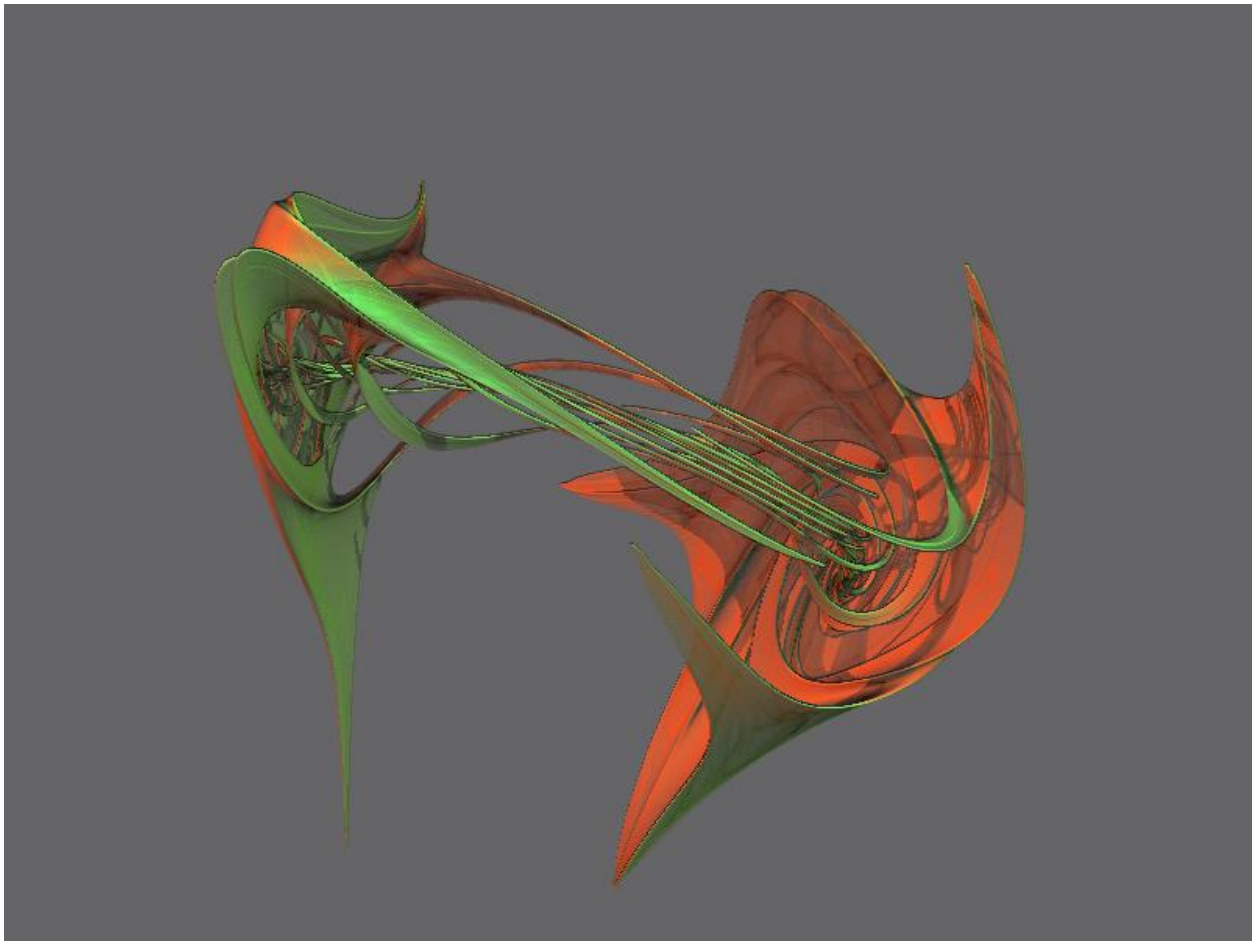
Domino eXplorer

Component Architecture 3



Author: Ian Tree
Owner: HMNL b.*v.*
Customer: Public
Status: Final
Date: 21/01/2015 14:30
Version: 3.14

Disposition: Open Source

Document Usage

Revision History

Date of this revision: 21/01/2015 14:30	Date of next revision *None*

Revision Number	Revision Date	Summary of Changes	Changes marked
3.12	02/02/12	Initial Base Version	No
3,12.0	30/03/12	QE Version	No
3.14.0	21/01/15	Updated for x64 support	No

Acknowlegements

Frontpiece Design was produced by the chaoscope application.

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.

Contents

1. Introduction to the DX Classes 4
2. Basic Run Time Initialisation 5
3. Multi-Threaded Service Initialisation 7
3.1 Initialise Pools 7
3.2 Start the Control Threads 8
3.3 Start the Worker Threads 10
4. Activate a Command Handler 11
5. Request Queue Element (RQE) Lifecycle 12
5.1 Posting a Request 12
5.1.1 The Request Owner 13
5.1.2 Request Priority 14
5.1.3 Constrained Multi Lane Scheduling 14
5.2 Thread Scheduling and Dispatch 16
5.3 Request Execution 17
5.4 Rejoining a Completed Request 17
5.4.1 Case #1: Request Found and Returned 17
5.4.2 Case #2: No Completed Requests are Available 19
5.4.3 Case #3: No More Requests Exist 21
6. Asynchronous Logging 23
7. Runnable Objects 25

1. Introduction to the DX Classes

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 2008 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.

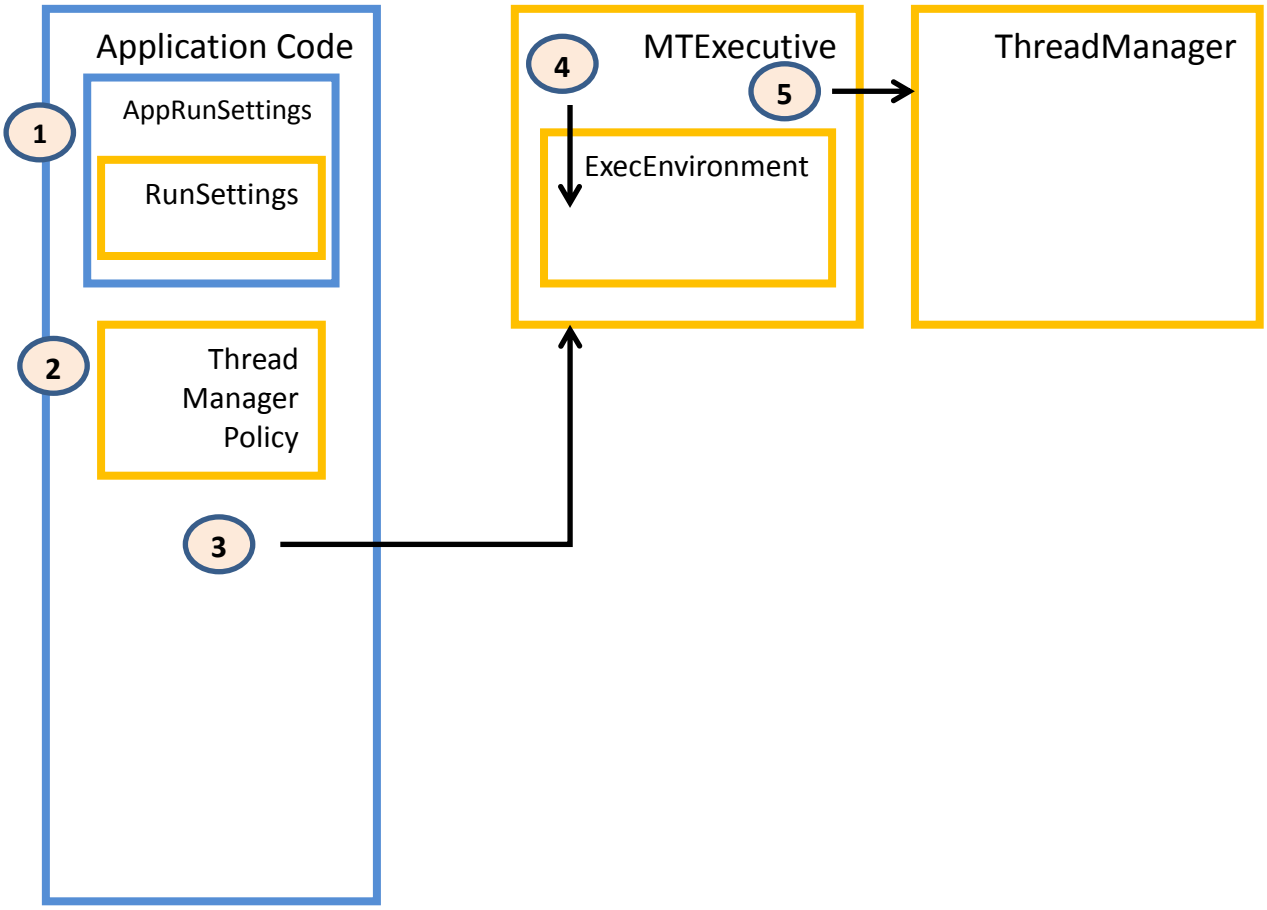
Standardised utilities have also been built around some of the functional DX classes, these are published as “DX Tools” and can save time by providing off-the-shelf processing to be incorporated into complex transformations that need high throughput rates.

DX consists of a set of “Kernel” classes and a collection of “Functional” classes. This document presents the gross architecture of the DX kernel and the functional classes.

The component architecture is examined by following typical execution sequences in a representative DX Tools application.

2. Basic Run Time Initialisation

This section looks at the start sequence in a DX application, the application in question is a multi-threaded server add-in task.



1

Once started the application creates and populates an “AppRunSettings” object the class extends the “RunSettings” class that defines the basic configuration of the application.

2

The application creates a “ThreadmanagerPolicy” object, this class defines the configuration of the multi-threaded kernel.

3

The application creates the Run Time (“MTExecutive”) object, the basic initialisation of the Run Time is performed in the constructor.

4

The constructor for the “MTExecutive” invokes the service initialisation in the ExecEnvironment base class, the service initialisation takes care of initialising the following.

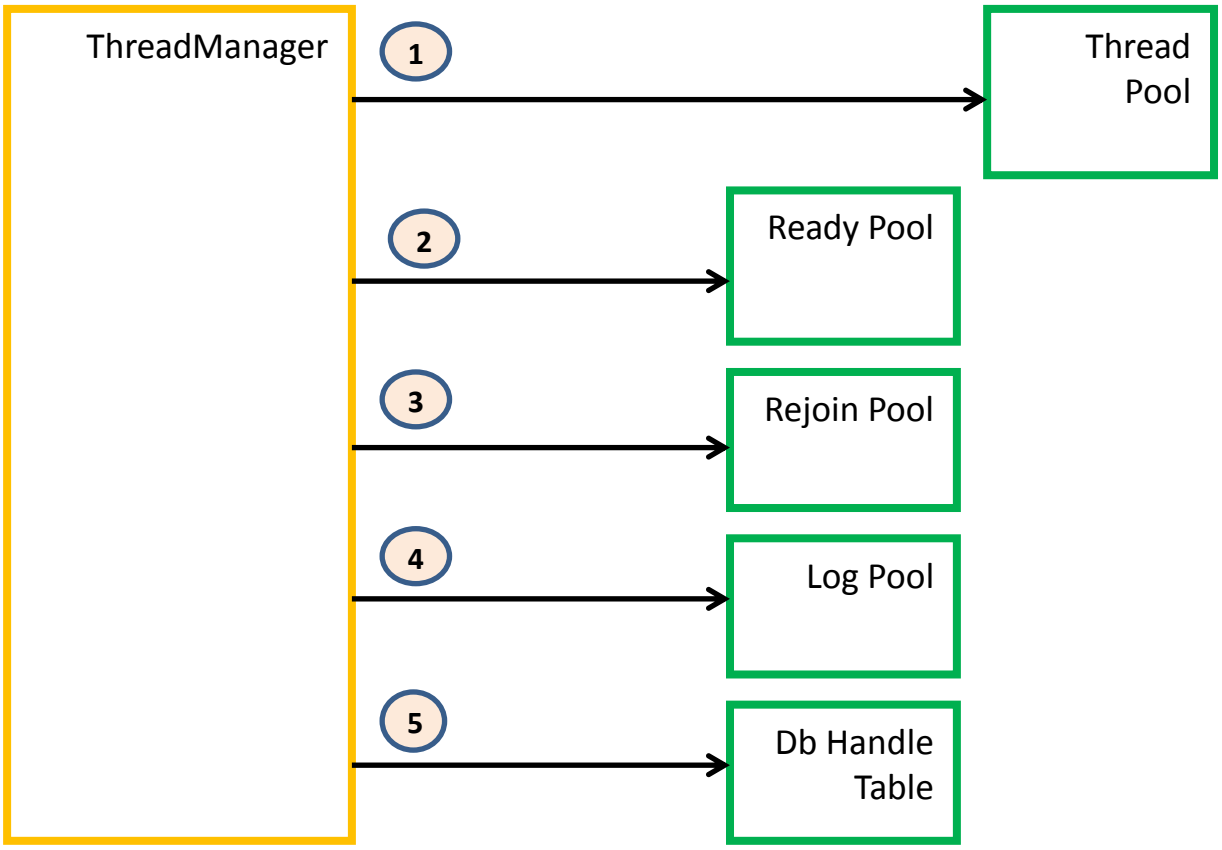
- The default elapsed timer.
- The Notes Run Time.
- Task descriptor and message queue (MQ) if the application is an add-in.
- The synchronous logging service.

The constructor then creates a “ThreadManager” singleton object and invokes the Start() function, this transitions to the next stage in the initialisation.

3. Multi-Threaded Service Initialisation

This phase of the initialisation transitions the kernel and services from a single-threaded mode of operation to a multi-threaded mode. All of these transitions are driven by the Thread Manager.

3.1 Initialise Pools



1

The Thread Manager allocates and initialises the Thread Pool, the Thread Pool contains a header area that has pointers to all of the important structures needed by the kernel, including pointers to the other allocated pools and stateful information about the kernel. The pool also contains an array of members each one representing a thread in the DX kernel, control threads and worker threads have a member in the Thread Pool.

2

Next the Thread Manager allocates and initialises the Ready Pool this pool contains Request Queue Elements (RQEs) that contain information on every request that is ready for execution by the worker threads.

NOTE: In the application code this pool is also referred to as the “Pending Pool”.

3

The Thread Manage allocates and initialises the Rejoin Pool, this pool is used to store Request Queue Elements (RQEs) that contain information about requests that have been executed and are waiting to rejoin their code stream.

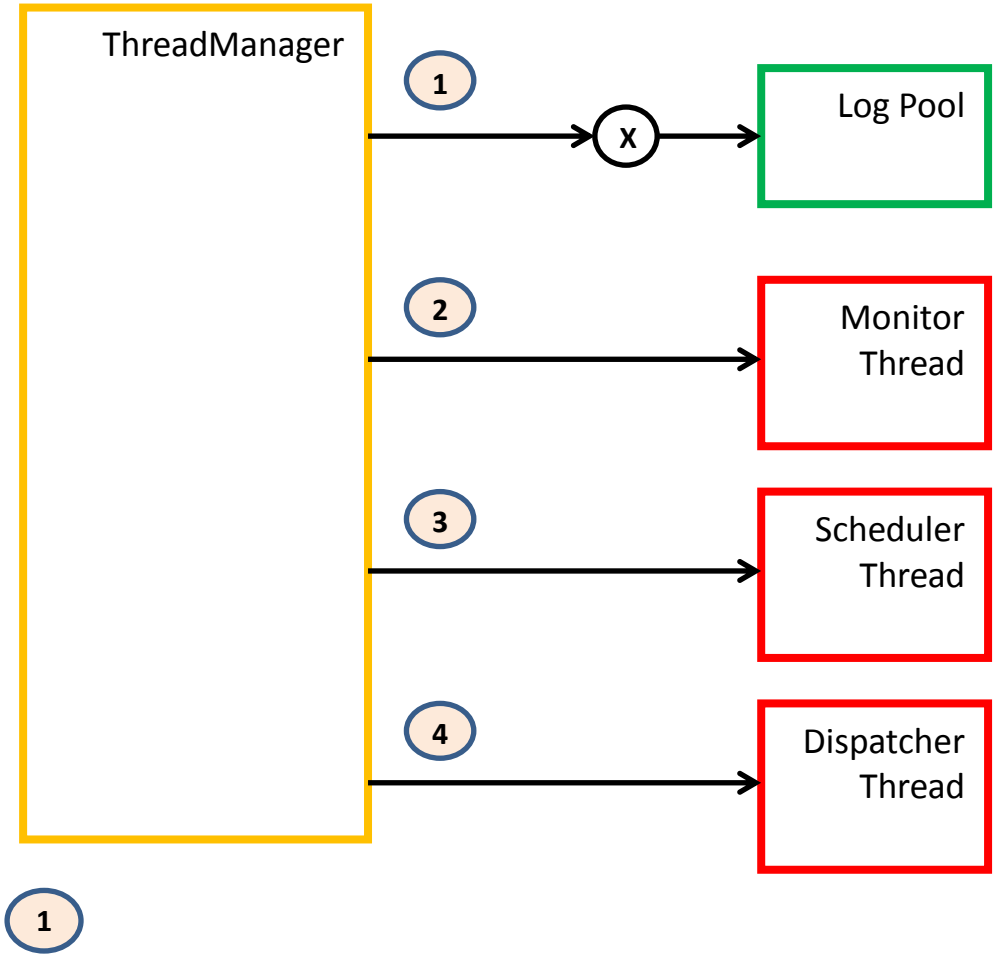
4

The Thread Manager allocates and initialises the Log Pool, this pool is used to store logging messages that are waiting to be written to the persistent log.

5

The Thread Manager allocates and initialises the Database Handle Table, this table is used to store the per thread database handles that are mapped onto a native database handle. The Notes code base only allows a single thread to use a database handle the kernel provides an on-demand mapping capability that will (re)open a new database handle for an individual thread and dispose of these handles when the native handle is closed.

3.2 Start the Control Threads



1

The Thread Manager switches the logging mode from synchronous mode to asynchronous mode. In asynchronous mode all logging messages are written to the log pool rather than being written directly to the persistent log.

The Thread Manager constructs a “ThreadMonitor” singleton and invokes the Start function on a new OS thread, the Thread Manager waits until the Monitor Thread has reached the “Running” state before proceeding. The Monitor Thread is responsible for the following functions of the multi-threaded kernel.

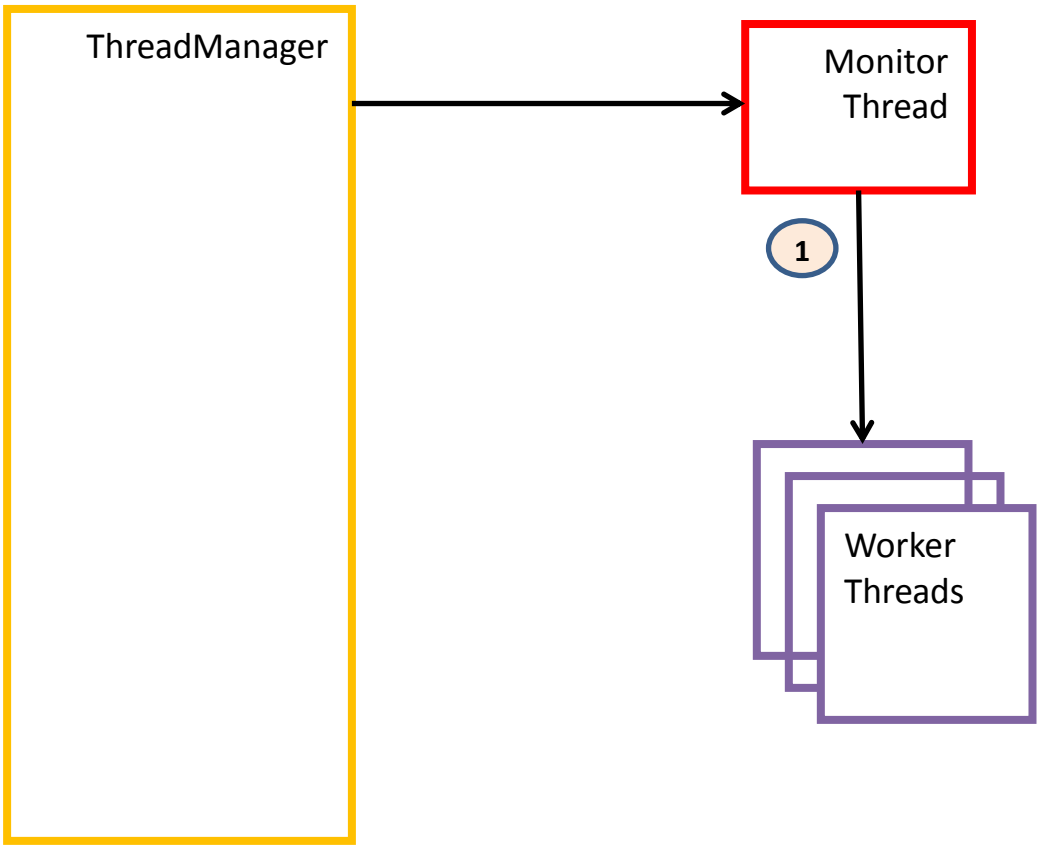
- Starting the worker threads.
- Asynchronous logging.
- Command handling.
- Instrumentation Package recording.
- Reviewing kernel settings.

The Thread Manager constructs a “ThreadScheduler” singleton and invokes the Start function on a new OS thread. The Scheduler Thread is responsible for adjusting the priority of requests that are in the “Ready Pool”.

The Thread Manager constructs a “ThreadDispatcher” singleton and invokes the Start function on a new OS thread. The Dispatcher Thread is responsible for the following kernel functions.

- Posting requests to the Ready Pool on behalf of client threads.
- Posting requests from the Ready Pool to an available worker thread in the Thread Pool.
- Polling the Rejoin Pool for completed requests on behalf of client threads.

3.3 Start the Worker Threads



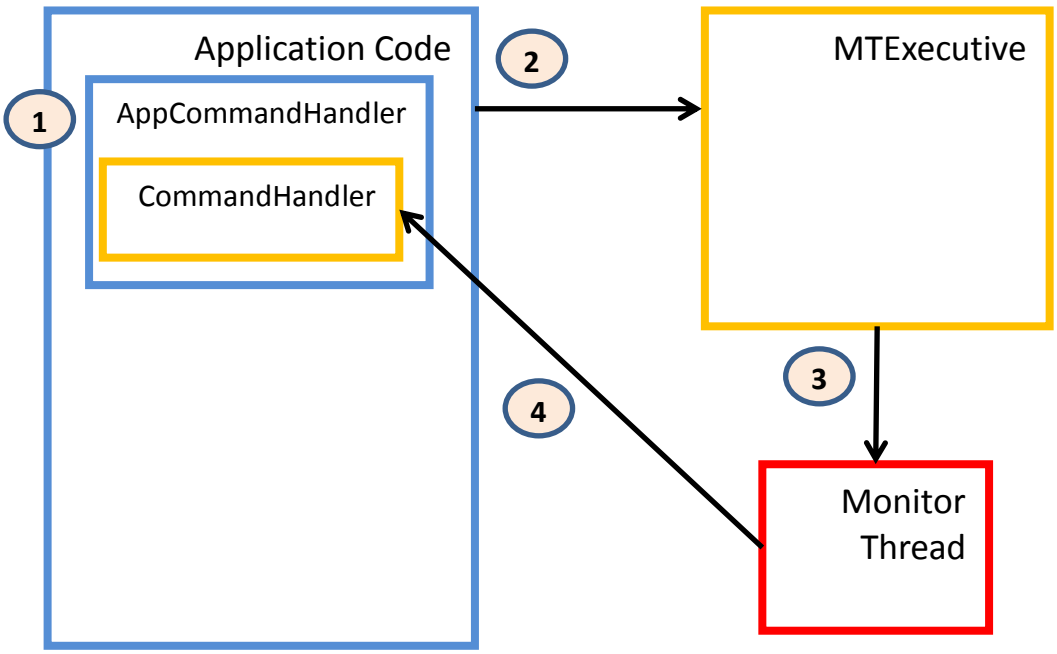
1

The monitor thread creates a unique “WorkerThread” object for each worker thread and invokes the Start function on a new OS thread.

NOTE: Threads are identified by a positive integer, the following describes the allocation of Thread IDs.

- Thread 0 – The main application thread.
- Thread 500 – The Monitor thread.
- Thread 600 – The Scheduler thread.
- Thread 700 – The Dispatcher thread.
- Thread 1 – <max threads> - The worker threads.

4. Activate a Command Handler



1

The application creates a new object that extends the CommandHandler class. The extension implements any custom commands and replaces or extends the functionality of any system commands.

2

The application calls the AttachCommandHandler API function passing the address of the Command Handler object.

3

The API sets the address of the Command Handler in the Monitor Thread.

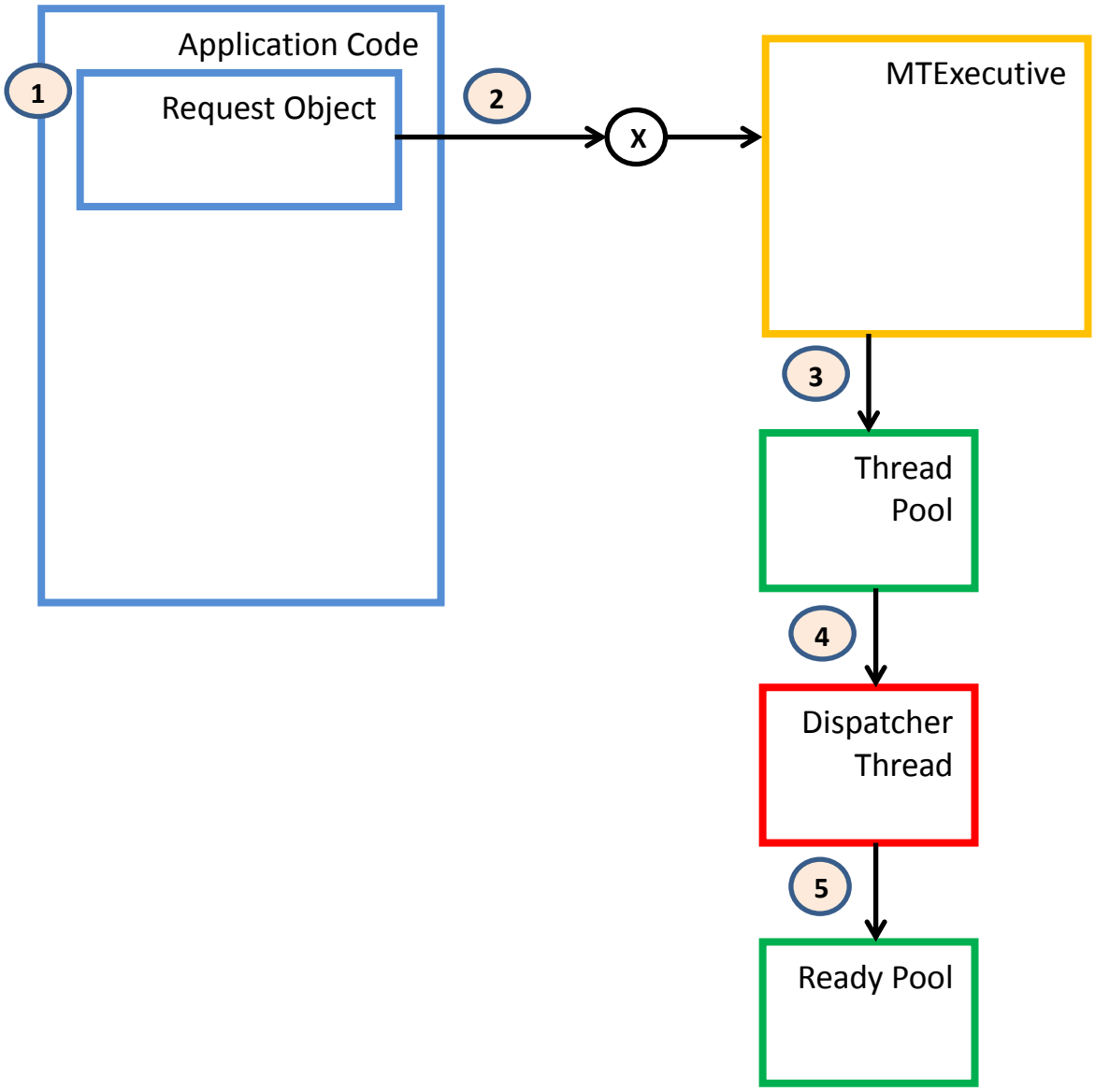
4

The Monitor Thread will invoke the Command Handler at regular intervals for it to check the Message Queue and to check for any timer driven automated commands.

5. Request Queue Element (RQE) Lifecycle

This section examines the posting, dispatching and rejoining of a Request.

5.1 Posting a Request



1

The application constructs the request object that is to be executed.

2

The application makes a call to the PostARequest API function. The call passes the following parameters.

- The address of the request object.
- The address of an object that implements the Runnable interface this object will execute the request.
- An arbitrary (void *) address that identifies the parent of the request (see later).
- The request priority (see later).
- Flags indicating the attributes of the request.

The call will block if the “Ready Pool” is already full of requests and the PXR_WAITIF_BUSY flag was set on the call.

The call will block again if Constrained Multi-Lane Scheduling (CMLS) is active and the requested lane is full and the PXR_WAITIF_BUSY flag was set on the call.

The call will block again if a previous posting of a request still has the request posting semaphore still set.

The PostARequest function builds the Request Queue Element (RQE), copies it into the Thread Pool member for the calling application thread and posts the semaphore to signal that a new RQE is available and block any subsequent calls until the RQE is moved to the “Ready Pool”. The function then returns to the caller.

```
//  Request Queue Element (RQE)
typedef struct {
       volatile UINT RQEState;                  //  State of the request
       UINT                RQEAttrs;            //  Attributes of this RQE
       int                 RQEPriority;         //  Priority
       int                 RQEInspectCount;     //  Inspection count
       void far            *pOObject;           //  Pointer to the Owner Object
       void far            *pXObject;    //  Pointer to the Executable Object
       void far            *pPObject;    //  Pointer to the Parameter Object
} RQElement;
```

The Dispatcher Thread periodically checks the Thread Pool Members, once it sees that a new RQE has been posted (semaphore is set) it continues with step 5.

The Dispatcher Thread finds a free RQE slot in the “Ready Pool” and moves the posted RQE to the free slot and clears the posting semaphore.

5.1.1 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 a 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.

5.1.2 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 level 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.

5.1.3 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”.

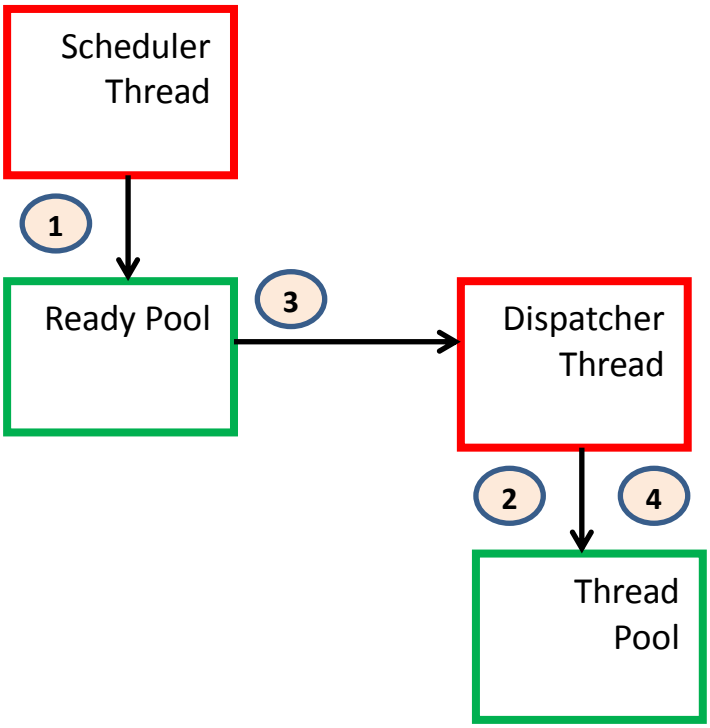
Constraints may be applied as a percentage of the threads in the thread pool that can be executing requests from levels 0, 1 and 2. These constraints are applied by setting the appropriate members in the ThreadManagerPolicy that used to initialise the kernel. The constraints not only apply to the worker 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.

5.2 Thread Scheduling and Dispatch



1

The Scheduler Thread periodically inspects every RQE in the Ready Pool and adjusts the priority of the request as appropriate. If the Thread Manager Policy has the Request Aging policy set (PRIO_POLICY_AGERQS) then the priority is increment (+1) on each scheduler interval, if additionally the Preferential Request Boosting policy is in effect and the particular request is marked for boosting then an additional increment of the priority is made by a value specified in the Thread Manage Policy.

2

The dispatcher thread monitors all of the Thread Pool Members locating worker threads that are now free to execute work. When the thread has one or more worker threads that can execute new requests and there are requests waiting to be executed in the Ready Pool then work is dispatched.

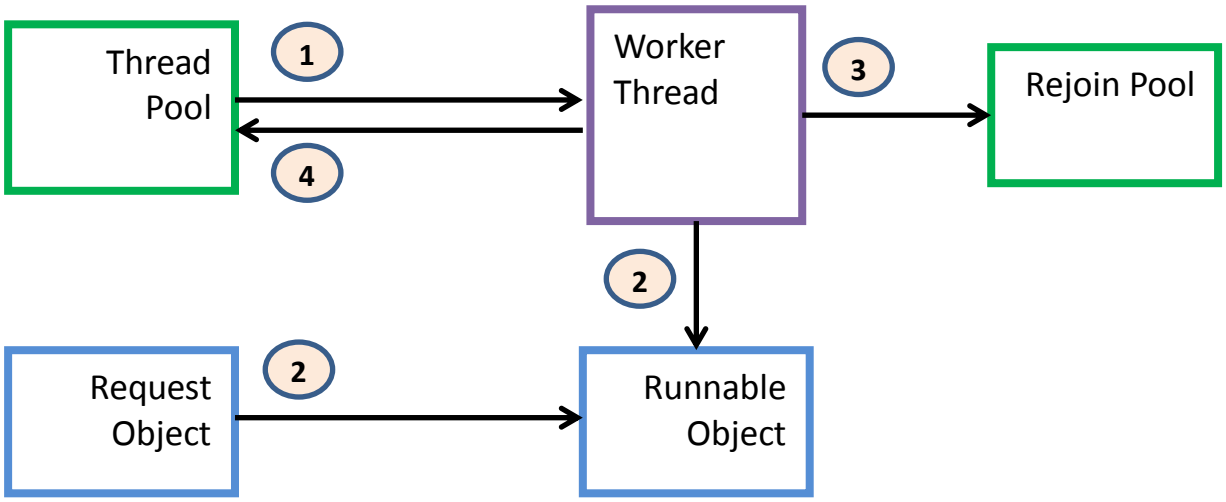
3

The dispatcher thread copies the RQE from the ready pool and marks the slot in the pool as free.

4

The Dispatcher Thread posts the RQE into the available Thread Pool Member and marks the member to show that there is work waiting to be executed.

5.3 Request Execution



1

Each Worker Thread monitors its own Thread Pool Member and on detecting that the Dispatcher has posted work for it to do it marks the thread state as Busy.

2

The Worker Thread then invokes the Runnable interface on the executor object via the ExecuteThis Request call, passing the Request object. The request is then executed.

3

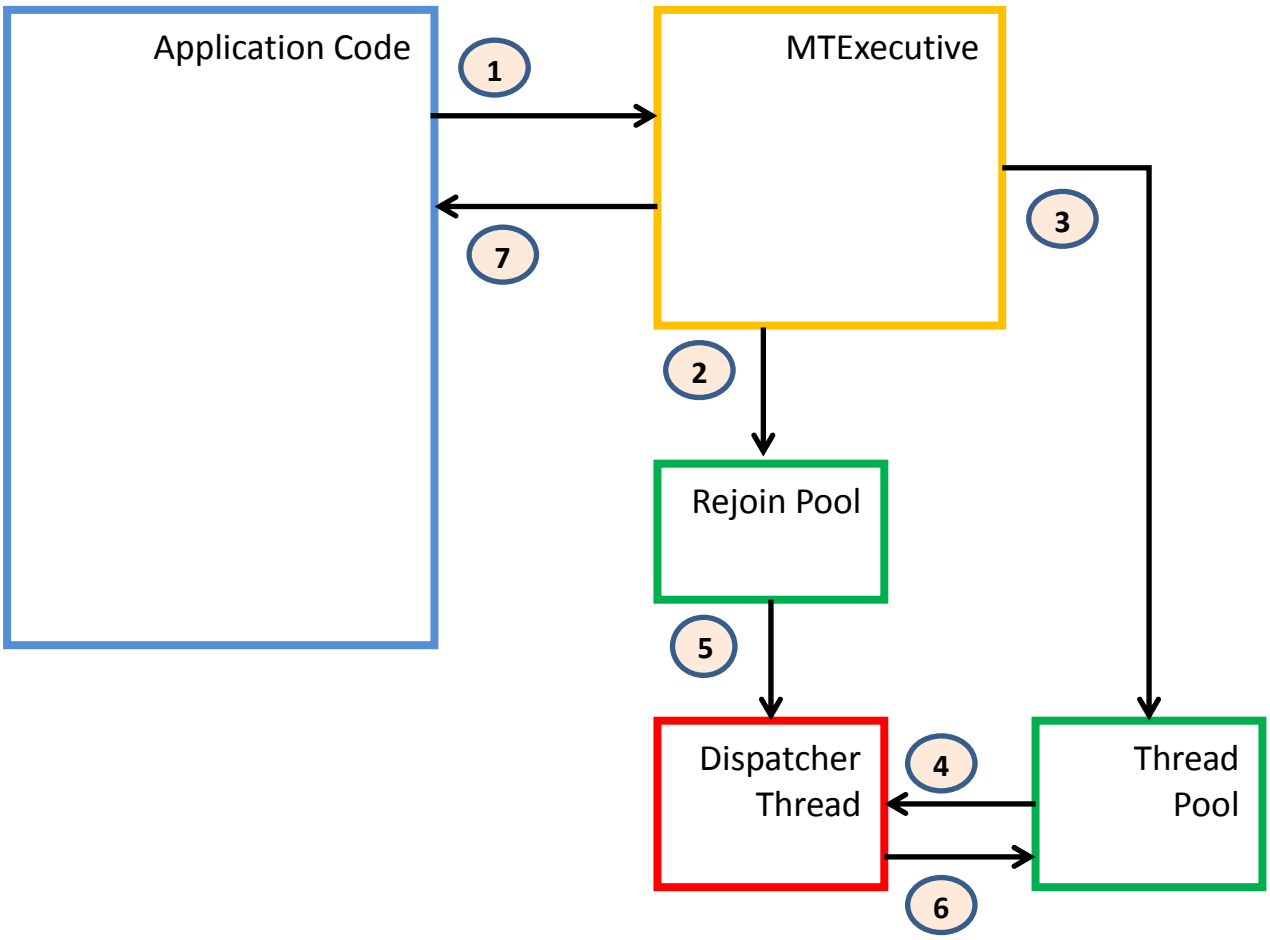
Once the request has finished executing it returns from the ExecuteThisRequest call then if the request was marked as rejoinable the RQE is written to a free slot in the Rejoin Pool. The Worker Thread will release any thread local database handles that are no longer valid or needed.

4

The Worker Thread then updates its state to show that it is again available for work.

5.4 Rejoining a Completed Request

5.4.1 Case #1: Request Found and Returned



1

The application calls the GetRejoinRequest API function, passing the arbitrary address of the “Owner”.

2

The function scans the Rejoin Pool for a completed request for the specified “Owner”.

3

The function posts the Rejoin Poll Semaphore in its own Thread Pool Member.

4

The Dispatcher Thread scans each Thread Pool Member and is triggered by the Rejoin Poll Semaphore being set in one of the members.

5

The Dispatcher Thread locates a completed Request in the Rejoin Pool, copies the RQE and frees the slot in the pool.

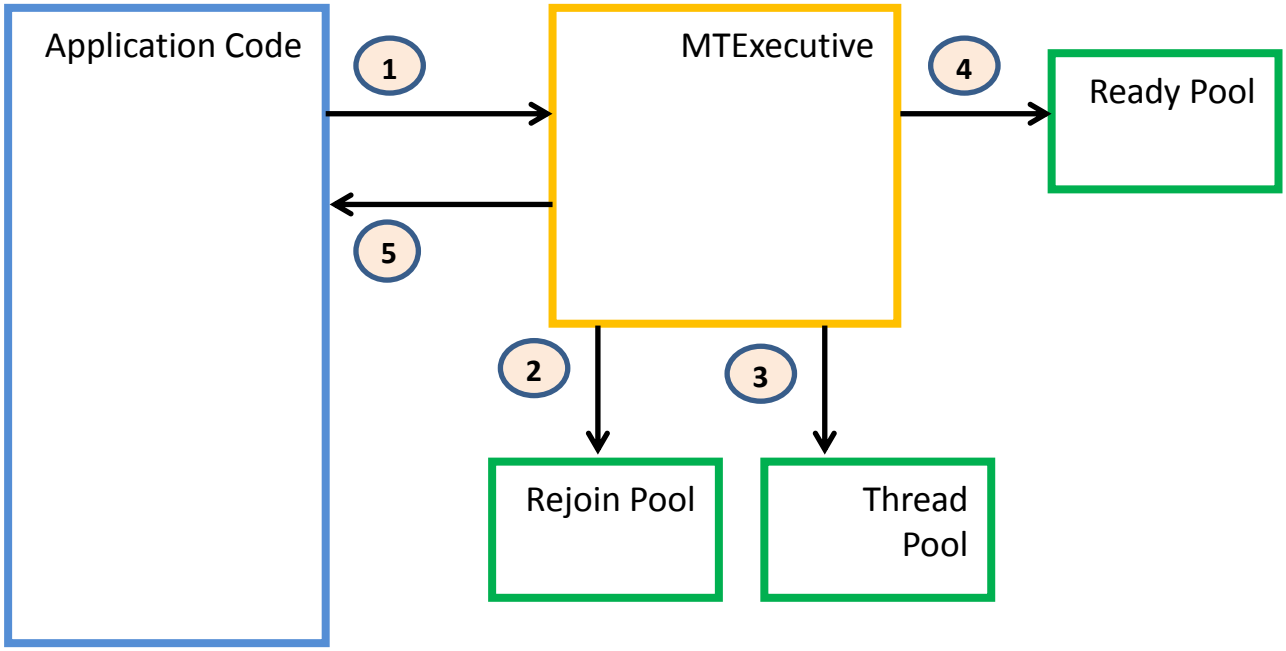
6

The Dispatcher Thread stores the completed RQE in the Thread Pool Member and clears the Rejoin Poll Semaphore.

7

The API function has been waiting for the Rejoin Poll Semaphore to become free again and now retirns the address of the completed Request Object to the application code with a return code indicating that a valid object has been returned (RJR_RETURNED).

5.4.2 Case #2: No Completed Requests are Available



1

The application calls the GetRejoinRequest API function, passing the arbitrary address of the “Owner”.

2

The function scans the Rejoin Pool for a completed request for the specified “Owner” and finds none.

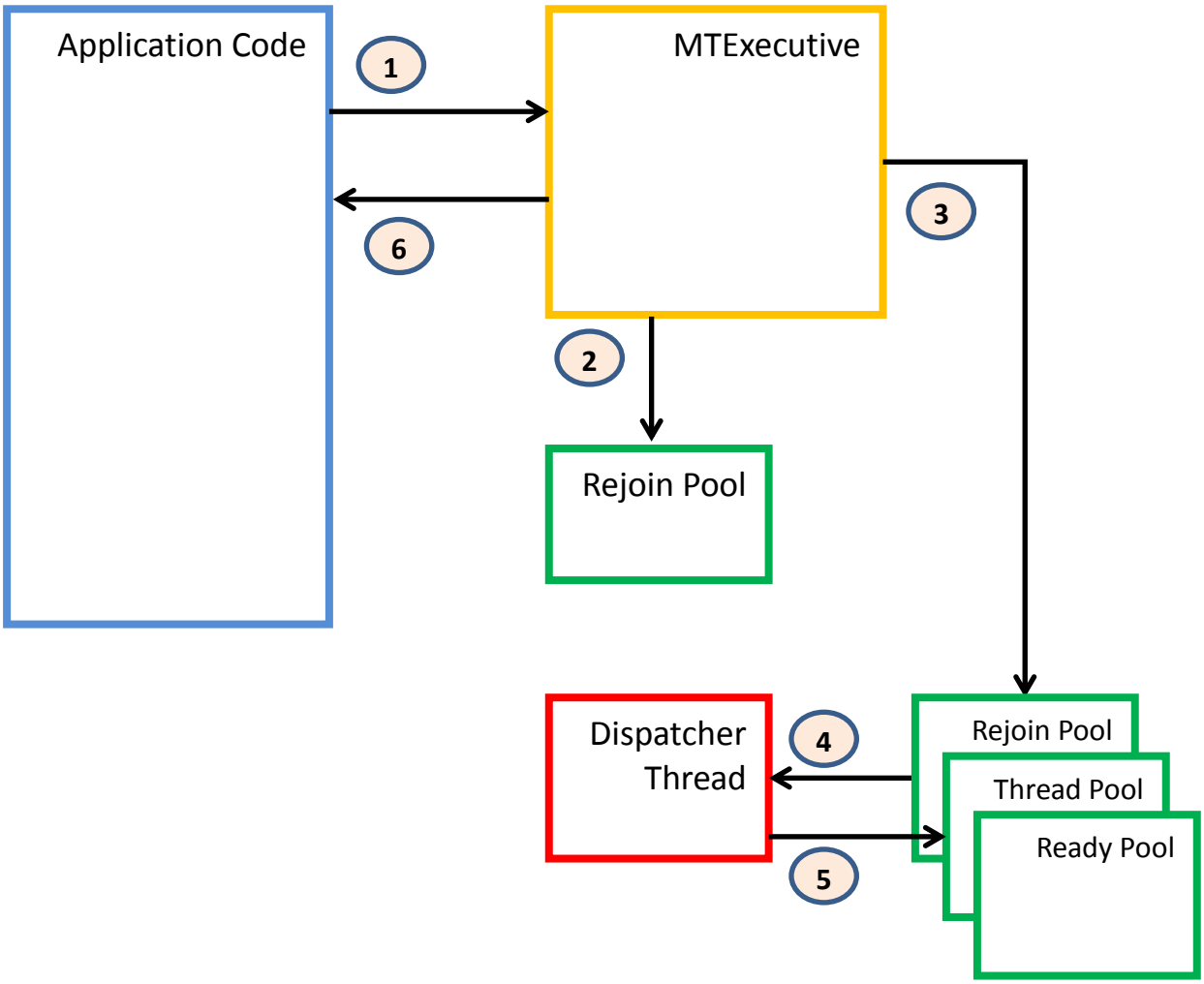
3

The function will then scan the Thread Pool Members checking if any thread is currently executing a request for the specified “Owner” if one is found then the function will return the RJR_NONE_READY indication. In this case we assume that none were found.

The function scans the Ready Pool for a request for the specified “Owner”.

When a request is located that matches the specified “Owner” the function returns the RJR_NONE_READY indication to the caller.

5.4.3 Case #3: No More Requests Exist



1

The application calls the GetRejoinRequest API function, passing the arbitrary address of the “Owner”.

2

The function scans the Rejoin Pool for a completed request for the specified “Owner” and finds none.

As in Case #2 the function also checks the Thread Pool Members and the Ready Pool and again finds no entries for the specified “Owner”. The result is however regarded as non-authoratative as requests could have been between states during the tests.

3

The function posts the Rejoin Poll Semaphore in its own Thread Pool Member, then waits until the semaphore is cleared again.

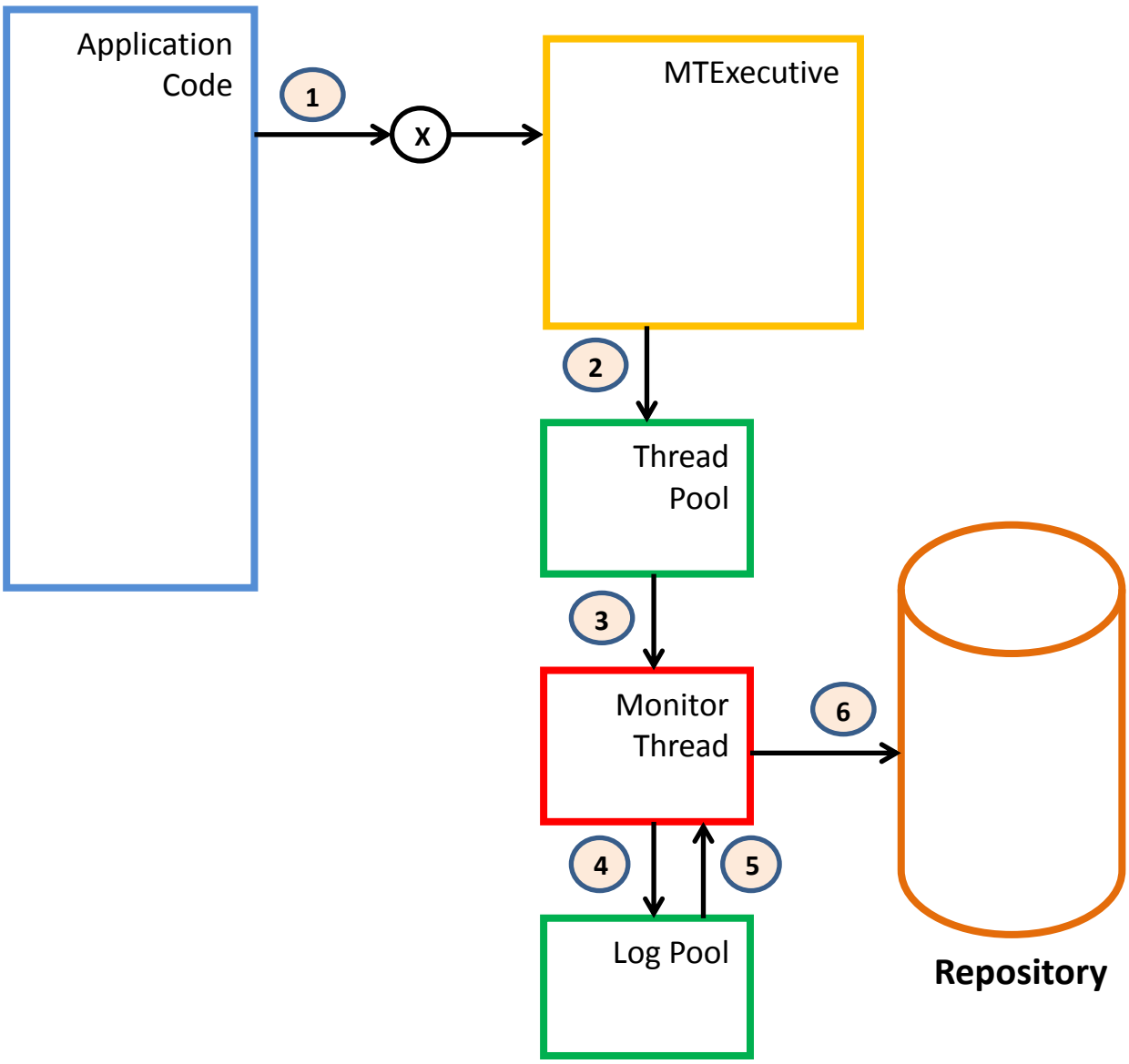
4

The Dispatcher Thread then performs checks of the Rejoin Pool, Thread Pool and Ready Pool to find any request for the specified “Owner”, in this case none is found. As these checks are performed by the Dispatcher Thread no state transitions of requests can happen during the search, the answer is therefore authoritative.

The Dispatcher Thread posts the return notification indicating that no more requests exist RJR_NONE_EXIST and clears the Rejoin Poll Semaphore.

The API function returns the RJR_NONE_EXIST notification to the caller.

6. Asynchronous Logging



1

The application code calls one of the logging API functions, passing the message to be logged.

2

The API function will block if the Log Semaphore is still asserted from a previous call on the same thread. When the semaphore is free the function stores the log message in the Thread Pool Member for the current thread and assets the Log Semaphore.

3

The Monitor Thread periodically checks the Thread Pool Members and is triggered on detecting the Log Semaphore being posted.

4

The Monitor Thread then moves the log message to the Log Pool and clears the Log Semaphore.

The Log Pool is configured as a clock buffer, this is a circular buffer with a head and tail pointer, entries are added to the tail and removed from the head.

5

The Monitor Thread periodically inspects the Log Pool, it is triggered by the presence of log messages in the pool.

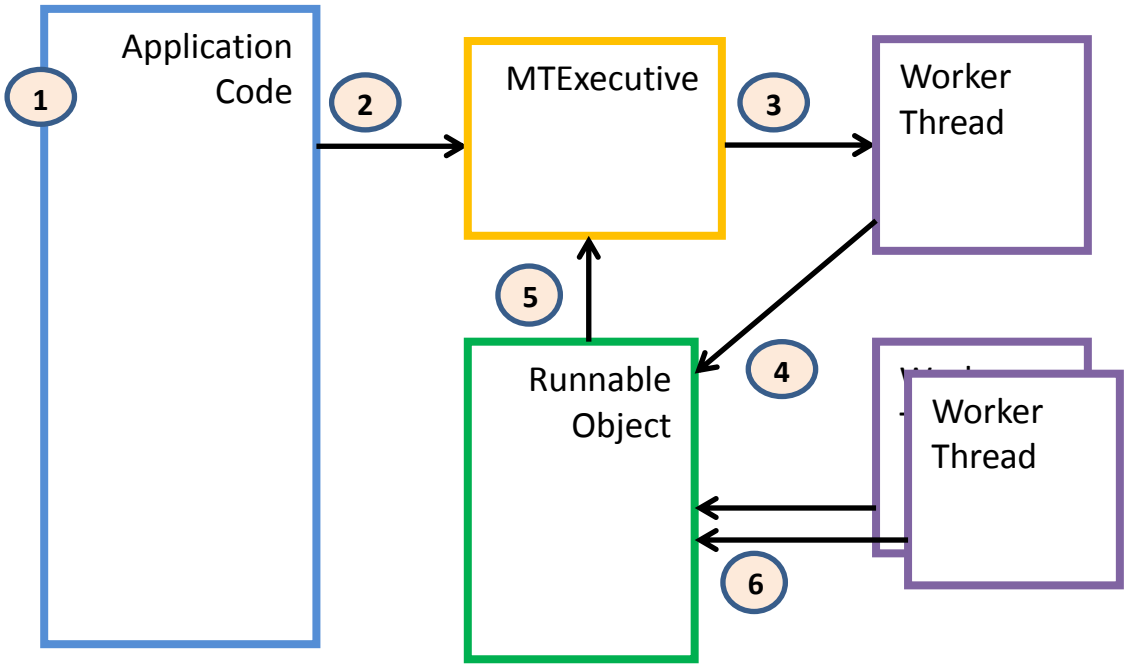
6

A number of messages are then written to the persistent log (usually the repository). Messages are written from the head pointer and the entry is freed by moving the head pointer towards the tail.

7. Runnable Objects

Runnable objects implement the Runnable interface that allows them to perform processing on multiple worker threads.

Runnable objects are the workhorse engines in a DX application, they execute the core functions of the applications on multiple threads in parallel. The sequence below shows the typical execution model of a Runnable object.



1

The mainline application code constructs a singleton instance of the Runnable object.

2

The application code creates a top level request object and posts it for execution by the Runnable object.

3

The multi-threading kernel dispatches the top level request object to a worker thread for execution by the Runnable object.

4

The Runnable object starts executing the top level request.

5

The Runnable object creates multiple sub-requests and posts them for execution by the Runnable object. The requests are then dispatched to available worker threads for execution.

6

The Runnable object is now executing the top level request and multiple sub-requests in parallel.