

---

---

**DMIS DISTRIBUTED SERVICE ARCHITECTURE**

DMIS Core Framework v3.0.1

**Development Guide**

July 2005

---

### Revision History

| Date      | Version | Author     | Remarks         |
|-----------|---------|------------|-----------------|
| May 2005  | 1.0.1   | J. Kessler | Initial version |
| July 2005 | 1.1.1   | J. Kessler |                 |
|           |         |            |                 |
|           |         |            |                 |

---

# Table Of Contents

|   |           |
|---|-----------|
| <b>1. INTRODUCTION.....</b>                                       | <b>8</b>  |
| 1.1. PURPOSE.....   | 8         |
| 1.2. SCOPE.....   | 8         |
| 1.3. AUDIENCE .....   | 8         |
| <b>2. IMPLEMENTING SERVICES .....</b>                             | <b>9</b>  |
| 2.1. SOURCE CODE LAYOUT.....                                      | 9         |
| 2.1. JDK VERSIONS .....   | 11        |
| 2.2. BASIC IMPLEMENTATION STEPS .....                             | 11        |
| 2.2.1. Define a Message API for the New Service .....             | 11        |
| 2.2.2. Create Service Descriptors .....                           | 12        |
| 2.2.3. Implement Service Code .....                               | 12        |
| 2.3. DEPLOYMENT DESCRIPTORS .....                                 | 12        |
| 2.3.1. *.svc Descriptors (Service Configuration Files).....       | 12        |
| 2.3.2. *.wfd Descriptors (Workflow Configuration Files) .....     | 15        |
| 2.3.3. Dynamic Deployment and Undeployment.....                   | 17        |
| 2.4. CLIENT STUBS .....   | 17        |
| 2.5. THE CORECONTEXT CLASS .....                                  | 18        |
| 2.5.1. Variable Access .....                                      | 18        |
| 2.5.2. Command-Line Parameter Access.....                         | 19        |
| 2.5.3. DatabaseManager .....                                      | 20        |
| 2.5.4. DataHelper .....   | 21        |
| 2.5.5. SerializationManager .....                                 | 23        |
| 2.5.6. Logger.....  | 23        |
| 2.5.7. Clock.....   | 24        |
| <b>3. MESSAGING DESIGN CONSIDERATIONS.....</b>                    | <b>26</b> |
| 3.1. MESSAGE STRUCTURE.....                                       | 26        |
| 3.1.1. Message Hierarchies .....                                  | 26        |
| 3.1.2. Appropriate Subclassing.....                               | 27        |
| 3.1.3. Naming Conventions.....                                    | 27        |
| 3.2. COMMON MESSAGING PATTERNS .....                              | 28        |
| 3.2.1. Request-Response .....                                     | 28        |
| 3.2.2. Post .....   | 29        |
| 3.2.3. Acknowledgement .....                                      | 29        |
| 3.2.4. Notification .....   | 29        |
| 3.2.5. Return Receipt .....                                       | 29        |
| 3.2.6. Synchronization .....                                      | 29        |
| <b>4. MESSAGE CLASS DETAILS.....</b>                              | <b>30</b> |
| 4.1. HEADER-RELATED OPERATIONS .....                              | 30        |
| 4.1.1. getCorrelationID( ) / setCorrelationID( ).....             | 30        |
| 4.1.2. getDestinationID( ) / setDestinationID( ) .....            | 31        |
| 4.1.3. getExpirationTimestamp( ) / setExpirationTimestamp( )..... | 32        |
| 4.1.4. getIsCarbonCopy( ) / setIsCarbonCopy( ).....               | 33        |
| 4.1.5. getIsVolatile( ) / setIsVolatile( ).....                   | 34        |
| 4.1.6. getMessageKey( ) / setMessageKey( ) .....                  | 35        |
| 4.1.7. getPriority( ) / setPriority( ) .....                      | 35        |
| 4.1.8. getReplyToID( ) / setReplyToID( ).....                     | 36        |
| 4.1.9. getSourceID( ) / setSourceID( ) .....                      | 36        |
| 4.1.10. getTraceID( ) .....                                       | 37        |

|         |  |    |
|---------|--|----|
| 4.2.    | ATTRIBUTE-RELATED OPERATIONS .....                           | 37 |
| 4.2.1.  | <i>attributeExists( )</i> .....                              | 37 |
| 4.2.2.  | <i>validateProperties( )</i> .....                           | 37 |
| 4.2.3.  | <i>clearAttributes( )</i> .....                              | 39 |
| 4.2.4.  | <i>getAttributeNames( )</i> .....                            | 40 |
| 4.2.5.  | <i>getAttributeBoolean( ) / setAttributeBoolean( )</i> ..... | 40 |
| 4.2.6.  | <i>getAttributeByte( ) / setAttributeByte( )</i> .....       | 40 |
| 4.2.7.  | <i>getAttributeDouble( ) / setAttributeDouble( )</i> .....   | 40 |
| 4.2.8.  | <i>getAttributeFloat( ) / setAttributeFloat( )</i> .....     | 41 |
| 4.2.9.  | <i>getAttributeInt( ) / setAttributeInt( )</i> .....         | 41 |
| 4.2.10. | <i>getAttributeLong( ) / setAttributeLong( )</i> .....       | 41 |
| 4.2.11. | <i>getAttributeObject( ) / setAttributeObject( )</i> .....   | 41 |
| 4.2.12. | <i>getAttributeShort( ) / setAttributeShort( )</i> .....     | 42 |
| 4.2.13. | <i>getAttributeString( ) / setAttributeString( )</i> .....   | 42 |
| 4.2.14. | <i>clearContent( )</i> .....                                 | 42 |
| 4.2.15. | <i>getContent( ) / setContent( )</i> .....                   | 42 |
| 4.3.    | DEBUGGING FACILITIES .....                                   | 43 |
| 4.3.1.  | <i>clearActivity( )</i> .....                                | 44 |
| 4.3.2.  | <i>getActivityLog( )</i> .....                               | 44 |
| 4.3.3.  | <i>recordActivity( )</i> .....                               | 45 |
| 4.3.4.  | <i>getDisplayProfile( )</i> .....                            | 45 |
| 4.3.5.  | <i>getVisitList( )</i> .....                                 | 45 |
| 4.4.    | HOUSEKEEPING OPERATIONS .....                                | 46 |
| 4.4.1.  | <i>clone( ) / clone( keepExistingTraceID )</i> .....         | 46 |
| 4.4.2.  | <i>HTTP Server Module</i> .....                              | 46 |
| 4.4.3.  | <i>Hang Detection Module</i> .....                           | 46 |
| 5.      | SERVICE INTERFACES .....                                     | 48 |
| 5.1.    | IBACKGROUNDSERVICE INTERFACE .....                           | 48 |
| 5.1.1.  | <i>start( )</i> .....  | 48 |
| 5.1.2.  | <i>stop( )</i> .....   | 48 |
| 5.2.    | IMESSAGESERVICE INTERFACE .....                              | 48 |
| 5.2.1.  | <i>Start( )</i> .....  | 49 |
| 5.2.2.  | <i>Stop( )</i> .....   | 49 |
| 5.2.3.  | <i>getStatus( )</i> .....                                    | 49 |
| 5.2.4.  | <i>onMessageReceived( )</i> .....                            | 50 |
| 5.3.    | ISERVICEHELPER INTERFACE .....                               | 50 |
| 5.3.1.  | <i>getCoreContext( )</i> .....                               | 51 |
| 5.3.2.  | <i>sendMessage( )</i> .....                                  | 51 |
| 5.3.3.  | <i>sendMessageToServersBelow( )</i> .....                    | 53 |
| 5.3.4.  | <i>sendMessageToServersAbove( )</i> .....                    | 54 |
| 5.3.5.  | <i>sendMessageAndWait( )</i> .....                           | 54 |
| 5.3.6.  | <i>forwardMessage( )</i> .....                               | 55 |
| 5.3.7.  | <i>notifyPeers( )</i> .....                                  | 56 |
| 5.3.8.  | <i>getOnlineOperators( )</i> .....                           | 56 |
| 5.3.9.  | <i>getKnownOperators( )</i> .....                            | 57 |
| 5.3.10. | <i>isConnected( )</i> .....                                  | 57 |
| 5.3.11. | <i>getRoutingIDForSelf( )</i> .....                          | 57 |
| 5.3.12. | <i>getRoutingIDForParent( )</i> .....                        | 58 |
| 5.3.13. | <i>hasParent( )</i> .....                                    | 58 |
| 5.3.14. | <i>getServersAbove( )</i> .....                              | 58 |
| 5.3.15. | <i>getServersBelow( )</i> .....                              | 59 |
| 5.3.16. | <i>getPeerRoutingIDs( )</i> .....                            | 59 |
| 5.3.17. | <i>getTiersToCentral( )</i> .....                            | 59 |
| 5.3.18. | <i>getTiersToHere( )</i> .....                               | 60 |
| 5.3.19. | <i>isEmbeddedServer( )</i> .....                             | 60 |

---

|         |  |    |
|---------|--|----|
| 5.3.20. | <i>isIntermediateServer( )</i> .....                 | 60 |
| 5.3.21. | <i>isCentralServer( )</i> .....                      | 61 |
| 5.3.22. | <i>refersToEmbeddedServer( )</i> .....               | 61 |
| 5.3.23. | <i>refersToIntermediateServer( )</i> .....           | 61 |
| 5.3.24. | <i>refersToCentralServer( )</i> .....                | 61 |
| 5.3.25. | <i>refersToAnyServer( )</i> .....                    | 62 |
| 5.3.26. | <i>refersToThisServer( )</i> .....                   | 62 |
| 5.3.27. | <i>getPhysicalName( )</i> .....                      | 62 |
| 5.3.28. | <i>getUserContext( )</i> .....                       | 62 |
| 5.3.29. | <i>getSharedMemoryHelper( )</i> .....                | 63 |
| 5.4.    | ISharedMemory Interface .....                        | 63 |
| 5.4.1.  | <i>getSharedMap( )</i> .....                         | 64 |
| 5.4.2.  | <i>getSharedList( )</i> .....                        | 66 |
| 5.4.3.  | <i>getSharedObject( ) / setSharedObject( )</i> ..... | 68 |
| 5.4.4.  | <i>getSharedCriticalSection( )</i> .....             | 69 |
| 5.4.5.  | <i>sharedMapExists( )</i> .....                      | 70 |
| 5.4.6.  | <i>sharedListExists( )</i> .....                     | 70 |
| 5.4.7.  | <i>sharedObjectExists( )</i> .....                   | 71 |

---

## Table Of Figures

|  |    |
|--|----|
| FIGURE 1: DMIS SOURCE CODE LAYOUT                          | 9  |
| FIGURE 2: SERVICE DESCRIPTOR (*.SVC ) FILE FORMAT          | 13 |
| FIGURE 3: WORKFLOW DESCRIPTOR (*.WFD) FILE FORMAT          | 16 |
| FIGURE 4: SUBCLASSING A <i>MESSAGE</i> FOR AN API FUNCTION | 27 |
| FIGURE 5: MESSAGE CLASS METHODS                            | 30 |
| FIGURE 6: MESSAGE VALIDATION WITHIN SERVICES               | 39 |
| FIGURE 7: SAMPLE SHAREDMAP CODE                            | 65 |
| FIGURE 8: SAMPLE SHAREDLIST CODE                           | 67 |
| FIGURE 9: SAMPLE SHAREDOBJECT CODE                         | 69 |



---

## 1. Introduction

### 1.1. Purpose

This document is a guide to developing services against the DMIS Application Server.

### 1.2. Scope

Specific interfaces, steps, and techniques are discussed as they related to the task of developing against the Distributed Services architecture.

### 1.3. Audience

Audience is anyone on the DMIS team who is interested in designing or developing DMIS services and applications.



---

## 2. Implementing Services

Implementing services for this version of DMIS bears considerable similarity to the previous implementation paradigm. The main differences involve enhancements to the DMIS directory structure, and the fact that many functions that were once implemented in code are now contained within various deployment descriptors.

### 2.1. Source Code Layout

The following is a visual summary of the updated DMIS code structure.

**Figure 1: DMIS Source Code Layout**

Some of the most notable classes, as they relate to the AppServer, include:

**org.dmis.api.beans.\***

Classes referenced by various messages that are common across multiple services.  
Examples include Person, Address, Operator, etc.

**org.dmis.appserver.ApplicationServer**

Main integration point for the DMIS Application Server.

**org.dmis.appserver.auth.DMISAuthenticator**

---

Authentication module for DMIS used by the AppServer to validate credentials. This module achieves integration with the AppServer by implementing to `IAuthenticator` interface.

### **org.dmis.appserver.client.desktop.ClientGateway**

Provides robust access to the AppServer from a client-size application such as the DMIS Desktop. This class is the modern equivalent to the `CMISServicesProxy` class that has been used heavily in prior versions of DMIS. This implementation is best suited for use by rich client applications rather than web service implementations.

### **org.dmis.appserver.client.thinrpc.ThinRPCClient**

The ThinRPCClient offers a lightweight alternative to the ClientGateway that can be efficiently used in web-based scenarios. It requires a direct connection to the associated AppServer, and thus offers no offline or reconstitution functionality, but its directness is conducive to a web services implementation. Semantically it parallels ClientGateway, making it easy to develop code that works in all contexts.

### **org.dmis.appserver.modules.servicemanager.\***

Definitions for important interfaces used by DMIS services. They include the following:

*IBackgroundService* – Background service interface

*IMessageService* – Message-oriented service interface

*IServiceHelper* – Interface providing major AppServer functions to services

*ISharedMemoryHelper* – Interface providing shared memory functions to services

### **org.dmis.appserver.module.servicemanager.ServiceManager**

Implementation of the Service Manager module, which is the container for DMIS services running on an AppServer.

### **org.dmis.core.foundation.CoreContext**

The core configuration class whose instance is created immediately after startup, passed around, and used by nearly all DMIS code. It provides access to configuration, database, logging, and other base functions for the context in which the code is running. Most code does not need to know where these settings originate, just that they are the ones to be used in the “current situation”.

### **org.dmis.launcher.\***

Launcher, or the “official entry point”, for all parts of DMIS. The launcher parses \*.*launch* files, which are profiles that describe entry points and configuration information, and launches the appropriate part of DMIS as configured.

---

### **org.dmis.messaging.\***

Messaging implementation, including definitions for pivotal classes such as:

Message – The base type for all DMIS messages

MessageRoutingHub – Main integration point for the messaging subsystem

RoutingID – Identifier used to perform logical addressing in DMIS

RoutingTable – Table used to track the layout of the DMIS network in real time

MessageQueue – Message queuing implementation

### **org.dmis.service.\***

Implementations of various services that run in the Service Manager container. All services implement either the `IBackgroundService` or `IMessageService` interface.

### **org.dmis.util.dmisconfig.\***

Code behind the `DMISConfig` utility.

### **org.dmis.util.serverdiag.\***

Implementation of the `ServerDiag` utility.

## **2.1. JDK Versions**

DMIS is currently designed to operate with JDK v1.5.x (also called Java 5). Since it uses new features offered by Java 5 that did not previously exist, the code will not compile under previous JDKs.

## **2.2. Basic Implementation Steps**

### **2.2.1. Define a Message API for the New Service**

Client-side applications and DMIS services will interoperate by agreement about the messages can be exchanged. For each message, the following minimum elements must be clearly designed and documented:

- Class of the message and any related messages (such as response messages),
- Message attributes (parameters) expected from the originator of the message according to the definition (or contract) of the API,
- Content that is returned or other activities that are generated,
- A brief example of where how the message might be used,
- Any other relevant notes.

---

Message code and message-specific beans should be placed in the **org.dmis.api.nnnnn** package, where **nnnnn** is the name of the associated service to which the message applies. Any common beans, ones potentially used across multiple services, can be placed in **org.dmis.api.beans** for common consumption.

### 2.2.2. Create Service Descriptors

There are two types of deployment descriptors that may be relevant to any given service:

#### **\*.svc (Service Descriptor) file**

A service descriptor is required for each service that is to run within the Service Manager container. It contains the class name of the service, pool sizes, and other symbolic information. In addition, and perhaps most significantly, it contains mappings that describe which messages the service is interesting in receiving. This portion of the descriptor replaces the functionality of the `isMessageRelevant()` in previous versions of DMIS.

#### **\*.wfd (Workflow Descriptor) file**

A workflow descriptor is *not* always required. It is used to describe how specific messages should be routed when no `DestinationID` is explicitly provided for an outgoing message. Since this is typically the case, these descriptors can have considerable control over message flow. If no workflow for a given message is explicitly defined, it follows the default resolutions path (typically sending messages to the next available server in the DMIS hierarchy).

The primary effect of these descriptor files is to move message routing decisions into a more dynamic form, rather than relying on it being fixed in code. This is important because the Distributed Services architecture requires more flexibility from services than can easily be implemented in code. Furthermore, service code can no longer make assumptions regarding the layout of the DMIS network as they have in the past. The descriptor file approach removes the necessity for such fixed assumptions.

### 2.2.3. Implement Service Code

Create a module that implements either the `IMessageService` or `IBackgroundService` interface. The code should be placed in the **org.dmis.service.nnnnn** package, where **nnnnn** is the name of the service under development. Each of these interfaces requires only a small handful of methods to be implemented.

## 2.3. Deployment Descriptors

### 2.3.1. \*.svc Descriptors (Service Configuration Files)

---

Each service must have a Service Descriptor (\*.svc) in order to be deployed to the Service Manager container. The naming convention for these files is:

SVC - *nnnnn.svc*

Where *nnnnn* is the name of the service being described. This convention makes it very easy to find a particular descriptor in a directory list, as shown by this example:

SVC - Authentication.svc  
SVC - FTP.svc  
SVC - GIS.svc  
SVC - Instant Messenger.svc  
SVC - Notification.svc  
SVC - TIE.svc  
SVC - Weather.svc

These files are formatted as XML documents, each with an expected layout and predefined elements.

```
<?xml version="1.0"?>
<services>
  <messageservice>
    ← General service-level parameters →
    <name>Displayable Name</name>
    <symbol>Symbolic Name</symbol>
    <description>Description</description>
    <class>classname</class>
    <poolsize>nnnn</poolsize>

    ← Load throttling for specific message classes. →
    <loadlimit>
      <limit>nnn</limit>
      <msg>
        <class>msgclass</class>
        <content>contentvalue</content>
        <attributes></attributes>
      </msg>
    </loadlimit>

    ← Message mappings →
    <message [class|correlatedclass]="messageclass">
      <attribute type=[ "attribute" | "content" | "tier" ]
        [ name="valuenam" ]
        value="expectedvalue"
        test=[ "equal" | "notequal" | "exists" ]/>
    </message>
  </messageservice>
</services>
```

**Figure 2: Service Descriptor (\*.svc ) File Format**

The general parameters include the name of the service, the name of the class that implements it, the poolsize to be used, along with other values. The *Symbolic Name* is a short, arbitrary, but unique string used for categorizing log entries and providing advanced queuing features.

---

A **<loadlimit>** element defines a ceiling for the number of messages of a certain type can be processed at any given time. The specific message to be throttled is identified by its class name, content value (expressed as a String), and the set of attributes that it uses (which follows the syntax of the `Message.validateProperties()` method described in this document). Other messages that are not specifically throttled have an upper limit defined by the **<poolsize>** specified for the service. In all cases, the **<poolsize>** is the absolute upper limit of concurrent processing for the entire service.

Each **<message>** element defines a type of Message that the service is designed to handle. It can filter messages based on a wide variety of criteria.

- The *class* or *correlatedclass* attribute specifies the class name of a message that is a candidate for the service. If *class* is used, the class name of the incoming message is matched – as would be expected at an intuitive level. If *correlatedclass* is used, however, then it will only match messages that are derivatives of *CorrelatedResponseMsg* and that have a matching *CorrelatedRequestMsg* type of *classname*. *This allows services to receive and process either requests and responses.*

The *messageclass* may include the wildcard (\*) character in order to match all messages in a given package hierarchy. When different mappings overlap due to the use of wildcards, the more specific one (with a more qualified *messageclass*) is always preferred.

- The **<attribute>** element describes a specific test to be performed on the incoming message that it must match in order to be mapped to the service. The element is optional and any number of tests can be defined. The “*tier*” *test type* can be used to determine the context of the AppServer (e.g. an EMBEDDED server, a LAN server, of the CENTRAL server).

Most services will use only a subset of these elements. The following example demonstrates how this file might look for the TIE service:

```
<?xml version="1.0"?>
<services>
  <messageservice>
    ← General service-level parameters →
    <name>TIE Service</name>
    <symbol>TIE</symbol>
    <description>TIE Service</description>
    <class>org.dmis.service.tie.TIEServer</class>
    <poolsize>16</poolsize>

    ← Load throttling for specific message classes. →
    <loadlimit>
      <limit>3</limit>
      <msg>
        <class>org.dmis.api.tie.TIECorrelatedRequestMsg</class>
        <content>retrieveall</content>
```

```

        <attributes></attributes>
    </msg>
</loadlimit>

← Message mappings →
<message class="org.dmis.api.tie.TIERemoteCorrelatedRequestMsg">
    <attribute type="tier"
        value="central"
        test="equal"/>
</message>

<message class="org.dmis.api.tie.TIECorrelatedRequestMsg"/>
<message class="org.dmis.api.tie.TIEUpdateRequestMsg"/>
<message class="org.dmis.api.tie.TIECorrelatedResponseMsg">
    <attribute type="tier"
        value="central"
        test="notequal"/>
</message>
</messageservice>
</services>

```

Notice that this descriptor describes (without any code) a lot of the work performed in code by the services themselves in previous versions of DMIS. This can virtually eliminate the need for service-level code that tests for a “parent server” and forwards messages in convoluted and relatively inflexible ways.

### 2.3.2. \*.wfd Descriptors (Workflow Configuration Files)

*Workflow descriptors* add an additional dimension to the dynamic routing nature of DMIS. It is used to describe how specific messages should be routed by default when no DestinationID is explicitly provided for an outgoing message. Since this is typically the case for messages originating from the DMIS Desktop, these descriptors can have considerable influence on the overall message flow.

Workflow descriptors are usually defined in parallel with Service Descriptors when they are both appropriate to the logic of a given service. If the default behaviors are acceptable for a service, then no Workflow Definitions are required; they are reserved mainly for special cases (such as exists for the DMIS Authentication Service).

As with Service Descriptors, Workflow Descriptors are XML files that contain particular elements.

```

<?xml version="1.0"?>
<workflow>
    <message [class|correlatedclass]="msgclass"
        pattern=[ "lan,central,embedded" |
            "lan,topdown" |
            "topdown" |
            "bottomup" ]
        [ notavailable= [ "defer", "discard" ]

```

---

```
cc=[ "upstream" , "downstream" ] />
</workflow>
```

**Figure 3: Workflow Descriptor (\*.wfd) File Format**

Each **<message>** element defines a type of Message that requires special handling in the event that a message is not already addressed to a specific entity.

- The *class* or *correlatedclass* attribute specifies the class name of a message that is to be directed. If *class* is used, the class name of the incoming message is matched – as would be expected at an intuitive level. If *correlatedclass* is used, however, then it will only match messages that are derivatives of *CorrelatedResponseMsg* and that have a matching *CorrelatedRequestMsg* type of *classname*. *This allows either requests and responses to be handled.*

The *messageclass* may include the wildcard (\*) character in order to match all messages in a given package hierarchy. When different mappings overlap due to the use of wildcards, the more specific one (with a more qualified *messageclass*) is always preferred.

- The **<pattern>** attribute describes the pattern in which the message is to be distributed by default. The standard patterns that are listed are defined in the *NetworkTopology.wfd* descriptor, and though an \*.wfd file can define its own custom patterns as well, the predefined ones are nearly always sufficient since they were modeled after the historical needs of DMIS.
- When the node is not currently connected to the DMIS network, the **<notavailable>** attribute allows the resolution of the given message's destination to be delayed until there is connectivity, or for the message to be selectively discarded. This allows the addressing logic to make a more informed choice based upon the logical network topology detected via the routing tables. In nearly every case, the default setting for this value is ideal.
- The **<cc>** attribute causes Carbon Copies of the given message to be automatically generated and sent according to the specific pattern (either "up the hierarchy" or "down the hierarchy"). For each duplicate message, the *isCarbonCopy()* flag (discussed in this document along with the Message class) is set enabling the target to differential between the original and a copy. This feature can eliminate much of the forwarding code that is built into services aimed at previous versions of DMIS.

Most applications will use only a subset of these elements. The following example demonstrates how this file might look for the Authentication service:

```
<?xml version="1.0"?>
<workflow>
  <message profile="*.RequestAuthenticationMsg"
    pattern="lan,topdown"/>
</workflow>
```



---

### 2.3.3. Dynamic Deployment and Undeployment

Services can now be deployed and disabled dynamically. To deploy a new service against an AppServer that is already running simply place the corresponding \*.svc descriptor in the configuration path. The Service Manager's deployment scanner will recognize the new file and install the service on-the-fly.

A convenient way to disable a running service is to simply rename the associated \*.svc file in the configuration path. For example, changing this filename...

```
svc - Alerts.svc
```

to this...

```
svc - Alerts.svc.disabled
```

...will cause the Service Manager's deployment scanner to undeploy the associated service. If the file is then renamed back to its original name, the Service Manager will attempt to redeploy it in real time. By convention, the *.disabled* extension is used for this purpose. This also allows service descriptors to be developed in place without necessarily activating them

Since Workflow Descriptors (\*.wfd) are handled in a similar manner, the same conventions apply to those files as well.

## 2.4. Client Stubs

Your service will be accessible to any application that uses the *ClientGateway* or *ThinRPCClient* classes. The DMIS Desktop provides a pre-built framework in which this is readily available. For testing purposes, it is relatively simple to create a client stub implementation that you can use to test a service:

```
// Create a ThinRPC instance for testing.
ThingRPCClient rpcClient =
    new ThinRPCClient( "hostname", 3101);

// Authenticate the operator.
rpcClient.authenticate("user1", 2);

// Send messages to your service.
MyServiceMsg msg = new MyServiceMsg();
...
rpcClient.sendMessage(msg);
```

---

### Figure 4: Simple DMIS Client Stub Example

Once installed and working according to the standard outlined in this document, your service should be able to receive message from just this little bit of code.

## 2.5. The CoreContext Class

The *CoreContext* class is new but is now possibly the most ubiquitous object type in DMIS. It is created at startup, passed around, and used by nearly every module to some extent. Though simple in concept, the *CoreContext* class is the backbone of many of the more advanced features in the new DMIS framework. Its functionality enables, among other thing, the following features:

- 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 parameters from anywhere in DMIS,
- Convenient time-keeping facility that is NTP-synchronized where possible,
- 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 created prior to the point where applications and services ever receive control. In virtually all cases, a *CoreContext* instance is provided by the framework.

### 2.5.1. Variable Access

*CoreContext* allows all modules in DMIS to access key system-wide variables. Some variables, such as *home*, *profilename*, and those related to *Java system variables*, are set automatically at startup and are thus always available. All system environment variables are available as well:

```
// Fetch the DMIS home directory.
System.out.println( coreCtx.getValues.getString("home") );

// Fetch the name of the current configuration profile.
System.out.println( coreCtx.getValues.getString("profilename") );

// Fetch the JAVA_HOME system variable.
System.out.println( coreCtx.getValues.getString("JAVA_HOME") );
```

... Produces the following output ...

```
C:\Program Files\DMIS\
Desktopfull
c:\j2sdk1.4.2_06
```

---

The `getValues()` method returns the map in the form of a `CoersionMap` (implemented in `org.dmiservices.common.foundation.util.CoersionMap`) that allows values to be easily converted and retrieved in native forms (such as *int*, *long*, *double*, etc.) even if they are actually stored as Strings.

Values can be read and set by any DMIS application or service. Depending upon the situation, due to the sandboxing performed by the DMIS AppServer, values set by a given module may or may not be visible to other modules.

### 2.5.2. Command-Line Parameter Access

An extension of variable access is the ability to retrieve command-line parameters that were passed as DMIS was started. Parameters on the command line are parsed in *-key value* form, where *key* becomes the name of the parm and *value* becomes its content. All command-line entries that follow this form are converted into accessible values; any others are not.

Command-line variable are accessed in the same way as other values, as illustrated in the previous section, except that they are named using a special convention. For any given parameter name *nnnn*, the following variables are defined for it:

|                        |  |
|------------------------|--|
| <b>parm.nnnn</b>       | = Value of the given parameter (or its first occurrence)       |
| <b>parm.nnnn.count</b> | = Number of times the parameter listed on the command line     |
| <b>parm.nnnn[idx]</b>  | = Value of a specific occurrence of the parameter (zero-based) |

These conventions allow a variety of operations and tests to be performed on the values passed into the program from anywhere within DMIS. The following example fetches a simple command-line value that was passed into the DMIS VM:

```
CoersionMap envMap = coreCtx.getValues();

// Determine if a 'flag1' parameter was passed.
if (envMap.containsKey("flag1")) {
    System.out.println( "flag1 was specified" );
}

// Get the value of 'parm2', or use a default of "<default>".
System.out.println( envMap.getString("parm2", "<default>" );

... Assuming the following command-line contains:

    -flag1

... Produces the following output ...

Flag1 was specified
<default>
```

---

This more elaborate example reads a parameter that appeared multiple times on the command line. The CoreContext presents this through its value table as an array:

```
CoersionMap envMap = coreCtx.getValues();

// Determine if a given parameter was passed.
if (envMap.containsKey("parm.mykey")) {
    // Dump out a list of occurrences on the command line.
    int count = envMap.containsKey("parm.mykey.count");
    System.out.println( "Occurrences: " + count);

    for (int i=0; i<count; i++) {
        String val = envMap.get("parm.mykey [" + i + "]");
        System.out.println( val );
    }
}
```

... Assuming the following command-line contains:

-mykey val1 - mykey val2 - mykey val3

... Produces the following output ...

```
Occurrences: 3
val1
val2
val3
```

Modules can then process arbitrary command-line values that hold specific meaning to it (and are perhaps intended solely for that module's use).

### 2.5.3. DatabaseManager

The **dbMgr** member of CoreContext encapsulates the *DatabaseManager* instance that is feeding connections to DMIS. Applications and services can use this object to obtain connections:

```
Connection conn = coreCtx.dbMgr.getConnection();
```

Connections are created according to the parameters specified in the *DatabaseManager.xml* file. The DMIS Desktop is normally configured against a McKoi database, while the CENTRAL AppServer usually leverages Oracle. A LAN-level AppServer may use either one (or perhaps something else entirely different) depending upon requirements.

DatabaseManager pools connections in order to enhance the performance of DMIS. When connections are obtained and released, they are returned to a pool where they may be used again. This avoids the overhead of creating new connections each time database operations are required, and in most cases produces a ready connection without further delay.

---

Ideally, services in DMIS do not hold database connections open for longer periods of time. The general pattern is:

1. Obtain a connection,
2. Perform work,
3. Release the connection as soon as possible

There are several reasons for this. First, returning connections to the pool allows other code to use them and reduces the overall number of connections to the database. In addition, database connections on the DMIS network expire after a certain time period and may not be valid when a service needs it. Connections from the pool are kept fresh by DatabaseManager.

#### 2.5.4. DataHelper

The **dataHelper** member of CoreContext offers shorthand methods for modules to perform SELECTs, INSERTs, UPDATEs on the database. Connections are obtained from the **dbMgr** member, so it is an additional facility that rides upon the parameters established by the DatabaseManager. The **dataHelper** facility offers the following major features:

- It can simplify most JDBC-related code,
- It manages the creation and destruction of database connections,
- It handles nested database-related activity smoothly efficiently,
- It provides convenient wrappers for common operations.

Assume, as an illustration, that we wish to select row from a specific table. Traditional JDBC-oriented code in DMIS for this might look like this:

```
// Get connection from DatabaseManager.
Connection conn = coreCtx.dbMgr.getConnection();

// Get a statement from the connection
Statement stmt = conn.createStatement();

// Execute the query
ResultSet rs = stmt.executeQuery("SELECT * FROM MyTable");

// Loop through the result set
while( rs.next() )
    System.out.println( rs.getString(1) );

// Close the result set, statement and the connection
rs.close();
stmt.close();
conn.close();
```

In many cases, the code is designed to reuse connections such that nested database operations (perhaps organized as independent child functions) can ‘ride’ on the same connection. This simple and often effective approach does have a few complications:

- 
- It requires some extra logic to determine whether a connection already exists and whether it is still usable.
  - Since any given method cannot be sure that it is the last to use the connection, it can be difficult to determine when a connection can be closed. In many cases the connection is simply left open for the next request, whose timeframe is often indeterminate in DMIS, rather than being returned to the ‘pool’ for potential reuse.
  - Data access modules that use this approach are generally not thread-safe. This means that a different instance of the data access module must be generated for each higher level module (e.g. a DMIS service) that requires its services. Combined with the fact that services are often allocated in pools, and each data access module may keep its connection open, this can result in a nontrivial resource drain.

None of these obstacles are insurmountable, and in many cases these approaches may be the best way, but **dataHelper** does offer some features that can simplify these problems. The following code is an example of how the same query can be performed using DataHelper:

```
// Connect.
coreCtx.dataHelper.connect();

// Execute the query.
ResultSet rs = coreCtx.dataHelper.query("SELECT * FROM MyTable") ;

// Loop through the result set.
while( rs.next() )
    System.out.println( rs.getString(1) ) ;

// Close everything we've used here.
coreCtx.dataHelper.close();
```

The **dataHelper** version of this query requires less housekeeping code (fewer setup and cleanup steps) for both connections and statements. The helper keeps track of all resources used and in most cases can clean them up automatically. This example is small, but the effect is much more pronounced when more complex operations (such as multiple queries or prepared statements) are implemented.

A major feature of **dataHelper** is that connection states are managed at the thread level (by leveraging ThreadLocal storage in Java). This means that any code using dataHelper is thread-safe by nature and, barring other circumstances, the same accessor instance can be used concurrently by multiple processes. In addition, this thread-oriented management scheme simplifies the problem of nested functions and connection cleanup. Consider the following example:

```
void mainFunc() {
    try {
        // Connect to the database.
```

---

```

        coreCtx.dataHelper.connect();

        // Do some work that requires calls to child methods.
        for (int=1; i<=10; i++)
            subFunc(i, db);

        // Conclude our work.
        coreCtx.dataHelper.commit();
    }
    catch (SQLException sqle) {
        coreCtx.dataHelper.rollback();
    }
}

void subFunc(int i, JDBCHelper db) {
    try {
        // Connect to the database.
        coreCtx.dataHelper.connect();

        // Do some work.
        db.exec("INSERT .....blah.....");

        // Conclude our work.
        db.commit();
    }
    catch (SQLException sqle) {
        db.rollback();
    }
}

```

Even though both `mainFunc()` and `subFunc()` call `connect()`, because they are on the same thread they will automatically use the same connection. The `dataHelper` facility understands this nesting construct and behaves intelligently. The `subFunc()` method can be used in the nested context as illustrated here or in a stand-alone fashion, and in all cases connections are handled efficiently. When the highest-level connection in the thread closes its connection (via `commit()`, `rollback()`, or `close()`), the connection is cleaned up at the physical level. There is thus no need to leave connections open for any longer than they are needed.

#### 2.5.5. **SerializationManager**

The **serMgr** member is an interface to the DMIS object serializer, used in transmission and persistence of various object instances (most notably, `Message` instances). In the past, several types of serializers were provided, each with different characteristics. In this version of DMIS, however, the JDK-based serializer is always used. The **serMgr** instance is offered as a bridge to existing code and as a shortcut to simplify serialization tasks.

#### 2.5.6. **Logger**

---

The **logger** member allows DMIS code to generate log entries for information and debugging purposes. Generating a log entry is an easy matter of invoking the correct method on the Logger that correlates to the threshold of the activity:

```
// Sample logger commands.
coreCtx.logger.debug("debug value: 1");

coreCtx.logger.info("Module xxx started");

coreCtx.logger.warn("Unexpected condition xxxx");

coreCtx.logger.error("Error while xxxxx");

coreCtx.logger.fatal("Big problems - terminating");
```

Log output is normally directed to a log file placed in the **.logs** directory, though log output can be directed to other locations. The log filename are designed to simplify location of the correct log when viewing the directory at a high level:

```
DMIS [central] Started 2005-02-10 17.05.05.log
DMIS [central] Started 2005-02-11 10.00.54.log
DMIS [desktoplite] Started 2005-02-11 10.01.20.log
```

Generation of log output is controlled by the *Logger.xml* configuration file. This file can be edited manually or graphically through the DMISConfig utility. Features of the logging subsystem include:

- Output to text files, with flexible options regarding their destination,
- Console output to simplify viewing logs in real time,
- Flexible filtering using *Category Selectors* to achieve the desired granularity,
- Stack analysis that allows source code locations to be tied to specific log entries,
- SMTP output that can be used to notify an administrator when certain events occur

These features imply that, dependent upon settings, log output may or may not be generated for a given Logger method call. In addition, any output that is written may be directed to one or more handlers (and thus appear in multiple forms of output).

### 2.5.7. Clock

The **clock** member provides access to time functions tied to the most accurate measure currently available in the given context. In addition to keeping time, the clock also offers methods to convert between time zones. These functions include:

```
public abstract class TimeFetcher {
    public static final int ONE_SECOND = 1000;
    public static final int ONE_MINUTE = ONE_SECOND * 60;
    public static final int ONE_HOUR = ONE_MINUTE * 60;
```



---

```
public static final int ONE_DAY = ONE_HOUR * 24;
public long getMillisecondTime();
public Date getDate();
public long getOffset();
public Timestamp getTimestamp();
public Time getTime();
public long getMillisecondTime(TimeZone timeZone);
public Date getDate(TimeZone timeZone);
public Timestamp getTimestamp(TimeZone timeZone);
public Time getTime(TimeZone timeZone);
public long normalize(long millisecondTime,
                     TimeZone fromTimeZone,
                     TimeZone toTimeZone);
public Date normalize(Date date,
                     TimeZone fromTimeZone,
                     TimeZone toTimeZone);
public Time normalize(Time time, TimeZone fromTimeZone,
                     TimeZone toTimeZone);
public Timestamp normalize(Timestamp timestamp,
                          TimeZone fromTimeZone,
                          TimeZone toTimeZone);
public java.sql.Date normalize(java.sql.Date date,
                              TimeZone fromTimeZone,
                              TimeZone toTimeZone);
```

The **clock** object is abstract in nature and thus the true source of its measurement is dependent upon context. For the DMIS Desktop and all DMIS Application Servers, the NTP (Network Time Protocol) is used whenever possible – making the clock extremely accurate. The code behind it is based on the *TimeSyncManager* class that existed in previous versions of DMIS, applied to the common timekeeping interface used by CoreContext.

*Note:* Due to the fact the this clock is independent of `System.currentTimeMillis()` function, it may or may not produce the same value as the clock running on the local machine. When measuring intervals between points in the same program, relative times may be sufficient and the timekeeper may not be crucial. Where an accurate, absolute time is required, however, the **clock** member of CoreContext is the most reliable timekeeper.

---

## 3. Messaging Design Considerations

### 3.1. Message Structure

Messages are a fundamental unit in DMIS. They can represent requests, responses, acknowledgements, notifications, and many other types of constructs. All messages in DMIS are based on the abstract *org.dmis.messaging.core.Message* class, and thus have similar semantics.

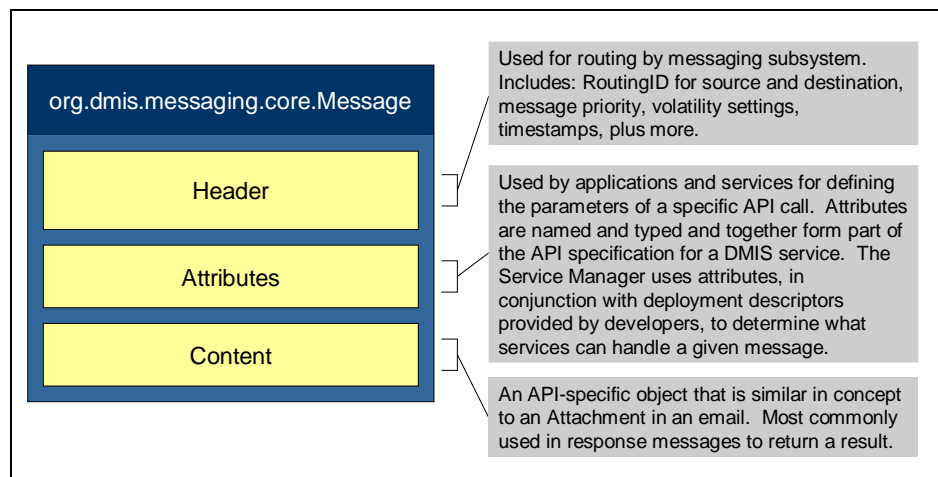
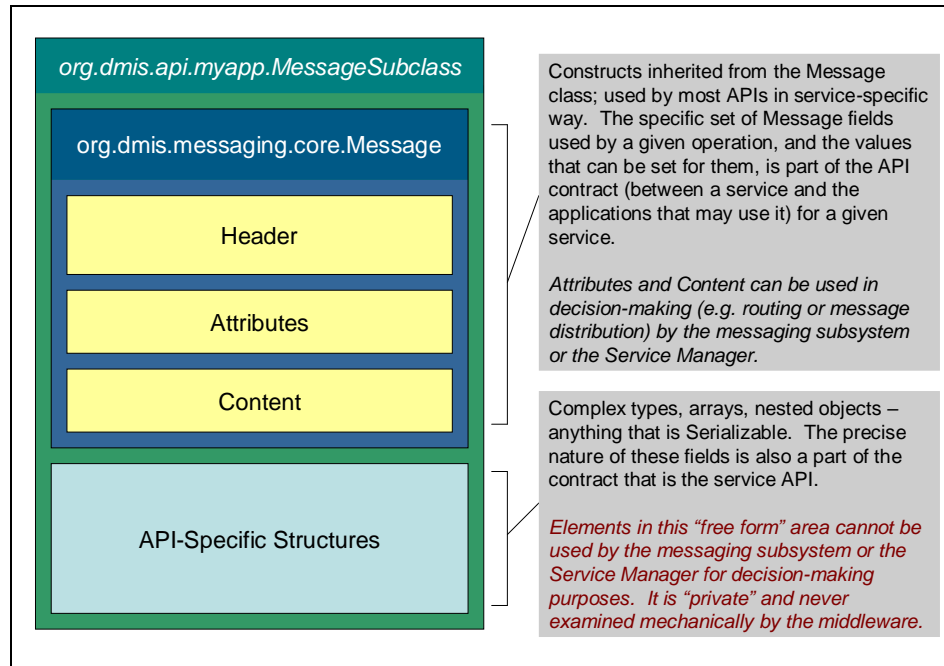


Figure 5: Layout of the *Message* Class

#### 3.1.1. Message Hierarchies

Applications and services create subclasses of the Message class to encapsulate specific functions and parameters within their defined APIs. Subclasses can also be used to accommodate more complex structures that otherwise may not be suited for the simple Attributes and Content found in the Message class.



**Figure 4: Subclassing a Message for an API function**

Generally, one subclass exists for each major operation supported by the service API. A Chat Service, for example, might have a set of message classes such as the following:

```
org.dmis.api.chat.RequestEnterChatRoomMsg
org.dmis.api.chat.RequestExitChatRoomMsg
org.dmis.api.chat.RequestPostChatMsg
org.dmis.api.chat.RequestGetChatRoomMembers
```

Each message class can then include an appropriate set of attributes and methods to describe the operation being requested. This approach allows a set of messages for an API to become self-documenting.

### 3.1.2. Appropriate Subclassing

New message classes should be derived from the most appropriate class in the existing hierarchy. *Messages are not usually derived directly from Message*, although this is not prohibited. There is a predefined set of subclasses that encapsulate the most common messaging scenarios, such as the *CorrelatedRequestMsg* and *CorrelatedResponseMsg* routinely used in RPC-centric calls. Some portions of the messaging subsystem are specially designed to handle messages of key subclasses of message and rely on the expected nomenclature.

### 3.1.3. Naming Conventions

There are some naming conventions that have been established for messages. Some of these conventions are new for this version of DMIS.

- **Message class names end with Msg.** This is simply a convention that has been agreed upon in order to balance clarity with an attempt to keep class names reasonably short.

- **Message classes and beans are kept in a dedicated package area `org.dmis.api.xxxx`**, where `xxx` is the name of the API (example: `org.dmis.api.chat`). Keeping messages in a dedicated area makes it easier to see the broader set of API functions offered by DMIS. It also makes the system more modular and easier to deploy, since - in the ideal case - messages (and any associated beans) are the only objects shared between the DMIS Desktop and DMIS AppServers. Separating the messages and associated beans makes them easier to package in a modular fashion
- **Consistent class names.** The following table suggests how you might name your message subclasses to maintain clarity and consistency based on the message type:

| Message Type                 | Suggested Naming Pattern For Derived Classes | Example                        |
|------------------------------|--|--------------------------------|
| <b>CorrelatedRequestMsg</b>  | <b>Request<code>xxx</code>Msg</b>            | RequestJoinChatSessionMsg      |
| <b>CorrelatedResponseMsg</b> | <b>Response<code>xxx</code>Msg</b>           | ResponseJoinChatSessionMsg     |
| <b>PostNotificationMsg</b>   | <b>Notify<code>xxx</code>Msg</b>             | NotifyOperatorJoinedSessionMsg |
| <b>PostRequestMsg</b>        | <b>Post<code>xxx</code>Msg</b>               | PostChatMsg                    |
| <b>ReturnReceiptMsg</b>      | <b>Received<code>xxx</code>Msg</b>           | ReceivedAttachedFileMsg        |
| <b>SynchronizationMsg</b>    | <b>Sync<code>xxx</code>Msg</b>               | SyncPermissionDataMsg          |
| <b>UpdateNotificationMsg</b> | <b>Notify<code>xxx</code>Msg</b>             | NotifyTIERRecordUpdatedMsg     |
| <b>UpdateRequestMsg</b>      | <b>Update<code>xxx</code>Msg</b>             | UpdateTIERRecordMsg            |

## 3.2. Common Messaging Patterns

The DMIS messaging arena is not completely unstructured, in that we can observe specific messaging patterns that occur and that the transport layer must support. Many existing parts of message hierarchy and subsystem already embody these basic patterns. Other possible messaging patterns may be variations, applications, or combinations of the basic patterns described here, and can be implemented using the prescribed infrastructure.

### 3.2.1. Request-Response

The Request-Response pattern is an RPC-like dialogue that represents a request for processing by the client, and an expected return value from the server that is coupled to the original request via a `CorrelationID`. The `CMISClientGateway` includes functionality to simplify the correlation of Response message with Request message based on the `CorrelationID`.

The client formulates a Message based upon operations that are available as part of the API defined by the corresponding service. Operations are keyed by the name of the message class that the service has derived from the abstract Message base, and may include other attributes and Content to further specify the request. The message content is validated by the service on whom the operation is invoked, and invalid messages are rejected and kicked back to the client as an exception. The exception propagation mechanism also in this document.

All messages in DMIS are processed asynchronously, and the Response portion of the Request-Response cycle is returned to the client in the form of another message which is coupled to the

---

original request. The CMISClientGateway provides facilities for applications to block until the response is received or until a certain timeout threshold has been reached. Note that, in an asynchronous system such as DMIS, the client could wait for a significant period of time before receiving a response, during which time the user might be waiting with hands tied. For this reason, blocking RPC operations must be approached with caution.

### **3.2.2. Post**

In the Post pattern, the client sends a message to the server but does not necessarily expect a response. This pattern may be used in distributing information to other clients, or posting of information to a database. Although a response is not mandatory for a post, and it depends upon the functionality implemented in the Service for handling the message, a response can come in the form of Acknowledgement messages in which the server notifies the client that the information has been received and processed. In this case, the Acknowledgement message can be coupled to the original Post message by a unique number (such as the CorrelationID).

### **3.2.3. Acknowledgement**

When the transaction is received by an application or by a service running on a DMIS AppServer, an Acknowledgement message may be sent to confirm that the message was received and the operation it represents is accepted. An acknowledgement is normally not sent until the recipient (e.g. the running service) (1) has received the transaction and all of the associated data, and (2) has successfully processed it according to the agreement between application and service described by a message-oriented API.

### **3.2.4. Notification**

Notification messages are sent from a service running on a DMIS AppServer (generally to the desktop but not necessarily) to inform applications of a change in state or that information is waiting to be retrieved. In some cases the notification message itself may carry all of the information relevant to describing the event. Other times, such as when the backing dataset is large, the recipient may then use the Request-Response pattern to retrieve additional the information from the corresponding service through defined API. The situations in which a client will receive Notification messages, and the information that each message may carry, will vary along with the Services available at the AppServer.

### **3.2.5. Return Receipt**

When an application or service sends a message to another entity, the originator of the message may wish to receive confirmation that the message was received. The usage of this pattern is heavily application and service dependent and must be incorporated in the service API.

### **3.2.6. Synchronization**

Synchronization messages are used for communication between different modules (or layers) in the DMIS AppServer hierarchy. They are normally sent when some function results in data changes to which other tiers might need to react. The precise usage of these kinds of messages is heavily dependent upon the particular services and applications.

---

## 4. Message Class Details

The following table shows the major methods belonging to the Message class. Each of these is described in detail in the sections that follow.

### Header-related

```
getCorrelationID()
setCorrelationID(newValue)
getDestinationID()
setDestinationID(newValue)
getExpirationTimestamp()
setExpirationTimestamp(newValue)
getIsCarbonCopy()
setIsCarbonCopy(newValue)
getIsVolatile()
setIsVolatile(newValue)
getMessageKey()
setMessageKey(newValue)
getPriority()
setPriority(newValue)
getReplyToID()
setReplyToID(newValue)
getSourceID()
setSourceID(newValue)
getTraceID()
```

### Attribute Operations

```
attributeExists(name)
validateProperties(propertyList)
clearAttributes()
getAttributeNames()
getAttributeBoolean(name)
setAttributeBoolean(name, newValue)
getAttributeByte(name)
setAttributeByte(name, newValue)
```

```
getAttributeDouble(name)
setAttributeDouble(name, newValue)
getAttributeFloat(name)
setAttributeFloat(name, newValue)
getAttributeInt(name)
setAttributeInt(name, newValue)
getAttributeLong(name)
setAttributeLong(name, newValue)
getAttributeObject(name)
setAttributeObject(name, newValue)
getAttributeShort(name)
setAttributeShort(name, newValue)
getAttributeString(name)
setAttributeString(name, newValue)
clearContent()
getContent()
setContent(oNewContent)
```

### Debug Support

```
clearActivityLog()
getActivityLog()
recordActivity(text)

getDisplayProfile()
getVisitList()
```

### Housekeeping

```
clone()
clone(keepExistingTraceID)
```

Figure 5: Message class methods

### 4.1. Header-Related Operations

#### 4.1.1. getCorrelationID( ) / setCorrelationID( )

##### Signature:

```
long getCorrelationID();
void setCorrelationID(long newValue);
```

##### Usage:

Sets or returns the *CorrelationID* value for the message, which refers to the TraceID of another Sets or returns the *CorrelationID* value for the message, which refers to the TraceID

---

of another message. The `CorrelationID` is used most frequently in request-response (RPC) scenarios to determine if an incoming message is related to a message that was earlier sent. In an RPC-style situation, this incoming message will then normally be treated as the *response* from the requested operation.

```
if ( incomingMsg.getCorrelationID( ) == earlierSentMsg.getTraceID( ) ) {  
    System.out.println("The response has arrived!");  
}
```

Because it is such a common idiom in DMIS, the middleware has special support for the RPC paradigm. The *CorrelatedRequestMsg* and *CorrelatedResponseMsg* are designed to simplify request-response patterns by providing default behaviors conducive to most situations. The *CorrelatedResponseMsg*, for example, is automatically correlated to the matching *CorrelatedRequestMsg*. Further, the *CorrelatedResponseMsg* class allows the original request message to be accessed directly using the *getRegarding( )* method in order to simplify state management.

```
if ( incomingMsg.getRegarding() instanceof AuthenticationRequestMsg ) {  
    // Handle the results of the authentication process.  
    ...  
}
```

Operations following the RPC paradigm should use *CorrelatedRequestMsg* and *CorrelatedResponseMsg* in order to benefit from all of the existing functionality and to ensure proper handling.

In non-RPC scenarios, the relationship between the `CorrelationID` and the `TraceID` of another message can be much looser. For example, there may be multiple messages that list the same `CorrelationID` if the originating operation generates a series of messages as output (such as for distributing notifications). In these cases the precise relations of these fields is defined by the service being used.

#### 4.1.2. `getDestinationID( )` / `setDestinationID( )`

##### Signature:

```
RoutingID getDestinationID();  
void setDestinationID(RoutingID newValue);
```

##### Usage:

The `DestinationID` is used to specify the target for a given message. It is a `RoutingID`, in the form described by section entitled [Intelligent Message Routing](#), that addresses either a specific entity (any DMIS installation including Desktops and AppServers) or a category of entity within the context of the current entity (e.g. the LAN server for the operator's COG).

---

```
// Create a message and initialize it for the call we are making.
Message myMsg = new MyMessageClass ( );
...

// Send it to the nearest LAN server.
myMsg.setDestinationID(RoutingConstants.ROUTINGID_LAN_SERVER);
gateway.sendMessage(myMsg);
```

Applications running on the desktop typically do not provide specific DestinationIDs for messages, since the client applications cannot always know what servers exist to be addressed in a given deployment (e.g. EMBEDDED, LAN, CENTRAL, etc). When a DestinationID is not specified, it is resolved automatically based on *workflow* rules that are provided by the application developer (or the developer of the service being used by the application). Workflows are simply deployment descriptors that are read by the messaging subsystem based on the message criteria and applied according to the situation.

```
-----[ Sample Authentication.wdf file ]-----

<?xml version="1.0"?>
<workflow>
  <!-- Authentication routing is different than the default.
        Authentication is performed at the LAN level first
        if defined, then the CENTRAL level, then locally if
        all else fails. -->
  <message profile="*.RequestAuthenticationMsg"
    pattern="lan,topdown"
    notavailable="defer"/>
</workflow>
```

Workflows are discussed in more detail in this document.

It is very important to note that the DestinationID is a *logical* address, usually for a specific Operator or AppServer. Even if the DestinationID refers to an entity several “steps away” or across any number of levels in the DMIS hierarchy, the application does not need to know how they are physically connected. The messaging subsystem will find the entity corresponding to the DestinationID and route it there if at all possible, crossing whatever “hops” are needed.

#### 4.1.3. getExpirationTimestamp( ) / setExpirationTimestamp( )

##### Signature:

```
long getExpirationTimestamp();
void setExpirationTimestamp(long newValue);
```

##### Usage:



---

Messages can be set to expire at a given time expressed in UTC. When that time is reached, the message will be discarded by whatever entity controls it at that time. The message will not be delivered to its specified destination if it has not already reached it.

#### 4.1.4. `getIsCarbonCopy()` / `setIsCarbonCopy()`

##### Signature:

```
boolean getIsCarbonCopy();  
void setIsCarbonCopy(boolean newValue);
```

##### Description:

Services in DMIS have traditionally included code to synchronize events between the same service running at different levels in the hierarchy. A common idiom in DMIS services has been:

```
(sample service code snippet)  
  
public void onMessageReceived(Message msg) {  
    // Handle an update request (such as a Post operation).  
    if (msg instanceof MyUpdateMsg) {  
        // Attempt to pass the request to higher server if possible.  
        // With some exceptions, requests are usually serviced  
        // at lower levels only when needed (perhaps due to  
        // being offline). This logic is hard-coded in the service.  
        if (parent AppServer online) {  
            forward message to parent  
        }  
        // If this is the highest reachable server then process  
        // the message inform the sender of any updates for  
        // synchronization purposes.  
        else {  
            process message  
            send response to client  
            send MyUpdateMsgNotification to forwarder // so it can also  
                                                         // record changes  
        }  
    }  
    // Handle a sync notification from a higher server that actually  
    // processed the request. This gives us a chance to update our  
    // own internal state (e.g. the database) as possible.  
    else if (msg instanceof MyUpdateMsgNotification) {  
        update our internal state to keep our DB in sync w/ master  
    }  
}
```

This code is not only convoluted, but in many cases it is not very flexible due to other assumptions that the code makes. The Distributed Server model requires services to be more dynamic in nature in order to function effectively in a variety of configurations that cannot be determined until runtime.

---

Fortunately, the enhanced messaging subsystem offers features to simplify this type of construct while at the same time making it more modular and flexible. The CarbonCopy attribute is a part of that feature set.

Services can now specify, in a workflow descriptor file (\*.wfd), that when a given message is sent to a service running on a specific AppServer, a CarbonCopy should automatically be sent to the services running in parallel at other locations in the hierarchy. Those copies are clones of the original message except that the IsCarbonCopy flag is set to TRUE. This feature allows all levels of the service to simultaneously be notified (using a single message) of the operation and also whether it should actually respond or just manage any related state information it is holding.

```
public void onMessageReceived(Message msg) {  
    // Handle an update request (such as a Post operation).  
    if (msg instanceof MyUpdateMsg) {  
        // If not a CarbonCopy, then handle the request and response.  
        if (!msg.getIsCarbonCopy) {  
            process message  
            send response to client  
        }  
        // CarbonCopy of a request actually handled at another level.  
        // Just update our internal state.  
        else {  
            update our internal state based on the request  
        }  
    }  
}
```

In some cases this facility is not sufficient to handle state management between instances of services running across the hierarchy. For example, updating state information may require the results of the operation performed at another level and the CarbonCopy should be made of the result, not the request.

In these situations, methods of the IServiceHelper class (described in this document) can be used to implement more complex rules fairly easily.

#### 4.1.5. `getIsVolatile()` / `setIsVolatile()`

##### Signature:

```
boolean getIsVolatile();  
void setIsVolatile(boolean newValue);
```

##### Usage:

This flag indicates whether the message can be treated as "volatile". A volatile message is non-critical (or may not have meaning beyond the scope of the current session) and is not persisted when added to a message queue. The purpose of this option is to allow applications and services to notify the middleware that some messages can be treated as expendable, and

---

no logic depends upon its sanctity. As a result, the Guaranteed Delivery feature of the messaging subsystem does not apply to these messages since they could be lost in the event of system failure. This flag should always be used with care.

It is important to note that all derivatives of *CorrelatedRequestMsg* are marked volatile by default. The driving assumption is that, given an RPC-style operation, the entire operation is meaningless if the requesting application terminates. An *CorrelatedResponseMsg* follows the volatility setting of the corresponding request by default. When this default behavior is not appropriate for a given situation, the flag can be simply reset in the application or in a derived message class.

#### 4.1.6. `getMessageKey()` / `setMessageKey()`

##### Signature:

```
String getMessageKey();  
void setMessageKey(String newValue);
```

##### Usage

Retrieves or assigned the "key" associated with this message. The key is an arbitrary unique identifier, chosen and formulated by an application or service, that refers to a message symbolically. When a message is added to a queue (such as when a message is sent), it replaces and immediately expires any other message already in the queue that has the same key. For messages containing transient status information, this ensures that only the most recent message is in a given queue. Note that, if a message is keyed and sent, the new keyed message will only replace the old version if the old version is still in the given queue (such as in offline mode). If the original message has been transmitted to another location, both versions of the message may be sent. Therefore, applications and services that use this feature must be aware that keying a message does not guarantee uniqueness.

#### 4.1.7. `getPriority()` / `setPriority()`

##### Signature:

```
int getPriority();  
void setPriority(int newValue);
```

##### Usage:

Sets or returns the Priority of the message, which is a constant that affects the order in which messages are queued, routed, and processed. The value is normally expressed using one of the `PRIORITY_XXX` constants of the Message class. Possible values include `PRIORITY_LOW`, `PRIORITY_NORMAL`, and `PRIORITY_HIGH`.

```
// Send a low priority message.  
Message msg1 = new MyMessage();  
msg1.setPriority(PRIORITY_LOW);  
gateway.sendMessage(msg1);
```

---

```
// Send a high priority message. This message is likely to be
// processed first even though it was created and sent after the
// first message.
Message msg1 = new MyMessage();
msg1.setPriority(PRIORITY_HIGH);
gateway.sendMessage(msg1);
```

By default, messages are given a priority of `PRIORITY_NORMAL`. Message priorities are not fully deterministic but higher priority messages will, on average, be handled ahead of lower priority ones. Messages within the same priority level are generally serviced in the order in which they are generated or sent.

#### 4.1.8. `getReplyToID()` / `setReplyToID()`

##### Signature:

```
RoutingID getReplyToID();
void setReplyToID(RoutingID newValue);
```

##### Usage:

For RPC-style messages, the `ReplyToID` is the `RoutingID` of the entity that should receive the response. In most cases, this is the same value as the `SourceID` though not necessarily. The *CorrelatedResponseMsg* automatically chooses the `ReplyToID` based on the `SourceID` of the corresponding *CorrelatedRequestMsg*, but this value can be overridden as required.

#### 4.1.9. `getSourceID()` / `setSourceID()`

##### Signature:

```
RoutingID getSourceID();
void setSourceID(RoutingID newValue);
```

##### Usage

The `SourceID` is a `RoutingID` that specifies the originator of a given message. It is normally filled out automatically when a message is sent by an application or service, but it can be provided explicitly in rare cases. Because the `SourceID` is in the same format as a `DestinationID`, the `SourceID` can be used to send additional messages back to the sender.

```
// Assume we have received a message for another entity.
void processMessage(Message incomingMsg) {

    // Send something back to the originator.
    Message newMsg = new MyMessage();
    newMsg.setDestinationID(incomingMsg.getSourceID());
    gateway.sendMessage(newMsg);
}
```

---

```
}
```

#### 4.1.10. getTraceID()

##### Signature

```
long getTraceID();  
void setTraceID(long newValue);
```

##### Usage

The TraceID is a unique number assigned by the messaging subsystem with the purpose of identifying and distinguishing each message. In an RPC scenarios, the CorrelationID refers to the TraceID of another message (and refers symbolically and numerically to the original request).

In general, every message has a different TraceID and in concept they are not reused (though in practice they may cycle after an enormous interval where all IDs have already been used once). Sometimes a new message can assume the TraceID of an existing message, and in that case the new message assumes the “identity” of the original message for correlation purposes. Although this is a valid technique, is used in various circumstances to achieve the desired behavior, and may be presented in this document where appropriate, care should be taken when using this approach.

## 4.2. Attribute-Related Operations

#### 4.2.1. attributeExists()

##### Signature:

```
boolean attributeExists();
```

##### Usage:

Determines whether the message has an attribute with the given name.

#### 4.2.2. validateProperties()

##### Signature:

```
String validateProperties(String conformanceDescriptor);
```

##### Usage:

This method determines whether the given set attributes are defined and typed as expected. This is a “quick and easy” way for a service to perform basic parameter validation on incoming messages. Input to the method consists of a string that lists the attributes, by name,

---

that are expected to exist as part of the message header. The list is delimited by commas (,) to allow multiple properties to be checked in one operation, and is never case-sensitive.

```
String missingProperties =  
    msg.validateProperties("value1,value2,value3");
```

The function returns a list of any attributes that are requested but are missing from the message, or are provided but do not match the expected type (if specified). If the result is NULL, then the message passed completely through validation without any mismatches detected.

Datatypes can also be validated by appending the type name to each value, separated by a colon (:) delimiter.

```
String missingOrMistypedProperties =  
    msg.validateProperties("value1:string,value2:integer,value3");
```

Datatypes that are arbitrary class types do not need to be fully qualified unless there is potential confusion between similarly named classes.

Some values may be optional and can be absent without representing an error condition, which are denoted by surrounding the attribute with [ and ] characters.

```
String missingProperties =  
    msg.validateProperties("[value1:string],value2:integer,value3");
```

These values will then pass validation even if they are missing, since this is not necessarily an error condition according to the API being validated. If they are present, though, their type is validated according to the specified datatype.

Finally, simple constant values can also be included as part of the validation for checking. The comparison is performed based on the toString() value of the given message attribute.

```
// Check for an expected value that is a constant.  
msg.validateProperties("value1:string=EXPECTEDVALUE");  
  
// Check for an expected value that may be one of a few values.  
// Each possible value is delimited by the pipe ("|") character.  
msg.validateProperties("value1:string=VALUE1|VALUE2|VALUE3");
```

Calls to this function can be combined to perform validation for single API calls that allow for different sets of parameters to be specified. The following example shows a more complete approach to validating an incoming request that includes reasonably detailed exceptions when there is a problem with the message format.

---

```

// Function to validate a given message and throw a
// MessageValidationException containing details if an error is
// detected in the message format.
void validateNotificationMessage(Message msg)
    throws MessageValidationException {
    // Validate the operation type.
    if (msg.validateProperties("optype:notifyoperator|notifycog")
        != null)
        throw new MessageValidationException(
            "'optype' is missing or invalid");

    // Validate parameters for specific sub-operations.
    String operationType = msg.getAttributeString("optype");
    String badAttributes;

    if (opType.equals("notifyoperator"))
        String badAttributes =
            msg.validateProperties(
                "cogid:integer,opname:string,text:string");

    else if (opType.equals("notifycog"))
        String badAttributes =
            msg.validateProperties(
                "cogid:integer,text:string,[priority:1|2|3]");

    else
        badAttributes = null;

    // Throw a fit if a problem is detected.
    if (MessageValidationException!= null)
        throw new MessageValidationException("Invalid"
            + " attributes specified"
            + " for '" + operationType
            + "' operation: "
            + badAttributes);
}

```

**Figure 6: Message validation within services**

Though the specific details are always somewhat unique to each message API, this approach can be very clean in general and also provide helpful feedback to the caller about the nature of any problems that arise.

#### 4.2.3. clearAttributes()

##### Signature:

```
void clearAttributes();
```

##### Usage:

Clears any attributes currently associated with the message.

---

#### 4.2.4. `getAttributeNames( )`

##### Signature:

```
Enumeration getAttributeNames();
```

##### Usage:

Returns an Enumeration of the names of all attributes currently defined for the message.

#### 4.2.5. `getAttributeBoolean( ) / setAttributeBoolean( )`

##### Signature:

```
boolean getAttributeBoolean(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *boolean* type. If the attribute is not a native boolean value, its String representation is converted into an appropriate boolean value based on the rules of behavior established by the Java Boolean class. The result will be TRUE if the attribute exists and is equal, ignoring case, to the string "true". In any other case, the result is FALSE. If the caller must distinguish between the absence of the attribute and an existing value of FALSE, this method should be used in conjunction with the *attributeExists( )* method.

#### 4.2.6. `getAttributeByte( ) / setAttributeByte( )`

##### Signature:

```
byte getAttributeByte(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *byte* type. The method throws a *NumberFormatException* if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.7. `getAttributeDouble( ) / setAttributeDouble( )`

##### Signature:

```
double getAttributeDouble(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *double* type. . The method throws a *NumberFormatException* if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.



---

#### 4.2.8. `getAttributeFloat()` / `setAttributeFloat()`

##### Signature

```
float getAttributeFloat(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *float* type. . The method throws a `NumberFormatException` if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.9. `getAttributeInt()` / `setAttributeInt()`

##### Signature:

```
int getAttributeInt(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *int* type. . The method throws a `NumberFormatException` if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.10. `getAttributeLong()` / `setAttributeLong()`

##### Signature:

```
Long getAttributeLong(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *long* type. . The method throws a `NumberFormatException` if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.11. `getAttributeObject()` / `setAttributeObject()`

##### Signature:

```
Object getAttributeLong(Object attributeName);
```

##### Usage:

Returns the given attribute as a generic object instance. For Java primitive types (such as *int* and *boolean*) the result will be expressed as the corresponding object type (e.g. Integer and Boolean). Other arbitrary object types are returned unchanged.

---

#### 4.2.12. `getAttributeShort()` / `setAttributeShort()`

##### Signature:

```
short getAttributeShort(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *short* type. . The method throws a `NumberFormatException` if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.13. `getAttributeString()` / `setAttributeString()`

##### Signature:

```
String getAttributeString(String attributeName);
```

##### Usage:

Returns an attribute coerced to the *String* type. . The method throws a `NumberFormatException` if the attribute does not exist or exists as a different datatype and cannot be coerced into the expected type.

#### 4.2.14. `clearContent()`

##### Signature:

```
void clearContent();
```

##### Usage:

Clears any Content currently associated with the message.

#### 4.2.15. `getContent()` / `setContent()`

##### Signature:

```
Object getContent ();  
void setContent (Object newValue);
```

##### Usage:

Returns any Content associated with the message. Content is raw data (carried in the form of an object instance) associated with the message. Its precise format and meaning are service-specific and can include of any object type that is `Serializable`.

Like attributes, the Content field can be used for message routing operations as described by various deployment descriptors. In case where Content is reference in a routing descriptor,

comparisons are performed on the `getContent().toString()` value of the field since the messaging subsystem cannot interpret arbitrary (e.g. service-specific) types during routing.

### 4.3. Debugging Facilities

As messages travel from point to point, and pass notable internal waypoints along the way, history records are collected in order to provide debugging information at the individual message level. Message histories are maintained even across machines, VMs, and network traversals. These records can be displayed in logs, inspected in a debug environment, or viewed in real time through the ServerDiag tool.

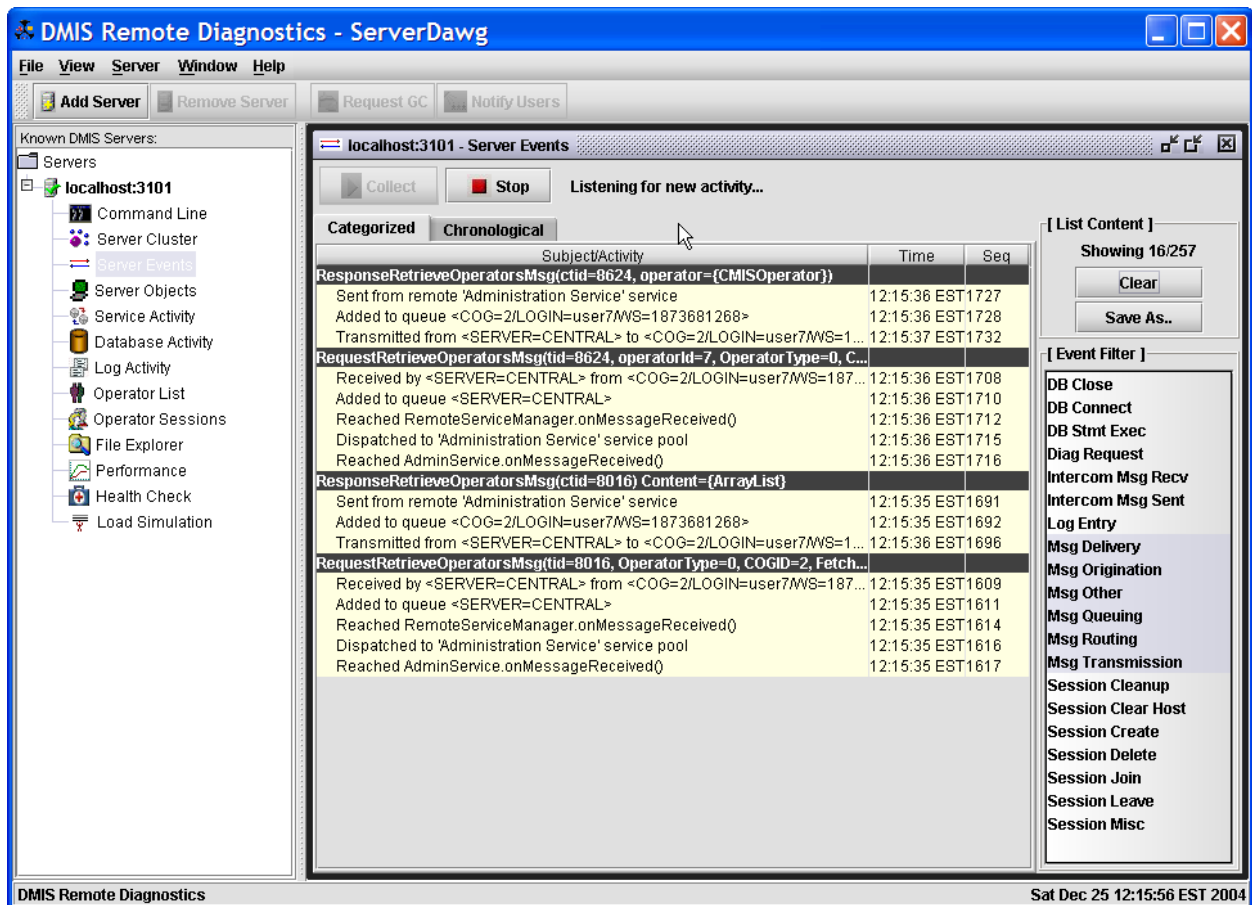


Figure 9: Message-level activity shown in ServerDiag

Applications and service code can obtain and augment this information dynamically. Custom-defined waypoints will automatically show up in the ServerDiag tool for easier (and context-specific) tracking of individual messages. Note that ServerDiag normally connects to a specific AppServer and therefore shows the message activity only as seen by that server. Entries that are recorded after the message leaves the custody of that AppServer will not show up, but may show up if ServerDiag is then focused on the next AppServer in the series. The Message activity log can be used to determine where a message has been routed next.

---

This section explains some of the programmatic tools related to this debugging feature (and others).

#### 4.3.1. `clearActivity()`

**Signature:**

```
void clearActivity();
```

**Usage:**

Clears any history records currently associated with the message. Any subsequent entries will be added to this fresh history and previous records will be lost.

#### 4.3.2. `getActivityLog()`

**Signature:**

```
String getActivityLog();  
String getActivityLog(String commonPrefix);  
void dumpActivityLog(Logger log, Level lev, String commonPrefix);
```

**Usage:**

Returns the current contents of the activity log as a String. This can then be directed to a log or displayed on the console as needed.

```
// Print out history for this message.  
System.out.println( msg.dumpHistory() );  
  
... the following information appears on the console ...  
  
t: instantiated at  
org.dmis.client.common.CMISClientGateway:internalConnect  
t+313ms: Reached DMISServicesGateway.sendMessage()  
t+313ms: Send route: null  
t+313ms: Added to queue <SERVER=CENTRAL>  
t+516ms: Pulled from queue <SERVER=CENTRAL>  
t+547ms: Transmitted from <COG=2/LOGIN=user7/WS=1873681268> to  
<SERVER=CENTRAL>  
t+641ms: Received by <SERVER=CENTRAL> from  
<COG=2/LOGIN=user7/WS=1873681268>  
t+641ms: Added to queue <SERVER=CENTRAL>  
t+641ms: Pulled from queue <SERVER=CENTRAL>  
t+657ms: Reached RemoteServiceManager.onMessageReceived()  
t+657ms: Dispatched to 'Authentication Service' service pool  
t+657ms: Reached AuthenticationService.onMessageReceived()
```

---

Note that each entry has a corresponding timestamp that is expressed in milliseconds since the message instance was created. When a message is cloned, the history is duplicated also and an entry is added to signify that the new instance is a copy of an original.

#### 4.3.3. `recordActivity()`

**Signature:**

Blah!

**Usage:**

Adds an entry to the message to the history log, time stamping it and ordering it appropriately with respect to the existing entries.

#### 4.3.4. `getDisplayProfile()`

**Signature:**

Blah!

**Usage:**

This method returns a display-friendly String that describes the given message. This is used throughout the middleware for logging activity on specific messages as they cross various waypoints.

```
// Display basic information for this message.
msg.getDisplayProfile();

... the following information appears on the console ...

RequestAuthenticationMsg(tid=3312, cog=2, operatorname="user7",
password=???, AUTH_LOCAL_ONLY=false)
```

The `toString()` implementation of the `Message` class uses `getDisplayProfile()` internally, meaning that it is the default String representation for any given message unless overridden.

#### 4.3.5. `getVisitList()`

**Signature:**

Blah!

**Usage:**

This method returns a list of all entities that have been visited by this message. The results are returned as a list of `RoutingIDs` in chronological order of each visit.

---

## 4.4. Housekeeping Operations

### 4.4.1. clone() / clone( keepExistingTraceID )

#### Signature:

Blah!

#### Usage:

This function creates a clone of the given message. The clone is a "shallow copy", meaning that embedded objects such as Content are not duplicated and will refer to the same instances (and thus modification on contained objects will affect both message).

The *keepExistingTraceID* parameter allows the clone to optionally have the same TraceID as the original message, meaning that correlations (via the CorrelationID) will function the same for the clone as for the original. This is useful in some situations where a message must be altered slightly prior to be forwarded to another entity, but cloned message is essentially the same. In RPC scenarios, the cloned message can be processed as a request and result can act as the response to the original message.

e HTML form.

### 4.4.2. HTTP Server Module

The HTTP Server module is a very crude web server that is a byproduct of RPC research performed while developing the AppServer architecture. It is included since it is functional in spite of being quite limited. In this incarnation the web server handles GET requests, passes them to various others AppServer modules for interpretation, or provides a default file serving behavior is necessary. The appropriate configuration entry to enable this module is:

```
<module name="HTTP Server"
      class="org.dmis.appserver.module.http.HTTPServerModule">
  <parm name="port" value="port" />
  <parm name="root" value="rootpath" />
</module>
```

The *port* and *rootpath* parameters specify the HTTP port and path to the web root, respectively.

Heavy use of this prototype module is not expected, since it duplicates a functional subset of much more powerful and standardized implementations such as Tomcat.

### 4.4.3. Hang Detection Module

The Hang Detection module constantly reviews various critical processes running on the AppServer and reports any that are not responding as expected. The appropriate configuration entry is:

```
<module name="Hang Detection"
```

---

```
class="org.dmis.appserver.module.hangdetect.HangDetectionModule"/>
```

Log output that is produced resembles the following redacted sample:

```
[Thu 1/20/05 15:01:25 EST]-[SEVERE] : One or more processes appear to be hung:  
[Thu 1/20/05 15:01:25 EST]-[SEVERE] :     Multiplexed Message Distribution Thread  
[Thu 1/20/05 15:01:25 EST]-[SEVERE] :     BaseSocketIONode Inbound Data Detector  
[Thu 1/20/05 15:01:25 EST]-[SEVERE] :     Message I/O Poll Initiator
```

Expected and actual ping times are also provided as a guide.

---

## 5. Service Interfaces

### 5.1. IBackgroundService Interface

The IBackgroundService interface is implemented by all *background services* that are run within the Service Manager module. Background services are started with the AppServer along with other services, but are not message-oriented in nature.

```
start(svcHelper)
stop()
```

#### 5.1.1. start ()

**Signature:**

```
void start (IServiceHelper sh) throws Exception;
```

**Usage:**

Sends a *start signal* to the given background service, which the typical service uses to allocate any resources require by other modules that will be running in the same container. An instance of `IServiceHelper` is passed in order to provide access to the full facilities of the AppServer.

#### 5.1.2. stop ()

**Signature:**

```
void stop () throws Exception;
```

**Usage:**

Sends a *stop signal* to the given background service, which the typical service uses to release any resources previously acquired by its corresponding `start()` method.

### 5.2. IMessageService Interface

The IMessageService interface is implemented by all *message services* that are run within the Service Manager module. Message services are started with the AppServer along with other services, following the initialization of any declared background services.



---

```
STATUS_OK = 1
STATUS_ERROR = 2

start(svcHelper)
stop()
getStatus()

onMessageReceived(msg)
```

### 5.2.1. Start()

#### Signature:

```
void start(IServiceHelper sh) throws Exception;
```

#### Usage:

`Start()` is called automatically by the Service Manager when the service is first started and needs to become operational. The service may take this opportunity to allocate common resources that it will need during the processing of its messages. `Start()` is normally called only once during the lifespan of a service, and always coupled with a corresponding call to `Stop()` by the ServiceManager. Services should be coded, however, with the capability to be started and stopped multiple times.

An instance of `IServiceHelper` is passed in order to provide access to the full facilities of the AppServer. If the Service starts as expected, it may begin receiving messages via the `onMessageReceived()` method.

### 5.2.2. Stop()

#### Signature:

```
void stop() throws Exception;
```

#### Usage:

`Stop()` is called by the ServiceManager when the service is to be shut down. The service may take this opportunity to release resources that it used while processing messages. Once this method has been called, the Service Manager will no longer send incoming messages to this service instance (it will no longer receive new messages through the `onMessageReceived()` method). `Stop()` is always coupled with a previous call to `Start()`.

### 5.2.3. getStatus()

---

**Signature:**

```
int getStatus();
```

**Usage:**

Returns the current status of the Service, using one of the established STATUS\_XXX constants defined in the DMISService interface. The Service Manager uses this status information for reporting and messaging dispatching (e.g. avoiding server instances that are in an error state). Many services return STATUS\_OK all the time, since they are generally never in an unqualified error state, but this may not always be the case.

**5.2.4. onMessageReceived( )****Signature:**

```
void onMessageReceived(Message msg) throws Exception;
```

**Usage:**

Invoked automatically by ServiceManager when a message is received from another entity (such as the DMIS Desktop) that should be processed by this AppServer. By the time this method is invoked, the message has already been determined to be relevant to this service. This determination is made according to the contents on the \*.svc descriptor file associated with the service.

If the service encounters uncorrectable problems while processing the message, it may simply throw an exception and the Service Manager will respond by automatically wrapping the exception in a TransportExceptionMsg and returning it to the original sender.

*If there is any type of exception, the handler is free to simply throw the appropriate type of Exception.* The larger framework will automatically catch any stray exceptions and wrap them in a MessagingExceptionMsg for delivery back to the client. The MessagingExceptionMsg that is produced is automatically correlated to the original request, so an application on the client-side waiting for a reply will receive the MessagingExceptionMsg as the reply.

The predefined *MessageValidationException* class, for example, can be thrown when a message is received with one or more expected elements missing or incorrect.

**5.3. IServiceHelper Interface**

The IServiceHelper interface allows DMIS services to access AppServer functionality. An instance of IServiceHelper is provided to each service upon startup, from which services can be leveraged.

---

|  |   |
|--|---|
| getCoreContext();  | <b>Topology Discovery</b>   |
| <b>Message Handling</b>  | getRoutingIDForSelf()   |
| sendMessage(msg)   | getRoutingIDForParent()   |
| sendMessage(msg,<br>distributionList)                            | hasParent(connectedParentOnly)  |
| sendMessage(msg,<br>distributionList,<br>ccList)                 | isEmbeddedServer()  |
|  | isIntermediateServer()  |
|  | isCentralServer()   |
| sendMessageToServersBelow(msg,<br>clientRoutingId)               | getServersBelow(clientRoutingId)  |
| sendMessageToServersAbove(msg)                                   | getServersAbove()   |
|  | getPeerRoutingIDs()   |
| sendMessageAndWait(msg)  | getTiersToCentral(includeSelf,<br>includeCentral,<br>includeServers,<br>includeWorkstations)                        |
| sendMessageAndWait(msg, timeout)                                 |   |
| forwardMessage(msg, newDestinationID)                            |   |
| forwardMessage(msg,<br>newDestinationID,<br>keepExistingTraceID) | getTiersToHere(sourceRoutingID,<br>includeSelf,<br>includeStartingPoint,<br>includeServers,<br>includeWorkstations) |
| forwardMessageToParent(msg,<br>onlyIfConnected)                  |   |
| notifyPeers(msg)   | getPhysicalName(serverRoutingId)  |
| <b>Connection Detection</b>                                      | <b>Shared Memory Access</b>   |
| getOnlineOperators()   | getSharedMemoryHelper()   |
| getKnownOperators()  |   |
| isConnected(routingID)   |   |
| getUserContext(routingID)  |   |
| <b>RoutingID Interpretation</b>                                  |   |
| refersToEmbeddedServer(routingID)                                |   |
| refersToIntermediateServer(routingID)                            |   |
| refersToCentralServer(routingID)                                 |   |
| refersToAnyServer(routingID)                                     |   |
| refersToThisServer(routingID)                                    |   |

### 5.3.1. getCoreContext( )

#### Signature:

```
CoreContext getCoreContext();
```

#### Usage:

Returns the **CoreContext** instance for the service, which provides access to primitive functions such as database connections and logging. The **CoreContext** interface is described in more detail in this document.

### 5.3.2. sendMessage( )

#### Signature:

---

```
void sendMessage(Message msg) throws MessagingException;

void sendMessage(Message msg,
                 List distributionList) throws MessagingException;

void sendMessage(Message msg,
                 List distributionList,
                 List ccList) throws MessagingException;
```

### Usage:

Sends the given message to an entity identified by its DestinationID, which is expressed in the form of a RoutingID. For DMIS, this method group is traditionally used to send a response to a client that has issued a request of some type (in RPC style):

```
void processRequest(MyRequestMsg requestMsg) {
    // Perform the requested action.
    ... process request ...

    // Formulate a response to the request.
    CorrelatedResponseMsg responseMsg =
        new CorrelatedResponseMsg(requestMsg);
    responseMsg.setContent(responseData);

    // Send out the response to the client. They are probably
    // waiting for it in synchronous fashion..
    svcHelper.sendMessage(responseMsg);
}
```

This version of the middleware offers two new variations of this method as well, designed to simplify sending the same message to multiple destinations in one operation. The first of these two accepts the Message to be send along with a distribution list of which entities should receive the message:

```
void processRequest(MyRequestMsg requestMsg) {
    // Perform the requested action.
    ... process request ...

    // Formulate a response to the request.
    CorrelatedResponseMsg responseMsg =
        new CorrelatedResponseMsg(requestMsg);
    responseMsg.setContent(responseData);

    // Send out the response to the client. They are probably
    // waiting for it in synchronous fashion..
    svcHelper.sendMessage(responseMsg);
}
```

If the distribution list is not NULL or empty, the DestinationID of the Message is ignored and addressing is performed based solely upon the contents of the distribution list. If the

---

distribution list is NULL or empty, the method functions the same as the base `sendMessage()` function.

The third form is the most generic and takes three arguments: the Message to be sent, a distribution list, and a CC list. The distribution list may be NULL or empty, in which case the DestinationID from the message instance is used. The CC list causes Carbon Copies of the message to be sent to the given entities in addition to the primary copies listed in the distribution list. Carbon Copies are messages with an `isCarbonCopy()` value of TRUE, and are typically used to notify other entities of an operation even when they need not take immediate action. This can be useful for synchronization between servers in a multi-tier hierarchy.

```
void processRequest(MyRequestMsg requestMsg) {
    // Perform the requested action.
    ... process request ...

    // Formulate a response to the request.
    CorrelatedResponseMsg responseMsg =
        new CorrelatedResponseMsg(requestMsg);
    responseMsg.setContent(responseData);

    // CC the response to any AppServers below us in the hierarchy
    // so they have the opportunity to update their own internal
    // states as appropriate.
    List ccList = svcHelper.getServersBelow();

    // Send out the response to the client and CC the appropriate
    // servers.
    svcHelper.sendMessage(responseMsg, null, ccList);
}
```

The Carbon Copy attribute of the Message class is discussed in more detail elsewhere in this document.

### 5.3.3. `sendMessageToServersBelow()`

#### Signature:

```
void sendMessageToServersBelow(Message msg, RoutingID clientRoutingId)
    throws MessagingException;
```

#### Usage:

Sends a Message to all AppServers in the hierarchy that exist along the route from the given client to here. In essence, all AppServers between there are here. The following code:

```
...
```

---

```
svcHelper.sendMessageToServersBelow(msg,  
                                     requestMsg.getSourceID());  
...
```

is equivalent to:

```
...  
svcHelper.sendMessage(msg,  
                      getServersBelow( requestMsg.getSourceID() ));  
...
```

In both instances, the message is sent to all servers between the originating client and this AppServer (not inclusive).

#### 5.3.4. sendMessageToServersAbove()

##### Signature:

void sendMessageToServersAbove(Message msg) throws MessagingException;

##### Usage:

Sends a Message to all AppServers in the hierarchy that exist along the route *from the current server* to (and including) the CENTRAL server. The following code:

```
...  
svcHelper.sendMessageToServersAbove(msg);  
...
```

is equivalent to:

```
...  
svcHelper.sendMessage(msg, getServersAbove());  
...
```

In both instances, the message is sent to all servers from *here* to CENTRAL.

#### 5.3.5. sendMessageAndWait()

##### Signature:

CorrelatedResponseMsg sendMessageAndWait(CorrelatedRequestMsg msg)  
 throws MessagingException;

CorrelatedResponseMsg sendMessageAndWait(CorrelatedRequestMsg msg,  
 long timeoutInMS)  
 throws MessagingException;

---

## Usage:

The `sendAndWait` method allows a service to send a `CorrelatedRequestMsg` and wait synchronously for a response. Prior to the version of the core framework, this functionality was available to clients but not to DMIS services. Now it is available everywhere.

```
// Send a request to the parent AppServer and wait for the result.
if (svcHelper.hasParent() &&
    svcHelper.isConnected( getRoutingIDForParent() )) {
    requestMsg.setDestinationID( svcHelper.getRoutingIDForParent() );
    Message responseMsg = svcHelper.sendMessageAndWait(requestMsg);

    // Do something useful with the response data.
    ...
}
```

Service designers must exercise caution in leveraging this feature from the server side. While the service is waiting for a response from another entity, it is tied up and cannot process any other incoming requests. This can drastically impact performance and scalability in some situations. One workaround is to use a large *poolsize* for the service to increase the likelihood that at least one instance will always be available for new operations. Designers must also be careful to avoid logic using `sendMessageAndWait( )` that contains circular interdependencies (where two services wait specifically for one another), as those can result in deadlocked services across machines.

### 5.3.6. `forwardMessage( )`

#### Signature:

```
void forwardMessage(Message msg,
                    RoutingID newDestinationID)
    throws MessagingException;

void forwardMessage(Message msg,
                    RoutingID newDestinationID,
                    boolean bkeepExistingTraceID)
    throws MessagingException;

boolean forwardMessageToParent(Message tm,
                               boolean onlyIfConnected)
    throws MessagingException;

Exception;
```

#### Usage:

The `forwardMessage( )` series are shorthand methods to simplify the common task of cloning and forwarding a request to a parent server. The first two forms, which takes a reference to a `Message` and `DestinationID` of the new target, simply perform a `clone( )` on the message instance and then send the clone.

---

```

void processRequest(MyRequestMsg requestMsg) {
    // Forward to the parent server if there is one available.
    if (svcHelper.hasParent() &&
        svcHelper.isConnected( getRoutingIDForParent() )) {
        forwardMessage( requestMsg, getRoutingIDForParent() )
    }
    else {
        ... process the message here (at this level) ...
    }
}

```

The third format is similar but is designed specifically for forwarding messages to a parent server. It determines, in one operation, whether there is an available parent server. If so the message is cloned and forwarded there and returns TRUE. If there is no available parent, the method returns FALSE. This can help clarify service code in some cases.

```

void processRequest(MyRequestMsg requestMsg) {
    // Forward to the parent server if there is one available.
    // Or process it here if we are the highest available server.
    if (!forwardMessageToParent( requestMsg ))
        ... process the message here (at this level) ...
    }
}

```

**Important Note:** Embedding forwarding logic directly in services is supported, but is no longer recommended. Message flow should now be modeled using \*.wfd (Workflow Descriptor) files to describe the way a given message should be routed. This allows services to operate in a more flexible manner while at the same simplifying the service implementation by removing extra conditions.

### 5.3.7. notifyPeers( )

#### Signature:

void notifyPeers(Message msg) throws MessagingException;

#### Usage:

The notifyPeers() allows services to easily leverage the server intercom by sending high-priority messages to all peers in the cluster to which this server belongs. The content and routing of these messages follows the same rules defined for the rest of DMIS, except that these messages receive a HIGH priority. This may cause them to be transmitted and serviced prior to messages with lower priorities.

### 5.3.8. getOnlineOperators( )

#### Signature:



---

```
List getOnlineOperators();
```

**Usage:**

Returns a list of all operators currently connected to the same DMIS network as the AppServer. The list includes only operators that are currently reachable (directly or indirectly) through the messaging subsystem. Other operators are considered offline until they become accessible. The result is returned as a List of RoutingIDs, each representing one operator logically, that can be fed directly into other `IServiceHelper` functions

### 5.3.9. `getKnownOperators()`

**Signature:**

```
List getKnownOperators();
```

**Usage:**

Returns a list of all operators that are known to be part of the DMIS network, including operators that are not currently connected.

### 5.3.10. `isConnected()`

**Signature:**

```
boolean isConnected(RoutingID routingID);
```

**Usage:**

Determines whether the given entity is currently online and accessible through the DMIS network. If so, the entity is likely ready to receive messages from the given service. The RoutingID can refer to any DMIS entity – an AppServer, operator, or another or class.

### 5.3.11. `getRoutingIDForSelf()`

**Signature:**

```
RoutingID getRoutingIDForSelf();
```

**Usage:**

Returns the fully-qualified RoutingID that represents this AppServer on the DMIS network.

---

### 5.3.12. getRoutingIDForParent( )

#### Signature:

```
RoutingID getRoutingIDForParent();
```

#### Usage:

Returns the fully-qualified RoutingID that represents the parent AppServer on the DMIS network. Returns NULL if this AppServer has no immediate parent (such as when a service is running on the CENTRAL server).

### 5.3.13. hasParent( )

#### Signature:

```
boolean hasParent(boolean connectedParentOnly);
```

#### Usage:

Returns TRUE if the current AppServer has a parent in the DMIS hierarchy, and optionally whether it is currently reachable.

### 5.3.14. getServersAbove( )

#### Signature:

```
List getServersAbove();
```

#### Usage:

Returns a list of AppServers above this one in the DMIS hierarchy. This method is a shorthand for the more general `getRouteToCentral( )` method of `IServiceHelper`.

**Called from an EMBEDDED context returns these RoutingIDs:**  
LAN server *(if one is defined and reachable)*  
CENTRAL server

**Called from a LAN context returns these RoutingIDs:**  
CENTRAL server

**Called from CENTRAL context returns these RoutingIDs:**  
*(empty List)*

The result is returned as a list of RoutingIDs, suitable for use in other `IServiceHelper` functions.

---

### 5.3.15. getServersBelow()

#### Signature:

```
List getServersBelow(RoutingID clientRoutingId);
```

#### Usage:

Returns a list of AppServers above this one in the DMIS hierarchy. This method is a shorthand for the more general `getRouteToHere()` method of `IServiceHelper`.

**Called from the CENTRAL context returns these RoutingIDs:**

LAN server *(if one is defined and reachable)*  
EMBEDDED server *(when used)*

**Called from a LAN context returns these RoutingIDs:**

EMBEDDED server *(when used)*

**Called from an EMBEDDED context returns these RoutingIDs:**

*(empty List)*

The result is returned as a list of RoutingIDs, suitable for use in other `IServiceHelper` functions.

### 5.3.16. getPeerRoutingIDs()

#### Signature:

```
List getPeerRoutingIDs();
```

#### Usage:

Returns a list of RoutingIDs for all AppServers that are in the same logical cluster as the current server.

### 5.3.17. getTiersToCentral()

#### Signature:

```
List getTierstoCentral(boolean includeSelf,  
                       boolean includeCentral,  
                       boolean includeServers,  
                       boolean includeworkstations);
```

#### Usage:

---

Computes the set of logical tiers between this AppServer, regardless of where it is in the DMIS hierarchy, and the CENTRAL server. The method includes parameters to filter portions of the list in order to simplify specific algorithm-level requirements. The result is returned as a list of RoutingIDs, suitable for use in other `IServiceHelper` functions.

#### 5.3.18. `getTiersToHere()`

##### **Signature:**

```
List getTiersToHere(RoutingID sourceRoutingID,  
                    boolean includeSelf,  
                    boolean includeStartingPoint,  
                    boolean includeServers,  
                    boolean includeworkstations);
```

##### **Usage:**

Computes the set of logical tiers between the given starting point, generally representing a low-level entity such as an operator or workstation, and the CENTRAL AppServer. The method includes parameters to filter portions of the list in order to simplify specific algorithm-level requirements. The result is returned as a list of RoutingIDs, suitable for use in other `IServiceHelper` functions.

#### 5.3.19. `isEmbeddedServer()`

##### **Signature:**

```
boolean isEmbeddedServer();
```

##### **Usage:**

Returns TRUE if the service is running in the context of an EMBEDDED server. An EMBEDDED server is running at the workstation level in order to provide offline support.

#### 5.3.20. `isIntermediateServer()`

##### **Signature:**

```
boolean isIntermediateServer();
```

##### **Usage:**

Returns TRUE if the service is running in the context of an INTERMEDIATE server. In DMIS, an INTERMEDIATE server is an AppServer running on a LAN and managed by the local organization.

---

#### 5.3.21. isCentralServer( )

##### Signature:

```
boolean isCentralServer();
```

##### Usage:

Returns TRUE if the service is running in the context of the CENTRAL server. The CENTRAL server bank is responsible for the broadest array of operators and services of any AppServer in the DMIS network, and is generally the “last stop” for any RPC-style requests.

#### 5.3.22. refersToEmbeddedServer( )

##### Signature:

```
boolean refersToEmbeddedServer(RoutingID routingID);
```

##### Usage:

Returns TRUE if the given RoutingID refers to any EMBEDDED server. An EMBEDDED server is running at the workstation level in order to provide offline support. The service can discover more details regarding the server’s identity by examining various dimensions of the RoutingID.

#### 5.3.23. refersToIntermediateServer( )

##### Signature:

```
boolean refersToIntermediateServer(RoutingID routingID);
```

##### Usage:

Returns TRUE if the RoutingID refers to any INTERMEDIATE server. In DMIS, an INTERMEDIATE server is an AppServer running on a LAN and managed by the local organization. The service can discover more details regarding the server’s identity by examining various dimensions of the RoutingID.

#### 5.3.24. refersToCentralServer( )

##### Signature:

```
boolean refersToCentralServer(RoutingID routingID);
```

##### Usage:

---

Returns TRUE if the RoutingID refers to the CENTRAL server. The CENTRAL server bank is responsible for the broadest array of operators and services of any AppServer in the DMIS network, and is generally the “last stop” for any RPC-style requests.

#### 5.3.25. refersToAnyServer( )

**Signature:**

```
boolean refersToAnyServer(RoutingID routingID);
```

**Usage:**

Returns TRUE if the given RoutingID refers to any form of AppServer. Any other type of entity, such as an operator or workstation, will return FALSE.

#### 5.3.26. refersToThisServer( )

**Signature:**

```
boolean refersToThisServer(RoutingID routingID);
```

**Usage:**

Returns TRUE if the given RoutingID refers to the AppServer on which the current service is running.

#### 5.3.27. getPhysicalName( )

**Signature:**

```
String getPhysicalNames(RoutingID serverRoutingId);
```

**Usage:**

Given a RoutingID for an AppServer, this method attempts to return a host name that can be used to contact the server installation via TCP/IP. This allows services to interact with non-DMIS services that are installed in parallel on a DMIS server, while retaining some of the dynamic routing afforded by the messaging subsystem. The RoutingID may be fully qualified or not; if not, the closest server that offers the given service will be provided. NULL is returned if no such AppServer exists, is unavailable, or has no known public address.

#### 5.3.28. getUserContext( )

**Signature:**

---

`IUserContext getUserContext(RoutingID strRoutingID)` throws `Exception`;

**Usage:**

Given the `RoutingID` of an operator or any other entity, returns an `IUserContext` node that can be used to store “session-level” values. These values are then available on that `AppServer` across requests from that client, and can be cached for efficiency (must like a “shopping cart” implementation on the web).

Since the `IUserContext` instance may be discarded at any point due to `AppServer` conditions, a service that uses it must be coded to handle missing values in any field it tracks there. Also, an `IUserContext` instance is not shared across members of a cluster, meaning that if multiple servers receive requests from the client, each server will have its own `IUserContext` for that client by default.

### 5.3.29. `getSharedMemoryHelper()`

**Signature:**

`ISharedMemoryHelper getSharedMemoryHelper()` throws

**Usage:**

Returns the `ISharedMemory` interface for the service, which can be used to manage values *that are shared* with all members of an `AppServer` cluster. Different servers within the cluster can safely read and write values as though they are within the same VM. The `ISharedMemory` interface is described in more detail in this document.

## 5.4. `ISharedMemory` Interface

The `ISharedMemory` interface allows services running on an `AppServer` to share values with other servers running as members in the same cluster. Values can be read and modified, with automatic synchronization and protection from concurrent modification.

### Shared Element Allocation


```
getSharedMap(name)
getSharedList(name)
getSharedObject(name)
setSharedObject(name, newValue)

getSharedCriticalSection(name)
```

### Discovery Functions

```
sharedMapExists(String name)
sharedListExists(String name)
sharedObjectExists(String name)
```

---



Shared objects are referenced by name, allowing a service running on two or more machines to easily connect to the appropriate value. Share names are always relative to the given service, meaning that a Map named “test” by the TIE service will not collide with a Map named “test” in the IM service. This enables services that use these features to remain atomic without the concern of interference from other pieces of code.

#### 5.4.1. `getSharedMap()`

**Signature:**

```
sharedMap getSharedMap(String name);
```

**Usage:**

Allocates a SharedMap instance with the given symbolic name, or returns a previously-defined instance with the given name. The SharedMap implements the *java.util.Map* interface, and can thus be used anywhere the Map interface is accepted.

Assume that the following code is executing for the first time on two different AppServer within the same cluster. The following sample illustrates the functionality of the SharedMap:



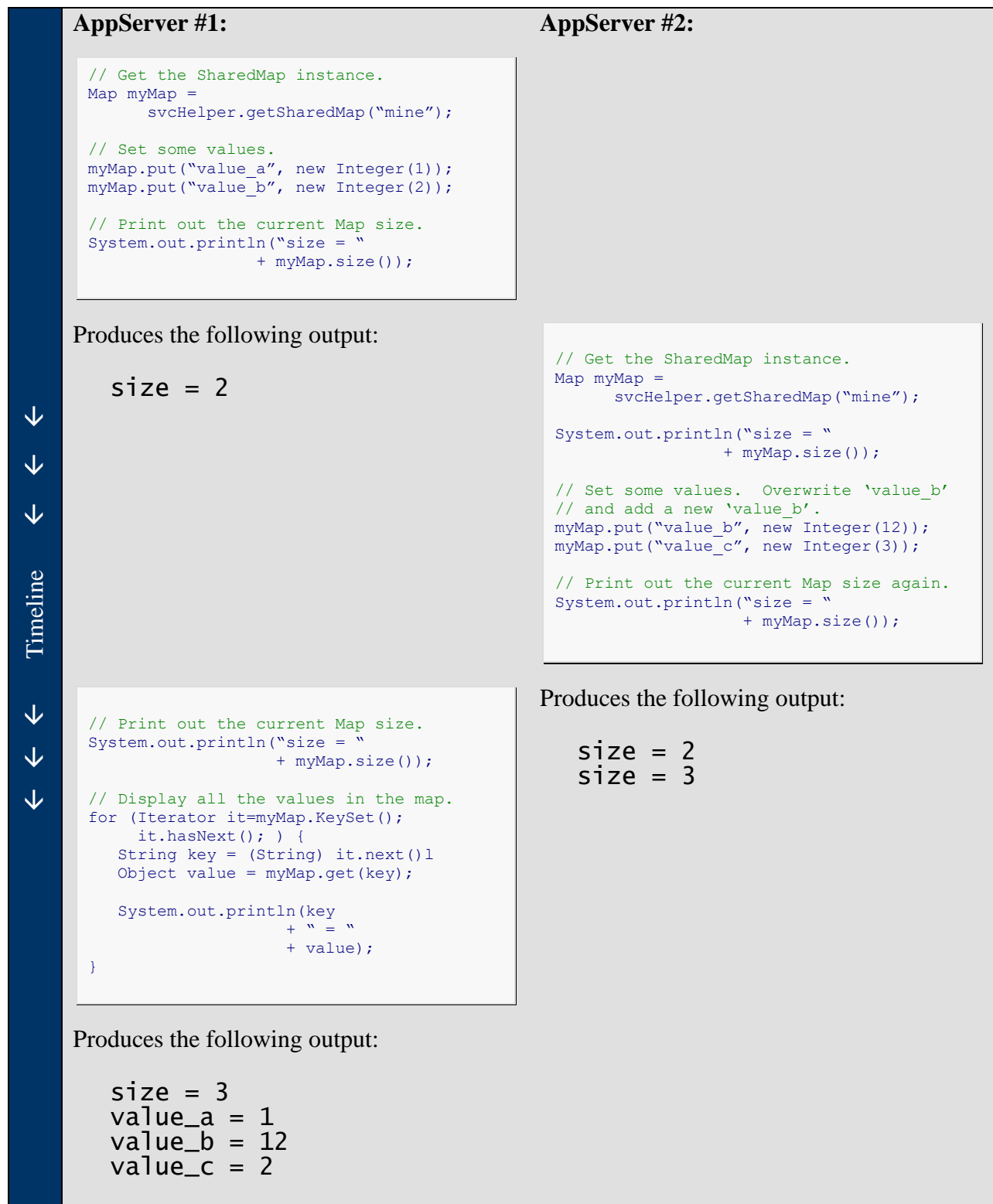


Figure 7: Sample SharedMap Code

*Note:* Although the SharedMap class fully implements the *java.util.Map* interface, the use of the Java *synchronized* keyword must be considered carefully. The *synchronized* keyword is

---

typically used when changes are made to a Map instance that require exclusive access between multiple primitive calls (e.g. `put()`, `remove()`, etc):

```
// Lock within the current VM.
synchronized(myMap) {
    // Do some complex stuff on the Map.
    ...
}
```

However, it only locks the Map instance within the same VM. In order to lock the Map for the entire cluster (including between threads in the current VM), use the `lock()` and `unlock()` methods of the `SharedMap` instance.

```
// Lock within the current VM and the entire cluster.
myMap.lock();

    // Do some complex stuff on the Map.
    ...

myMap.unlock();
```

These methods ensure that no other members of the cluster can change the Map's content concurrently.

#### 5.4.2. `getSharedList()`

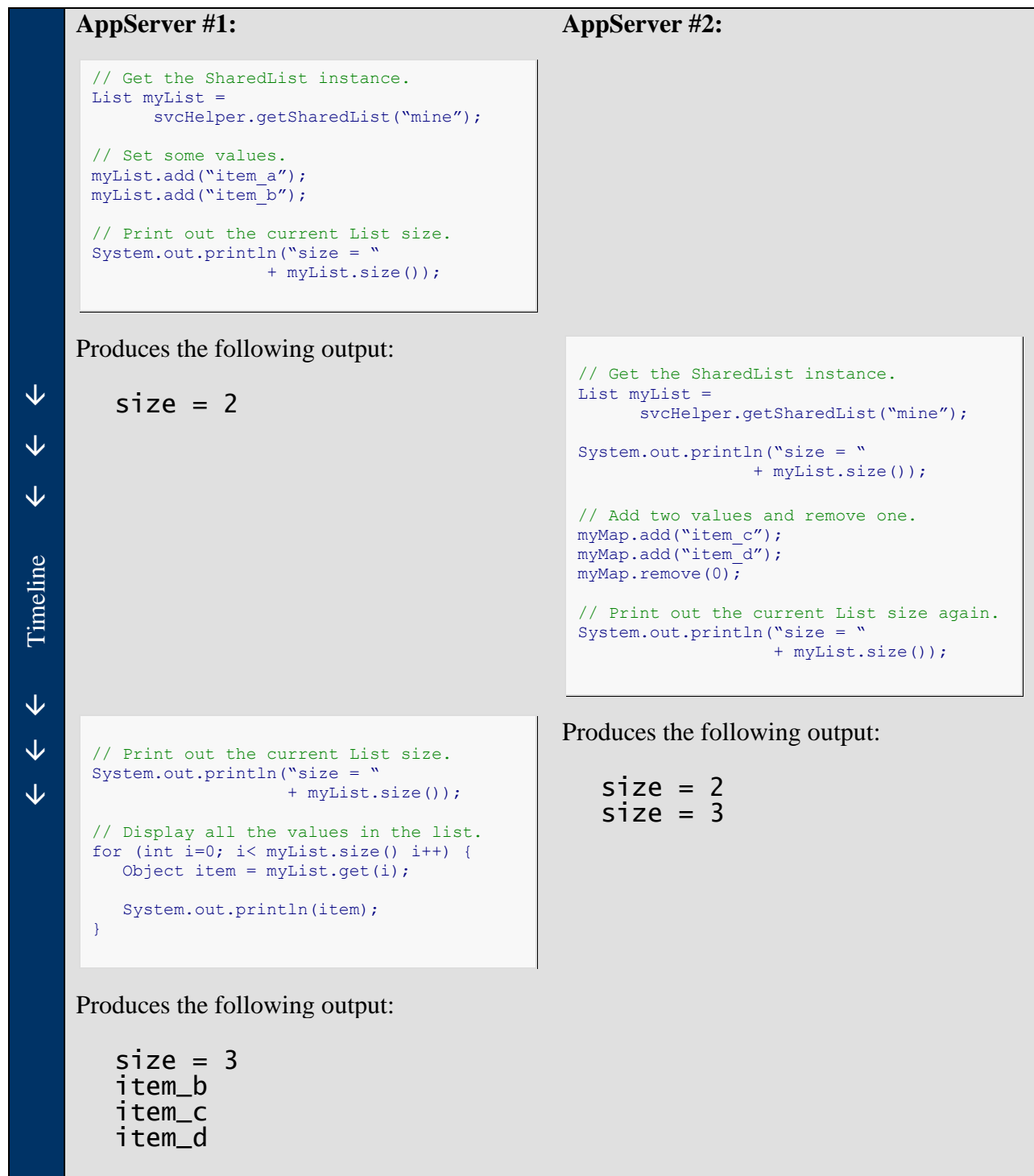
##### Signature:

```
SharedList getSharedList(String name);
```

##### Usage:

Allocates a `SharedList` instance with the given symbolic name, or returns a previously-defined instance with the given name. The `SharedList` implements the *java.util.List* interface, and can thus be used anywhere the `List` interface is accepted.

Assume that the following code is executing for the first time on two different `AppServer` within the same cluster. The following sample illustrates the functionality of the `SharedList`:



**Figure 8: Sample SharedList Code**

*Note:* Although the SharedList class fully implements the *java.util.List* interface, the use of the Java *synchronized* keyword must be considered carefully. The *synchronized* keyword is typically used when changes are made to a List instance that require exclusive access between multiple primitive calls (e.g. *add()*, *remove()*, etc):

---

```
// Lock within the current VM.
synchronized(myList) {
    // Do some complex stuff on the List.
    ...
}
```

However, it only locks the List instance within the same VM. In order to lock the List for the entire cluster (including between threads in the current VM), use the `lock()` and `unlock()` methods of the Shared List instance.

```
// Lock within the current VM and the entire cluster.
myList.lock();

    // Do some complex stuff on the List.
    ...

myList.unlock();
```

These methods ensure that no other members of the cluster can change the List content concurrently.

#### 5.4.3. `getSharedObject()` / `setSharedObject()`

##### Signature:

```
Object getSharedObject(String name);
void setSharedObject(String name, Object newValue);
```

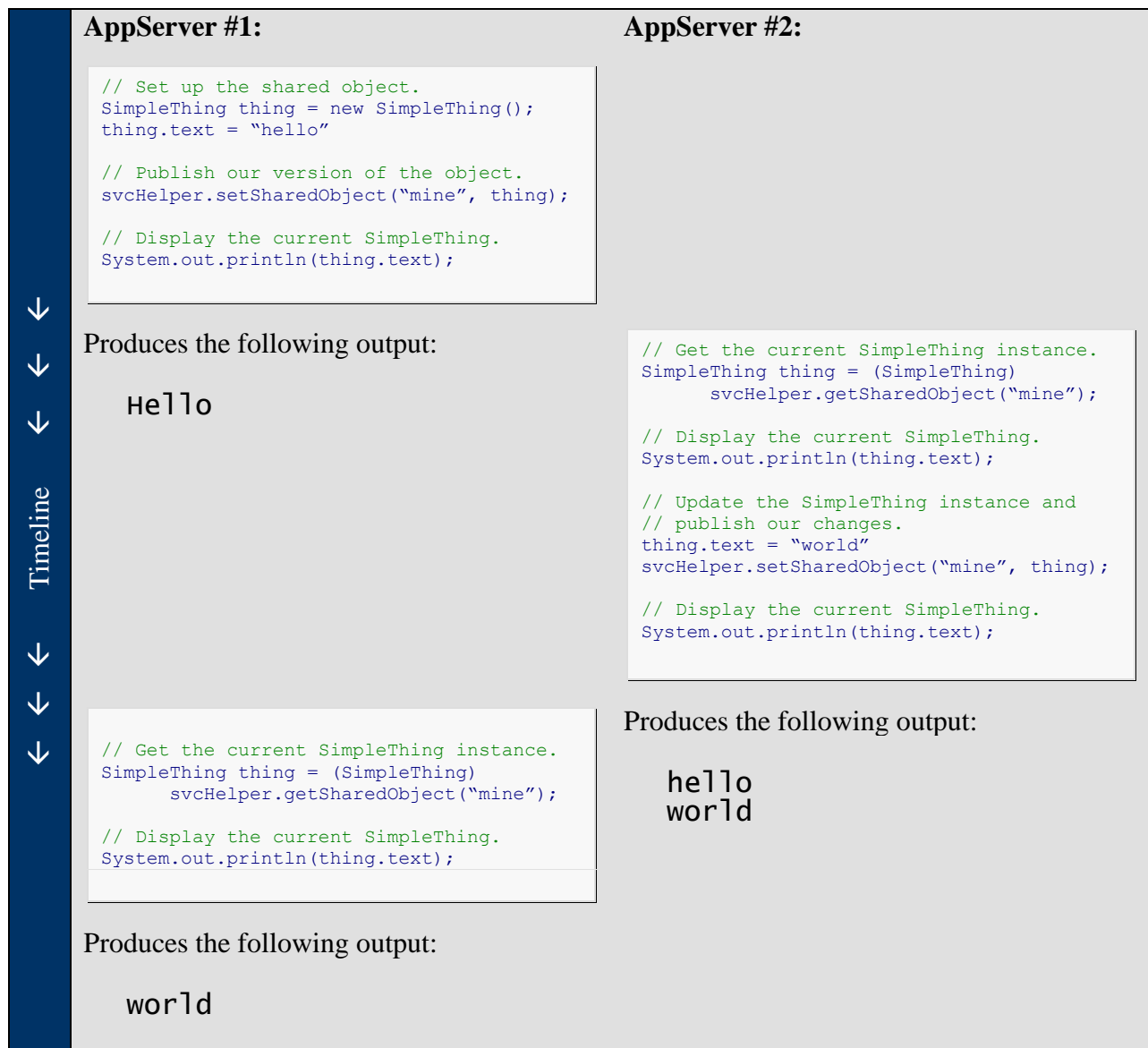
##### Usage:

Fetches a previously-defined object instance, with the given symbolic name, that can be shared among cluster members to enhance cluster-level interaction.

Assume that the following code is executing for the first time on two different AppServer within the same cluster. Assume also that a class definition exists with the following layout:

```
class SimpleThing {
    String text;
}
```

The following sample illustrates this object-sharing functionality:



**Figure 9: Sample SharedObject Code**

In order to receive the most recently “published” object instance, the service must invoke `getSharedObject()`. An instance previously retrieved via `getSharedObject()` will be updated in real time; the service must obtain a new snapshot if current states are required.

*Note:* Since shared objects are handled as the root Object type, it is the responsibility of the service to ensure that all service instances know how to correctly cast each shared instance.

#### 5.4.4. `getSharedCriticalSection()`

##### Signature:

```
sharedCriticalSection getSharedCriticalSection(String name);
```

##### Usage:

---

Retrieves a `SharedCriticalSection` instance that can be used to coordinate execution across members of a cluster. In the Java environment, blocks of code can be protected from concurrent execution by using the `synchronized` keyword:

```
// Ensure only one thread executes this at any given time.
synchronized(myObject) {
    // Do some critical stuff.
    ...
}
```

However, this only works within the same VM. The `SharedCriticalSection` extends this to the entire cluster to which the server belongs:

```
// Ensure only one entity executes this at any given time
// (anywhere within the VM or within the cluster).
SharedCriticalSection cs =
    svcHelper.getsharedCriticalSection("this.section");

cs.lock();
    // Do some critical stuff.
    ...
cs.unlock();
```

Critical sections are named using an arbitrary string that has meaning only to the service.

#### 5.4.5. `sharedMapExists()`

**Signature:**

```
boolean sharedMapExists(String name);
```

**Usage:**

Returns `TRUE` if the given `SharedMap` has been defined anywhere in the current cluster.

#### 5.4.6. `sharedListExists()`

**Signature:**

```
boolean sharedListExists(String name);
```

**Usage:**

Returns `TRUE` if the given `SharedList` has been defined anywhere in the current cluster.

---

#### 5.4.7. `sharedObjectExists()`

##### **Signature:**

```
boolean sharedObjectExists(String name);
```

##### **Usage:**

Returns TRUE if the given shared object instance has been defined anywhere in the current cluster.

---

This Page Intentionally Left Blank