



CORBA Architecture and Application Programming Interface

This chapter describes the Common Object Request Broker Architecture (CORBA) adapter architecture and application programming interface (API) for the Cisco BTS 10200 Softswitch.

CORBA Adapter Architecture

The CORBA adapter (CAD) interface leverages the adapter architecture of the Element Management System (EMS) component in the Cisco BTS 10200 Softswitch. This architecture allows for a variety of adapters to provide operations, administration, management, and provisioning (OAM&P) by adapting the external interface to a common infrastructure in the EMS. [Figure 1-1](#) illustrates the overall architecture for the CAD and shows the CORBA architecture.

The architecture provides dual mode support for secure and nonsecure CORBA, which are active at the same time. This was an install option in previous releases. The non-secure mode and the secure mode are fully supported. As part of this dual mode support, there are two java processes on the Cisco BTS 10200 Softswitch EMS that manage the CIS application. Each process is marked with a unique name to indicate its unique function.

- Nonsecure is `-D_CIS_IIOPI`
- Secure is `-D_CIS_SSLIOP`

There are side effects to this dual mode of support. When both modes are active, the new secure CORBA mode does not behave as it did in previous releases. This new behavior includes name space collisions, so a new secure POA context and name space is provided to avoid problems.

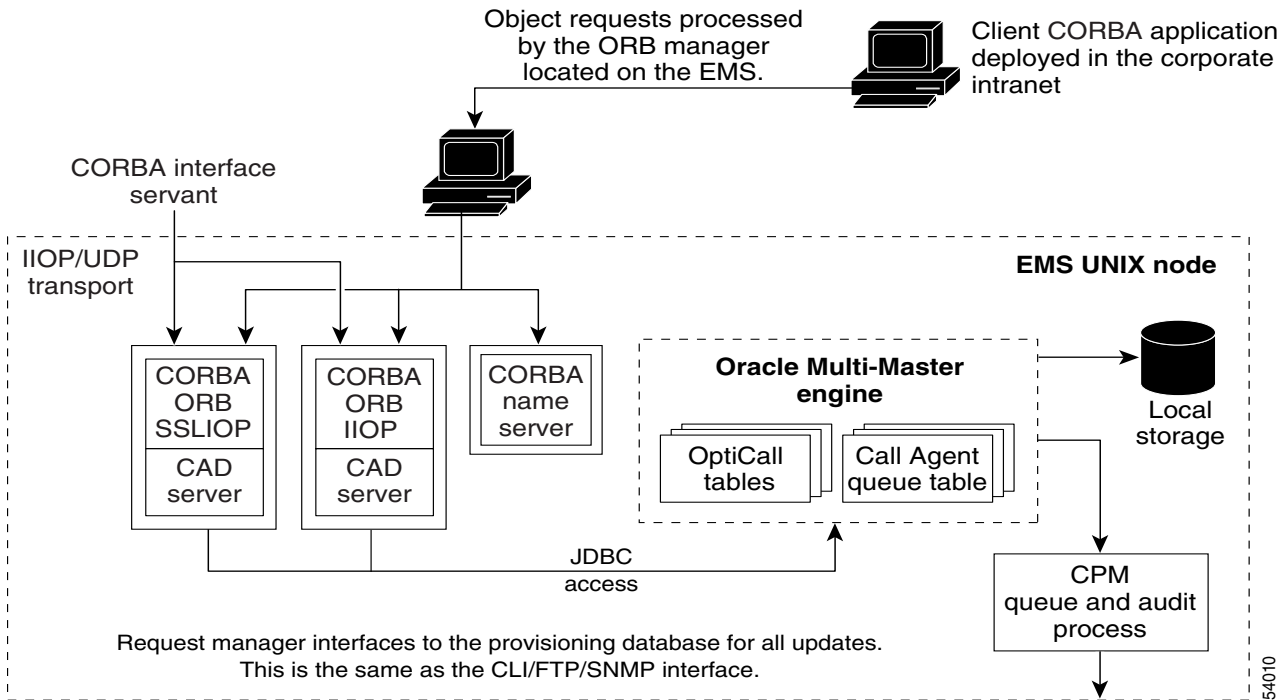
Both the non-secure mode and the secure mode only allow connections on the active EM01 EMS application and automatically drop connections if the active EM01 application fails or manually switches over to its redundant mate. It will, in addition, remove its objects from the local INS or NameService so that no new queries can successfully resolve to that particular EMS. This solves a legacy CORBA issue by preventing any provisioning from a standby EMS.

The CORBA Installation automatically uses VIP (Virtual IP Address) as the `iiop.hostname`, if the VIP is configured. If VIP is not configured, the first EMS Management IP address is used as the `iiop.hostname`. This allows the NameService to listen to all IP addresses on the active EMS. For the use of VIP, please refer to the *Cisco BTS 10200 Softswitch CLI Reference Guide, Operations, Administration and Maintenance*.

Installing CORBA also installs the CORBA SDK onto the EMS. The installer can run the sample test program to verify the CORBA installation. See [Appendix C, “Sample CORBA Client Package \(BTSxsdk\) Implementation”](#) for more information.

The `bts.properties` file in the CIS application affects client application development. Login sessions expire in 10 minutes with no activity. This means that a command must traverse each session once every 10 minutes to keep a session alive. This is important for any client application that deploys the use of connection pools.

Figure 1-1 CORBA Architecture



ORB Specifications

The Object Request Broker (ORB) used in the CAD interface is the OpenORB 1.3.1 compliant package. The ORB also supports other advanced features like the portable object adapter (POA). POA is the implementation model used in CAD.

Compiler Tools

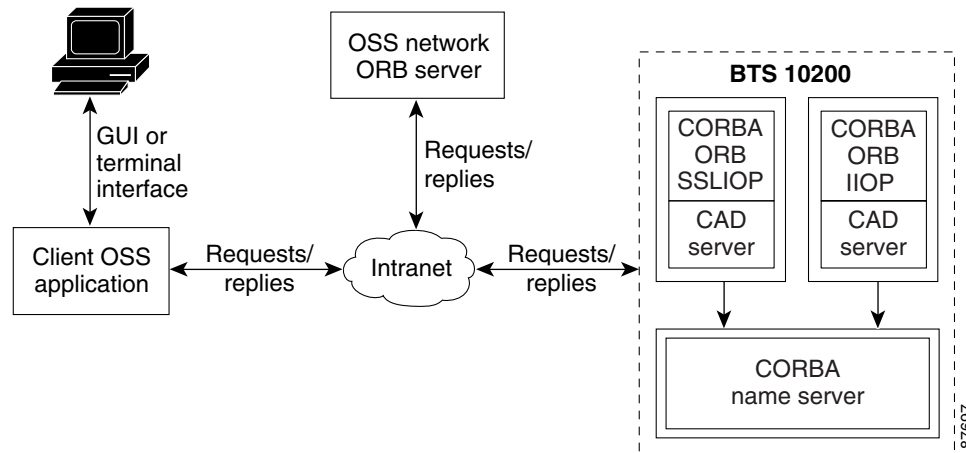
The minimum required compiler is the J2SE Development Kit (JDK)1.4.1. The Cisco BTS 10200 Softswitch uses **JDK1.5.0** or **JDK5** for compilation and for the Java Runtime Environment (JRE). Additional tools that may be required for the client side application are:

- Xerces parsers
- ECS Report Builder

ORB Deployment

Figure 1-2 shows the ORB deployment process.

Figure 1-2 ORB Deployment



When using OpenORB for the client side application, a few basic steps must be performed to ensure that the client environment is properly set up and ready for the application. The first is to configure the JVM on the client machine to use OpenORB as the primary ORB for Java. This can be done at runtime or it can be permanently set in the orb.properties of the JVM. To perform the latter requires privileges of the owner of the JVM install. In most cases this is root. The root user must execute the following command:

```
java -jar openorb_1.3.1.jar
```

This places the orb.properties settings in the correct location with the following values.

```
org.omg.CORBA.ORBClass=org.openorb.CORBA.ORB
org.omg.CORBA.ORBSingletonClass=org.openorb.CORBA.ORBSingleton
```

Otherwise, the settings must be supplied as environmental overrides to each invocation of the client application.

Additional values that are of use to the client programmer are the OpenORB DEBUG options. These control the volume of debug information produced by the client application. The debug information is sent to the standard output of the application. If a programmer wishes to capture this data into log files, use the basic shell redirect commands to redirect the data. Other environments such as web services can behave differently. The options for getting the highest degree of debug output are listed below. These produce IIOP/SSLIOP message dumps which are the most useful in debugging issues with communications with the Cisco BTS 10200 Softswitch.

```
-Dopenorb.debug.trace=DEBUG -Dopenorb.debug.level=HIGH
```

These options are supplied as an environment setting to the client application as part of the Java invocation. The following examples illustrate the interface definition language (IDL) compilation and Java code required for locating the POA and binding it to the compiled IDL objects.

This OpenORB generic process is a critical part of the development of any client application interfacing to the Cisco BTS 10200 Softswitch. The IDL supplied by the Cisco BTS 10200 Softswitch must be compiled into interface classes and then used in the client application.

**Note**

IDL is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. In distributed object technology, new objects must discover how to run in any platform environment to which they are sent. An ORB is a middleware program that brokers client/server relationships between objects.

This code example uses the Java package tree as developed in the Cisco BTS 10200 Softswitch product. This code can vary. Other clients can specify a different package tree to contain the IDL interface objects. See the SDK for a detailed breakdown of this script.

```
#!/bin/sh
#####
# Copyright (c) 2002, 2006 by Cisco Systems, Inc.
#
# AUTHOR: A. J. Blanchard
#
# DESC: Invoke the IDL compiler for the OpenORB package.
#
#####
set -e
set -a
#set -x

#
# List required jar files
#
CLASSPATH=./opt/BTSorb/lib/logkit.jar:/opt/BTSorb/lib/openorb-1.0.1.jar:/opt/BTSorb/lib/
/openorb_tools-1.3.1.jar:/opt/BTSorb/lib/xerces.jar:/opt/BTSorb/lib/avalon-framework.jar
:/opt/BTSorb/lib/openorb_ots-1.3.1.jar:/opt/BTSorb/lib/openorb_pss-1.3.1.jar:/opt/BTSor
b/lib/openorb_ins-1.3.1.jar:/opt/BTSorb/lib/openorb_tns-1.3.1.jar

export CLASSPATH

java -classpath $CLASSPATH org.openorb.compiler.IdlCompiler $1 -jdk1.4 -all -verbose -d ./
```

Java files are generated in a local directory tree specified in the package directory. This package path is required in the bind logic to find the object interface implementation.

```
#!/bin/sh
#####
# Copyright (c) 2002, 2006 by Cisco Systems, Inc.
#
# AUTHOR: A. J. Blanchard
#
# DESC: Compile Java ORB programs with the required components from OpenORB.
#
#####
set -e
set -a
#set -x

CLASSPATH=./opt/BTSorb/lib/logkit.jar:/opt/BTSorb/lib/openorb-1.3.1.jar:/opt/BTSorb/lib/
/openorb_tools-1.3.1.jar:/opt/BTSorb/lib/xerces.jar:$HOME/mb/devel/em/lib/cad.jar:$HOME/m
b/devel/em/lib/ecs-1.4.1.jar:/opt/BTSorb/lib/openorb_ots-1.3.1.jar:/opt/BTSorb/lib/openo
rb_pss-1.3.1.jar:/opt/BTSorb/lib/openorb_ins-1.3.1.jar:/opt/BTSorb/lib/openorb_tns-1.3.1
.jar

export CLASSPATH

javac -classpath $CLASSPATH -d ./ $*
```

Compile a package of Java files to generate the required class files. These class files must exist in the client classpath.

**Note**

This is a common example where all the java files in a single directory are built with a single command. This is one of the fastest ways to compile bulk java code.

The Cisco BTS 10200 Softswitch offers a Software Developers Kit (SDK) with a complete range of examples that utilize the CORBA interface. These include many topics such as:

- CLI
- Batch file processing
- Multi-thread concurrency
- SSL

**Note**

The example below illustrates the basic abstraction of the BTS 10200 objects. If other tools are used, you must modify the IDL objects as well as the OpenORB files.

```
package com.sswitch.oam.ccc;

import java.lang.*;
import java.io.*;
import java.util.*;
import java.text.*;
// CORBA stuff
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.Messaging.*;
// XML Stuff
import org.apache.ecs.xml.*;
import org.apache.ecs.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.xml.serialize.*;
// BTS Code jar files...
import com.sswitch.oam.cad.*;
import com.sswitch.oam.xml.*;
import com.sswitch.oam.util.*;

/**
 * CorbaXmlIntf.java
 * Copyright (c) 2002, 2006 by Cisco Systems, Inc.
 * -- This is the client side driver stub. This allows the client application
 * to generate the Request object which is then digested in this class as a
 * XML document and sent as a request to the CORBA server. The results are
 * then returned to the user or the CORBA exception is thrown.
 *
 * @author   A. J. Blanchard
 * @version  4.0
 * @since    BTS 10200 4.0
 */

public class CorbaXmlIntf {

    /*
     * Class private data
     */
    private String []                objArgs;
    private org.omg.CORBA.ORB        objOrb;
    private org.omg.CosNaming.NamingContextExt objContext;
```

```

private com.sswitch.oam.cad.Bts10200          objBts;
private com.sswitch.oam.cad.Bts10200_Security objBtsSec;
private org.omg.CORBA.StringHolder          objKey;

/**
 * Generic Constructor for the test driver.
 */
public CorbaXmlIntf(String[] args)
{
    // Initialize the ORB.
    objOrb = org.omg.CORBA.ORB.init(args, null);
    objArgs = args;
    return;
}

/**
 * This is the primary execution method for the object. It performs the
 * actual request and calls for the print of the reply.
 */
public void connect() throws CadExceptions
{
    //
    // Log into the target machine with generic optiuser
    //
    try {
        bind();
        objKey = new org.omg.CORBA.stringholder();
        objBtsSec.login("btsadmin", "btsadmin", objKey);
        Log.info("BTS10200 Login successful: "+objKey.value);
    }
    catch (Exception e) {
        Log.error("Exception in CORBA Bind/Login = "+
            Util.stackTraceToString(e));
        throw new CadExceptions(1, e.toString());
    }
}

/**
 * This method generate the request to the CORBA server and returns
 * the reply or an exception if the interface throws an exception.
 * The argument "request" must be an XML formatted document.
 *
 * @param request This XML request document.
 * @returns String This is the XML formatted answer.
 */
public String request(String request)
    throws CadExceptions
{
    String answer=null;
    try {
        org.omg.CORBA.StringHolder reply = new org.omg.CORBA.StringHolder();

        // Issue request to BTS 10200
        objBts.request(request, objKey.value, reply);

        // Build an XMLReply from the document

        answer= reply.value;
    }
    catch (Exception e) {
        Log.warning("Request Command Exception:\n " +
            Util.stackTraceToString(e));
        throw new CadExceptions(1, e.toString());
    }
    return answer;
} // end request()

/**
 * This method generate the request for a command document to the
 * CORBA server and returns the reply or an exception if the interface

```

```

* throws an exception.
*
* @param noun      This noun for the request.
* @param verb     This verb for the request.
* @returns String  This is the XML formatted answer.
*/
public String  getCommandDoc(String verb, String noun)
    throws CadExceptions
{
    String  answer=null;
    try {
        org.omg.CORBA.StringHolder reply = new org.omg.CORBA.StringHolder();

        // Issue request to BTS 10200
        objBts.getCommandDoc(noun, verb, objKey.value, reply);

        // Build an XMLReply from the document

        answer= reply.value;
    }
    catch (Exception e) {
        Log.warning("Request Command Exception:\n " +
            Util.stackTraceToString(e));
        throw new CadExceptions(1, e.toString());
    }
    return answer;
} // end getCommandDoc()

/**
* This method generate the request for a command document to the
* CORBA server and returns the reply or an exception if the interface
* throws an exception.
*
* @param noun      This noun for the request.
* @param verb     This verb for the request.
* @returns String  This is the XML formatted answer.
*/
public String  getExtCommandDoc(String verb, String noun)
    throws CadExceptions
{
    String  answer=null;
    try {
        org.omg.CORBA.StringHolder reply = new org.omg.CORBA.StringHolder();

        // Issue request to BTS 10200
        objBts.getExtCommandDoc(noun, verb, objKey.value, reply);

        // Build an XMLReply from the document
        answer= reply.value;
    }
    catch (Exception e) {
        Log.warning("Request Command Exception:\n " +
            Util.stackTraceToString(e));
        throw new CadExceptions(1, e.toString());
    }
    return answer;
} // end getExtCommandDoc()

/**
* This module disconnects the user from the BTS 10200 CORBA interface.
*/
public void  disconnect()  throws CadExceptions
{
    objBtsSec.logout(objKey.value);
    return;
}

/*=====
* Internal processing methods...
*=====*/

```

```

/**
 * This method binds to the target CORBA objects for us to operate
 */
protected void      bind()
    throws org.omg.CORBA.ORBPackage.InvalidName,
           org.omg.CosNaming.NamingContextPackage.InvalidName,
           org.omg.CosNaming.NamingContextPackage.NotFound,
           org.omg.CosNaming.NamingContextPackage.CannotProceed
{
    org.omg.CosNaming.NameComponent[] nameComponent = null;
    org.omg.CORBA.Object result = null;

    insLocate();

    result = objContext.resolve(objContext.to_name("Bts10200_Security_poa"));
    objBtsSec = Bts10200_SecurityHelper.narrow(result);

    result = objContext.resolve(objContext.to_name("Bts10200_poa"));
    objBts = Bts10200Helper.narrow(result);
    Log.info("Basic POA(s) have been located and bound.");
    return;
}

/**
 * Load the name service and find the context for the CORBA objects.
 * Remember, the INS must be the one located on the BTS. This has the
 * object references. Use a 'corbaloc:' for now but later a migration
 * to URL for name service location would be good.
 */
protected void      insLocate()
    throws org.omg.CORBA.ORBPackage.InvalidName
{
    //System.out.println("Locate NameService in system.");
    org.omg.CORBA.Object initial_context_obj =
        objOrb.resolve_initial_references("NameService");
    objContext =
        org.omg.CosNaming.NamingContextExtHelper.narrow(initial_context_obj);
    Log.info("NameService found in initial context.");
    return;
}
} // end CorbaXmlIntf

```

- Actual implementations can make the POA selection dynamic and based on some form of navigation to a site (for example, to a softswitch home location or perhaps part of the softswitch ID). Once a POA is selected, all object implementations are the same. No site-specific behaviors are exhibited in any object. However, site-specific attributes are present, and are derived based on the local database contents.

NameService

The OpenORB NameService module provides an Object Management Group (OMG) compliant implementation of the NameService Specification Version 1.2 (September 2002). This module is required for CORBA operations on the Cisco BTS 10200 Softswitch. Clients attach to the NameService to obtain the references to the Cisco BTS 10200 Softswitch CORBA objects through the corbaloc process. When using OpenORB on the client side of the application, apply the following syntax to connect to the NameService:

```
"corbaloc::1.2@<Host Name>:14001/NameService"
```

The *corbaloc* string can be supplied in the OpenORB.xml configuration file located at */opt/BTSorb/config* directory as an initial reference or it can be dynamically built in the client application as required. Note that Cisco recommends that the hostname be an IP address.

Each Cisco BTS 10200 Softswitch EMS comes with management interfaces, and the INS or NameService must utilize a hostname that resolves across both of these management interfaces. This is required. But client side access does not care which interface is utilized. The client must be aware that a given management interface can be down for various reasons and that this can impact access to the name service. A recommendation to utilize both the IP address or retries on the hostname may be required to deal with switch or router troubles that may naturally occur over time on any given subnet.

Each Cisco BTS 10200 Softswitch comes with its own pair of duplex INS. Each INS represents the objects from a single side of the Cisco BTS 10200 Softswitch. Use this resource location string to derive a reference to the NameService. Each Cisco BTS 10200 Softswitch comes with its own instance of a name service, and a name service can be utilized separately for each EMS. The Cisco BTS 10200 Softswitch default UDP port for this module is 14001. In the OpenORB model, this value is passed in the configuration file OpenORB.xml. The Software Development Kit (SDK) contains examples of this configuration, as well as example code for building working examples using the OpenORB client implementation.

Multiple NameService modules can be used by applying a request interceptor. A proxy object allows a request to be forwarded using the ForwardRequest (CORBA 3.0 spec. 1.3.1) protocol. See [Chapter 4, "Proxy"](#) for more information.

The following example shows an object resolution using the NameService module. Note that at this time the basic POA is used as a root-level reference to the local Cisco BTS 10200 Softswitch.

```
org.omg.CORBA.Object initial_context_obj =
    objOrb.resolve_initial_references("NameService");
objContext =
    org.omg.CosNaming.NamingContextExtHelper.narrow(initial_context_obj);
result = objContext.resolve(objContext.to_name("Bts10200_poa"));
objBts = Bts10200Helper.narrow(result);
```

BTS 10200 Softswitch IDL

The IDL is used to express the object-level interface in the CAD interface. This object interface includes the attributes and behaviors of the objects. This section provides an overview of the IDL for the Cisco BTS 10200 Softswitch. These IDL objects define access to the XML descriptions and documents used to provision the Cisco BTS 10200 Softswitch. A full description of the XML document is covered in a later chapter. For the most part, CORBA acts as the transport for these XML documents.

The `bts10200.idl` file contains the general system-wide data structures and type definitions. It also contains the error interfaces (exceptions). See the [“Cisco BTS 10200 Softswitch IDL Code” section on page 2-6](#) for the full text of the `bts10200.idl` file. This file contains all objects that are defined for use in the Cisco BTS 10200 Softswitch. The breakdown of each object is listed below.

- **Bts10200_Security**—The primary security object. It is used to create login keys for use in another object. This object is required to access the Cisco BTS 10200 Softswitch.
- **Bts10200**—The basic object used to retrieve XML description documents as well as provisioning and control documents.
- **CadException**—The object used to report all errors in the Cisco BTS 10200 Softswitch CAD interface.
- **Macro**—This object defines and executes custom *show* or *display* commands on the Cisco BTS 10200 Softswitch. This allows a user to create simplified display commands from complex relationships and permanently store them for recall later as “macro” commands.

Bts10200 API

This section covers the actual API calls to the CAD interface. The assumption is that the client application is developed in the Java language. This does not prohibit the use of C++. However, that is not within the scope of this document.

All parameters that are listed are required for each invocation of methods in the associated object.

Bts10200 Security API

The Cisco BTS 10200 Softswitch security object (`Bts10200_Security`) provides a user several levels of security for the CAD interface in the Cisco BTS 10200 Softswitch. It allows authorized users to obtain a security key and use this key for all future transactions. This object must be used prior to all other CORBA method invocations in the interface. This key is valid in the CAD interface for the life of the user's session. The key is no longer valid once the logout method has been invoked. Likewise, the security key expires after 10 minutes if the system has not been accessed during that period of time, and the user is automatically logged out of the CAD interface. The user name and password are the same values allowed in the CLI /MAC adapter interfaces, and the same authorization permissions apply.

Each method in this section is part of the `Bts10200_Security` interface. The parameters listed are required for each method and must contain data.

Login

The login method provides authentication of a CORBA interface user. It utilizes the same user security as the FTP or CLI adapters. This method returns a string value defined as a key. This key is required for all transactions against the CAD interface. It is an authentication key indicating the specific authorization of a particular user. The method signature is defined by the following code:

```
int login (java.lang.String user, java.lang.String passwd, java.lang.StringHolder key)
throws CadExceptions
```

- **Return value**—Status indicating success or failure of the operation. Failure means the facility is unavailable. Success means the operation was completed.
- **Exception**—A CadException means there is an operational error in processing the request. This includes faults with the parameter types, ranges, and database access.

**Note**

The Cisco BTS 10200 Softswitch implementation of CORBA does not support RADIUS authentication. Therefore, if a user presumes that an attempt to login is supported by a pluggable authentication module (PAM) and RADIUS authentication, the operation fails and the user is required to accomplish a UNIX login. In such cases, attempts to login will fail.

Logout

The logout method terminates a login session. This destroys the validity of the authentication key. Once this method is complete, the key can no longer be used for other method invocations. The method signature is defined in the following code:

```
int logout (java.lang.String) throws CadExceptions
```

- **Return value**—Status indicating success or failure of the operation. A failure indication means the facility is unavailable. A successful return indicates the operation was completed.
- **Exception**—A CadException means there is an operational error in processing the request. This includes faults within the parameter types, ranges, and in database access.

Bts10200 Provisioning API

The Cisco BTS 10200 Softswitch object (Bts10200) provides provisioning interface functions to the Cisco BTS 10200 Softswitch CLI engine for authorized users. Both input and output are in XML as described in [Chapter 2, “Extensible Markup Language Processing.”](#) The CLI commands are parsed into an XML document before sending the commands through the CORBA interface. The CORBA CIS server then executes the CLI provisioning commands and sends back the reply in an XML document. Each method in this section is part of the Bts10200 interface. The parameters listed are required for each method and must contain data.

getCommandDoc

The `getCommandDoc` method provides command description retrieval. This method obtains the XML document describing the command syntax and options for a specific noun/verb combination. The method signature is defined by the following code:

```
void getCommandDoc (java.lang.String noun, java.lang.String verb, java.lang.String key,
org.omg.CORBA.StringHolder reply) throws CadExceptions
```

where:

- *parameter noun* is the command noun
- *parameter verb* is the command verb
- *parameter key* is the authorization key obtained in login
- *parameter reply* is the XML reply of the command syntax and options
- *exception (cadexception)* means there is an operational error in processing the request. This includes faults with the parameter types, ranges, and database access.

request

The `request` method processes an XML document based provisioning request through CORBA interface. The CORBA CIS server executes the provisioning commands and sends back the reply in an XML document. The method signature is defined by the following code:

```
void request (java.lang.String command, java.lang.String key, org.omg.CORBA.StringHolder
reply) throws CadExceptions
```

where:

- *parameter command* is the provisioning command request in the XML document
- *parameter key* is the authorization key obtained in login
- *parameter reply* is the XML reply of the command execution in the CLI engine
- *exception (cadexception)* means there is an operational error in processing the request. This includes faults with the parameter types, ranges, and database access.

Macro Command

The Cisco BTS 10200 Softswitch requires the ability to provide a high-level view of complex data in its database. Normally, determining relationships between database items requires several commands and multiple requests to the database. This slow and costly process impedes the progress of Operations Support System (OSS) management systems. Therefore, an optimized approach is required where several operations can be collapsed into a single request to the database, which returns the correct related data based on the set of defined rules.

The ability to view complex data relationships in the Cisco BTS 10200 Softswitch is referred to as a macro. A macro is a single command that builds complex queries across multiple commands by using relationship rules defined by the user.

The Macro command is specific to the Cisco BTS 10200 Softswitch and works with both simplex and duplex configurations. The primary focus of this command in this document is its utilization in the CORBA interface definition language (IDL) interface of the CORBA Adapter (CAD) feature.

**Note**

The Macro command was available in Release 3.5. It was not available in Releases 3.2 and 3.3.

Behaviors and Attributes

The Macro command interface allows users to select and define multiple tables against the Cisco BTS 10200 Softswitch database. Normal commands only operate on single devices and/or tables.

The Macro command interface does not allow users to write to the tables in the database. There are many rules and constraints that apply to the database tables that prevent this activity.

Macro Command Management

Macro command management is composed of the user commands that create, change, and delete Macro command definitions. These user commands allow the definition and manipulation of the actual Macro commands and execute as standard provisioning commands. The other primary component is the macro execution. This is provided through the CORBA interface and uses the macro command management rules with additional user-specified data to return the instance values of the macro parameters.

Macro Definition

The values that are used to build a Macro command are validated internally. Technically, each macro is a superset **show** command of multiple nouns and their associated parameters. All values used in the creation of the Macro command are derived from the parameters of nouns and not from the actual table and column names. This helps to preserve the abstraction over the Cisco BTS 10200 Softswitch schema. The following examples of Macro command management show typical creation, alteration, and deletion of a Macro command. The sections following the examples define the actual values used in the macro definition and the constraints imposed on them.

```
add macro id=CTXG_NUMBERS; \
  parameters=office_code.NDC, \
             office_code.EC, \
             dn2subscriber.DN, \
             subscriber.CTXG_ID; \
  rules="office_code.OFFICE_CODE_INDEX=\
        dn2subscriber.OFFICE_CODE_INDEX, \
        dn2subscriber.SUB_ID=subscriber.ID";
```

When editing an existing macro, you must enter the parameter to be modified. In this case, the *parameters* and *rules* (and, or) are a list. The entire list must be reentered. The list that is stored in the macro database entry is then replaced. The following example demonstrates this.

```
change macro id=CTXG_NUMBERS; \
  parameters= office_code.NDC, \
             office_code.EC, \
             dn2subscriber.DN, \
             subscriber.ID;
```

When a macro is no longer required, it can be removed. This is achieved through the **delete macro** command. The only required parameter is the macro ID. All associated definitions for the macro are then removed from the database. The **delete macro** command takes following format:

```
delete macro id=CTXG_NUMBERS;
```

To display a macro, only the macro ID is required. All static components of the command are returned. The following example shows the format of this command.

```
show macro id=CTXG_NUMBERS;
```

Macro ID

The macro ID is used as the handle for all references to a macro definition. If the macro is to be invoked, the ID becomes the noun by which the macro can be invoked. The ID is also the primary key or parameter used in the **change macro, delete macro, or show macro** commands for editing a particular definition. The ID is a character field that must be unique to all other macros. It can be up to 79 characters long. The macro ID provides a unique macro definition that can also be verbose enough to describe the desired operation.

Parameter List

The parameter list is used to list the nouns and parameters that are displayed in the Macro command. They constitute the selected items to place in the reply. The value of the parameter list field is designed to be a comma-separated list of nouns and parameters from related Cisco BTS 10200 commands. This list has the following form as input:

```
parameters=<noun.parameter>,<noun.parameter>,...,<noun.parameter>;
```

Each item in the list takes the form of the noun with a period followed by the parameter from that noun. These nouns and parameters are validated through the Element Management System (EMS). Each item in the list must always be addressed in full with the noun and parameter. This is due to the reuse of common parameter names such as ID.

No implied order of importance is given to the parameters. They are supplied to the command processing in the order they are defined in the macro. No additional data items from other parameters of a given noun are included.

Rules

The most critical parts of the Macro command are the rules. These rules manage the relationship of the data to be selected from the Cisco BTS 10200 Softswitch database. These rules amount to the “where” clause of a database selection statement. They help focus the data subset required for operations. There are two basic sets of rules that can be applied to a macro: the **and** rules and the **or** rules as well as **equivalence** and **not**. The generic parameter rules are applied as **and** rules. The definitions of these rules are:

- **And rules**—One or more sets of data qualifiers indicating a required conditional for the selection of data in the macro. A single **and** rule specifies that a particular noun and parameter must equal some other specific noun and parameter.
- **Or rules**—One or more sets of data qualifiers indicating an optional conditional for the selection of data in the macro. A single **or** rule specifies that a particular noun and parameter can equal some other specific noun and parameter.

- **Equivalence and Not**—The rules section of the Macro command can contain two variations. They can describe a noun/parameter pair as equivalent or not equivalent. The common syntax for this expression is “=” for equal, or “!=” for not equal.

User Input

The user input component is not required for the rules list. This is supplied at the time the macro is invoked. This additional input is treated as an **and** rule input. This data is intended to be the qualifying data that defines what subset of data to select. This rule data can use the equivalence syntax to express variations in the data selection. For example, a user may want to find all subscribers that do not have a specific feature.

CORBA/XML Interface

The CORBA interface servant (CIS) must support a new IDL interface for the Macro command interface. This involves the use of the following new components. The CIS subsystem supports the definition and execution of the Macro commands.

- **Macro command definition**—Macros that can be defined through the “Bts10200.request(...);” interface as normal provisioning requests are managed.
- **IDL interface definition**—A new IDL method is added to access a Macro command that is separate from the standard feature-provisioning interface. This is for the execution of the macro only.
- **CIS implementation of the IDL interface**—An implementation that follows the standard behavior of other CIS interface objects by accepting strings for the authorization key and input arguments. It also returns a string to indicate the response to the macro execution. The arguments and format are described as follows:
 - **Request**—The XML Request document that contains the additional user-supplied rules for the request as well as the paging facility parameters. The XML Request document must also have the noun key set to the desired macro name to execute the request. At this time, the verb key is not used. As a default, *show* is the best option. This avoids future conflicts if writes are allowed.
 - **Key**—A simple string object that contains the actual authentication key provided through the Bts10200_Security interface.
 - **Reply**—The XML Reply document that contains the returned data from the macro execution. This follows the same format as the Reply XML document from the Bts10200 interface.

External Interfaces

This section details the extensions provided in the Cisco BTS 10200 software that are reflected in the external interface of the CORBA adapter (CAD) for the Cisco BTS 10200 Softswitch. The paging parameters are available for Macro command execution. These include the limit and start-row parameters.

CAD Interface

This section describes the new IDL method signature in the CAD interface.

```
interface Macro {

    //-----
    // Issue a Macro Command XML document
    //-----
    void          execute(in string          request,
                        in string          key,
                        out string         reply)
                    raises(CadExceptions);

}; // end Macro
```

Request Format

The following example shows a macro XML Request document, which requests execution of the CTXG_NUMBERS example that was defined in the “[Macro Definition](#)” section on page 1-13. The additional paging parameters are added. These limit the volume of data that is returned during any single reply. Large XML documents fail in the interface. The current default limit is defined as 500 records per request.

```
<Request Noun="CTXG_NUMBERS" Verb="show">
  <Entry Key=" subscriber.ctxg_id" Value="rcdn_grp"/>
  <Entry Key="limit" Value="1"/>
</Request>
```

Reply Format

The following is an example of a XML Reply document. This reply is based on the example in the Request Format section.

```
<Reply id="Reply">
  <Status>true</Status>
  <Reason>Success: Entry 1 of 5002 returned.</Reason>
  <Size>1</Size>
  <AbsoluteSize>5002</AbsoluteSize>
  <StartRow>1</StartRow>
  <DataTable>
    <Row id="0">
      <Column id="NDC">601</Column>
      <Column id="EC">227</Column>
      <Column id="DN">1013</Column>
      <Column id="CTXG_ID">rcdn_grp</Column>
    </Row>
  </DataTable>
</Reply>
```

Operations

This section describes changes to the operations user interface as a result of the Cisco BTS 10200 Macro command. The standard paging parameters are available for the Macro command execution. These apply to large data sets.

You can use the operator interface for additional commands to manage the Macro command in the Cisco BTS 10200 Softswitch. These commands are available from the CLI interface. In addition, these same commands are also available from the CORBA and bulk-provisioning interface.

[Table 1-1](#) shows the user commands that can be generated. [R] in the table indicates *required*. These are user-defined commands that have a variety of noun (id) and parameter combinations.

Table 1-1 User-Defined Macro Commands

Noun	Verb	Options	Description
macro	add	id [R]	Identifier for the macro. The identifier can be from 1 to 79 characters. It must be unique to all other IDs.
macro	add	parameters [R]	Comma-separated list of the actual data to return as a result of the macro execution.
macro	add	rules [R]	Criteria for the display of data in the macro.
macro	change	id [R]	Identifier for the macro.
macro	change	parameters	Comma-separated list of the actual data to return as a result of the macro execution.
macro	change	rules	Criteria for the display of data in the macro.
macro	show	id [R]	Defines the macro to be displayed.
macro	delete	id [R]	Identifier for the macro.

Cadexceptions

The following basic cadexceptions can be returned. The numbers given in the sample code refer to the text in the explanation. The text is returned if the sample code is used. See the [Cadexceptions](#) section in [Chapter 5, “Troubleshooting”](#) for recommended actions if necessary.

Error Message No Error

Explanation This is a placeholder since zero is not an error.

Sample Code `public static final int EM_NONE=0;`

Error Message CIS Error

Explanation This error is used for internal processing errors that may relate to ORB interaction or other runtime exceptions.

Sample Code `public static final int EM_ERROR=1;`

Error Message CIS No Data

Explanation This error indicates that there was no data to return from a **show** command. This may not be a “real” error; however, it is cleaner to throw an exception than a NULL object.

Sample Code `public static final int EM_NODATA=2;`

Error Message User Security Error

Explanation A fault was found in the user security. This could result from an invalid login through a password or username. This could also result from an internal error in the security system that failed to validate the user identity.

Sample Code `public static final int EM_USERSEC=5;`

Error Message Permission Error

Explanation A command was attempted that failed the authorization tests for that command. The user does *not* have permission to execute this command.

Sample Code `public static final int EM_PERMISSION=6;`

Error Message Error Message: Block Error

Explanation All provisioning on the switch has been blocked. The command may have been perfectly well formed and the connection is still valid. This just indicates the BTS 10200 is in a maintenance mode.

Sample Code `public static final int EM_BLOCK=7;`

Error Message Linkage Error

Explanation The linkage failed.

Sample Code `public static final int EM_LINKAGE=10;`

Error Message Exception In Initializer Error

Explanation The initialization provoked by this method fails.

Sample Code `public static final int EM_INIT=11;`

Error Message Class Not Found Exception

Explanation The class cannot be located.

Sample Code `public static final int EM_NOTFOUND=12;`

Error Message Illegal Access Exception

Explanation The class or initializer is not accessible.

Sample Code `public static final int EM_ACCESS=13;`

Error Message Instantiation Exception

Explanation The class represents an abstract class, an interface, an array class, a primitive type, or void; or if the instantiation fails for some other reason.

Sample Code `public static final int EM_INSTANCE=14;`

Error Message Security Exception

Explanation There is no permission to create a new instance.

Sample Code `public static final int EM_SECURITY=15;`

Error Message Invalid Request Exception

Explanation The request is not valid or cannot be initialized.

Sample Code `public static final int EM_REQUEST=16;`

Error Message Invalid Noun Exception

Explanation The command noun is not found or is invalid.

Sample Code `public static final int EM_NOUN=17;`

Error Message Invalid Verb Exception

Explanation The command verb is not found or is invalid.

Sample Code `public static final int EM_VERB=18;`

Error Message SQL Exception

Explanation The database cannot be accessed, if the constraints are violated, if there is another table conflict, if there is a resource issue, or any other Oracle-related cause.

Sample Code `public static final int EM_DATABASE=19;`

Error Message Invalid Value Exception

Explanation A parameter value exceeds the valid range or some other restrictions like text length, pick-list, and so forth.

Sample Code `public static final int EM_VALUE=20;`

Error Message Invalid Key Exception

Explanation An invalid key or token was used to describe some data value.

Sample Code `public static final int EM_KEY=21;`

Error Message Missing Parameter Exception

Explanation One or more required parameters were not included in the command parameter data.

Sample Code `public static final int EM_PARAM=22;`