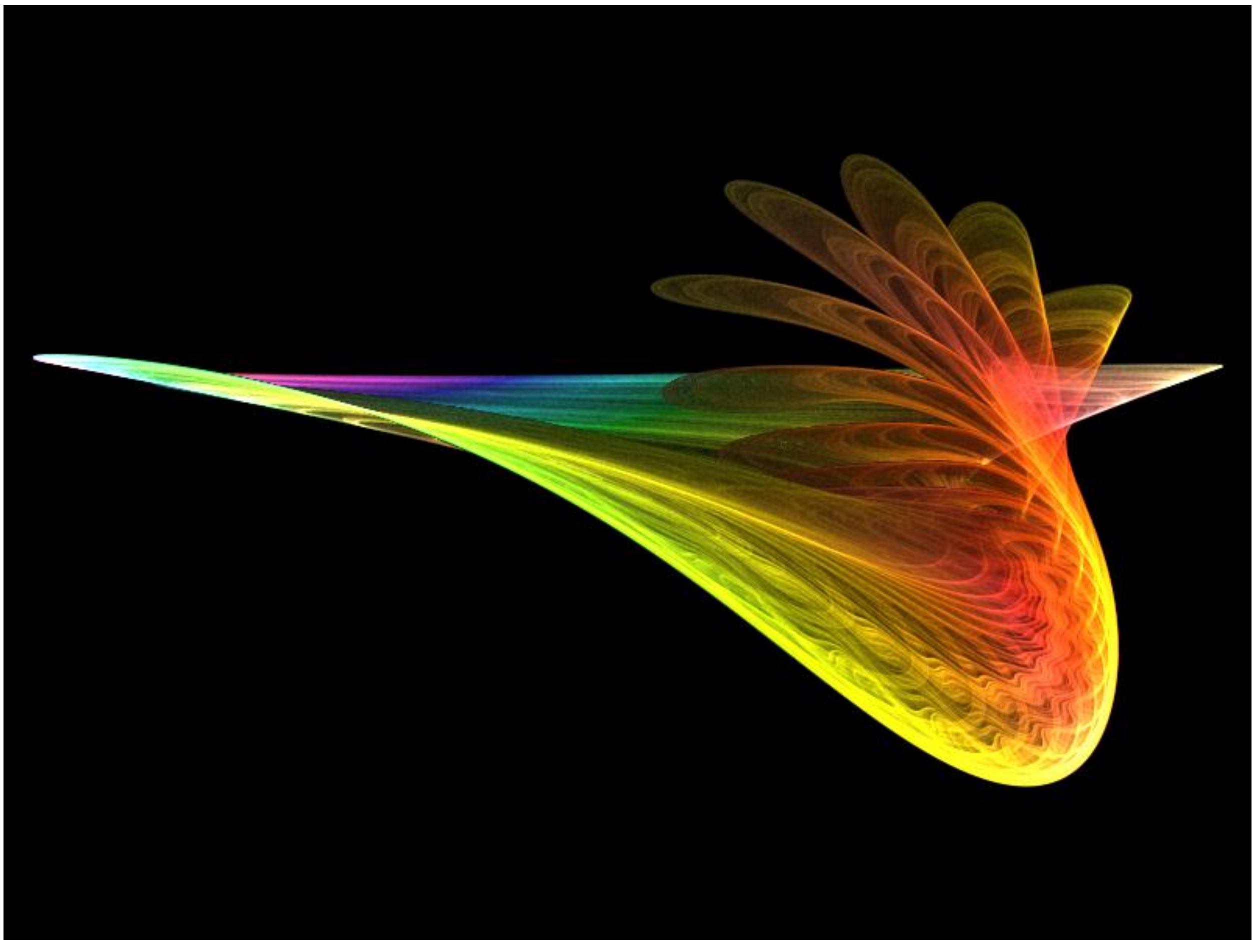
DX Tools

Using QMove 1.4



Author: Ian Tree
Owner: HMNL b.v.
Customer: Public
Status: Final
Date: 09/03/2012 16:03
Version: 1.4

Disposition: Open Source

Document History

Document Usage

This is an open source document you may copy and use the document or portions of the document for any purpose.

Revision History

Date of this revision: 10/03/2012 08:11	Date of next revision *None*

Revision Number	Revision Date	Summary of Changes	Changes marked
0.1	11/11/11	Initial Base Version	No
1.4	07/03/12	QE Version	No

Acknowlegements

Frontpiece Design was produced by the chaoscope application.

CHAOSCOPE.org
3D strange attractor rendering software

IBM, the IBM Logo, Domino and Notes are registered trademarks of International Business Machines Corporation.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

UNIX is a registered trademark of The Open Group.

All code and documentation presented is the property of Hadleigh Marshall (Netherlands) b.v. All references to HMNL are references to Hadleigh Marshall (Netherlands) b.v.

Contents

1. Introduction to QMove

QMove was originally developed to solve the problem of redistributing notes databases across multiple storage volumes without shutting down the Domino server or limiting the use of databases while they are being moved between storage volumes. Having addressed the problem of storage redistribution it was noted that QMove had other capabilities such as creating media backups and deleting databases already plumbed in and so minor code changes turned it into a more generic Notes database storage management tool. The name QMove was retained as storage redistribution remains the principal use of the utility.

QMove is fast and resilient. Multiple databases can be processed in parallel allowing sustained high rates of data transfer.

The QMove application is constructed as a Domino Server Add-In task, as such it is perfectly suited to unattended operations.

QMove is a C++ application constructed on the DXCommon application kernel supporting Domino version from 6.0 through 8.5 on both Widows & Linux server platforms.

The move engine at the heart of the QMove application is mature and well tested component having been used in production environments for many years. It has been used in different configurations to move over 50,000 databases with more than 50Tb of data.

2. QMove Transactions

This section of the document identifies the different options that can be used on a QMove transaction and how these options should be used.

The DbMover engine at the heart of QMove recognises three primitive actions that can be combined in different ways to accomplish different tasks. The primitive actions are Copy (C) which makes an online copy of an open (transaction logged) database, Delete (D) which deletes a database while it is online and Link (L) which creates a database link in place of the original database pointing to a new storage location (which can be outside of scope of the Notes Data Directory).

2.1 Pre-Processing Options

These options invoke processing that happens before the move operation begins.

2.1.1 Delete Database

This option will delete the database that is the target of the move operation. If the database does not exist then the transaction is not failed but continues with the move operation.

2.2 Source and Target Options

These options are used to select and optionally verify the source database and target directory and to specify the operation that is to be performed.

Note: The specifications for the selection and moving collections of databases (“Feeder Transactions”) are dealt with in a later section of this document.

2.2.1 Operation

This parameter determines which of the following operations is to be carried out on the specified database.

2.2.1.1 Move

This action will move an online, transaction logged database from the current storage location to a new location and bring the database online at the new location. This is accomplished by performing a Copy primitive action followed by Delete and then Link (CDL).

2.2.1.2 Retire

This action is used to “Retire” a database from a server, this involves taking a copy of the database to a location outside the Notes data directory and then deleting the original database (CD).

2.2.1.3 Backup

This action is used to create a media backup of a transaction logged database while it is online. This action uses only a Copy primitive action (C).

2.2.1.4 Delete

This action is used to safely delete a transaction logged database while it is online. This uses the Delete primitive action (D).

2.2.2 Source Database

Supply the name of the database that is to be processed, the name should be supplied with the path relative to the Notes Data Directory on the current server. This parameter is mandatory.

2.2.3 Target Directory

If the action that is selected invokes a Copy (C) primitive action then this parameter determines the directory (usually outside the Notes Data Directory) where the database(s) will be copied to.

2.3 Error Recovery Options

These options will determine how the move engine will respond to particular errors.

2.3.1 Allow Retries

This parameter is mandatory but defaults to “No”. If set to “Yes” then in the case of an unrecoverable error condition the transaction will be reset and tried again at a later time.

2.3.2 Maximum Number of Retries

If the “Allow Retries” is set to “Yes” then this parameter determines how many time the transaction will be re-tried before being flagged as a permanent error.

2.3.3 Restart Options

If the “Allow Retries” option is selected then these settings determine what actions will be taken before the re-tried copy operation is attempted.

2.3.3.1 Delete Database

This setting causes the target database to be deleted before attempting the requested operation.

2.3.3.2 Fixup Source

This setting will cause a fixup to be performed on the source database before starting the requested operation. In certain cases this setting may be automatically set by error detection mechanisms in the move engine.

2.4 Generic Transaction Options

The following settings do not affect the requested operation itself but are related to the logistics of executing the transaction itself.

2.4.1 Status

This parameter determines the current state of the transaction in the processing cycle. Refer to the section “Transaction Workflow” for more details.

2.4.2 Transaction Request ID

This parameter assigns a unique transaction identifier to each transaction that passes through the processing cycle. For manually created transactions this is set to a value computed by the @Unique formula language function. Refer to the section on “Programmatic Interfaces” for more details on how this item is used. Refer to the section on “Feeder Transactions” for more details on automatically generated transaction request IDs.

2.4.3 Approval Status

Transactions that are in the “NEW” transaction state are not available for processing until the approval status is set to “Approved”. This provides a simple workflow which can be useful when dealing with source and target servers that exist in different administrative or organisational domains.

Transactions that require approval are held in a separate queue and can be selected and approved “in bulk”.

2.4.4 Urgent Flag

Transactions can be selected and marked as being urgent, this moves the transactions to the head of the queue of transactions that are ready for execution.

2.5 Scheduling Options

Transactions may be scheduled for repeated or one-off execution. The following parameters are used to control individual transaction scheduling.

2.5.1 First Run Time

Supply the date and time when the transaction should be run for the first or only time.

2.5.2 Repeat

Specify when the transaction should be repeated. A value of “Never” causes the transaction to be run only a single time. “Daily”, “Weekly” or “Monthly” indicate that after the intial run the transaction will be run again at the indicated interval. The time supplied in the “First Run Time” setting will be the target time of day when the transaction will run again.

3. Feeder Transactions

3.1 General

“Feeder” transactions are single transactions that can be submitted to QMove for processing actions that apply to a collection of databases rather than a single database. The simplest example of a collection of databases is a directory. Rather than having to supply an individual transaction for each database in a directory you can instead supply just the directory name. QMove will recognise such a transaction and process each database in the directory as if a single database had been specified in each transaction.

3.1.1 Source Selection

The largest practical unit of source selection that you can apply is a Server. Specify an asterisk (*) in the source database field on a transaction to select every database on the current server for processing. There are not many situations that lend themselves to processing complete servers, taking backup snapshots of test servers for use in regression testing is about the only valid situation that we have come across in the wild.

The next unit of source selection that can be applied is multiple directories that match a pattern. Specify a directory name containing wildcard characters in the source database field on a transaction to select every directory within the Notes Data Directory that matches the pattern and every database within those directories that are selected. For example supplying a value of “mail?” for the source database would copy all databases in the “mail1”, “mail2”, “mail3” etc. directories. The selection of source databases will recurse into the sub-directories of the selected directories.

Specifying a directory name without any wildcards would result in all of the databases within the directory and any sub-directories and their databases to be selected for processing.

Further refinement can be made to selection, for instance, specifying “mail1\a*.nsf” would select all of the databases that start with the letter “a” from the “mail1” directory.

3.1.2 Action and Error Processing Options

All options specified on the Feeder transaction will be used for each of the individual database processing operations.

3.1.3 Method of Operation

When a Feeder transaction is executed then each database that matches the source selection is located and a new QMove transaction is built, copying all of the options from the Feeder transaction. The new transaction is saved in the control database and will make its way up the ready queue until it is executed. Database move transactions that are generated from a “Feeder” transaction are marked as urgent so that they will execute soon after the “Feeder” transaction that generated them and before any other “Feeder” transactions that are on the ready queue.

4. Transaction Workflow

This section describes the different states that a transaction moves through during normal operation and fault handling.

4.1 New Transaction

When a transaction is created it is assigned a status of “NEW”, assuming that the approval flag has taken the default setting of “Not Approved” then the transition will sit in the “To Be Approved” queue. The transaction will remain in this queue indefinitely until it is approved. Multiple transactions can be selected in the “To Be Approved” queue and approved using the action at the top of the view.

4.2 Approved Transaction

When a NEW transaction is approved it is moved to the “Ready” queue. This queue is maintained in the order that transactions are created, however the Urgent Flag can be used to move individual transactions to the head of the queue.

The “Ready” queue is continuously monitored by the QMove server add-in task, as soon as the add-in is in a state where it can run another transaction it will select the transaction at the head of the queue and start to execute it.

4.3 In Progress Transaction

As soon as the QMove server add-in task starts to execute a transaction then it is marked “IN PROGRESS”.

Should the server add-in task or indeed the server fail catastrophically then any transactions that were executing at the time of the failure (i.e. marked “IN PROGRESS”) will be automatically restarted.

The transaction will remain in the “IN PROGRESS” state until it completes or fails.

4.4 Completed Transaction

When a transaction completes all process steps without any faults being detected it is changed to the “COMPLETED” state. This is a terminal state the transaction makes no further transitions.

4.5 Error Transaction

If the transaction experienced some kind of fault during execution and the “Allow Retries” option is **not** set to “Yes” on the transaction then the transaction is marked as an “ERROR” and moved to the errors queue. This is a terminal state the transaction makes no further transitions.

4.6 Retried Transaction

If the transaction experienced some kind of fault during execution and the “Allow Retries” option is set and the retry count for the transaction has not exceeded the Maximum Number of Retries then the transaction will be retried. One copy of the failed transaction has its state set to “RETRIED” and moved to the retried queue, this copy of the transaction holds the log and processing information from the failed execution attempt. A second fresh copy is put on the delayed queue.

4.7 Delayed Transaction

If a transaction experienced a fault but was eligible to be retried then it is marked as “DELAYED” and moved to the delayed transaction queue. The QMove server add-in tasks monitors the delayed transaction queue any time that it has no work available on the transaction ready queue and will execute transactions from that queue providing that they have been on the queue for a minimum of 15 minutes.

5. Programmatic Interface

It is a relatively trivial task to interface the QMove control databases to other applications. There are minimal requirements for creating a new transaction and a view is provided for monitoring the progress of individual transactions.

5.1 Creating New Transactions

The following fields should be set as indicated as a minimum to form a valid QMove transaction.

- Form – “QMTX”.
- Status – “NEW”.
- Approved – “Yes” unless you require the transaction to be manually approved in the control database before it is executed in which case set it to “No”.
- TransReqID – This is the transaction ID and should be set to a unique value on each transaction. **Hint**, you can set this to the UNID (@DocumentUniquID) of the source transaction to provide an easy reference between the application and the control database.
- OperationMode – Set this to a combination of the primitive operations that are to be carried out (C) copy, (D) delete, (L) link. So a “move” request would have “CDL” specified, a “retire” request would have “CD” specified, a “backup” request would have “C” specified and a “delete” request would have “D” specified.
- SourceDatabase – The name of the database to be moved.
- TargetDirectory – If the operation is a “move” operation then set this to the name of the directory that the database(s) are to be moved to.

The following fields can optionally be set to change the processing options of the transaction.

- Urgent – set the value to “Yes” or “No” to affect the urgency of the transactions.
- AllowRetry – set the value to “1” to permit the transaction to be retried if it fails.
- RetryLimit – set this value to a string containing the number of times that the transactions can be retried.
- PreProcessActions – set this value to a string containing the sum of any required actions
  - 1 – Delete the target database before processing.

5.2 Monitoring Transaction Progress

Use the “LookupByTransactionID” view in the database to track the progress of individual transactions. The view is keyed on the Transaction ID. Applications should normally check the “Status” field on an individual transaction to determine the current state. The following states are regarded as being terminal and should be acted on by the application.

- “COMPLETED” – the transaction has been executed and has completed processing without any errors.
- “ERROR” – the transaction has failed, it may have been retried a number of times before reaching this state.

All other values of the transaction status field should be considered as transient and should not be acted on by the application.

If permitted and a transaction fails and is retried then there will be more than one transaction document in the control database with the same transaction ID. Refer to the “Transaction Workflow” section of this document for details.

6. Installing QMove

6.1 Building the QMove Server Add-In Task

The QMove server add-in task is built on the Domino eXplorer Tools DXCommon kernel and the Notes C API, you need to download and install these before you can build the QMove application.

6.1.1 Reference Environments

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

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 8.5.

Execution Environment:

Windows Standard Server 2008 R2 (32 bit).

Domino Server 8.5.1 FP3.

Note: Execution environments from Domino 6.5.x through 8.5.x are regularly used.

Linux:

Build Environment:

Gcc Version: 4.1.2 for i386-redhat-linux.

Running on Redhat Linux 2.6.18-238.12.1.el5PAE #1 SMP Sat May 7 20:37:06 EDT 2011 i686 i686 i386 GNU/Linux

Notes API Version 8.5

Execution Environment:

Redhat Linux 2.6.18-238.12.1.el5PAE #1 SMP Sat May 7 20:37:06 EDT 2011 i686 i686 i386 GNU/Linux

Domino Server 8.5.1 FP3.

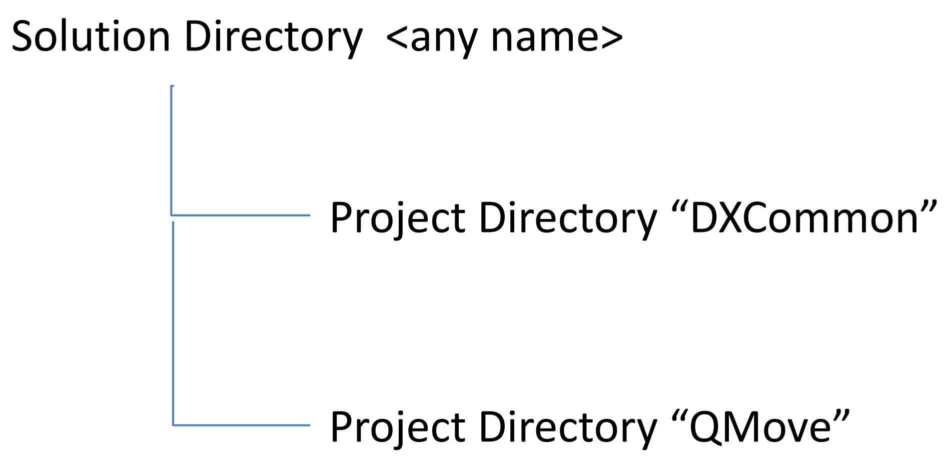
Note: Execution environments from Domino 7.0.x through 8.5.x are regularly used.

6.1.2 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.

6.1.3 Directory Structure

For both Windows and Linux DXTools uses a reference development directory structure based on the Visual Studio structure.


Solution Directory <any name>
Project Directory “DXCommon”
Project Directory “QMove”

In Visual Studio the DXCommon project directory should have the “Do Not Build” property set.

In both the Windows and Linux environments it is possible to use a symbolic link for the ”DXCommon” directory. This is a common deployment pattern for development environments where different versions of an API might need to be supported.

6.1.4 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
```

6.1.5 *Installing the QMove Sources*

Windows:

The QMove sources are supplied as a zipped archive (.zip). Create an empty project called “QMove” 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.

Header Files

AppCommandHandler.h

AppRunSettings.h

AppTransactionHandler.h

QFeedRequest.h

QMFeeder.h

QMFeedProcessor.h

QMove.h

QMover.h

QMoveRequest.h

Source Files

AppCommandHandler.cpp

AppRunSettings.cpp

AppTransactionHandler.cpp

QFeedRequest.cpp

QMFeeder.cpp

QMFeedProcessor.cpp

QMove.cpp

QMover.cpp

QMoveRequest.cpp

Linux:

The QMove sources are supplied as a gzipped archive (.tar.gz). Create the “QMove” project directory within the <solution directory> unpack the contents of the archive into that directory.

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

6.1.6 *Build Settings*

Windows:

Add each source and header file that is used from the DXCommon kernel to the QMove project. It is convenient to add these source and header files into subsets that can then be copied into other projects, in Visual Studio 2005 these collections are referred to as “filters”. To create a filter right-click on either the “Header Files” folder or the “Source Files” folder and select “Add” then “New Filter”. The following filters are convenient to use in the QMove project “Mover”, “Explorer” and “Runtime” these filters should be created in both the “Source Files” and “Header Files” folders. To populate the individual filter right-click on the filter then select “Add” then “Existing Item” navigate to the required source or header file(s), select the file(s) and click the “Add” button. The contents of each filter are listed below.

Header Files\Mover

DXCommon\DBC\DbMover.h

DXCommon\DBC\MoveRequest.h

DXCommon\DBC\PartMoverequest.h

Header Files\Explorer

DXCommon\MTDX\DominoExplorer.h

DXCommon\MTDX\DXDbScanner.h

DXCommon\MTDX\DXDirScanner.h

DXCommon\MTDX\DXFilter.h

DXCOmmon\MTDX\DXDXReporter.h

DXCommon\MTDX\DXRequest.h

DXCommon\MTDX\DXServerScanner.h

DXCommon\MTDX\DXSpider.h

DXCommon\MTDX\DXSStash.h

Header Files\Runtime

DXCommon\APIPackages.h

DXCommon\MTX\CommandHandler.h

DXCommon\DXException.h

DXCommon\DXGlobals.h

DXCommon\ElapsedTimer.h

DXCommon\Debug\Helper.h

DXCommon\MTX\MTExecutive.h

DXCommon\Platform\NotesBase.h

DXCommon\Platform\PlatBase.h

DXCommon\Threads\Runnable.h

DXCommon\RunSettings.h

DXCommon\Threads\ThreadDispatcher.h

DXCommon\Threads\ThreadManager.h

DXCommon\Threads\ThreadManagerPolicy.h

DXCommon\Threads\ThreadMonitor.h

DXCommon\Threads\ThreadScheduler.h

DXCommon\Threads\ThreadStructs.h

DXCommon\MTX\TransactionHandler.h

DXCommon\MTX\TransactionQueue.h

DXCommon\Threads\WorkerThread.h

Source Files\Mover

DXCommon\DBC\DbMover.cpp

DXCommon\DBC\MoveRequest.cpp

DXCommon\DBC\PartMoverequest.cpp

Source Files\Explorer

DXCommon\MTDX\DominoExplorer.cpp

DXCommon\MTDX\DXDbScanner.cpp

DXCommon\MTDX\DXDirScanner.cpp

DXCommon\MTDX\DXFilter.cpp

DXCOmmon\MTDX\DXDXReporter.cpp

DXCommon\MTDX\DXRequest.cpp

DXCommon\MTDX\DXServerScanner.cpp

DXCommon\MTDX\DXSpider.cpp

DXCommon\MTDX\DXSStash.cpp

Source Files\Runtime

DXCommon\APIPackages.cpp

DXCommon\MTX\CommandHandler.cpp

DXCommon\DXException.cpp

DXCommon\ElapsedTimer.cpp

DXCommon\Debug\Helper.cpp

DXCommon\MTX\MTExecutive.cpp

DXCommon\Threads\Runnable.cpp

DXCommon\RunSettings.cpp

DXCommon\Threads\ThreadDispatcher.cpp

DXCommon\Threads\ThreadManager.cpp

DXCommon\Threads\ThreadManagerPolicy.cpp

DXCommon\Threads\ThreadMonitor.cpp

DXCommon\Threads\ThreadScheduler.cpp

DXCommon\MTX\TransactionHandler.cpp

DXCommon\MTX\TransactionQueue.cpp

DXCommon\Threads\WorkerThread.cpp

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 (/MDd)
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

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.

/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.

Linux:

A makefile is supplied in the source distribution of QMove. The makefile is listed below along with any specific notes. The Makefile supports the following invocation models.

make QMove

This form of the command will build any object modules that are out of date and re-link the executable.

make rebuild QMove

This form of the command will force a rebuild of all object modules and re-link the executable.

make rebuild QMove BV=DBG

This form of the command will force a rebuild of all object modules with the _DEBUG define set and will re-link the executable.

```
#
#   Build for QMove 1.4.2 -- Build: 39 -- DXCommon 3.12.0
#
#   Optional targets:
#
#   rebuild      -   force a complete rebuild of the target
#
#   macros:
#
#   BV=DBG       -   Builds the DEBUG variant of the target
#
```

```
TARGET = QMove

#  Define the primary source files

SOURCES = $(TARGET).cpp
SOURCES += AppRunSettings.cpp
SOURCES += AppCommandHandler.cpp
SOURCES += AppTransactionHandler.cpp
SOURCES += QFeedRequest.cpp
SOURCES += QMFeeder.cpp
SOURCES += QMFeedProcessor.cpp
SOURCES += QMover.cpp
SOURCES += QMoveRequest.cpp
HEADERS = $(TARGET).h
HEADERS += AppRunSettings.h
HEADERS += AppCommandHandler.h
HEADERS += AppTransactionHandler.h
HEADERS += QFeedRequest.h
HEADERS += QMFeeder.h
HEADERS += QMFeedProcessor.h
HEADERS += QMover.h
HEADERS += QMoveRequest.h

#  Define the D/B Mover modules to build

MOVE_SOURCES = ../DXCommon/DBC/MoveRequest.cpp
MOVE_SOURCES += ../DXCommon/DBC/DbMover.cpp
MOVE_SOURCES += ../DXCommon/DBC/PartMoveRequest.cpp
MOVE_HEADERS = ../DXCommon/DBC/MoveRequest.h
MOVE_HEADERS += ../DXCommon/DBC/DbMover.h
MOVE_HEADERS += ../DXCommon/DBC/PartMoveRequest.h

#  Define Core modules to build

CORE_SOURCES = ../DXCommon/APIPackages.cpp
CORE_SOURCES += ../DXCommon/DXException.cpp
CORE_SOURCES += ../DXCommon/ExecEnvironment.cpp
CORE_SOURCES += ../DXCommon/ElapsedTimer.cpp
CORE_SOURCES += ../DXCommon/RunSettings.cpp
CORE_HEADERS = ../DXCommon/APIPackages.h
CORE_HEADERS += ../DXCommon/DXException.h
CORE_HEADERS += ../DXCommon/ExecEnvironment.h
CORE_HEADERS += ../DXCommon/ElapsedTimer.h
CORE_HEADERS += ../DXCommon/RunSettings.h
CORE_HEADERS += ../DXCommon/DXGlobals.h

#  Define the Multi-Threaded Core modules to build

MTCORE_SOURCES += ../DXCommon/MTX/CommandHandler.cpp
MTCORE_SOURCES += ../DXCommon/MTX/MTExecutive.cpp
MTCORE_SOURCES += ../DXCommon/MTX/TransactionHandler.cpp
MTCORE_SOURCES += ../DXCommon/MTX/TransactionQueue.cpp
MTCORE_SOURCES += ../DXCommon/Threads/ThreadManagerPolicy.cpp
MTCORE_SOURCES += ../DXCommon/Threads/ThreadManager.cpp
MTCORE_SOURCES += ../DXCommon/Threads/ThreadScheduler.cpp
MTCORE_SOURCES += ../DXCommon/Threads/ThreadDispatcher.cpp
MTCORE_SOURCES += ../DXCommon/Threads/ThreadMonitor.cpp
MTCORE_SOURCES += ../DXCommon/Threads/WorkerThread.cpp
MTCORE_SOURCES += ../DXCommon/Threads/Runnable.cpp
MTCORE_HEADERS = ../DXCommon/MTX/CommandHandler.h
MTCORE_HEADERS += ../DXCommon/MTX/MTExecutive.h
MTCORE_HEADERS += ../DXCommon/MTX/TransactionHandler.h
MTCORE_HEADERS += ../DXCommon/MTX/TransactionQueue.h
MTCORE_HEADERS += ../DXCommon/Threads/ThreadManagerPolicy.h
MTCORE_HEADERS += ../DXCommon/Threads/ThreadManager.h
MTCORE_HEADERS += ../DXCommon/Threads/ThreadScheduler.h
MTCORE_HEADERS += ../DXCommon/Threads/ThreadDispatcher.h
MTCORE_HEADERS += ../DXCommon/Threads/ThreadMonitor.h
MTCORE_HEADERS += ../DXCommon/Threads/WorkerThread.h
MTCORE_HEADERS += ../DXCommon/Threads/Runnable.h

#  Define the Domino Explorer modules to build

EXPLORER_SOURCES = ../DXCommon/MTDX/DominoExplorer.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXDbScanner.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXDirScanner.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXFilter.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXReporter.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXRequest.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXServerScanner.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXSpider.cpp
EXPLORER_SOURCES += ../DXCommon/MTDX/DXSStash.cpp
EXPLORER_HEADERS = ../DXCommon/MTDX/DominoExplorer.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXDbScanner.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXDirScanner.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXFilter.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXReporter.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXRequest.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXServerScanner.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXSpider.h
EXPLORER_HEADERS += ../DXCommon/MTDX/DXSStash.h

#  Define the Object Lists

OBJECTS = $(TARGET).o
OBJECTS += AppRunSettings.o
OBJECTS += AppCommandHandler.o
OBJECTS += AppTransactionHandler.o
OBJECTS += QFeedRequest.o
OBJECTS += QMFeeder.o
OBJECTS += QMFeedProcessor.o
OBJECTS += QMover.o
OBJECTS += QMoveRequest.o
MOVE_OBJECTS = MoveRequest.o
MOVE_OBJECTS += DbMover.o
MOVE_OBJECTS += PartMoveRequest.o
CORE_OBJECTS = APIPackages.o
CORE_OBJECTS += DXException.o
CORE_OBJECTS += ExecEnvironment.o
CORE_OBJECTS += ElapsedTimer.o
CORE_OBJECTS += RunSettings.o
MTCORE_OBJECTS = CommandHandler.o
MTCORE_OBJECTS += MTExecutive.o
MTCORE_OBJECTS += TransactionHandler.o
MTCORE_OBJECTS += TransactionQueue.o
MTCORE_OBJECTS += ThreadManagerPolicy.o
MTCORE_OBJECTS += ThreadManager.o
MTCORE_OBJECTS += ThreadScheduler.o
MTCORE_OBJECTS += ThreadDispatcher.o
MTCORE_OBJECTS += ThreadMonitor.o
MTCORE_OBJECTS += WorkerThread.o
MTCORE_OBJECTS += Runnable.o
EXPLORER_OBJECTS = DominoExplorer.o
EXPLORER_OBJECTS += DXDbScanner.o
EXPLORER_OBJECTS += DXDirScanner.o
EXPLORER_OBJECTS += DXFilter.o
EXPLORER_OBJECTS += DXReporter.o
EXPLORER_OBJECTS += DXRequest.o
EXPLORER_OBJECTS += DXServerScanner.o
EXPLORER_OBJECTS += DXSpider.o
EXPLORER_OBJECTS += DXSStash.o

#  Define the build options

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

#  Rules to build DEBUG or release targets

$(TARGET): $(OBJECTS) $(MOVE_OBJECTS) $(CORE_OBJECTS) $(MTCORE_OBJECTS) $(EXPLORER_OBJECTS)
        $(CC) $(LINKOPTS) $(OBJECTS) $(MOVE_OBJECTS) $(CORE_OBJECTS) $(MTCORE_OBJECTS)
$(EXPLORER_OBJECTS) -L$(NOTESDIR) -Wl,-rpath-link $(NOTESDIR) $(LIBS)
$(OBJECTS): $(SOURCES) $(HEADERS)
ifeq ($(BV), DBG)
        $(CC) $(CCOPTS) $(DEFINES) -D_DEBUG -I$(INCDIR) $(SOURCES)
else
        $(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) $(SOURCES)
```

```
endif
$(MOVE_OBJECTS): $(MOVE_SOURCES) $(MOVE_HEADERS)
ifeq ($(BV), DBG)
	$(CC) $(CCOPTS) $(DEFINES) -D_DEBUG -I$(INCDIR) $(MOVE_SOURCES)
else
	$(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) $(MOVE_SOURCES)
endif
$(CORE_OBJECTS): $(CORE_SOURCES) $(CORE_HEADERS)
ifeq ($(BV), DBG)
	$(CC) $(CCOPTS) $(DEFINES) -D_DEBUG -I$(INCDIR) $(CORE_SOURCES)
else
	$(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) $(CORE_SOURCES)
endif
$(MTCORE_OBJECTS): $(MTCORE_SOURCES) $(MTCORE_HEADERS)
ifeq ($(BV), DBG)
	$(CC) $(CCOPTS) $(DEFINES) -D_DEBUG -I$(INCDIR) $(MTCORE_SOURCES)
else
	$(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) $(MTCORE_SOURCES)
endif
$(EXPLORER_OBJECTS): $(EXPLORER_SOURCES) $(EXPLORER_HEADERS)
ifeq ($(BV), DBG)
	$(CC) $(CCOPTS) $(DEFINES) -D_DEBUG -I$(INCDIR) $(EXPLORER_SOURCES)
else
	$(CC) $(CCOPTS) $(DEFINES) -I$(INCDIR) $(EXPLORER_SOURCES)
endif

#  Phony target for forcing a complete rebuild

.PHONY : rebuild
rebuild:
	-rm *.o
	-rm qmove
```

Notes:

There is currently little difference between the release and debug builds of the QMove application, the debugging helper that adds additional runtime debugging capabilities to the application is only available for the Windows builds. There is an enhancement request to port the debugging helper functionality to Linux.

The executable name is coerced to lower case “qmove”, this is a Domino convention.

6.1.7 Building and Deploying the Application

Windows:

Select “Build” and then “Build QMove”.

Copy the resulting executable (QMove.exe) and the associated Program Debug Database (QMove.pdb) to the Notes Executable directory on the server where you want to run the Add-In.

Linux:

From the QMove project directory issue the “make QMove” command or the “make rebuild QMove” or the “make rebuild QMove BV=DBG” according to the type of build that you require.

make QMove

This form of the command will build any object modules that are out of date and re-link the executable.

make rebuild QMove

This form of the command will force a rebuild of all object modules and re-link the executable.

make rebuild QMove BV=DBG

This form of the command will force a rebuild of all object modules with the _DEBUG define set and will re-link the executable.

Copy the resulting executable (qmove to the Notes Executable directory where you want to run the Add-In. 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 qmove

chgrp notes qmove

chmod 2555 qmove

6.2 Installing the QMove Control Database

The installation of the control database 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.

6.2.1 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.

For the QMove Control Database, use the following URL:

`http://hmnl.nl/HMNL/DX/VTTDepot.nsf/Payloads/qmove2.manifest/$File/manifest.xml`

7. Starting QMove on the Server

The QMove application can now be loaded on the server where it was installed. From a remote console session with the server issue the following command.

“load QMove <database name> <options>”

Where:

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

<options> are the command line options for the applications. (see below).

7.1.1 Command Line Options

The following command line options are available with QMove.

Option	Meaning
-V	Sets the logging level to verbose. This provides more detailed logging messages to be written to the application log.
-T[:nn]	Sets the logging level to trace, This is a diagnostic setting that provides detailed logging messages to the application log. The optional “nn” setting restricts tracing to a designated are of the DXCommon kernel code. Refer to the DXGlobals.h header file for values that this setting can take.
-D[:nn]	Sets the logging level to trace, This is a diagnostic setting that provides even more detailed logging messages to the application log. The optional “nn” setting restricts tracing to a designated are of the DXCommon kernel code. Refer to the DXGlobals.h header file for values that this setting can take.
-E	Sets console echo mode on. Application log entries are written to documents in the control database. When console echo mode is on then all application log messages are echoed to the Domino console and the Domino system log database.
-I:nnn	Sets the duration of a compulsory delay between transactions in seconds, the value can be between 1 and 600. This can be used to reduce the stress on a busy server caused by pinning of the transaction logs by QMove copy operations.
-B:nnn	Specifies the size of the transfer buffer to be used in copy operations, the value is the number of Kb that will be used. Specify a value between 4 and 1024.
-X:nn	Sets the number of transactions that can execute in parallel. This defaults to 1. Therefore, by default, QMove transactions will execute serially.

8. QMove Tell Commands

8.1 QMove Message Queues

While it is running certain aspects of QMove operation can be controlled by the use of “Tell” commands entered through the Domino console.

QMove is capable of running multiple instances on the same Domino server, each instance will have a separate message queue that it monitors for commands. When an instance of QMove is started it locates the lowest number available to use for the message queue name in the form “QMove”<suffix> where <suffix> is a digit that starts at 1. Therefore the first, or only, instance of QMOve can be addressed in the form “Tell QMove1 <command>”.

8.2 Commands

8.2.1 Quit

The quit command is not normally used, however, during a server shutdown the server issues this command on all message queues. QMove will respond to the command by shutting down.

8.2.2 Stop [now]

The stop command is used to shut down QMove in an orderly manner. Any transactions that are currently running will be completed, no new transactions are dispatched and the Server Add-In will shut down. Specifying the optional “now” parameter on the stop command causes QMove to fail any transactions that are currently running and then shut down in an orderly manner.

8.2.3 Abort

The abort command is an alias for the “stop now” command.

8.2.4 Suspend

The suspend command tells QMove to stop executing new transactions from the ready queue. Any transactions that are currently executing are completed, the Add-In task continues to run but will not process any transactions until the “resume” command is executed.

8.2.5 Resume

The resume command is the counterpart of the suspend command. The command only has any effect if the Add-In task is in the suspended state, then it causes the processor to resume processing transactions from the ready queue.

8.2.6 Verbose

The verbose command causes the logging mode of the processor to be switched to verbose mode, in this mode more detailed logging is made to the application log.

8.2.7 Loud

The loud command is an alias for the verbose command.

8.2.8 Terse

The terse command switches the logging mode of the processor back to normal mode, in this mode minimal logging is done to the application log.

8.2.9 Quiet

The quiet command is an alias for the terse command.

8.2.10 Echo [on|off]

The echo command without any parameters is the same as the “echo on” command it will cause all current application logging to be echoed to the Domino Server console and therefore the Domino log. The “echo off” command turns off the echo of application logging.

8.2.11 Noecho

The noecho command is an alias for “echo off”.

8.2.12 Trace [nnn]

The trace command sets the processor logging functions into trace mode. The number on the command designates a particular are of function to be traced. Refer to the DXGlobals.h header file for the different trace area specifications.

This command should only be used for problem diagnosis.

In this mode very detailed logging is produced in the application log.

8.2.13 Debug [nnn]

The debug command sets the processor logging functions into debug mode. The number on the command designates a particular are of function to be traced. Refer to the DXGlobals.h header file for the different trace area specifications.

This command should only be used for problem diagnosis.

In this mode even more detailed logging is produced in the application log.

8.2.14 Refresh

The refresh command causes the QMove processor to finish processing any transactions that are currently processing and then reset the processing environment to the default configuration and resume processing transactions.

8.2.15 Status

The status command causes the processor to display the current state of the processor and some volumetric information about how many transactions have been processed.

Sample output:

```
07/12/2010 08:51:50 CET: DXR0907I: Command received: status. [500]
07/12/2010 08:51:50 CET: QMV0201I: 1 transaction have been dispatched, 1 completed, 0 are
running, max concurrency is 1. [500]
07/12/2010 08:51:50 CET: QMV0208I: Transactions marked Completed: 0, Error: 1, Retried: 0,
Delayed: 0. [500]
```

8.2.16 Stats [thread|debug]

The stats command causes a number of current values of statistics from the DXCommon kernel to be written to the applications log. The thread parameter on the command adds certain additional “per thread” statistics to be output. The debug operand on the command causes the “per thread” statistics to be included along with more details. Understanding these statistics is beyond the scope of this document, refer to the documents about the architecture of the DXCommon kernel to gain insight into the meaning of these statistics.

8.2.17 Panic [message]

The panic command will trigger an NSD exception from within the processor. This is an extreme diagnostic aid as it will cause an NSD of the entire server. The optional message is recorded in the log and in the memory displayed in the NSD dump.

8.2.18 Maxtrans nn

The maxtrans command causes the processor to change the number of transactions that can be executed in parallel to be changed to the value supplied on the command.

8.3 Debugging Commands

The following commands are **ONLY** available in the debug compiled version of the QMove executable.

8.3.1 Memory

The memory command shows a report on current memory usage by the application, these statistics are reported to the server console and the application log.

8.3.2 Dump

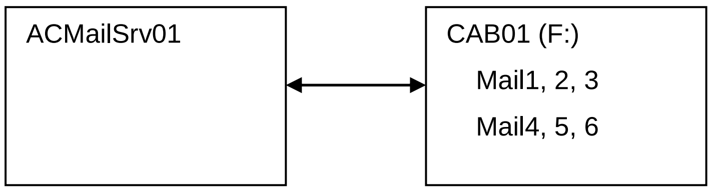
The dump command causes the processor to generate a Windows Core Minidump of the application. The application continues to execute so the command can be issued a number of times during an execution of the application. The contents of the dump can be investigated using the standard Windows debugging tools (e..g. windbg).

9. Common Usage Scenarios

9.1 Storage Redistribution

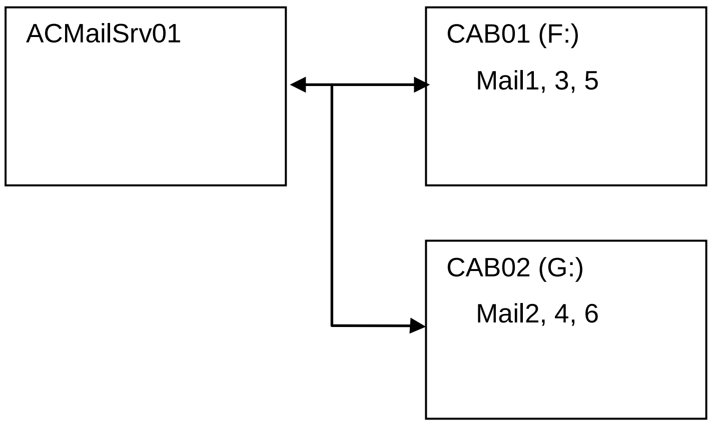
In this scenario the “ACME Corporation” has a problem the disk storage on one of their mail servers is up to 90% utilisation. The disk storage is in a single Direct Attached cabinet which has reached capacity and cannot be expanded. They will add a new cabinet to the existing server and would then like to move three out of their six mail directories from the original cabinet onto the new cabinet. Users expect this change to be implemented without any disruption to the mail service.

As is:



Notes Data Directory: F:\NOTESDATA

To be:



Once the new cabinet (CAB02) has been attached to the server then create a data directory G:\NOTESDATA on it. You can now create three QMove (feeder style) transactions one each for the mail2, mail4 and mail6 sub-directories and specifying the “move” operation and a target directory of G:\NOTESDATA\mail2, G:\NOTESDATA\mail4 and G:\NOTESDATA\mail6 respectively.

The three transactions will migrate the mail files into the “to be” configuration without any interruption to the mail service.

After the three transactions have completed the mail<n> directories on the F: drive will contain a database link file for each mail file that was in the directory each link will point to the new location for the mail file (G:\NOTESDATA\mail<n>\<mailfile.nsf>).

It should be noted that if a new mail file is created in the mail2 sub-directory the physical database will be created in the mail2 directory on the F: drive. A small manual step can be undertaken to make it so that mail2 directory on the G: drive operates as a true directory, from the Administrator client delete all of the links in the mail2 directory and the mail2 directory itself then create a “Directory Link” called “mail2” that points to G:\NOTESDATA\mail2, from that point on new mail files created in the mail2 directory will be created on the G: drive.

9.2 Regular Media Backups

The mail server from the previous scenario has a “backup” storage device attached (using iSCSI) as the H: drive. A single “scheduled” transaction is setup to run weekly overnight on a Monday to execute a “backup” operation for the “mail1” directory directed at the H:\Backups\ACMailSrv01\mail1 directory. The transaction specifies pre-processing options to delete the target file before copying.

This will create new backup copy of each mail file in the mail1 sub-directory each Monday night. The backups will be on the H: drive and invisible to the Domino Server.

10. Tuning Considerations

10.1 Transaction Logs

The QMove engine uses the Domino “Backup” API set to accomplish the copy primitive operations. The backup API keep transaction log files “pinned” in memory for data that has not yet been written to databases that are being copied. On a server that is processing a large number of updates and is copying multiple databases at the same time it may be that the logging system is not able to dequeue log datasets fast enough. In these circumstances run only a single instance of QMove on the server and run a single transaction stream (-X:1) also increase the time delay between individual transactions using the –I:nnn command line switch to reduce stress on the transaction logging sub-system.