OBJECTSTORE

COMPONENT SERVER FRAMEWORK USER GUIDE

RELEASE 5.1

March 1998

ObjectStore Component Server Framework User Guide

ObjectStore Release 5.1 for all platforms, March 1998

Object Design, Inc.
Twenty Five Mall Road
Burlington, MA 01803-4194

Part number: SW-OS-DOC-CSF-510

Contents

Preface

Purpose

The *ObjectStore Component Server Framework User Guide* describes how to use the Component Server API, the thin client API, and the Component Server executable to build scalable, multitier ObjectStore applications that make optimal use of ObjectStore's *Cache-Forward* architecture. This book supports ObjectStore Release 5.1.

Audience

This book assumes the reader is experienced with C++.

Scope

Information in this book assumes that ObjectStore and the Component Server Framework are installed and configured.

How This Book Is Organized

The book begins with a chapter that introduces the Framework APIs and the Component Server executable. The chapter also contains a section on using OLEDB and ADO instead of the thin client API. The last three sections of the chapter provide detailed information on building and deploying thin clients and components, as well as on starting and configuring the ObjectStore Component Server.

The next two chapters provide overviews of the Server and client APIs. The two chapters after that provide a basic example and an intermediate example, respectively. The next chapter discusses using ObjectStore Inspector data views together with the Framework.

The subsequent several chapters provide references on each class in the Server and client APIs. The last chapter provides an exception reference.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
Bold	Bold typeface indicates user input or code.
Sans serif	Sans serif typeface indicates system output.
Italic sans serif	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.
Italic serif	In text, italic serif typeface indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ *a* | *b* | *c* }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.
	Indicates that the operating system named inside the circle supports or does not support the feature being discussed.

ObjectStore C++ Release 5.1 Documentation

The ObjectStore Release 5.1 documentation is chiefly distributed on line in Web-browsable format. If you want to order printed books, contact your Object Design sales representative.

Your use of ObjectStore documentation depends on your role and level of experience with ObjectStore. You can find an overview description of each book in the ObjectStore documentation set at URL **http://www.objectdesign.com**. Select **Products** and then select **Product Documentation** to view these descriptions.

Internet Sources of More Information

World Wide Web

Object Design's support organization provides a number of information resources. These are available to you through a web browser such as Internet Explorer or Netscape. You can obtain information by accessing the Object Design home page with the URL **http://www.objectdesign.com**. Select **Technical Support**. Select **Support Communications** for detailed instructions about different methods of obtaining information from support.

Internet gateway

You can obtain such information as frequently asked questions (FAQs) from Object Design's Internet gateway machine as well as from the Web. This machine is called **ftp.objectdesign.com** and its Internet address is 198.3.16.26. You can use **ftp** to retrieve the FAQs from there. Use the login name **odiftp** and the password obtained from **patch-info**. This password also changes monthly, but you can automatically receive the updated password by subscribing to **patch-info**. See the **README** file for guidelines for using this connection. The FAQs are in the subdirectory **./FAQ**. This directory contains a group of subdirectories organized by topic. The file **./FAQ/FAQ.tar.Z** is a compressed **tar** version of this hierarchy that you can download.

Automatic email notification

In addition to the previous methods of obtaining Object Design's latest patch updates (available on the **ftp** server as well as the Object Design Support home page) you can now automatically be notified of updates. To subscribe, send email to **patch-info-request@objectdesign.com** with the keyword **SUBSCRIBE patch-info <***your siteid***>** in the body of your email. This will subscribe you to Object Design's patch information server daemon that automatically provides site access information and notification of other changes to the online support services. Your site ID is listed on any shipment from Object Design, or you can contact your Object Design Sales Administrator for the site ID information.

Training

If you are in North America, for information about Object Design's educational offerings, or to order additional documents, call 781.674.5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time.

If you are outside North America, call your Object Design sales representative.

Your Comments

Object Design welcomes your comments about ObjectStore documentation. Send your feedback to **support@objectdesign.com**. To expedite your message, begin the subject with **Doc:**. For example:

Subject: Doc: Incorrect message on page 76 of reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
Introduction

The ObjectStore Component Server Framework makes it easy to build component-based applications that make optimal use of ObjectStore's *Cache-Forward* architecture. The Framework provides an API and a Server executable that allow you to build, deploy, and administer highly scalable components and thin clients.

This chapter introduces you to the following topics:

There is also detailed information on the following:

Taking Advantage of the *Cache-Forward* Architecture

ObjectStore is designed to maximize performance and ease of development for distributed, component-based applications by supporting

- Management of data as C++ or Java objects
- Delivery of data from database storage to application components (ObjectStore client shared libraries)
- Automatic maintenance of local, transaction-consistent caches associated with each component, which act as in-memory databases

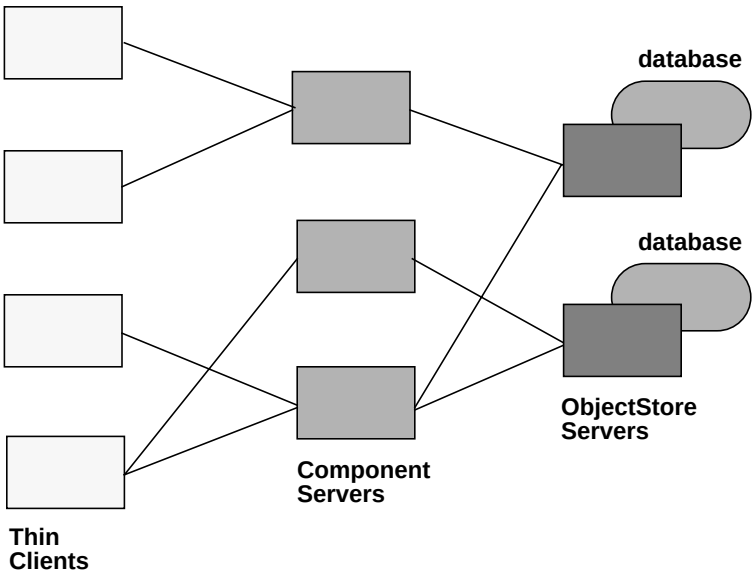
With many applications, using the component caches most effectively means allowing each end user to use many caches, as well as allowing many end users to use the same cache.

That way requests from different end users that require similar data sets can be routed to the same cache. This minimizes invalidation of other component caches, reducing network traffic due to callbacks and refetched data.

Moreover requests from the same end user that require different data sets can each be routed to different caches. Each request can be handled by a cache that is likely to contain the necessary data already. This means most data access can take place at in-memory speeds, with no database Server overhead.

This requires a multitier architecture in which thin client processes send requests to component servers, which service requests by using components in the form of shared libraries. These servers, in turn, are clients of ObjectStore database Servers, which maintain cache validity, propagate updates to persistent storage, and service any component requests for additional data.

Multitier Architecture for ObjectStore Applications



For web-based applications, thin clients run on the web server host, either as a CGI executable or as a shared library executing in the web server process space (for example as an NSAPI or ISAPI shared library).

The ObjectStore Component Server Framework provides a Component Server executable for which you can build thin clients and components. You can build thin clients with the Framework's thin client API, or with OLE DB or ADO. You can build components with the Framework's Server API together with the ObjectStore C++ API.

The ObjectStore Component Server

The Component Server Framework includes the ObjectStore Component Server, which handles interaction between thin clients and ObjectStore components.

Each Component Server process supports a named *service*. A service is a collection of *components*, that is, shared libraries that act as plug-ins to the Server. Each component, in turn, supports one or more operations.

The Component Server supports high throughput and scalability by providing

- Flexible transaction management: Servers can execute requests using a variety of transaction types. With most transaction types, the Server batches requests from different users into the same transaction, reducing commit overhead and increasing throughput.
- Multiple-thread dispatching: Each Server handles requests using multiple threads, so requests share resources more efficiently and inexpensive requests do not have to wait for more costly ones.

Load balancing: To scale your application as the number of requests increases, you can configure Servers to use more threads. As the number of requests increases further, you can replicate Servers as many times as necessary for the required performance. One Server acts as router and distrubutes requests to Servers on a round-robin basis.

The Component Server is an ObjectStore client, that is, a client of ObjectStore database Servers.

You can start and stop ObjectStore Component Servers from the command line, or you can create a customized component server that manages start-up and shutdown of ObjectStore Component Servers.

Most applications use multiple Servers. Each Server is dedicated to a subset of the operations supported by a service. In many cases, the data set required by those operations can fit entirely, or almost entirely, into the Server's cache (which is an ObjectStore client cache).

When you start or configure a Server, you specify the following parameters, which are related to routing client requests to the appropriate Server:

- Transaction type: Each Server runs a specific type of transaction, **isolated_update**, **shared_update**, **read**, or **mvccRead**. Each operation is implemented to use one of these transaction types. Requests for execution of given operation are routed to a Server running that operation's type of transaction.
- Optional routing data: This is a pointer to any user-specified data, such as a character string. A thin client can specify routing data along with the request, and the request is routed to a Server with matching routing data.

Because clients can specify routing data on a per-operation basis, clients can route requests based on operation arguments. For example, an application that operates on bank accounts can route operations based on account number. That way, different Servers can be dedicated to the data for different ranges of account numbers. This allows you to scale the application by adding more Servers as the number of accounts increases, and keep the application operating almost entirely on hot caches.

You can manually warm up each cache when you initialize the application, or you can let thin client requests populate the caches incrementally. In the latter case, because of ObjectStore's *Cache-Forward* architecture, each cache is automatically populated with the desired data set by virtue of the routing that takes place. In both cases, ObjectStore automatically maintains the validity of all Component Server caches.

Using multiple Servers can also increase the fault tolerance of your application. By replicating Servers, including router Servers, you can easily avoid single-point-of-failure vulnerability. In addition, routers maintain route data persistently, which also increases applications' fault tolerance.

The Thin Client API

The client API provides C++ functions for connecting to a service and sending requests. Requests take the form of operations named and defined by the Server.

Before sending a request, the client retrieves an operation object from the Component Server, and sets its arguments using one of the following types of value for each argument:

- Integer type
- Floating numerical type
- Character string
- Object ID
- Bit stream

After a client request is serviced, the client receives the results in the form of a list of **ostc_Object**s, which are essentially lists of attribute / value pairs. The API provides functions for retrieving attribute values, which are each values of one of the preceding fundamental types.

If the number of objects to be returned is large, the client can specify that the results be returned incrementally. The incremental nature of the results is transparent to the client, since the usual API functions for retrieving list elements automatically trigger a Server interaction whenever a new batch of results must be sent.

The client maintains a cache of object lists returned by the Server. When you issue a request, you can specify that the results should be obtained from the cache, if possible, instead of from the Server.

The client API is entirely independent of ObjectStore. You can use the API to build an independent executable, for example a CGI executable, or to build a plug-in, for example an NSAPI or ISAPI plug-in.

For more information on the Client API, see the Thin Client API Overview on page 19.

The Component Server API

You use the Server API, together with the ObjectStore C++ API, to build components (that is, plug-in shared libraries) that can be loaded by an ObjectStore Component Server.

You build each component by defining a class for each operation you want the component to support, and implementing certain functions. Some of the API functions specify the protocol for functions you must implement. You call other API functions from these implementations.

Among the functions you must implement are **execute()** member functions for each operation the component supports. An **execute()** function retrieves the arguments sent by the client and uses them for ObjectStore-dependent processing. It then specifies the results as one or more persistent or transient C++ objects.

Another function you must implement for each operation is the **format()** member function. When **execute()** completes, the Server calls **format()** on each return value to convert it into one or more **ostc_Object**s. All the resulting objects are returned to the client.

Beginning and ending transactions is handled automatically by the Server, so you do not have to write transaction management code.

The Server API also provides an alternative to supporting an operation by defining a class and implementing its members. You can support an operation by designating (with a Server API function) an ObjectStore Inspector data view. When you do this, the component automatically supports an operation that returns an **ostc_Object** for each row of the data view. See Using and Configuring the DataView Reader on page 59.

For more information on the Server API, see the Component Server API Overview on page 29.

The Client and Server API Classes

Below is a table that shows the classes in the client API, the classes in the Server API, and the classes in both APIs.

Thin Client	*Both Client and Server*	*Component Server*
ostc_Session	**ostc**	**ostc_ServerSession**
ostc_Operation	**ostc_Object**	**ostc_ServerOperation**
ostc_OperationSet	**ostc_ObjectList**	**ostc_OperationResult**
	ostc_AttributeDescriptor	**ostc_ApplicationServer**
	ostc_AttributeDescriptorList	
	ostc_OID	

OLE DB and ADO Clients

On Windows platforms, you can create a client of the ObjectStore Component Server using the standard interfaces provided by OLE DB and ADO. When you install the Component Server Framework, the installation program automatically registers the Component Server as a version 1.5 OLE DB provider.

Operations defined with the Server API can be executed in OLE DB or ADO as commands, and lists of OSTC objects returned by Server operations can be accessed as Rowsets or RecordSets.

What Is OLE DB?

OLE DB is Microsoft Corporation's component database architecture, which is the fundamental component object model (COM) building block for storing and retrieving records. This set of interfaces for accessing and manipulating data provides data integration regardless of data type. Microsoft's Open Database Connectivity (ODBC) data access interface is included in the OLE DB architecture, and continues to provide access to relational data.

You can access OLE DB providers through ActiveX Data Objects (ADO). ADO is a high-level object model that provides access to any OLE DB source. The ADO interface is similar to the Data Access Objects (DAO) interface and the Remote Data Objects (RDO) interface.

For details about the OLE DB object model, refer to the Microsoft OLE DB SDK. To obtain a complete description of the ADO object model, consult the on-line documentation for Microsoft Visual Basic, Active Server Pages, or Visual C++. For more information about OLE DB and ADO, browse the Microsoft web site at **http://www.microsoft.com/data**.

Using ADO to Create a Client of the Component Server

The examples included with the Framework installation show how to access ObjectStore Thin Client OLE DB through ADO from a Visual Basic or ASP / VBScript application.

In general, to open an OSTC OLE DB connection, provide a string in this format:

provider=ObjectStore Thin Client OLE DB Provider;
data source=*service-name***;**
[**router=***router_name***:***port_number*]

where

- **ObjectStore Thin Client OLE DB Provider** is the name under which the Component Server is registered as an OLE DB provider. The Framework installation procedure performs this registration automatically.
- *service-name* is the name of the **ostc** service you want to use (as specified by **-service_name** or **ostc_ ServerSession::setServiceName()**).
- **router** specifies the full router associated with a Server running the service.

For example, using ASP and ADO you can create and open an OLE DB connection to the provider:

```
Set adoConnection = Server.CreateObject("ADODB.Connection")
Call adoConnection.Open(
    "provider=ObjectStore Thin Client OLE DB Provider; \
    data source=service-name"
)
```

Then you can open a RecordSet:

```
Set RS = Server.CreateObject("ADODB.RecordSet")
Call RS.Open("operation-name", adoConnection)
```

where *operation-name* is the name of a Server operation exposed by the *service-name* service, running on the default router (as specified in the registry).

If the operation requires arguments, before creating a RecordSet ADO object, you should create a Command object and then obtain a RecordSet from the Command:

```
Set aCommand = Server.CreateObject("ADODB.Command")
Set aCommand.ActiveConnection = adoConnection
Command.CommandText = "operation-name"
aCommand.Parameters("arg-name") = arg-value
Set RS = aCommand.Execute
```

What Parts of ADO Are Supported?

The provider supports the following schema tables:

- **DBSCHEMA_TABLES** (**adSchemaTables**)

- **DBSCHEMA_COLUMNS** (**adSchemaColumns**)
- **DBSCHEMA_PROVIDER_TYPES** (**adSchemaProviderTypes**)
- **DBSCHEMA_PROCEDURES** (**adSchemaProcedures**)
- **DBSCHEMA_PROCEDURE_PARAMETERS** (**adSchemaProcedureParameters**)
- **DBSCHEMA_PROCEDURE_COLUMNS** (**adSchemaProcedureColumns**)

Schema tables can be opened by the **ADODB.Connection.OpenSchema** method, using the enumerator specified in the parentheses.

The following table shows which data types are supported, together with the corresponding OSTC types:

ADO Enumerator	***Corresponding OSTC** Enumerator*
adBSTR	**ostc::string**
adInteger	**ostc::int32**
adBigInt	**ostc::int64**
adSingle	**ostc::float**
adDouble	**ostc::double**

Use of ADO is subject to the following restrictions:

- Server operations can perform updates, but updates are not supported with ADO methods such as the **AddNew**, **UpdateBatch**, **CancelUpdate**, **Delete**, and **Update** methods of **ADODB.Recordset**. Attempts to use these methods result in exceptions.
- Manual transaction handling is not supported (so **ADODB.Connection** methods like **BeginTrans**, **CommitTrans**, **RollbackTrans** are not supported).
- Timeouts are not supported.
- Commands cannot be prepared.

Accessing the ObjectStore Thin Client OLE DB Source Through ODBC

Since it is possible to build a bridge between any OLE DB provider and ODBC, you can use ODBC to create clients of ObjectStore Component Servers.

ISG Navigator/Bridge includes an ODBC driver for OLE DB data sources. It consumes OLE DB providers and allows ODBC applications to access any OLE DB data source. ISG Navigator/Bridge is distributed by ISG International Software Group (http://www.isgsoft.com) and by Microsoft Corporation(http://www.microsoft.com/oledb/download/download.htm).

Download and install ISG Navigator/Bridge. Then you can configure it to access Component Servers through any ODBC client.

Retrieving Error Information

When an ADO method that accesses the Component Server causes an error, ADO returns a standard ADO error code or generates the error code E_FAIL, #80040005. The provider uses the error code to identify its internal error conditions. In either case, you can retrieve extended information about the generated error.

For example, this code attempts to open a database in an active server page:

```
On Error Resume Next
Set adoConnection = Server.CreateObject("ADODB.Connection")
Call adoConnection.Open(
    "provider=ObjectStore Thin Client OLE DB Provider;\
    datasource=testService","",""
)
if Err.Number <> 0 then
    Response.Write("Error# " & Hex(Err.Number) & "<BR>")
    Response.Write("Generated by: " & Err.Source & "<BR>")
    Response.Write("Description: " & Err.Description & "<BR>")
    Exit Sub
End If
```

If you are accessing OLE DB directly (that is, without using the ADO layer), you can retrieve information about the errors generated by the provider. Refer to Microsoft's OLE DB documentation for details.

Building the Component Shared Library and Client Shared Library or Executable

The following tables show what static libraries and shared libraries to use when building thin clients and components.

Thin client programs must include the header file **ostc.h**.

Component programs must include **ostcsrvr.h**.

Libraries on WIndows Platforms

On WIndows, the Framework header files have **#pragma comment()**s that specify which **.lib**s to link with, so you do not have to mention them explicitly on your link lines. The **.lib**s do have to be in your **libpath**, however.

	Link with	***Put in path***
Thin Client	**ostc.lib**	**ostc.dll**
	oscs.lib	**oscs.dll**
Server DLL	**ostc.lib**	**ostc.dll**
	oscs.lib	**oscs.dll**
	ostcsrvr.lib	**oscsostore.dll**
	ostore.lib and any other ObjectStore 5.1 libraries the component requires	**ostcsrvr.dll**

For thin clients that use OLE DB or ADO, the following must be registered:

ostc.dll
oscs.dll
OSTCDB.dll
OSTCErr.dll
OSTCEnum.dll

If you use the DataView Reader, you must also include the following libraries in your path. Debug libraries are shown in parnetheses on the same line as the corresponding production library:

ostcDataViewReader.dll (ostcDataViewReaderd.dll)
mkivitos30.dll (mkivitos30d.dll)
osivitos30.dll (mkivitos30d.dll)

osischema30.dll (osischema30d.dll)
osidma30.dll (osidma30d.dll)
osi_osmm.dll
osi_osmm20.dll
osmmtype.dll
console.dll
LIBPL.DLL
LIBQP.DLL
QPENG.DLL
mfc42.dll (MFC42D.DLL)
MFC42D.DLL
mfcans32.dll
mfcuia32.dll
MSVBVM50.DLL
msvcrt40.dll
OC30.DLL
OLEAUT32.DLL
OLEPRO32.DLL
stdole2.tlb

Libraries on UNIX Platforms

Put the following libraries in your path as well as on your link line:

	UNIX, except HP–UX	***HP–UX***
Thin Client	**libostc.so**	**libostc.sl**
	liboscs.so	**liboscs.sl**
Server DLL	**libostc.so**	**libostc.sl**
	liboscs.so	**liboscs.sl**
	libostcsrvr.so	**libostcsrvr.sl**
	libos.a and any other ObjectStore 5.1 libraries the component requires	**libos.a** and any other ObjectStore 5.1 libraries the component requires

Build the Server shared library using ObjectStore 5.1. Be sure to include these measures:

- Generate a shared library schema by using the **OS_SCHEMA_DLL_ID()** macro in the library's schema source file.
- On Solaris and HP–UX, link with the ObjectStore library **libosthr**, *not* **libosths**.

Configuring, Starting, and Stopping the Server Executable

There are two ways to start an ObjectStore Component Server:

- From the command line, with the executable **ostccompsrvr**. The Server runs until the process is killed.
- From a program, with the function **ostc_ApplicationServer::start()**. The Server runs until stopped with **ostc_ApplicationServer::~ostc_ApplicationServer()**.

Using ostccompsrvr

The Component Server is **ostccompsrvr.exe** on Windows NT and **ostccompsrvr** on UNIX platforms. The Server executable takes the following command-line switches:

-plugin *library_name*	Plug-in library name. This parameter is required and must be the full library name. For Servers that support multiple plug-ins, specify this switch multiple times. **ostc_MissingLib** is thrown if the library could not be loaded. **ostc_RequiredParmMissing** is thrown if this switch is not supplied.
-service *component_name*	Logical component name used to match clients to Servers. This parameter is required. **ostc_RequiredParmMissing** is thrown if this switch is not supplied.
-port *port_number*	Port number of the Server. This parameter is required. **ostc_RequiredParmMissing** is thrown if this switch is not supplied.
-threads *number_of_threads*	Maximum number of threads that can concurrently handle requests in the Server. This parameter defaults to 1 for **isolated_update** Servers and 10 for all other Servers.

-txntype *txn_type*	Transaction type of operations supported by this Server. This parameter is required, and must be one of the following: **read**, **mvccRead**, **shared_update**, **isolated_update**, or **none**. Note that you must start at least one Server for each transaction type of operations supported by the specified components. **ostc_RequiredParmMissing** is thrown if this switch is not supplied.
-routedata *routing_string*	Routing string. If a client specifies a routing string when issuing a request, the request is routed to a Server with a matching string. The comparison is made with **memcmp()**.
-routerhost *host:port_number*	Network address for a system router. If this parameter is not specified, the Server being started acts as a router. Routers distribute requests among those Servers that have the appropriate service, transaction type, and routing data. You can supply this switch multiple times.
-routerfile *file_name*	File used by router to maintain routes persistently, in case of failures or restarts.
-inctimeout *timeout_in_seconds*	Timeout for access to incremental object sets on the Server. If the set is not accessed in the amount of time specified, the Server discards it. This parameter defaults to 1800 seconds.
-boundtimeout *timeout_in_seconds*	Timeout for access to a running bound transaction. If a bound transaction is started but is not being used, it will abort when this timeout is exceeded. This is used to keep the server from hanging if the client fails in the middle of a bound transaction. This parameter defaults to 10 seconds.
-sharedtimeout *timeout_in_seconds*	Timeout for access to a shared transaction. Shared transactions commit after a certain number of requests complete or this timeout is exceeded, whichever comes first. This parameter defaults to five seconds.

-sharedmaxops *number_of_ops*	Maximum shared operations. This sets the number of requests that can use a shared transaction before a commit occurs. This parameter is ignored for **isolated_update** Servers. This parameter defaults to 10.
-logging	Turns on Server logging. With logging on, the server writes information to the file **ostcserver.log** in the directory from which the Server was executed. Logging is off by default.

ostc_InvalidParm is thrown if a parameter name appears without an associated parameter value.

Example

```
ostccompsrvr.exe \
    -plugin my_plugin.dll \
    -port 2090 \
    -txntype read \
    -service test \
    -inctimeout 60
```

Server processes started from the command line run until you kill them with an operating system command.

Creating a Custom Component Server

You can create a custom component server that manages configuration, start-up, and shutdown of ObjectStore Component Servers with members of the class **ostc_ApplicationServer**.

Debug Version

ostccmpsrvrd is the debug version of **ostccompsrvr**. Use this when debugging components.

Deploying Thin Clients and Component Servers

When you deploy thin clients and component Servers, you need to ship the production version of all the libraries you used to build them.

If you are using OLE DB, you must also ship the following:

- **OSTCDB.dll**
- **OSTCErr.dll**
- **OSTCEnum.dll**

These are self-registering COM components. If you use Installshield, you need to unpack them using the **SELFREG** flag to **CompressGet**, or otherwise arrange to have **regsvr32** called for them.

If you have a link-compatible version of **osmmtype.dll** that is a later version than the one shipped with CSF, use that one instead.

For the DataView Reader, you also need to install the following run-time files in the windows system directory if the versions shipped with the Framework are more recent than those that have been installed already:

- **OC30.DLL**
- **OLEAUT32.DLL**
- **OLEPRO32.DLL**
- **mfcans32.dll**
- **mfcuia32.dll**
- **msvcrt40.dll**
- **stdole2.tlb**
- **MFC42.dll**

Chapter 2
Thin Client API Overview

The thin client API allows programs to perform the following main tasks:

Thin client programs must include the header file **<ostc.h>**.

Setting Routers

In any deployed configuration of Component Servers, one or more Servers act as routers and distribute requests among those Servers that have the appropriate service, transaction type, and routing data. You specify whether a Server is a router when you launch the Server processes (see Configuring, Starting, and Stopping the Server Executable on page 15).

Each client must specify one or more routers as well. You do this with the API **ostc::addRouter()**. On Windows platforms, you can use the executable **ostcSetRouter.exe** instead of this API.

Example

```
ostc::addRouter(argv[1]);
```

argv[1] is a character string specifying the host machine of the *router* Component Server, as well as the port on which it is listening for thin client requests.

The string has the syntax

host:*port*

When a client calls **ostc_Session::doOperation()**, a router tells the client which Component Server to send requests to. The router makes this determination based on

- The specified service name
- A round-robin schedule of Component Server processes that match the specified service name.

Click here for the example in context.

ostcSetRouter.exe

Windows only: With this executable you can add and remove routers:

ostcsetrouter -add *host*:*port*

ostcsetrouter -remove *host*:*port*

ostcsetrouter -clear

The **-clear** switch removes all routers.

Running this utility affects the Windows NT registry.

Connecting to a Service

Before using a component's operations, the thin client must first connect to a Server running a service that supports the component. You do this with the function **ostc::connect()**.

Example

```
_session = ostc::connect("quickstart");
```

Click here for the example in context.

Service Name

"quickstart" is the name of a service that supports the component or components you want to use during the current session. If there is more than one Server that is running the specified service, a router chooses a Server to connect to.

Return Value

_session, the return value, is a pointer to an instance of the class **ostc_Session**, which represents the thin client's current session with the specified service. You use members of this class to retrieve the operations supported by the Server's service, as well as to send operation execution requests.

Multithreaded Thin Clients

You should retrieve only one session per client thread. Session objects are cached, making **connect()** (and **disconnect()**) inexpensive operations.

Retrieving Operations from the Server

Once you connect to a service, you retrieve the operation or operations you want to perform, using **ostc_Session::getOperation()**:

Example

```
ostc_Operation * transfer = _session->getOperation("transfer");
```

"transfer" is the name of the operation you want to perform.

transfer is the operation, an instance of **ostc_Operation**.

You use **ostc_Operation::setArgument()** to set the arguments of the operation you want to perform. Once the arguments are set, you pass the instance of **ostc_Operation** to **ostc_Session::doOperation()** to execute the operation.

Click here for the example in context.

Setting Operation Arguments

The operation you retrieve using **ostc_Session::getOperation()** contains default value arguments; that is, every argument is set to 0. To set arguments to nondefault values, use **ostc_Operation::setArgument()**:

Example

```
transfer->setArgument("from_acct_name", from_name);
transfer->setArgument("to_acct_name", to_name);
transfer->setArgument("amount", amount);
```

"from_acct_name", **"to_acct_name"**, and **"amount"** are the names of the arguments being set.

from_name, **to_name**, and **amount** are the values to which the arguments are being set.

Click here for the example in context.

Argument Value Types

The value of an argument must be one of the following types:

- Integer type
- **float**
- **double**
- **char***
- **ostc_OID*** (object identifier set by user)
- **void*** (for bit vectors)

These are the fundamental types used to pass data between thin client and Component Server. There is an overloading of **set_argument()** for each of these types.

Getting Arguments

You can also use members of **ostc_Operation** to retrieve an operation's formal and actual arguments.

Executing Operations

Once the arguments are set, you execute the operation with **ostc_Session::doOperation()**:

Example

```
ostc_ObjectList * result = _session->doOperation(transfer);
```

transfer is the operation to be executed.

result is the a list of **ostc_Object**s that constitute the operation result. Each **ostc_Object** is made up of attribute / value pairs. The possible value types are the same as the possible argument types of an operation.

0 indicates that the operation should not use the thin client cache of operation results.

There are additional optional arguments that allow you to

- Specify routing data that determines which Server executes the operation
- Specify that the results should be returned from the Server in batches of a specified size

See **ostc_Session::doOperation()** on page 87 for more information.

Click here for the example in context.

Extracting Operation Results

Since operation results take the form of lists of **ostc_Object**s, you must extract the fundamental values making up the result by

- Retrieving **ostc_Object**s from the list, using **ostc_ObjectList::first()** and **ostc_ObjectList::next()**
- Retrieving the objects' attribute values, using members of **ostc_Object**

Example

```
ostc_Object * account = 0;
account = result->first();
printf("%-20.15s%15f\n",
    from_name, account->getFloatValue("val"));

account = result->next();
printf(
    "%-20.15s%15f\n",
    account->getStringValue("name"),
    account->getFloatValue("val")
);
```

Click here for the example in context.

Get-Value Functions

ostc_Object defines functions for getting attributes with values of each fundamental OSTC type:

- **ostc_Object::getInt32Value()** on page 115
- **ostc_Object::getInt64Value()** on page 115
- **ostc_Object::getStringValue()** on page 115
- **ostc_Object::getFloatValue()** on page 116
- **ostc_Object::getDoubleValue()** on page 116
- **ostc_Object::getObjectValue()** on page 116
- **ostc_Object::getBinaryValue()** on page 116

These functions return the attribute value. You can also use the various overloadings of **ostc_Object::getValue()**, which assign the value to a variable that you pass in.

Disconnecting from the Server

Call **ostc::disconnect()** to end a session.

Example

```
ostc::disconnect(_session);
```

Click here for the example in context.

Managing the Thin Client Cache

The client maintains a cache of object lists returned by **doOperation()** during the current session. **doOperation()** does not interact with a Server and instead returns a cached list if

- The **use_cache** argument to **doOperation()** is 1.
- The same operation with the same arguments was executed earlier in the current session.
- The returned list from that execution is still in the thin client's cache.

You can manage the contents of the cache with **ostc_Session::flushObjects()**.

Chapter 3
Component Server API Overview

To create a Component Server plug-in, you must implement the following global functions:

Function	*When Called by Server*	*Called Within an ObjectStore Transaction?*	*Task*
ostcInitialize()	When the Server starts	No	Application-specific initialization
ostcConnect()	After **ostcInitialize()**, when the Server starts	No	Adding Operations to a Server Session on page 36
ostcDisconnect()	When the Server shuts down	No	Application-specific cleanup

In addition, you must derive a type from **ostc_ServerOperation** for each operation you want the Server to handle. Each derived type must implement the following member functions:

Function	*When Called by Server*	*Called Within an ObjectStore Transaction?*	*Task*
constructor	Called by **ostcConnect()**	No	Specifying Server Operation Name, Transaction Type, and Formal Parameters on page 32
execute()	Whenever a thin client calls **doOperation()**	Yes	Implementing Server Operation Execution on page 38
format()	After **execute()** completes, called once for each return value	Yes, same transaction as **execute()** (except with incremental results)	Converting operation results into **ostc_Object**s

Your implementations of these functions (except **ostcConnect()** and **ostcInitialize()**) must be reentrant. That is, you are responsible for synchronizing access to any transient state that could be shared across Server threads. In addition, for operations that use **shared_update** transactions, you must synchronize access to persistent data if that is necessary to prevent interference among threads.

ostcConnect() and **ostcInitialize()** are called only when the Server is running a single thread, so they do not require synchronization.

Component Server programs must include the header file **<ostcsrvr.h>**.

Defining an Operation Subtype

For each operation you want a component to support, you must define a subtype of **ostc_ServerOperation** that defines a constructor, the members **execute()** and **format()**, and optionally the member **operationComplete()**.

Example

```
class txfer_operation : public ostc_ServerOperation
{
public:
    txfer_operation();
    ~txfer_operation() {}

    void execute(ostc_Object * Arguments, ostc_OperationResult*);
    void format(
        const void*,
        ostc_ObjectList*,
        ostc_OperationResult*
    );
};
```

You are free to define additional members of the subclass.

Click here for the example in context.

Specifying Server Operation Name, Transaction Type, and Formal Parameters

For each class you derive from **ostc_ServerOperation**, the constructor must do the following:

- Pass the operation name to the base type constructor.
- Pass the operation transaction type to the base type constructor.
- Specify the operation formal parameters and return type using **ostc_ServerOperation::addArgument()**.
- Specify the operation return type using **ostc_ServerOperation::addReturnSetAttribute()**.

Example

```
txfer_operation::txfer_operation() :
    ostc_ServerOperation("transfer", ostc::isolated_update)
{
    addArgument("from_acct_name", ostc::ostc_string);
    addArgument("to_acct_name", ostc::ostc_string);
    addArgument("amount", ostc::ostc_float);

    addReturnSetAttribute("val", ostc::ostc_float);
    addReturnSetAttribute("name", ostc::ostc_string);
}
```

Operation Name

"transfer" is the name of the operation. It must be passed to the base type constructor.

Transaction Type

ostc::isolated_update is the type of transaction to use to execute the operation. As with the operation name, it must be passed to the base type constructor. Use this type of transaction for operations that update persistent data. The possible transaction types are

- **read_only**
- **mvccRead**
- **isolated_update**

- **shared_update**
- **any**
- **none**

With **read_only**, **mvccRead**, and **shared_update**, requests from different thin clients are batched into the same ObjectStore transaction. This increases throughput by reducing commit overhead.

Important note

Use **shared_update** *only with extreme care.* Using **shared_update** inappropriately can cause incorrect results and database corruption.

With **any**, the router chooses a Server running any type of transaction. The operation must be read-only if there are **read** or **mvccRead** Servers running. The operation must be safe for batch transactions if there are **shared_update** Servers running.

An operations that uses the transaction type **none** must access no persistent data.

Formal Parameters

Each call to **addArgument()** specifies an argument name and type. These are the same names and types that the thin client must use when setting the actual arguments.

addReturnSetAttribute() specifies the attribute name and value type for an attribute of the returned objects. Call this function onece for each attribute you want the returned objects to have.

Click here for the example in context.

Type Enumerators

The types are specified with enumerators defined in the scope of the class **ostc**. The following table shows what type each enumerator signifies:

ostc_int32	**32-bit integer**
ostc_int64	**64-bit integer**
ostc_string	**char***
ostc_float	**float**
ostc_double	**double**

ostc_oid	**ostc_OID*** (object identifier set by user)
ostc_binary	**void*** (for bit streams)

Initializing a Component

For each component, you must implement the global function **ostcInitialize()**, which provides any initialization required before operations can be executed. Typical tasks for **ostcInitialize()** include opening databases and retrieving database roots.

The Component Server calls **ostcInitialize()** when the Server first starts up.

Example

```
void ostcInitialize()
{
   OS_ESTABLISH_FAULT_HANDLER

   _db = os_database::open(dbname,  0, 0664);

   OS_BEGIN_TXN(intialize,0,os_transaction::update)
   {
   // Look up the Account extent root
   os_database_root * root = _db->find_root("Account_root");

   ...

   OS_END_TXN(intialize)

   OS_END_FAULT_HANDLER
}
```

Click here for the example in context.

Adding Operations to a Server Session

For each component you must implement the global function **ostcConnect()**. Use the operation subtype constructors to construct each operation the component supports. Use **ostc_ServerSession::addOperation()** to register each operation. The Component Server calls **ostcConnect()** after **ostcInitialize()**, when the Server first starts up.

Note that for every Server running a given service, you must register every operation the service supports, even if the operation has a different transaction type from that of the Server. Note also that for each type of transaction used by operations in a given service, you must start at least one Server running the service and using that transaction type.

Example

```
void ostcConnect(ostc_ServerSession * session)
{
    session->addOperation(new txfer_operation());
    session->addOperation(new AddAccount());
    session->addOperation(new withdraw());
    session->addOperation(new deposit());
    session->addOperation(new balance());
}
```

txfer_operation() is a call to the operation subtype constructor for the operation being specified. You pass a pointer to an instance of the class. Include one call to **addOperation()** for each operation you want the component to support.

Click here for the example in context.

Declaration of Global Functions

A source or header file for your component must include the following declaration of the global functions you are responsible for implementing (including **ostcConnect()**):

```
extern "C" {

#ifdef WIN32
__declspec( dllexport )  void ostcInitialize();
__declspec( dllexport )  void ostcConnect(ostc_ServerSession *);
__declspec( dllexport )  void ostcDisconnect(void);
#else
```

```
void ostcInitialize();
void ostcConnect(ostc_ServerSession *);
void ostcDisconnect(void);
#endif
}
```

Note that you must use C linkage when declaring these functions..

This use of **_declspec** is required for Windows platforms.

Implementing Server Operation Execution

Implement the member **execute()** of each class you derive from **ostc_ServerOperation**. The protocol for this function is defined by **ostc_ServerOperation::execute()**. The Server calls **execute()** whenever the thin client calls **ostc_Session::doOperation()**.

The function must extract the arguments by calling a get-value member of **ostc_Object** on **Arguments**, for each attribute of **Arguments**. It must then process the arguments and set the result (in the form of C++ objects) by performing members of **ostc_OperationResult** on **result**.

Example

```
void txfer_operation::execute(
    ostc_Object * Arguments,
    ostc_OperationResult* result
)
{
    const char* from_acct_name =
        Arguments->getStringValue("from_acct_name");
    const char* to_acct_name =
        Arguments->getStringValue("to_acct_name");
    float amount =
        Arguments->getFloatValue("amount");

    Account * from_account = _app->getAccount(from_acct_name);
    Account * to_account = _app->getAccount(to_acct_name);

    _app->transfer(from_account, to_account, amount);

    os_collection *new_balances =
        &os_collection::create(
            os_segment::get_transient_segment()
        );

    new_balances->insert( from_account );
    new_balances->insert( to_account );

    result->setReturnValue(new_balances);
}
```

This example passes an **os_collection*** to **ostc_OperationResult::setReturnValue()**. The collection has two elements, a pointer to the balance of the account from which the transfer was made, and a pointer to the balance of the account to which the transfer was made.

You can also pass a **void*** or an **os_cursor*** to **setReturnValue()**.

Click here for the example in context.

Formatting Operation Results

You must implement the member **format()** of each class you derive from **ostc_ServerOperation**. The protocol for this function is defined by **ostc_ServerOperation::format()**.

In order to send the results of **execute()** to the thin client, the Server converts each return value into one or more **ostc_Object**s. It does this by calling **format()** on each return value.

If **execute()** passed a nonzero **void*** to **ostc_OperationResult::setReturnValue()**, the **void*** is considered to be the only return value. If **execute()** passed a nonzero **os_cursor*** to **setReturnValue()**, each element that can be visited with the cursor is a return value. If **execute()** passed a nonzero **os_collection*** to **setReturnValue()**, each element of the collection is a return value.

Example

```
void txfer_operation::format(
    const void * OSobj,
    ostc_ObjectList * ostcList,
    ostc_OperationResult*)
{
    Account * acct = (Account*)OSobj;

    ostc_Object * obj = ostcList->addObject();

    obj->setValue("name", acct->getName());
    obj->setValue("val", acct->getBalance());
}
```

OSobj is the return value being converted. Since it is passed in as a **void***, **format()** must cast it to the appropriate type.

ostcList is an empty list of **ostc_Object**s. **format()** adds one or more objects to the list (with **ostc_ObjectList::addObject()**) and sets their attribute values, so that the list ends up containing the object or objects into which **OSobj** is to be converted.

In this example, a pointer to a **float** is converted into an **ostc_Object** with a single attribute, **val**, whose value is the **float**. The value is set with **ostc_Object::setValue()**.

Click here for the example in context.

Execution Postprocessing

For nonincremental operations, the Server calls **ostc_ServerOperation::operationComplete()** when **execute()** and all associated **format()** executions complete. For incremental operations, the Server calls **ostc_ServerOperation::operationComplete()** when **execute()** and the first batch of associated **format()** executions complete.

By default, **ostc_ServerOperation::operationComplete()** deletes the collection or cursor, if any, passed to **setReturnValue()**. You can override the default implementation by implementing **operationComplete()** in the derived type.

Implementing **ostcDisconnect()**

You can use **ostcDisconnect()** to perform application-specific processing whenever the Server is about to shut down.

You must implement this function, even if there is no such processing to perform.

Example

```
// You must implement ostcDisconnect, even if as a no-op

void ostcDisconnect(void)
{
}
```

Chapter 4
QuickStart Example

In your Framework installation directory, **examples/ostc/quickstart** contains the source for a simple application that performs various operations on bank accounts. This chapter describes the application, focusing on an operation that transfers funds from one account to another.

examples/ostc/quickstart contains these two subdirectories:

- **console_client** contains the source for a thin client with a command-line user interface.
- **plugin** contains the source for a component.

Client Header File

This thin client uses one header file, **quickstartclient.h**, which declares the following classes:

- **QuickStartClient**: Declares a member function for each operation the application uses, including the transfer operation. Each member sends a request to a Component Server for execution of an operation.
- **QuickStartDisplay**: Encapsulates an instance of **QuickStartClient**, and declares a member function for each operation, including the transfer operation. Each member handles end-user I/O for the operation, and calls the corresponding member of **QuickStartClient**.

One way to modularize thin clients is to define one class, like **QuickStartClient,** for each component the client uses. This client uses a single component that supports all the operations.

All thin clients must include **ostc.h**:

```
#include <ostc.h>
```

Client Source Files

The thin client uses two **.cpp** files:

- **clientmain.cpp**
- **quickstartclient.cpp**

clientmain.cpp

clientmain.cpp instantiates **QuickStartClient** and **QuickStartDisplay** (on the stack), and starts the I/O event loop. The main program also uses the Client API to set the router:

```
// Main program for the client
// Must specify router host:port pair as argument

#include <stdio.h>
#include "quickstartclient.h"

int main(int argc, char ** argv)
{
    ...
    // Set the router
    ostc::addRouter(argv[1]);
    ...
}
```

quickstartclient.cpp

quickstartclient.cpp implements members of **QuickStartClient** and **QuickStartDisplay**. The constructor and destructor *connect* and *disconnect* from a Server:

```
#include <stdio.h>
#include "quickstartclient.h"

// Construction/Destruction
QuickStartClient::QuickStartClient()
{
    // Connect to the server
    _session = ostc::connect("quickstart");
}

QuickStartClient::~QuickStartClient()
{
    // Disconnect from the server
    ostc::disconnect(_session);
}
```

Here is **QuickStartDisplay::transfer()**, which is called from the event loop (in **QuickStartDisplay::run()**). It accepts input for the

name of the account to which to transfer the funds, the name of the account from which to transfer the funds, and the transfer amount.

It passes these to **QuickStartClient::transfer()**, which returns the result in the form of a list of **ostc_Object**s. Then it uses the Client API to extract the results.

```
void QuickStartDisplay::transfer()
{
    // Input from_name, to_name, and amount
    ...

    // Call QuickStartClient::transfer()
    ostc_ObjectList * result =
        _client->transfer(from_name, to_name, amount);

    ...

    // Extract the results

    ostc_Object * account = 0;

    account = result->first();
    printf(
        "%-20.15s%15f\n",
        account->getStringValue("name"),
        account->getFloatValue("val")
    );

    account = result->next();
    printf(
        "%-20.15s%15f\n",
        account->getStringValue("name"),
        account->getFloatValue("val")
    );

    ...
}
```

Here is **QuickStartClient::transfer()**. It receives **from_name**, **to_name**, and **amount** from **QuickStartDisplay::transfer()**. The function uses the client API to

- Retrieve the operation to be performed.
- Set the operation arguments.
- Execute the operation.

The function returns the result in the form of a list of **ostc_Object**s.

```
ostc_ObjectList * QuickStartClient::transfer(
    char * from_name,
    char * to_name,
    float amount
)
{
    // Get the transfer operation
    ostc_Operation * transfer = _session->getOperation("transfer");
    if ( !transfer )
        return 0;

    // Set up the arguments
    transfer->setArgument("from_acct_name", from_name);
    transfer->setArgument("to_acct_name", to_name);
    transfer->setArgument("amount", amount);

    // Execute the operation without using the cache
    ostc_ObjectList * result = _session->doOperation(transfer,0);

    if (!result) {
        return 0;
    }
    else {
        return result;
    }
}
```

Component Header Files

The Server plug-in uses the following header files:

- **account.h**: Declares the classes of persistent objects used by the plug-in.
- **db_name.h**: Specifies the pathname of the database containing the accounts.
- **quickstart.h**: Declares the class **QuickStart**, which declares a member function corresponding to each operation.
- **ostcinterface.h**: Declares global functions **ostcInitialize()**, **ostcConnect()**, and **ostcDisconnect()**.
- **ostcoperations.h**: Declares the subtype of **ostc_ServerOperation** corresponding to each operation.

ostcinterface.h

The plug-in header should include **ostcsrvr.h**:

```
#include <osctsrvr.h>
```

You must declare three global functions:

```
// Forward declarations
```

```
class ostc_ServerSession;

extern "C" {

#ifdef WIN32
__declspec( dllexport )  void ostcInitialize();
__declspec( dllexport )  void ostcConnect(ostc_ServerSession *);
__declspec( dllexport )  void ostcDisconnect(void);
#else
void ostcInitialize();
void ostcConnect(ostc_ServerSession *);
void ostcDisconnect(void);
#endif
}
```

This use of **_declspec** is required for Windows platforms.

ostcoperations.h

The Server plug-in defines one subtype of **ostc_ServerOperation** for each operation it supports. Here is the one corresponding to the transfer operation:

```
class txfer_operation : public ostc_ServerOperation
{
public:
    txfer_operation(QuickStart*);
    ~txfer_operation() {}

    void execute(ostc_Object * Arguments, ostc_OperationResult*);
    void format(
        const void*,
        ostc_ObjectList*,
        ostc_OperationResult*
    );

private:
    QuickStart * _app;
};
```

Server Plug-in Source Files

The plug-in uses the following **.cpp** files:

- **account.cpp**: Implements persistent class members.
- **ostcinterface.cpp**: Implements global plug-in functions.
- **quickstart.cpp**: Implements QuickStart members.
- **ostcoperations.cpp**: Implements operation subtype members.

ostcinterface.cpp

ostcInitialize() instantiates **QuickStart**, which instantiates the operation subtypes, opens or creates a database, and retrieves a root:

```
// This function is called once at server start-up
void ostcInitialize()
{
    // Generate an instance of the app
    if (!QuickStart::theServer)
        QuickStart::theServer = new QuickStart();

}
```

ostcConnect() registers the operation objects with the session:

```
// This function is also called once at start-up.
void ostcConnect(ostc_ServerSession * session)
{
    // Add the QuickStart operations to the server
    QuickStart::theServer->addOperations(session);
}
```

ostcDisconect() performs cleanup:

```
// This function is called once when the server is being shut down
void ostcDisconnect(void)
{

    // If we have no more connections then delete the QuickStart
    if (QuickStart::theServer)
        delete QuickStart::theServer;
}
```

quickstart.cpp

This is called by **ostcInitialize()**:

```
QuickStart::QuickStart()
{
    // The member initialize() opens or creates db and gets root
    initialize(example_db_name);

    // Create all of the server operations
    _addAccount = new AddAccount(this);
    _withdraw = new withdraw(this);
    _deposit = new deposit(this);
    _balance = new balance(this);
    _transfer = new txfer_operation(this);
}
```

This is called by **ostcConnect()**:

```
// Adds Server operations to the session
void QuickStart::addOperations(ostc_ServerSession * session)
{
```

```
    // Add all the operations to all servers for this service
    session->addOperation(_addAccount);
    session->addOperation(_withdraw);
    session->addOperation(_deposit);
    session->addOperation(_transfer);
    session->addOperation(_balance);
}
```

ostcoperations.cpp

The operation subtype constructor specifies the operation name, transaction type, and formal parameters:

```
txfer_operation::txfer_operation(QuickStart * app) :
    ostc_ServerOperation("transfer", ostc::isolated_update)
{
    // initialize
    _app = app;

    addArgument("from_acct_name", ostc::ostc_string);
    addArgument("to_acct_name", ostc::ostc_string);
    addArgument("amount", ostc::ostc_float);
    addReturnSetAttribute("val", ostc::ostc_float);
    addReturnSetAttribute("name", ostc::ostc_string);
}
```

execute() performs the operation:

```
void txfer_operation::execute(
    ostc_Object * Arguments,
    ostc_OperationResult* result
)
{
    const char* from_acct_name =
        Arguments->getStringValue("from_acct_name");
    const char* to_acct_name =
        Arguments->getStringValue("to_acct_name");
    float amount =
        Arguments->getFloatValue("amount");

    Account * from_account = _app->getAccount(from_acct_name);
    Account * to_account = _app->getAccount(to_acct_name);

    _app->transfer(from_account, to_account, amount);

    os_collection *new_balances =
        &os_collection::create(
            os_segment::get_transient_segment()
        );

    new_balances->insert( from_account );
    new_balances->insert( to_account );

    result->setReturnValue(new_balances);
```

```
}
```

format() formats the operation results as **Ostc_Object**s:

```
void txfer_operation::format(
    const void * OSobj,
    ostc_ObjectList * ostcList,
    ostc_OperationResult*)
{
    Account * acct = (Account*)OSobj;

    ostc_Object * obj = ostcList->addObject();

    obj->setValue("name", acct->getName());
    obj->setValue("val", acct->getBalance());
}
```

Chapter 5
Intermediate Example

This chapter discusses a sample application that allows end users to purchase tickets to a specified performance of a show such as a play, concert, or movie. The source code for the example is in **/examples/ostc/ticketapp**.

The thin client in this application allows the end user to perform the following actions:

- **ListShows**: Display list of all shows.
- **ListPerformances**: List all performances of a specified show.
- **ViewSeats**: List all available seats for a specified performance.
- **BuySeats**: Purchase specified seats for a specified performance.

There are also administrative actions for adding and removing shows, performances, and theaters.

Each of these actions corresponds to an **ostc_ServerOperation** defined by the Server plug-in in this example. The thin client assumes there are two Component Server processes running, one for read-only operations (**ListShows**, **ListPerformances**, and **ViewSeats**), and one for update operations (**BuySeats** and administrative actions).

The two Servers run the same service, but different transaction types, **read** and **isolated_update**, specified as command-line arguments to the Server executable.

When the thin client starts, it establishes a session with the service. When the user selects an action, the thin client uses the session to do the following:

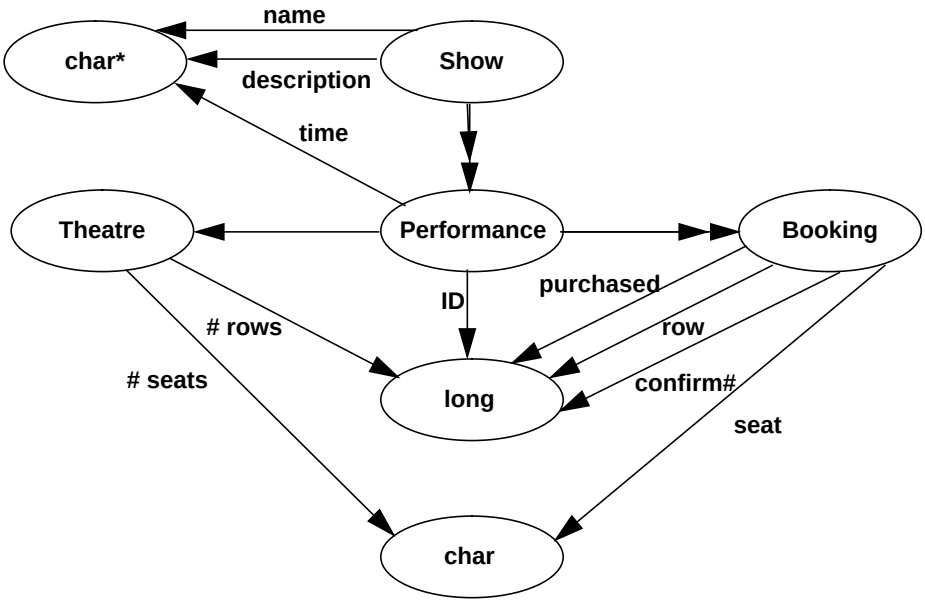
- Retrieve the operation corresponding to the action
- Set the arguments
- Execute the operation
- Extract the results
- Output the results to the user

Then the client redisplays the menu of actions.

Data Model

The **ostc_ServerOperation**s access persistent data stored in an ObjectStore database. This data uses the following classes:

- **Show:** Each show has a name, description, and an associated collection of performances.
- **Performance:** Each performance has an associated show, theater, and date and time (represented by a **char***). Each performance also has an associated collection of bookings objects.
- **Booking:** Each booking has an associated seat number, row number, price, and confirmation number (which identifies the purchaser). It also has an associated Boolean that indicates whether the seat has been purchased.
- **Theater**: For simplicity, a given theater is assumed to have the same number of seats in every row. So a theater's seating is described by two numbers: the number of rows in the theater, and the number of seats per row. Each theater object stores these two numbers, as well as the theater name.



The operation implementations rely on the following database entry points:

- List of all shows: used by **ListShows** operation
- Dictionary of all shows, keyed by name: used by **ListPerformances** operation
- Dictionary of all performances, keyed by ID: used by **ViewSeats** and **BuySeats** operations

The Client Code

The thin client uses the following files:

- **clientmain.cpp**
- **ticketappclient.h**
- **ticketappclient.cpp**

clientmain.cpp

The client main program does the following:

- Sets the router host and port
- Creates an instance of **TicketAppClient**, which establishes a session with the **ticketapp** service
- Creates an instance of **TicketAppDisplay** and runs it, which displays the main menu and starts the main processing loop
- Handles all exceptions thrown by the client API, by catching **oscs_Exception** (and its descendents)

ticketappclient.h and ticketappclient.cpp

These files define two classes:

- **TicketAppClient**
- **TicketAppDisplay**

TicketAppClient defines a function for each operation (**ListShows**, **ListPerformances**, **ViewSeats**, **BuySeats**, and the administrative operations). Each function retrieves the corresponding **ostc_ServerOperation**, sets the arguments, executes the operation, and extracts the results.

The class **TicketAppDisplay** handles end-user I/O. It defines a function for each operation. Each item of the main menu invokes one of these functions. Each function takes input from the end user, passes it to the corresponding member of **TicketAppClient**, and outputs the results.

The Server Code

The Server code consists of the following files:

- **ostcoperations.h** and **ostcoperations.cpp**
- **ostcinterface.h** and **ostcinterface.cpp**
- **ticketapp.h** and **ticketapp.cpp**
- Files for the classes **Show**, **Performance**, **Booking**, and **Theater**

ostcoperations.h and ostcoperations.cpp

ostcoperations.h and **ostcoperations.cpp** define subtypes of **ostc_ServerOperation**:

- **ListShows**
- **ListPerformances**
- **ViewSeats**
- **BuySeats**

Each subtype defines a constructor, a destructor, **execute()**, and **format()**. **ListShows** and **ListPerformances** also define **operation_complete()**, to override the default cleanup behavior.

ostcinterface.h and ostcinterface.cpp

ostcinterface.h and **ostcinterface.cpp** define the Server API functions **ostcInitialize()**, **ostcConnect()**, and **ostcDisconnect()**.

ostcInitialize() creates an instance of **TicketApp**, which creates an instance of each subtype of **ostc_ServerOperation**, and performs various initialization tasks, such as opening the database and retrieving database roots.

ostcConnect() adds the operations to the current session by calling **TicketApp::addOperations()**.

ostcDisconnect() deletes the instance of **TicketApp** created by **ostcInitialize()**.

ticketapp.h and ticketapp.cpp

The class **TicketApp** defines

- Initialization functions

- Functions for looking up shows, performances, and theaters by name or ID
- Members for controlling allocation and clustering of shows, performances, and theaters

Chapter 6
Using and Configuring the DataView Reader

Windows only

On Windows platforms, the DataView Reader provides read-only access to data views you define using ObjectStore Inspector. A data view is a tabular representation of a persistent ObjectStore collection, optionally filtered and ordered by an ObjectStore query.

Each persistent instance is displayed in a specified format. The instance format designates the set of data members to be displayed. For each specified member, the data view has a corresponding column. The instance format also specifies how strings are presented and how one-to-many and many-to-many relationships are presented. See the *ObjectStore Inspector User Guide* for more information about data views.

The DataView Reader makes it easy to define a Server operation that corresponds to a data view. Instead of defining a subtype of **ostc_ServerOperation** and adding an instance to the Server session in **ostcConnect()**, you simply *expose* the data view by calling **ostc_ServerSession::exposeDataView()** from **ostcConnect()**. You designate the data view by name, and specify the name of the corresponding Server operation.

When a data view is exposed in this way, the Server acts just as if you had defined a Server operation whose

- Arguments are those of the data view (see the *ObjectStore Inspector User Guide*).

- Return value is a pointer to an **ostc_ObjectList** with one element for each row of the data view, and one column for each member specified by the instance format.

If you want, you can specify the instance format independently of the data view's definition in Inspector. Do this with the DataView Reader Configuration Utility. When you expose a data view, you can also specify further filtering and ordering using SQL.

For you to use the DataView Reader, ObjectStore Inspector must be installed. When you install the Component Server Framework, the installation program verifies that the Inspector is installed, examines the Inspector configuration to determine where the Inspector metaknowledge resides, and configures the DataView Reader with that location. This allows the DataView Reader to access the data views and instance formats you define using Inspector. See the *ObjectStore Inspector User Guide*.

If the location of the Inspector metaknowledge changes, you can inform the DataView Reader of the new location using the DataView Reader Configuration Utility.

Accessing Data Views from the DataView Reader

To use the DataView Reader to access a data view defined in Inspector, build a component that explicitly exposes some or all the data views defined in an ObjectStore database. You build the component by implementing **ostcConnect()**, **ostcInitialize()**, and **ostcDisconnect()**.

You do not have to define any subtypes of **ostc_ServerOperation**. Instead, your implementation of **ostcConnect()** must expose one or more data views with one or a combination of the following functions.

To expose all the data views in a specified database, use the following member of **ostc_ServerSession**:

```
void exposeAllDataViews(oscs_ConstString database_name);
```

You can call this function several times, specifying different databases. Each data view is mapped to a Server operation whose name is the same as the data view's name.

To expose a data view with a specified name in a specified database, use the following member of **ostc_ServerSession**:

```
void exposeDataView(
    oscs_ConstString database_name,
    oscs_ConstString data_view_name,
    oscs_ConstString operation_name = 0
);
```

operation_name, if nonzero, specifies the name of the Server operation to which the data view is mapped. If **operation_name** is 0, **data_view_name** is used as the operation name.

You can expose only one database with a given name at a time. If you have exposed a data view from one database, and then attempt to expose a data view with the same name from another database, the attempt is silently ignored. Only the first data view will be accessible.

data_view_name can also contain an SQL statement that customizes the data view defined in **database_name**. The SQL statement must follow SQL syntax rules. The next paragraph describes the SQL syntax accepted by the DataView Reader.

Filtering and Ordering: SQL Support in DataView Reader

The DataView Reader supports a subset of the SQL syntax that lets developers dynamically modify data views defined in the Inspector by customizing the instance format and the filter and ordering definitions. The DataView Reader supports these statements:

- **SELECT**
- **FROM**
- **WHERE**
- **ORDER BY DESC/ASC**
- **AS**

Suppose you are inspecting a data view called **my_data_view**, based on a collection of **Customer** instances:

- **Customer** is a class containing the **name** and **address** data members, as well as a relation to the **Vehicle** class, implemented by the **cars** data member.
- **Vehicle** is a class containing the **make** and **model** data members and a reverse relation to the class **Customer** called **owner**.

The DataView Reader would accept SQL statements such as these:

SELECT * FROM my_data_view	This statement maintains the default settings of **my_data_view**.
SELECT name, address FROM my_data_view	This statement modifies the instance format used in **my_data_view** to include only the **name** and **address** of each customer.
SELECT name, address FROM my_data_view WHERE name like `john*'	This statement modifies the instance format and filter defined in **my_data_view.** The resulting table contains only those customers whose names begin with **john** and also displays the customers' names and addresses. The DataView Reader translates the filter defined in the SQL statement into an ObjectStore query. This allows you to perform efficient ObjectStore queries through an SQL statement.

SELECT name, address FROM my_data_view ORDER BY name DESC	This SQL statement modifies the instance format and order defined in **my_data_view**. The resulting table shows the customers' names and addresses displayed alphabetically in descending name order.
SELECT name, cars#make FROM my_data_view	The DataView Reader maps the SQL column names to data member names. To navigate implicitly from one class to another, you can specify a column name containing the concatenation of the navigated data members separated by the **#** character. This lets you build a table based on the persistent collection used to define **my_data_view**. This adopts the same filtering and ordering settings, but it contains a column that shows the customer's name and the makes of the cars the customer owns. In general, you can concatenate any number of relations. For example, **cars#owner#cars#model** would be a valid column name. It would contain the models of the cars, which are owned by the owner of the cars, which are owned by each customer.
SELECT name, cars#make, cars#model FROM my_data_ view WHERE cars#make='Ford'	This SQL statement uses the navigated data members to build a new filter in **my_data_view**. As the filter is translated into a native ObjectStore query expression, the ObjectStore query executed by the above SQL statement **iscars[: !strcmp(make,"Ford") :]** This query is satisfied by any customer who owns *at least one* **Ford** *car*. This result is different from what you could expect reading the SQL expression.

DataView Reader Configuration Utility

The DataView Reader configuration utility is an application that you can use to change the DataView Reader default settings. The CSF installation program establishes these defaults based on the locations of DataView Reader and the Inspector. You can synchronize all the DataView Reader settings with the corresponding Inspector settings, or you can reconfigure individual DataView Reader settings.

The individual settings include

- The location of the utility's application schema database
- The location of the Inspector metaknowledge
- The format for displaying one-to-many and many-to-many relationships
- The format for displaying characters, strings, and BLOBs

Preparing to Run the Configuration Utility

The database schema is stored in the DataView Reader library, and all other settings are stored in the Windows NT registry. Before configuring the DataView Reader, unlock the DataView Reader libraries by closing all application servers that use the DataView Reader. If the DataView Reader libraries are locked when you run the DataView Reader configuration utility, it cannot modify the application schema path. Furthermore, any changes applied to the DataView Reader settings are not visible to the already running DataView Reader instances.

Starting the Utility

The DataView Reader configuration utility is in the CSF program group. It has three tabs:

- Application Schema
- General Settings
- String Formats

Synchronizing with Inspector

The DataView Reader and the Inspector share metaknowledge, data views, instance formats, and the ObjectStore application

schema. To synchronize these DataView Reader settings, the DataView Reader string display format, and all other DataView Reader settings with the corresponding Inspector settings, click **Synchronize with Inspector** from any tab.

Setting the Application Schema Path

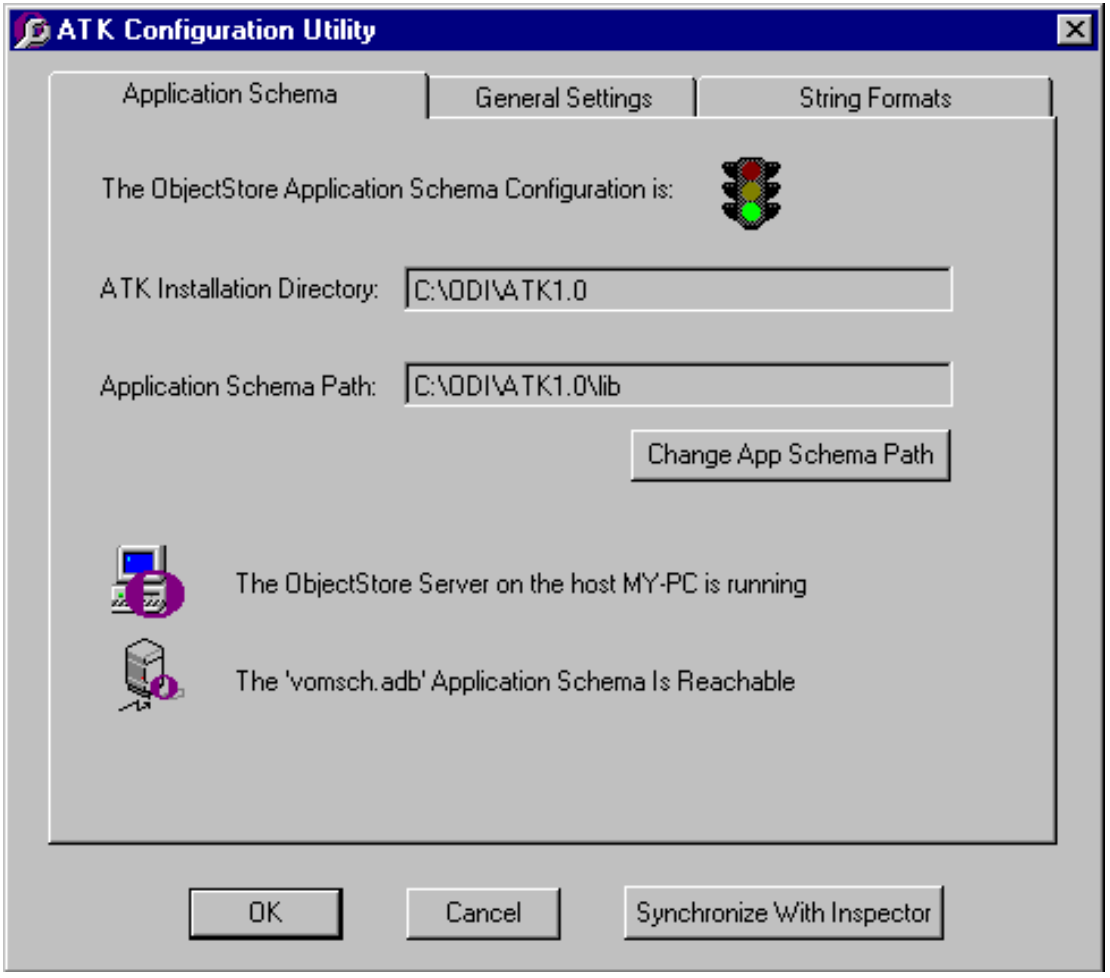
The application schema for the DataView Reader is an ObjectStore database in the file **vomsch30.adb**. In order for the application to access the application schema, the schema must be under the control of an ObjectStore Server. If you manually move the application schema, you must reconfigure the DataView Reader with the schema's new location.

For example, suppose an application running on a Windows NT workstation that has no ObjectStore Server queries a Solaris ObjectStore Server. To successfully query the Solaris Server database, you must copy the application schema to the Solaris machine and reconfigure the DataView Reader on the Windows NT machine with the remote location of the Solaris workstation.

If you installed CSF using the default installation location, **vomsch30.adb** was placed in **C:\ODI\CSF1.0\lib**. Use the Application Schema tab to check or modify the DataView Reader application schema path, and to configure DataView Reader to use a particular ObjectStore Server located on your network.

The Application Schema tab displays the DataView Reader application schema configuration:

- The configuration status, signified by a green, yellow, or red traffic signal icon
- The location of the application schema database
- Whether the application schema database is accessible to an ObjectStore Server



The color of the traffic signal summarizes the application schema configuration:

- Green: DataView Reader is properly configured to use an application schema database on a workstation where an ObjectStore Server is running.
- Yellow: Although DataView Reader is configured to use an application schema database on a workstation where an ObjectStore Server is running, the application schema database has not been copied to the specified directory.

- Red: The DataView Reader is configured to use an application schema database on a host where no ObjectStore Server is running.

A CSF application server that uses the DataView Reader and that runs when the traffic signal is yellow or red encounters an error when it attempts to open a database.

To establish a new location:

1 Click **Change App Schema Path**.

2 Enter another directory location. You can specify a directory on the local machine, an ObjectStore rawfs, or a network file server directory.

3 Copy **vomsch30.adb** to the directory location you specified.

The **Application Schema** tab also indicates whether an ObjectStore Server is running on the host designated by the application schema path:

- An ObjectStore Server is running on the host.

- No ObjectStore Server is running on the host. The traffic signal is red.

- An ObjectStore Server is running on the host. Because the DataView Reader is properly configured and ready to be run, the traffic signal is green.

- The DataView Reader application schema database is not accessible, because an ObjectStore Server is not running on the host designated by the application schema path, or because the

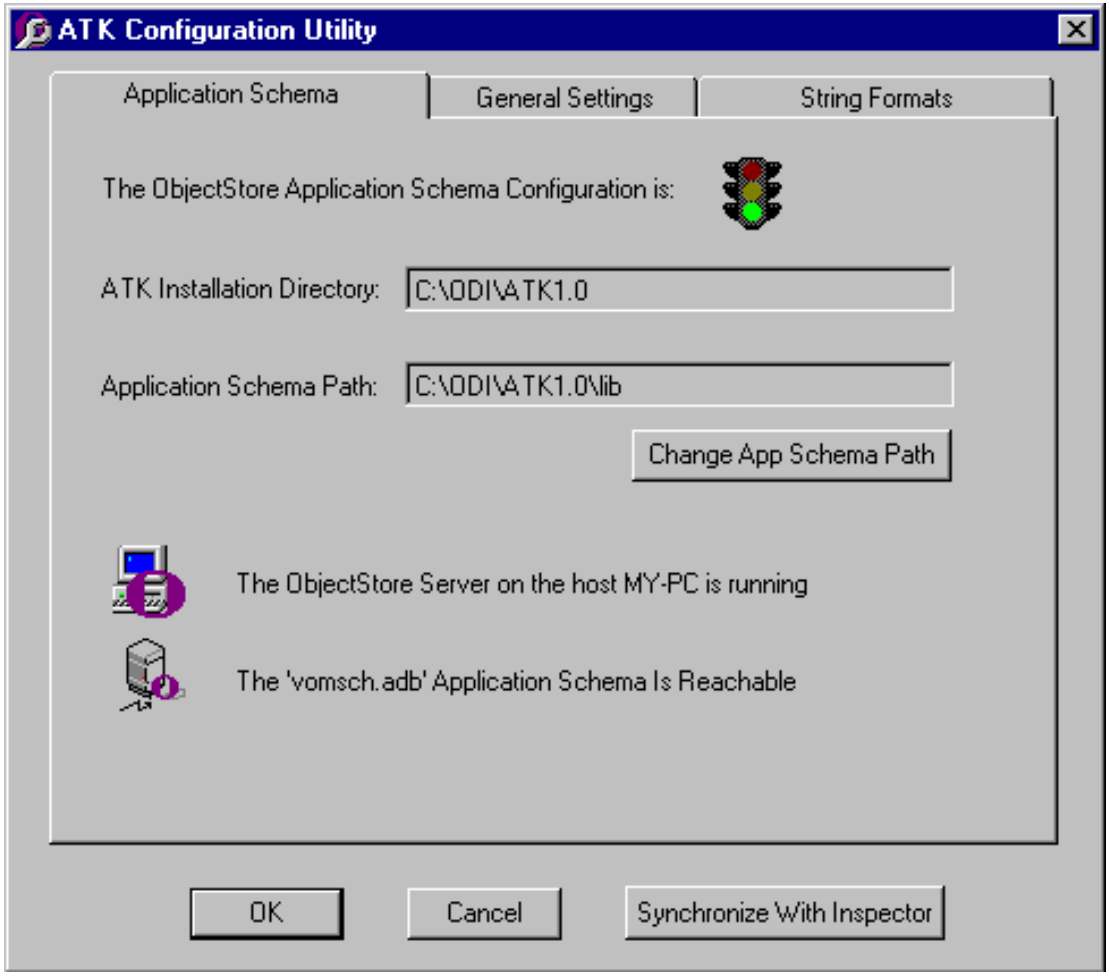
database file **vomsch30.adb** is not present in that directory. The traffic signal is yellow if the Server is running, or red if no Server is running.

To use the application schema database path that Inspector uses, click **Synchronize with Inspector**. Note that this synchronizes all the DataView Reader configuration settings in all tabs with the equivalent Inspector settings.

Modifying the Metaknowledge Location and Specifying Relationship Format

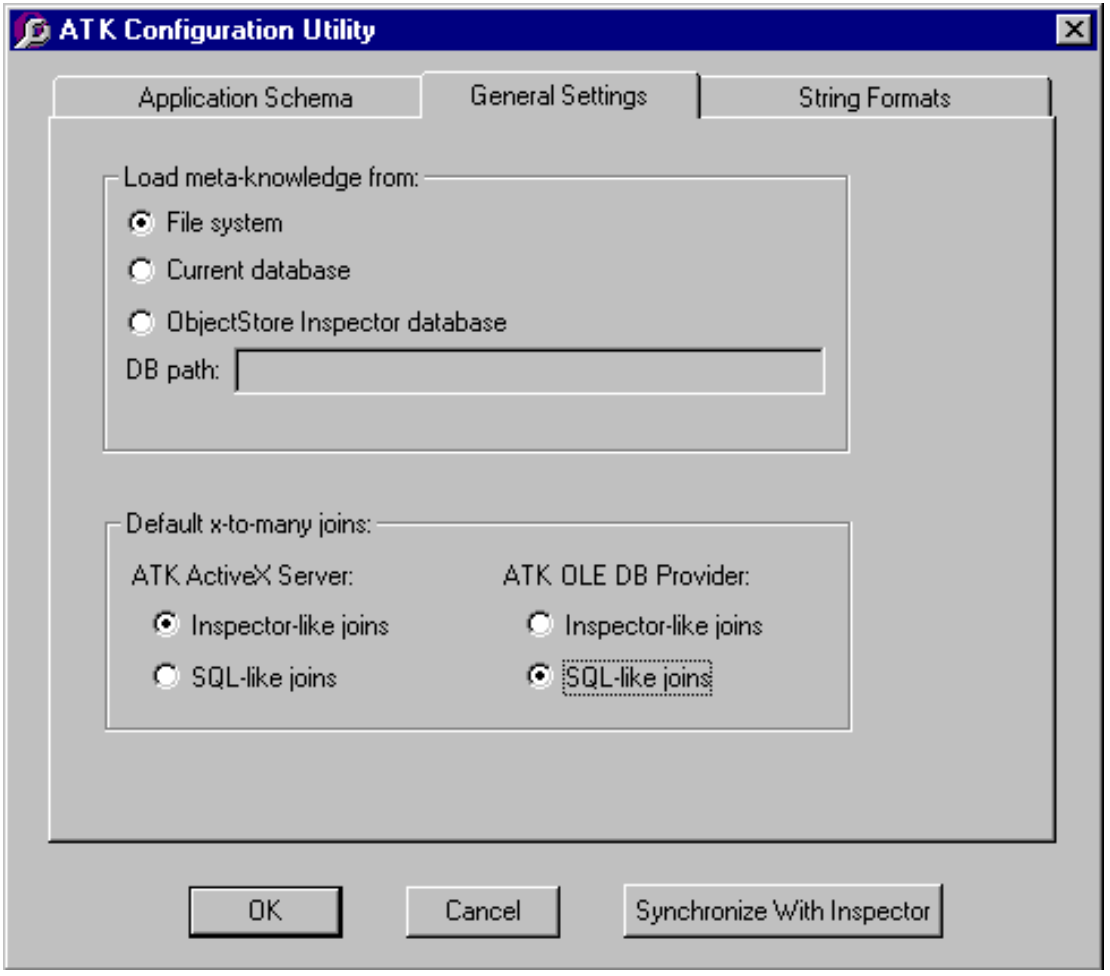
The DataView Reader uses the Inspector metaknowledge (see the *ObjectStore Inspector User Guide*) to obtain data view definitions, including the query and ordering information set by the query's author, and other information about the database schema and the defined instance formats.

Inspector metaknowledge can reside in the file system, in the



same inspected database, or in another ObjectStore database that contains only Inspector metaknowledge. The metaknowledge location is specified in the Inspector configuration.

Use the General Settings tab to specify the location of the DataView Reader metaknowledge, and to choose the default type of join the DataView Reader uses when displaying relations in a single table.



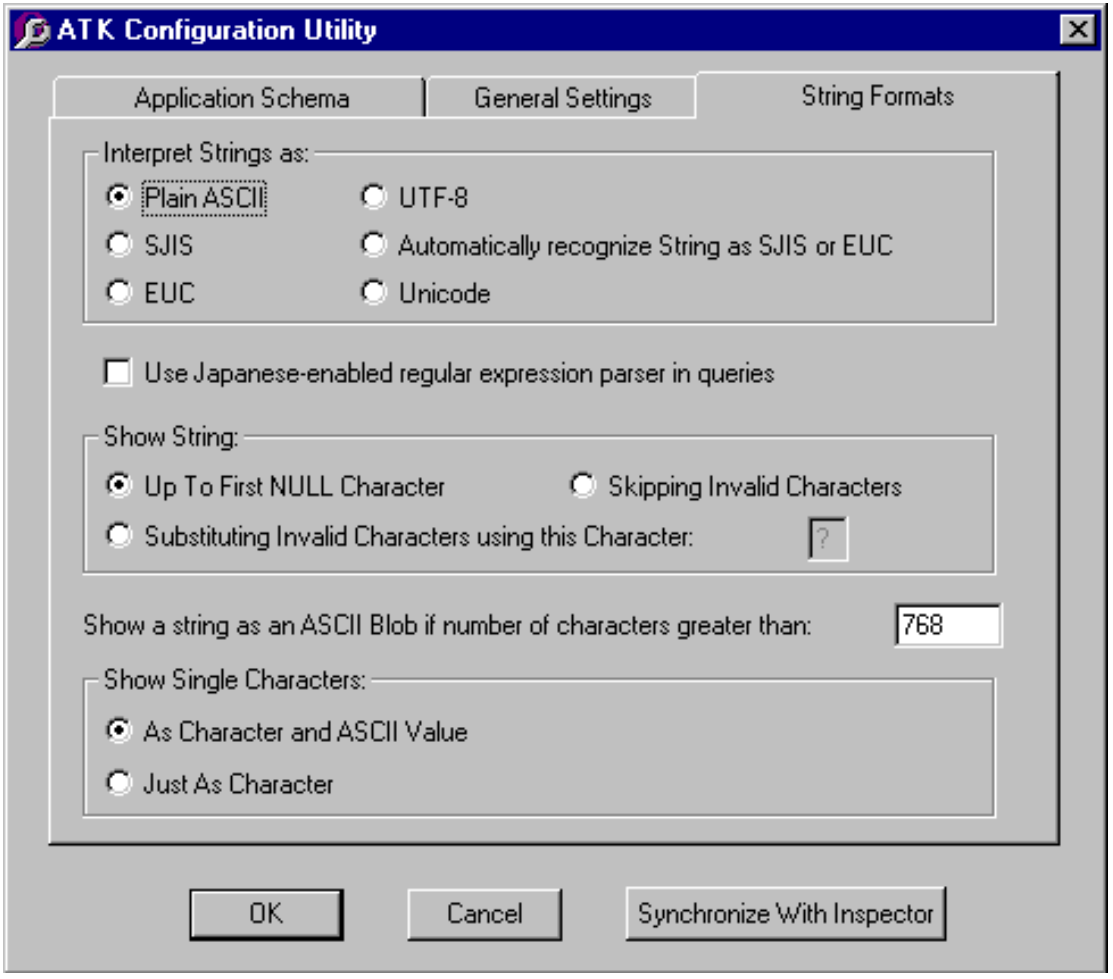
By default, the CSF installation designates the file system as the metaknowledge location. To manually modify this setting, specify the location from which the DataView Reader should load the metaknowledge.

By default, the DataView Reader displays one-to-many and many-to-many relations using SQL-like joins. You can choose the join format you prefer.

Modifying the Character, String, and BLOB Formats

Use the String Formats tab to specify the default string display format for the current database. For additional information about

string formats and string interpretation, refer to Viewing Blobs in the *ObjectStore Inspector User Guide*.



Interpret Strings as: By default, DataView Reader interprets strings as plain ASCII text. You can interpret strings using any format that the Inspector supports. This is especially helpful if the database contains strings that are encoded using SJIS, EUC, or Unicode.

Japanese expressions: If the application retrieves objects encoded in Japanese characters, you can ensure that DataView Reader uses a Japanese-enabled parser that supports DBCS and Unicode. This parser is included in the DataView Reader kernel.

Show String: By default, the DataView Reader displays strings up to the first null character it encounters. Instead, DataView Reader can skip displaying invalid characters or display an alternate character for each invalid one.

BLOBs: By default, the DataView Reader displays ASCII strings that are longer than 768 characters as binary large objects (BLOBs). You can increase or decrease this limit.

Show Single Characters: The DataView Reader displays single characters as a character and an ASCII value. For example, the DataView Reader would display the character A as A' (65). Instead, you can display only the character value, such as A.

Registry Keys

The DataView Reader registry keys are logically split into the following categories: metaknowledge location, string encoding, string handling, and navigation handling.

Metaknowledge Location

The DataView Reader task is to expose data views defined in Inspector. The knowledge about these data views is called metaknowledge, and can be stored by Inspector in different locations, according to its settings.

The key **"Where to save"** (**REG_DWORD**) can contain the following values:

- Metaknowledge is saved in the file system; in this case, the key **"database GPH"** (**REG_SZ**) contains the full path of the directory where the metaknowledge is stored (it *must* terminate with a **/** character). Inspector by default sets **"Where to save"** to 1 and **"database GPH"** to "*inspector-rootdir*\db_gph\".
- Metaknowledge is saved in the database to which the metaknowledge is referred. This is the recommended choice if you want to deploy an application that uses the DataView Reader to expose the contents of your database.
- Metaknowledge is saved in an ObjectStore database dedicated to storing it; in this case the **"ObjectStore database"** (**REG_SZ**) key contains the full file name path of the ObjectStore database.

String Encoding

Character strings can be encoded in a wide variety of formats inside a database. Inspector and the DataView Reader let you specify the default encoding type of the character strings stored in your databases (this default can be overridden on a per-class basis in each database, using Inspector). The DataView Reader always returns strings in ASCII or SJIS encoding type.

The default character string encoding is stored in the **"AttributeFormatDisplayingDefault"** (**REG_DWORD**) key. The possible values are

- **1**: Character strings are simple ASCII, and they do not use any particular encoding (default).
- **2**: Character strings are encoded in SJIS (Windows-Japanese) format.
- **4**: Character strings are encoded in EUC (UNIX-Japanese) format.
- **8**: Encoding type of character strings must be recognized automatically by Inspector or DataView Reader.
- **16**: Character strings are encoded in Unicode format.
- **32**: Character strings are encoded in UTF8 (Java-Unicode) format.

The **"UseJapaneseRegex"** (**REG_DWORD**) registry key contains information on whether the DataView Reader must invoke the Japanese-compliant version of the regular expression parser used in the queries:

- **0**: (Default) Use the standard regular expression parser.
- **1**: Use the Japanese-compliant regular expression parser.

String Handling

Strings can be stored in buffers of characters in many different ways. Inspector and the DataView Reader let you specify the default way in which strings stored inside your databases should be retrieved (this default can be overridden on a per-class basis in each database, using Inspector).

The default setting is stored in the **"AttributeFormatInterpretationDefault"** (**REG_DWORD**) key. The possible values are

- **32**: Show the string until the first NULL character in the buffer is met (default).
- **64**: Show all the characters in the allocated buffer, skipping the invalid ones.
- **128**: Show all the characters in the allocated buffer, and use a special character to replace the invalid ones. This special character is stored in the **"AttributeFormatSubstitutionDefault"** (**REG_SZ**) registry key.

In all the preceding cases, if the resulting string is longer than the Blob Size, the string is not displayed, and the "**XXX bytes at address 0x123456**" message is returned. The Blob Size is stored in the **"MaxNonBinaryBlobSize"** (**REG_DWORD**) registry key (the default is 768).

- **256**: Like 32, but truncate the string at the Blob Size if needed.

The **"SingleCharacterFormat"** (**REG_DWORD**) key contains information about how to report single-character data members:

- **0**: (Default) Single character is returned in the format **'A' (65)**.
- **1**: Only the single character is returned: **A**

Navigation Handling

When a data view uses a data member that represents a one-to-many or many-to-many relationship, Inspector and the DataView Reader can choose between two possible algorithms. Information about which algorithm should be used is stored in the **"OLEDB SQL-like join"** (**REG_DWORD**) registry key:

- **1**: (Default) Use an SQL-like approach. The DataView Reader expands one-to-many relations to show all the possible values that meet the conditions of the query (as an SQL join would). Consider a data view that lists the names of customers together with the makes and the models of the cars they own. With this approach, the output takes this form:

Name	***Make***	***Model***
John, Smith	Cadillac	Deville
John, Smith	Ford	Escort
Jones, Mary	Ford	Ranger
Jones, Mary	Mazda	626
Jones, Mary	Mazda	626

- **2**: Use a tree-like approach. With this approach, the data view above takes this form:

Name	***Make***	***Model***
John, Smith	Cadillac	Deville
	Ford	Escort
Jones, Mary	Ford	Ranger

Name	*Make*	*Model*
	Mazda	626
	Mazda	626

Usually a tree-like view is more readable, but an SQL-oriented tool such as a report generator cannot easily use it. In the SQL-like view, if you navigate multiple one-to-many relations in a row, the number of generated rows is much greater than in the tree-like view.

Chapter 7
ostc

This class is used by both the client and the Server. It defines the functions the client uses to set the router and to connect to and disconnect from a Component Server. It also defines an enumeration relating to the fundamental types of data Servers and thin clients can exchange, as well as an enumeration relating to transaction types.

OSTC Fundamental Data Types

The types of values that clients can specify as arguments to operations (see **ostc_Operation::setArgument()** on page 100) are limited to the following:

- Integer types
- **float**
- **double**
- **char***
- **ostc_OID*** (object identifier set by user)
- **void*** (for bit streams)

Attribute values for **ostc_Object**s are also limited to these types.

The following typedefs are used to designate the fundamental types:

oscs_Int64

oscs_Int32

oscs_String

oscs_Bool

oscs_Float

oscs_Double

Operation formal parameters are specified (see **ostc_ServerOperation::addArgument()** on page 150 and **ostc_ServerOperation::addReturnSetAttribute()** on page 150) with enumerators that correspond to the fundamental types. The following enumeration is defined in the scope of the class **ostc**:

```
enum types {
    ostc_int32 = 1,
    ostc_int64,
    ostc_float,
    ostc_double,
    ostc_string,
    ostc_oid,
    ostc_binary
};
```

The following table shows the correspondences:

ostc_int32	**32-bit integer**
ostc_int64	**64-bit integer**
ostc_string	**char***
ostc_float	**float**
ostc_double	**double**
ostc_oid	**ostc_OID***
ostc_binary	**void***

Transaction Types

The class **ostc** (named for the ObjectStore thin client) defines an enumeration that specifies transaction types:

```
enum transaction_type {
    read, mvccRead, shared_update, isolated_update, none, any
};
```

Each instance of **ostc_ServerOperation** has a transaction type, which specifies the type of transaction in which it must be executed. You specify the transaction type when you create the instance.

Each Server also has a transaction type, which specifies the type of transaction the Server uses to execute operations. You specify a Server's transaction type when you configure the Server. You can also modify a Server's transaction type with **ostc_ServerSession::setTxnType()**.

With all transaction types except **isolated_update**, the Server batches requests from different users into the same transaction, reducing commit overhead and increasing throughput.

read transactions can be used for operations that perform no updates to persistent data.

mvccRead transactions can be used for operations that perform no updates to persistent data and do not need a completely up-to-date version of the data — see *Using the ObjectStore C++ Interface*.

isolated_update transactions are for operations that perform updates to persistent data. Each transaction handles just one operation execution, whose results are committed once the operation completes.

shared_update transactions should be used only with *extreme care*. They are for update operations that can safely be batched into the same transaction with operations executing on other Server threads. For this to be safe, the operations must not potentially *interfere* with each other when accessing persistent data. *Unsafe batching can produce incorrect results or database corruption.*

any transactions should be used for operations that you want executed on the first available Server, regardless of transaction

type. This *cannot* be used to specify the transaction type of a Server.

none is for operations that either access no persistent data.

Determining Whether Update Operations Can Interfere with Each Other

Operations potentially interfere with each other if some way of interleaving their primitive database reads and writes produces incorrect results.

Consider, for example, an operation to purchase a ticket for a specified seat at a particular performance. Suppose the operation performs one database read, to see if the seat is available, and one database write, to mark the seat as sold (if it was available). Now consider two executions of this operation running on different threads, in the same transaction. Their database reads and writes might be performed in the following order:

thread1	***thread2***
read	
	read
	write
write	

In this case, both executions will find the seat to be available, and two people will end up with tickets for the same seat. This is an incorrect result, so the operations can interfere with one another and cannot be batched safely.

Note that what counts as an incorrect result is application dependent. Note also that, even if some possible ways of ordering the primitive reads and writes produce correct results, as long as there is at least one ordering that produces an incorrect result, the operations cannot be batched safely.

It is often very difficult to determine if update operations can be batched safely. When in doubt, use **isolated_update** for update operations.

With **read** and **mvccRead** transactions, the operations perform only database reads, which can always be safely batched.

Batch Transaction Commits

The results of a **shared_update** operation are not committed to the client until the completion of all the operation executions in the same batch. From the client's point of view, this means that **ostc_Session::doOperation()** does not return until all the batch's operations complete. This also means that one thread performing a relatively lengthy operation can delay other threads. However, in many cases overall throughput is increased.

Results of **read** and **mvccRead** operations are returned to the client without waiting for all operations in the batch to commit, since an abort of one (read-only) operation does not affect the results of other operations in the batch.

shared_update Aborts

If one operation execution in a **shared_update** transaction aborts, all executions in the batch must be aborted.This means that using **shared_update** can increase the likelihood of your operation's being aborted.

Setting Routers

In any deployed configuration of Component Servers, one or more Servers act as router and distributes requests to Servers on a round-robin basis. You specify whether a Server is a router when you launch the Server processes (see Configuring, Starting, and Stopping the Server Executable on page 15).

Each client must specify a router as well. You do this with the API **ostc::addRouter()**. On Windows platforms, you can use the executable **ostcSetRouter.exe** instead of this API.

ostc::addRouter()

static ostc_Session* addRouter(char *host_and_port);

host_and_port is a character string specifying the host machine of a router Component Server, as well as the port on which it is listening for thin client requests.

*host***:***port*

When the client calls **ostc::connect()**, a router tells the client which Component Server to send requests to. The router makes this determination based on

- The specified service name
- A round-robin schedule of Component Server processes that match the specified service name

Click here for the example.

Connecting to and Disconnecting from a Service

ostc::connect()

```
static ostc_Session* connect(
    oscs_ConstString service_name,
);
```

Establishes a session for using a specified service.

service_name is the name of the service.

When the client calls **connect()**, a router finds a Component Server that supports the specified service.

The function returns a pointer to a new **ostc_Session**.

You should retrieve only one session per client thread. Session objects are cached, making **connect()** (and **disconnect()**) inexpensive operations.

ostc_RequiredParmMissing is thrown if a router has not been set with **ostc::addRouter()** or **ostcSetRouter.exe**.

Click here for an example.

ostc::disconnect()

```
static void disconnect(ostc_Session*);
```

Ends the specified session, and disconnects from the session's associated Component Server.

Chapter 8
ostc_Session

This class is used by the thin client for the following tasks:

Getting a Session's Operations

ostc_Session::getOperations()

```
virtual ostc_OperationSet* getOperations() ;
```

Returns a pointer to an **ostc_OperationSet** containing the operations supported by **this** service.

ostc_Session::getOperation()

```
virtual ostc_Operation* getOperation(
    const char* operation_name
) ;
```

Returns a pointer to the operation that is named **operation_name** and supported by **this** service. Returns **0** if there is no such operation.

Click here for an example.

Executing Operations

ostc_Session::doOperation()

```
virtual ostc_ObjectList* doOperation(
    ostc_Operation *op,
    oscs_Bool use_cache,
    oscs_Bool incremental = 0,
    oscs_Uint32 max_objs_returned = 0,
    void * userRouteData = 0,
    oscs_Uint32 userRouteDataLen = 0
) ;
```

Executes the operation **op** on a Component Server.

When you call this function for a given operation, the client does one of the following:

- Contacts a router to locate a Server that *matches* the operation.
- Finds a *matching* Server based on information cached during the current process on the client side.

If no client transaction (started with **ostc_Session::beginTxn()**) is in progress, a Server matches an operation if all the following hold:

- The Server is running the service associated with **this** session.
- The Server's transaction type matches the operation's transaction type. Any Server transaction type except **none** matches the operation transaction type **ostc::any**.
- If the argument **userRouteData** is nonzero, the Server's route data matches **userRouteData**. Matches are identified with **memcmp()**.

In this case, the Server starts (or, for a batch transaction, has already started) the transaction in which the operation is executed. The Server commits or aborts the transaction before **doOperation()** returns.

If a client transaction *is* in progress, a Server matches an operation if all the following hold:

- The Server is running the service associated with **this** session.
- The Server's transaction type is **ostc::isolated_update**.

- If the argument **userRouteData** is nonzero, the Server's route data matches **userRouteData**. Matches are identified with **memcmp()**.

In this case, the Server started this operation's transaction when the client called **ostc_Session::beginTxn()**.

The client maintains a cache of object lists returned by **doOperation()** during the current session. **doOperation()** does not interact with the Server and instead returns a cached list if

- **use_cache** is 1.
- The same operation with the same arguments was executed earlier in the current session.
- The returned list from that execution is still in the thin client's cache.

Note that **use_cache** does not affect whether an operation's result is added to the cache. An operation's result is added regardless of the value of **use_cache**.

Note that the cache does not reflect changes to persistent data made by other processes; cache validity is *not* automatically maintained.

If **incremental** is nonzero, **doOperation()** returns a pointer to an incremental **ostc_ObjectList,** and **max_objects_returned** specifies the batch size of the returned set.

If the number of objects to be returned is large, the client can specify that the results be returned incrementally. The incremental nature of the results is transparent to the client, since the usual API functions for retrieving list elements automatically trigger a Server interaction whenever a new batch of results must be sent.

For **isolated_update** operations, each batch is sent in its own transaction. The Server creates a collection of references to the operation results, which can be used across transactions.

See also **ostc_ObjectList::done()** on page 130.

If **incremental** is 0, **max_objects_returned** specifies the maximum size of the returned set.

The returned **ostc_ObjectList** contains **op**'s return values. 0 is returned if there are no return values.

For batched transactions, **doOperation()** does not return until the execution of each operation in the batch completes.

ostc_ServerException is thrown if the Server aborts the transaction in which the operation executes.

Click here for an example.

Managing the Cache

ostc_Session::flushObjects()

virtual void flushObjects() ;

Clears the thin client cache; that is, removes all object lists from the cache.

virtual void flushObjects(ostc_ObjectList&*) ;

Removes the specified list from the thin client cache. Sets the argument to 0.

Managing Transactions

ostc_Session::beginTxn()

virtual void beginTxn() ;

You do not usually have to use this function, since transactions are started automatically by the Server. Use **beginTxn()** only to group two or more calls to **doOperation()** into the same transaction.

Starts a transaction for the execution of operations with **ostc_Session::doOperation()**. When you call this function, the router starts an **isolated_update** transaction on a Server such that both of the following hold:

- The Server is running the service associated with **this** session.
- The Server's transaction type is **isolated_update**.

If you do not use this function, operations are automatically executed in the appropriate type of transaction when you call **ostc_Session::doOperation()**.

ostc_InvalidTxnType is thrown if no **isolated_update** Server is running.

ostc_NoNestedTxn is thrown if a transaction started with **beginTxn()** is already in progress on the client.

ostc_Session::commitTxn()

virtual void commitTxn() ;

Commits the current transaction. See also **ostc_Session::beginTxn()** on page 91.

ostc_NoTxnInProgress is thrown if there is no current transaction.

ostc_Session::abortTxn()

virtual void abortTxn() ;

Aborts the current transaction. See also **ostc_Session::beginTxn()** on page 91.

ostc_NoTxnInProgress is thrown if there is no current transaction.

Getting the Name of a Session's Associated Service

ostc_Session::getServiceName()

virtual oscs_ConstString getServiceName() ;

Returns the name of the service with which **this** is a session. Do not modify or delete the returned string.

Chapter 9
ostc_Operation

Instances of this class represent operations that the thin client can send to a Component Server for execution. This class is used by the thin client for the following tasks:

Getting Operation Formal Parameters

ostc_Operation::getArguments()

virtual ostc_AttributeDescriptorList* getArguments() ;

Returns a pointer to a list of **ostc_AttributeDescriptor**s representing **this** operation's formal parameters.

virtual ostc_AttributeDescriptorList* getReturnListAttributes() ;

Returns a pointer to a list of **ostc_AttributeDescriptor**s representing the attributes of the objects returned by **this** operation.

Getting an Operation's Name and Description

ostc_Operation::getName()

virtual oscs_String getName() ;

Returns the name of **this** operation. See **ostc_ ServerOperation::ostc_ServerOperation()** on page 153. The return value is **const**, so the user should not modify or delete it.

ostc_Operation::getDescription()

virtual oscs_String getDescription() ;

Returns the description of **this** operation. See **ostc_ ServerOperation::setDescription()** on page 151. The return value is **const**, so the user should not modify or delete it.

Getting an Operation's Timestamp

ostc_Operation::getTimestamp()

virtual oscs_Int32 getTimestamp() ;

Returns a timestamp indicating the last time that executing **this** operation produced a result that was not taken from the cache. The format of the return value is the same as that of the C function **time()**.

Getting Operation Actual Parameters

ostc_Operation::getInt32Argument()

```
virtual oscs_Int32 getInt32Argument(
    oscs_ConstString argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Int32**.

ostc_Operation::getInt64Argument()

```
virtual oscs_Int64 getInt64Argument(
    oscs_ConstString argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Int64**.

ostc_Operation::getStringArgument()

```
virtual oscs_ConstString getStringArgument(
    oscs_String argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_String**.

ostc_Operation::getFloatArgument()

```
virtual oscs_Float getFloatArgument(
    oscs_ConstString argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Float**.

ostc_Operation::getDoubleArgument()

```
virtual oscs_Double getDoubleArgument(
    oscs_ConstString argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Double**.

ostc_Operation::getObjectArgument()

```
virtual ostc_OID * getObjectArgument(
    oscs_ConstString argname
) ;
```

Returns the actual parameter with the specified name.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_OID**.

ostc_Operation::getBinaryArgument()

```
virtual void * getBinaryArgument(
    oscs_ConstString argname,
    oscs_Uint32& length
) ;
```

Returns the actual parameter with the specified name.

length is the length of the bit stream making up the parameter value.

If the actual parameter has not been set (by **ostc_Operation::setArgument()**), returns **0**.

ostc_NoArguments is thrown if **this** operation has no arguments.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_binary**.

Setting Operation Actual Parameters

ostc_Operation::setArgument()

```
virtual void setArgument(
    oscs_ConstString name,
    oscs_Int32 value
) ;
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Int32**.

```
virtual void setArgument(
    oscs_ConstString name,
    oscs_Int64 value
) ;
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Int64**.

```
virtual void setArgument(
    oscs_ConstString name,
    oscs_ConstString value
);
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_string**.

```
virtual void setArgument(
    oscs_ConstString name,
    oscs_Float value
) ;
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Float**.

```
virtual void setArgument(
    oscs_ConstString name,
    oscs_Double value
) ;
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **oscs_Double**.

```
virtual void setArgument(
    oscs_ConstString name,
    ostc_OID* value
) ;
```

Sets the value of the actual parameter named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_OID***.

```
virtual void setArgument(
    oscs_ConstString name,
    void* value,
    oscs_Uint32 length
) ;
```

Sets the value of the actual parameter named **name** to the bit stream **value**.

length is the length of the **value**.

ostc_UnknownAttribute is thrown if there is no argument with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_binary**.

Chapter 10
ostc_OperationSet

Instances of this class are unordered collections of **ostc_Operation**s.

Cursor Validity

A set's associated cursor is *valid* if it is positioned at an element of the set, and is *invalid* if it is not positioned at any element of the set. Each set's cursor is initially invalid. Calling either **ostc_OperationSet::first()** or **ostc_OperationSet::next()** on the set renders the cursor valid.

ostc_OperationSet::first()

virtual ostc_Operation* first() ;

Positions the set's cursor at the set's first element, and returns that element. If the set is empty, returns **0**. The ordering used is arbitrary, but stable across traversals of the set.

ostc_OperationSet::next()

virtual ostc_Operation* next() ;

If **this** set's cursor is valid and is not positioned at the set's last element, **next()**

- Advances the cursor
- Returns the element at which the cursor is positioned after being advanced

If the cursor is positioned at the set's last element, **next()**

- Renders the cursor invalid

- Returns **0**

If the cursor is invalid and the set is nonempty, **next()**

- Positions the cursor at the set's first element
- Returns the first element

If the set is empty, **next()** returns **0**.

The ordering used is arbitrary, but stable across traversals of the set.

ostc_OperationSet::more()

virtual oscs_Bool more() ;

Returns **oscs_True** if **this**'s cursor is valid, that is, positioned at an element of the set. Returns **oscs_False** otherwise.

ostc_OperationSet::cardinality()

virtual oscs_Uint32 cardinality() ;

Returns the number of elements in the specified set.

ostc_OperationSet::getOperation()

virtual ostc_Operation* getOperation(oscs_ConstString name) ;

Returns the element of **this** that points to the operation with the specified name. Returns **0** if there is no such element.

Chapter 11
ostc_AttributeDescriptor

Instances of this class are used to specify **ostc_Object** attribute types and operation formal parameters.

ostc_AttributeDescriptor::getName()

virtual oscs_String getName() ;

Returns the name of the specified attribute descriptor.

ostc_AttributeDescriptor::getType()

virtual ostc::types getType() ;

Returns an enumerator indicating the type of the specified attribute descriptor.

ostc_AttributeDescriptor::getMaxLength()

virtual oscs_Uint32 getMaxLength() ;

Returns the value passed as **maxlength** in the last call to **ostc_ServerOperation::addReturnSetAttribute()** or **ostc_ServerOperation::addArgument()** that passed the name of **this** as the **name** argument. Returns **0** if there was no such call.

Chapter 12
ostc_ AttributeDescriptorList

Instances of this class are ordered collections of **ostc_ AttributeDescriptor**s. This class defines functions for the following tasks:

Cursor Validity

A list's associated cursor is *valid* if it is positioned at an element of the list, and is *invalid* if it is not positioned at any element of the list. Each list's cursor is initially invalid. Calling either **ostc_AttributeDescriptorList::first()** or **ostc_AttributeDescriptorList::next()** on the list renders the cursor valid.

Traversing Lists

ostc_AttributeDescriptorList::first()

virtual ostc_AttributeDescriptor* first() ;

Positions the list's cursor at the list's first element, and returns that element. If the list is empty, returns **0**.

ostc_AttributeDescriptorList::next()

virtual ostc_AttributeDescriptor* next() ;

If **this** list's cursor is valid and is not positioned at the list's last element, **next()**

- Advances the cursor
- Returns the element at which the cursor is positioned after being advanced

If the cursor is positioned at the list's last element, **next()**

- Renders the cursor invalid
- Returns **0**

If the cursor is invalid and the list is nonempty, **next()**

- Positions the cursor at the list's first element
- Returns the first element.

If the list is empty, **next()** returns **0**.

ostc_AttributeDescriptorList::more()

virtual oscs_Bool more() ;

Returns **oscs_True** if **this**'s cursor is valid, that is, positioned at an element of the set. Returns **oscs_False** otherwise.

Getting List Cardinality

ostc_AttributeDescriptorList::cardinality()

virtual oscs_Uint32 cardinality() ;

Returns the number of elements in the specified set.

Getting the Element with a Specified Name

ostc_AttributeDescriptorList::getAttributeDescriptor()

```
virtual ostc_AttributeDescriptor* getAttributeDescriptor(
    oscs_ConstString attrname
) ;
```

Returns the element of the list that points to the attribute descriptor with the specified name. Returns **0** if there is no such element.

Chapter 13
ostc_Object

Instances of this class are essentially attribute-name / value pairs. The class, used by both client and Server, defines members for the following tasks:

Getting an Object's Attribute Descriptors

ostc_Object::getAttributeDescriptors()

virtual ostc_AttributeDescriptorList* getAttributeDescriptors() ;

Returns an **ostc_AttributeDescriptorList** of the specified object's **ostc_AttributeDescriptor**s.

Getting an Object's Attribute Values

ostc_Object::getInt32Value()

```
virtual oscs_Int32 getInt32Value(
    oscs_ConstString attrname
) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_int32**.

ostc_Object::getInt64Value()

```
virtual oscs_Int64 getInt64Value(oscs_ConstString attrname) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_int64**.

ostc_Object::getStringValue()

```
virtual oscs_ConstString getStringValue(
    oscs_ConstString attrname
) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_string**.

ostc_Object::getFloatValue()

```
virtual oscs_Float getFloatValue(
    oscs_ConstString attrname
) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_float**.

ostc_Object::getDoubleValue()

```
virtual oscs_Double getDoubleValue(
    oscs_ConstString attrname
) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_double**.

ostc_Object::getObjectValue()

```
virtual ostc_OID * getObjectValue(
    oscs_ConstString attrname
) ;
```

Returns the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_OID***.

ostc_Object::getBinaryValue()

```
virtual void * getBinaryValue(
```

```
    oscs_ConstString attrname,
    oscs_Uint32& length
) ;
```

Returns the value of the attribute with the specified name.

length is the length of the bit stream making up the value.

If the value has not been set (by **ostc_Object::setValue()**), returns **0**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_binary**.

ostc_Object::getValue()

```
virtual void getValue(
    oscs_ConstString name,
    oscs_Int32 & value
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_int32**.

```
virtual void getValue(
    oscs_ConstString name,
    oscs_Int64 & value
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **oscs_Int64**.

```
virtual void getValue(
    oscs_ConstString name,
```

```
    oscs_String & value
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **oscs_String**.

```
virtual void getValue(
    oscs_ConstString name,
    oscs_Float & value
) ;
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **oscs_Float**.

```
virtual void getValue(
    oscs_ConstString name,
    oscs_Double & value
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **oscs_Double**.

```
virtual void getValue(
    oscs_ConstString name,
    ostc_OID*& value
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_OID**.

```
virtual void getValue(
    oscs_ConstString name,
    void*& value,
    oscs_Uint32 & length
);
```

Sets **value** to the value of the attribute with the specified name.

If the value has not been set (by **ostc_Object::setValue()**), **value** is set to 0.

length is the length of the bit stream making up the value.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the attribute with the specified name does not have the type **ostc_binary**.

Setting an Object's Attribute Values

ostc_Object::setValue()

```
virtual void setValue(
    oscs_ConstString name,
    oscs_Int32 value
) ;
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_int32**.

```
virtual void setValue(
    oscs_ConstString name,
    oscs_Int64 value
) ;
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_int64**.

```
virtual void setValue(
    oscs_ConstString name,
    oscs_ConstString value
);
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_string**.

```
virtual void setValue(
    oscs_ConstString name,
    oscs_Float value
) ;
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_float**.

```
virtual void setValue(
    oscs_ConstString name,
    oscs_Double value
) ;
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_double**.

```
virtual void setValue(
    oscs_ConstString name,
    ostc_OID* value
) ;
```

Sets the value of the attribute named **name** to **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_OID***.

```
virtual void setValue(
    oscs_ConstString name,
    void* value,
    oscs_Uint32 length
) ;
```

Sets the value of the attribute named **name** to the bit stream **value**.

length is the length of **value**.

ostc_UnknownAttribute is thrown if there is no attribute with the specified name.

ostc_DatatypeMismatch is thrown if the formal parameter with the specified name does not have the type **ostc_binary**.

Getting an Object's Session

ostc_Object::getSession()

virtual ostc_Session* getSession() ;

Returns a pointer to the current **ostc_Session**, the session that executed the operation that resulted in **this** object.

Getting and Setting Object IDs

ostc_Object::getOID()

virtual ostc_OID* getOID() ;

Returns a pointer to **this**'s **ostc_OID**. Returns **0** if the OID has not been set.

ostc_Object::setOID()

virtual void setOID(ostc_OID *) ;

Sets **this**'s OID to the specified **ostc_OID**.

Chapter 14
ostc_ObjectList

Instances of this class are ordered collections of **ostc_Object**s. This class defines functions for the following tasks:

Cursor Validity

A list's associated cursor is *valid* if it is positioned at an element of the list, and is *invalid* if it is not positioned at any element of the list. Each list's cursor is initially invalid. Calling either **ostc_ObjectList::first()** or **ostc_ObjectList::next()** on the list renders the cursor valid.

Traversing Object Lists

ostc_ObjectList::first()

virtual ostc_Object* first() ;

Positions the list's cursor at the list's first element, and returns that element. If the list is empty, returns **0**.

ostc_ObjectList::next()

virtual ostc_Object* next() ;

If **this** list's cursor is valid and is not positioned at the list's last element, **next()**

- Advances the cursor
- Returns the element at which the cursor is positioned after being advanced

If the cursor is positioned at the list's last element, **next()**

- Renders the cursor invalid
- Returns **0**

If the cursor is invalid and the list is nonempty, **next()**

- Positions the cursor at the list's first element
- Returns the first element

If the list is empty, **next()** returns **0**.

ostc_ObjectList::more()

virtual oscs_Bool more() ;

Returns **oscs_True** if **this**'s cursor is valid, that is, positioned at an element of the set. Returns **oscs_False** otherwise.

ostc_ObjectList::current()

virtual ostc_Object* current() ;

Returns the element at which the cursor is positioned, or **0** if the cursor is null. Does not reposition the cursor.

Getting Object List Cardinality

ostc_ObjectList::estimatedCardinality()

virtual oscs_Uint32 estimatedCardinality() ;

Returns the number of elements in the specified list, if **this** is nonincremental. If **this** is incremental, returns a lower bound on the cardinality (in particular, the function returns the number of C++ objects that constitute the unformatted operation result from which **this** list is derived).

Getting and Setting Object List OIDs

ostc_ObjectList::getOID()

virtual ostc_OID* getOID() ;

Returns a pointer to the OID of **this**. Returns **0** if the OID has not been set.

ostc_ObjectList::setOID()

virtual void setOID(ostc_OID *) ;

Sets the OID of **this**.

Truncating Incremental Lists

ostc_ObjectList::done()

virtual void done() ;

Call this function if you have traversed part of an incremental list, and you do not need access to the rest of the list. After the call to **done()**, the list contains just those elements that have already been returned by the Server.

Adding and Removing List Elements

ostc_ObjectList::addObject()

virtual ostc_Object * addObject() ;

Creates an **ostc_Object** and adds it to the end of the list. Returns a pointer to the new object.

ostc_ObjectList::deleteObject()

virtual void deleteObject(ostc_Object*&) ;

If called by the Server, removes the specified object from the list, and sets the argument to 0. Repositions the cursor at the element immediately before the removed element. If there is no previous element, the cursor is rendered invalid. This function has no effect if called by the thin client.

Chapter 15
ostc_OID

This class is an abstract base class. If you want to use OIDs, you must derive a class from **ostc_OID** and implement the functions listed below.

ostc_OID::stringify()

virtual oscs_ConstString stringify() const;

Returns a string that serves as a unique identifier.

ostc_OID::match

virtual oscs_Bool match(ostc_OID*) const;

Returns **1** if the OIDs designate the same object; returns **0** otherwise.

Example

```
class MyOID : public ostc_OID {
public
    MyOID(int id){_id = id;}
    int getID() {return id;}
    char * stringify() {
        sprintf(string_version, "%ld", id); return string_version;
    }
private:
    int id;
    char string_version[10];
};
```

On the Server (**format()** function):

```
MyOID oid{10);
ostc_Object * obj  = objlist->addObject();
```

```
obj->setValue("object", &oid);
```

On the client:

```
ostc_OID * oid = obj->getObjectValue("object");
MyOID oid(atoi(oid->stringify()));
```

Chapter 16
Global Functions

To create a Component Server plug-in, you must implement the following global functions:

- **ostcConnect()**
- **ostcDisconnect()**
- **ostcInitialize()**

ostc_MissingFunc is thrown if you fail to implement one of these functions.

A source or header file for your Server plug-in must include the following declaration of the global functions:

```
extern "C" {

#ifdef WIN32
__declspec( dllexport )  void ostcInitialize(void);
__declspec( dllexport )  void ostcConnect(ostc_ServerSession *);
__declspec( dllexport )  void ostcDisconnect(void);
#else
void ostcInitialize(void);
void ostcConnect(ostc_ServerSession *);
void ostcDisconnect(void);
#endif
}
```

This use of **_declspec** is required for Windows platforms.

Implementing **ostcConnect()**

ostcConnect()

void ostcConnect(ostc_ServerSession *);

Implement this function as part of your Server plug-in. The Component Server calls **ostcConnect()** when the Server first starts up.

The function must call **ostc_ServerSession::addOperation()** on the specified server session in order to add each operation the session is to execute. The function can also call other members of **ostc_ServerSession** to modify Server start-up parameters or to expose ObjectStore Inspector data views.

You can also implement this function to set routing data with **ostc_ServerSession::setUserRouteData()**.

The Server does *not* call this function within an ObjectStore transaction. You are responsible for using transactions to access persistent data within this function.

Click here for an example.

Implementing **ostcDisconnect()**

ostcDisconnect()

void ostcDisconnect();

You can use **ostcDisconnect()** to perform application-specific processing whenever the Server shuts down. You must implement this function, even if there is no such processing to perform.

The Server does *not* call this function within an ObjectStore transaction. You are responsible for using transactions to access persistent data within this function.

Implementing **ostcInitialize()**

ostcInitialize()

void ostcInitialize();

For each Server plug-in, you must implement the global function **ostcInitialize()**, which provides any initialization required before operations can be executed. Typical tasks for **ostcInitialize()** include opening databases and retrieving database roots.

The Component Server calls **ostcInitialize()** after **ostcConnect()**, when the Server first starts up.

The Server does *not* call this function within an ObjectStore transaction. You are responsible for using transactions to access persistent data within this function.

Click here for an example.

Chapter 17
ostc_ServerSession

The Server API provides members of this class for performing the following tasks:

Adding Operations to a Session

ostc_ServerSession::addOperation()

void addOperation(ostc_ServerOperation *);

Call this function within your implementation of **ostcConnect()**. Adds the specified operation to the supported operations for the current session.

Click here for an example.

Exposing Data Views

See Using and Configuring the DataView Reader on page 59.

ostc_ServerSession::exposeAllDataViews()

```
void exposeAllDataViews(oscs_ConstString dbname);
```

Windows only

Exposes all the data views in the database named **dbname**.

You can call this function several times specifying different databases. Each data view is mapped to a Server operation whose name is the same as the data view's name.

You can expose only one database with a given name at a time. If you have exposed a data view from one database, and then attempt to expose a data view with the same name from another database, the attempt is silently ignored. Only the first data view will be accessible.

ostc_ServerSession::exposeDataView()

```
void exposeDataView(
    oscs_ConstString dbname,
    oscs_ConstString viewname,
    oscs_ConstString operation_name = 0
);
```

Windows only

Exposes the data view named **viewname** in the database named **dbname**.

operation_name, if nonzero, specifies the name of the Server operation to which the data view is mapped. If **operation_name** is 0, **data_view_name** is used as the operation name.

viewname can also contain an SQL statement that customizes the data view defined in **database_name**. The SQL statement must follow SQL syntax rules. See Filtering and Ordering: SQL Support in DataView Reader on page 62.

You can expose only one database with a given name at a time. If you have exposed a data view from one database, and then attempt to expose a data view with the same name from another database, the attempt is silently ignored. Only the first data view will be accessible.

Modifying Start-up Parameters

These functions modify the start-up parameters specified at start-up for the Server for which **this** is a session.

ostc_ServerSession::setServiceName()

void setServiceName(oscs_String servicename);

Sets the name of the Server, the name by which clients can connect to it using **ostc::connect()**.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setPort()

void setPort(oscs_Int32 port);

Sets the port the Server uses to listen for client requests.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setTxnType()

void setTxnType(ostc::transaction_type txntype);

Sets the transaction type of the current session.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setUserRouteData()

void setUserRouteData(void *, oscs_Uint32 length);

Sets the Server's routing data. The **void*** can point to any user-specified data, such as a character string. A thin client can specify routing data along with a request for execution of an operation, and the request is routed to a Server with matching routing data. Matches are identified with **memcmp()**.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setNumThreads()

void setNumThreads(oscs_Uint32 numthreads);

Sets the number of threads used by the Server to process client requests.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::addRouterHostName()

void addRouterHostName(oscs_String routerhostname);

Specifies the network address for the system router.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setRouterFile()

void setRouterFile(oscs_String routerfilename);

Sets the file used by the router to maintain routes persistently, in case of failures or restarts.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setIncrementalTimeout()

void setIncrementalTimeout(oscs_Uint32 timeout);

Sets the timeout for access to incremental object sets on the Server. If the set is not accessed in the amount of time specified, the Server discards it.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setBoundTimeout()

void setBoundTimeout(oscs_Uint32 timeout);

Sets the timeout for bound transactions (that is, transactions started by the thin client with **ostc_Session::beginTxn()**). If a bound transaction is started but is not being used, it aborts when this timeout is exceeded. This prevents the Server from hanging if the client fails in the middle of a bound transaction.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setSharedTimeout()

void setSharedTimeout(oscs_Uint32 timeout);

Sets the timeout for shared (that is **shared_update**, **read**, or **mvccRead**) transactions. Shared transactions commit after a certain number of requests complete or this timeout is exceeded, whichever comes first.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setSharedMaxOps()

void setSharedMaxOps(oscs_Uint32 maxops);

Sets the number of requests that can use a shared transaction before a commit occurs. This parameter is ignored for **isolated_update** Servers.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setRouterPingInterval()

void setRouterPingInterval(oscs_Uint32 interval);

A router pings all its routes at this interval. This parameter is ignored for Servers that are not routers.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ServerSession::setServerLogging()

void setServerLogging(oscs_Bool logging);

Turns Server logging on or off. With logging on, the Server writes information to the file **server.log** in the directory from which the Server was executed.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

Chapter 18
ostc_ServerOperation

This class defines the protocol for members you must implement in derived classes:

- **execute()**
- **format()**

It also provides functions for the following tasks:

Implementing **execute()**

ostc_ServerOperation::execute()

```
virtual void execute(
    ostc_Object * arguments,
    ostc_OperationResult* result
) = 0;
```

For each operation you want a Server to handle, you must define a subtype of **ostc_ServerOperation**, and implement the member **execute()** (among other functions). The component Server calls this function in response to thin client calls to **ostc_Session::doOperation()** for the operation.

ostc_ServerOperation::execute() is a pure virtual that specifies the protocol for the function you must implement.

The function must extract the arguments by calling a get-value member of **ostc_Object** on **arguments**, for each attribute of **arguments**.

It must then process the arguments and set the result (in the form of C++ objects) by performing members of **ostc_OperationResult** on **result**.

The Server calls this function within an ObjectStore transaction. You do not have to start a transaction explicitly in order to access persistent data within this function.

Your implementation of **execute()** must be reentrant. That is, you are responsible for synchronizing access to any transient state that could be shared across Server threads. In addition, for operations that use **shared_update** transactions, you must synchronize access to persistent data if that is necessary to prevent interference among threads.

Click here for an example.

Implementing **operationComplete()**

ostc_ServerOperation::operationComplete()

virtual void operationComplete(ostc_OperationResult*) const;

Performs any required cleanup after **execute()** and associated executions of **format()** complete.

For a given class that you derive from **ostc_ServerOperation**, you can either implement this function in the subtype, or use the default version as inherited from **ostc_ServerOperation**.

The Component Server calls this function after the operation results have been formatted and sent to the thin client. If the result is incremental, the Server calls the function after the first batch of results has been sent.

The default version calls **delete** on the **os_cursor*** and **os_collection*** (if nonzero) that **execute()** passes to **ostc_OperationResult::setReturnValue()**:

```
void ostc_ServerOperation::operationComplete(
    ostc_OperationResult * result
) const
{
    os_collection * coll;
    os_cursor * cursor;

    result->getReturnValue(coll, cursor);

    if (coll)
        delete coll;

    if (cursor)
        delete cursor;
}
```

The Server calls this function within an ObjectStore transaction. You do not have to start a transaction explicitly in order to access persistent data within this function.

Your implementation of **operationComplete()** must be reentrant. That is, you are responsible for synchronizing access to any transient state that could be shared across Server threads. In addition, for operations that use **shared_update** transactions, you must synchronize access to persistent data if that is necessary to prevent interference among threads.

Implementing **format()**

ostc_ServerOperation::format()

```
virtual void format(
    const void* value,
    ostc_ObjectList* values,
    ostc_OperationResult* result
) =0;
```

Converts C++ objects that result from **execute()** into **ostc_Object**s.

For a given class that you derive from **ostc_ServerOperation**, you must implement this function in the subtype. **ostc_ServerOperation::execute()** is a pure virtual that specifies the protocol for the function you must implement.

value is the object being converted.

values is an initially empty list of **ostc_Object**s.

result is the operation result whose return values are being formatted.

The function should call **ostc_ObjectList::addObject()** to add to **values** the object or objects that result from converting **value**.

After **execute()** completes, the Component Server calls this function for each return value.

If **execute()** passes a nonzero **void*** to **ostc_OperationResult()::setReturnValue()**, the **void*** is considered to be the only return value, so the Server calls **format()** once, passing the **void*** as the **value** argument.

If **execute()** passes a nonzero **os_cursor*** to **ostc_OperationResult()::setReturnValue()** (and does not pass a nonzero **void*** to **ostc_OperationResult()::setReturnValue()**), the Server calls **format()** on each element that can be visited with the cursor.

If **execute()** passes a nonzero **os_collection*** to **ostc_OperationResult()::setReturnValue()** (and does not pass a nonzero **void*** or **os_cursor*** to **ostc_OperationResult()::setReturnValue()**), the Server calls **format()** on each element of the collection.

The Server calls this function within the same ObjectStore transaction in which **execute()** is performed (unless the operation

result is incremental). You do not have to start a transaction explicitly in order to access persistent data within this function.

Your implementation of **format()** must be reentrant. That is, you are responsible for synchronizing access to any transient state that could be shared across Server threads. In addition, for operations that use **shared_update** transactions, you must synchronize access to persistent data if that is necessary to prevent interference among threads.

Click here for an example.

Setting Operation Formal Parameters

ostc_ServerOperation::addArgument()

```
void addArgument(
    oscs_ConstString name,
    ostc::types type,
    oscs_Uint32 maxlength = 0
);
```

Adds an argument named **name** to the end of the list of arguments of **this** operation. The argument's type is specified by **type**.

If **type** is **ostc_string**, **maxlength** specifies the argument's maximum length. Actual argument values are truncated when necessary to adhere to this maximum. If **type** is not **ostc_string**, **maxlength** is ignored.

See also **ostc_OperationResult::getArguments()** on page 157.

ostc_ServerOperation::addReturnSetAttribute()

```
void addReturnSetAttribute(
    oscs_ConstString name,
    ostc::types type,
    oscs_Uint32 maxlength = 0
);
```

Adds an attribute named **name** to the end of the list of attributes of objects returned by **this** operation. The attribute's value type is specified by **type**.

If **type** is **ostc_string**, **maxlength** specifies the attribute's maximum length. Actual attribute values are truncated when necessary to adhere to this maximum. If **type** is not **ostc_string**, **maxlength** is ignored.

See also **ostc_OperationResult::GetReturnedObjectAttributes()** on page 157.

Setting the Description of an Operation

ostc_ServerOperation::setDescription()

void setDescription(oscs_ConstString description);

Associates the specified string with this operation. The string can be retrieved by the client with **ostc_Operation::getDescription()**.

Getting the Name of an Operation

ostc_ServerOperation::getName()

oscs_String getName();

Returns the name of this operation. See **ostc_ServerOperation::ostc_ServerOperation()** on page 153.

Creating and Deleting Server Operations

ostc_ServerOperation::ostc_ServerOperation()

```
ostc_ServerOperation(
    oscs_ConstString name,
    ostc::transaction_type type)
);
```

Creates an **ostc_ServerOperation** with the specified name. When you define a subtype of **ostc_ServerOperation**, you must pass the operation name and transaction type to this constructor.This name is used to identify the operation when you invoke **execute()**.

ostc_ServerOperation::~ostc_ServerOperation()

```
virtual ~ostc_ServerOperation();
```

Frees memory associated with **this** operation object.

Chapter 19
ostc_OperationResult

Instances of this class represent operation results in the form of C++ objects. The Server calls **ostc_ServerOperation::format()** on each C++ object to convert the results to **ostc_Object**s. The class provides functions for

Setting and Getting the Return Value or Values

ostc_OperationResult::setReturnValue()

void setReturnValue(os_collection*, os_cursor* = 0);

Call this function from **ostc_ServerOperation::execute()** to specify multiple return values.

If **execute()** passes a nonzero **os_cursor*** to **ostc_OperationResult()::setReturnValue()** (and does not pass a nonzero **void*** to **setReturnValue()**), the Server calls **format()** on each element that can be visited with the cursor.

If **execute()** passes a nonzero **os_collection*** to **ostc_OperationResult()::setReturnValue()** (and does not pass a nonzero **void*** or **os_cursor*** to **setReturnValue()**), the Server calls **format()** on each element of the collection.

void setReturnValue(void*);

Call this function from **ostc_ServerOperation::execute()** to specify a single return value.

If **execute()** passes a nonzero **void*** to **setReturnValue()**, the **void*** is considered to be the only return value, so the Server calls **format()** once, passing the **void*** as the **value** argument.

If you pass 0 to this function, **ostc_ServerOperation::format()** is not called, and (on the client side) **ostc_Session::doOperation()** returns **0**.

ostc_OperationResult::GetReturnValue()

void GetReturnValue(os_collection*&, os_cursor*&);

Sets the arguments to the values passed to **setReturnValue()**. Sets the arguments to 0 if **setReturnValue(os_collection*, os_cursor*)** has not been called on **this** result. This function can be called from **ostc_ServerOperation::operationComplete()**.

void GetReturnValue(void*&);

Sets the argument to the value passed to **setReturnValue()**. Sets the argument to 0 if **setReturnValue(void*)** has not been called on **this** result. This function can be called from **ostc_ServerOperation::operationComplete()**.

Getting Operation Formal Parameters

ostc_OperationResult::getArguments()

const ostc_AttributeDescriptorList* getArguments();

Returns a pointer to a list of **ostc_AttributeDescriptor**s representing the formal parameters of the operation for which **this** is the result.

ostc_OperationResult::GetReturnedObjectAttributes()

const ostc_AttributeDescriptorList* GetReturnedObjectAttributes();

Returns a pointer to a list of attribute descriptors representing the attributes of each object returned by the operation for which **this** is the result.

Setting and Getting Result IDs

ostc_OperationResult::setOID()

void setOID(ostc_OID *);

Returns a pointer to the OID of **this**. Returns **0** if the OID has not been set.

ostc_OperationResult::getOID()

ostc_OID* getOID();

Sets the OID of **this**.

Chapter 20
ostc_ApplicationServer

You can use members of this class to create a custom component server that manages start-up and shutdown of ObjectStore Component Servers. This class provides members for the following tasks:

Do not call these functions from your implementation of Server API functions.

A minimal custom server can be coded as follows:

```
#include <ostc/ostcappsrvr.h>
main(int argc, char** argv)
{
    ostc_ApplicationServer * server =
        new ostc_ApplicationServer(argc, argv);

    server->start();

    sleep();
}
```

Creating and Starting ObjectStore Component Servers

ostc_ApplicationServer::ostc_ApplicationServer()

ostc_ApplicationServer(int argc, char ** argv);

Constructs an ObjectStore Component Server with the parameters specified on the command line.

```
ostc_ApplicationServer(
   oscs_String * plugin_names,
   oscs_Uint32 num_plugins,
   oscs_String service_name,
   oscs_Int32 port,
   ostc::transaction_type txntype
);
```

Constructs an ObjectStore Component Server with the required parameters specified as constructor arguments.

ostc_ApplicationServer::start()

void start();

Starts the Server.

Stopping Component Servers

ostc_ApplicationServer::~ostc_ApplicationServer()

~ostc_ApplicationServer();

Shuts down **this** ObjectStore Component Server.

Adding Components to a Server

ostc_ApplicationServer::ostc_ApplicationServer()

```
ostc_ApplicationServer(
    oscs * framework,
    oscs_String * plugin_names,
    oscs_Uint32 num_plugins,
    oscs_String service_name,
    ostc::transaction_type txntype
);
```

Adds the components named in **plugin_names** to the Server associated with **framework**. **ostc_ApplicationServer::getFramework()** returns a Server's associated framework.

ostc_ApplicationServer::getFramework()

```
oscs * getFramework();
```

Returns the instance of **oscs** associated with **this**. Use this function in conjunction with **ostc_ApplicationServer::ostc_ApplicationServer()** to add a component to a Server.

Modifying Parameters

ostc_ApplicationServer::setUserRouteData()

void setUserRouteData(void *, oscs_Uint32 length);

Sets the Server's routing data. The **void*** can point to any user-specified data, such as a character string. A thin client can specify routing data along with a request for execution of an operation, and the request is routed to a Server with matching routing data. Matches are identified with **memcmp()**.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setNumHandlingThreads()

void setNumHandlingThreads(oscs_Uint32 numthreads);

Sets the number of threads used by the Server to process client requests.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::addRouter()

void addRouterHostName(oscs_String routerhostname);

Specifies the network address for the system router.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setRouterFile()

void setRouterFile(oscs_String routerfilename);

Sets the file used by the router to maintain routes persistently, in case of failures or restarts.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setIncrementalTimeout()

void setIncrementalTimeout(oscs_Uint32 timeout);

Sets the timeout for access to incremental object sets on the Server. If the set is not accessed in the amount of time specified, the Server discards it.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setBoundTxnTimeout()

void setBoundTxnTimeout(oscs_Uint32 timeout);

Sets the timeout for bound transactions (that is, transactions started by the thin client with **ostc_Session::beginTxn()**). If a bound transaction is started but is not being used, it aborts when this timeout is exceeded. This prevents the Server from hanging if the client fails in the middle of a bound transaction.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setSharedTxnTimeout()

void setSharedTxnTimeout(oscs_Uint32 timeout);

Sets the timeout for shared (that is **shared_update**, **read**, or **mvccRead**) transactions. Shared transactions commit after a certain number of requests complete or this timeout is exceeded, whichever comes first.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setSharedMaxOps()

void setSharedMaxOps(oscs_Uint32 maxops);

Sets the number of requests that can use a shared transaction before a commit occurs. This parameter is ignored for **isolated_update** Servers.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setRouterPingInterval()

void setRouterPingInterval(oscs_Uint32 interval);

A router pings all its routes at this interval. This parameter is ignored for Servers that are not routers.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

ostc_ApplicationServer::setLogging()

void setLogging(oscs_Bool logging);

Turns Server logging on or off. With logging on, the Server writes information to the file **server.log** in the directory from which the Server was executed.

otcs_ArgsLocked is thrown if this function is called outside the dynamic scope of **ostcConnect()**.

Chapter 21
Exception Reference

The client and Server APIs signal error conditions by throwing C++ exceptions. **oscs_Exception** is an ancestor of all exceptions that the APIs can throw (note that the prefix is **oscs**, *not* **ostc**).

Below is a list of all exceptions, together with the APIs that can throw each one:

Exception	***Thrown By***
ostc_DatatypeMismatch	**ostc_Operation::getInt32Argument()** **ostc_Operation::getInt64Argument()** **ostc_Operation::getStringArgument()** **ostc_Operation::getFloatArgument()** **ostc_Operation::getDoubleArgument()** **ostc_Operation::getObjectArgument()** **ostc_Operation::getBinaryArgument()** **ostc_Operation::setArgument()**
ostc_InvalidParm	**ostccompsrvr**
ostc_InvalidTxnType	**ostc_Session::beginTxn()**
ostc_MissingLib	**ostccompsrvr**

ostc_NoArguments	**ostc_Operation::getInt32Argument()** **ostc_Operation::getInt64Argument()** **ostc_Operation::getStringArgument()** **ostc_Operation::getFloatArgument()** **ostc_Operation::getDoubleArgument()** **ostc_Operation::getObjectArgument()** **ostc_Operation::getBinaryArgument()**
ostc_NoNestedTxn	**ostc_Session::beginTxn()**
ostc_NoTxnInProgress	**ostc::connect()** **ostc_Session::commitTxn()** **ostc_Session::abortTxn()**
ostc_RequiredParmMissing	**ostccompsrvr**
ostc_ServerException	**ostc_Session::doOperation()**

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W