



VistALink 1.5

Developer Guide

May 2006

Application Modernization Program
Health Systems Design & Development (HSD&D)
Department of Veterans Affairs

Revision History

Date	Version	Description	Contacts
5/10/06	1.5	Initial VistALink 1.5 release.	Jim Alexander, technical writer Dawn Clark, project manager

Revision History

Table of Contents

1. Introduction.....	1
1.1. About this Guide.....	1
1.2. Additional Resources.....	1
1.2.1. VistALink 1.5.....	1
1.2.2. BEA Systems.....	2
1.3. About J2EE Connectors.....	2
1.4. Public VistALink APIs Documentation.....	2
1.5. Sample Applications for J2EE Server.....	2
2. Developer Workstation Setup.....	3
2.1. J2EE Development.....	3
2.1.1. IDE.....	3
2.1.2. J2EE Runtime.....	3
2.2. J2SE Development.....	3
2.2.1. IDE.....	3
2.2.2. J2SE Runtime.....	3
3. Using VistALink in J2EE.....	5
3.1. Using Station Number (Institution) and Subdivision.....	5
3.1.1. System Locator: Institution-Connector Mapping.....	5
3.1.2. Multidivision-Aware Application Code: ConnectionSpec Credentials.....	5
3.1.3. Example.....	5
3.2. Request Cycle.....	6
3.2.1. Retrieving the Connection Factory.....	6
3.2.2. Instantiating a Connection Spec for Re-authentication.....	6
3.2.3. Getting a Connection (Connection Spec).....	7
3.2.4. Executing a Request.....	7
3.2.5. Closing the Connection.....	8
3.2.6. Connectivity Failures and Retry Strategies.....	9
3.3. More about Re-authentication.....	9
3.3.1. Overview.....	9
3.3.2. Connection Specification Classes.....	10
3.3.3. Institution/Division Rules for Re-authentication.....	11
3.3.4. Application Proxy User.....	11
3.3.4.1. J2EE Application Proxy Usage Example.....	13
3.4. Timeouts.....	13
3.4.1. Socket-Level Forced Timeout.....	13
3.4.1.1. Setting Socket-Level Timeouts.....	14
3.4.1.2. Default Socket-Level Timeout.....	14
3.4.1.3. Changing Socket Timeout as a Multiple of Default Timeout.....	14
3.4.2. Graceful (Request-Level) Timeout.....	14
3.4.2.1. STOP^XOBVLIB().....	15
3.4.2.2. \$\$GETTO^XOBVLIB().....	16
3.4.2.3. \$\$SETTO^XOBVLIB().....	16

3.4.2.4.	Java and M Code RPC Timeout Call Examples	16
3.5.	Institution Mapping.....	17
3.5.1.	How to Configure Mappings	17
3.5.2.	How to View the Currently Loaded Mappings.....	17
3.5.3.	Retrieving Mappings for Applications.....	17
3.5.4.	Subdivisions.....	18
3.6.	VistALink Java API Reference.....	18
4.	Executing Requests	19
4.1.	Remote Procedure Calls.....	19
4.1.1.	RPC Security (“B”-Type Option)	19
4.1.2.	RPCs for Use by Application Proxy Users	19
4.2.	Request Processing	19
4.2.1.	Get an RpcRequest Object: RpcRequestFactory Class.....	20
4.2.1.1.	getRpcRequest() Example	20
4.2.2.	Set RpcRequest Parameters: “Explicit” Style.....	21
4.2.2.1.	Literal RPC Parameter Example.....	21
4.2.2.2.	Reference RPC Parameter Example	21
4.2.2.3.	List RPC Parameter Example:	22
4.2.2.4.	Combination RPC Parameter Example:.....	22
4.2.3.	Set RpcRequest Parameters: “setParams” Style	22
4.2.3.1.	Literal RPC Parameter Example:.....	23
4.2.3.2.	Reference RPC Parameter Example:	23
4.2.3.3.	List RPC Parameter Example:	23
4.2.3.4.	Combination RPC Parameter Example:.....	23
4.2.4.	Specifying Indices for List-Type RPC Parameters.....	24
4.2.4.1.	List RPC Parameter Example (Explicit Index).....	24
4.2.4.2.	List RPC Parameter Example (Explicit Multi-Level Index).....	24
4.2.5.	Other Useful RpcRequest Methods	25
4.2.5.1.	Clear Previous Request Parameters	25
4.2.5.2.	Set the Message Format (Proprietary or XML)	25
4.2.5.3.	Set the RPC Context	25
4.2.5.4.	Set the RPC Name	25
4.2.5.5.	Set the RPC Client Timeout.....	26
4.3.	Response Processing.....	26
4.3.1.	RpcResponse Class	26
4.3.2.	Request/Response Example	26
4.3.3.	Parsing RPC Results	27
4.3.4.	XML Responses.....	28
4.4.	How to Write RPCs	28
4.4.1.	Write Stateless RPCs Whenever Possible.....	28
4.4.2.	When State is Needed	28
4.4.2.1.	Session ID as Temporary Storage Index.....	29
4.4.2.2.	FileMan-Based Lock File	29
4.4.3.	Pitfalls of Using of \$JOB in Stateful RPCs	29
4.4.4.	Pitfalls of Global Locking in Stateful RPCs.....	29

5.	VistALink Exception Reference	31
5.1.	Checked and Unchecked Exceptions	31
5.2.	Catching Exceptions	32
5.3.	VistALink Exception Hierarchy	33
5.4.	J2EE and J2SE Connectors Exceptions	34
5.4.1.	VistaLinkResourceException	34
5.4.2.	FoundationsException.....	34
5.4.3.	VistaLinkFaultException	35
5.4.4.	Common FoundationsExceptionInterface	35
5.4.5.	Exception Nesting.....	35
5.5.	Working with Nested Exceptions	36
5.5.1.	ExceptionUtils.....	37
5.5.2.	ExceptionUtils:: getFullStackTrace(Throwable e)	37
5.5.3.	ExceptionUtils:: getNestedExceptionByClass().....	37
6.	Foundations Library Utilities	39
6.1.	Encryption: gov.va.med.crypto.....	39
6.2.	J2EE Environment: gov.va.med.environment	39
6.2.1.	Environment.isProduction().....	39
6.2.2.	Environment.getServerType()	40
6.3.	Exception: gov.va.med.exception	40
6.4.	Audit Timer: gov.va.med.monitor.time	40
6.4.1.	Sample Code	40
6.5.	XML: gov.va.med.xml.....	41
6.5.1.	XmlUtilities Class.....	41
6.5.2.	XMLUtilities Example.....	42
6.6.	Network: gov.va.med.net.....	43
7.	Using VistALink with J2SE Applications.....	45
7.1.	Authenticating and Connecting to Vista in Client-Server Mode.....	45
7.1.1.	JAAS Overview	45
7.2.	VistALink JAAS Implementation.....	46
7.2.1.	VistaLoginModule	46
7.2.2.	JAAS Login Configuration Overview	46
7.2.3.	VistALink-Specific JAAS Login Configuration	46
7.2.4.	Passing the JAAS Login Configuration(s) to Your JVM	47
7.2.5.	Selecting the JAAS Configuration From an Application	48
7.2.6.	VistaLoginModule Callback Handlers	48
7.3.	Putting the Pieces Together: VistALink JAAS Login	48
7.3.1.	Logging in to Vista	48
7.4.	After Successfully Logging In.....	49
7.4.1.	Retrieving the VistaKernelPrincipal	49
7.4.2.	Retrieving the Authenticated Connection From the Principal.....	50
7.4.3.	Retrieving User Demographic Information	50
7.4.4.	Executing RPCs	51
7.4.5.	Logging Out.....	51
7.4.5.1.	Logging Out of Swing Applications.....	51

Contents

7.5. Catching Login Exceptions.....	52
7.5.1. VistaLoginModule Exception Hierarchy.....	52
7.6. Unit Testing and VistALink.....	54
Glossary	55

List of Figures

Figure 1. Java Base Exception Classes	31
Figure 2. VistALink Exception Hierarchy	34

List of Tables

Table 1. Connection Specification Classes	10
Table 2. Mapping RPC Return Types to VistALink Result String Format	27
Table 3. VistALink Utility Methods	42
Table 4. VistALink Utility Variables	42
Table 5. Demographics Keys and Values	51
Table 6. VistALink Login Exceptions	53

Contents

1. Introduction

1.1. About this Guide

This document is a guide to developing applications that utilize VistALink 1.5. The VistALink 1.5 resource adapter is a transport layer that provides communication between HealthVet Java applications and VistA/M servers, in both client-server and n-tier environments. It allows RPCs to execute on the VistA/M system and return results to the Java enterprise system.

VistALink consists of Java-side adapter libraries and an M-side listener. The adapter libraries use the J2EE Connector Architecture (J2CA 1.0) specification to integrate Java applications with legacy systems. The M listener process receives and processes requests from client applications.

The term *resource adapter* is often shortened in this guide to *adapter*, and is also used interchangeably with the term *connector*.

1.2. Additional Resources

The VistALink website (<http://vista.med.va.gov/migration/foundations/index.htm>) summarizes VistALink architecture and functionality and gives status updates for all VistALink products.

1.2.1. VistALink 1.5

When Enterprise VistA Support (EVS) releases VistALink 1.5, the documentation set for VistALink will be available from the **anonymous.software** directory at **download.vista.med.va.gov**. It will include the following:

- *VistALink 1.5 Installation Guide*: Provides detailed instructions for setting up, installing, and configuring the VistALink 1.5 listener on VistA/M servers and the VistALink resource adapter on J2EE application servers. Its intended audience includes server administrators, IRM IT specialists, and Java application developers.
- *VistALink 1.5 System Management Guide*: Contains detailed information on J2EE application server management, institution mapping, the VistALink console, M listener management, and VistALink security, logging, and troubleshooting.
- *VistALink 1.5 Developer Guide*: Contains detailed and background information about writing code utilizing VistALink.
- *VistALink 1.5 Release Notes*: A list of all the features included in each VistALink 1.5 release.
- *Getting Started With the BDk, Chapter 3: RPC Overview*. A short guide on writing RPCs from the *RPC Broker* manual.


1.2.2. BEA Systems

VistALink 1.5 has been tested and is supported on BEA WebLogic Server 8.1 (Service Pack 4) only. WebLogic product documentation can be found at the following website:

<http://edocs.bea.com/>.

1.3. About J2EE Connectors

VistALink is a resource adapter that implements and is fully compliant with the J2EE Connector Architecture Specification 1.0. VistALink is accessed programmatically through the interfaces specified in the J2EE Connector Architecture specification.

	For more information about J2EE Connectors, see the book <i>J2EE Connector Architecture and Enterprise Application Integration</i> , by Sharma, Stearns, and Ng (Addison-Wesley Professional). Also see the J2CA 1.0 Connector Specification and the J2EE 1.3 standard.
---	---

1.4. Public VistALink APIs Documentation

The VistALink 1.5 distribution zip file supplies full Javadoc API reference documentation in the `/javadoc` folder. The VistALink Javadoc describes the various Java classes that make up the public VistALink programming API. These APIs may be used under the conditions listed in the Javadoc documentation. VistALink classes that are not documented in the VistALink Javadoc are not part of the supported VistALink API.

For more information about the Javadoc documentation format, please see

<http://java.sun.com/j2se/javadoc/>.

1.5. Sample Applications for J2EE Server

See the J2EE Developer Samples for examples of how to use VistALink with a J2EE server. The Developer Samples are supplied in the VistALink 1.5 distribution zip file for both J2EE and J2SE modes.

2. Developer Workstation Setup

2.1. J2EE Development

2.1.1. IDE

The following libraries should be on the project classpath of your J2EE project in your integrated development environment (IDE):

- vljConnector-1.5.0.jar
- vljFoundationsLib-1.5.0.jar
- A J2EE 1.3 library (e.g., j2ee.jar, weblogic.jar, MyEclipseIDE's j2ee library, etc.)

The **vlj* library** jars are provided in the **/jars** folder of the distribution zip file.

2.1.2. J2EE Runtime

At runtime, additional libraries are required for your J2EE environment. See the *VistALink 1.5 Installation Guide* and the *VistALink 1.5 System Management Guide* for more information on setting up VistALink connectors in a J2EE container.

2.2. J2SE Development

2.2.1. IDE

The following libraries need to be on the project classpath of your J2SE project in your IDE:

- vljConnector-1.5.0.jar
- vljFoundationsLib-1.5.0.jar
- vljSecurity-1.5.0.jar
- A J2EE 1.3 library (e.g., j2ee.jar, weblogic.jar, MyEclipseIDE's j2ee library, etc.)

The **vlj* library** jars are provided in **/jars** folder of the distribution zip file.

2.2.2. J2SE Runtime

At runtime, these additional jar files are required for your J2SE application to launch and run:

- j2ee.jar
- jaxen-core.jar
- jaxen-dom.jar
- saxpath.jar
- log4j-1.2.8.jar
- xbean.jar

The **j2ee.jar** can be obtained from either the J2EE 1.3.SDK or the **weblogic.jar**. The others can be found in the **rar/ExplodedVistaLinkRAR/lib** folder, in the VistALink 1.5 distribution file.

3. Using VistALink in J2EE

3.1. Using Station Number (Institution) and Subdivision

VistALink asks application code to provide VHA institution station numbers in two situations, for different purposes: system location and multi-division awareness. Because these two modes of use are separate, the meaning of *station number* is “overloaded.” The two modes are described in the sections below.

3.1.1. System Locator: Institution-Connector Mapping

Your code must provide a station number to retrieve a connector JNDI name from VistALink's institution-mapping API. Lacking anything better, the station number is used as a location designator for a particular system.

Each connector has a `primaryStation` attribute, which is configured by the J2EE administrator. On the M system, this value should exactly match the DEFAULT INSTITUTION value in the Kernel Systems Parameter File. VistALink uses `primaryStation` to confirm that a connector is accessing the correct M system: if the two values don't match, connections are rejected. The `primaryStation` attribute is also the mapping source between station numbers and connector JNDI names.

In 99 percent of the cases, this attribute is all-numeric (e.g., "523"). When asked for the JNDI name for a station number with a subdivision suffix, such as "523A", the institution mapping API will return a JNDI name mapped to “523”. The API handles some special cases too. For example, "5239" would be a nursing home at "523" where "9" is the nursing home indicator.

3.1.2. Multidivision-Aware Application Code: ConnectionSpec Credentials

Your code must also use the station number to create a connection spec, which it then uses to obtain a VistALink connection. When your code executes a remote procedure call (RPC), the station number specified in the connection spec is passed to M to populate the variable `DUZ (2)` on the M system. On M systems, `DUZ (2)` is used to make applications multidivision-aware, to give the right view of the data. On merged systems, for example, an end-user might only see the set of patients matching the station number specified in their `DUZ(2)` (e.g., "523B").

Only one value can be set into `DUZ (2)` at a time. So, if you construct a connection spec (e.g., `connSpec = new VistaLinkVpidConnectionSpec(division, vpid)`) and pass in "523B", VistALink sets `DUZ (2)` on the M side to "523B". RPCs are then executed under that setting (if it is granted as a permissible subdivision to the end-user.)

3.1.3. Example

In a merged site, where an end-user has access to "636A" but not to "636", the institution mapping lookup for "636A" will return the connector whose `primaryStation` attribute is "636". If you create a connection spec and connection to execute an RPC and specify "636A",

this value will be set into DUZ(2) on the M side. The RPC will execute with multi-division awareness, i.e., that the current institution is "636A".

3.2. Request Cycle

Using a J2CA connector such as VistALink in a J2EE environment to execute requests (RPCs) involves the following sequence of steps:

1. Retrieve a particular VistALink connector's connection factory from JNDI
2. Instantiate a ConnectionSpec to use for re-authentication over the connection
3. Get a connection from the connection factory
4. Execute a request over the connection
5. Close the connection.

These steps are discussed in more detail in the sections below.

3.2.1. Retrieving the Connection Factory

To retrieve a connection factory from JNDI, you need to know the JNDI name of the connection factory. Each VistALink connector going to a different destination will have a different JNDI name. For this example, assume that the JNDI name of the connection factory is "vlj/testconnector."

To retrieve the connection factory, make the following calls to JNDI:

```
Context ic = new InitialContext();
String jndiName = "vlj/testconnector";
VistaLinkConnectionFactory cf = null;
// cast JNDI object to VistaLinkConnectionFactory
cf = (VistaLinkConnectionFactory) ic.lookup(jndiName);
```

Note that the object retrieved from the JNDI must be cast to the `VistaLinkConnectionFactory` class.

If your application hard-codes JNDI connection factory names, you may want to use the J2EE `resource-ref` mechanism to loosely couple a hard-coded JNDI name in your application source code to an administrator-modifiable mapping in your application deployment descriptors.

More likely, however, in a VA environment, your application would retrieve JNDI connection factory names *dynamically*, based on VA facility station number. See the [“Institution Mapping”](#) section later in this chapter, which describes how to connect to a particular VA site by getting the JNDI name for a connector's connection factory.

3.2.2. Instantiating a Connection Spec for Re-authentication

Before retrieving a connection from the connection factory object, you need to instantiate a connection spec. The connection spec is used to pass re-authentication information to the

connection, identifying the user to run the request under on the target VistA M system. (For a complete discussion of VistALink's re-authentication mechanism, see "[More about Re-authentication](#)" below, as well as the titled "Security" in the *VistALink System Management Guide*.)

The following connection specs are provided with VistALink for general application use:

- `VistaLinkVpidConnectionSpec`(division, vpid)
- `VistaLinkDuzConnectionSpec`(division, duz)
- `VistaLinkAppProxyConnectionSpec`(division, appProxyName)

To instantiate a `VistaLinkVpidConnectionSpec`, for example:

```
String division="523A";
String vpid = "0000002987654321V654321000000";
VistaLinkVpidConnectionSpec connSpec = new
    VistaLinkVpidConnectionSpec(division, vpid);
```

For more information on connection specs, see the section "[Connection Specification Classes](#)" in this document.

3.2.3. Getting a Connection (Connection Spec)

The following code retrieves a VistALink connection from a given connection factory:

```
VistaLinkConnection myConnection = null;
// cast connection factory getConnection to VistaLinkConnection
myConnection = (VistaLinkConnection) cf.getConnection(connSpec);
myConnection.setTimeout(10000); // set request timeout to 10 seconds
```

Every connection has a default timeout, which is the time the J2EE side of the connection waits for a response from the M system when executing an RPC. You can manually increase the timeout if you expect the RPCs to take a long time (as in the example above).

3.2.4. Executing a Request

Once you have a connection, you can execute a request over the connection. In general the steps to execute a request are:

1. Create an `RpcRequest` object
2. Set the `RpcRequest` name of the RPC to execute (`RpcRequest.setRpcName`)
3. Set the type of request transmission format to use (we recommend proprietary in all cases, rather than XML)
4. Assert an `RpcRequest` authorization context (`RpcRequest.setRpcContext`)
5. Set request parameters (if the request has any)
6. Execute the request
7. Process the results.

For example:

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
vReq.setRpcName("XOBV TEST PING");
vReq.setUseProprietaryMessageFormat(true);
vReq.setRpcContext("XOBV VISTALINK TESTER");
RpcResponse vResp = myConnection.executeRPC(vReq);
String results = vResp.getResults();

```

More complete information on setting up a request, passing parameters to the request, and processing the response, is provided in the [“Executing Requests”](#) section.

3.2.5. Closing the Connection

The final step in executing a request is to close the connection. Doing this does not close the physical connection in J2EE; instead, it returns the connection to the connection pool it came from, for possible re-use on a subsequent request.

It is strongly recommended that you close the connection in a final block surrounding all of the code, beginning at `getConnection()`. Otherwise, errors will result in leaked connections that have not been returned to the pool, possibly causing the pool to run out of available connections for other callers.

The following example shows the complete request cycle, including closing the connection.

```

VistaLinkConnection myConnection = null;
String results = null;

try {
    Context ic = new InitialContext();
    String jndiName = "vlj/testconnector";
    VistaLinkConnectionFactory cf = (VistaLinkConnectionFactory)
        ic.lookup(jndiName);
    myConnection = (VistaLinkConnection) cf.getConnection(connSpec);
    myConnection.setTimeout(10000); // set 10 second socket timeout

    RpcRequest vReq = RpcRequestFactory.getRpcRequest();
    vReq.setUseProprietaryMessageFormat(true);
    vReq.setRpcName("XOBV TEST PING");
    vReq.setRpcContext("XOBV VISTALINK TESTER");
    RpcResponse vResp = myConnection.executeRPC(vReq);
    results = vResp.getResults();

} catch (VistaLinkFaultException e) {
    // ...
} catch (NamingException e) {
    // ...
} catch (ResourceException e) {
    // ...
} catch (FoundationsException e) {
    // ...
} finally {

```

```

if (myConnection != null) {
    try {
        myConnection.close();
    } catch (ResourceException e) {
        // ...
    }
}
}
}

```

3.2.6. Connectivity Failures and Retry Strategies

During an `executeRPC()` call, connectivity to the VistA system can fail due a network failure or anything else that breaks connectivity. The request's implementation of the `VistaLinkRequestRetryStrategy` interface determines whether VistALink automatically retries the request or not.

Connectivity can fail at any point while executing the original request. If it does, most or all of the work of the original request may have been performed already. You should take this into account when considering what retry strategy to use for your requests -- for example, whether a an RPC retry could potentially add a second instance of a particular entry to a file.

By default, the implementation class `VistaLinkRequestRetryStrategyAllow` is set as the retry strategy for a request. Its single method, `execute()`, returns "true," causing VistALink to attempt to obtain a new connection and retry the request, exactly once.

In addition to `VistaLinkRequestRetryStrategyAllow`, an additional implementation class, `VistaLinkRequestRetryStrategyDeny`, is supplied. Its single method, `execute()`, returns "false." You can use this class in cases where you never want the request to be retried.

You can also supply your own implementation class for the `VistaLinkRequestRetryStrategy` interface that uses its own logic to determine whether to return "true" or "false" to allow or deny the retry attempt.

For example:

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
vReq.setRpcName("XOBV TEST PING");
vReq.setRpcContext("XOBV VISTALINK TESTER");
vReq.setRetryStrategy(new VistaLinkRequestRetryStrategyDeny());
RpcResponse vResp = myConnection.executeRPC(vReq);
String results = vResp.getResults();

```

3.3. More about Re-authentication

3.3.1. Overview

When a connection is used by an *application*, VistALink uses re-authentication for the following reasons:

- The connection proxy user used by the adapter to connect to M should not have privileges
- Most RPCs need to run in the context of specific end-users.

Re-authentication is a lightweight security context switch from the application server adapter's user identity to the actual end-user identity.

The architecture of VistALink makes the assumption that the identity of the end-user has already been authenticated (verified) by the calling application. Therefore VistALink does not attempt to authenticate the end-user's identity. Instead, the re-authentication process matches the end-user identity already established by the calling application with a matching VistA New Person file entry.

When retrieving a VistALink connection from a connection factory, the application supplies end-user credentials as part of the connection specification. These credentials are used to switch security context on the M side, to re-authenticate the connection. This re-authentication process establishes the correct end-user environment on the M server (VPID, DUZ, etc) for the duration of the connection's use.

3.3.2. Connection Specification Classes

To link identities, the application selects one of the following connection specifications to retrieve a connection:

Table 1. Connection Specification Classes

Connection Specification class	Required Credentials
VistaLinkDuzConnectionSpec	Division; known DUZ value of a specific end-user (to be deprecated in favor of VPID)
VistaLinkVpidConnectionSpec	Division; known VPID value of a specific end-user
VistaLinkAppProxyConnectionSpec	Division; name of a user of the special user type "Application Proxy"

VPID is the connection specification expected to be used in most production scenarios. Whatever end-user authentication mechanism is used by a HealthVet -VistA application, the application should be able to obtain the VPID for a given end-user as an output from the authentication process. During RPC execution, when the end-user is authenticated, the J2EE application can use the VPID as a way to identify the end-user to any VistA M system. Kernel patch XU*8.0*309 is required to support the VPID connection specification on M-VistA systems.

DUZ (`DuzConnectionSpec`) is the primary re-authentication mechanism until the VPID infrastructure is fully rolled out. At that point `DuzConnectionSpec` will be deprecated, and `VpidConnectionSpec` will be the primary mechanism. In both cases, it is expected that the login will have been performed already through a VHA-approved authentication mechanism, and that the authentication mechanism will make the DUZ or VPID available for use by the application.

The Application Proxy connection specification is designed to be used in cases where the RPC should run on the Kernel/M system under the identity of an application, rather than an end-user.

3.3.3. Institution/Division Rules for Re-authentication

For every connection spec, a division must be passed: the division parameter is mandatory. This requirement ensures that the division requested for a connection on behalf of a user matches the division under which the user's request is executed on the M system. It also helps to ensure that RPCs are not executed on the wrong M system.

All applications should already be multidivision-aware, so the VistALink requirement that an application know the end-user's division should be no additional burden to applications. This value is set by Kernel into DUZ(2) for the re-authenticated end-user.

The value to pass for the division parameter for any connection specification is the division station number, e.g., "523" or "523BZ". This is the value found in field 99 (Station Number) of the corresponding entry in the Institution File on the M system.

The following applies to user-based connection specs (`VistaLinkVpidConnectionSpec` and `VistaLinkDuzConnectionSpec`) on the M side:

- If the end-user's DIVISION (#200.02) multiple of their New Person file entry is empty, the division passed in with the connection spec must be the station number of the division set into the DEFAULT INSTITUTION (#217) field of the KERNEL SYSTEM PARAMETERS (#8989.3) file entry for the site.
- If an end-user has one or more divisions specified in the DIVISION (#200.02) multiple of their New Person file entry, the division passed in with the connection spec must be the station number for one of the divisions present in that multiple.

For the `VistaLinkApplicationProxyConnectionSpec`, the division must be a division supported on the computing system being connected to. In both cases, if the division passed does not meet the conditions above, re-authentication fails, and a `SecurityDivisionDeterminationFaultException` is returned to the calling application.

3.3.4. Application Proxy User

Kernel patch XU*8.0*361 provides the following public API for application proxy user support:

- `CREATE (NAME, FMAC, OPT) ;Create an APPLICATION PROXY user`
For a description of this API, see the Kernel Programmer Manual on the Kernel API Web site (<http://vista.med.va.gov/kernel/apis/index.shtml>) or the VistA Document Library (<http://www.va.gov/vdl/Infrastructure.asp?applD=10>).

VistALink uses patch 361 functionality to implement a new re-authentication connection spec, `VistaLinkAppProxyConnectionSpec`. With this, re-authentication can be performed using an application proxy account on the M system, rather than under an end-user account.

The Application Proxy connection specification is expected to be used in either of the following special situations:

- The J2EE end-user does not have a user account on the M system on which an RPC is to be executed – i.e., when a service uses VistALink from an EJB (when an end-user is not practical)
- It is not appropriate for the RPC to execute under the identity of a particular end-user

To use the Application Proxy connection, you should understand the following:

- `VistaLinkAppProxyConnectionSpec(String division, String appProxyName)` is the constructor for the new connection spec.
- Kernel patch XU*8*361 supports the new functionality.
- `SecurityIdentityDeterminationFaultException` is thrown if re-authentication fails.
- For the division, you can specify any division that is valid for the site. Division checking is done against what is a valid division for the site, not against user-specific division settings.

If your functionality is not multi-divisional, you can use "primary station" for the M system. This is the station number in the DEFAULT INSTITUTION field of the Kernel System Parameters file in the M account in question.

Note: According to Infrastructure & Security Services, a valid division for a Kernel/M site is currently any division whose numeric M value (e.g., when “plussed”) (1) equals the DEFAULT INSTITUTION station number, and (2) is not marked inactive in the Institution file at the site.

- A "tester" application proxy user is distributed with VistALink, and added to the New Person file during the installation post-init. The proxy user is named `XOBVTESTER,APPLICATION PROXY`.
- The VistALink sample Web application demonstrates the use of all connection specs, including `VistaLinkAppProxyConnectionSpec`. The sample application makes use of the `XOBVTESTER,APPLICATION PROXY` user created during the VistALink installation post-init.

The M example of adding an application proxy user is shown below. In this case, the proxy name is passed into the function; no FileMan access code is added to the user; and the [XOBV VISTALINK TESTER] B-type option is added to the secondary menu of the proxy user:

```
ADDPROXY(XOBANAME)    ; add application proxy if not present
; depends on XU*8*361
NEW XOBID
;
SET XOBID=$$CREATE^XUSAP(XOBANAME,"","XOBV VISTALINK TESTER","")
```

```

IF (+XOBID)>0 DO
. ; actions if successfully added
IF (+XOBID)=0 DO
. ; actions if proxy user was already present
IF (+XOBID)<0 DO
. ; actions if error - could not add user for some reason
QUIT

```

3.3.4.1. J2EE Application Proxy Usage Example

```

StringBuffer results = new StringBuffer();
String appProxyName = "XOBVTESTER,APPLICATION PROXY";
String division="11000";

try {

    VistaLinkConnectionSpec connSpec = new
        VistaLinkAppProxyConnectionSpec(division, appProxyName);
    String jndiName = InstitutionMappingDelegate.
        getJndiConnectorNameForInstitution(division);
    Context ic = new InitialContext();
    VistaLinkConnectionFactory cf = (VistaLinkConnectionFactory)
        ic.lookup(jndiName);
    VistaLinkConnection myConnection = (VistaLinkConnection)
        cf.getConnection(connSpec);
    RpcRequest vReq = RpcRequestFactory.getRpcRequest();
    vReq.setUseProprietaryMessageFormat(true);
    vReq.setRpcContext("XOBV VISTALINK TESTER");
    vReq.setRpcName("XOBV TEST PING");
    RpcResponse vResp = myConnection.executeRPC(vReq);
    results.append("<p>" + rpcName + " Results: <b>" +
        vResp.getResults() + "</b>");
} catch (Exception e) {
    // ...
} finally {
    if (myConnection != null) {
        try {
            myConnection.close();
        } catch (ResourceException e) {
            //...
        }
    }
}
}

```

3.4. *Timeouts*

3.4.1. **Socket-Level Forced Timeout**

A simple socket timeout capability is provided so that the Java side of the connection can simply time out the M side of the connection if an RPC is taking too long to execute. If the timeout is reached, the socket will drop the connection to M. On the M side, the RPC will terminate ungracefully.

For RPCs that are known to be long-running, you may want to use the socket timeout in conjunction with a graceful RPC timeout. Set the socket timeout slightly longer than you set the graceful RPC-based request-level timeout. (See the “[Graceful \(Request Level\) Timeout](#)” section below for more information about how to implement graceful timeouts.)

3.4.1.1. **Setting Socket-Level Timeouts**

The socket-level timeout can be set in two ways:

- On the connection object, the `setTimeout()` method will set a socket timeout, in milliseconds, that will be used for all RPCs executed over the connection. For example:

```
// set timeout for all requests sent over this connection to 10 seconds
myConnection.setTimeout(10000);
```

- On the RPC request, the socket timeout can be set for a single request:

```
// set request timeout to 10 seconds, for this request only
myRpcRequest.setTimeout(10000);
```

3.4.1.2. **Default Socket-Level Timeout**

All connectors are configured with a default socket timeout, so that if a request takes infinitely long to complete, a socket timeout will eventually always be triggered.

Administrators can selectively configure (tune) the default socket timeout for each connector, depending on the WAN performance and system performance for reaching any given M system. The default timeout is set in the `gov.va.med.vistalink.connectorConfig.xml` file. (See the section “Connector Settings,” in the *VistALink 1.5 System Management Guide*.)

3.4.1.3. **Changing Socket Timeout as a Multiple of Default Timeout**

If you are going to programmatically adjust the timeout, you may want to obtain the current timeout for the connector first, multiply it by a factor, and then set the new timeout. This takes advantage of any tuning the administrators may have done for a particular connector.

Example:

```
// increase timeout by a factor of two
int timeout = myConnection.getTimeout();
myConnection.setTimeout(timeout*2);
```

3.4.2. **Graceful (Request-Level) Timeout**

In addition to using a socket timeout, the calling Java application can also pass a graceful timeout value to the RPC it is going to execute. To implement a graceful timeout, the RPC code that is executing must check whether it has timed out against this timeout value. A set of M APIs

is provided for applications to check whether their RPC has timed out, based on the value passed by the calling application with the request.

Implementing a graceful timeout requires a delicate synchronization process between the graceful timeout, the socket timeout, and the RPC execution on the M side. Therefore, the graceful timeout is only recommended if your application needs more than what the socket timeout provides.

To implement a graceful timeout:

1. Java-side: The calling application sets the request level `RpcClientTimeout` property, with `RpcRequest.setRpcClientTimeout(numberOfSeconds)`.
2. Java-side: Make sure the current connection-level socket timeout (in milliseconds) evaluates to a longer period than the graceful timeout (in seconds). Otherwise the socket may timeout anyway before the graceful timeout is reached. For example, if you set `RpcRequest.setRpcClientTimeout(10)`, you could do `myConnection.setTimeout(15000)`, i.e., 15 seconds.
3. M-side RPC: The RPC code should periodically check (e.g., if code is executing an iterative loop) if the RPC has exceeded the client timeout by calling `$$STOP^XOBVLIB()`. Return values from `$$STOP^XOBVLIB` are: “1” (application should stop processing, timeout has been exceeded) or “0” (continue processing).
4. Java-side: The calling application can catch the `RpcTimeOutFaultException` in the try/catch code surrounding its RPC execute call. If the calling application catches this exception when executing an RPC, it means the M-side RPC checked `STOP^XOBVLIB`, was notified that a timeout had occurred, and did not reset the timeout value.

Note: If the RPC continues processing without resetting the timeout after receiving a timeout indicator from `$$STOP^XOBVLIB`, the RPC may complete, but VistALink will return an `RpcTimeOutFaultException` to the client.

3.4.2.1. STOP^XOBVLIB()

Used by the application to determine if processing should stop.

Input variables: (none)

Output variables: Return value. “1” is an indicator to stop processing; “0” is an indicator to continue processing. If “1” is returned, and internal “timed out” indicator is also set.

Note: If you call `STOP^XOBVLIB` and it returns 1, a fault will be returned to the calling Java application. This will happen even if your RPC code completes, unless you call `SETTO^XOBVLIB` to increase the timeout and then call `STOP^XOBVLIB` to clear the "timed out" indicator based on the higher timeout value.

3.4.2.2. \$\$GETTO^XOBVLIB()

Get the current timeout value (default = 300 seconds). An application would call this to obtain the current timeout value.

Output Variable: Return value, which is the timeout value, if it exists (in seconds), or the default of 300 seconds.

3.4.2.3. \$\$SETTO^XOBVLIB()

Override the current "graceful" timeout setting received from the client via `RpcRequest.setRpcClientTimeout(int)` or the default.

It is suggested that you call `STOP^XOBVLIB` immediately after resetting the timeout value, in order to reset the current timeout indicator based on the new timeout value.

Input Variable: TO. TO is the RPC timeout value in seconds. This is always the total number of seconds since the RPC began; it is not an increment from the current time.

Output Variable: Return value. The function sets the RPC timeout value (in seconds) and returns a "1" to indicate value successfully reset or 0 if not successful.

For example, if the M-side RPC wants "more time" after getting a "1" from `$$STOP^XOBVLIB`, it can use the `$$SETTTO` call to do so. However, this is risky, because the socket-level timeout may time out the connection anyway, particularly if the calling application set the socket-level timeout just higher than it set the graceful RPC timeout.

To add "more time" (though it risks running into a socket-level timeout), an RPC could get the current timeout value (`$$GETTO^XOBVLIB`), increase it, and then reset the higher value with `$$SETTO^XOBVLIB`. It should also call `$$STOP^XOBVLIB` immediately after calling `$$SETTO`, in order to reset the "timed out" indicator based on the new value. `$$STOP` needs to be called again because the new `$$SETTO` value may not have been large enough. The timeout check is always calculated from the start of the RPC, not the reset.

3.4.2.4. Java and M Code RPC Timeout Call Examples

Java-side example:

```
// increase socket timeout by a factor of two and
// use the original socket timeout as RPC client timeout
int timeout = myConnection.getTimeout();
myConnection.setTimeout(timeout*5);
myConnection.setRpcClientTimeout(timeout/1000);
```

M/VistA-side example:

```

...
F  S IEN=$O (ARR (IEN)) Q:IEN="" DO I $$$STOP^XOBVLIB () DO CLEANUP QUIT
. DO PROCESS (IEN)
...

```

3.5. Institution Mapping

HealthVet VistA applications need to be able to dynamically retrieve connectors to various VistA systems. The connecting systems will change over time, so hard-coding of connector references is out of the question.

VistALink provides an Institution Mapping facility so that administrators deploying VistALink connectors can map each connector to a specific VHA institution, using that institution's station number. HealthVet VistA applications can then retrieve the JNDI name for a connector to a particular institution, using the institution mapping facility. This utility is in the **gov.va.med.vistalink.institution** package.

There is no requirement to use this utility to use VistALink. The utility merely provides a way for administrators to associate station numbers with JNDI names, and for runtime code to retrieve the mapping.

3.5.1. How to Configure Mappings

Each connector is configured in a file named **gov.va.med.vistalink.connectorConfig.xml**. Each connector's settings are stored in a unique <connector> element in that file. The station number and JNDI name it maps to are both XML attributes of the <connector> element.

Refer to the *VistALink 1.5 Installation Guide* and the *VistALink System Management Guide* for a complete description of how to configure VistALink's institution mappings for each VistALink connector.

3.5.2. How to View the Currently Loaded Mappings

You can view the currently loaded institution mappings for a given server by using the Institution Mappings tab of the VistALink console. Refer to the *VistALink 1.5 System Management Guide* for a complete description of the VistALink console.

3.5.3. Retrieving Mappings for Applications

A static method, `getJndiConnectorNameForInstitution`, is provided in the class **gov.va.med.vistalink.institution.InstitutionMappingDelegate**. This class provides application access to the institution mappings. For example:

```

String stationNumber = 500;
String jndiConnectorName = null;
try {
    jndiConnectorName =

```

```
InstitutionMappingDelegate.getJndiConnectorNameForInstitution(  
    stationNumber);  
} catch (InstitutionMappingNotFoundException e) {  
    // take some action  
} catch (InstitutionMapNotInitializedException e) {  
    // take some action  
}
```

3.5.4. Subdivisions

When retrieving the JNDI name for a particular station number, you should pass the exact subdivision you are working with to the `getJndiConnectorNameForInstitution()` call (e.g., "523A", "523B", or "523"). This API determines the correct connector associated with a given station number, even if the station number parameter passed to it is a subdivision (usually *but not always* signified by the presence of alpha characters after the numeric portion of the station number).

3.6. VistALink Java API Reference

For a complete reference to all of the Java-side VistALink interfaces, classes, methods and exceptions, please see the Javadoc API documentation provided in the VistALink 1.5 distribution file.

4. Executing Requests

4.1. Remote Procedure Calls

A remote procedure call (RPC) is a defined call to M code that runs on an M server. Through the RPC Broker, a client application can make a call to the M server and execute an RPC on the M server. This is the mechanism through which a client application can:

- Send data to an M server
- Execute code on an M server
- Retrieve data from an M server

An RPC can take optional parameters to do a task and then return either a single value or an array to the client application.



For detailed information on RPCs, please refer to *Getting Started With the Broker Development Kit (BDK)* and/or the *RPC Broker Technical Manual*. You can find both publications at <http://www.va.gov/vdl/>.

4.1.1. RPC Security (“B”-Type Option)

All RPCs are secured with an RPC context (a "B"-type option). The end-user on whose behalf an RPC is executed must have the “B”-type option associated with the RPC in their menu tree. Otherwise an exception is thrown.



For more information on RPC security, see *Getting Started with the BDK, Chapter 3 (Extract): RPC Overview*, which is bundled in the VistALink distribution, as the file **xwb1_1p13dg-rpc_extract.pdf**.

4.1.2. RPCs for Use by Application Proxy Users

RPCs must be explicitly marked as supporting execution by an application proxy user, in order to be used by one. The new field APP PROXY ALLOWED (#.11) in the REMOTE PROCEDURE file (#8994) must be marked “YES.” RPCs should only be marked for application proxy use if:

- The business logic behind the RPC is valid when the DUZ represents an application proxy user (rather than end-user)
- The application expects to be executed by an application proxy user.

4.2. Request Processing

During interactions with M from Java, developers use the `RpcRequest` object. The `RpcRequest` object encapsulates the data that will be sent to M to execute an interaction, i.e. the RPC name, RPC parameters, and RPC context. The `RpcRequest` object is constructed

using the `RpcRequestFactory` object. The RPC parameters are accessed through the `RpcRequest` objects via the `clearParams()`, `getParams()` and `setParams()` methods.

4.2.1. Get an `RpcRequest` Object: `RpcRequestFactory` Class

The `RpcRequest` class represents a request from Java to M. As the transport format, it permits the use of either XML or a proprietary format. The proprietary format is the default and is recommended because it is faster. This class also exposes methods for specifying `Rpc Name`, `Rpc Context` and the parameters used by M to execute the RPC.

The `RpcRequestFactory` class is responsible for creating instances of `RpcRequest`. In order to create an `RpcRequest`, the developer must call the static `getRpcRequest` method on this class. In the example shown below, `getRpcRequest` is overloaded with three declarations:

```
public static RpcRequest getRpcRequest() throws FoundationsException
```

This method is used to create a default `RpcRequest` with no specified `Rpc Name` or `Rpc Context`. You must specify the `Rpc Context` and the `Rpc Name` on the `RpcRequest` object before you can use this object in an interaction. Refer to javadoc on `RpcRequest` for more information:

```
public static RpcRequest getRpcRequest(String rpcContext) throws  
FoundationsException
```

This method is used to create a `RpcRequest` with the specified `Rpc Context`. You must specify the `Rpc Name` on the `RpcRequest` object before you can use this object in an interaction.

```
public static RpcRequest getRpcRequest(String rpcContext, String rpcName)  
throws FoundationsException
```

Refer to the Javadocs on `RpcRequest` for more information.

This method is used to create a `RpcRequest` with the specified `Rpc Context` and the `Rpc Name`. You may still specify another `Rpc Context` and `Rpc Name` on this object. Refer to the JavaDocs on `RpcRequest` for more information.

4.2.1.1. `getRpcRequest()` Example

```
RpcRequest vReq = null;  
  
//The Rpc Context  
String rpcContext = "XOBV VISTALINK TESTER";  
  
//The Rpc to call  
String rpcName = "XWB GET VARIABLE VALUE";  
  
// Construct the request object  
try{  
    vReq = RpcRequestFactory.getRpcRequest(rpcContext, rpcName);
```

```

} catch(FoundationsException e) {
    // process exception as needed
}

```

4.2.2. Set RpcRequest Parameters: “Explicit” Style

There are two ways of passing RPC parameters to an `RpcRequest` object. For convenience, the first method is referred to as “explicit” style, the second method as “setParams” style.

In explicit style, the method of passing RPC parameters corresponds very closely to the underlying RPC parameters. The RPC Broker has three input parameter types for RPC calls. These map to VistALink `RpcRequestParam` parameter types as follows:

RPC parameter type	VistALink <code>RpcRequestParam</code> type	Java value type
Literal	"string"	String
Reference	"ref"	String
List	"array"	List (Specify array indices yourself; supports non-integer, negative, or multi-level indices) Map or Set (index automatically generated as a single-level, integer, sequence)

To pass parameters into an `RpcRequest`, you should retrieve the `RpcRequest`'s mutable `RpcRequestParam` object, accessible by `RpcRequest.getParams()`. Then, call `setParam()` on that object to define each RPC parameter:

```
void setParam(int position, String type, Object value)
```

The parameters to this call are:

- **position:** the expected RPC parameter list position where the RPC expects to see the RPC parameter
- **type:** can be "string", "ref" or "array" (to match the expected RPC parameter type)
- **value:** an object that is the value of the RPC parameter. For "string" or "ref" types, the object should be a string. For "array" types, the object should implement the Map, List or Set interface.

4.2.2.1. Literal RPC Parameter Example

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
vReq.getParams().setParam(1, "string", "I am a string");

```

4.2.2.2. Reference RPC Parameter Example

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
vReq.getParams().setParam(1, "ref", "DTIME");

```

4.2.2.3. List RPC Parameter Example:

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList nums = new ArrayList();
nums.add("3");
nums.add("5");
nums.add("4");
nums.add("1");
nums.add("7");
nums.add("8");
nums.add("2");
nums.add("6");
nums.add("9");
vReq.getParams().setParam(1, "array", nums);

```

4.2.2.4. Combination RPC Parameter Example:

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList nums = new ArrayList();
nums.add("3");
nums.add("5");
nums.add("4");
nums.add("1");
nums.add("7");
nums.add("8");
nums.add("2");
nums.add("6");
nums.add("9");
vReq.getParams().setParam(1, "array", nums);
vReq.getParams().setParam(2, "string", "I am a string");
vReq.getParams().setParam(3, "ref", "DTIME");

```

4.2.3. Set RpcRequest Parameters: “setParams” Style

A second style of passing RPC parameters for an RPC into an `RpcRequest` object is called the “setParams” style. This method offers a small amount of abstraction from the underlying RPC, although parameters still must correspond directly to what is expected by the RPC being invoked.

With the `setParams` style, you create an object that implements the `List` interface and holds each of the parameters as an object entry in the list. Add each parameter to the `List` as an object value, and then use `RpcRequest.setParams(List)` to pass the RPC parameters to the request..

The `RpcRequest` object processes the `List` internally extracting the RPC parameter characteristics for the request as follows:

- **position:** Determined by the order each object was added to the `List`. The first object added becomes the first RPC parameter, the second becomes the second, and so forth.
- **type:**
 - If an object found in the `List` is a `String`, it is passed as an RPC literal parameter.

- If an object found in the List implements the Map, List, or Set interfaces, it is passed as an RPC List parameter.
- If an object found in the List is an instance of the special `RpcReferenceType` class, it is passed as an RPC reference parameter.
- **value:** The value for the RPC parameter is simply the object added to the List for each parameter.

4.2.3.1. Literal RPC Parameter Example:

```
RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
params.add("I am a string");
vReq.setParams(params);
```

4.2.3.2. Reference RPC Parameter Example:

```
RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
params.add(new RpcReferenceType("DTIME"));
vReq.setParams(params);
```

4.2.3.3. List RPC Parameter Example:

```
RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
ArrayList nums = new ArrayList();
nums.add("3");
nums.add("5");
nums.add("4");
nums.add("1");
nums.add("7");
nums.add("8");
nums.add("2");
nums.add("6");
nums.add("9");
params.add(nums);
vReq.setParams(params);
```

4.2.3.4. Combination RPC Parameter Example:

```
RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
ArrayList nums = new ArrayList();
nums.add("3");
nums.add("5");
nums.add("4");
nums.add("1");
nums.add("7");
```

```

nums.add("8");
nums.add("2");
nums.add("6");
nums.add("9");
params.add(nums);
params.add("I am a string");
params.add(new RpcReferenceType("DTIME"));
vReq.setParams(params);

```

4.2.4. Specifying Indices for List-Type RPC Parameters

List-type RPC parameter values can be passed to VistALink in Java objects that implement the `List`, `Set`, or `Map` interfaces.

If you pass List-type parameter values in an object that implements `List` or `Set` (but not `Map`), the array index is automatically generated for you, as a single-level, integer, sequential index starting at “1.” (The array becomes the array subscript level in M for the data.) This can be convenient if the array index is not significant for your RPC.

However, if you need any of the following features in your array when it is passed to M, use an object that implements `Map` (such as `HashMap`) instead:

- Negative index values
- Non-sequential index values
- Control over the index start point
- Non-integer index values
- Multi-level indices (>1 subscript level in M)

For each entry added to the `Map` object, the key becomes the M subscript, and the value becomes the M value.

To pass a multi-level subscript, use the `RpcRequest.buildMultipleMSubscriptKey()` method to generate the `HashMap` key.

4.2.4.1. List RPC Parameter Example (Explicit Index)

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
HashMap hm = new HashMap();
hm.add(".11", "0");
hm.add("1202", "23");
hm.add("1205", "595");
hm.add("time", "NOW");
params.add(hm);
vReq.setParams(params);

```

4.2.4.2. List RPC Parameter Example (Explicit Multi-Level Index)

```

RpcRequest vReq = RpcRequestFactory.getRpcRequest();
ArrayList params = new ArrayList();
HashMap hm = new HashMap();
hm.put(RpcRequest.buildMultipleMSubscriptKey("\FRUIT\","1"), "Apple");
hm.put(RpcRequest.buildMultipleMSubscriptKey("\FRUIT\","2"), "Orange");
hm.put(RpcRequest.buildMultipleMSubscriptKey("\FRUIT\","3"), "Pear");
hm.put(RpcRequest.buildMultipleMSubscriptKey("\FRUIT\","4"), "'nana");
params.add(hm);
vReq.setParams(params);

```

4.2.5. Other Useful RpcRequest Methods

For a complete list of other available `RpcRequest` methods, please see the Javadoc for `RpcRequest`.

4.2.5.1. Clear Previous Request Parameters

If you are re-using a request object for additional requests, the `RpcRequest.clearParams` method is provided so that you can clear any existing parameters. You should call this method before attempting to set RPC parameters for subsequent requests:

```

//clear the params
vReq.clearParams();

```

4.2.5.2. Set the Message Format (Proprietary or XML)

VistALink's XML message format is based on the XML standard. As the transport format, it permits the use of either XML or a proprietary format. `RpcRequest` defaults to the proprietary format because it is faster.

```

//Set the request to use the proprietary message format
vReq.setUseProprietaryMessageFormat(true);

//Set the request to use the XML message format
vReq.setUseProprietaryMessageFormat(false);

```

4.2.5.3. Set the RPC Context

If you are re-using a request object for additional requests, you can change the RPC Context with this method:

```

private static final String RPCCONTEXT = "XOBV VISTALINK TESTER";
vReq.setRpcContext(RPCCONTEXT);

```

4.2.5.4. Set the RPC Name

If you are re-using a request object for additional requests, you can change the RPC Name with this method:

```

vReq.setRpcName("XOBV TEST NOT IN CONTEXT");

```

4.2.5.5. Set the RPC Client Timeout

This method sets a “graceful” timeout period that is made available to the RPC code on the M system. It is up to the M code to honor the timeout period, however. The value sent defaults to 600 seconds. You can change this value by calling the `setRpcClientTimeOut` method:

```
private static final int TIMEOUT = 300;
vReq.setRpcClientTimeOut(TIMEOUT);
```

For more information on the different kinds of timeouts, see the "[Timeouts](#)" section.

4.3. *Response Processing*

Once you have set up `RpcRequest`, you can execute an RPC interaction on the `VistaLinkConnection` object, using the `RpcRequest` object. Doing this returns an `RpcResponse` object. `RpcResponse` is a value object that provides information about the response returned from M.

4.3.1. `RpcResponse` Class

The `RpcResponse` class is a value object that provides information about the response returned from M. The `RpcResponse` object exposes methods to retrieve the results, results type, and an `org.w3c.dom.document` object that contains the results, if the results are in XML format.

4.3.2. Request/Response Example

```
//request and response objects
RpcRequest vReq = null;
RpcResponse vResp = null;

//The Rpc Context
String rpcContext = "XOBV VISTALINK TESTER";

//The Rpc to call
String rpcName = "XOBV TEST STRING";

//Construct the request object
try {
    vReq = RpcRequestFactory.getRpcRequest(rpcContext, rpcName);
} catch (FoundationsException e) {
    // process exception as needed
}

//clear the params
vReq.clearParams();

//Set the params
vReq.getParams(). setParam(1, "string", "This is a test string!");
```

```

//Set the request to use the proprietary message format
vReq.setUseProprietaryMessageFormat(true);

//Execute the Rpc and construct the response with the
//RpcResponseFactory
try {
    vResp = vistaLinkConnection.executeRPC(vReq);
} catch(VistaLinkFaultException e) {
    // process exception as needed
} catch(FoundationsException e) {
    // process exception as needed
}

//Display the response
System.out.println(vResp.getResults());

```

4.3.3. Parsing RPC Results

All results from RPCs are returned as a single string from `RpcResponse.getResults()`.

The RPC Broker defines five return types for RPCs (at the time of the current VistALink release). The mapping between these return types and the single string returned through VistALink are as follows:

Table 2. Mapping RPC Return Types to VistALink Result String Format

RPC Return Type	VistALink Result String Format
Single Value	As-is
Global Instance	As-is
Array	All array nodes concatenated sequentially, with each delimited by linefeed (ASCII 10) character
Global Array	All array nodes concatenated sequentially, with each delimited by linefeed (ASCII 10) character
Word Processing	Each word processing "line" concatenated sequentially, separated by a linefeed (ASCII 10) character

One easy way to parse array-type results concatenated with line feeds is with the Java string tokenizer. For example:

```

StringTokenizer st = new StringTokenizer(vResp.getResults(), "\n");
int cnt = st.countTokens();
for (int i = 0; i < cnt; i++) {
    system.out.println("Result node " + i + ": " + st.nextToken());
}

```



In JDK 1.5 and forward, Sun deprecates `StringTokenizer` in favor of the `split()` function in `java.lang.String`.

4.3.4. XML Responses

Some newer RPCs may choose to return their results as an XML document. The `RpcResponse` class provides two helper methods to turn the normal results string into an XML document. If you expect the results from an RPC to be an XML document, you can call `RpcResponse.isXmlResponse` to confirm if the response is in XML format, and `RpcResponse.getResultsDocument` to convert the result string into an XML document:

```
//Get a org.w3c.dom.document object that contains the results if set
if (vReq.isXmlResponse()) {
    org.w3c.dom.document xmlDoc = null;
    try{
        xmlDoc = vResp.getResultsDocument();
    }catch(RpcResponseTypeIsNotXmlException e){
        // process exception as needed
    }catch(FoundationsException e){
        // process exception as needed
    }
}
```

4.4. How to Write RPCs

For guidelines on how to write RPC Broker RPCs, please refer to the extract *Getting Started With The BDK Chapter 3 (Extract): RPC Overview* from the *RPC Broker* manual. Some special considerations apply to writing RPCs in the new modes supported by VistALink, including n-tier. These are discussed in the sections below.

4.4.1. Write Stateless RPCs Whenever Possible

When writing new RPCs to execute in an n-tier environment, we recommend making them stateless whenever possible. “Stateless” means that each RPC execution is standalone: when a sequence of RPCs is executed, there is no state maintained on the M system between RPC executions. This way, if you need to use an RPC from one sequence of RPCs in another sequence, your code will be more flexible.

The two main M-side state mechanisms traditionally used in RPCs -- \$J (as a temporary storage index) and global locking -- are both problematic in an n-tier environment. In the n-tier model, there is no guarantee that your requests will execute in the same M job partition. Middle-tier code will be retrieving and returning VistALink connections to a connection pool across a number of client-tier accesses, and because connections are not “dedicated” to a single user session, the underlying M partition may be a different one each time, with a different \$J.

4.4.2. When State is Needed

When you do need to maintain state on the M system between RPC invocations, the two approaches discussed in the sections below are recommended. They are more compatible with the n-tier model.

4.4.2.1. Session ID as Temporary Storage Index

Rather than using \$J for a temporary storage index, we recommend that you implement a session ID, guaranteed to be unique on the M system, to index storage on the M system, and that you maintain the session ID state on the middle tier. You can then use the session ID between middle-tier invocations on a variety of connections with different underlying \$Js and still be able to consistently retrieve your indexed temporary data from the M system.

Note: Kernel may provide a new API for middle tiers to obtain a guaranteed unique M system session identifier to index temporary storage locations.

4.4.2.2. FileMan-Based Lock File

An alternative to the M LOCK command is to implement a FileMan-based lock file mechanism to synchronize locking across requests, connections, and middle-tier accesses. Note that a lock file needs to be used in all application code that needs to honor the lock – whether “roll & scroll,” client-server, or n-tier.

You will probably also need APIs for locking and unlocking. Locks should probably be unlocked automatically after a period of time (e.g., lost connection) or based on some state change or event in the application session.

Note: Foundations may provide FileMan-based lock file functionality in the future.

4.4.3. Pitfalls of Using of \$JOB in Stateful RPCs

When running RPCs in an n-tier model, there is no guarantee that all your requests will execute in the same M job partition. Your middle tier will be retrieving and returning VistaLink connections to a connection pool across client-tier accesses, and each connection retrieved from the connection pool may have a different \$J.

If you have an RPC sequence that shares data across RPCs using temporary M storage indexed by \$J, the RPC sequence must be performed over the same connection (or else \$J will change). In n-tier mode, this means the RPC sequence must be executed over one connection before closing it (returning it to the connection pool).

Another \$J-related issue can occur even if you execute an RPC sequence over a single connection. The default `VistaLinkRequestRetryStrategy` implementation allows a request to be retried. If you use it and connectivity fails, the request will be retried on a new connection, and therefore using a new \$J. So if your RPCs communicate by leaving values on the M system indexed by \$J, you should consider using `VistaLinkRequestRetryStrategyDeny` for those requests.

4.4.4. Pitfalls of Global Locking in Stateful RPCs

When running RPCs in an n-tier model, locking global nodes across RPCs presents the same issues as when using \$J across RPCs. In particular, you cannot lock data in an RPC over one connection and unlock data in a different connection. Also, if the M partition that sets a lock

exits, the M operating system automatically clears the lock. Therefore locking can only be used safely within a single RPC, or within an RPC sequence that will be executed over a single connection.

As with \$J, another locking-related issue can occur if network connectivity fails and your request is retried (which would be over a different connection). Therefore you should consider using `VistaLinkRequestRetryStrategyDeny` for requests that depend on locking across RPCs: if connectivity fails and a request is retried, the request will be retried over a new connection that does not own the lock. (The lock may be released anyway, since the previous partition will probably exit as a consequence of the network connectivity failure).

5. VistALink Exception Reference

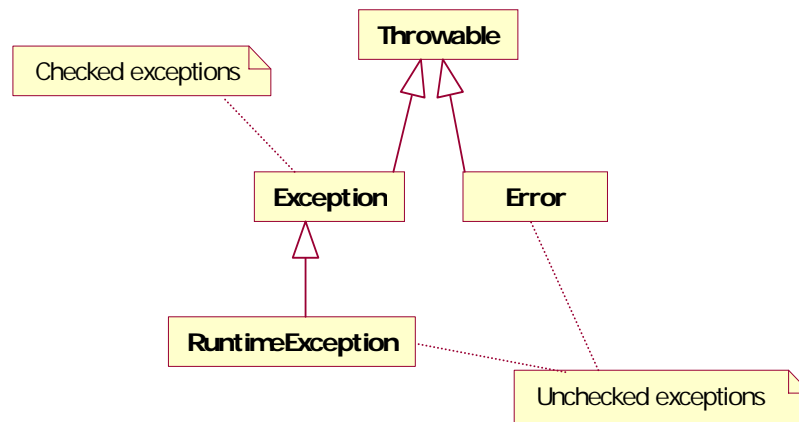
VistALink, like any other Java application, uses exceptions to indicate various error conditions that can occur during execution.

5.1. Checked and Unchecked Exceptions

There are two types of exceptions in the Java programming language:

- **Checked exceptions:** These have to be declared in the method signature “throws” clause if they are thrown by the method. Checked exceptions have to be explicitly caught by the caller within a “try / catch” block.
- **Unchecked exceptions:** These do not have to be declared in the method signature if they are thrown by the method. Unchecked exceptions do not need to be explicitly caught by the caller. Unchecked exceptions can be caught by the caller even if the method does not explicitly throw them.

The diagram below depicts a class hierarchy of base Java exception classes. VistALink throws only checked exceptions.



<code>java.lang.Error</code>	unchecked, reserved to JVM exceptions - memory, out of stack etc.
<code>java.lang.RuntimeException</code>	unchecked, reserved for JVM exceptions that are not as fatal as <code>java.lang.Error</code> exceptions, such as array index out of bound, null pointer exception etc.
<code>class java.lang.Exception</code>	checked, application level exceptions

Figure 1. Java Base Exception Classes

5.2. Catching Exceptions

It is important to remember when an application calls a method that (in turn) throws exception *A*, the application can do two things:

- Catch exception *A*
- Catch any exception *AB* that is subclassed from exception *A*, even though calling a method declaration only declares to throw a parent exception. Example:

```
public class ParentException extends Exception {
}
public class SubException extends ParentException {
}
public class ExceptionSample {

    public static void test() throws ParentException {
        throw new SubException();
    }

    public static void main(String[] args) {
        try {
            ExceptionSample.test();
        } catch (SubException e) {
            System.out.println("Caught SubException exception");
        } catch (ParentException e) {
            System.out.println("Caught ParentException exception");
        }
    }
}
```

In the example above, we are declaring `ParentException` and its subclass `SubException`. `ExceptionSample` class has a method `test()` that declares to throw `ParentException`. Method `test()` implementation instead throws a more specific exception `SubException`. The `test()` method could choose to declare the fact that it is throwing both `ParentException` and `SubException`, but that is not required by the Java specifications.

The `test()` method caller can only catch `ParentException`. This takes care of catching `ParentException` and all `ParentException` subclasses. But if the `test()` method caller knows that `test()` method throws a more specific exception (a subclass of the `ParentException`), then the caller can choose to catch a more specific `SubException`, even though `test()` method does not explicitly declare the fact that it throws `SubException`.

This is an important point, as both the VistALink security modules and the VistALink resource adapter modules often throw more specific VistALink exceptions, even though those exceptions are not declared to be thrown and only parent exceptions are declared to be thrown from VistALink methods.

5.3. VistALink Exception Hierarchy



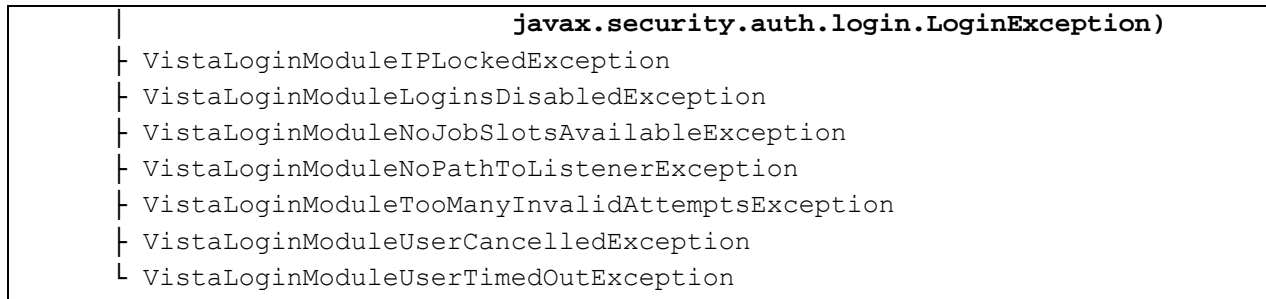


Figure 2. VistALink Exception Hierarchy

Because VistALink implements various Java specifications (e.g., Java Authentication and Authorization Service (JAAS) and J2EE Connectors), each Java specification dictates usage of a specific base exception class. To be able to work with all types of exceptions, VistALink defines one unifying exception interface: `gov.va.med.exception.FoundationExceptionInterface`.

Two methods are defined in this interface: `getFullStackTrace()` and `getNestedException()`. These methods are used by utility classes such as `gov.va.med.exception.ExceptionUtils` to retrieve nested exception information.

5.4. J2EE and J2SE Connectors Exceptions

J2EE Connectors requires adapter methods to throw `javax.resource.ResourceException`.

5.4.1. VistaLinkResourceException

VistALink implements `gov.va.med.vistalink.adapter.cci.VistaLinkResourceException` that extends `javax.resource.ResourceException`. `VistaLinkResourceException` is thrown from all J2EE Connectors required methods as well as any custom method in VistALink that implements connection management interfaces.

`VistaLinkResourceException` more specific exception subclasses include (see the [VistALink Exception Hierarchy](#) figure above):

- `ConnectionHandlesExceededException`
- `VistaLinkSocketClosedException`
- `HeartBeatFailedException`
- `HeartBeatInteractionFailedException`.

5.4.2. FoundationsException

The J2EE Connector Architecture specification defines optional record and interaction management interfaces that are not implemented in VistALink. Instead, VistALink uses `VistaLinkConnection::executeRPC()`, `VistaLinkConnection::executeInteraction()` and classes in `gov.va.med.vistalink.adapter.record` package to implement interaction and record management. These methods are not governed by the J2EE Connector Architecture

specification. Hence VistALink uses `gov.va.med.exceptionFOUNDATIONSException` and its subclasses in these methods.

5.4.3. VistaLinkFaultException

VistALink communications with M can produce exceptions that originate from the Java code side. In that case, `VistaLinkResourceException` and `FOUNDATIONSException` will be thrown (described above).

VistALink communications with M can also produce exceptions that originate from the M side. In those cases a Fault message is sent back to VistALink from M. Fault messages are parsed and `gov.va.med.vistalink.adapter.record.VistaLinkFaultException` that extends `FOUNDATIONSException` are constructed to be thrown in those cases.

`VistaLinkFaultException` will have all the information that is sent back from the M side to Java, including: `faultCode`, `faultString`, `faultActor`, `errorCode`, `errorType` and `errorMessage`.

`VistaLinkFaultException` more specific exception subclasses include (see the [VistALink Exception Hierarchy](#) figure above):

- `NoJobSlotsAvailableFaultException`
- `LoginsDisabledFaultException`
- `SecurityFaultException` (contain more specific subclasses)
- `RpcFaultException` (contain more specific subclasses)

5.4.4. Common FOUNDATIONSExceptionInterface

`gov.va.med.exceptionFOUNDATIONSExceptionInterface` is defined to be able to have common interface for all types of VistALink exceptions. Implementation of this interface allows `gov.va.med.exception.ExceptionUtils` to work exceptions, no matter what they inherit from.

5.4.5. Exception Nesting

Exception nesting is a technique used to collect full information about the error that occurred in the processing method call. Example:

Exception A can be thrown from a library. Client code catches the exception A and rethrows new exception B while preserving the original exception A within new exception's member variables.

This new exception B once again could be caught by some other client code that could throw new exception C while preserving caught exception B within exception's C member variables:

A nested within B nested within C.

This way, all caught and rethrown exceptions are kept as a linked exception list. This lets us keep all information about the error's origination and how the error is handled by the code.

JDK 1.4 has native support for exception nesting, while JDK 1.3 does not have native support for exception nesting. Since VistALink runs both on JDK 1.3 and 1.4, VistALink has implemented a custom exception nesting framework.

All VistALink base exception classes implement exception nesting:

- `gov.va.med.foundations.utilities.FoundationsException`
- `gov.va.med.foundations.adapter.cci.VistaLinkResourceException`
- `gov.va.med.foundations.security.vistalink.VistaLoginModuleException`

5.5. Working with Nested Exceptions

If your code catches one of the above exceptions, here are some features that can be expected:

`exception.getMessage()` returns all nested exception messages in the following format:

Wrapper exception;

Root cause exception:

`java.lang.Exception: Here is my nested exception.`

`exception.printStackTrace()` will print both nested exception messages and full stack trace:

```
gov.va.med.exception.FoundationsException:
```

Wrapper exception;

Root cause exception:

`java.lang.Exception: Here is my nested exception.`

```
java.lang.Exception: Here is my nested exception.
```

```
at
```

```
gov.va.med.vistalink.utilities.test.FoundationsExceptionTest.t2
(FoundationsExceptionTest.java:106)
```

```
at
```

```
gov.va.med.vistalink.utilities.test.FoundationsExceptionTest.t1
(FoundationsExceptionTest.java:109)
```

```
at
```

```
gov.va.med.vistalink.utilities.test.FoundationsExceptionTest.testConstructorStrExc(FoundationsExceptionTest.java:128)
```

```
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
```

```
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

```
at java.lang.reflect.Method.invoke(Unknown Source)
```

```
at junit.framework.TestCase.runTest(TestCase.java:154)
```

```
at junit.framework.TestCase.runBare(TestCase.java:127)
```

```
at junit.framework.TestResult$1.protect(TestResult.java:106)
```

```
at junit.framework.TestResult.runProtected(TestResult.java:124)
```

```
at junit.framework.TestResult.run(TestResult.java:109)
```

```

at junit.framework.TestCase.run(TestCase.java:118)
at junit.framework.TestSuite.runTest(TestSuite.java:208)
at junit.framework.TestSuite.run(TestSuite.java:203)
at junit.swingui.TestRunner$16.run(TestRunner.java:623)

```

`exception.getNestedException()` will return a nested exception. Note: do not use this method to unwind the nested exception linked list as there is a helper method in `ExceptionUtils.getNestedExceptionByClass()` that does just that.

See the section “ExceptionUtils” below for more methods that can be used when working with exceptions.

5.5.1. ExceptionUtils

`gov.va.med.exception.ExceptionUtils` is a utility class that contains static helper methods such as `getFullStackTrace()` and `getNestedExceptionByClass()` to help unwind the nested exception stack trace.

5.5.2. ExceptionUtils:: getFullStackTrace(Throwable e)

`ExceptionUtils:: getFullStackTrace(Throwable e)` returns the full stack trace, in case you need to print the stack trace for exception *e* to some other place other than the system output console, such as the HTML error page or the Swing text box. To print the stack trace to system output console just use `exception.printStackTrace()`.

Notice that we are using `Throwable` instead of `Exception` here, as this will allow us to print a stack trace even for `Errors`. The `getFullStackTrace` method prints out both the nested exception message set as well as the full stack trace. If you are using this method to return the full stack trace, this means the error message will be part of it and you don’t need to use `exception.getMessage()`.

5.5.3. ExceptionUtils:: getNestedExceptionByClass()

`ExceptionUtils:: getNestedExceptionByClass(Throwable e, Class exceptionClass)` returns an exception if an exception of a class type `exceptionClass` exists in the exception *e* nested exception linked list.

So if your nested exceptions look like this:

(a instance of A) nested within (b instance of B) nested within (c instance of C)

Then *a* calls:

```

ExceptionUtils:: getNestedExceptionByClass(c, C) returns c.
ExceptionUtils:: getNestedExceptionByClass(c, B) returns b.
ExceptionUtils:: getNestedExceptionByClass(c, A) returns a.

```


6. Foundations Library Utilities

The Java APIs discussed in this chapter are the Foundations Library utilities. They are provided in `vljFoundationsLib-1.5.0.jar`.

6.1. Encryption: *gov.va.med.crypto*

The `VistaKernelHash` utility class implements a two-way hash via two static methods, `encrypt` and `decrypt`. These provide the encoding and obfuscation algorithms used by the RPC Broker and Kernel to encode and decode data strings.

Using these algorithms makes it harder to sniff the contents of text sent over the network. This is not, however, encryption-class encoding, nor does it protect against replay attacks of un-decoded strings. As a two-way hash, this algorithm should not be considered an implication or achievement of any particular security level beyond obfuscation.

Example (encoding):

```
String encodedString =  
    VistaKernelHash.encrypt("some text to encode", true);
```

The second parameter is useful if the text is to be passed in a CDATA section of an XML message. If this parameter is set to true, the returned encoded strings will contain neither `"]]>` nor `"<![CDATA["`.

Otherwise, it is possible a returned encoding may contain those character sequences. If, in a reasonable number of tries, an encoded string cannot be created without these CDATA boundaries, an exception is thrown of type `VistaKernelHashCountLimitExceededException`.

Example (decoding):

```
String decodedString =  
    VistaKernelHash.decrypt(encodedString);
```

6.2. J2EE Environment: *gov.va.med.environment*

6.2.1. `Environment.isProduction()`

Returns whether or not the administrator has configured the J2EE server to be "production" in a VA-medical-center sense, i.e., the system is operating on production VA data. The source of the setting is the `gov.va.med.environment.production` JVM argument passed to the J2EE server upon startup. A setting of "true" designates the server as a production server; any other value (including not passing the JVM argument at all) marks the server as a non-VA production server.

Returns: true if the server is a VA production server, false if not.

6.2.2. Environment.getServerType()

Returns the J2EE server type. The source of the setting is the `gov.va.med.environment.serverType` JVM argument passed to the J2EE server upon startup. Defaults to return UNKNOWN if the JVM argument is not present.

Returns: ServerType: JBOSS | ORACLE | SUN_RI_13 | UNKNOWN | WEBLOGIC | WEBSPPHERE

6.3. Exception: gov.va.med.exception

For more information on the Exception utilities provided in the `gov.va.med.exception` package, see the [VistALink Exception Reference](#) section of this document.

6.4. Audit Timer: gov.va.med.monitor.time

VistALink uses `gov.va.med.monitor.time.AuditTimer` to capture and log information on the length of execution for VistALink interactions using `io4j`-logging capabilities. Special logger `gov.va.med.vistalink.adapter.spi.VistaSocketConnection.AuditLog` is used to output this information. Applications can use `AuditTimer` independently if they want to report timer information for various processing requests.

Two types of constructors can be used to construct the `AuditTimer` instance:

- For `public AuditTimer()`, default logger `gov.va.med.monitor.time.AuditTimer` is used.
- For `public AuditTimer(Logger logger)`, an application-specific logger is used.

`AuditTimer` logs milliseconds elapsed between `start()` and `stop()` calls. The number of elapsed milliseconds can be retrieved using `getTimeElapsedMillis()`.

Logging can be done using either `log()` method:

- `public void log()`
- `public void log(String message)`: Since the logger can be passed into the constructor and the output pattern for a specific logger can be configured using the `log4j` configuration file, there should be no need to pass info message. Instead, different loggers should be used.

6.4.1. Sample Code

```
import gov.va.med.monitor.time.AuditTimer;

public class AuditTimerTest {
    private static AuditTimer timer = null;
```

```

    private static Logger auditLogger =
Logger.getLogger(AuditTimerTest.class.getName() + ".AuditLog");

    public static void main(String[] args) {
        // Initialize Log4j configuration
        DOMConfigurator.configureAndWatch("props/log4jConfig.xml",
10000);

        timer = new AuditTimer(auditLogger);
        // Start timer
        timer.start();

        // ... perform your operations

        // Stop timer
        timer.stop();

        // Log elapsed time information
        timer.log();

        // Get time elapsed if not for logging purposes
        long timeElapsed = timer.getTimeElapsedMillis();
    }
}

```

The following is a snippet from the log4j configuration file:

```

<appender name="auditTimerTestConsoleAppender"
class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="Audit time - %m%n"/>
    </layout>
</appender>

<logger name="gov.va.med.vistalink.utilities.test.AuditTimerTest.AuditLog"
>
    <level value="debug" />
    <appender-ref ref="auditTimerTestConsoleAppender"/>
</logger

```

6.5. XML: gov.va.med.xml

6.5.1. XmlUtilities Class

This class contains a number of static utility methods to help developers work with XML documents, nodes, attributes and strings. These utilities are XML-parser independent.

The tables below describe the VistALink utility methods and variables.

Table 3. VistALink Utility Methods

Static Method Signature	Description
String convertXmlToStr(Document doc)	Converts a DOM document to a string
Document getDocumentForXmlString(String xml)	Returns an XML DOM Document for the specified String
Document getDocumentForXmlInputStream(InputStream xml)	Returns an XML DOM Document for the specified InputStream
Attr getAttr(Node node, String attrName)	Returns the Attribute with the given attrName at node
Node getNode(String xpathStr, Node node)	Returns the first node at the specified XPath location

Table 4. VistALink Utility Variables

Static Final Variables	Description
String XML_HEADER	Represents the default header used for all xml documents that communicate with an M server via VistALink. It is important to use this header as this keeps the client and M server in sync.

6.5.2. XMLUtilities Example

```
String xmlStr = XmlUtilities.XML_HEADER
    + "<VistaLink messageType='"
    + RpcRequest.GOV_VA_MED_RPC_REQUEST
    + "'
    + " mode='singleton'"
    + " version='1.0'"
    + " xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'"
    + " xsi:noNamespaceSchemaLocation='rpcRequest.xsd'"
    + " xmlns='http://med.va.gov/Foundations'"
    + ">"
    + " <RpcHandler version='1.0'/>"
    + " <Request rpcName='' rpcClientTimeOut='600' version='1.0' >"
    + "   <RpcContext></RpcContext>"
    + "   <Params><Param type='array' ></Param></Params>"
    + " </Request>"
    + "</VistaLink>";

Document doc = XmlUtilities.getDocumentForXmlString(xmlStr);

Node param = XmlUtilities.getNode("/VistaLink/Request/Params/Param",
requestDoc);

String type = XmlUtilities.getAttr(param, "type").getValue();
String xmlCopy = XmlUtilities.convertXmlToStr(xmlDoc);
...
```

```
FileInputStream xmlStream = FileInputStream("myRequest.xml");
Document myReq = XmlUtilities.getDocumentForXmlInputStream(xmlStream);
...
```

6.6. Network: *gov.va.med.net*

The following classes provide basic socket-based network functionality:

- **SocketManager**: Represents a socket that can be used to communicate with IP end points.
- **VistaSocketException**: Represents an exception thrown during read/write operations on a socket
- **VistaSocketTimeoutException**: Represents an exception identifying a timeout has occurred during read/write operations.

For more information on *gov.va.med.net* classes, see the Javadoc documentation provided in the VistALink distribution file.

7. Using VistALink with J2SE Applications

Using VistALink to build J2SE client/M server applications is very similar to using VistALink in J2EE. The main differences are in the areas of authentication and obtaining a connection. Sample J2SE applications are provided in the VistALink distribution file, in the **samples/J2SE** folder.

7.1. *Authenticating and Connecting to VistA in Client-Server Mode*

In J2EE, applications retrieve and return VistALink connections to a connection pool. In J2SE mode, however, VistALink establishes a direct, persistent connection on behalf of the J2SE application to the M server. Unless the server is shut down, this connection remains open until closed by the client.

The high-level steps to establish a VistALink connection to M in J2SE mode are:

1. Provide server configuration information to VistALink (IP address and port of the M VistALink listener to connect to)
2. Authenticate the end-user over the connection
3. Execute RPCs
4. Close the connection (log out)

VistALink uses the Java Authentication and Authorization Service (JAAS) framework for steps 1, 2 and 4 above. For information on step 3, see the section [“Executing Requests.”](#)

7.1.1. JAAS Overview

JAAS is a Java pluggable framework for user authentication and authorization. “Pluggability” means that different security modules (e.g., authentication modules) can be added or “plugged in” to an application without recompiling the application. VistALink uses the JAAS framework to authenticate end-users to an M/Kernel system, via the users' customary Kernel access and verify codes.

A JAAS-compliant login module contains all of the logic required to authenticate a user to a given system. The login module class does not itself, however, include the user interface to gather authentication credentials (e.g., access and verify codes) from the end-user. Instead, a set of JAAS-compliant callbacks, along with a JAAS-compliant callback handler, are used to decouple the user interface from the login module. VistALink provides a JAAS-compatible login module and JAAS-compliant callbacks and callback handlers to perform a VistA login.

Although the JAAS framework also provides authorization capabilities, VistALink uses JAAS for authentication only. VistALink does not make any use of the permission/authorization portions of the JAAS specification at this time.

7.2. VistALink JAAS Implementation

7.2.1. VistaLoginModule

VistALink provides a single JAAS-compliant login module class, `VistaLoginModule`. As a developer, you do not use this class directly because your application does the following:

- Specifies which login module to use, via a JAAS configuration file
- Creates a `LoginContext` instance, and passes it a supported callback handler instance to collect user input
- Invokes the login method of the `LoginContext` class to initiate the login process for the configured login module

7.2.2. JAAS Login Configuration Overview

By default, VistALink uses the default JAAS configuration reader to load login configurations. The default JAAS configuration reader class loads login configurations from a JAAS configuration file, which it expects to be in a predefined format.

One or more configuration entries are defined in the JAAS configuration file. The configuration file itself can have any name, and can be located anywhere. Each entry in the JAAS configuration file defines a particular login configuration. Generically, the format of this file is as follows:

```
ConfigurationName {
    ModuleClass Flag ModuleOptions;
};
ConfigurationName {
    ModuleClass Flag ModuleOptions;
};
```

7.2.3. VistALink-Specific JAAS Login Configuration

The following is an example of the JAAS configuration file format needed specifically for VistALink:

```
Test {
    gov.va.med.vistalink.security.vistalink.VistaLoginModule requisite
    gov.va.med.vistalink.security.vistalink.ServerAddressKey="10.21.185"
    gov.va.med.vistalink.security.vistalink.ServerPortKey="18010";
};
Production {
    gov.va.med.vistalink.security.vistalink.VistaLoginModule requisite
    gov.va.med.vistalink.security.vistalink.ServerAddressKey="10.21.1.85"
    gov.va.med.vistalink.security.vistalink.ServerPortKey="8005";
};
```


This example defines two login configurations, one named "Test" and one named "Production." An application uses this name (Test or Production) as the index to retrieve a particular configuration from the JAAS configuration file.

To configure a VistALink login to a Vista system, configure a single login module per login configuration entry, within each entry's {braces}, as follows:

1. Name the VistALink login module class, including package name:

```
gov.va.med.vistalink.security.vistalink.VistaLoginModule
```

2. Follow with a flag indicating what action to take if login fails. For VistALink, use `requisite`.
3. Follow with options for the VistALink login module. There are two options that must be set, in "name=value" format:

```
gov.va.med.vistalink.security.vistalink.ServerAddressKey
gov.va.med.vistalink.security.vistalink.ServerPortKey
```

Use quotes around the server address and server port values.

4. Before the closing brace, end with a semicolon.
5. Follow the closing brace with a semicolon.

For more information about the JAAS configuration file format expected by the default JAAS configuration file reader class, see:

<http://java.sun.com/j2se/1.4.1/docs/guide/security/jgss/tutorials/LoginConfigFile.html>

Note: It is possible to define your own JAAS configuration reader class, instead of using the default class. If you do this, you are still responsible for providing the package/name of the `VistaLoginModule` class, the JAAS `requisite` flag, and the two options required by the `VistaLoginModule`.

7.2.4. Passing the JAAS Login Configuration(s) to Your JVM

The JAAS Login Configuration needs to be passed to the JVM (and hence to your application). The JAAS configuration can be passed in two ways:

- With the `javax.security.auth.login.Configuration` Java virtual machine (JVM) argument when launching your application, e.g.:

```
java -Djava.security.auth.login.config=jaas.config MyApp
```

- In the Java security properties file.

In most cases it is preferable to use the JVM argument, since it allows the setting to be application-specific rather than machine-wide.

7.2.5. Selecting the JAAS Configuration From an Application

Once your application is running, it should select a specific configuration. In order to allow local administration of JAAS configuration files, you should, in most cases, provide a command-line parameter to allow local administrators to pass a particular JAAS configuration into your application. For example:

```
java -Djava.security.auth.login.config=jaas.conf MyApp Production
```

7.2.6. VistaLoginModule Callback Handlers

In order to decouple the user interface for logon from the login module, the JAAS standard allows login modules such as `VistaLoginModule` to supply different callback handlers. VistALink supplies two callback handler classes, one for an interactive logon, and one for non-interactive unit testing:

- `CallbackHandlerSwingCCOW`: for production application use. Performs single-signon if credentials are already present in CCOW user context. Otherwise, collects access code, verify code, division and "change verify code" input via a set of Swing dialogs; stores single sign-on credentials in CCOW user context for subsequent application use.
- `CallbackHandlerSwing`: for production application use. Collects access code, verify code, division and "change verify code" input via a set of Swing dialogs.
- `CallbackHandlerUnitTest`: for unit testing only (not production use). Access code, verify code, division are passed as parameters to the class constructor, resulting in a "silent" login suitable for (non-interactive) unit testing. The "change verify code" functionality is not supported.

Part of the JAAS VistALink login involves instantiating one of these two callback handler classes and passing the class as a parameter to create a JAAS login context (see "[Putting the Pieces Together](#)," below).

7.3. Putting the Pieces Together: VistALink JAAS Login

7.3.1. Logging in to VistA

The following is an example login. If application execution succeeds through the try block, the user has successfully logged in to the specified VistA listener.

```
// variable holding LoginContext should have application scope
```

```

// since it will be needed to log out, later on
private LoginContext loginContext = null;

try {

    // create the callback handler to use to collect user input
    // pass current Frame as parameter
    CallbackHandlerSwing cbhSwing = new CallbackHandlerSwing(myFrame);

    // create the LoginContext to control the login process;
    // pass the JAAS configuration to connect to, and the callback
    // handler (jaasConfigName value could be passed in from command
    // line)
    loginContext = new LoginContext(jaasConfigName, cbhSwing);

    // login to server through the LoginContext
    loginContext.login();

} catch (VistaLoginModuleException e) {

    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",
        JOptionPane.ERROR_MESSAGE);

} catch (LoginException e) {

    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",
        JOptionPane.ERROR_MESSAGE);

}

```

7.4. After Successfully Logging In

7.4.1. Retrieving the VistaKernelPrincipal

The JAAS subject is available from the JAAS LoginContext class after a successful login. It contains a JAAS principal (user entity), which holds:

- Demographic information about the logged-in user
- The authenticated VistALink connection object

The following code shows how to retrieve the Kernel principal after a successful login:

```

// variable holding Kernel principal may need application scope
// since it will be needed for RPC execution
private VistaKernelPrincipalImpl userPrincipal = null;

// . . . login code

// get the Kernel principal after logon
try {
    userPrincipal = VistaKernelPrincipalImpl.getKernelPrincipal(
        loginContext.getSubject());
} catch (FoundationsException e) {
    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",

```

```

    JOptionPane.ERROR_MESSAGE);
}

```

In the future, it is conceivable that *more than one principal* could be contained in the JAAS subject after login, if multiple login modules are used. This might happen when a compound login has been configured, requiring several logins to complete, e.g., one for a Kernel M system and one for a separate health data repository.

Only one *Kernel* principal should ever be returned, however. Use the `getKernelPrincipal` helper method in the `VistaKernelPrincipalImpl` class to retrieve the single Kernel principal.

7.4.2. Retrieving the Authenticated Connection From the Principal

To execute RPCs, you need to retrieve the authenticated connection. The authenticated connection object over which you make requests is stored in the Kernel principal, and can be retrieved with the `getAuthenticatedConnection` method.

Once a successful login has been completed, you should retrieve the associated authenticated connection from the Kernel principal. This connection is "logged in" to the M system under the end-user's identity. You can then use it to execute requests such as RPCs on behalf of the end-user.

For more information on executing requests, see the chapter of this manual titled "Executing Requests." An example of successful login appears below.

```

VistaLinkConnection myConnection =
    userPrincipal.getAuthenticatedConnection();

// . . . now you can execute requests

```

For information on how to use the `VistaLinkConnection` object to execute requests, see the section "[Executing RPCs](#)," below.

7.4.3. Retrieving User Demographic Information

Use the following predefined static KEY* strings to retrieve user demographic values via the Kernel principal's `getUserDemographicValue` method. For example:

```

// get the DUZ
String duz = this.userPrincipal.getUserDemographicValue(
    VistaKernelPrincipalImpl.KEY_DUZ);

// get the name
String name = userPrincipal.getUserDemographicValue(
    VistaKernelPrincipalImpl.KEY_NAME_DISPLAY);

```

The table below shows a complete set of returned demographics information and keys.

Table 5. Demographics Keys and Values

Key	Value
KEY_DIVISION_IEN	Login division station IEN
KEY_DIVISION_STATION_NAME	Login division station name
KEY_DIVISION_STATION_NUMBER	Login division station number
KEY_DTIME	User timeout value
KEY_DUZ	DUZ
KEY_LANGUAGE	User language
KEY_NAME_DEGREE	User degree
KEY_NAME_FAMILYLAST	Name component family-last
KEY_NAME_GIVENFIRST	Name component given-first
KEY_NAME_MIDDLE	Name component middle
KEY_NAME_NEWPERSON01	New Person .01 Field name
KEY_NAME_PREFIX	Name component prefix
KEY_NAME_SUFFIX	Name component suffix
KEY_SERVICE_SECTION	User service/section
KEY_NAME_DISPLAY	Concatenated standard name
KEY_TITLE	User title

7.4.4. Executing RPCs

Once you have a `VistaLinkConnection` connection object to work with, you can execute requests in exactly the same fashion as for VistALink's J2EE mode. For more information, see chapter 4 of this document, "[Executing Requests](#)." The entire chapter is valid for J2SE mode.

For information on timeouts for RPC execution, see the "Timeouts" section of this document titled "Using VistALink in J2EE." Most of the information there on timeouts is also valid for the J2SE mode.

7.4.5. Logging Out

Your application should *always* call the `logout` method of the JAAS `LoginContext` class to log out of VistA before exiting. This ensures that proper Kernel cleanup (e.g., of the `^TMP` global) occurs on the M server to which the user was connected.

7.4.5.1. Logging Out of Swing Applications

In a Swing application, the application should always call the `LoginContext`'s `logout` method when the application is shut down. There are a number of ways an application can be shut down:

The user closes the application window, the application is terminated from the Windows control panel, etc.

A good way to catch all of these shutdown cases is to implement a `WindowAdapter` as a window listener in the application, and provide an implementation of its `windowClosing` method that calls the `LoginContext`'s `logout` method.

For example:

```
// loginContext has been defined earlier, with application scope
// add event listener to log out when window closes
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        mylogout();
        System.exit(0);
    }
});

// Method called from event handler to perform logout
private void mylogout() {
    // Kernel logout
    if (this.userPrincipal != null) {
        try {
            loginContext.logout();
        } catch (LoginException e) {
            JOptionPane.showMessageDialog(null, e.getMessage(),
                "Logout error", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

7.5. Catching Login Exceptions

The `LoginContext` `login()` and `logout()` methods only throw exceptions that derive from `LoginException`. So at a minimum, when executing the login or logout methods of a `LoginContext` object, your application needs a try/catch block to catch `LoginException`.

7.5.1. VistaLoginModule Exception Hierarchy

JAAS requires `LoginModules` to throw `javax.security.auth.login.LoginException` from JAAS classes implementation methods. So the VistALink login module throws exceptions of type `gov.va.med.vistalink.security.vistalink.VistaLoginModuleException`, which extends `javax.security.auth.login.LoginException`.

The VistALink login module provides more granular exceptions from `VistaLoginModuleException`, so that your application can optionally filter exceptions at a finer level of granularity. This means that your application can detect and implement specific processing for login exception types that might be of interest.

Table 6. VistALink Login Exceptions

Exception	Description
VistaLoginModuleException	Like a LoginException, but may contain nested exception(s) that were the cause for the LoginException.
VistaLoginModuleLoginsDisabledException	Logins are disabled on the M server.
VistaLoginModuleNoJobSlotsAvailableException	Job slot maximum has been exceeded on M server.
VistaLoginModuleNoPathToListenerException	No reachable listener was found on the path represented by the specified IP address and Port.
VistaLoginModuleTooManyInvalidAttemptsException	The user tried to login too many times with invalid credentials.
VistaLoginModuleUserCancelledException	The user cancelled the login.
VistaLoginModuleUserTimedOutException	The user timed out of the login.

For example, if your application is interested in whether the IP and port specified were "bad" (at least at the time the login was attempted), you can trap for the `VistaLoginModuleNoPathToListener` exception, in addition to the standard `LoginException`. **Example:**

```
try {
    // create the callback handler to use to collect user input
    CallbackHandlerSwing cbhSwing = new CallbackHandlerSwing(myFrame);

    // create the LoginContext to control the login process.
    loginContext = new LoginContext(serverAlias, cbhSwing);

    // login to server through the LoginContext
    loginContext.login();
} catch (VistaLoginModuleLoginsDisabledException e) {
    JOptionPane.showMessageDialog(
        null,
        "Logins are disabled; try later.",
        "Login error",
        JOptionPane.ERROR_MESSAGE);
} catch (LoginException e) {
    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",
        JOptionPane.ERROR_MESSAGE);
}
```

7.6. Unit Testing and VistALink

The login user interface collects end-user information, including the access and verify code and division number. It is separate from the logic that implements login, because of the pluggable JAAS architecture. The user interface is contained in a JAAS-compliant set of callbacks, and the login logic is contained in a JAAS-compliant login module. Therefore, the JAAS framework makes it straightforward to implement alternative user interfaces for login.

VistALink provides an alternative callback handler that implements a "silent" (non-interactive) login suitable for unit testing purposes. **This silent login is not suitable for any production environment.** Your application passes the access and verify code, and (optionally) division to silently log in your application. Changing the verify code is not supported with this callback handler.

For example:

```
// Connection info
String cfgName = "Production";

// signon credentials for unit test callback handler
String accessCode = "asdf.123";
String verifyCode = "asdf.456";
String division = "";

try {

    // create the "unit test" callbackhandler for JAAS login
    CallbackHandlerUnitTest cbhUnitTest =
        new CallbackHandlerUnitTest(accessCode, verifyCode, division);

    // create the JAAS LoginContext for login
    lc = new LoginContext(cfgName, cbhUnitTest);

    // login to server
    lc.login();

} catch (VistaLoginModuleException e) {

    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",
        JOptionPane.ERROR_MESSAGE);

} catch (LoginException e) {

    JOptionPane.showMessageDialog(null, e.getMessage(), "Login error",
        JOptionPane.ERROR_MESSAGE);

}
```


Glossary

Access Code	A password used by the Kernel system to identify the user. It is used with the verify code.
Adapter	Another term for <i>resource adapter</i> or <i>connector</i> .
Administration Server	Each BEA WebLogic server domain must have one server instance that acts as the administration server. This server is used to configure all other server instances in the domain.
Alias	An alternative filename.
Anonymous Software Directories	M directories where VHA application and patch zip files are placed for distribution.
Application Proxy User	A Kernel user account designed for use by an application rather than an end-user.
Application Server	Software/hardware for handling complex interactions between users, business logic, and databases in transaction-based, multi-tier applications. Application servers, also known as app servers, provide increased availability and higher performance.
Authentication	Verifying the identity of the end-user.
Authorization	Granting or denying user access or permission to perform a function.
Base Adapter	Version 8.1 of WebLogic introduced a "link-ref" mechanism enabling the resources of a single "base" adapter to be shared by one or more "linked" adapters. The base adapter is a standalone adapter that is completely set up. Its resources (classes, jars, etc.) can be linked to and reused by other resource adapters (linked adapters). The deployer only needs to modify a subset of the linked adapters' deployment descriptor settings.
Caché	Caché is an M environment, a product of InterSystems Corp.
CCOW	A standard defining the use of a technique called "context management," providing the clinician with a unified view on information held in separate and disparate healthcare applications that refer to the same patient, encounter or user.
Classpath	The path searched by the JVM for class definitions. The class path may be set by a command-line argument to the JVM or via an environment variable.
Client	Can refer to both the client workstation and the client portion of the program running on the workstation.
Connector	A system-level driver that integrates J2EE application servers with Enterprise Information Systems (EIS). VistALink is a J2EE connector module designed to connect to Java applications with VistA/M systems. The term is used interchangeably with <i>connector module</i> , <i>adapter</i> , <i>adapter module</i> , and <i>resource adapter</i> .
Connection Factory	A J2CA class for creating connections on request.
Connection Pool	A cached store of connection objects that can be available on demand and reused, increasing performance and scalability. VistALink 1.5 uses connection pooling.
Connector Proxy User	For security purposes, each instance of a J2EE connector must be granted access to the M server it connects to. This is done via a Kernel user account set up on the M system. This provides initial authentication for the app server and establishes a trusted connection. The M system manager must set up the connector user account and communicate the access code, verify code and

	listener IP address and port to the J2EE system manager.
DCL	<i>Digital Command Language</i> . An interactive command and scripting language for VMS.
Division	VHA sites are also called <i>institutions</i> . Each institution has a <i>station number</i> associated with it. Occasionally a single institution is made up of multiple sites, known as <i>divisions</i> . To make a connection, VistALink needs a station number from the end-user's New Person entry in the Kernel Site Parameters file. It looks first for a division station number and if it can't find one, uses the station number associated with default institution.
DSM	<i>Digital Standard MUMPS</i> . An M environment, a product of InterSystems Corp.
DUZ	A local variable holding a number that identifies the signed-on user. The number is the Internal Entry Number (IEN) of the user's record in the NEW PERSON file (file #200)
EAR file	<i>Enterprise archive</i> file. An enterprise application archive file that contains a J2EE application.
File #18	System file #18 was the precursor to the KERNEL SYSTEMS PARAMETERS file, and is now obsolete. It uses the same number space that is now assigned to VistALink. Therefore, file #18 must be deleted before VistALink can be installed.
Global	A multi-dimensional data storage structure -- the mechanism for persistent data storage in a MUMPS database.
HealthVet-VistA	The VHA is converting its MUMPS-based VistA healthcare system to a new J2EE-based platform and application suite. The new system is known as HealthVet-VistA.
IDE	<i>Integrated development environment</i> . A suite of software tools to support writing software.
Institution	VHA sites are also called <i>institutions</i> . Each institution has a <i>station number</i> associated with it. Occasionally a single institution is made up of multiple sites, known as <i>divisions</i> . To make a connection, VistALink needs a station number from the end-user's New Person entry in the Kernel Site Parameters file. It looks first for a division station number and if it can't find one, uses the station number associated with default institution.
Institution Mapping	The VistALink 1.5 release includes a small utility that administrators can use to associate station numbers with JNDI names, and which allows runtime code to retrieve the a VistALink connection factory based on station number.
J2CA	<i>J2EE Connector Architecture</i> . J2CA is a framework for integrating J2EE-compliant application servers with Enterprise Information Systems, such as the VHA's VistA/M systems. It is the framework for J2EE connector modules that plug into J2EE application servers, such as the VistALink adapter.
J2EE	<i>Java 2 Enterprise Edition</i> . A standard suite of technologies for developing distributed, multi-tier, enterprise applications.
J2SE	<i>Java 2 Standard Edition</i> . Sun Microsystem's programming platform based on the Java programming language. It is the blueprint for building Java applications, and includes the Java Development Kit (JDK) and Java Runtime Environment (JRE).
JAAS	<i>Java Authentication and Authorization Service</i> . JAAS is a pluggable Java framework for user authentication and authorization, enabling services to authenticate and enforce access controls upon users.
JAR file	Java archive file. It is a file format based on the ZIP file format,

	used to aggregate many files into one.
Java Library	A library of Java classes usually distributed in JAR format.
Javadoc	Javadoc is a tool for generating API documentation in HTML format from doc comments in source code. Documentation produced with this tool is typically called Javadoc.
JDK	<i>Java Development Kit</i> . A set of programming tools for developing Java applications.
JNDI	<i>Java Naming and Directory Interface</i> . A protocol to a set of APIs for multiple naming and directory services.
JRE	The <i>Java Runtime Environment</i> consists of the Java virtual machine, the Java platform core classes, and supporting files. JRE is bundled with the JDK but also available packaged separately.
JSP	<i>Java Server Pages</i> . A language for building web interfaces for interacting with web applications.
JVM	<i>Java Virtual Machine</i> . The JVM interprets compiled Java binary code (byte code) for specific computer hardware.
Kernel	Kernel functions as an intermediary between the host M operating system and VistA M applications. It consists of a standard user and program interface and a set of utilities for performing basic VA computer system tasks, e.g., Menu Manager, Task Manager, Device Handler, and security.
KIDS	<i>Kernel Installation and Distribution System</i> . The VistA/M module for exporting new VistA software packages.
LDAP	Acronym for Lightweight Directory Access Protocol. LDAP is an open protocol that permits applications running on various platforms to access information from directories hosted by any type of server.
Linked Adapter	Version 8.1 of WebLogic introduced a “link-ref” mechanism enabling the resources of a single “base” adapter to be shared by one or more “linked” adapters. The base adapter is a standalone adapter that is completely set up. Its resources (classes, jars, etc.) can be linked to and reused by other resource adapters (linked adapters). The deployer only needs to modify a subset of linked adapters’ deployment descriptor settings.
Linux	An open-source operating system that runs on various types of hardware platforms. HealthVet-VistA servers use both Linux and Windows operating systems.
Listener	A socket routine that runs continuously at a specified port to field incoming requests. It sends requests to a front controller for processing. The controller returns its response to the client through the same port. The listener creates a separate thread for each request, so it can accept and forward requests from multiple clients concurrently.
log4J Utility	An open-source logging package distributed under the Apache Software license. Reviewing log files produced at runtime can be helpful in debugging and troubleshooting.
logger	In log4j, a logger is a named entry in a hierarchy of loggers. The names in the hierarchy typically follow Java package naming conventions. Application code can select a particular logger by name to write output to, and administrators can configure where a particular named logger’s output is sent.
M (MUMPS)	<i>Massachusetts General Hospital Utility Multi-programming System</i> , abbreviated M. M is a high-level procedural programming computer language, especially helpful for manipulating textual data.
Managed Server	A server instance in a BEA WebLogic domain that is not an

	administration server, i.e., not used to configure all other server instances in the domain.
Messaging	A framework for one application to asynchronously deliver data to another application, typically using a queuing mechanism.
Multiple	A VA FileMan data type that allows more than one value for a single entry.
Namespace	A unique 2-4 character prefix for each VistA package. The DBA assigns this character string for developers to use in naming a package's routines, options, and other elements. The namespace includes a <i>number space</i> , a pre-defined range of numbers that package files must stay within.
New Person File	The New Person file contains information for all valid users on an M system.
Patch	An update to a VistA software package that contains an enhancement or bug fix. Patches can include code updates, documentation updates, and information updates. Patches are applied to the programs on M systems by IRM services.
Plug-in	A component that can interact with or be added to an application without recompiling the application.
ra.xml	ra.xml is the standard J2EE deployment descriptor for J2CA connectors. It describes connector-related attributes and its deployment properties using a standard DTD (Document Type Definition) from Sun.
Re-authentication	When using a J2CA connector, the process of switching the security context of the connector from the original application connector "user" to the actual end-user. This is done by the calling application supplying a proper set of user credentials.
Resource Adapter	J2EE resource adapter modules are system-level drivers that integrate J2EE application servers with Enterprise Information Systems (EIS). This term is used interchangeably with <i>resource adapter</i> and <i>connector</i> .
Routine	A program or sequence of computer instructions that may have some general or frequent use. M routines are groups of program lines that are saved, loaded, and called as a single unit with a specific name.
RPC	<i>Remote Procedure Call</i> . A defined call to M code that runs on an M server. A client application, through the RPC Broker, can make a call to the M server and execute an RPC on the M server. Through this mechanism a client application can send data to an M server, execute code on an M server, or retrieve data from an M server
RPC Broker	The RPC Broker is a client/server system within VistA. It establishes a common and consistent framework for client-server applications to communicate and exchange data with VistA/M servers.
RPC Security	All RPCs are secured with an RPC context (a "B"-type option). An end-user executing an RPC must have the "B"-type option associated with the RPC in the user's menu tree. Otherwise an exception is thrown.
Servlet	A Java program that resides on a server and executes requests from client web pages.
Socket	An operating system object that connects application requests to network protocols.
Verify Code	A password used in tandem with the access code to provide secure user access. The Kernel's Sign-on/Security system uses the verify code to validate the user's identity.

Vista	<i>Veterans Health Information Systems and Technology Architecture</i> . The VHA's portfolio of M-based application software used by all VA medical centers and associated facilities.
VistALink Libraries	Classes written specifically for VistALink.
VMS	<i>Virtual Memory System</i> . An operating system, originally designed by DEC (now owned by Hewlett-Packard), that operates on the VAX and Alpha architectures.
VPID	<i>VA Person Identifier</i> . A new enterprise-level identifier uniquely identifying VA 'persons' across the entire VA domain.
WAR file	<i>Web archive</i> file. Contains the class files for servlets and JSPs.
WebLogic Server	A J2EE application server manufactured by BEA WebLogic Systems.
XOB Namespace	The VistALink namespace. All VistALink programs and their elements begin with the characters "XOB."

