughput. Each thread uses a separate connection between the source and target server and therefore a separate TCP/IP state this results in a lower probability that the network connection will be in a completely idle state. For an individual connection you should experiment with the thread setting to determine the minimum number of threads that maximises the throughput over the connection.

9.2 Small Databases

When performing transfers that involve manipulating large numbers of very small databases then throughput may suffer due to the overhead of serial operations on the database compared to the time spent in data transfer. To alleviate this constraint increase the number of transactions that can be executed simultaneously, this may also result in needing to increase the number of threads that are being used.