Sun StorEdge™ Media Central Client Programmer's Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 Fax 650 969-9131

Part No. 806-1798-10
October 1999, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Contents

Preface xiii

1. Introducing Media Central Clients 1

Media Central Clients and Servers 1

Video Server Objects 3

VideoBeans Objects and their Proxies 3

Factories 3

Event Channels 4

2. Hello, World Examples 5

Introducing the Examples 5

HelloFactories: Listing Factories 6

HelloBean: Using a Factory and a Proxy 8

HelloEvent: Handling Events 11

HelloVtr: Playing a Video Tape 17

3. Infrastructure Classes and Services 21

Properties 21

Common Classes 22

`MediaContent` and `ContentLib` 22

`LatencyInfo` 23

Timecode 24

Events and Event Channels 24

Factories and Naming 27

Access Control 28

Resource Recovery 28

Standard Proxy Methods 29

Exceptions 30

4. VideoBeans Catalog 33

Player 33

Methods 35

Events 35

Recorder 36

Methods 37

Events 38

ContentLib 38

Methods 39

Events 39

Importer 40

Methods 40

Events 40

Exporter 41

Methods 41

Events 41

Migrator 42

Methods 42

Events 42

Vtr 42

Methods 43

Events 43

5. Javadoc Guide 45

com.sun.videobeans.directory 45

com.sun.videobeans.util 46

com.sun.videobeans 46

com.sun.videobeans.beans 46

com.sun.videobeans.event 47

com.sun.videobeans.security 47

com.sun.broadcaster.vssmbeans 47

com.sun.broadcaster.vssmproxy 48

com.sun.broadcaster.vtrproxy 49

Index 51

Figures

FIGURE 1-1 Media Central Client/Server Architecture 2

FIGURE 1-2 Media Central Client/Server Calling Relationships 3

Tables

TABLE 3-1 `MediaContent` Metadata 22

TABLE 3-2 `ContentLib` Methods for `MediaContent` Objects 23

TABLE 3-3 Event Codes 24

TABLE 3-4 Standard Proxy Methods 29

TABLE 4-1 Principal `Player` Methods 35

TABLE 4-2 Principal `Recorder` Methods 37

TABLE 4-3 Principal `ContentLib` Methods 39

TABLE 4-4 Principal `Importer` Methods 40

TABLE 4-5 Principal Exporter Methods 41

TABLE 4-6 Principal `Migrator` Methods 42

TABLE 4-7 Principal Vtr Methods 43

Code Samples

CODE EXAMPLE 2-1 `HelloFactories.java` Source 6

CODE EXAMPLE 2-2 `HelloBean.java` Source 8

CODE EXAMPLE 2-3 `HelloEvent.java` Source 11

CODE EXAMPLE 2-4 `HelloVtr.java` Source 17

Preface

The *Sun StorEdge Media Central Client Programmer's Guide* is for Java™ programmers who want to write applications that use the Sun StorEdge™ Media Central video file server.

Before You Read This Book

Read the *Sun StorEdge Media Central Release Notes* in conjunction with this guide. Limitations and problems described in the *Release Notes* override instructions and features described in this guide.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- *Solaris Handbook for Sun Peripherals*
- AnswerBook2™ online documentation for the Solaris™ operating environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Installation	*Sun StorEdge Media Central Installation and Configuration Guide*	806-1800-05
Using Client and Server Software	*Sun StorEdge Media Central User's Guide*	806-1799-05
All	*Sun StorEdge Media Central Release Notes*	806-1797-05

For the latest Media Central information, consult the product web site:

```
http://www.sun.com/storage/media-central
```

Accessing Sun Documentation Online

The `docs.sun.com`[sm] web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

```
http://docs.sun.com
```

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

```
docfeedback@sun.com
```

Please include the part number (806-1798-10) of your document in the subject line of your email.

CHAPTER 1

Introducing Media Central Clients

This chapter introduces the main elements of a Sun StorEdge™ Media Central client. A Media Central client is a Java class or program that interacts with one or more Media Central servers to accomplish a task or provide a user with a service.

The chapter covers these topics:

- "Media Central Clients and Servers" on page 1
- "Video Server Objects" on page 3

If you want to see examples of very simple clients before you read this introduction, skip to Chapter 2.

Note – The application programming interfaces (APIs) described in this guide are subject to change without notice.

Media Central Clients and Servers

The Media Central software architecture follows the client/server model. The following figure shows the general arrangement. A user interacts with one or more clients that typically provide graphical interfaces to server operations. For examples of clients, see the *Sun StorEdge Media Central User's Guide*.

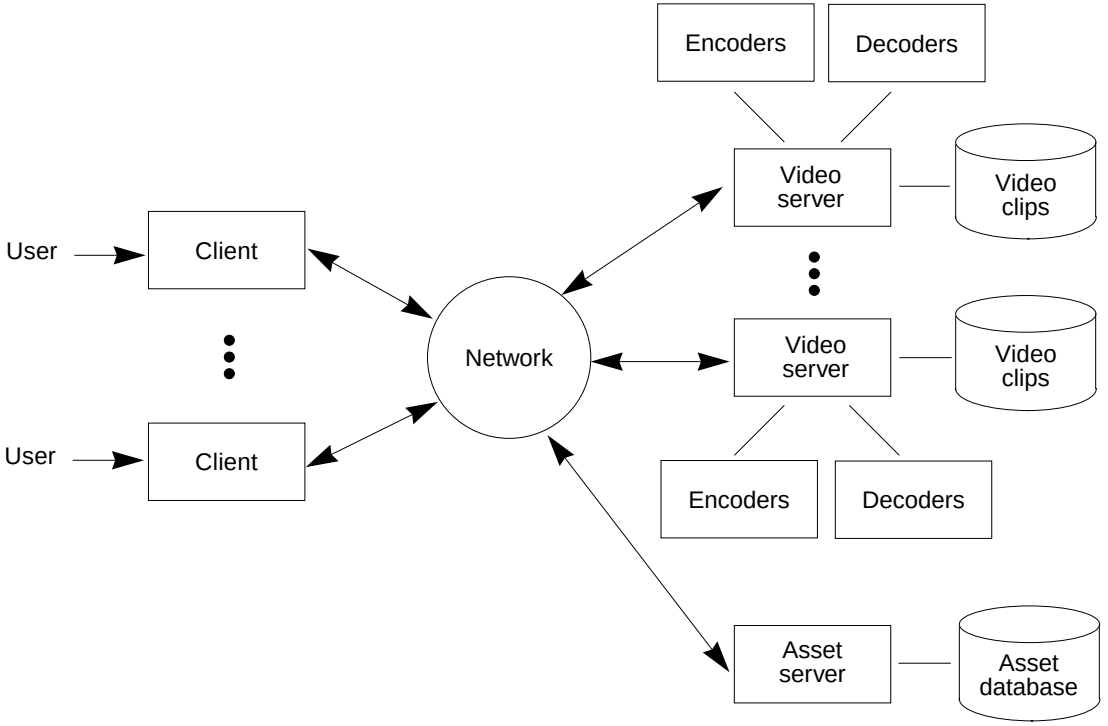

FIGURE 1-1 Media Central Client/Server Architecture

There are two kinds of Media Central servers. A *video server* (often called simply "a server") captures, stores, and streams video clips, and controls video devices such as encoders. A video server constitutes a dedicated, real-time system that runs on a general-purpose operating system (the Solaris™ operating environment) and hardware (Sun servers). The *asset server* (also called, in a programming context, the user metadata server) cooperates with video servers to provide descriptive, searchable information on clips, such as title and creator. The asset server is an optional component; it can run on any Sun system that is not also hosting a video server. The asset server uses a database server that supports the Java™ Database Connectivity (JDBC™) protocol.

A client can communicate with any Media Central video server on the network; that server and its video equipment can be next door or in another building. Servers use the network to communicate among themselves. To a client, a video server is a collection of Java objects that are callable via the Java remote method invocation (RMI) service (see the following figure). For example, to schedule a video server to play a clip to a decoder, a client calls `Player.startStreamAt()`. A server can also call client objects to notify them of events; for example, that a `Player` has finished playing.

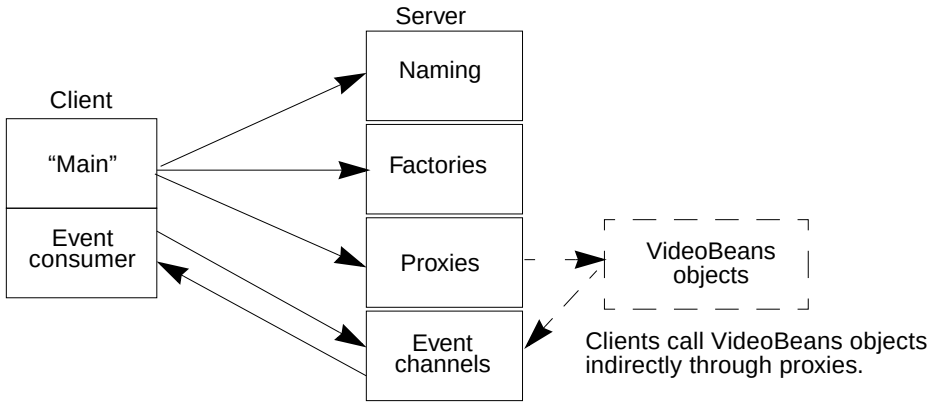

FIGURE 1-2 Media Central Client/Server Calling Relationships

Video Server Objects

A Media Central video server's client facilities are represented in objects that clients call across the network. Video servers provide four main kinds of objects for clients.

VideoBeans Objects and their Proxies

A *VideoBeans™ object* is an interface to a physical or virtual device. A `ContentLib` VideoBeans object, for example, represents a server's video file system; it holds video clips. (To a user, a `ContentLib` is a Clip Folder.) A Player is another kind of VideoBeans object; it streams a video clip from a `ContentLib` to a decoder. There are currently seven kinds of VideoBeans objects; more may be added.

Clients call VideoBeans methods indirectly through objects called *proxies*. Proxies insulate VideoBeans objects from network management and provide services that are independent of VideoBeans type. To a client, a proxy for a VideoBeans object *is* the VideoBeans object.

Factories

To use a VideoBeans object, a client needs a proxy for it; to obtain a proxy, a client calls a factory object's `createBean()` method. A *factory* returns a reference to a proxy. On each server, there is a factory for each VideoBeans object that represents a

piece of hardware. For example, a decoder device is represented by a `Player` VideoBeans object. If there are eight decoders attached to a host, there is a `Player` factory for each of them. Because decoders are single-user devices, a `Player` factory will create a `Player` instance only if no `Player` instance already exists. When a `Player` client is finished with a `Player`, it closes the `Player` to signal the factory that it can create another instance.

Some VideoBeans objects represent virtual rather than physical devices; an `Importer`, for example, copies a UNIX file into a `ContentLib` clip. Unlike a `Player`, an `Importer` VideoBeans object does not have a dedicated underlying hardware device; the single `Importer` factory will create multiple `Importer` instances if asked.

Factories have names, which are URLs. For example, on video server host `alpha`, a `Player` factory might have this name:

```
vbm://alpha/Player/VELA.MPEG.2000-04001.0
```

The protocol portion of the URL (`vbm`) stands for VideoBeans Manager. The trailing numeric part of the URL (`04001.0`) distinguishes this `Player` factory from other `Player` factories on the same server. Because factory names can be awkward to remember and understand, they can be given *aliases* with the Media Central Administrator. The same factory might have this alias:

```
vbm://alpha/Alias/Decoder1
```

To obtain a reference for a factory, a client passes the factory's URL to the `Naming` class's `lookup()` method. (The `Naming` class is a client-side class that has only static methods, so the client needs no reference to a `Naming` instance.) The `Naming` class has other methods that list factories by type or list all factories on a server or on all servers in the network.

Event Channels

VideoBeans objects post objects called *events* to signal changes in their state. For example, when a `Player` VideoBeans object stops, it posts a `STOPPED` event. "Posting an event" means calling a method in all *event channels* that have registered interest in that kind of event, passing the event as an argument. The event channels, in turn, call methods in registered client objects, passing them the events. The client objects that channels invoke are called event *consumers*; they implement a single method called `handleEvent()`, which the Media Central software arranges to have called in it own thread. By intermediating the flow of events between VideoBeans objects and clients, event channels ensure that VideoBeans objects are not held up by network or other delivery problems. Event channels also give developers a great deal of flexibility; clients can create any number of channels, can register a channel with any number of VideoBeans objects, and can register their consumers with any combination of channels created by themselves or other clients.

CHAPTER 2

Hello, World Examples

This chapter contains simple client programming examples. It covers these topics:

- "Introducing the Examples" on page 5
- "HelloFactories: Listing Factories" on page 6
- "HelloBean: Using a Factory and a Proxy" on page 8
- "HelloEvent: Handling Events" on page 11
- "HelloVtr: Playing a Video Tape" on page 17

Introducing the Examples

The examples in this chapter are very simple, similar in concept to the "Hello, world" example from Kernighan and Ritchie's *The C Programming Language*. Each example includes a description of its logic, the Java code, and sample runs in the C shell and the Bourne shell.

If the examples have been installed, they are located in this directory:

installdir`/doc/example`

The default value for *installdir* is `/opt/MediaCentral`. You install the examples when you install the Media Central documentation; see the *Sun StorEdge Media Central Installation and Configuration Guide* for details.

HelloFactories: Listing Factories

The program shown in the following code example asks a Media Central video server to list its factories.

CODE EXAMPLE 2-1 `HelloFactories.java` Source

```
// HelloFactories.java
import com.sun.videobeans.directory.*;

public class HelloFactories
{
    public static void main(String[] args)
    {
    if (args.length>0)
        Naming.setBootstrap(args[0]);
    try {
        String[] types = Naming.listTypes(“vbm”, false);
        for (int i=0; i<types.length; i++)
    {
        System.out.println(“type = “ + types[i]);
        String[] urls = Naming.list(“vbm”, types[i], false);
        for (int j=0; j<urls.length; j++)
       System.out.println(“    URL = “ + urls[j]);
    }
    }
    catch (Exception e) { e.printStackTrace(); }
    }
}
```

HelloFactories takes a single argument, the name of a video server host. It passes the host name to `Naming.setBootstrap()`, which sets up the naming facility on a server. It then obtains the types and URLs of the VideoBeans factories on the server and prints them.

The following examples show how to compile and run the HelloFactories example in the C shell and the Bourne shell. Observe the following:

- `HelloFactories.java` is assumed to be in the current directory.
- *installdir* stands for the directory in which the Media Central client software is installed. By default, the directory is `/opt/MediaCentral`.

- *host* stands for a machine running a Media Central video server.

C shell:

```
% setenv CLASSPATH .:installdir/classes/bw.jar
% javac HelloFactories.java
% java HelloFactories host
type = Exporter
    URL = vbm://host/Exporter/Vssmx
type = Importer
    URL = vbm://host/Importer/Vssmx
type = MetadataManager
    URL = vbm://host/MetadataManager/synchronizer
type = Migrator
    URL = vbm://host/Migrator/Vssmx
...
```

Bourne shell:

```
$ CLASSPATH=.:installdir/classes/bw.jar
$ export CLASSPATH
$ javac HelloFactories.java
$ java HelloFactories host
type = Exporter
    URL = vbm://host/Exporter/Vssmx
type = Importer
    URL = vbm://host/Importer/Vssmx
type = MetadataManager
    URL = vbm://host/MetadataManager/synchronizer
type = Migrator
    URL = vbm://host/Migrator/Vssmx
...
```

HelloBean: Using a Factory and a Proxy

The following code example shows a minimal Media Central client. It finds a `Player` factory, obtains a proxy for a `Player` from the factory, calls a proxy method, and displays the result.

CODE EXAMPLE 2-2 `HelloBean.java` Source

```
// HelloBean.java
import com.sun.broadcaster.vssmproxy.*;
import java.rmi.RemoteException;
import com.sun.videobeans.security.*;
import com.sun.videobeans.directory.Naming;
import com.sun.broadcaster.vssmbeans.TimecodeFormat;

/* Connect to a Player factory, instantiate a Player proxy
 * and retrieve a property of the Player
 */
public class HelloBean
{
    public static void main(String args[])
    {
    if (args.length < 1)
        {
            System.err.println("HelloBean <VBM Player URL>");
            System.err.println(" URL Example : "+
           "vbm://<hostname>/Player/VELA.MPEG.2000-0401.0");
            return;
        }
    HelloBean test = new HelloBean();
    test.runTest(args[0]);
    System.exit(0);
    }

    public void runTest(java.lang.String player_url)
    {
    try {
        // Connect to a Factory of the specified Player
        PlayerFactory fact =
(PlayerFactory)Naming.lookup(player_url);
```

CODE EXAMPLE 2-2 `HelloBean.java` Source *(Continued)*

```
        // A real client should pass genuine userid, hostname,
password
        GranteeContext ctx =
            new GranteeContextImpl(“user”, “host”, “pword”);
        Credential cdt = fact.createCredential(ctx);

        // Get a Player proxy
        PlayerProxy player = fact.createBean(cdt);

        // Set the time code format first
        player.setDefaultTimecodeFormat(TimecodeFormat.VITC);

        // Get a property value: TimecodeFormat
        TimecodeFormat tc = player.getDefaultTimecodeFormat();
        String tcf = tc.toIDString();
        System.out.println(“TimecodeFormat = “+ tcf);

        // Close the Player so someone else can use it
        player.close();
    }
    catch (RemoteException re) {
        System.out.println(“RemoteException: “+ re);
        re.printStackTrace();
        System.exit(0);
    }
    catch (java.io.IOException ie) {
        ie.printStackTrace();
        System.exit(0);
    }
    }
}
```

When you invoke `HelloBean`, you pass a URL argument that names a `Player` factory. To get a reference to the `Player` factory object, `runTest()` calls `Naming.lookup()`, passing the URL.

Note – This example, for simplicity, passes artificial fixed values for user, host, and password to the `GranteeContext` constructor. These values work in the current Media Central release, which does not implement a security policy. A future Media Central release will require actual user, host, and password values. Developers should write client code that passes legitimate values to minimize disruption when the Media Central software implements security.

Having created a credential, the program calls the factory's `createBean()` method to get a reference to the `Player` proxy. Then the program sets the proxy's `DefaultTimecodeFormat` property, asks the proxy for the value of the same property, and prints it. Finally, the program closes the `Player` proxy to free resources allocated to it by the server and to allow another client to create a proxy for the same `Player`.

Before running HelloBean, as described next, use the Administrator to set the `TimecodeFormat` property of the Decoder VideoBean whose URL you will pass to HelloBean. (Users see `Player` VideoBeans objects as Decoders.) The *Sun StorEdge Media Central User's Guide* describes the Administrator.

The following examples show how to compile and run the HelloBean example in the C shell and the Bourne shell. Observe the following:

- `HelloBean.java` is assumed to be in the current directory.
- *installdir* stands for the directory in which the Media Central client software is installed. By default it is `/opt/MediaCentral`.
- *host* stands for a machine running a Media Central server with a Vela decoder.
- Your host may not have a decoder named `VELA.MPEG.2000-0401.0`. To get a list of your host's factories, compile and run the program described in "HelloFactories: Listing Factories" on page 6.

C shell:

```
% setenv CLASSPATH .:installdir/classes/bw.jar
% javac HelloBean.java
% java HelloBean vbm://host/Player/VELA.MPEG.2000-0401.0
TimecodeFormat = VITC
```

Bourne shell:

```
$ CLASSPATH=.:installdir/classes/bw.jar
$ export CLASSPATH
$ javac HelloBean.java
$ java HelloBean vbm://host/Player/VELA.MPEG.2000-0401.0
TimecodeFormat = VITC
```

HelloEvent: Handling Events

The following code example shows the essentials of client event handling. The program creates a `MediaContent` (clip) and deletes it. Creation triggers three events (`CREATED`, `RESIZED`, and `METADATA_CHANGED`); deletion triggers one event (`REMOVED`). The program displays the contents of the four events that the `ContentLib` generates.

CODE EXAMPLE 2-3 `HelloEvent.java` Source

```
// HelloEvent.java
import com.sun.broadcaster.vssmproxy.*;
import com.sun.videobeans.directory.Naming;
import com.sun.videobeans.event.*;
import com.sun.videobeans.security.*;
import java.rmi.RemoteException;

public class HelloEvent
{
    public static void main(String args[])
    {
    if (args.length < 1)
    {
    System.err.println("HelloEvent <ContentLib URL>");
    System.err.println("  <URL> = vbm://<host>/ContentLib/<type>/<name>");
    return;
    }
    HelloEvent test = new HelloEvent();
    test.runTest(args[0]);
    System.exit(0);
    }

    public void runTest(java.lang.String urlstr)
    {
    ContentLibFactory fact = null;

    // Bind to the Factory
    try {
        fact = (ContentLibFactory)Naming.lookup(urlstr);
        }
    catch (java.io.IOException e) {
```

CODE EXAMPLE 2-3 HelloEvent.java Source *(Continued)*

```
        System.out.println("ContentLibFactory was not found in registry");
        System.exit(0);
        }

    ContentLibProxy lib = null;
    try {
        // A real client should pass genuine userid, hostname, password
        GranteeContext ctx =
           new GranteeContextImpl("user", "host", "pword");
        Credential cdt = fact.createCredential(ctx);

        // open the contentlib
        lib = fact.createBean(cdt);
        System.out.println("    ContentLib is opened");

        // register event channel with the ContentLib
        Channel ch = fact.getEventChannel("test", "contentlib", 1);
        lib.registerEventChannel(ch, null);

        // register event with MyCallBack class
        MyCallBack cb = new MyCallBack();
        ConsumerImpl consumer = new ConsumerImpl(cb);
        ChannelHelper helper = new ChannelHelper(ch);
        String cookie = new String("It is me");
        helper.registerConsumer(consumer, cookie);

        // name of the MediaContent
        String myname = new String("HelloEvent-test");

        // create new MediaContent - ContentLib.CREATED event
        System.out.println("    Create a MediaContent");
        lib.createMediaContent(myname, 100);

        // wait for the event
        java.lang.Thread.sleep(5000);

        // delete the clip
        System.out.println("    Delete the MediaContent");
        lib.deleteMediaContent(myname);

        // close it
        lib.close();
        }
```

CODE EXAMPLE 2-3 `HelloEvent.java` Source *(Continued)*

```
    catch (java.lang.InterruptedException e)
        { /* ignore exception thrown by sleep() */ }
    catch (RemoteException e) {
        System.out.println("testContentLib(): RemoteException: " + e);
        e.printStackTrace();
        }
    }

    /** inner class to monitor event
   */
    private class MyCallBack implements ConsumerCallBack
    {
    public void handleEvent(java.io.Serializable sender, Event ev)
    {
        System.out.println(" !!!! Got an Event type     = " + ev.type);
        System.out.println(" !!!!              code     = " + ev.code);
        System.out.println(" !!!!              Time     = " + ev.time);
        System.out.println(" !!!!              Info     = " + ev.info);
        System.out.println(" !!!!              cookie   = " + ev.cookie);
        System.out.println(" !!!!              sender   = " + sender);
    }
    }
}
```

This example is essentially identical to HelloBean up through getting a reference to a `ContentLib` (instead of a `Player`) proxy.

Note – This example, like HelloBean, passes artificial credential data; a real client should pass real data obtained from its user.

The rest of the program, which is the part that deals with events, proceeds as follows:

1. The client gets an event channel from the `ContentLib` factory.
2. The client registers the event channel with the `ContentLib` proxy so the `ContentLib` VideoBeans object will send events to the channel.
3. The client instantiates the client's event consumer, a class called `MyCallBack`, which implements the `ConsumerCallBack` interface. The event channel delivers events by calling this object's `handleEvent()` method.

4. The client creates a `ConsumerImpl` for the `MyCallBack` object. The `ConsumerImpl` does housekeeping work that the client developer does not need to know about.

5. The client creates a `ChannelHelper` for the event channel. The `ChannelHelper` is another housekeeping object.

6. The client creates a cookie, a string that will be returned with events from this channel-VideoBeans object combination. Although not needed in this example, a cookie is a useful identifier when events from different VideoBeans objects are multiplexed through a common channel.

7. The client registers the `ConsumerImpl` and the cookie with the `ChannelHelper`.

8. The event channel calls `MyCallBack.handleEvent()` when the `ContentLib` sends an event to the channel. The program directs its `ContentLib` proxy to create a `MediaContent` and then sleeps to give the `ContentLib` time to perform the operation and emit the `CREATED` event. When the sleep concludes, the main method closes the `ContentLib` proxy, which signals the server to release resources allocated to it.

9. Shortly after the main method calls `lib.createMediaContent()`, the program's `MyCallBack.handleEvent()` is invoked and prints the fields in the event it is passed. In particular, it prints the cookie created earlier by the main method: `cookie = It is me`. Notice that `handleEvent()` is short and does not call a server method.

The following examples show how to compile and run the HelloBean example in the C shell and the Bourne shell. Observe the following:

- `HelloEvent.java` is assumed to be in the current directory.
- *installdir* stands for the directory in which the Media Central client software is installed. By default it is `/opt/MediaCentral`.
- *host* stands for a machine running a Media Central video server.
- *ContentLibName* stands for the name of the `ContentLib`; use the Administrator to discover this name. In the Administrator, `ContentLib`s are called Clip Folders.

C shell:

```
% setenv CLASSPATH .:installdir/classes/bw.jar
% javac HelloEvent.java
% java HelloEvent vbm://host/ContentLib/vsma/ContentLibName
    ContentLib is opened
    Create a MediaContent
 !!!! Got an Event type     = 0
 !!!!              code     = 1
 !!!!              Time     = 934421989490534479
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!              cookie   = It is me
 !!!!              sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[2]]]]
 !!!! Got an Event type     = 0
 !!!!              code     = 4
 !!!!              Time     = 934421989490944432
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!              cookie   = It is me
 !!!!              sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[3]]]]
 !!!! Got an Event type     = 0
 !!!!              code     = 5
 !!!!              Time     = 934421989490985617
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!              cookie   = It is me
 !!!!              sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[4]]]]
    Delete the MediaContent
 !!!! Got an Event type     = 0
 !!!!              code     = 2
 !!!!              Time     = 934421996818420255
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!              cookie   = It is me
 !!!!              sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[5]]]]
```

Bourne shell:

```
$ CLASSPATH=.:installdir/classes/bw.jar
$ export CLASSPATH
$ javac HelloEvent.java
$ java HelloEvent vbm://host/ContentLib/vsma/ContentLibName
    ContentLib is opened
    Create a MediaContent
 !!!! Got an Event type     = 0
 !!!!               code     = 1
 !!!!               Time     = 934421989490534479
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!               cookie   = It is me
 !!!!               sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[2]]]]
 !!!! Got an Event type     = 0
 !!!!               code     = 4
 !!!!               Time     = 934421989490944432
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!               cookie   = It is me
 !!!!               sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[3]]]]
 !!!! Got an Event type     = 0
 !!!!               code     = 5
 !!!!               Time     = 934421989490985617
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!               cookie   = It is me
 !!!!               sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[4]]]]
    Delete the MediaContent
 !!!! Got an Event type     = 0
 !!!!               code     = 2
 !!!!               Time     = 934421996818420255
 !!!!           Info    = com.sun.broadcaster.vssmbeans.VssmEvent[source=null]
 !!!!               cookie   = It is me
 !!!!               sender   =
com.sun.broadcaster.vssmproxy.ContentLibProxyImpl[RemoteStub [ref:
[endpoint:[firwood:33764](local),objID:[5]]]]
```

HelloVtr: Playing a Video Tape

The HelloVtr example is similar to HelloBean but it calls methods on a `Vtr` instead of a `Player`, emphasizing the essential commonality of VideoBeans objects. When invoked with the URL of a `Vtr` factory, the program starts the VTR, runs it for 30 seconds, and stops it. The program produces no output.

CODE EXAMPLE 2-4 `HelloVtr.java` Source

```
// HelloVtr.java
import com.sun.videobeans.*;
import com.sun.videobeans.security.*;
import com.sun.videobeans.directory.*;
import com.sun.videobeans.util.*;
import com.sun.broadcaster.vtrproxy.*;
import com.sun.broadcaster.vtrbeans.*;

public class HelloVtr {

    public static void main(String[] args)
    {
    if (args.length < 1) {
        System.err.println(“HelloVtr <VBM Vtr URL>”);
        return;
    }
    HelloVtr test = new HelloVtr();
    test.runTest(args[0]);
    System.exit(0);
    }

    public void runTest(String Vtr_url)
    {
    try {

        // Connect to factory of Vtr passed on command line
        VtrFactory vtrFactory =
(VtrFactory)Naming.lookup(Vtr_url);

        // Create GranteeContext and retrieve Credential
        // A real client should pass genuine userid, hostname,
password
```

CODE EXAMPLE 2-4 `HelloVtr.java` Source *(Continued)*

```
        GranteeContext ctx
           = new GranteeContextImpl("user", "host", "pword");
        Credential cdt = vtrFactory.createCredential(ctx);

        // Get a Vtr proxy
        VtrProxy vtr = vtrFactory.createBean(cdt);

        // Play for 30 seconds, stop, and eject
        vtr.play();
        java.lang.Thread.sleep(30000);
        vtr.stop();
        java.lang.Thread.sleep(5000);
        vtr.close();
        }
    catch (Exception e)
        {
           e.printStackTrace();
        }
    }

}
```

The following examples show how to compile and run the HelloVtr example in the C shell and the Bourne shell. Observe the following:

- Load a tape into the VTR before starting the program.
- `HelloVtr.java` is assumed to be in the current directory.
- *installdir* stands for the directory in which the Media Central client software is installed. By default it is `/opt/MediaCentral`.
- *host* stands for a machine running a Media Central video server.
- *VtrName* stands for the name of a Vtr VideoBeans component; use the Administrator to discover the name of a Vtr component.

C shell:

```
% setenv CLASSPATH .:installdir/MediaCentral/classes/bw.jar
% javac HelloVtr.java
% java HelloVtr vbm://host/Vtr/VtrName
%
```

Bourne shell:

```
$ CLASSPATH=.:installdir/MediaCentral/classes/bw.jar
$ export CLASSPATH
$ javac HelloVtr.java
$ java HelloVtr vbm://host/Vtr/VtrName
$
```

CHAPTER 3

Infrastructure Classes and Services

This chapter describes Media Central classes and services that are independent of VideoBeans type in these sections:

- "Properties" on page 21
- "Common Classes" on page 22
- "Events and Event Channels" on page 24
- "Factories and Naming" on page 27
- "Access Control" on page 28
- "Resource Recovery" on page 28
- "Standard Proxy Methods" on page 29
- "Exceptions" on page 30

Chapter 4 describes the methods that are particular to each VideoBeans type.

Properties

VideoBeans components have properties that affect the operation of the hardware or software they represent. Programmers can set property values with methods provided by the beans. For example, the Player bean has a `setOutputAudioLevel()` method. If you do not set the value of a property, the value is what was last set with the Administrator. See the *Sun StorEdge Media Central User's Guide* for a description of the Administrator and the properties that can be set with it.

Common Classes

A few classes are used by several VideoBeans objects as parameters or returned by them as results. They are:

- `MediaContent`
- `ContentLib`
- `LatencyInfo`
- `Timecode`

`MediaContent` and `ContentLib`

A `MediaContent` object is a video clip plus the following descriptive attributes, which are collectively called metadata:

TABLE 3-1 `MediaContent` Metadata

Attribute	Description
`name`	Clip title, such as `"Casablanca"` (Names with spaces are not recommended because they complicate URL references.)
`streamType`	Stream type, such as `"mpeg:/2/ts"` (MPEG 2 transport stream)
`lengthInBytes`	Number of bytes in the clip
`duration`	Clip's running time, expressed as a `com.sun.bpg.util.Time`, which can be converted to and from many formats
`bitRate`	Rate, in bits per second, at which the clip is to be streamed

Creating a MediaContent allocates space for it in the video file system but does not fill the space. Use a Recorder VideoBeans object to load video data into a MediaContent. MediaContents are stored in ContentLib VideoBeans objects. Each MediaContent's name must be unique within its ContentLib. The following table lists common ContentLib methods that operate on MediaContent objects.

TABLE 3-2 ContentLib Methods for MediaContent Objects

ContentLib Method	Description
createMediaContent()	Creates an empty MediaContent with a name and a lengthInBytes.
enumMediaContents()	Lists the MediaContents in a ContentLib.
setMediaContentInfo()	Changes MediaContent attribute values.

A MediaContent can be shared by multiple readers (such as Players) and at most one writer (such as a Recorder).

Note – If a client wants to play a MediaContent that is being recorded, the client must lag the writer by 2.5 * LatencyInfo.setupDelay. ("LatencyInfo" on page 23 describes setupDelay). This delay ensures that the writer's data reaches the disk before the Media Central software fetches it for the reader.

LatencyInfo

A com.sun.broadcaster.vssmbeans.LatencyInfo object describes three kinds of video file system latency. The values are expressed in nanoseconds and are a function of a host's hardware configuration (faster hardware has less latency):

- setupDelay – the time required to prepare the video file system to transfer a clip—to set up buffers, for example. You must allow for this delay when setting up a transfer to occur at a particular time.
- tearDownDelay – the time required to clean up the video file system after transferring a clip. You must allow for this delay and setupDelay when stopping and starting a Player or Recorder bean in quick succession.
- transferDelay – for use in a future Media Central implementation

You do not need to consider setupDelay or tearDownDelay when playing a sequence of staged clips; the video file system overlaps the interclip delays with other file processing. setupDelay and TearDownDelay are only evident for the first and last clips.

Timecode

A `com.sun.videobeans.util.Timecode` object specifies a `TimeCodeType` and hours, minutes, seconds, and frames. The `TimecodeType` can have these values:

- NTSC DROP (NTSC drop frame)
- NTSC NON DROP (NTSC non-drop frame)
- PAL
- MPAL DROP (PAL drop frame)
- MPAL NON DROP (PAL non-drop [29.97 frames/sec.] frame)

Events and Event Channels

VideoBeans objects post event objects to signal changes in their state. Each event object has an identifying code containing one of the values shown in the following table. (Not all VideoBeans objects emit all of the events listed in the table; Chapter 4 details the event codes for each VideoBeans type.) You must include the VideoBeans class name to refer to an event code value. For example:

```
if (ev.code == Player.ON) ...
```

TABLE 3-3 Event Codes

Event	VideoBeans Object	Description
`ON`	`Recorder`, `Player`	The VideoBeans object's hardware has been switched on.
`OFF`	`Recorder`, `Player`	The VideoBeans object's hardware has been switched off.
`STARTED`	`Recorder`, `Player`	The VideoBeans object has started.
`STOPPED`	`Recorder`, `Player`	The VideoBeans object has stopped.
`SWITCHED`	`Player`	The Player has switched clips.
`ERROR`	`Recorder`, `Player`	The VideoBeans object has detected an error.
`CREATED`	`ContentLib`	A `MediaContent` has been created.

TABLE 3-3 Event Codes *(Continued)*

Event	VideoBeans Object	Description
METADATA_CHANGED	ContentLib	A MediaContent's metadata (attributes) has been changed.
REMOVED	ContentLib	A MediaContent has been removed.
RENAMED	ContentLib	A MediaContent has been renamed.
RESIZED	ContentLib	A MediaContent has been resized.
COMPLETED	Recorder	The clip's operational metadata has been generated.

Clients can ignore events, learn of all events posted by all VideoBeans objects, or designate particular events posted by particular VideoBeans objects as of interest. To learn of an event, a client instantiates an object called an event consumer, which implements the Media Central ConsumerCallBack interface, and registers the event consumer with an event channel. An event channel is a server object that mediates the propagation of events from VideoBeans objects to client event consumers.

To post an event, a VideoBeans object creates an event object and passes it to an event channel. For each event consumer registered with it, the event channel invokes the consumer's handleEvent() method, passing the event and a sender reference. Clients can use this reference to identify the source of an event by comparing the sender to references for proxies they hold. The handleEvent() method runs in its own thread which is set up by the Media Central software. Typically, the method updates some state (such as turning an indicator light green upon receipt of a STARTED event), and returns.

Note – Event handlers should be short and must not call VideoBeans methods. If an event handler needs to perform a lengthy operation, implement it in another thread.

Communication between VideoBeans objects and clients via event channels can be one-to-one, one-to-many, or many-to-many. Any number of clients can register event consumers with any event channel or channels; the channel calls every registered event consumer when it receives an event. Similarly, clients can connect one event channel to one VideoBeans object or to multiple VideoBeans objects.

When a client registers an event consumer with an event channel, the client provides a "cookie"; the channel returns the cookie with the events it passes to that consumer. If a client registers one consumer with several channels, the consumer can distinguish the channel that invoked it if the client registers a different cookie value with each channel.

To obtain an event channel, call factory's getEventChannel() passing:

- type – a client-defined string

- `name` – a client-defined string; the combination of `type` and `name` identifies a channel
- `nclients` – the number of clients you expect to use this channel; this is a hint that the factory uses to optimize channel performance. If the number of clients using the channel substantially exceeds `nclients`, performance may be sub-optimal; if `nclients` exceeds the number of clients, server resources allocated for the channel will be wasted. The `nclients` argument has no effect on channel function; it is only a performance-tuning mechanism.

If a channel of the `type` and `name` exists, the factory returns a reference to it; if a channel of that `type` and `name` does not exist, the factory creates the channel and returns a reference to it. A client can use any factory's `getEventChannel()` method; for example, if a client wants a channel for Player events, it can get an event channel from any Player factory or from any Recorder factory or any other factory.

An event channel persists until the server running it terminates.

The following pseudocode summarizes the essentials of event handling:

1. `Channel ch = aFactory.getEventChannel();` Client gets an event channel from a factory. Arguments give the channel type and name; clients can choose any values for these.

2. `aProxy.registerEventChannel(ch);` Client registers the channel with the proxy whose VideoBeans object posts the events the client wants to receive. An array argument can specify the subset of event IDs the client wants to receive.

3. `MyCallback myConsumer = new MyEventConsumer();` Client instantiates its consumer class that implements `ConsumerCallBack`.

4. `ConsumerImpl consumer = new ConsumerImpl(myConsumer);` Client passes its consumer object to a housekeeping object constructor.

5. `ChannelHelper helper = new ChannelHelper(ch);` Client passes the channel to another housekeeping object constructor.

6. `helper.registerConsumer(consumer, cookie);` Client registers its event consumer object with the channel that is to invoke it. `cookie` is a client identifier that is meaningful to the consumer; the channel will return the `cookie` with each event.

7. VideoBeans object associated with `aProxy` posts an event.

8. Event channel `ch`, running on the video server, remotely invokes client's `myConsumer.handleEvent()` passing a `sender` and an `event`. The `sender` is a reference to the sending VideoBeans object's proxy. The `event` contains the `cookie` the client passed to `helper.registerConsumer()`, the event code (for example, `STARTED`), the posting VideoBeans object's type, and `info`, which contains a `com.sun.bpg.vsmbeans.VssmEvent`.

For a simple example of a client that receives an event, see Chapter 2.

Factories and Naming

VideoBeans factories produce VideoBeans proxies, which clients call to record clips, play them back, and so on. A VideoBeans factory may reflect limitations of the hardware associated with its VideoBeans object. For example, a Recorder VideoBeans object controls an encoder, which is a device that can have only one user at a time. If a client asks a Recorder factory (there is one Recorder factory per encoder) to create a Recorder proxy, the factory will throw an exception if the underlying encoder is already in use.

Factories are identified by URLs. Factory URLs that have the following form:

`vbm://`*host*[`:`*port*]`/`*VideoBeansType*`/`*FactoryInstance*

The elements of a factory URL are defined as follows:

- *host* – The host whose Media Central video server manages the VideoBeans factory
- *port* – Optionally, the port on which the Media Central server listens for requests; the default port is 3058.
- *VideoBeansType*: – One of the following:
  - `ContentLib`
  - `Recorder`
  - `Player`
  - `Importer`
  - `Exporter`
  - `Migrator`
  - `VTR`
  - `Alias`
- *FactoryInstance*: – Distinguishes among factories for the same type of VideoBeans object; for example, a host with eight decoders has eight Player factories

`Alias` is a pseudo-VideoBeans type that indicates that *FactoryInstance* is a nickname. Aliases are handy for VideoBeans objects whose *FactoryInstance* names are generated by the Media Central software. For example, it is easier for most users to remember the alias `Monitor1` than `VELA.MPEG.2000-0401.0`. Use the Administrator to create aliases for factories; it is described in the *Sun StorEdge Media Central User's Guide*.

To obtain the URLs of the factories on a video server, use the static `Naming` class methods. `Naming.listTypes()` returns a list of VideoBeans types, such as `Player` and `ContentLib`. `Naming.list()` returns a list of all factories of a given type. If you have a factory URL, obtain a reference to the factory by calling `Naming.lookup()`, passing the factory URL. To obtain a VideoBeans proxy from a factory, create a credential as described in "Access Control" on page 28 and pass it in a call to the factory's `createBean()` method.

Access Control

Media Central access control centers on an object called a `Credential`. A client obtains a credential by first constructing an object called a `GranteeContext` with a userid, password, and host name. The host name specifies the host running the Media Central video server that the client wants to use; a client needs a credential for each server it uses. The client passes the `GranteeContext` to any factory's `createCredential()` method. The factory returns a credential which the client passes in subsequent `createBean()` calls; the one credential can be used with all factories on the server from which the credential was obtained. When the client invokes a factory's `createBean()` method, the factory evaluates the credential to see if the client user is allowed to create the proxy, in other words, is entitled to use the VideoBeans object. If the factory does not accept the credential passed by the client, it throws a `com.sun.videobeans.security.SecurityException` instead of returning a proxy.

Note – In the current Media Central release, a user credential passed to `createBean()` is always accepted; a more meaningful security policy may be implemented in a future release. Client developers must nevertheless observe the security conventions in Release 1 because a `GranteeContext` is also used for recovering resources, as described in "Resource Recovery" on page 28.

Resource Recovery

Media Central video servers use `GranteeContext`s for access control and resource recovery. (See "Access Control" on page 28 for a description of Media Central access control.) When a client asks a factory to create a proxy, the client passes the factory a `GranteeContext`, which the factory retains. A `GranteeContext` has an `isAlive()` method, which the server associated with the factory periodically invokes until the client closes the proxy. If the invocation fails, the server knows that the client has

exited or died, so the server releases the resources it has allocated in behalf of the client. In particular, it releases hardware resources. Releasing a decoder associated with a `Player` proxy, for example, frees the decoder for use by another client.

Note – Automatic resource recovery takes 1–5 minutes. Clients should `close()` VideoBeans proxies when they are finished to release resources immediately.

Standard Proxy Methods

All VideoBeans proxies provide the methods described in the following table. Chapter 4 describes the type-specific VideoBeans methods.

TABLE 3-4 Standard Proxy Methods

Method	Description
`getType()`	Returns the VideoBeans object's standard (unlocalized) type, for example `Player` or `Recorder`. Whether the client is running in Tokyo or Toledo, `getType()` returns the same value for a given proxy type.
`getTypeName()`	Returns a localized version of the VideoBeans object's type that is suitable for display to a user. The value varies according to the client's locale.
`getName()`	Returns a VideoBeans object's standard (unlocalized) name, for example, `VELA.MPEG.2000-0401.0`.
`getAliasName()`	Returns a VideoBeans object's alias, if it has one, or its standard (unlocalized) name.
`getFactoryURL()`	Returns the VideoBeans object factory's URL.
`registerEventChannel()`	Registers an event channel with a proxy so the proxy's VideoBeans object will send events to the channel. A subset of event types can be specified.
`unregisterEventChannel()`	Unregisters an event channel so it stops receiving events from a VideoBeans object.

TABLE 3-4 Standard Proxy Methods *(Continued)*

Method	Description
`waitTilFinished()`	Blocks the client until the outstanding asynchronous operation has completed.
`examineResult()`	Blocks the client until the outstanding asynchronous operation has completed and then catches an exception propagated from the proxy or nothing.
`close()`	Releases the proxy so the underlying VideoBeans object can be used by another client, and resources allocated to the proxy can be reclaimed by its server.

Most VideoBeans methods are synchronous: They return when they have completed the requested operation. A few VideoBeans objects also have asynchronous methods that start or schedule an operation and then return—without waiting for the operation to complete. Asynchronous method names begin with `start`; for example, `PlayerProxy.startStreamAt()` schedules playback for a future time.

A client can have at most one asynchronous operation outstanding per VideoBeans object. For example, if a client calls a `Player`'s `startStreamAt()` method, it must not call that method or another asynchronous method on the same `Player` until the playback scheduled by `startStreamAt()` has completed.

There are two ways for a client to learn that an asynchronous operation has completed:

- Calling `examineResult()` blocks the client until the operation has completed and then returns either nothing or the exception thrown by the VideoBeans object.
- Calling `waitTilFinished()` blocks the client until the operation has completed without risking catching a VideoBeans exception. When you are ready to risk catching the VideoBeans exception, call `examineResult()`.

Exceptions

Most Media Central server methods throw `java.rmi.RemoteException`. Some methods throw subclasses of `java.rmi.RemoteException` as follows:

- Several `Naming` methods throw `com.sun.videobeans.directory.NamingException`.
- The factory `createBean()` methods throw `com.sun.videobeans.security.SecurityException`.

- Proxy `registerEventChannel()` and `unregisterEventChannel()` methods throw `com.sun.videobeans.NoSuchChannelException`, which is a subclass of `com.sun.videobeans.VideoBeanException`.

CHAPTER 4

VideoBeans Catalog

This chapter describes the following Media Central VideoBeans classes:

- "`Player`" on page 33
- "`Recorder`" on page 36
- "`ContentLib`" on page 38
- "`Importer`" on page 40
- "`Exporter`" on page 41
- "`Migrator`" on page 42
- "`Vtr`" on page 42

A VideoBeans object represents a physical device, such as an encoder, or a virtual device. To use a VideoBeans object, you obtain a proxy by calling the `createBean()` method in the VideoBeans object's factory object. A proxy provides an interface to its underlying VideoBeans object's public methods and also implements the VideoBeans-type-independent methods described in "Standard Proxy Methods" on page 29.

`Player`

A `Player` streams one or more `MediaContents` (clips) from a `ContentLib` to a decoder, starting immediately or at a time you specify. There is a `Player` factory for each decoder; each factory has a different name. A decoder is not a sharable device, so at any instant there can be at most one `Player` proxy instance for a given decoder. If a `Player` factory is asked to create a proxy when one exists, it throws `com.sun.videobeans.security.VideoBeanException`. A client should close a `Player` proxy when it is no longer needed so another client can create a proxy for the `Player`, and so the server will release resources allocated to the proxy.

A `Player` maintains an internal queue of `VideoFileSegment` objects, which consist of a `MediaContent` and a `from` offset and a `to` offset. The `MediaContent` names the clip to be played, and the offsets specify the subset of the clip. A client adds `VideoFileSegments` to the back of the queue. A `Player` streams the `VideoFileSegment` at the front of the queue and removes it when playing is complete.

The `Player`'s `startStreamAt()` method schedules the `Player` to play the first clip in its queue. However, before a `Player` can play this clip, it must perform some setup operations. The `Player`'s `getLatencyInfo()` method returns the time required to perform these operations (the `setupDelay`). You can explicitly direct the `Player` to set itself up with the `startPrerollAt()` method, or you can let the `startStreamAt()` method implicitly perform the setup before it begins streaming. If you do not precede `startStreamAt()` with `startPrerollAt()`, you must allow for the `setupDelay` in the time you pass to `startStreamAt()`.

For example, suppose `setupDelay` is five seconds and you want to start playing at 11:30:00. You can schedule `startPreroll()` at 11:29:55 (or earlier) to do the setup, then schedule `startStreamAt()` 11:30:00. If you `startStreamAt()` 11:30:00 without having called `startPreroll()` first, `startStreamAt()` will incur the `setupDelay` and will actually begin streaming at 11:30:05.

A typical client uses a `Player` as follows:

1. The client obtains a `PlayerProxy` by calling `PlayerFactory.createBean()`.
2. The client optionally sets the output formats and levels (alternatively, you can rely on a user to set them with the Administrator described in *Sun StorEdge Media Central User's Guide*).
3. The client queues the video file segments to be played.
4. The client schedules the `Player` to set itself up to play the first segment (preroll).
5. The client waits for the setup to complete.
6. The client schedules the `Player` to begin playing the segments in the queue.
7. The client waits for the playing to complete.

Methods

The following table lists the most frequently used `Player` methods. Consult the Javadoc files for the signatures of these methods and other less frequently used `Player` methods.

TABLE 4-1 Principal `Player` Methods

Method	Description
`getLatencyInfo()`	Returns the `setupDelay` and `teardownDelay` values; you must allow for these delays when you schedule a `Player`.
`stage()`	Adds a `MediaClip` to the `Player` queue. You can specify a subset of the clip in the `from` and `to` parameters.
`removeLast()`	Negates the previous `stage()` call, removing the most recently staged clip from the `Player` queue.
`getList()`	Returns the contents of the `Player` queue as an enumeration of `VideoFileSegments`, which contain a `MediaContent`, a `from`, and a `to`.
`startStreamAt()`	Use this method to schedule the playing of the staged clip(s). `startStreamAt()` is an asynchronous operation; use `waitTilFinished()` or `examineResult()` to block until the operation has completed. To make the operation to start immediately, pass a `timeOfDay` of 0.
`stopAt()`	Call this method to cancel or abort a `startStreamAt()` and empty the `Player` queue. If you pass 0 as `timeOfDay`, playing will stop immediately if it has started, or it will not start if it has not started.
`setOutputAudioLevel()`	Use this method to set the output audio level in decibels.
`setOutputTimecodeFormat()`	Use this method to set the output timecode to LTC (longitudinal timecode), or VITC (vertical interval timecode).

Events

A `Player` posts `Player.ON`, `Player.OFF`, `Player.STARTED`, `Player.STOPPED`, `Player.ERROR`, and `Player.SWITCHED` events.

Recorder

A Recorder captures the output of an encoder into a MediaContent. There is one RecorderFactory per encoder. An encoder is not a sharable device, so at any instant it can be represented by at most one Recorder proxy instance. If a Recorder factory is asked to create a proxy when one exists, it throws com.sun.videobeans.security.VideoBeanException. A client should close a Recorder proxy when it is no longer needed so another client can create a proxy for the Recorder, and so server resources allocated to the proxy can be reclaimed.

A Recorder does not create the MediaContent it writes into; create the MediaContent, if necessary, with ContentLib.createMediaContent() as described in "Methods" on page 39. To specify the name of the target MediaContent, create a VideoFileSegment object. A VideoFileSegment has a URL that names the MediaContent, and a to offset that defines the portion of the MediaContent to be overwritten. (MediaContents are freely overwritable.)

The Recorder's startStreamAt() method schedules the Recorder to begin recording. However, before a Recorder can record, it must perform some setup operations. The Recorder's getLatencyInfo() method returns the time required to perform these operations (the setupDelay). You can explicitly direct the Recorder to set itself up with the startPrerollAt() method, or you can let the startStreamAt() method implicitly perform the setup before it begins recording. If you do not precede startStreamAt() with startPrerollAt(), you must allow for the setupDelay in the time you pass to startStreamAt().

For example, suppose setupDelay is five seconds and you want to start recording at 11:30:00. You can schedule startPreroll() at 11:29:55 (or earlier) to do the setup, then schedule startStreamAt() 11:30:00. If you startStreamAt() 11:30:00 without having called startPreroll() first, startStreamAt() will incur the setupDelay and will actually begin recording at 11:30:05.

A typical client uses a Recorder as follows:

1. The client obtains a RecorderProxy by calling RecorderFactory.createBean().
2. The client creates a MediaContent (see "ContentLib" on page 38) to store the received data if it does not already exist.
3. The client optionally sets the Recorder input format parameters (you may instead rely on the user to have set them with the Administrator).
4. The client creates a VideoFileSegment which names the target MediaContent and offset.
5. The client optionally calls startPrerollAt() to schedule the setup.

6. The client calls `waitTilFinished()` or `examineResult()` to block until the setup is complete.
7. The client calls `startStreamAt()` to schedule the recording.
8. The client calls `waitTilFinished()` or `examineResult()` to block until the recording is complete.

Methods

The following table lists the most frequently used `Recorder` methods. Consult the Javadoc files for the signatures of these methods and other less frequently used `Recorder` methods.

TABLE 4-2 Principal `Recorder` Methods

Method	Description
`getLatencyInfo()`	Returns the `setupDelay` and `teardownDelay` values; you must allow for these delays when you schedule a `Recorder`.
`setOutputFileSegment()`	Use this method to specify the `MediaContent`, and offset into it, where the `VideoFileSegment` is to be stored. You can overwrite any portion of a `MediaClip`; the method assumes that you know what you are doing.
`getOutputFileSegment()`	Returns the `VideoFileSegment` previously set with `setOutputFileSegment()`.
`startStreamAt()`	Use this method to schedule the recording of the segment. To make the operation to start immediately, pass a `timeOfDay` of 0.
`stopAt()`	Call this method to cancel or abort a `startStreamAt()`. If you pass 0 as `timeOfDay`, playing will stop immediately if it has started, or will not start if it has not started.
`setInputVideoFormat()`	Use this method to set the input video format to NTSC, PAL, SDI, SECAM, YC, or YUV.
`setInputAudioLevel()`	Use this method to set the output audio level in decibels.
`setInputAudioFormat()`	Use this method to set the output audio format to one of the following: • AES EBU • RCA BAL (RCA balanced) • RCA UNBAL (RCA unbalanced)
`setInputTimecodeFormat()`	Use this method to set the output timecode to one of the following: • AUTO TC (auto timecode) • LTC (longitudinal timecode) • TT1 (tape timer 1) • TT2 (tape timer 2) • VITC (vertical interval timecode)

TABLE 4-2 Principal `Recorder` Methods *(Continued)*

Method	Description
`setVideoFrameRate()`	Use this method to set the video frame rate to one of the following: • R23_976 (24 x 1000 / 1001 frames / sec.) • R24 (24 frames / sec.) • R25 (25 frames / sec.) • R29_97 (30 x 1000 / 1001 frames / sec.) • R30 (30 frames / sec.) • R50 (50 frames / sec.) • R59_94 (60 x 1000 / 1001 frames / sec.) • R60 (60 frames / sec.).
`setAudioCompressionRate()`	Use this method to set the audio compression rate.
`setAudioSamplingRate()`	Use this method to set the audio sampling rate frequency to 32, 44.1, or 48 KHz.
`setVideoCompressionRate()`	Use this method to set the video compression rate.
`setMuxCompressionRate()`	Use this method to set the multiplexor compression rate.
`getBytesWritten()`	Returns the size of the data captured so far.

Events

A `Recorder` posts `Recorder.ON`, `Recorder.OFF`, `Recorder.STARTED`, `Recorder.ERROR`, `Recorder.COMPLETED`, and `Recorder.STOPPED` events.

ContentLib

A `ContentLib` represents a server's video file system. There is one instance per host, but it can be shared among clients, each of which must obtain its own `ContentLib` proxy from the server's `ContentLib` factory. The default `ContentLib` factory name is:

`vbm://`*host*[:*port*]`/ContentLib/vsma/`*host*

However, the trailing *host* can be set to something different when the Media Central software is installed on the server.

A `ContentLib` holds clips (`MediaContent` objects). The `MediaContent`s in a `ContentLib` are distinguished by their names. A `ContentLib`'s namespace is flat: compared to a UNIX file system, a `ContentLib` has one "directory," which is always "the current directory." Unlike UNIX files, `MediaContent`s are not

automatically grown to accommodate the data recorded into them. For performance reasons, the entire `MediaContent` must be preallocated. Make a `MediaContent` long enough to hold the data you plan to record into it; when you have finished recording, you can release unused space by resizing the `MediaContent`.

Methods

The following table lists the most frequently used `ContentLib` methods. Consult the Javadoc files for the signatures of these methods and other less frequently used `ContentLib` methods.

TABLE 4-3 Principal `ContentLib` Methods

Method	Description
`getMaxRate()`	Use this method to obtain the maximum aggregate bit rate that the host can guarantee.
`enumMediaContents()`	Returns an array of `MediaContents`.
`getMediaContent()`	Returns a reference to the `MediaContent` you name in the argument.
`getInfo()`	Returns a `ContentLibID` object whose variables include the size of the `ContentLib` and the space that has been used.
`createMediaContent()`	Creates an empty `MediaContent` with a name and length you specify.
`deleteMediaContent()`	Deletes a `MediaContent`.
`setMediaContentInfo()`	Sets `MediaContent` attribute values (metadata), such as bit rate and duration.
`renameMediaContent()`	Renames a `MediaContent`.
`resizeMediaContent()`	Resizes a `MediaContent`. Use this method after recording to shrink an over-allocated `MediaContent` to the number of bytes actually used. Use `Recorder.getBytesWritten()` to learn how much of a `ContentLib` has been written.

Events

A `ContentLib` posts `ContentLib.CREATED`, `ContentLib.METADATA_CHANGED`, `ContentLib.RENAMED`, `ContentLib.REMOVED`, and `ContentLib.RESIZED` events. These events are associated with `MediaContents` in the `ContentLib`, not the `ContentLib` itself.

Importer

An `Importer` creates a `MediaContent` and writes encoded data into it from either:

- A local or NFS™ file. Specify the file as a URL; for example `file:/x/y`.
- A TCP or UDP port on the same host. (TCP is slower but reliable; UDP is faster but does not guarantee data integrity.) An `Exporter` (see "`Exporter`" on page 41) on another host supplies the data the `Importer` reads, thus copying a `MediaContent` between `ContentLib`s. Specify the same port for `Importer` and `Exporter` as URLs:

  `udp://`*hostname*`:`*port* or `tcp://`*hostname*`:`*port*.

There is one `Importer` factory. It can create multiple `Importer` instances but the instances cannot not use the same ports. The `Importer` factory is named:

`vbm://`*host*`[:`*port*`]/Importer/Vssmx`

Methods

The following table lists the most frequently used `Importer` methods. Consult the Javadoc files for the signatures of these methods and other less frequently used `Importer` methods.

TABLE 4-4 Principal `Importer` Methods

Method	Description
`setImportRate()`	Set this to the maximum bandwidth, in bits per second, you want the operation to consume; it will consume less if the supplier runs slower.
`startImporting()`	Creates a `MediaContent` of the name and length specified in arguments, and begins importing from the specified file or port. Start the Importer before the Exporter if you are copying between servers.
`abort()`	Aborts an importing operation. Aborting does not erase data that has already been written, and it may leave the data in a corrupted state.

Events

An `Importer` posts `Importer.STARTED`, and `Importer.STOPPED` events.

Exporter

An Exporter writes a MediaContent's data to either:

- A local or NFS file; the Exporter will create the file if it does not exist. Specify the file as a URL; for example, file:/x/y.
- A TCP or UDP port on the another host. (TCP is slower and reliable; UDP is faster but does not guarantee data integrity.) An Importer reads the data the Exporter sends, thus copying a MediaContent between ContentLibs. Specify the port as a URL; for example, udp://*hostname*:*port* or tcp://*hostname*:*port*. Specify the same port for Importer and Exporter.

There is one Exporter factory. It will create multiple Exporter instances but the instances must not use the same ports. The Exporter factory is named:

vbm://*host*[:*port*]/Exporter/Vssmx

Methods

The following table lists the most frequently used Exporter methods. Consult the Javadoc files for the signatures of these methods and other less frequently used Exporter methods.

TABLE 4-5 Principal Exporter Methods

Method	Description
setExportRate()	Sets the rate, in bits per second, according to the amount of available bandwidth you want the transfer to consume. The Exporter will run at the slower of this rate and the Importer's rate.
startExporting()	Begins writing a MediaContent of the name and length specified in arguments, to a specified file or port. Start the Importer before the Exporter if you are copying between servers.
abort()	Aborts an exporting operation. Aborting does not erase data that has already been written, and may leave it in a corrupted state.

Events

An Exporter posts Exporter.STARTED and Exporter.STOPPED events.

Migrator

A Migrator makes copy of a `MediaContent` within a `ContentLib`.

There is one `Migrator` factory. It will create multiple `Migrator` instances. The `Migrator` factory is named:

`vbm://`*host*`[:`*port*`]/Migrator/Vssmx`

Methods

The following table lists the most frequently used `Migrator` methods. Consult the Javadoc files for the signatures of these methods and other less frequently used `Migrator` methods.

TABLE 4-6 Principal `Migrator` Methods

Method	Description
`setMigrationRate()`	Sets the amount of bandwidth, in bps, you are willing for the operation to consume.
`startMigrating()`	Begins copying a `MediaContent`. This is an asynchronous operation.
`abort()`	Aborts a migrating operation. Aborting does not erase data that has already been written, and may leave it in a corrupted state.

Events

A `Migrator` posts `Migrator.STARTED` and `Migrator.STOPPED` events.

Vtr

A `Vtr` controls a video tape recorder (VTR) that is connected to a Media Central server by a V-LAN controller.

There is a `Vtr` factory for each video tape recorder. An VTR is not a sharable device, so at any instant it can be represented by at most one `Vtr` proxy instance. If a `Vtr` factory is asked to create a proxy when one exists, it throws

com.sun.videobeans.security.VideoBeanException. A client should close a Vtr proxy when it is no longer needed so another client can create a proxy for the Vtr, and so server resources allocated to the proxy can be reclaimed.

Methods

The following table lists the most frequently used Vtr methods. Consult the Javadoc files for the signatures of these methods and other less frequently used Vtr methods.

TABLE 4-7 Principal Vtr Methods

Method	Description
play()	Plays the tape.
stop()	Stops the tape.
fastForward()	Fast forwards the tape.
rewind()	Rewinds the tape.
goToTimeCode	Moves the tape to the timecode passed as an argument.
getPositionTimeCode()	Returns the timecode representing the tape's current position
still()	Pauses the tape.
record()	Begins recording the tape.

Events

A Vtr does not post events.

CHAPTER 5

Javadoc Guide

The low-level documentation of Media Central client packages is provided in Javadoc files. This chapter introduces the following packages:

- "com.sun.videobeans.directory" on page 45
- "com.sun.videobeans.util" on page 46
- "com.sun.videobeans" on page 46
- "com.sun.videobeans.beans" on page 46
- "com.sun.videobeans.event" on page 47
- "com.sun.videobeans.security" on page 47
- "com.sun.broadcaster.vssmbeans" on page 47
- "com.sun.broadcaster.vssmproxy" on page 48
- "com.sun.broadcaster.vtrproxy" on page 49

The starting page for the Javadoc files is *installdir*/doc/api/index.html. The default installation directory is /opt/MediaCentral.

com.sun.videobeans.directory

The directory package contains the Naming class, which all clients use to obtain references to factories. It also defines NamingException, which some Naming methods throw.

Ignore the other classes in this package; they are used internally.

com.sun.videobeans.util

The `util` package contains these widely used classes:

- `Timecode` is a time representation expressed in hours, minutes, seconds, and frames.
- `Time` is an encapsulated representation of time used by VideoBeans. The class provides methods for converting a `Time` to and from several other formats, for example, `java.util.Date`, nanoseconds, PCR (27 MHz ticks) and `Timecode`. Client developers can minimize conversions when calling VideoBean methods by using `Time` objects to represent time in their code.

  When you convert a `Time` to a `Timecode`, the fraction of a frame, if any, is truncated, losing data that was present in the `Time`. Do not assume, therefore, that you can convert a value in nanoseconds to a `Time`, convert the `Time` to a `Timecode`, and then recover the original nanosecond value by converting the `Timecode` back to a `Time` and the `Time` to nanoseconds. That will be true only if the nanosecond value converts to an integral number of frames. If you need the original nanosecond value, keep it; do not assume you can re-create it from a `Timecode`.

Ignore other classes in the `util` package; they are used internally.

com.sun.videobeans

This package defines:

- `VideoBeanProxy` interface, which all proxies implement
- `VideoBeanFactory` interface, which all factories implement
- `VideoBeanException` class, and its subclass `NoSuchChannelException`, which is thrown by proxy register and unregister event channel methods.

Ignore other classes and interfaces in this package; they are used internally.

com.sun.videobeans.beans

This package has no definitions of interest to client developers.

com.sun.videobeans.event

This package defines:

- `ChannelHelper` class, which you must instantiate for each event channel your client uses
- `ConsumerCallBack` interface, which your event consumer class must implement; a channel calls this interface's `handleEvent()` method to pass an event to a consumer.
- `ConsumerImpl` class, which you must instantiate for each event consumer object you instantiate.
- `Event` class, which defines the fields contained in events.

Ignore the other interfaces and classes in this package which are for internal use.

com.sun.videobeans.security

This package defines:

- `Credential` interface, which, although opaque, is what factory `createCredential()` methods create and what factory `createBean()` methods require
- `GranteeContextImpl`, which a client instantiates as a prerequisite to obtaining a `Credential`
- `SecurityException`, which factory `createBean()` methods throw when they reject a `Credential`

com.sun.broadcaster.vssmbeans

This package defines many classes that enumerate permissible data values as `static final` variables. For example, the `AudioSamplingRate` class defines variables named `R32K`, `R44_1K`, and `R48K`. When you need to know the permitted values for a parameter or return value, look for a class in this package. Note in particular that the event codes produced by VideoBeans classes are defined here, for example, `Player.OFF`, and `ContentLib.CREATED`.

Developers can use the following classes:

- com.sun.broadcaster.vssmbeans.AbstractVideoFormat
- com.sun.broadcaster.vssmbeans.AccessMode
- com.sun.broadcaster.vssmbeans.AudioFormat
- com.sun.broadcaster.vssmbeans.AudioSamplingRate
- com.sun.broadcaster.vssmbeans.AudioProfile
- com.sun.broadcaster.vssmbeans.ContentLibID
- com.sun.broadcaster.vssmbeans.DeviceBusyException
- com.sun.broadcaster.vssmbeans.FileAccessException
- com.sun.broadcaster.vssmbeans.InvalidURLException
- com.sun.broadcaster.vssmbeans.LatencyInfo
- com.sun.broadcaster.vssmbeans.MediaContent
- com.sun.broadcaster.vssmbeans.MetadataLevel
- com.sun.broadcaster.vssmbeans.SplicerMode
- com.sun.broadcaster.vssmbeans.SplicerModeOption
- com.sun.broadcaster.vssmbeans.StreamType
- com.sun.broadcaster.vssmbeans.TICKS_PER_SECOND
- com.sun.broadcaster.vssmbeans.TimecodeFormat
- com.sun.broadcaster.vssmbeans.VideoFileSegment
- com.sun.broadcaster.vssmbeans.VideoFormat
- com.sun.broadcaster.vssmbeans.VideoFrameRate
- com.sun.broadcaster.vssmbeans.VideoProfile
- com.sun.broadcaster.vssmbeans.VssmEvent
- com.sun.broadcaster.vssmbeans.VSSMException

Ignore all other classes defined in this package; they are for internal use.

com.sun.broadcaster.vssmproxy

This package defines the proxy and factory interfaces for most of the VideoBeans classes. In the factory interfaces, only call `createBean()` methods; ignore the other factory methods. Ignore the classes defined in this package. The following methods are not supported:

- `ContentLib.getMaxRate()`
- `Player.jog()`
- `Player.streamNextAt()`
- `Recorder.pauseAt()`, `Recorder.pauseOn()`, `Recorder.startPrerollAt()`, `Recorder.startStreamOn()`, `Recorder.startStreamAt()`

com.sun.broadcaster.vtrproxy

This package defines the proxy and factory methods for `Vtr` VideoBeans objects. Use only the methods listed in TABLE 4-7; other methods in this package are not supported.

Index

A

access control, 28
asset server, defined, 2

C

client
  communicating with video server, 2
  defined, 1
`com.sun.broadcaster.vssmbeans` package, 47
`com.sun.broadcaster.vssmproxy` package, 48
`com.sun.broadcaster.vtrproxy` package, 49
`com.sun.videobeans.beans` package, 46
`com.sun.videobeans.directory` package, 45
`com.sun.videobeans.event` package, 47
`com.sun.videobeans.security` package, 47
`com.sun.videobeans.util` package, 46
ContentLib VideoBeans object, 23
  events, 39
  general description, 38
  `MediaContent` naming, 38
  principal methods, 39
credential, 28

E

event
  codes, 24
  consumer, 4, 25
  consumer, `handleEvent()` method, 25
  cookie, 25
  defined, 4
event channel, 25
  communication patterns, 25
  defined, 4
  obtaining, 25
  persistence, 26
  usage summary, 26
examples
  HelloBean, 8
  HelloEvent, 11
  HelloFactories, 6
  HelloVtr, 17
Exporter VideoBeans object
  events, 41
  general description, 41
  principal methods, 41

F

factory
  alias, 4, 51
  defined, 3
  names, 4
  obtaining a server's URLs, 28
  URLs, 27

H

HelloBean example, 8
HelloEvent example, 11
HelloFactories example, 6
HelloVtr example, 17

I

Importer VideoBeans object
  events, 40
  general description, 40
  principal methods, 40

L

`LatencyInfo` object, 23

M

Media Central web site, xv
`MediaContent` object, 22
metadata, defined, 22
Migrator VideoBeans object
  events, 42
  general description, 42
  principal methods, 42

P

Player VideoBeans object
  events, 35
  general description, 33
  principal methods, 35
  queue operation, 34
  setup and scheduling, 34
  usage summary, 34
proxy
  asynchronous methods, 30
  closing to recover resources, 29
  defined, 3
  exceptions, 30
  standard methods, 29

R

Recorder VideoBeans object
  events, 38
  general description, 36
  principal methods, 37
  setup and scheduling, 36
  usage summary, 36

S

`setupDelay`, 23

T

`tearDownDelay`, 23
`Timecode` object, 24

V

video server, defined, 2
VideoBeans object
  defined, 3
  invoking through proxy, 3
  properties, 21
Vtr VideoBeans object
  events, 43
  general description, 42
  principal methods, 43