



CISCO CONFIDENTIAL

CHAPTER **31**

Using the Common Services Transport Mechanism

The Common Services Transport Mechanism (CSTM) provides a single, consistent, simple and platform-agnostic method for handling all types of programmatic communications, including:

- Inter-Process Communication (IPC): Multiple processes on the same machine.
- Remote Procedure Calls (RPC): Multiple processes on different machines.
- In-Process Calls: Processes executing under the same virtual machine.

The following topics describe CSTM and how to use it in your applications:

- [Understanding CSTM](#)
- [Installing CSTM](#)
- [Controlling CSTM Logging](#)
- [Publishing Objects](#)
- [Accessing Published Objects](#)
- [Handling Special Requirements](#)
- [Using the CTMTest Tools and Samples](#)
- [Guidelines for Using CSTM](#)

For more information about CSTM, see

- The CSTM Functional Specification, ENG-124878.
- CSTM: Software Unit Design Specification: ENG-155861.
- CSTM User guide: ENG 161448.

Understanding CSTM

In the past, the way to create rich application-to-application communication has been to employ DCOM, CORBA, or RMI. These rich environments typically require that applications use the same object model at both ends of a connection, which is often impractical. They also assume that both sender and the receiver have full knowledge of the message context and do not encode meta-information. This gives good performance, but makes it hard for intermediaries to process messages. Finally, each system uses a different binary encoding, making it hard to build systems that interoperate.

CISCO CONFIDENTIAL

CSTM provides a way to handle communications by abstracting them, providing the same API for all of them, and founding the API on well-know non-proprietary standards.

In CSTM, communications across machines are handled using XML and Serialized Java Objects over HTTP transport. Communications on the same machine are handled with Sockets. During inter-thread communication, CSTM plays a part during initial reference resolutions only.

Even though CSTM encapsulates the use of XML for message transfer, it also offers compatibility with a well-defined XML message protocol (SOAP). Use of XML RPC facilitates means there is no need to care about what operating system, programming language or object model is being used on either the client or server side.


Installing CSTM

CSTM supports a wide variety of communications involving one or more machines. Depending on how you plan to implement it, you will want to install it using the procedures in one of the following sections:

- [Installing Basic CSTM, page 31-2](#)
- [Installing CSTM with the Tomcat Servlet Engine, page 31-3](#)

Installing Basic CSTM

This is the basic installation for a single machine without a Java servlet engine. To install CSTM on a single machine:

-
- Step 1** Before installing CSTM:
- If you have a previous version of CSTM installed, delete any old ctmregistry and ctmregistry.backup files. These are normally found in the same directory as the CTM.jar file.
 - Ensure that you have JDK 1.3.1 or later installed.
 - CSTM is supplied on the CWCS SDK disk as a WAR file. If needed, you can download CTM 1.0 and 1.1. CTM 1.0 is available at the following URL:
http://www.in-nmbu/auto/cw/cdimages/ctm1_0/daily/NT_CTM1_0_INTEGRATION_READY/kits/. CTM 1.1 is available at
http://www.in-nmbu/auto/cw/cdimages/ctm1_1/daily/NT_CTM1_1_INTEGRATION_READY/kits/. The same CSTM WAR file can be used for both Solaris and Windows. Extract the WAR file into a suitable location.
- Step 2** Add CTM.jar and log4j.jar to the classpath.
- Step 3** Save the ctm_static_registry.txt and ctm_config.txt files in the same directory as the CTM.jar file.
-
-  **Note** As soon as CSTM begins running, it will create ctmregistry and ctmregistry.backup files in the same directory as the CTM.jar file. Do not tamper with these files while running CSTM.
-
- Step 4** To run the CSTM samples, add samples.jar to the classpath. The source code for all samples is in samples_source.jar, which is available from the CSTM portal at <http://embu-web.cisco.com/eng/teams/Mjollnir/> (follow the CTM or CSTM link under “Subprojects”).
-

CISCO CONFIDENTIAL

Installing CSTM with the Tomcat Servlet Engine

For full functionality, you will want to install CSTM on one or more machines with a servlet engine. CWCS supports the Tomcat servlet engine by default. No other servlet engine is supplied with CWCS, and the CWCS team recommends that you use Tomcat exclusively.

To install CSTM with a servlet engine, perform the basic CSTM install on each machine (using the instructions in the [“Installing Basic CSTM” section on page 31-2](#)), then ensure that:

- The servlet engine is installed and running on each machine. This is necessary to handle remote calls.
- You have updated the CTMServlet parameter in the CSTM configuration file (ctm_config.txt) with the URL for the servlet engine you have installed on that machine.

**Note**

To make CSTM work with any servlet engine, you *must* modify the `CTM_URL` parameter in the `ctm_config.txt` file to point to the appropriate CTMServlet URL.

To install CSTM using the Tomcat servlet engine, follow the steps below.

- Step 1** Make sure you have Tomcat 3.2.1 or higher installed.
- Step 2** In the Tomcat installation’s conf directory, in the server.xml file, under the heading “Special webapps” and after the Context path for examples, add the following lines: f

```
<Context path="/ctm"
docBase="drive:/ctm"
debug="1"
reloadable="true">
</Context>
```

(Where `drive:/ctm` is the directory where CSTM is currently installed).

- Step 3** In your CSTM directory, create a WEB-INF directory.
- Step 4** Copy the web.xml file in the CSTM distribution into the WEB-INF directory.

**Note**

The web.xml file has entries to start off TestServlet, which exposes the class TestClass using the Unique Resource Name CheckServlet. This TestServlet class file is included in samples.jar.

- Step 5** In the WEB-INF directory, create a lib directory and place CTM.jar in it. If your application needs any other jars, place them in this directory.

Controlling CSTM Logging

CSTM uses log4j to handle log messages. You can adjust CSTM log4j logging as explained in the following topics:

- [Setting Up CSTM Logging](#)
- [Viewing the CSTM Log File](#)

CISCO CONFIDENTIAL

**Note**

CSTM uses log4j for logging, but does not set up or initialize the log4j framework during runtime. If your product already ships with a log4j configuration file, you must either add CSTM-related log4j categories to it or add them to log4j.properties and add the location of log4j.properties to your application's classpath. All entries in the log4j.properties file that ships with CSTM are commented out by default.

Setting Up CSTM Logging

By default, a running instance of CSTM on a single machine logs all messages of FATAL severity to the console. If you have a running log4j server, it will log these messages to the ctm.log in the log4j server directory.

To set up log4j logging and ensure that it works as desired, you must:

- In log4j.properties: Set the logging levels as needed.
- In log4j.properties: Set the log destination you want.
- Include the log4j.jar and log4j.properties file directory in the classpath, and start a log4j server.

The following topics explain how to perform these tasks:

- [Setting the CSTM Logging Levels](#)
- [Changing the CSTM Logging Destination](#)
- [Starting a Log4j Server](#)

**Note**

CSTM does not explicitly load the log4j.properties file. If your product is already using log4j, you may not need to have a separate log4j.properties file for CSTM. Instead, you can add the CSTM logging levels and appenders to your existing log4j properties file.

Setting the CSTM Logging Levels

Log4j message logging levels are set using log4j.category entries in the log4j.properties file.

The default logging level is FATAL, and is set by the log4j.rootCategory=FATAL entry. The other logging levels are DEBUG, INFO, WARN and ERROR.

You can change the level of logging for CSTM or any of the CSTM modules using log4j.category entries in the log4j.properties file. For example: You could set explicit category entries for individual CSTM modules as follows:

```
log4j.category.CTM.server=DEBUG
log4j.category.CTM.client=INFO
log4j.category.CTM.registry=WARN
```

All CSTM modules inherit and use the log4j.rootCategory setting unless there is a log4j.category set for them.

Changing the CSTM Logging Destination

In log4j, server message destinations are called appenders. Appenders are created using log4j.appender entries in the log4j.properties file. Any appender can be assigned to a logging levels using the logging level's log4j.category entries in the same file.

CISCO CONFIDENTIAL

In the default log4j.properties file, only the console appender is defined, and the log4j.rootCategory that sets the default logging level is set to the console appender. If you are running a log4j server, all messages will be sent first to the server, then to the ctm.log file (if this log4j server is not running, the messages will only be logged to the console).

It is possible to define appenders for your own purposes. For example: On a single machine, you may have multiple virtual machines making CSTM calls. To have all CSTM log messages coming from all the VMs logged to a single ctm.log file, you would update the log4j.properties file with necessary appenders to send the messages to the log4j server and then run a log4j server. Every VM will then send its log messages to the log4j server, which in turn logs it into the ctm file.

To define and use appenders, you must:

- Update the log4j.properties file with appenders to which you want messages to be sent.
- Run the log4j server with the new appenders.

For example: To create an appender named “A2”, add the following socket appender configurations to the log4j.properties file:

```
log4j.appender.A2=org.apache.log4j.net.SocketAppender
log4j.appender.A2.RemoteHost=localhost
log4j.appender.A2.Port=8888
```

Then, to have your custom logging levels all send messages to this appender, you would update your log4j.category entries as follows:

```
log4j.category.CTM.server=DEBUG, A2
log4j.category.CTM.client=INFO, A2
log4j.category.CTM.registry=WARN, A2
```

Starting a Log4j Server

Once categories and appenders are defined, you can start the log4j server by adding log4j.jar to the classpath and then issuing the following command:

```
>> org.apache.log4j.net.SocketServer port proppath propdir
```

Where:

- *port* is the port on which the server is to run.
- *path* is the complete path and file name for the log4j.server.properties file.
- *dir* is the directory in which log4j.server.properties file is stored.

For example: if log4j.server.properties is stored in D:\ctm, and the log4j.jar and log4j.properties files are also stored there, you would start the server with the following commands:

```
>> set classpath = d:\ctm\log4j.jar;d:\ctm
>> org.apache.log4j.net.SocketServer 8888 d:\ctm\log4j.server.properties d:\ctm
```



Note

The ctm.log file will be created in the same directory from which this command was executed.

Viewing the CSTM Log File

You can view the ctm.log file with any ASCII text editor. For each message, the log will include the:

- Date: The message date in dd/mm/yyyy format

CISCO CONFIDENTIAL

- Time: The message time in hh:mm:ss:msec format.
- Logging level: INFO, DEBUG, WARN, ERROR, or FATAL
- Classname: The complete name of the class that output the message.
- Methodname: The method name of the calling method.
- Message: The text of the message logged from the calling method.

For example:

```
31/Jan/2002 17:52:31:984 DEBUG com.cisco.nm.xmls.ctm.server.CTMServer <clinit> -
Serverport:40000 MinThreads:3 maxThreads:20
```

For message sizes larger than 32MB, you must set the startup memory to a higher value. To do this, compile using the javac option for startup memory. For example, compile using `>> Javac -J-Xms48m` to set startup memory to 48MB, or use `>> Javac -J-Xms64m` to set it to 64MB.

Publishing Objects

When you publish an object, you expose it to other processes. The CSTM files and CSTM Server API provide several means for publishing and unpublishing objects. The following topics explain these methods and the guidelines you should follow when:

- [Publishing Objects Statically, page 31-6](#)
- [Publishing Objects Dynamically, page 31-7](#)
- [Handling Remote Objects, page 31-7](#)
- [Publishing Objects Securely, page 31-8](#)
- [Unpublishing Objects, page 31-9](#)

Publishing Objects Statically

CSTM allows you to register classes statically by adding them to the file `ctm_static_registry.txt`. Entries in the static registry file take the form:

```
urn=classname,
```

where:

- `urn` is the Unique Resource Name of the object
- `classname` is the object's classname.

Static registration using `ctm_static_registry.txt` always takes precedence over dynamic registration done using a Publish call (see [“Publishing Objects Dynamically” section on page 31-7](#)). For example: If `ctm_static_registry.txt` contains the entry `abc=TestClass`, and you later try to publish a resource with the URN `abc` and classname `TestClass`, CSTM will report that this resource is already registered. If a class is registered statically, then CSTM will read and use the static registry entry whenever your client invokes that object.

To use static registration successfully, always ensure that:

- The `ctm_static_registry.txt` file is stored in the same directory as `CTM.jar`.
- All classnames appearing in `ctm_static_registry.txt` also appear in the Classpath.

CISCO CONFIDENTIAL

Publishing Objects Dynamically

CSTM provides methods to publish objects dynamically, at runtime. Objects published dynamically are available via all the types of communication CSTM supports, including In-Process Calls, IPC, and RPC.

Ideally, the published object should expose its functionality via an interface. If the exposed object provides an interface, the CSTM client can invoke the exposed object either by binding to this interface using CTMClientProxy, or as a generic CTMClient. If the exposed object does not provide an interface, you can use only CTMClient. Note that the CTMClientProxy feature is available only if you are using JDK 1.3.1 or later.

To publish an object dynamically, use the CTMServer.publish method:

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(java.lang.String urn, java.lang.Object
object, com.cisco.nm.xml.ctm.common.CTMServerProperties properties) throws CTMException
```

CTMServer.publish takes the following input arguments:

<i>urn</i>	A Unique Resource Name for the object you want to publish. URNs need only be unique for the management application area or server on which they are used.
<i>object</i>	The published object's classname.
<i>properties</i>	Additional properties for the published object.

For example: To publish a single reference of class `TestClass`, with the URN `xyz` (in this case the same object will be used, irrespective of the type of client; this functionality is not available with static registration):

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(xyz, new TestClass())
```

To publish the same class with the same URN, by passing the class definition (in this case the functionality varies based upon the type of client):

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(xyz, TestClass.class)
```

Handling Remote Objects

CTMServer provides methods to register and publish remote objects, as follows:

- [Registering Remote Objects Statically](#)
- [Registering Remote Objects Dynamically](#)
- [Publishing Remote Objects](#)

Registering Remote Objects Statically

Static registration can be done by adding an entry in the static CSTM registry file (`ctm_static_registry.txt`).

```
Urn name = class name
```

CISCO CONFIDENTIAL**Example**

```
abc=com.cisco.cmf.test.TestServer
```

Since this kind of registration is done at compile time, it does not need any API.

Registering Remote Objects Dynamically

The CTMServer class provides methods to publish the object dynamically at runtime. The published object is available through all types of communication such as external applications (Non-Appliance), Inter-Appliance, Intra-Appliance and Intra-JVM. If there are errors, CTMServer throws CTMException.

If the exposed object implements an interface, it can be used by the CSTM Client as a remote interface. CSTM client can invoke the exposed object by binding to this interface using CTMClientProxy or by using generic CTMClient, which does not use this remote interface. But generally, it is recommended to implement the interface since it allows the CSTM clients to use CTMClientProxy.

Usage

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(
    java.lang.String urn,
    java.lang.Object object,
    com.cisco.nm.xml.ctm.common.CTMServerProperties properties)
```

where urn is any string, including spaces or even an empty string as "".

Publishing Remote Objects

You can publish a remote object in two ways:

- Passing a single reference

To publish class `TestClass`, with URN name `xyz`, using the same object regardless of type of client:

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(xyz,
    new TestClass())
```

- Passing the class definition

To publish class `TestClass` with URN `xyz`, by passing the class definition.

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(xyz,
    TestClass.class)
```

In this case, the functionality varies based upon type of client.

Publishing Objects Securely

CSTM allows you to publish objects securely. CSTM uses CWCS security to secure the access to the published object.

The CTMServerProperties is passed in the Publish call to set the security options. This means that any remote object published securely with a set of user authorized roles, can only be accessed by an authorized user, whose role/permissions are then authenticated against the ones set while publishing.

When the CTMServerProperties is not given as a parameter to the publish method, then by default, the security option is treated as false, that is, no security checks are performed on the clients attempting to access this resource.

CISCO CONFIDENTIAL

The security option makes most sense in the context of calls coming from outside the box. In such cases, the publisher might intend to secure the access to the published resource. Hence, security checks are performed only for the calls coming from outside the box. Security checks are not performed for the calls within the same box.

However, in the CTMServerProperties option, whenever the security option is set to true, it must have a list of roles that are authorized to access this URN.

The CTMServerProperties for a published URN (Unique Resource Name) can be specified with a specific security value, that is, security value of True or False depending on whether security needs to be turned ON or OFF. Also, if security is turned On, the string [] of roles gives the list of roles that are authorized to securely access this URN. When no arguments are provided to the CTMServerproperties call, then the default security value is set to "False" and the string [] of roles is set to null.

Usage

```
public CTMServerProperties()
public CTMServerProperties(boolean security,
String [] roles)
```

Input Arguments

security A boolean value that indicates if security is true/false

roles A string array that contains the names of roles authorized to access the published resource/URN

Example

To publish a class called `TestClass`, with security options turned on:

1. Set the permissions allowable/authorized roles

```
String [] roles = {"CsAuthServlet.SA",
"CSAuthServlet.NA"};
```

2. Set these permissions in the CTMServerProperties

```
com.cisco.nm.xml.ctm.common.CTMServerProperties props =
new com.cisco.nm.xml.ctm.common.CTMServerProperties(true,
roles);
```

3. Publish `TestClass`, with security options

```
com.cisco.nm.xml.ctm.server.CTMServer.publish(xyz,
TestClass.class,
props);
```

To publish this URN with security option turned off:

```
com.cisco.nm.xml.ctm.server.publish("abc",
TestClass.class)
```

Unpublishing Objects

Unpublishing an object cancels exposure of that object and its functionality via CSTM. Note that, when you unpublish a remote object, you can re-use its URN.

CISCO CONFIDENTIAL**Note**

If your application publishes a URN through CTM Server, make sure you unpublish the URN when your application no longer requires it to be maintained or exposed. If your application uses CWCS, this means you must (at a minimum) unpublish the URN during a Daemon Manager “stop” notification.

To unpublish an object, use the `CTMServer.unpublish` method:

```
com.cisco.nm.xms.ctm.server.CTMServer.unpublish(java.lang.String.urn) Throws CTMException
```

`CTMServer.unpublish` has only one input argument, *urn*, which is the Unique Resource Name of the object to be unpublished.

For example:

To unpublish the URN *xyz*:

```
com.cisco.nm.xms.ctm.server.CTMServer.unpublish(xyz)
```

Accessing Published Objects

CSTM provides three ways to invoke the server side functionality from a client:

- **CTMClient**: Uses the URN of the remote object and a static method of a class which takes the method signature. For details, see the [“Using CTMClient” section on page 31-10](#).
- **CTMClientProxy**: Obtains a proxy for the corresponding remote object. For details, see the [“Using CTMClientProxy” section on page 31-11](#).
- **CTMCall**: Instantiates a call that maintains a connection with the remote object. **CSTMCall** provides static methods for binding and invoking remote object methods. For details, see the [“Using CTMCall” section on page 31-13](#).

Developers should choose the method that best suits their application’s needs.

All three of these methods are affected by or make use of:

- **CTMServer.properties**, which sets variables that control remote sessions. For details, see the [“Changing CTM Client Properties” section on page 31-14](#).
- **CTMConstants**, which sets the encoding style for client sessions. For details, see the [“Using CTMConstants” section on page 31-15](#).
- The `ctm_config.txt` CSTM configuration files, which provide additional parameters. For details, see the [“Using the CTM Configuration File” section on page 31-15](#).
- **CTMException**, which is the generic exception handler. For details, see the [“Handling CTM Exceptions” section on page 31-17](#).

Using CTMClient

CTMClient provides static methods for invoking a remote object, and supports user-defined exceptions. If the server-side application throws an exception in the remote method, **CTMClient** re-throws the same exception. **CTM** will throw **CTMException** whenever there is an error during publishing or invoking a remote object.

For IPC and RPC communications, you can provide the remote host IP address as a parameter to the **CTMClient.invoke** call. The data is then tunneled over HTTP.

CISCO CONFIDENTIAL**Usage**

```
com.cisco.nm.xmls.ctm.client.CTMClient.invoke(
    java.lang.String urn,
    java.lang.String host,
    java.lang.String methodName,
    java.lang.Object[] parameters
    CTMClientProperties properties,
    CTMSerializer returnType,
    CTMParameterDesc[] par)
    throws Exception
```

```
com.cisco.nm.xmls.ctm.client.CTMClient.invoke(
    java.lang.String urn,
    java.lang.String host,
    java.lang.String methodName,
    java.lang.Object[] parameters
    CTMClientProperties properties,
    CTMSerializer returnType,
    throws Exception
```

Input Arguments

<i>urn</i>	The Unique Resource Name of the published object you want to call.
<i>host</i>	The host name or IP address of the remote host.
<i>methodName</i>	The name of the remote method that the client call wants to execute.
<i>parameters</i>	An object array of the parameters that must be passed to the remote method the client call wants to execute.
<i>properties</i>	The CTMClient properties that control the session. See the “Changing CTM Client Properties” section on page 31-14.
<i>par</i>	Parameter descriptor that has more information about the parameter passed. This is a wrapper class for type org.apache.axis.description.ParameterDesc.
<i>returnType</i>	Argument to register or unregister the serializer or deserializer in the IMarshal interface.

Example

The remote object `TestClass`, with method `testmethod`, is already published with URN `xyz`. The method parameters (paralist) can be accessed as an `Object []`, as follows:

```
com.cisco.nm.xmls.ctm.client.CTMClient.invoke("xyz",
    "testmethod",
    paralist)
```

Using CTMClientProxy

A client using `CTMClientProxy` has to know the interface implemented by the remote object. This means you can use `CTMClientProxy` only if the server-side object has implemented a remote interface. `CSTM` does not pose any restriction on this interface. By generating dynamic proxies, `CTMClientProxy` eliminates the need to generate stubs or a skeleton for each and every server-side remote object.

CISCO CONFIDENTIAL**Note**

The CTMClientProxy feature is available only if you are using JDK 1.3.1 or later.

If you do not have access to a remote interface on the client side, use CTMClient (see the [“Using CTMClient” section on page 31-10](#)) instead of CTMClientProxy.

CTMClientProxy simplifies remote method invocation and hides the mechanism for passing method names and parameters to the called object. However, this does not mean that the server maintains a separate instance for every proxy object. The total number of remote objects instantiated in the server process depends on the server policy. CSTM clients should not rely on the state of a remote object.

Like CTMClient, CTMClientProxy supports user-defined exceptions. If the server-side application throws an exception in the remote method, CTMClientProxy re-throws the same exception. CSTM will throw CTMException whenever there is an error during publishing or invoking a remote object.

As with CTMClient IPC and RPC communications, you can provide the remote host IP address as a parameter to the CTMClient.invoke call. The data is then tunneled over HTTP.

Usage

```
com.cisco.nm.xms.ctm.client.CTMClientProxy.getProxy(
    java.lang.Class[] interfaceClasses,
    java.lang.String urn,
    java.lang.String host,
    CTMClientProperties properties)
    throws CTMException
```

Input Arguments

<i>interfaceClasses</i>	An array containing the name of the remote- object interface(s) (a class or classes).
<i>urn</i>	The Unique Resource Name of the published object you want to call.
<i>host</i>	The host name or IP address of the remote host.
<i>properties</i>	The CTMClient properties that control the session. See the “Changing CTM Client Properties” section on page 31-14 .

Using the remote interface creates a local instance. You can then access any method of the remote interface; just use this local instance (you can also do this with an array of interfaces).

Example

To invoke a method called testmethod, using knowledge of the remote interface TestInterface:

```
TestInterface iTest=(TestInterface)CTMClientProxy.getProxy( TestInterface.class,
URN);
int result1 = iTest.testmethod(new byte[messageSize],
"2",
"0");
```

CISCO CONFIDENTIAL

Using CTMCall

CTMCall provides static methods for binding and invoking remote-object methods.

A client using CTMCall need not know the interface implemented by the remote object, eliminating the need to implement interfaces like this. Like CTMClient, the only contract required between the client and the server is the remote object's URN.

Unlike CTMClient, CTMCall maintains a socket connection with the remote object, so the same connection can be used for multiple requests. The CTMCall object will keep the remote connection alive until the object either explicitly closes the connection or the object itself is garbage-collected. Despite this, you should not rely on the state of the remote object. CTMCall will try to reconnect to the server once before throwing an exception.

Like CTMClientProxy, CTMCall is performance efficient. Note that CTMClientProxy internally constructs an instance of a CTMCall object to perform the remote invocation.

Like CTMClient and CTMClientProxy, CTMCall supports user-defined exceptions. If the server-side application throws an exception in the remote method, CTMCall re-throws the same exception. A CTMException is thrown by CTM incase of error during publishing and invoking the remote object.

As with CTMClient and CTMClientProxy, you can provide the remote host IP address as a parameter to the CTMClient.invoke call. The data is then tunneled over HTTP.

Usage

```
CTMCall ctmCall = new com.cisco.nm.xmls.ctm.client.CTMCall(
    java.lang.String urn,
    java.lang.String host,
    CTMClientProperties properties)

ctmCall.setMethodAndInvoke(String methodName, Object parameters[])

ctmCall.setMethodAndInvoke(String methodName, Object parameters[],CTMSerializer
returnType, CTMParameterDesc[] par)throws Exception (where ENCODING_STYLE is set to soap)
```

Input Arguments

<i>urn</i>	The Unique Resource Name of the published object you want to call.
<i>host</i>	The host name or IP address of the remote host.
<i>properties</i>	The CTMClient properties that control the session. See the “Changing CTM Client Properties” section on page 31-14.
<i>par</i>	Parameter descriptor that has more information about the parameter passed. This is a wrapper class for type org.apache.axis.description.ParameterDesc.
<i>returnType</i>	Argument to register or unregister the serializer or deserializer in the IMarshal interface.

CISCO CONFIDENTIAL**Examples**

To invoke `testmethod`, with parameter object `[] paraList`, when this method is defined in a class called `TestClass`, which has been exposed under the URN of `xyz`:

```
CTMCall call = new CTMCall ("xyz");
call.setMethod("testmethod", paraList);
call.invoke();
call.closeConnection();
```

To invoke an overloaded `testmethod`, with parameters object `[] paraList` ,

```
CTMSerializer ctmSerializer
= new CTMSerializer(namespace,localPart,beanName),
ParameterDesc paramDesc
= new CTMParameterDesc(xmlQname,mode,xmlTypeQname,javaType)
```

When this method is defined in a class called `TestClass`, which has been exposed under the URN of `xyz`:

```
CTMCall call = new CTMCall ("xyz");
call.setMethod("testmethod", paraList,ctmSerializer,paramDesc );
call.invoke();
call.closeConnection();
```

Changing CTM Client Properties

Pass `CTMClientProperties` to `CTMClient`, `CTMClientProxy` or `CTMCall` in order to alter default values for the following properties required to access the remote server:

- Timeout
- Encoding Style
- CTMServlet or other remote server URL
- Access Port

The relevant constants for these properties are defined in the `CTMConstants` (see the [“Using CTMConstants”](#) section on page 31-15).

Usage

```
public CTMClientProperties(
int timeout,
int encodingStyle,
String url,
int port)

public CTMClientProperties(int timeout)

public CTMClientProperties(int encodingStyle)
```

Input Arguments

<i>timeout</i>	The time (in milliseconds) that the client call waits for a response from the server before reporting a timeout exception. The default is 60000 milliseconds (1 minute). The timeout feature is currently disabled, since a connection pool is used internally.
<i>encodingStyle</i>	The encoding for messages to and from the remote object. The default is <code>BINARY</code> .

CISCO CONFIDENTIAL

url The URL string needed to access the CTMServlet or other remote server that will handle the CTM client requests. The default value is blank.

port The default value for the HTTP port is 80.

Example

To set CTMClient Call to use encoding style CTM_SOAP:

```
CTMClientProperties properties = new CTMClientProperties(CTM_SOAP);
```

To set the CTMClient call timeout to four minutes

```
CTMClientProperties properties = new CTMClientProperties(240000);
```

**Note**

The timeout feature is disabled since a connection pool is used internally.

Using CTMConstants

CTMConstants is a placeholder for all the public constants defined in CSTM. Currently, the only constants defined are:

- CTM_BINARY: The default encoding style for all IPC and RPC communications. Request and response objects are serialized between client and server.
- CTM_SOAP: The default encoding style for third-party applications. SOAP request and SOAP response are used for communication between the client and the server

Using the CTM Configuration File

The CSTM configuration file, `ctm_config.txt`, is stored in the same directory as the `CTM.jar` file, and sets parameters for the CSTM sessions. The configuration file entries and their default values are shown in [Table 31-1](#).

To change any of the default settings in this file, first stop all the VMs using CSTM, and then delete the `ctmregistry` and `ctmregistry.backup` files. These files are located in the same directory as the jar files.

Table 31-1 Configuration File Parameters

Parameter	Default Value	Description
SERVER_PORT=	40000	Sets the starting port number. Ports will be used in ascending order from this port. You can customize this port.
MAX_VM_PORTS =	20	Sets the maximum number of VMs (each instance of a CSTM Server runs in a separate VM). This parameter also sets the number of ports required for CSTM (MAX_VM_PORTS + 1). The registry server will use one port, while CSTM Servers running in different VMs will use the other ports.

CISCO CONFIDENTIAL**Table 31-1 Configuration File Parameters**

Parameter	Default Value	Description
SERVER_TIMEOUT=	7200000	Sets the time (in milliseconds; the default is 2 hours) after which the server deletes unused client entries and connection details. Note: This feature is currently disabled, per customer requirements.
MIN_THREADS=	10	Sets the minimum number of threads in the ThreadPool.
MAX_THREADS=	100	Sets the maximum number of threads in the ThreadPool.
CTM_URL=	:8080/ctm/CTMServlet	Stores the remote URL at which the CTMServlet can be accessed. The URL varies with the type of servlet engine running on the remote machine. For Tomcat, the parameter is usually set as follows: CTM_URL=:8080/ctm/CTMServlet.
CTM_SSL=	0	When set to 1, turns on SSL for URL connections. This provides secure communications between products.
THREAD_SLEEP=	180000	Sets the frequency (in milliseconds; the default is once every three minutes) at which the client connection pool-cleanup process occurs (that is, the cleanup thread will activate at this frequency).
SOCKET_IDLETIME=	600000	Sets the time (in milliseconds; the default is 10 minutes) limit after which an idle socket connection in the client connection pool is removed (that is, the socket is closed).
MAX_VM_CLIENT_CONNECTION=	10	The maximum number of connections maintained in the client connection pool per VM. Above this limit, the client will wait for a free connection. A client connection that is in use is made available when the client releases it. If this parameter is assigned a value of zero, CSTM will default it to 1 automatically.
CTM_FILE_UPLOAD_URL=	:8080/ctm/FileUpload	Specifies the remote URL at which the FileUpload servlet can be accessed. This varies with the servlet engine being run. For Tomcat it is usually the default value.
CTM_FILE_DOWNLOAD_URL=	:8080/ctm/CTMServlet FileDownload	Specifies the remote URL at which the FileDownload servlet can be accessed. This varies with the servlet engine being run. For Tomcat it is usually the default value.

CISCO CONFIDENTIAL**Table 31-1** Configuration File Parameters

Parameter	Default Value	Description
REGISTRY_LOCATION=	D:/Progra~1/CSCOpX/MDC/tomcat/webapps/cwhp/WEB-INF/lib	Sets the location of a common registry to be used under Tomcat. You must specify this value if you want to use a common registry. By default, the registry location is set to the directory where ctm.jar was installed.
REGISTRY_SERVER_TIME_OUT=	20000	Sets the socket timeout (in milliseconds) for reading from the common registry server. You should configure this value as appropriate for the number of concurrent publish actions you expect for the application.
REGISTRY_SERVER_RETRIES	3	Sets the number of times CSTM should attempt to connect to the common registry server.
SECURITY_WRAPPER	com.cisco.nm.xml.ctm.security.CMFSecurityWrapper	Identifies the class name of the security implementation used for authentication and authorization of remote calls through the servlet. This class must implement the com.cisco.nm.xml.ctm.common.ISecurityWrapper interface.

Handling CTM Exceptions

During publishing and invoking of remote objects, CSTM throws the generic CTMException. CTMException is thrown only by CSTM. [Table 31-2](#) lists the CTMExceptions and their occurrence conditions.

CTMException also provides messages indicating when an exception occurred, as shown in [Table 31-3](#). You can retrieve the invocation status using the following call:

```
int getInvocationStatus();
```

To ensure that exceptions received by CSTM from the underlying environment are wrapped in CTMException, use the following API call:

```
Throwable getException(); // returns the wrapped exception
```

With Binary encoding, CSTM also supports user-defined exceptions. Individual applications can throw their own exceptions on exposed methods of a remote object, and CSTM will re-throw the same exception on the client side.

In SOAP encoding, CSTM always throws CTMException. SOAP does not support user-defined exceptions. To make the remote exception information available, use a `getMessage()` call.

Table 31-2 CTMException Messages and Conditions

Message	Occurs When
DUPLICATE_URN	The URN is already registered.
URN_NOT_FOUND	Accessing a URN not present in the registry.

CISCO CONFIDENTIAL**Table 31-2** *CTMException Messages and Conditions (continued)*

Message	Occurs When
ERR_IN_OBJECT_CREATION	The exposed class definition is in the Classpath during compile time, allowing the code to compile, but is not visible later. Can also occur if the exposed class does not have a zero argument constructor.
ERR_IN_INVOCATION	An incorrect method signature is provided. Can only occur when using CTMClient or CTMCall. When using CTMClientProxy, an incorrect method signature will prevent the code from being compiled.
ERR_IN_CLIENT_CREATION	The client event cannot be created.
ERR_IN_CTM_SERVER_STARTUP	The CSTM server cannot start up because the registry is corrupt or cannot be created (that is, the directory containing the jar files is not writable). Occurs on server startup only.
ERR_IN_REGISTRY	An error is noticed in the registry, either because the registry has been removed or corrupted. Can also occur when either the server or the client are attempting to access the absent or corrupt registry.
REQ_ID_INVALID	The client request REQ_ID is invalid, corrupt, or has timed out. The server associates every request with an internal REQ_ID, which it times out after the interval specified in the SERVER_TIMEOUT parameter.
UNAUTHORISED_ACCESS	The user was trying to access a remote method whose URN was published with security options on, and the user could not be authorized.
USER_NOT_AUTHENTICATED	The user does not have the required permissions or role to access the remote object or method. When publishing a remote method, the publisher can specify if the particular URN needs to be accessed securely and register a set of roles that are authorized to use it.
ENCODING_STYLE_NOT_SUPPORTED	There is an attempt to use an encoding style other than Binary or SOAP.
ERR_IN_SOAP_MARSHAL_METHOD_ARGS	The CSTM client cannot convert the method name and arguments into a SOAP request message. This error can occur due to the argument objects not being serialized, failure to register the soap serializer, or other SOAP errors.
ERR_IN_SOAP_UNMARSHAL_METHOD_ARGS	The CSTM server cannot unmarshal the method name and arguments from the SOAP request message. This error can occur due to the argument objects not being serialized back to java objects, failure to register the soap serializer, or other SOAP errors.
ERR_IN_SOAP_MARSHAL_RESULT	The CSTM server cannot marshal the result into a SOAP response message. This error can occur due to failure to register the soap serializer or other SOAP errors.
ERR_IN_SOAP_UNMARSHAL_RESULT	The CSTM client cannot convert the SOAP response back to java objects. This error can occur due to the serializer for the returned object not being registered with CSTM or other SOAP errors.
ERR_IN_SOAP_SERIALIZER_REGISTER	An attempt to register the SOAP serializer fails.
ERR_IN_SOAP_SERIALIZER_UNREGISTER	An attempt to unregister the SOAP serializer fails.

CISCO CONFIDENTIAL**Table 31-2** *CTMException Messages and Conditions (continued)*

Message	Occurs When
ERR_IN_CONNECTING_TO_SERVER	Trying to connect to the server for the first time (that is, opening a connection).
ERR_IN_WRITING_REQUEST	The client's attempt to write the request fails. This is usually due to an error in the connection or the object stream.
ERR_IN_READING_REQUEST	The server's attempt to read the client request fails.
ERR_IN_WRITING_RESPONSE	The server's attempt to write the response fails.
ERR_IN_READING_RESPONSE	The client's attempt to read the response from the server fails.
ERR_IN_RELEASING_CONNECTION	Attempting to release the connection between a client and the server.
UNKNOWN_EXCEPTION	Error conditions or exceptions not covered by the other conditions in this table. CTMException will catch conditions caused by general Java exceptions and send the exception stack trace information with the UNKNOWN_EXCEPTION comment.

Table 31-3 *CTMException Invocation Status Messages*

Message	Indicates
BEFORE_INVOCATION	Exception occurred before invoking the method on the server.
AFTER_INVOCATION	Exception occurred after invoking the method on the server.
UNKNOWN_INVOCATION_STATUS	CSTM cannot determine whether the method was invoked on the published object.

Handling Special Requirements

The following topics provide guidelines for using CSTM to handle special requirements, including:

- [Implementing Secure CSTM Clients](#)
- [Running Registry Server as a Separate Process](#)
- [Registering the CSTM Port](#)
- [Using SOAP Encoding With CSTM](#)
- [Using the IMarshal Interface](#)
- [Using IMarshal's Register Method](#)
- [Performing CSTM File Transfers](#)
- [Retrieving HTTP Errors](#)

Implementing Secure CSTM Clients

CSTM exposes published objects securely with the help of the underlying security service. The CWCS framework provides interfaces for validation of the request (authentication/authorization). Security is only applied for RPC communication, that is, if the request originates from a CSTM client on the same machine, CSTM server will allow access any remote object on that box.

CISCO CONFIDENTIAL

Registry handler provides list of user roles that can access a particular URN and the underlying security service does the actual validation.

Regardless of encoding style, CSTM uses HTTP transport for RPC communication. So CSTM servlet acts as front-end and receives all incoming requests. The servlet locates the corresponding CSTM server with registry client interface and delegates the request. Applications can also publish objects in servlet engine VM. Before forwarding the request, the servlet will validate (authentication/authorization) the request if URN is exposed securely.

If security on a remote machine is turned on, the CSTM client calls (CTMClient, CTMClientProxy and CTMCall) must perform authentication before calling a published object on that remote machine.

CSTM uses the CWCS shared-secret mechanism to authenticate client access. In the shared-secret scheme, each CSTM client is logged on as a user, and thus has a user name and password. The administrator maps each user name to a particular role, and each user also registers a password, called a shared secret, with CWCS. In order for the CWCS security server to authenticate this user, the client must provide the shared secret and his login information in the HTTP stream. This login and shared secret information is coded in a cookie tag in the HTTP stream.

The CWCS security server, for each securely published URN, first receives the HTTP stream from the client and checks for the cookie tag in the stream. It then does authentication checks for this user against the published resource and its allowable roles.

The following is an example of a CTMCall using security options:

```
import com.cisco.nm.cmf.security.secret.SecretClient;
import com.cisco.nm.xml.ctm.common.CTMClientProperties;
public class TestSecureCall
{
    public static void main(String arg[])
    {
        String username = "admin";
        String secret = "admin";
        //host where the resource is published.
        String host = "sujathab-nt";
        //published URN(unique resource name)
        String urn = "abc";

        SecretClient sc = new SecretClient();
        String httphost = "http://" + host + ":1741";
        String cookie = sc.secretLogon(httphost,
            username,
            secret);
        CTMClientProperties Properties =
        new CTMClientProperties();
        Properties.setHttpHeaderEntry("Cookie", cookie);
    }
}
```

The CTMClientProperties is sent out as part of the CTMClient call in CTMClient, CTMCall or CTMClientProperties. Refer to TestSecureCall.java and TestSecureClientProxy.java, in the samples.jar file.

Note that, if a call to a published object on the local machine is made that explicitly refers to the host with the IP address of the local machine or host name of the local machine, the call will be treated as remote call and security checks will be performed on it. Security checks are not performed for any other local calls, where the host is not specified or the host value is "localhost".

CISCO CONFIDENTIAL

Running Registry Server as a Separate Process

Registry Server need not share the process space of other processes. We recommend that you start RegistryServer as a separate process, since this will reduce the overhead required to start Registry Server on demand (during synchronous call execution).

You can use the `isRegistryServerRunning()` API to identify whether the Registry Server has started. Your application must do this for any application processes that depends on Registry Server to start.

Registering the CSTM Port

If your application uses CSTM ports, you need to register the Registry Server port in the `/etc/services` file. Your application should do this during the application install, and remove the port during uninstall.

To register the port properly, your application installation should do the following:

1. Know the package in which `ctm.jar` is shipped
2. Check that the server port is free. This should be done during the installation prerequisites check. If the port is not free, it should select a random port (and check whether it is already in use as well).
3. Update this port number in the `/etc/services` file
4. Update the `SERVER_PORT` value in the `ctm_config.txt` file. This should be done during the post-install procedure.
5. Mark this file `ctm_config.txt` as volatile.

We also recommend that you register the port with CANA.

To ensure uniformity across applications using CWCS, follow the port naming convention `cscxxxxcstm`, where `xxx` is the name of your application. For example: For Common Services, the CSTM Registry Server port is named `cscocscstm`.

Using SOAP Encoding With CSTM

You can invoke server side functionality with SOAP encoding using either `CTMClient` or `CTMCall`. For details on the differences between these two types of invocations, see the [“Using CTMClient” section on page 31-10](#) and the [“Using CTMCall” section on page 31-13](#).

In both cases, you must have the client set the encoding property in `CTMClient.properties` to `CTM_SOAP` (see the [“Changing CTM Client Properties” section on page 31-14](#)). Once this property is set, the invocation process on the client side is basically the same as the process you use when encoding is set to the default `CTM_BINARY`.

However, if any of the arguments of the invoked method is a user-defined class, you must call `IMarshal`’s `register` function first, before invocation (see the [“Using IMarshal’s Register Method” section on page 31-24](#)).

`IMarshal` registration is needed to serialize all the parameters into equivalent SOAP representations. Generally speaking, you will need to register whenever:

- You want to send a SOAP request containing instances of one or more user-defined classes to a CSTM server for processing.
- The CSTM server will return an instance of a user-defined class as a result of your method invocation.

CISCO CONFIDENTIAL

- You have a CSTM client that wants to create a SOAP request. The client will need to register all the classes which are going to be a part of the request.

For example: You have a client which would like to invoke a method on the server. One of the method's arguments is an instance of a user-defined class, or composite object. For this argument to be serialized into an equivalent SOAP representation, CSTM requires that the client register this composite object with IMarshal. Once this is done, CSTM can generate an equivalent SOAP query for the input arguments of the method to be invoked.

The requirement is the same when an attempt is made to deserialize an incoming SOAP request containing an instance of a composite object. You must pre-register every incoming user-defined class with IMarshal.

Another example: You have a client which would like to invoke a function of a server application. CSTM accepts the parameters needed to invoke this function, `methodName`, parameters for the method, `args` and `urn` that identifies the server/resource whose functionality is to be invoked. CSTM invokes the function from the IMarshal interface and passes the input from the client.

The call then moves to CTMSoapMarshaller, which uses Apache Axis to create a SOAP-XML query based on the input from the client. This query (SOAP-XML) is then returned by the interface to CSTM, which transports it to the server. After the processing in the server, a SOAP-XML response is generated, which is transported by CSTM to the client. This response is converted to an equivalent Java Object using IMarshal interface and then passed to the client.

Consider another scenario, in which you have an external application, which would like to invoke a functionality of a server API. The application supports XML-SOAP compatible interface, that is, the interface generates SOAP queries and accepts SOAP responses. A typical example would be third parties or external applications, which may use CSTM to invoke functionality within the network management solution and pass a SOAP-XML query as input.

The SOAP-XML query message would contain the required data to invoke the method as well as the URN to identify the service. This query message is fed to CSTM. CSTM passes this query to a function supported by IMarshal, which uses Apache Axis to extract the data contained in the query (the name of the method to be invoked, a set of arguments for it, and the URN that identifies the service to be invoked). CSTM uses this data to invoke the service. CSTM converts the response from the service to an equivalent SOAP response using Apache-Axis and this is then returned by CSTM to the external application.

Using the IMarshal Interface

The IMarshal interface, which is used to access the functionality of Apache Axis, is as follows:

```
package com.cisco.nm.xmls.ctm.common;

public interface IMarshal
{
    public String marshalMethodAndArgs(String urn, String methodName, Object args[]) throws
    Exception ;

    public RPCData unmarshalMethodAndArgs(byte[] message) throws Exception ;

    public byte[] marshalReturnValue(String urn, String methodName, Object returnValue) throws
    Exception ;

    public Object unmarshalReturnValue(byte[] message) throws Exception ;

    public Object unmarshalReturnValue(byte[] message, CTMSerializer
    returnType,CTMParameterDesc[] paraDesc)

    public void register(CTMSerializer serialzer) throws CTMException;
```

CISCO CONFIDENTIAL

```
public void unregister(CTMSerializer serialzer) throws CTMException;
}
```

Using marshalMethodAndArgs

This function is used to generate an equivalent SOAP-XML query for an input.

The method uses a the following Apache-Axis classes:

RPCElement	Stores data that forms the body of a SOAP envelope. The attributes of the object of this class are the name of the method which was invoked, the urn of the service and the arguments/return value of the method.
SOAPEnvelope	Defines a SOAP envelope and has methods to extracts the data contained in it.
ServiceDescription	Indicates the type of message that is being created (a request or response) and the type of encoding desired.
Message	Defines methods used to convert a SOAP envelope into a string or byte array. Acts as a placeholder for SOAP-XML messages.

Input parameters

methodName	Method name to be invoked
urn	Unique identity for a specific service
args'	Arguments needed to invoke the method

Output

Equivalent SOAP-XML query based on the input parameters.

Using unmarshalMethodAndArgs

This method is used to deserialize a SOAP-XML message (the byte array input) to extract the following information: name of the method to be invoked, the arguments to be supplied to the method and the URN that identifies the service.

This method returns a user-defined class called RPCData. The attributes of this class are the name of the method to be invoked, the service urn and the arguments for the method.

Input parameters

A byte array that contains the SOAP_XML message to be deserialized.

Output

An object of type RPCData with the following attributes:

CISCO CONFIDENTIAL

<code>urn</code>	Unique resource name that identifies the service to be invoked
<code>methodName</code>	Name of the method to be invoked
<code>arguments</code>	Arguments to the method to be invoked

Using `unmarshalReturnValue`

This method is used to extract the return value (Java object) of the method that was invoked.

Input parameters

- A byte array, which contains a SOAP-XML message which needs to be deserialized.
- A byte array, which contains a SOAP-XML message which needs to be deserialized, the return type of the `CTMSerializer`, and the parameter description.

Output

Return value (Java object) of the method that was invoked.

Using `marshalReturnValue`

This method generates a SOAP-XML message containing the result of the method.

Input parameters

<code>methodName</code>	The name of the method that was invoked
<code>urn</code>	The urn identifying the service
<code>object</code>	The result of the method

Output

A byte array containing a SOAP-XML message representing the input parameters.

Using `IMarshal's Register Method`

This method is intended for use in situations where you want to use SOAP encoding and one or more of the arguments of the remote method is a user-defined class (see [“Using SOAP Encoding With CSTM” section on page 31-21](#)). The signature of the register method in the `IMarshal` interface is described below:

```
public void register(CTMSerializer serializer) throws CTMException;
public void register(CTMSerializer serializer, String dataType) throws CTMException;
```

Input Arguments

The input argument `serializer` wraps the required registration information shown below.

CISCO CONFIDENTIAL

namespace	The namespace for the composite object.
localPart	The local name for the composite object or the (xsi:type)
BeanName	The name of the class which defines the composite object. The path location of the class must be included in the name.

Usage

```
public CTMSerializer(
String namespace,
String localPart,
String beanName)
```

The implementation of the IMarshal interface is in CTMSoapMarshaller.java.

Example

We want to invoke `testMethod`, which has `testObject` as one of its arguments. In the client code where we invoke CSTM, we would do the following:

1. Set the encoding in CTMClientProperties to “CTM_SOAP”.
2. Register the composite object which needs to be serialized, as follows:

```
Class _cls=null;
_cls = Class.forName("com.cisco.nm.xml.ctm.soap.
CTMSoapMarshaller");
Method m = _cls.getMethod("getCTMSoapMarshaller",null);
IMarshal soap = null;
soap =(IMarshal) m.invoke(soap , null);
```

3. With the instance of IMarshal, call the register method and pass CTMSerializer, which will wrap the namespace, localname and the name of the class which contains the definition of the object:

```
soap.register(new CTMSerializer(
namespace,
localPart,
beanName));
```

4. Invoke `testMethod` as with any other client.

The namespace in which a user-defined class is registered in CSTM client should be consistent throughout. For more examples of the use of CSTM and SOAP encoding, refer to the following files in `samples-source.jar`: `TestSoapCall.java`, `TestSoapClient.java`.

Performing CSTM File Transfers

You can perform file uploads and downloads to and from a remote machine using the CTM File Transfer utility.

CTMFileTransfer provides separate methods for upload and download. Both methods throw CTMFileTransferException, which is explained in the [“About CTMFileTransferException” section on page 31-27](#).

CTMFileTransfer can be used when the application requests a file download or a file upload.

CISCO CONFIDENTIAL

Consider the scenario when an application requests a file download from another machine. In this case, the call from the download method of CTMFileTransfer is directed to the FileDownload servlet. The servlet is invoked as a web server and downloads the required file.

The case for FileUpload is similar to the one given for file download except that in this case the upload method of CTMfileTransfer is called which invokes the FileUpload servlet.

CTMFileTransfer Client Side Functionality

The two methods exposed by CTMFileTransfer are upload and download.

```
public static void download(String destFile , String srcIP , String srcFile)throws
CTMFileTransferException ;
public static void upload(String srcFile , String destIP , String destFile)throws
CTMFileTransferException ;
```

Using CTMFileTransfer Upload

This method is invoked to upload a file to a remote machine/box. The input arguments to the method are:

srcFile	Name of source file to be uploaded
destFile	Name of destination file
destIP	Destination IP address to which the file must be uploaded

The method calls the FileUpload servlet and begins to upload the file in chunks of 1024 bytes at a time. This is done to reduce the amount of memory consumed by the program for transfer of large files.

A URLConnection is established to the FileUpload servlet using the destination IP address. If the connection is not established a CTMFileTransferException.Host_Unreachable is thrown. The name of the destination file and data read from the source file is transmitted to the servlet.

If the source or destination files are not found, the following exceptions are thrown:

- CTMFileTransferException.Source_File_Not_Found
- CTMFileTransferException.Destination_File_Not_Found

After the transfer is completed, if the size of the source file and the destination file is found to be unequal then a CTMFileTransferException.Transfer_Interrupted is thrown.

Using CTMFileTransfer Download

This method is invoked to download a file from a remote machine/box. The input arguments to the method are:

srcFile	Name of the source file to be downloaded.
destFile	Name of the destination file
destIP	Destination IP address from which the file must be downloaded

CISCO CONFIDENTIAL

The method calls FileDownload servlet and begins to download the file in chunks of 1024 bytes at a time. This is done to reduce the amount of memory consumed by the program for transfer of large files.

A URLConnection is established to the FileDownload servlet using the destination IP address. If the connection is not established a CTMFileTransferException.Host_Unreachable is thrown. The name of the source file is transmitted to the servlet, the servlet then transmit the data read from the source file to the client.

If the source or destination files are not found, the following exceptions are thrown

- CTMFileTransferException.Source_File_Not_Found
- CTMFileTransferException.Destination_File_Not_Found

After the transfer is completed, if the size of the source file and the destination file is found to be unequal then a CTMFileTransferException.Transfer_Interrupted is thrown

CTMFileTransfer Server Side Functionality

The two servlets in the CSTM server are FileUpload and FileDownload.

FileUpload Servlet

The FileUpload servlet is invoked by the upload method of CTMFileTransfer. It receives the name of the destination file to which data has to be written. After verifying the existence of the file, the servlet writes data sent by the client to this file. If the destination file is not present, an exception is thrown and the client is notified about it.

FileDownload Servlet

The FileDownload servlet is invoked by the download method of CTMFileTransfer. It receives the name of the source file from which data is to be read. After verifying the existence of the file, the servlet sends it to the client. If the source file is not present, an exception is thrown and the client is notified about it.

About CTMFileTransferException

The following CTMFileTransferException messages are thrown for various conditions.

Message	Cause
SOURCE_FILE_NOT_FOUND	Incorrect source file name or the source file is not found.
DESTINATION_FILE_NOT_FOUND	Incorrect destination file name or destination file name not found.
HOST_UNREACHABLE	The specified host cannot be reached. This is usually due to an invalid IP address or the fact that no web server is running on the specified IP address.
TRANSFER_INTERRUPTED	The file transfer was interrupted
DESTINATION_INVALID	The destination is invalid.

CISCO CONFIDENTIAL

Retrieving HTTP Errors

You can get HTTP error codes and response messages using CTMException and one of the following three calls.

```
Hashtable h = ctMex.getHttpErrorHash();
getHttpStatusCode()
getHttpResponseMessage()
```

Note that in the `getHttpErrorHash()` call, the hashtable contains the error code and error messages, with the keys `ErrorCode` and `ErrorMessage`, respectively.

Which call you use depends on the JRE you are using, and the value you want. If you are using JRE 1.3.1_04, use the `getHttpErrorHash()` call only. If you are using JRE 1.3.1_06 or later, use any of these three calls and get the required value (code, message, or both).

Note that only the `getHttpErrorHash()` call will have both the Error Message and ErrorCode values, and both will be converted into strings.

Using the CTMTest Tools and Samples

You can use the CTMTest tool to:

- Get a feel for the operations that CSTM lets you perform.
- Check the exception cases and error cases and make sure they are relevant.
- Check the performance using multiple scenarios.

You can execute the CTMTest operations using the test class in the samples.jar file. The test file lets you:

- Set the message size of an arbitrary byte stream.
- Invoke a client using short command line calls: CL for CTMClient, CLP for CTMClientProxy, or CLL for CTMCall.
- Set the timeout value as needed.
- Simulate the processing delay in the server method.

While running CTMTest, you can use the up-arrow and down-arrow to access previous and next commands you issued. Depending on your platform and command history, the CTMTest command history is stored across invocations. For example, if you access CTMTest and then exit, the next time you access CTMTest, you should still be able to access the history from the previous invocations. This is handy when you are trying to perform CTMClient calls with multiple options.

While calculating the number of messages per second, CTMTest tracks delays in accessing the serialized parameter file and does not use them in the calculation. Calculated time taken for each of the Client operations does not include parameter-file access delays.

The following topics describe several useful applications of the CTMTest tool, including:

- [Creating a Custom Test File, page 31-29](#)
- [Publishing a Test Object, page 31-29](#)
- [Unpublishing a Test Object, page 31-29](#)
- [Accessing a Test Method Using CTMClient, page 31-30](#)
- [Accessing a Test Method Using CTMClientProxy, page 31-30](#)

CISCO CONFIDENTIAL

- [Accessing a Test Method Using CTMCall](#), page 31-31
- [Testing CSTM Communications](#), page 31-31

Creating a Custom Test File

In order to test with your own classes rather than the test classes given in the samples.jar, you must:

- Add your new class to the classpath.
- Serialize the parameters that need to be passed in to your class.

You can do this by writing your own file, and including the serialization methods given in the Serializer class. The example file MyTestClass.java shows how to add the parameters to an object array and serialize them into a file.

If the parameters that need to be passed to the method are composite objects, they will need to implement Serializable and will need to have a no-argument constructor. The name of this serialized file will then be given as a command line argument to the CTMTest tool when you run the client CL, CLL or CLP commands.

Publishing a Test Object

To publish a test object:

-
- Step 1** At the command prompt, enter `java CTMTest`. The CTMTest tool runs.
- Step 2** At the `CTMTEST>>` prompt, enter the command `P URN Classname Option`, where:
- *URN* is the Universal Resource Name you want to assign to the object.
 - *Classname* is the name of the class to be exposed.
 - *Option* is `-s` or blank if you want to publish by passing a single reference to the class, or `-d` to publish by passing the class definition.
-

Unpublishing a Test Object

To unpublish a test object:

-
- Step 1** At the command prompt, enter `java CTMTest`. The CTMTest tool runs.
- Step 2** At the `CTMTEST>>` prompt, enter the command `U URN` where *URN* is the Universal Resource Name assigned to the previously published object.
-

CISCO CONFIDENTIAL

Accessing a Test Method Using CTMClient

To access a method on a test object using CTMClient:

Step 1 At the command prompt, enter `java CTMTest`. The CTMTest tool runs.

Step 2 At the `CTMTEST>>` prompt, enter the following command:

```
CL URN methodname parmlist options
```

Where:

- *URN* is the Universal Resource Name of the object whose method you want to call.
- *methodname* is the name of the method you want to call.
- *parmlist* is the name of a file containing the serialized parameters for this method. If this file is not in the current directory, specify the filename with entire path.
- *options* is one or more of the following:
 - `-c` is the number of clients to spawn. The default is 1. If you choose to spawn multiple clients, the system starts off separate threads for each CTMClient request.
 - `-i` is the IP_address or server name of the remote host to be accessed. The default is `localhost`.
 - `-t` is the CTMClient Timeout value, in milliseconds. The default is 5000. This is useful for setting the timeout value of the CTMClient call when it has delays accessing the server.
 - `-n` is the number of messages to be sent. The default is 1000. Changing this value is useful when you know you will have many messages to be sent.

Accessing a Test Method Using CTMClientProxy

To access a method on a test object using CTMClientProxy:

Step 1 At the command prompt, enter `java CTMTest`. The CTMTest tool runs.

Step 2 At the `CTMTEST>>` prompt, enter the following command:

```
CLP URN methodname parmlist options
```

Where:

- *URN* is the Universal Resource Name of the object whose method you want to call.
- *methodname* is the name of the method you want to call.
- *parmlist* is the name of a file containing the serialized parameters for this method. If this file is not in the current directory, specify the filename with entire path.
- *options* is one or more of the following:
 - `-c` is the number of clients to spawn. The default is 1.
 - `-i` is the IP_address or server name of the remote host to be accessed. The default is `localhost`.
 - `-t` is the CTMClient Timeout value, in milliseconds. The default is 5000. This is useful for setting the timeout value of the CTMClient call when it has delays accessing the server.

CISCO CONFIDENTIAL

- `-m` is the message size. The default is 1000 bytes. Changing this value is useful if you know that the specific test method has a larger message size defined in the `TestInterface`.
- `-n` is the number of messages to be sent. The default is 1000. Changing this value is useful when you know you will have many messages to be sent.

Accessing a Test Method Using CTMCall

To access a method on a test object using `CTMClientProxy`:

Step 1 At the command prompt, enter `java CTMTest`. The CTMTest tool runs.

Step 2 At the `CTMTEST>>` prompt, enter the following command:

```
CLL URN methodname parmlist options
```

Where:

- `URN` is the Universal Resource Name of the object whose method you want to call.
- `methodname` is the name of the method you want to call.
- `parmlist` is the name of a file containing the serialized parameters for this method. If this file is not in the current directory, specify the filename with entire path.
- `options` is one or more of the following:
 - `-c` is the number of clients to spawn. The default is 1.
 - `-i` is the IP_address or server name of the remote host to be accessed. The default is `localhost`.
 - `-t` is the CTMCall Timeout value, in milliseconds. The default is 5000. This is useful for setting the timeout value of the CTMCall when it has delays accessing the server.
 - `-n` is the number of messages to be sent. The default is 1000. Changing this value is useful when you know you will have many messages to be sent.

Testing CSTM Communications

If you want to test CSTM communication between applications on the same machine:

1. Open two CTMTest windows.
2. Send messages from one window to the other.
3. Try to connect to a published resource in the first command window from the second window.
4. Try to publish and perform client access from the same command window.

To simulate CSTM communications between applications on different machines:

1. Make sure Tomcat 3.2.1 is running on the machine that will act as the server.
2. Make sure that CSTM is registered with Tomcat on the server by following the steps in the [“Installing CSTM with the Tomcat Servlet Engine”](#) section on page 31-3.

CISCO CONFIDENTIAL

3. Publish the class on the server.
4. Access the exposed resource from one or more clients by specifying the IP address of the server machine.

Using the Sample TestClass

The samples file `samples.jar` contains a sample class called `TestClass`, which:

- Includes `TestMethod`.
- Includes the local variable `Count`.
- Implements `TestInterface`.

`TestMethod` does the following:

- Maintains the `Count` variable.
- Increments `Count` by 2 every time it is invoked.
- Makes a byte array of a size you can specify.
- Allows you to test timeout parameters by setting a sleep time value as a parameter.
- Contains a demonstration composite class object as a parameter. Please note that for composite classes, you must implement the interface `Serializable` and also have a zero-argument constructor.

To set these parameters for `TestMethod`, you set the values into the parameter array and serialize it into a file with a specific name. For example, you might enter `>>java MyTestClass 60000000 0 bigfile`, where:

- `60000000` is the size of the byte array (message size). In this case, the message size is 60Mbytes.
- `0` is the simulated delay (in milliseconds) in the method.
- `bigfile` is the name of the serialized file.

This serialized file can then be given as input to the `CTMTest` command prompt, so that the respective parameters are set in the particular method. The class `MyTestClass` is also included in the `samples.jar` file.

Using the CSTM Samples

The CSTM `samples.jar` file contains the `CTMTest` utility java files and classes, plus several other test and sample java files and classes. [Table 31-4](#) shows the java and class files included in `samples.jar`.

You can use these sample files and classes with `CTMTest` to perform a variety of useful tests, including:

- [Testing Parameter Passing, page 31-33](#)
- [Testing for Timeout Errors, page 31-33](#)
- [Testing Multiple Clients, page 31-34](#)

CISCO CONFIDENTIAL**Table 31-4 Java and Class Files in Samples.jar**

File/Class Name	Description
CTMTest	The CTMTest utility main class.
TestClass	This is accessed from the CTMTest utility. This class includes TestMethod and implements the TestInterface.
TestInterface	This Interface class is implemented by TestClass and is used by CTMClientProxy.
Serializer	Use methods of this class to serialize the method parameters.
MyTestClass	An instance of a class using Serializer methods to serialize the parameters to the TestMethod defined in TestClass.
CompositeObject	A file used to demonstrate that the parameters passed to the method of an exposed class can be a composite object. An instance of a composite object is passed to the TestMethod defined in TestClass.
TestServer	Standalone test programs for testing TestServer alone.
TestClient	Standalone test program for testing TestClient alone.
TestClientProxy	Standalone test program for testing TestClientProxy alone.
TestCall	Stand alone test program for testing TestCall alone.
TestServlet	Standalone test program for testing publishing an object from a servlet.

Testing Parameter Passing

You must first make a serialized file to pass the parameters to the test method:

```
>> java MyTestClass 1000 0 1kfile
```

Then start the CTMTest tool:

```
>> java CTMTEST
```

Publish the object containing the test method

```
CTMTEST>> P xyz TestClass
```

You can then pass the serialized file to the published method, using any of the three CSTM client calls:

```
CTMTEST>> CL xyz testmethod 1kfile -n 1000
CTMTEST>> CLL xyz testmethod 1kfile -n 1000
CTMTEST>> CLP xyz testmethod -n 1000 -m 1000
```

In the case of the CTMClientProxy call (CLP), remember that the remote interface is hard coded, so it always uses TestInterface and TestMethod. However, the message size parameter can be varied, so there is no need to use the serialized file to pass in parameters in this case, because the other parameters are hardcoded.

Testing for Timeout Errors

Use the following call to create a serialized file with message size set to 1000bytes and the server side delay set to 5000 milliseconds:

```
>> java MyTestClass 1000 5000 1k_5msec_file
```

CISCO CONFIDENTIAL

Then publish the TestClass:

```
CTMTEST>> P xyz TestClass
```

Then make a client call with the timeout set to 1000 milliseconds:

```
CTMTEST>>CL xyz testmethod 1k_5msec_file -n 1000 -t 1000
```

This will cause timeout error conditions, since the server delay is set at 5000 milliseconds, but the client timeout is set at 1000 milliseconds. The client will wait for 1000 milliseconds to hear from the server, and once the server calls a sleep method for 5000 milliseconds, the client call will timeout.

Testing Multiple Clients

Use the -c option to test for multiple (three, in this case) CTMClient calls:

```
CTMTEST>> CL xyz testmethod 1kfile -c 3 -n 1000
```

Guidelines for Using CSTM

Following are the recommendations or guidelines that applications should follow when using CSTM:

- Starting Registry Server

It is recommended that you start CSTM Registry Server as a separate process, since it receives most of the CSTM Calls (both server and client). Starting Registry Server in a heavily loaded environment like Tomcat is also not advisable.

Sample code is as follows:

```
import com.cisco.nm.xms.ctm.registry.*;
import org.apache.log4j.Category;
public class StartCTMRegistryServer
import com.cisco.nm.xms.ctm.registry.*;
import org.apache.log4j.Category;
public class StartCTMRegistryServer
{
    static Category cat = Category.getInstance("CTM.Registry");
    public static void main(String arg[])
    {
        if(!CTMRegistryServer.isRegistryServerRunning())
        {
            CTMRegistryServer.startRegistryServer();
            cat.debug("Registry Server started successfully");
        }
        else
        {
            cat.debug(" Registry Server was started successfully in other
JVM");
        }
    }
}
```

- Proper usage of CSTM Client APIs provided by CSTM
 - For applications which have both CSTMServer and CTMClient on the same machine:

CISCO CONFIDENTIAL

- If the requirement is to have several calls from the client to the server in a short span of time, create an instance CTMCall or CTMClientProxy object and use the same object for multiple calls, finally call CloseConnection. This reduces unnecessary overhead of creating sockets on the client side.
- If calls to the server are made infrequently, CTMClient can be considered.
 - For applications which have CSTMServer and CSTMClient on different machines:
Any instance of CTMCall or CTMClient or CTMClientProxy can be used.

Multithreaded-environment related guidelines:

- On Client side:
If the application wants to share same instance of CSTM client object, from multiple-threads, then use an instance of CTMClientProxy class, which is thread-safe and synchronized.
- On Server side:
 - Application can expose CSTM urn for either reference to the object whose functionality needs to be exposed or its class.
 - Application should supply the reference, if exposed object is thread-safe. In this case CTM will use the same object over simultaneous or multiple incoming requests and also Client should rely on the state of the server object.
 - Instead of reference if Class is supplied, CTM creates separate instance of remote object. This object gets garbage collection, after client loses connection with the server by either explicit close or when client call object gets garbage collected.
- CSTM also wraps HTTPReponse error code with CSTM Exception that occurs on the client side.
When CTMException is encountered on CTM client side, application can also check `ctmexception.getHttpResponseMessage` to get message / `ctmexception.getHttpResponseCode` for Code or `ctmexception.getHttpErrorHash` to get both error code and message as key value pairs.
- Application should properly unpublish all the Urns that they publish on processes shutdown. It is required for CSTM to properly cleanup CTM registry. When the processes start up later, this will avoid any port-in-use issues.
- The CSTM configuration file, `ctm_config.txt`, is stored in the same directory as the `CTM.jar` file, and sets parameters for the CSTM sessions. Applications can change these default settings depending on their needs. See [Using the CTM Configuration File](#) for more details.

CISCO CONFIDENTIAL