
**DMIS DISTRIBUTED SERVICE ARCHITECTURE**

DMIS Core Framework v3.0.1

# Technical FAQ

July 2005

# Table Of Contents

# 1. Architecture

## *1.1. Why do we need the distributed architecture?*

DMIS has traditionally operated in a client-server mode in which the DMIS Desktop accessed a single central server bank. This model is simple and has been effective in its context, but the larger technical vision for DMIS has always included a broader model of server distribution. Allowing organizations to run and manage DMIS on their own networks affords them a greater degree of control and improves the overall scalability of the DMIS network.

## *1.2. What are the distributed features of DMIS?*

Primary features include:

- Allowing for organizations to run an AppServer on their own network and to connect to other AppServers in a hierarchical fashion.
- Enhanced resilience through a fuller set of connectivity options such as intelligent connection cycling and multiple connection points.
- New features enabled by separating physical connections from logical identities.

## *1.3. Do all organizations have to implement a LAN server?*

No. They may choose to manage their own DMIS server or they may continue to access DMIS in a client-server capacity.

## *1.4. Does DMIS support multiple-level server hierarchies?*

Yes. Though the original intent was to support a simple DESKTOP->LAN->CENTRAL layout, the implementation also allows a server hierarchy of arbitrary depth to be established. This may be useful in situations where jurisdictions require this kind of model.

## *1.5. How is the server hierarchy declared?*

The *NetworkTopology.xml* config file is used to define the layout of the server hierarchy. Many organizations will use the hierarchy defined by the DMIS team, but in special cases customized hierarchies can be developed. A key concept is that an organization can define its own jurisdictions and corresponding server layout, and can also link to a connection point on another tree (perhaps an entry point in the server hierarchy owned by a “higher” jurisdiction).

## *1.6. Can distributed features be ignored or disabled?*

The distributed architecture is a superset of the traditional client-server functionality offered by DMIS. As such, it can emulate the legacy model very easily. In fact, this can be controlled by altering a single configuration file (*NetworkTopology.xml*). The current implementation includes a predefined version that emulates the original architecture, as well as one that includes support for a single-layer LAN hierarchy.

The LAN-aware version of this file is designed so that it works either in the presence or absence of LAN servers within a given organization. In other words, LAN servers are a factor only where they are used.

# 2. Application Server Modules

## *2.1. Can an AppServer run without any installed modules?*

At its core, an AppServer is a message routing hub and the messaging subsystem is always started as a part of the AppServer. Most other modules and functions, including the Service Manager that loads services and dispatches messages to them, are optional and can be installed by including them in the ApplicationServer.xml config file. When no modules are installed, however, the AppServer still functions as a *message routing hub*. Assuming that it is connected to some part of the larger DMIS network, it can serve as a ”hop” between other entities. In some cases, strategically places routing hubs can improve the redundancy of the network.

## *2.2. Is the Server Intercom still used?*

Yes, but (thanks to architectural enhancements) it has been simplified considerably and now uses the messaging subsystem for communications rather than the separate socket-based mechanism that it previous employed. This simplifies it configuration and eliminates the special attention that it sometimes required in the past. In addition, it enables clustering behaviors (such as shared memory constructs) that can operate even when clustered servers are in physically disparate locations (using the location abstraction features of the messaging subsystem that all DMIS entities now enjoy).

## *2.3. How can I determine whether the server is running?*

Start the ServerDiag tool and configure your AppServer by selecting the *Add Server* button. If, after successfully defining the server, the server is displayed in **bold letters**, the server is known to be UP. If it is not displayed in bold, another method to check a server’s availability is to open the **Command Line** window in ServerDiag and enter a command such as *envi*. If results are returned then the AppServer is UP and responding to diagnostic requests

## *2.4. How can I see the server logs?*

Logs for any AppServer can be viewed through the ServerDiag tool.

# 3. Connectivity

## *3.1. Can a LAN AppServer be accessed from outside its organization?*

Yes and, assuming that an organization's network security implementation allows it, there are a variety of ways this can happen:

- Dialing into the LAN server via RAS,
- Connecting to the DMIS LAN server through the internet,
- Tunneling through another part of the DMIS network (such as by connecting to the CENTRAL server when out in the field)

Tunneling through the DMIS network allows one to make a *physical* connection anywhere within the DMIS network that is convenient and continue to work as though *logically* connected to the LAN back home. Based upon routing information that's is tied to organizations and operators, the system routes traffic transparently to maintain a logical consistency.

## *3.2. What happens when a desktop's connection fails?*

This depends upon the configuration, but in the default situation the desktop will cycle to the next potential connection point in its preferred list. If a new connection is made through a secondary path, it is used until the preferred one becomes available. *If no connections can be made through any statically known channels, DMIS will begin a dynamic search for a suitable connection point based on its knowledge of the DMIS network.* In order to be "online" in DMIS, all a workstation needs is a connection with any other point that is already connected. Thanks to the magic of routing, that workstation then operates normally.

## *3.3. How are connection points manually defined?*

They can be established via the MessagingChannels.xml config file, which may be edited manually or through the DMISConfig tool.

## *3.4. How can DMIS automatically find connection points?*

Each DMIS workstation of server keeps an internal structure, called the Routing Table, representing the layout of the DMIS network. It also holds various pieces of metadata for every entity on the network of which it is aware. This table is kept current in real time as nodes connect and disconnect from the DMIS network, and is persisted on disk. When connectivity fails, this structure still contains useful information about the DMIS network. One set of useful information is URLs for all other entities that can be reached publicly on he DMIS network. The connection cycling code uses this data to locate alternate servers based on organization-oriented precedence rules.

### 3.5. Can a desktop connect to another desktop?

Yes! If a desktop is connected to the DMIS network, another desktop installation may connect to the first one and enter thus the DMIS network indirectly. All traffic bound for the second desktop will travel through the first one transparently. In fact, this concept works for any entity on the DMIS network (workstation, server, etc) – connecting to any node gets you into the entire network.

### 3.6. Is operator tracking still available? What are its features?

Yes, and it is more flexible than before. Many of its new features may not be used initially because the existing codebase is client-server oriented and written to access the AppServer for connectivity information rather than sensing it directly. Some of its modern characteristics include:

- It is now based entirely on the routing table, which is inherently reliable regarding connection states, rather than a database-level tracking. While the "old" approach worked fine for the client-server scenario, the distributed nature of DMIS requires this more flexible this new.

- Connectivity information is available both to services *and* to applications on the DMIS desktop. Previously, only services had access to any of this information.

- Services and applications can determine whether *specific operators* are online or otherwise reachable. In addition to offering operator connection information, reachability to any *AppServer* can be tested.

- Services and applications can set up listeners to receive notifications to any connectivity events. Masks can be specified to narrow the events to interesting entities only.

- It is possible to determine physical locations and possible routes between entities on the DMIS network given current connection states between nodes.

Operator tracking features are offered to all services when they are started, and applications can access them through the *ClientGateway* (a.k.a. the "proxy").

## 4. Deployment Features

### 4.1. How are services deployed?

Each service implements the `IMessageService` or `IBackgroundService` interface. Services are initialized and started as part of the AppServer boot process. Each service that is to be included has a corresponding **.svc* deployment descriptor. There is a deployment scanner that automatically locates any such descriptors in the

configuration path and loads those services. Those **.svc* files can exist anywhere within the config hierarchy for that AppServer's profile.

### 4.2. Can individual services be disabled?

Yes. There are two ways to disable a service.

1. Remove the **.svc* file associated with that service so that the service is not included in the AppServer boot process.

2. Rename the **.svc* file to **.svc.disabled*. This special extension causes the service descriptor to be ignored during startup with the need to actually delete the file. In addition, the **.disabled* extension can be used to disable a service that is declared at a higher level. Assume that the config directory contains the following deployment descriptor entry:

```
config
  +- production
       +- appserver
       |    +- central
       |    |- lan
       |    +- embedded
       +--- FTP.svc
```

In this configuration, the FTP.svc is always loaded no matter what kind of AppServer is started (CENTRAL, LAN, or EMDEDDED). The service can be disabled by inserting an empty "dummy" file in the appropriate spot to override the high entry:

```
config
 +- production
     +- appserver
     |   +- central
     |   |- lan
     |   +- embedded
     |       +--- FTP.svc.disabled (may be dummy)
     +--- FTP.svc
```

The presence of the **.disabled* entry disables the FTP service for EMBEDDED AppServers, while leaving it in place for all other deployment types.

### 4.3. Can DMIS applications determine whether a service is running?

Yes, using the `CapabilityQuery` class, either directly or through the *ClientGateway* or *IServerHelper*. The following code determines whether the TIE service is currently running on an AppServer:

```
// Determine the server just above us in the hierarchy.
RoutingID ourServer = gateway.getNextServer(false, true);

// Formulate a query to test for presence of the given service.
String query = CapabilityQuery.formQuery(
                  CapabilityQuery.QUERY_SERVICE_RUNNING, "TIE");

// Ask the server whether the service is running.
boolean tieIsRunning = "true".equals(
   CapabilityQuery.discoverCapability(RoutingID.CENTRALSERVER, query))
```

The CapabilityQuery.discoverCapability() method returns "true" if the service is running, or "false" or NULL if the service is not available.

# 5. Clustering

## *5.1. Does DMIS support clustering?*

Yes. Any number of machines can be tied together into a logical cluster, as might be needed for the CENTRAL server deployment (though LAN servers can be clustered as well as needed).

An interesting feature of DMIS is its ability to cluster servers that are on different physical networks or ones that are linked indirectly. Since clustering is now implemented through the messaging subsystem, any servers that are mutually reachable at a logical level (based on DMIS identity and not physical connections or locations) may be coupled into a cluster. All of DMIS' new flexibility and resilience features apply to clustering.

## *5.2. How was clustering previously implemented?*

DMIS has always supported clustering to some degree. Previous versions of DMIS contained fairly complex code to manage inter-server communications with the main purpose of allowing operators connected to one server to communicate (via messaging) to operators connected to another server in the cluster. The clustering mechanism was broken into several distinct parts:

- *Previous versions of DMIS used the Oracle database as a common point of communication*. As servers booted, they wrote an entry into a designated table to signal that they had "arrived on the scene". Other AppServers, which would poll this table periodically, would pick up the latest copy of the server list.

- *Communications between servers was accomplished using the* ServerIntercom *class, which could broadcast messages to servers in the peer list.* The implementation and format of this transmission was simplified and separate from the rest of the messaging implementation.

- *Operator connections were tracked and tagged to the AppServer to which they were connected.* This is because message traffic (e.g. messages sent from the Notification Service) must ultimately flow down the physical connection associated with that operator->server pair. Messages must end up on the correct server somehow in order to be transmitted.

- *Operator connections were validated on a regular basis to ensure the accuracy of the list.* Bouncing AppServers had the potential to easily invalidate operator states, so thread on each AppServer would periodically (every few seconds) check operator entries tagged to it against its list of actual socket connections. Corrections were made as necessary.

- *Queuing code, when accepting a message on the CENTRAL server, consulted (1) the server list and (2) the operator list to determine whether a message needed to be "shuttled" to another server*. If an operator was known to be connected to another machine, the message was transmitted to the AppServer where the operator was connected. The message was added to the operator's queue on that machine and was thus eligible to be transmitted using the link.

### *5.3. How is clustering currently implemented?*

The current implementation takes advantage of features already embedded within the messaging subsystem in order to simplify its logic considerably and at the same time enhance its own features. The RoutingTable used by the main messaging logic replaces a lot of the specialized functions previous implemented separately for clustering:

1. Server availability can be determined through the RoutingTable,

2. Real-time pruning of server and operator connection states is already a central feature of the RoutingTable,

3. Messages can be addressed to a specific server or to an indiscriminate part of a cluster (by simply omitting the InstallationID portion of a DestinationID), and

4. The messaging subsystem automatically and intelligently routes any given message across any number of connections in order to reach the correct operator.

The cluster, therefore, simply uses the features of the messaging subsystem to communicate.

### *5.4. What are the characteristics of DMIS clustering?*

The current implementation is much more flexible than previous versions, thanks to improvements in the messaging subsystem that it now leverages fully. Clustering support includes the following sophisticated features:

- AppServers are automatically clustered if (1) they are in the same "logical bank", and (2) each one has loaded the `ClusterManagerModule`. Two servers are in the same "bank" if their RoutingIDs are equal apart from the InstallationID that identifies a specific DMIS instance.

- AppServers can be clustered across distinct physical networks.

### *5.5. Can LAN servers use clustering?*

Yes. Any AppServer that loads the `ClusterManagerModule` (specified in the *ApplicationServer.xml* file) can be clustered. EMBEDDED AppServers (which run on the operator's workstation) are never clustered simply because clustering functionality serves no purpose there. Therefore, the `ClusterManagerModule` is simply not loaded.

### *5.6. How are servers added to a cluster?*

All DMIS AppServers are "activated" using the Activation Wizard, which validates and authenticates the logical identity of an AppServer. One or more AppServers are *automatically clustered* when:

1. Clustering is enabled by loading the `ClusterManagerModule`, and

2. They are in the same logical "bank". For example, all CENTRAL servers have the same identity except for the InstallationID, and thus they are considered to be part of the same "bank". Traffic may flow to any of these servers and they are logically the same. LAN servers are clustered in the same way.

Enabling clustering from a configuration aspect, therefore, is a process of (1) running the Activation Wizard for each AppServer to be tied together, and (2) ensuring that the `ClusterManagerModule` is enabled on each server via the *ApplicationServer.xml* config file.

## 6. Development

### *6.1. What is a CoreContext?*

The `CoreContext` class is a foundational "bus" that offers references to the basic resources required by any application or application server. This approach replaces the

older static "XXXXManager" approach used previously in DMIS. This avoidance of a static model allows for a more flexible implementation and at the same time clarifies true dependencies.

The following facilities are offered by this class:

- Access to core variables such as the DMIS base path,
- Consolidated functions from database handling, config files, and other facilities.
- Dynamic logging that is context-sensitive and can be controlled very finely,
- Access to command-line, system, and environment parameters from anywhere in DMIS,
- Convenient time-keeping facility that is NTP-synchronized (where possible) and offers time zone conversions,
- More flexibility than the previous static-class-driven approach to basic facilities.

Except in rare instances, application and service developers will never need to instantiate a `CoreContext` instance directly. It is normally constructed at the entry point of the application and passed around to all child modules as objects as needed. In virtually all cases, the appropriate `CoreContext` instance is provided by the "container".

### *6.2. How can I create a new DMIS service?*

The basic steps are simple:

1. Decide what the service will do and create an API by subclassing the Message class into appropriate operation types and parameters.
2. Create a new class that implements the IMessageService interface to handle the messages that form the API.
3. Create a *.svc deployment descriptor in the config directory. This descriptor is picked up automatically by the deployment scanner and the service is loaded by the ServiceManager module.

The details of this process are outline in the accompanying documents (see the *Developer's Guide*).

### *6.3. What happened to the isMessageRelevant( ) method of IMessageService?*

The *isMessageRelevant( )* method was used in previous versions of the middleware in order to map incoming messages to specific services on an AppServer. When a message arrived, the AppServer invoked *isMessageRelevant( )* on each service in order to determine if the service could handle the message. In this version of the middleware, however, messages are mapped using the deployment descriptors (*.svc files) that associated with each service.

### *6.4. What are the new features of ConfigurationManager?*

The `ConfigurationManager` class in DS is a significant update to previous versions. Features include:

- Flexible path declarations that allow locations to be specified as a base and relative locations, absolute locations, or a combination of both.
- Alternate config locations and precedence rules. These allows for constructs such as config directory structures that are resolved as a hierarchy (which is in fact how DMIS functions).
- Ability to easily write config files as well as read them. Files can be read, modified, and written in a non-destructive manner (e.g. all XML comments and other constructs from the input file are retained). Alternately, completely new config files can be created just as easily.
- Ability to dynamically and transparently substitute config files with values constructed in memory via "virtual" configs.
- Support for XML-based config files as well as `IConfiguration`s driven from any Map interface.
- Transparent on-the-fly encryption/decryption of config files.
- Support for symbolic variable replacement, in which symbols in any given config files are replaced at runtime with literal values. In practice, these value come from the CoreCtx which allows values to be pulled from the OS environment, system properties, or the command line and also allow for defaults.

Unlike previous versions of this class, this version does not offer many static methods. Instead, it must be instantiated and passed around to the any child objects that might use it. This is a deliberate attempt to improve the modularity of the approach, since this version of DMIS requires more flexibility than static methods alone can provide (which assume that every client of `ConfigurationManager` in the VM will use the same settings).

### 6.5. What is the CapabilityQuery class?

The `CapabilityQuery` class is a facility that allows applications and services to determine what kind of functionality a given server is configured provide. It can also provide information about that server. Some examples of information it can provide include:

- Whether a given service, such as TIE or FTP, is running.
- Whether a specific module, such as the Diagnostic Module, is installed.
- Whether the AppServer can process a given type of message based on the services that are installed.
- Connection point information such as *host:port* for an AppServer when only the RoutingID is known. Conversely, the RoutingID of an AppServer for which we only have the *host:port*.
- Whether a given DMIS version is compatible with the AppServer.

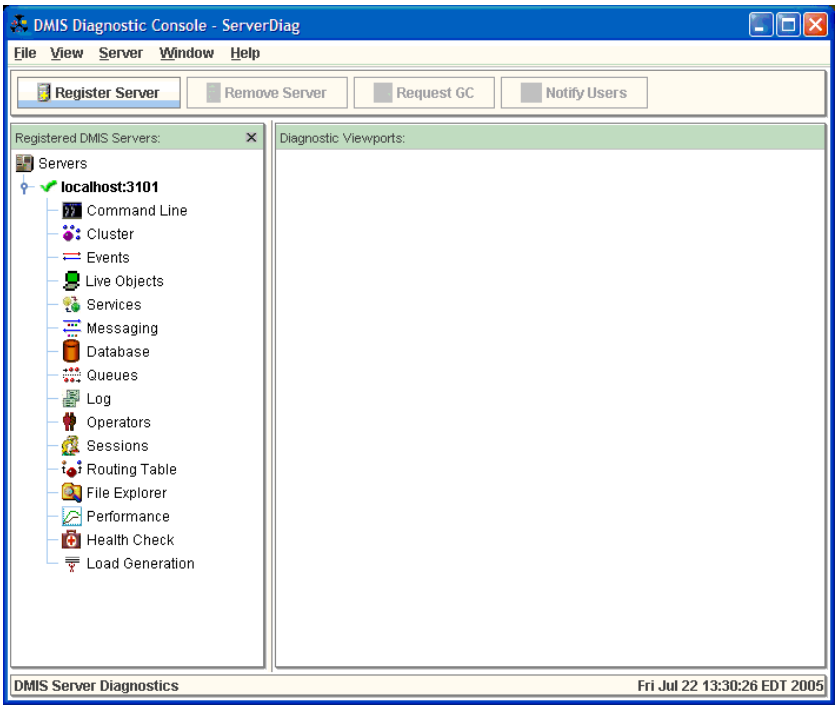
The `CapabilityQuery` class may be used directly or through helper methods offered by the *ClientGateway* and *IServiceHelper* classes.

## 7. Diagnostics

### 7.1. How can diagnostics be run against an AppServer?

There are two ways to run diagnostics on a DMIS server:

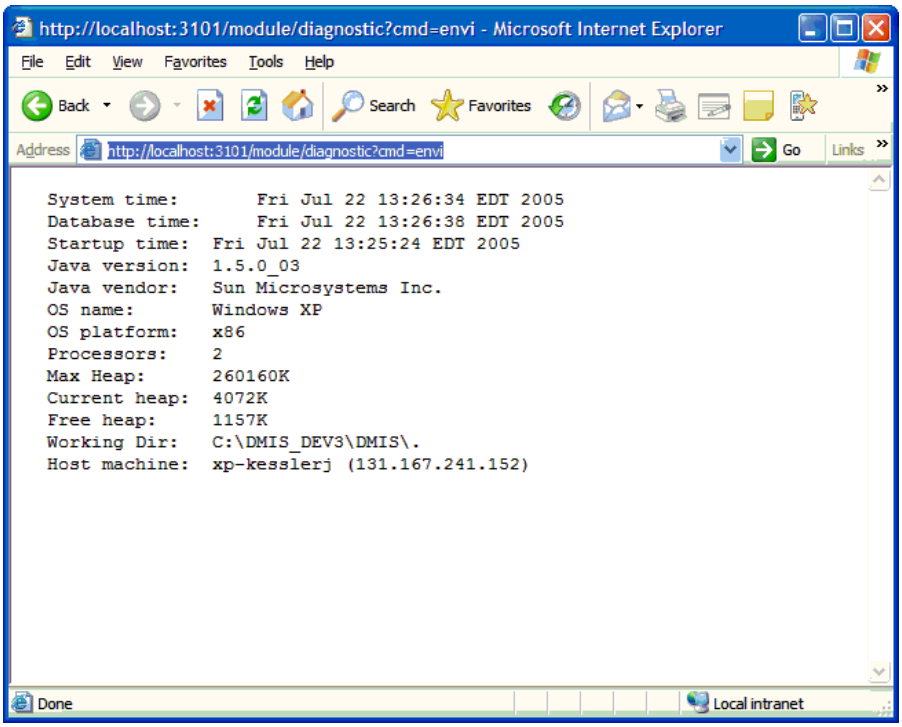
1. Using the DiagnosticConsole by running the *launch/ LaunchDiagnosticConsole.bat* batch file. This brings up a screen like the one illustrated here, from which servers can be registered and diagnosed:




2. Execute diagnostic commands directly in a web browser. The format for the URL is the following:

   http://*servername*:*serverport*/module/diagnostic?cmd=*query*

   Results are then returned directly in the browser:




Note that the DiagnosticModule must be running on the DMIS AppServer in order for any information to be returned from either method. In addition, unless the AppServer is running on the same machine as the diagnostic console or browser, administrative privileges to the server are required.

# 8. General

## *8.1. What is the current state of the DS codebase?*

It could be considered a "Beta" version. It is fully functional but probably contains defects, especially in the messaging subsystem (where the newest parts exist). In addition, it will need to be to be profiled for performance, load tested for scalability, and optimized appropriately. The areas of biggest concern for performance are the route computations in the RoutingTable class. They could benefit from (1) optimization, (2) caching, (3) minor alterations in assumptions to simplify its work, or (4) other ideas.

## *8.2. Why is the DMIS AppServer a proprietary solution?*

There is a lot of DMIS history behind this question. The original release of DMIS, called Fast Track, was produced rapidly in response to the events of September 11th, 2001 in order to provide a core capability. DMIS was built to specifications (notably cost and offline functionality constraints) that were, at the time, difficult to satisfy using other open source solutions. Given these pressures, a relatively simple client-server AppServer was produced to meet the immediate need. The intention was to refactor it into a J2EE solution when the time was right.

Over time, as the functionality of the AppServer expanded, a prototype conversion to J2EE was created but due to manpower issues and time constraints between releases, the existing (fairly stable) code continued to receive preference. In addition, the AppServer successfully met the needs of the project and contained features that would be harder to replicate with other solutions.

The application server itself, which distributes messages to server-side components, has been keep relatively simple. *The complexity of DMIS is really in the messaging subsystem*, which offers incredible features that are simply not found in other "free" messaging implementations.

### *8.3. Can the DMIS AppServer be standardized?*

Yes, it could be refactored in order to make it comply with certain industry standards. The most obvious strategy is to

1. **Refactor or "wrap" the messaging subsystem as a JMS provider**. The messaging subsystem, which contains most of the distinctive features of DMIS, could be wrapped as a *JMS provider* and thereby become usable from any J2EE AppServer such as JBOSS. DMIS currently implements *queues* but not *topics*, so it would map to a subset of the JMS SPI unless topics were implemented as well (they could be implemented as an extension of queues). Because the DMIS messaging paradigm has many parallels to JMS, such a conversion could be done in a fairly straightforward manner.

   The DMIS messaging implementation is more advanced than any existing open source JMS provider. It extended features would be offered as a superset of the JMS provider interface for those modules that need those functions. Because the core JMS interfaces would function, however, any components that operate at the lowest-common-denominator level of JMS would function as well. One known limitation would be that DMIS services running on the AppServer might need to avoid DMIS extensions or risk continuing to be DMIS-specific. Is that a real problem given that the code I meant only for DMIS? Probably not.

   **Replace the AppServer with an open source solution**. One the messaging subsystem is wrapped as a JMS provider, any J2EE AppServer could be used while retaining the advanced messaging features of DMIS. One known limitation would be that DMIS *services* running on the AppServer (but not the AppServer itself) might need to avoid DMIS extensions or risk continuing to be DMIS-specific. Is that a real problem given code is meant only for use with DMIS? Probably not.