Using Caché Direct

Version 2018.1
2020-11-13

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Caché Direct
Caché Version 2018.1 2020-11-13

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book tells programmers how to use Caché Direct to create clients that communicate with a Caché server by means of its ActiveX control or its C++ API.

This book contains the following chapters:

- Introduction to Caché Direct introduces Caché Direct, its major concepts, and the tools it provides to you.
- Basics of Using the VisM Control describes how to use the VisM control in general.
- Additional Features discusses how you can specify custom processing to perform at various times of the overall client/server interaction.
- Best Practices describes best programming practices for Caché Direct.
- VisM.ocx Control Details provides reference details for the Caché Direct ActiveX control (VisM.ocx).
- Using Caché Direct in Non-ActiveX Applications describes how to use Caché Direct properties and methods from an application such as C++ that does not use ActiveX (where you do not have VisM.ocx).
- Logging describes how to enable client or server logging, particularly to diagnose problems.

This book also contains the following appendixes:

- Installation and Upgrade describes how to upgrade the server without reinstalling Caché and how to install the client-side software on a machine where Caché is not installed.
- Notes for Users of the Previous Versions describes architectural changes of interest to users of previous versions, who may want to adapt their applications to take advantage of the new features.
- Example: Visual Basic Printer Support describes a sample that demonstrates callbacks by using the Windows default printer (the VB Printer object) from ObjectScript.

And a complete table of contents.

1

Introduction to Caché Direct

Caché Direct is a client/server connection mechanism that provides, over a TCP connection, direct control over server operation from a Windows COM/OLE or C++ client program. The connection is performed via a very fast, low overhead mechanism. Caché Direct includes built-in, transparent support for Caché security, Citrix/Windows Terminal services, IPv6, and so on.

Note: As of Caché 2015.1, the client is available in separate versions for 32-bit and 64-bit systems.

As suggested by its name, Caché Direct provides the most direct possible access from a client to a Caché server. That is, ObjectScript commands are sent from the client to the server, executed there, and the results returned to the client. Caché Direct sets up one or more TCP channels between the two parts of the application and manages the messages between them. The full power of Caché objects and embedded SQL are also available through Caché Direct; the client has full access to all the facilities of the server, restricted only by the Caché security settings.

Multiple server jobs/channels may be created and managed simultaneously from a single client process, providing the possibility of client multithreading and connections to multiple servers from a single client at the same time.

Caché Direct also provides facilities for calling back from the server to the client, logging flow of control and communications traffic on both the client and server, and various hooks for message transformations (such as compression), error handling, and shutdown tasks.

Note: Caché Direct is not intended for use in building Web-based applications. Instead it is meant for direct client/server applications.

1.1 Concepts

Caché Direct provides an ActiveX control (VisM.OCX, or VisM64.OCX) that is used most frequently in Visual Basic projects, as well as a C++ interface for C++ clients. The VisM control has properties and methods that you use to specify the connection to Caché and the commands to send.

Internally, the VisM control does not communicate directly with Caché. Instead, it starts and is attached to an object called *CDConnect*, which in turn connects to Caché, starting a slave server process on the server and managing the communication channel. The CDConnect can be attached to multiple VisM instances, can exist even if detached from all VisM instances, and can be redirected to a different server process. This flexibility means that Caché Direct can support various architectures such as the following:

- A setup in which each client VisM has its own server.
- A setup where all client VisMs share a single server.

- A setup in which a pool of client VisMs share a pool of servers, possibly moving among them, using whichever is currently free.
- A setup in which selected client VisMs share selected servers based on the specific databases those servers can access.

The Caché Direct client is supported on all Windows platforms that support the Caché client.

The Caché Direct server is supported on all platforms that support the Caché server.

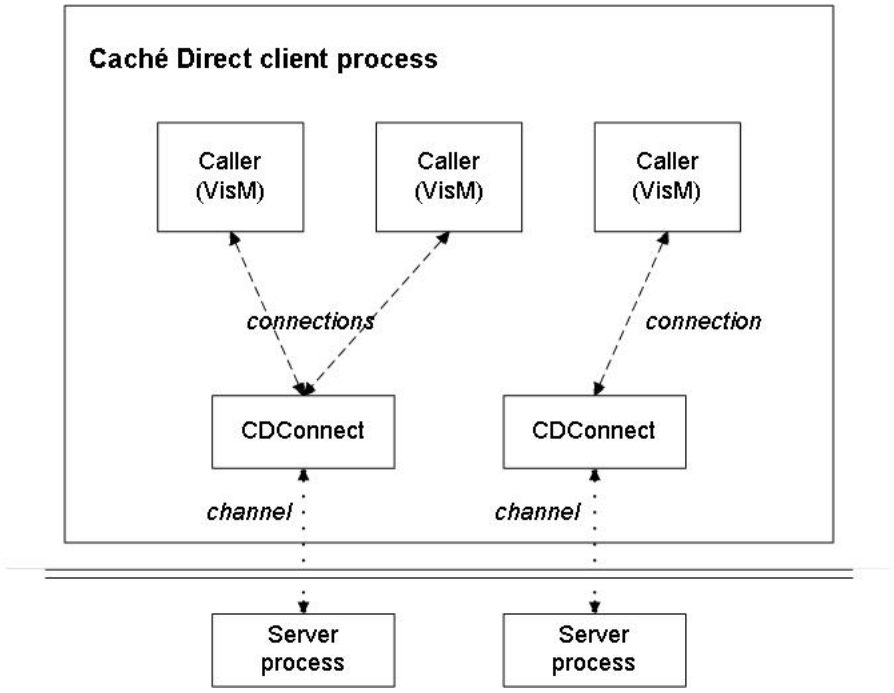
The servers can be on the same or different machines or databases, in any convenient distribution.

1.1.1 Terminology

The Caché Direct communications model uses the following terms:

- A *caller* is the client code from which you call Caché. These are usually VisM instances and are referred to as VisMs in this document. They may also be C++ code.
- A CDConnect is the intermediary object between the VisM client and the server. The CDConnects are created as needed and run on the client machine.
- The attachment between a VisM and a CDConnect is referred to as a *connection*.
- The TCP attachment between a CDConnect and a Caché server process is referred to as a *channel*.

The following figure shows a possible state of a Caché Direct application. The items in the upper area (above the double line) are all running on the client side, within a single process. The items below the double line run on the Caché server machine or machines.



1.1.2 Communication Model

Formally, the Caché Direct communications model obeys the following rules:

- Each VisM can have at most one connection at any given time. That is, any VisM is either unattached or is attached to a single CDConnect.

- Each CDConnect can have at most one channel at any given time. That is, any CDConnect is either unattached or is attached to a single Caché server process.
- Each CDConnect can have any number of connections. That is, any CDConnect can be attached to zero, one, or more VisMs. This means that multiple VisMs can share the same server context. These VisMs would have access to the same server local variables, except for the variables that are reinitialized by each message; see the section Mirrored VisM Properties, in the next chapter.
- A CDConnect does not have to be attached to a Cache server process. It lives independently of the VisMs and the server processes. A CDConnect exists until it is destroyed. It is destroyed automatically when it is no longer accessible (when it has no connections and if it has no tag), or you can explicitly destroy it via the **DeleteConnection()** method.

  Note: Before ISCDLink version 220, a CDConnect was not automatically destroyed when it became inaccessible.

- A CDConnect can be redirected to a different Caché server. This means, of course, that any VisMs that are attached to this CDConnect will be effectively attached to the newly chosen server as well.
- Each Caché server job is a single thread. Therefore, communication on each channel must be synchronous; that is, each client message must be processed and responded to before the next message can be read. This means that if multiple VisMs share a single CDConnect, they are constrained not to try to communicate simultaneously. If they do, you will get a "nonsynchronous communication" error.
- A Caché server job must have exactly one channel. When you close a channel from a CDConnect, you also end the corresponding Caché server job.

An analogy may be helpful. A CDConnect behaves as a speaker phone with a single telephone line. The listeners in the room are the VisMs; they all hear the same communication. There is one person on the other end of the call; that person is the Caché server. If the call is transferred to another server, all the VisMs are now connected to that new person.

1.2 Available Tools and Approaches

On the client side, Caché Direct is a set of layered C++ classes, which in turn are wrapped in the VisM control, which exposes all the necessary properties and methods. When you install the Caché client software, it installs and registers this control so that you can use it in an ActiveX host, such as Visual Basic. As a result, you can use Caché Direct at several different levels:

- You can include the Caché Direct ActiveX control in a Visual Basic project. Then your application (possibly with user interaction) can set values of properties, execute methods that send code to the Caché server, and display the results. This document assumes that this technique is the most common usage.
- You can use the Caché Direct C++ classes in your C++ project. Then you can create instances of these classes as needed, and set properties and execute methods, to communicate with the Caché server.

On the server side, Caché Direct is a transparent part of the Caché server installation. If you need to upgrade the Caché Direct server without installing a later version of Caché itself, there is a simple way to load the latest Caché Direct server code into your Caché installation. See Upgrading Your Caché Direct Server, near the end of this manual.

2

Basics of Using the VisM Control

This chapter describes the basics of using the VisM control:

- Accessing the control and adding it to your project.
- Connecting to and disconnecting from Caché.
- Establishing the Caché namespace in which your server-side code will run.
- Executing code on the server, from the VisM client.
- Mirroring VisM properties between the client and server, which gives your code an easy way to pass data between server and client.
- Referring to other client user interface properties and methods from the server, via callbacks.
- Calling Windows functions or Caché Direct utilities.

This chapter also discusses constraints on Caché Direct messages that may affect your code on either side. It concludes with a couple of simple examples.

Note: A leading underline indicates a callback reference to the Visual Basic user interface. Note that Caché does not allow this usage in any other situation.

2.1 Accessing the VisM Control

When you install the Caché client software, it installs and registers the VisM control so that it is available to any ActiveX host, such as Visual Basic.

Note: If the Caché client is not installed on a given machine, you can manually copy the Caché Direct client files into place and register them. See the section Installing VisM on a New Machine, near the end of this manual.

In the case of Visual Basic, to add this control to your project, click **Project —>Components**, scroll to VisM, and select the check box. Then you can add instances of the control to your forms as you do any other control.

2.2 Connecting and Disconnecting

To connect or disconnect the Caché Direct client from the server, you can use the Server property, the ConnTag property, the **SetServer()** method, the **Connect()** method, and the **DeleteConnection()** method. Each of these tools has specific uses, but there is overlap. The **SetServer()** and **Connect()** methods are similar, but have small differences for historical reasons and to preserve backward compatibility. The Server property and the **SetServer()** method are the same except for the second argument to **SetServer()**.

Also see the appendix Notes for Users of Previous Version.

2.2.1 Connection Strings and Connection Tags

Before describing the specific syntax for connecting to Caché, it is useful to know about the connection strings and connection tags that are used in that syntax.

2.2.1.1 Basic Connection String

A connection string is a pieced string of the following form:

```
"CN_IPTCP:server[port]"
```

The first piece of this argument, `CN_IPTCP`, is the connection method, which is always TCP. The second piece is the server name or IP address and port where the Caché superserver is running. For example, you could use the following syntax to set the connection of a VisM named VisM1:

```
VisM1.Server="CN_IPTCP:127.0.0.1[57772]"
```

There are extensions to this basic string (as described in VisM Extended Connection String Syntax in the chapter on VisM.ocx Control Details). These extensions are not commonly used.

2.2.1.2 Connecting with an Indirect Reference

Alternatively, you can use an indirect reference to a locally defined database alias. In this case, you use a string of the form `"@servername"` where *servername* is the server name as specified within the Caché Server Manager. (For details on using the Caché Server Manager, see Define a Remote Server Connection in the *Caché System Administration Guide*). Note that these aliases are local to the client machine. You would hard code this approach only if the client machines followed a naming convention for their Caché server aliases.

For example, you could use the following syntax to set the connection of a VisM object named `VisM1` to server name `unix1`:

```
VisM1.Server="@unix1"
```

Note: An indirect reference is the preferred way to open a connection that uses Kerberos authentication. The Caché Server Manager allows you to set Kerberos as the authentication method, specify the connection security level, and define the Service Principal Name.

2.2.1.3 User Prompt

In any place where you can use a connection string, you can instead use a quoted question mark (`"?"`). In this case, the user is prompted to choose one of the server aliases defined on the client machine.

For example, the following syntax would cause the user to be prompted for the connection of a VisM named `VisM1`:

```
VisM1.Server="?"
```

2.2.1.4 Connection Tags

In some cases, when you create a new connection to Caché, you can provide an optional connection tag. This is meant to serve as a name for the CDConnect that you are creating and its channel. It is your responsibility to ensure that all tags are unique within a given client process.

Then, when you connect a VisM, you can specify the connection tag of an existing CDConnect, rather than using a connection string. This is the main way to explicitly share a channel. For example, you could use the following syntax to set the connection of a VisM named `VisM1`:

```
VisM1.Server="TagA"
```

Alternatively, suppose that we wanted to connect `VisM1` to the same CDConnect to which `VisM2` is connected and that CDConnect has a tag assigned to it. If we did not want to use a connection tag itself, we could use the ConnTag property as follows:

```
VisM1.Server=VisM2.ConnTag
```

2.2.2 Connecting to Caché

Connecting is the action of attaching a VisM to a CDConnect (creating the CDConnect if necessary). If the VisM is currently attached to a CDConnect object, it is disconnected from that CDConnect first.

2.2.2.1 If Not Yet Connected

If the VisM is not yet connected to Caché, you can connect it to Caché by using any of the following techniques, which are listed here in order by how commonly they are used:

- Set the Server property equal to a connection string (or equivalent). For example:

  ```
  VisM1.Server="CN_IPTCP:127.0.0.1[57772]"
  ```

- Call the **SetServer()** method with a connection string (or equivalent) as the first argument. This is equivalent to the previous technique. For example:

  ```
  VisM1.SetServer("CN_IPTCP:127.0.0.1[57772]")
  ```

- Call the **Connect()** method with a connection string (or equivalent) as the first argument. For example:

  ```
  VisM1.Connect("CN_IPTCP:127.0.0.1[57772]")
  ```

In each case, Caché Direct creates a CDConnect and creates a connection from the VisM to the CDConnect object. It also starts a server process and creates a channel from the CDConnect to the server process.

If you use the **SetServer()** or the **Connect()** method, you can provide a connection tag as the second argument. For example:

```
VisM1.SetServer("CN_IPTCP:127.0.0.1[57772]","tagA")
```

2.2.2.2 If Already Connected

If the VisM is already connected to Caché, that means that a CDConnect has been created, possibly with an associated server process. You can connect the VisM to a different CDConnect, disconnecting from the original CDConnect and creating a server for the new CDConnect. To connect to a different CDConnect, use any of the following techniques.

- To create a *new* CDConnect and connect to it, call the **Connect()** method with a connection string as the first argument.
- To connect to the *most recently opened* CDConnect, call the **Connect()** method with an empty string as the first argument.
- To connect to an *existing* CDConnect, do any of the following:

- Set the Server property equal to a connection tag (permitted as of Caché 2007.1).
- Call the **SetServer()** method with a connection tag as the first argument (permitted as of Caché 2007.1).
- Call the **Connect()** method with a connection tag as the first argument.

For example:

```
VisM1.Server="tagA"
```

These actions do not destroy the original CDConnect; nor do they affect its server channel, if it has one. The CDConnect exists until it is no longer accessible (when it will be destroyed automatically) or until it is explicitly destroyed, as described in Destroying a CDConnect.

2.2.3 Changing the Channel of a CDConnect

To change the channel of an existing CDConnect, use either of the following equivalent techniques:

- Set the Server property equal to a connection string.
- Call the **SetServer()** method with a connection string as the first argument.

In either case, the CDConnect stops the server process to which it is currently connected, disconnects from it, and then starts and connects to a new server process.

2.2.4 Disconnecting from Caché

To disconnect from the Caché server, use either of the following equivalent techniques:

- Set the Server property equal to an empty string.
- Call the **SetServer()** method with an empty string as the argument.

These actions do not destroy the CDConnect; nor do they affect its server channel, if it has one. The CDConnect exists until it is no longer accessible (when it will be destroyed automatically) or until it is explicitly destroyed, as described in the next section.

2.2.5 Destroying a CDConnect

To destroy the CDConnect and its channel, call the **DeleteConnection()** method. This method also stops the server process, of course. For example:

```
VisM1.DeleteConnection()
```

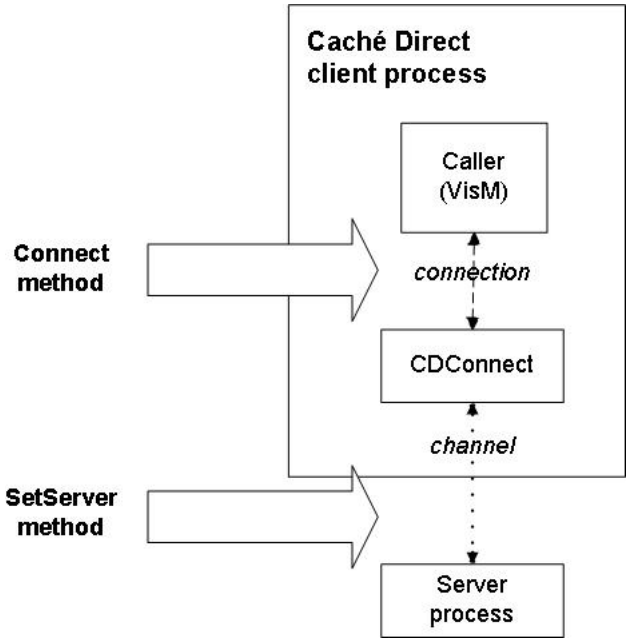
2.2.6 Summary of Techniques

The following table summarizes how to connect and disconnect from Caché. Notice that setting the Server property has the same effect as calling the **SetServer()** method, in all cases.

Action	How To Do This
Connecting to a new CDConnect, if not yet connected	Use any of the following techniques: • Set the Server property equal to a connection string. • Call the **SetServer()** method with a connection string as the first argument. • Call the **Connect()** method with a connection string as the first argument.
Connecting to a new CDConnect, if already connected*	Call the **Connect()** method with a connection string as the first argument.
Connecting to an existing CDConnect (after disconnecting from current CDConnect, if any)*	Use any of the following techniques: • Call the **Connect()** method with a connection tag as the first argument. • Set the Server property equal to a connection tag (permitted as of Caché 2007.1). • Call the **SetServer()** method with a connection tag as the first argument (permitted as of Caché 2007.1).
Connecting to the last opened CDConnect (after disconnecting from current CDConnect, if any)*	Call the **Connect()** method with an empty string as the first argument.
Changing the channel of the existing CDConnect	Use either of the following techniques: • Set the Server property equal to a connection string. • Call the **SetServer()** method with a connection string as the first argument.
Disconnecting*	Use either of the following techniques: • Set the Server property equal to an empty string. • Call the **SetServer()** method with an empty string as the argument.
Destroying the CDConnect and its channel	Call the **DeleteConnection()** method.

Key: *These actions do not destroy the CDConnect; nor do they affect its server channel, if it has one. The CDConnect exists until it is no longer accessible (when it will be destroyed automatically) or until it is explicitly destroyed.

The following figure summarizes the differences between the **Connect()** and the **SetServer()** methods.



2.3 Establishing the Namespace

When you connect to a server, the namespace is initially set to the current namespace of the superserver, which is generally not a suitable place to execute your application code. There are three equivalent ways to switch the namespace:

- Your code can explicitly change the namespace (via the **$ZNAME** command).
- You can use the VisM NameSpace property, which is sent to the server in every execution message. The server examines this property before executing any code, and it uses the following logic: If NameSpace is empty or equals the current namespace of the server, the server remains in this namespace. If NameSpace is different from the current namespace, then the namespace of the server is changed before the code is executed. The server job is then left in the new namespace.
- You can use a special kind of indirection. In this case, you specify a *formal namespace*, which is translated to an *actual namespace* at runtime by means of a registry entry.

  To establish a formal namespace, use syntax of the following form:

  ```
  VisM.NameSpace="@MyNS"
  ```

  Here *VisM* is the name of the VisM instance, and *MyNS* is the formal namespace.

  The registry must include an entry at Cache/CacheDirect/FormalNamespaces. Each entry is a subkey whose name is the formal namespace (for example *MyNS*) with a single string value, named `translation`, whose value is the actual namespace to use on this machine.

  With this option, you can easily build and test an application in a test namespace and then deploy it to another namespace when it is ready, without modifying the code. This option also permits you to use the identical application on different machines, with a different namespace on each.

Remember that changing namespace is a relatively expensive operation. In particular, the global and routine buffers are purged. For example, it would be undesirable to have messages alternating between namespaces, which would entirely eliminate the advantages of buffering data in memory. If you need to work alternately in two namespaces, you should

instead establish connections to two server jobs, each running in its own namespace – then each job can take advantage of the buffering efficiencies.

2.4 Executing Code

There are two general ways to use Caché Direct to execute ObjectScript:

- One technique is to use the **Execute()** method. The argument for the method is a line of code. This is the shortest way to the most common use of the control, executing a line of ObjectScript code on command. This technique is shown in the example later in this section.
- The less common technique is to put the code into the Code property and set the ExecFlag property to 1 (execute immediately). As soon as the ExecFlag property has been set to 1, you have effectively pushed a virtual execute button. The VisM immediately assembles a message to the server, sends it off, and receives the reply. When the reply has arrived, ExecFlag is set back to 0 (idle), and the virtual execute button is released.

  There are other possible settings of ExecFlag for use in special situations. See the reference section for the VisM.ocx properties.

These techniques have exactly the same net effect. (When you call **Execute()**, the client stores the string temporarily in the Code property and internally changes the ExecFlag setting to 1.) Choosing one method or the other depends on the application. If the code string is not changing, or is chosen by some separate computation, the ExecFlag technique is slightly more convenient. If you just need to execute a line of code, the **Execute()** technique is more convenient and more direct.

The line of code to be executed can be any legal line of code or expression.

If you call **Execute()** and the client fails to connect to a remote Cache instance, Caché Direct will try to connect to the default server. This behavior allows an application, including a non-interactive one, to connect to a default server without having to prompt the user or code the server into the application. If there is no default, then the client will prompt the user with a 'Communication Error' dialog. Clicking the 'Cancel' button causes the client to connect to the local default instance.

Because it is common to evaluate expressions, there is a useful convention: the server looks at the beginning of the line of code and tries to determine if the line has the form of an expression. Specifically, it checks whether the line begins with an equal sign or at least one dollar sign. If so, the server code prepends either `"Set VALUE "` or `"Set VALUE = "` to create a command that sets the VALUE property to the result of the expression. For example, `"$zv"` on the client side would be expanded to `"Set VALUE=$zv"` on the server side. Except for this special treatment, the VALUE property is no different from any of the other mirrored properties; see the next section.

2.5 Using Mirrored Properties

Caché Direct mirrors the values of certain VisM properties between the client and server, as follows.

1. When the client communicates with the server, it creates a message that contains the values of the mirrored properties (as well as other needed information). When the server receives the message, it creates and sets local variables that have the same names.
2. On the server side, you can use the local variables in the same way you would use any other local variables. There are no limitations on how you can use them, including, for example, indirection.
3. When the server replies, it creates and sends another message that contains the values of all the mirrored properties, *whether or not they have changed*, as well as other information.
4. The server then destroys (via **$KILL**) these local variables (but leaves your other local server variables untouched).

Therefore the mirrored variables are not preserved on the server between calls; they exist only while the server is executing a client command.

2.5.1 Using Basic Mirrored Values

The basic VisM properties that are mirrored are string properties named P0, P1, P2, P3, P4, P5, P6, P7, P8, P9, and VALUE. When the server is executing a client command, the server process will have local variables with the same names, with values equal to the properties in VisM. A change on the server is followed by a change to the properties on the client when the response message is received from the server.

For the VALUE property, there is one additional feature. If the value of the Code property is an expression (that is, if it begins with a dollar sign or an equal sign), then the server returns the result of that expression in the VALUE property, as noted in the section on Executing Code.

2.5.2 Using PLIST

One other VisM property is mirrored, the PLIST property. Caché Direct uses this property to pass array-like values between the server and client. Because the client and Caché have different representations of arrays, it is important to understand the transformations in both directions. The PLIST property has a different form in these two environments:

- On the client, PLIST is a pieced list that contains all the “array” elements, appended to each other with a delimiter. The delimiter is specified by the PDELIM property, which is used only on the client. For example, if PDELIM were equal to "^" then PLIST could equal "first^second^third".
- On the server, PLIST is a one-dimensional array with the following form:
  - PLIST contains a number that indicates how many items were in the list on the client.
  - PLIST(1) contains the first list element.
  - PLIST(2) contains the second list element.
  - And so on.

If the PDELIM property is an empty string, PLIST is considered to be a single piece.

When the server assembles its reply message, it treats PLIST as follows:

- If PLIST equals the number of elements that are currently in the list, the server returns the PLIST array as is.
- If PLIST is less than the number of elements that are currently in the list, the server adjusts the PLIST array by removing elements from the end of the list.

  For example, on the client you may have the following code:

```
VisM1.PDELIM = "^"
VisM1.PLIST = "armadillo^beaver^cobra"
```

  On the server, you will see

  - PLIST=3
  - PLIST(1)="armadillo"
  - PLIST(2)="beaver"
  - PLIST(3)="cobra"

  Suppose the server then runs this code:

```
Set PLIST(3)="cat"
Set PLIST(4)="donkey"
```

When you return to the client, the PLIST property will be "armadillo^beaver^cat". The third list element is updated, but the fourth list element has not been returned, because we have not updated the list count.

- If PLIST is greater than the number of elements that are currently in the list, the server adjusts the PLIST array by adding empty strings as elements, adding them in the empty list positions.

  Consider the previous example, suppose that the server code does the following instead:

  ```
  Set PLIST=5
  Set PLIST(5)="eagle"
  ```

  Then the client property would become "armadillo^beaver^cobra^^eagle".

- If PLIST does not equal a positive integer, the server counts the number of elements in the list — including any elements with nonnumeric subscripts — and sets PLIST equal to that number. (In contrast, in all the previous cases, the server returns only array values with numeric subscripts.)

  Note that both the count and the values returned are computed with **$Order**. This has the effect of eliminating the requirement of sequential, numbered entries in the array. This means that you can make use of the string subscripts and automatic sorting that Caché provides.

  Thus, either of the following would produce the same result as the four-piece example above:

  ```
  Set PLIST=""
  Set PLIST(10)="donkey"
  ```

  Or

  ```
  Set PLIST=""
  Set PLIST("don")="donkey"
  ```

The preceding behavior means that typically if the server changes the number of list elements, it should either update PLIST to the new count or clear PLIST (and let the server count the list elements).

2.6 Callbacks to the Visual Basic User Interface

If you use Visual Basic, VisM includes a special feature that allows your ObjectScript code to refer to elements of the client user interface. These property and method references are called *callbacks* because they cause a message to be sent from the server to the client when the reference is made. The client looks up the form and control, issues an OLE call, and returns the result to the server.

Specifically, you can access properties of any controls on the client user interface, via getter and setter methods. Your ObjectScript can *also* use the following Visual Basic methods, if your form includes controls that provide these methods:

- **AddItem()**
- **RemoveItem()**
- **Refresh()**
- **Move()**
- **Clear()**
- **Hide()**
- **Show()**

- **SetFocus()**

2.6.1 Requirements to Support Visual Basic Callbacks

In order to make callbacks available for a particular form, there are two requirements:

- The form must contain a VisM control.
- Before you make the first server reference to a control on that form, make a call like the following (usually in the Form_Load event code):

  ```
  CreateDispatch Me, VisM1
  ```

  Here, `Me` is the Visual Basic reference to the current form, and VisM1 is the name of the VisM on that form. The **CreateDispatch** function is part of the <cache-install-dir>\dev\cdirect\VBRUN.BAS module, which must be included in your project. This function creates a list of the controls on the form and stores them, for future reference, in a hidden property of the VisM.

2.6.2 Referring to Properties of a Control

To refer to a property of a control, use the following special ObjectScript syntax:

```
_[formname!]controlname[(controlindex)].propertyname[(propertyindex)]
```

The elements in square brackets are optional, and the brackets are not part of the syntax. The leading underline is required. If you omit *formname*, it is assumed to be the form whose VisM sent the current message. The *controlname* and *propertyname* are required. If the control is one of a collection, *controlindex* is required. If the property is a collection, *propertyindex* is required. For example, to get the text property of a text control named txt1 on the current form, use the following code:

```
Set x=_txt1.Text
```

To set the text property, use the following code:

```
Set _txt1.Text="something"
```

Because the underline is also a concatenation operator in ObjectScript, if there is any ambiguity about its meaning, use parentheses.

2.6.3 Executing Methods of a Control

To execute a method of a control, use the following syntax:

```
Do _[formname!]controlname[(controlindex)].method[(args)]
```

For example, to add an item to a list box named list1, use the following code:

```
Do _list1.AddItem("item data")
```

2.7 Using Windows Functions and Caché Utility Functions

Your ObjectScript code can also use the following general Windows functions:

- **MsgBox***(message,style, title*)
- **DoEvents**

- **WinExec***(name,flag)*

Also, it can use the following Caché Direct utilities, also available in the **%CDSrv** routine:

- **GetClientIP()** returns the IP address of the client
- **GetSvrNode()** returns the name of the server machine
- **GetTCPDevice()** returns the identifier of the TCP device that is serving this channel, in Caché format

To execute any of these functions, use syntax like the following:

```
Set varname=$$function^%CDSrv(arguments)
```

For example, to call the Windows **MsgBox** function, use code like the following:

```
Set reply=$$MsgBox^%CDSrv("Are you finished?",1)
```

2.8 Understanding Message Constraints

All communication between the CDConnect and the server is done as Caché Direct messages. These messages are sent over TCP connections (even when all components are local).

There are certain constraints on messages that in turn impose constraints on how you set the properties whose contents are sent in the messages. These constraints also affect how you write callbacks. While you do not need to know the internal details of the message structure, a general description is useful. Generally, a message consists of a 56-byte header followed by a series of fields for the data. Flags in the header describe the type of message and, by implication, what fields to expect. Types of messages include `NewTask`, `BeginTask`, `ExecuteCode`, `EndTask`, and other types. In addition to internal information, the message includes the mirrored properties plus a few additional ones noted in the Other VisM Properties section.

2.8.1 Unicode and Locale Issues

For all properties that are sent to the server, the values must have only text characters. This restriction also applies to any Visual Basic properties that are set or retrieved by Caché Direct; see the section Callbacks to the Client User Interface. (There are exceptions to this constraint that work in some environments, but the general requirement still exists.)

Caché servers operate in either 8-bit or 16-bit (Unicode) mode. In the initial communications between the client and server, the server notifies the client of the mode of the server. Then:

- If the server is Unicode, all clients convert all strings to Unicode before sending them to the server. This may make the messages larger, but then any client, operating in any locale, can communicate reliably with the server.
- If the server is in 8-bit mode, none of the clients convert strings in any way. If any client is not using the same locale as the server, then it is likely that data will be lost or confused when the server writes to the database, because some characters have different meanings in the different locales. Therefore, it becomes the responsibility of your application to either convert or interpret what it receives from the client.

Note: The requirement for properties to be strings arises from the conversions that are performed for Unicode servers. However, there are two situations in which non-text data is preserved: 8–bit servers and control codes with values in the range $c(1) to $c(31).

Because no conversion occurs for 8–bit servers, no corruption can take place. And because the low-range control codes are the same in all 8–bit locales and in Unicode, they also survive for any server.

However, because the client is written in C++ and uses C string conventions, embedded null values ($c(0)) may cause strings to be truncated.

2.8.2 Message Size

For historical reasons, the limit on the size of a single message in either direction between the client and server is 32Kb. Caché Direct does not split long messages into multiple shorter messages and cannot recover from an attempt to send a message that is too large. As a result, it is the responsibility of your application to make sure that no attempt is made to construct a message that exceeds this limit. To transmit more data than that, you send a series of messages. Tests have shown that there is no speed penalty for this arrangement, because TCP breaks large messages into small segments for transmission. All else being equal, optimum speed – in terms of total bytes per second – seems to be achieved with messages in the 12-20 Kb range.

To estimate message size, consider all characters in the VisM properties that are sent to the server, plus a few hundred bytes of overhead. The properties sent are the mirrored properties, plus a few smaller ones like CODE and NameSpace. A normal message has about a dozen fields plus the pieces of the PLIST property, which are each transmitted as a field. The data fields are one byte per character if the server is 8-bit or if the string is all Latin-1 characters. (Latin-1 characters have no bits on in the high byte of their Unicode representation. Such strings can be sent as 8-bit strings.) If the server is Unicode and the property contains any non-Latin-1 characters, the field contains two bytes per character. The result is that the maximum capacity of a message, instead of being about 30K characters, is less than half of that. A good guideline, which does not sacrifice any efficiency, is to make sure none of your messages are larger than about 12K characters.

2.9 Examples

2.9.1 Simple Example: A Lightweight Terminal

This is a minimal sample application that shows how to create a single client form in Visual Basic, where the user can enter a single line of ObjectScript and get its result. You will use defaults wherever they are available. There is also almost no error checking.

This sample assumes you have a Caché server running on your local machine and have also installed the Caché Direct components. Make sure Caché is running locally on your machine.

Start Visual Basic. Create a new Standard EXE project with one form. Using the Project/Components menu, add the VisM control to your toolbox. On your form, add a command button (`Command1`), two text boxes (`Text1` and `Text2`), and a VisM control. Make the first text box wide enough to enter a simple line of ObjectScript and the second big enough to hold a result string. In the Command1_Click event code, enter the following:

```
VisM.Execute Text1.Text
If VisM1.Error <> 0 Then
   Text2.Text = "Error " & VisM1.Error & ": " & VisM1.ErrorName
Else
   Text2.Text = VisM1.VALUE
Endif
```

In the `Form_Unload` method, add the following line:

```
VisM1.Server = ""
```

Run the project. When the form appears, click the command button. You should get a **Choose Server Connection** dialog. Choose **LOCALTCP** and press **OK**. Leave the `Text1` field empty and click the `Command1` command button. A syntax error message should appear in `Text2`, because you did not enter a line of ObjectScript.

Now enter "$H" in `Text1`. Click the command button again. The current date and time in $H format should appear in `Text2`.

Then exit the application.

2.9.1.1 Explanation

When you click the command button, the contents of `Text1` are temporarily put into the VisM Code property and sent to the server to be executed (by means of the VisM **Execute()** method). On the first click, because "Text1 "(the default contents of the text box) is not valid ObjectScript, you receive a syntax error in the Error and ErrorName properties. The code in the Command1_Click event routine displays the error in `Text2`.

In the second case, the server can execute the code you provide, and it sets the VALUE property to the current **$H**. The code in the Command1_Click event routine again displays the contents of the VALUE property in the `Text2` text box.

As you exit, the code in the Form_Unload event routine sets the Server property to the empty string, which disconnects the client from the server gracefully.

On the one hand, this is clearly a very simple example. On the other hand, it is also very powerful. Any line of ObjectScript can be executed on the client and any (small) result can be retrieved and displayed. A value from a global could be retrieved. A computation could be performed and the result returned. You could start a long-running background process via **$JOB**.

2.9.2 Another Example

Consider the following example code:

```
VisM1.P0 = "pig"
VisM1.Execute "Set VALUE=$e(P0,2,$l(P0))_$e(P0,1)_""ay"""
Print VisM1.VALUE
```

This would result in the VALUE property being set to "igpay". In slightly more detail, the P0 property is sent to the server and becomes a local server variable named *P0*. The server executes the line of code, which computes a variable named *VALUE*, and sends a message back to the client. The client updates the VALUE property correspondingly and then prints it.

3

Additional Features

The previous chapter discusses the basic requirements for a client application that uses Caché Direct. This chapter discusses how you can specify custom processing to perform at various times of the overall client/server interaction. Many of these features are hooks included in specific parts of the typical client/server interaction; most of these are specific to the server. The client has some additional features as well.

This chapter contains the following sections:

3.1 Overview

This section provides an overview of the client/server interaction and the server behavior in general. First, the typical client/server interaction consists of the following steps:

1. The client connects to the server and sends a NewTask message to it.
2. The client then sends a BeginTask message.

   If no license slot is available, the server returns a <No License> error and disconnects.
3. The server checks the **BeginTaskHook**, to which you can add your own processing for the server to execute.
4. The client then typically sends any number of ExecuteCode messages. Just before the client sends each message, there is a client-side write hook where you can add your own processing.
5. The server receives the message, performs any processing specified in the server read hook (**ReadHook**) and then reads the message.
6. The server executes the code as required.
7. The server performs any processing specified in the server write hook (**WriteHook**) and then sends the message.

8. The client receives the message, performs any processing specified in the client-side read hook and then reads the message.
9. At the end of the interaction, the client sends an EndTask message, which breaks the connection.
10. The server receives the EndTask message and then checks the **EndTaskHook**, to which you can add your own processing for the server to execute.
11. The server checks the **ShutDownHook**, to which you can add your own processing. The server then shuts down.

    Note: In contrast to your custom EndTask processing, your custom shutdown processing *also* occurs outside of the client/server interaction that is described here. The shutdown processing occurs if the server shuts down for any reason, such as a timeout failure.

As you can see from this description, Caché Direct provides hooks that you can use to insert processing just before messages are sent (a write hook) and just after they are received (a read hook). You can use these hooks for such tasks as custom logging, compression, encryption, or any other purpose chosen by the application. The write hook on the client is paired with the read hook on the server, and vice versa. If a write hook transforms a message, the corresponding read hook should ensure that the message is back in the format expected by Caché Direct.

3.2 Error Trapping

The server traps any nonfatal errors and reports them to the client, via the following procedure:

1. Set values for the server local variables *error* and *errorcode*.
2. Call the function named by the *local* error hook (**%cdEHOOK**).
3. Send a message to the client, passing the current values of the server local variables *error* and *errorcode*.

3.2.1 %cdEHOOK Local Variable

The **%cdEHOOK** error hook is a local variable on the server (that is, within an individual job). To use it, you set it equal to a string that contains a routine name in the form `label^rtn`. The server evaluates the variable when an error occurs. If the variable is not empty, the server then calls the routine. If the variable is empty, the server continues to its next step.

You use this hook to specify additional error processing. For example, you can extend the error message. Your function can use the local variables *error* and *errorcode* in any way, provided that it does not render them unusable by the rest of the error-handling routine. (That is, the variables must still be defined and must still be text and a number that fits in two bytes, respectively). For example, your code could change or expand the error number or text for better use by the client application code.

3.3 The Keep Alive Feature

In a client/server application, there are many ways for the client and server to lose connection with each other. Caché Direct provides a way for the client and server to periodically check the connection and to respond appropriately if it has been lost. For the server, shutting down gracefully is the only meaningful response. For the client, however, you might want to establish a new server connection, for example, or display a message for the user.

This feature includes the following elements:

- An initial setting of the keep alive interval, discussed in the following section.
- Properties on the client that specify the keep alive interval and timeout, discussed in a following section.
- The ShutDown event on the client, discussed in a following section.
- A hook on the server to which you can add your own processing. The ShutDownHook is used any time the server is shut down; for information, see the section Other Server-side Hooks (Global Variables), later in this chapter.

3.3.1 Initial Keep Alive Interval

The server shuts down if it has not received a message from its client within five keep alive cycles. When the server starts up, its initial KeepAliveInterval is 17280 seconds (1/5 of a day). So, by default, the server will shut down if it has not heard from its client in five keep alive cycles of 17280 seconds each (a total of 86400 seconds/24 hours). A lower setting like 300 (5 minutes) would usually be reasonable.

3.3.2 Keep Alive Settings (Client)

The client controls the keep alive interval and resends this value each time it sends a keep alive message to the server. The VisM has two properties that control the keep alive behavior:

- The KeepAliveInterval property specifies the communication idle time in seconds, for the purpose of the keep alive mechanism. When no client/server communication has occurred for this many seconds, the client sends a keep alive message to the server. If the client does not receive a reply within the period specified by KeepAliveTimeOut property, the client fires its ShutDown event. If it does receive a reply, it waits again and sends another keep alive message.
- The KeepAliveTimeOut property specifies a timeout for the keep alive round trip, which should be shorter than the general timeout period, TimeOut.

3.3.3 Shutdown Event for Keep Alive Failure (Client)

If the client does not receive a response to its keep alive message as noted previously, the client then does the following:

1. It changes the value of the ConnectionState property.

   This property always indicates the state of the connection. If ConnectionState is a zero, the connection is OK or a successful disconnect has occurred. If the property is nonzero, then it indicates the time of day (in seconds since midnight) when the server was lost. (This is the same as the second piece of **$Horolog**. The day is not indicated; it is presumed to be recent.)

2. It then fires the ShutDown event, passing one argument to this event, namely, the current value of the ConnectionState property.

If the client attempts to send a message after the connection is lost, a <ServerLost> error results.

Note: This event is triggered only if the client does not receive a response to its keep alive message as noted previously.

3.4 The Server Read Loop and Quit Check

After a Caché Direct server process has been established, it waits in a polling loop that begins with a timed read, listening for communication from the client. The read timeout is ten seconds. If the read is completed before the timeout, the server processes the client communication, writes the response message back to the client, and returns to the start of the polling

loop. If the timeout expires first, however, the server follows a specific procedure to determine whether this server process should shut down, as described next.

3.4.1 Server Quit Check Procedure

If the read loop times out, the server follows a specific procedure to determine whether this server process should shut down. If the server determines that it should shut down, it calls the **ShutDownHook**, performs any processing specified there, and then shuts down. Otherwise, it continues to the next step in the quit check procedure.

Several of the server-side hooks appear within this procedure, which is roughly as follows:

1. Call the function named by the *local* idle-time hook (**%cdPULSE**), and, if requested, perform idle-time processing. If the function returns 0, the server continues the server quit check procedure. If the function returns 1, the server shuts down.
2. Call the function named by the *global* idle-time hook (**IdleHook**), and, if requested, perform idle-time processing. If the function returns 0, the server continues the server quit check procedure. If the function returns 1, the server shuts down. For more information, see the section Other Server-side Hooks (Global Variables), later in this chapter.
3. Calculate the amount of time since the last communication from the client. If more than five keep alive intervals have passed without any communication from the client, the server shuts down.
4. Check to see whether the system has received a shutdown signal. If so, the server shuts down.
5. Check to see whether the server has received the **Stop^%CDSrv** command. If so, the server shuts down all Caché Direct server jobs.
6. Check to see whether the server has received the **StopJob^%CDSrv** command. If so, the server shuts down the specified server job.
7. Check to see whether the slave server should quit. If so, the server shuts down.

Note: At any step, if the check indicates that the server should shut down, the server calls the shutdown hook (**ShutDownHook**), performs any processing specified there, and *then* shuts down.

3.4.2 %cdPULSE Local Variable

This **%cdPULSE** hook is present only as a local variable (that is, within an individual server job). To use it, you set it equal to a string that contains a function call in the form `$$label^rtn`. The server evaluates the variable at a specific time as described in the previous section. If the variable is not empty, the server then performs the designated function call and quits if the result is not `0` or `""`. If the variable is empty, the server continues to its next step.

3.5 Read and Write Hooks

As noted earlier, Caché Direct provides hooks that you can use to insert processing just before messages are sent (a write hook) and just after they are received (a read hook). You can use these hooks for such tasks as custom logging, compression, encryption, or any other purpose chosen by the application. The write hook on the client is paired with the read hook on the server, and vice versa. If a write hook transforms a message, the corresponding read hook should ensure that the message is back in the format expected by Caché Direct.

Note: **Expected Message Format**

The first four bytes of the message are a 32-bit integer that contains the length of the message. You must be sure to update this if you change the length of the message.

It is the responsibility of the programmer to ensure that the server and client routines correspond appropriately with each other.

3.5.1 Server-side Read and Write Hooks

The hooks on the server side use the same general mechanism as other server-side hooks. For details, see the section Other Server-side Hooks (Global Variables).

3.5.1.1 ReadHook

You use this hook to specify processing that the server should perform right after it receives an ExecuteCode message. The server calls this hook as soon as it receives an ExecuteCode message from the client, before assigning values to the mirrored properties.

Your function uses the local variable named *%cdMSG*, which contains the incoming message. The function should perform its actions and return the converted message. If the message was not changed, just quit: `Quit ^%cdMSG`

3.5.1.2 WriteHook

You use this hook to specify processing that the server should perform before it sends a response to the client. The server calls this hook just before it sends the message to the client.

Your function uses the local variable named *%cdMSG*, which contains the outgoing message. The function should perform its actions and return the converted message. If the message was not changed, just quit: `Quit ^%cdMSG`

3.5.2 Client-side Read and Write Hooks

On the client side, you install the hooks by creating a DLL named CDHooks.dll (for 32-bit systems) or CDHooks64.dll (for 64-bit systems) in the same directory as the VisM.ocx or VisM64.ocx file, respectively. The hooks are entry points with specific names and signatures, namely:

```
unsigned char* ReadHook(unsigned char* pInMsg);
unsigned char* WriteHook(unsigned char* pInMsg);
void FreeHookMem(unsigned char* pMem);
```

The **ReadHook** and **WriteHook** routines are expected to behave as follows:

- Take an input argument that is a pointer to a block of bytes, the incoming message.
- Return a pointer to a block of data.
- Return a pointer to the newly allocated data. (In that case, the client will copy the new data block and then free what was returned with the **FreeHookMem** routine.)

If the return value is the same as the argument, then no memory will be freed. This case would be a situation where the data was either not changed or occupies the same space as the original message. It is your responsibility to write client routines that correspond appropriately with your routines on the server side.

The **FreeHookMem** routine is expected to free the memory allocated by the other routines.

3.6 Other Server-side Hooks (Global Variables)

This section provides reference information for other server-side hooks that are kept in a global variable.

- BeginTaskHook

- IdleHook
- EndTaskHook
- ShutDownHook

All the hooks in this section use the same general mechanism. You specify a value for the global **^%CDSwitch("***HookName***")**, which should be a string that contains a function call in the form `$$label^rtn`. The server evaluates the global at a specific time; see the Overview section of this chapter. If the global is not empty, the server then performs the designated function call. If the global is empty, the server continues to its next step.

For all of these, execution takes place by indirection, as follows:

```
Set @("r="_^%CDSwitch("HookName"))
```

where *r* is the return value.

3.6.1 BeginTaskHook

You use this hook for activities such as adjusting protection parameters or changing namespace for the process. The server calls this hook after creating the server job and before executing any code; see the Overview section of this chapter.

Your function can use the following local server variables as arguments:

- *username* – the Caché Direct username
- *taskname* – the name of the executable that is being run on the client, such as `myapp.exe`
- *clientIP* – the IP address of the client, in string form, such as `127.0.0.1`. This may or may not be useful, depending on how the client is connected. For example, a non-TCP Citrix connection receives an artificial IP address to satisfy the Caché licensing system.

Your function should return either:

- Success (the value 0)
- Failure (a string of the form ^*errornumber*^*errorname*", where *errornumber* may not be 0). The range 20900-20999 has been reserved in the Caché Direct error numbers for application-created errors. In the case of failure, the error number and text will appear in the VisM properties Error and ErrorName, and the error event will be signaled.

3.6.2 IdleHook

You use this hook to specify server-side processing to occur when the server is otherwise idle, for example, when the polling read interval has timed out. The server calls this hook as part of its quit check procedure; see the section Server Quit Check Procedure in the overview of this chapter.

3.6.3 EndTaskHook

You use this hook to specify any additional processing that the server should perform if it receives an EndTask message from the client. This hook allows for any application cleanup.

3.6.4 ShutDownHook

You use this hook to specify any additional processing that the server should perform when it shuts down for any reason. The server calls this hook whenever it shuts down.

The function should return either 0 (if the server should not shut down) or 1 (if the server should shut down).

3.7 User Cancel Option

When VisM sends a message to Caché, the server may take some time to process the message request, which leaves the user waiting for a response. If the delay is long, you can give users the option of canceling the request. To set this, you use the PromptInterval property. This property specifies how long, in seconds, the client application should wait (if the server has not yet responded) before displaying a prompt to the user. This prompt would give the user the option of waiting longer or of canceling the activity. The MsgText property is a four-piece string that specifies the message to use in this situation; for details, see the Other VisM Properties section of the chapter VisM.ocx Control Details.

4

Best Practices

As a general statement, communications and CPUs are very fast. In particular, unless the tasks are large, the computer can keep up with many users who are making small requests. However, the load on a client machine is only that imposed by a single client. On the other hand, the load on the network and the server is the accumulation of all the clients at the same time. So practices that increase message traffic, message size, or server load — even by a small amount — can have significant effects on overall throughput and capacity. Several of the following practices are aimed at minimizing these cumulative loads and thereby improving the throughput, response time, or total capacity of the system.

4.1 Clear Unused Properties

Each time the client sends an execute message to the server, the message includes fields for the NameSpace, Code, VALUE, P0 through P9, and PLIST properties. The return message drops the NameSpace and Code fields and returns the new values of all the other properties, *whether or not they have changed*. So the size of the message (and the time to assemble, transmit, and disassemble the message) depends on the total size of the current values of all the properties. If there are many busy clients, the effect on the network bandwidth and server capacity can become significant.

Although communications are fast and the messages relatively small, if you are sending many messages and speed is a concern, it is good practice to clear any properties that are not being used in a given message so they do not consume bandwidth. Remember that these VisM properties are intended for communication with the server and are inefficient as longer term storage on the client.

Also remember that the mirrored properties cannot be used as storage on the server, since they are created and destroyed for each message.

4.2 Disconnect Explicitly at Application Shutdown

While it is a small effect, if you shut down the client without first disconnecting from the server, you cause a TCP error on the channel. This causes an I/O error to be raised in the server code, which responds by shutting down the server job. (This is considered a valid response because the server has no further use if it cannot communicate with the client.) It is good practice to disconnect explicitly instead, allowing the server to shut down gracefully.

4.3 Recursive or Asynchronous Server Calls

All communication on each server channel is synchronous, that is, each message must be sent and received before the next message can be sent. (This requirement is enforced by Caché Direct, which will return an error condition if you try to send a message while another is pending.) For the most common case of a single-threaded client and a single server, this does not become a problem. However, there are ways around this safety – events that do not happen synchronously. Two of these are timer events and the Visual Basic **DoEvents()** method.

4.3.1 Timers on the Client

A timer is, by definition, an asynchronous event. They can be set up so that they do not interfere with whatever other processing is happening. For example, a timer could be set up to check on progress or if some event has happened on the server. But if you send a message to the server that results in a long operation, you tie up that channel for the duration of the operation. If you wish to check on progress, you must either have the server call back periodically and report progress or have the client send an inquiry on a different channel. Another way to perform a long server operation that does not tie up the client is to Job off a separate process to do the work and then return to the client. This does not require another channel, but it does still require a separate job. If a timer goes off while a message is being processed, it can result in an attempt to send another message. In this case, Caché Direct will reject the message with a “nonsynchronous communication” error.

4.3.2 Visual Basic DoEvents Function

Visual Basic has a built-in function, **DoEvents**, that is an explicit call to the Windows event loop. It is often used to allow an immediate repaint while other operations are still in progress. The hazard appears if **DoEvents** causes a server message while one is already in progress. (For example, **DoEvents** would allow the user to continue with another task. This task could result in a server message.) In theory, this can only happen in a callback. **DoEvents** is particularly onerous in this case, since a new, recursive message will be sent to the server, probably destroying the context of the original message. From there, it is a slippery slope. As a general rule, do not have a callback that calls **DoEvents**. If you do, the application should have some sort of flag that disables user input or at least disallows calls to the server until the stack has been unwound.

5

VisM.ocx Control Details

This chapter provides reference details for the Caché Direct ActiveX control (VisM.ocx). It discusses the following topics:

- Extended connection string syntax
- VisM properties
- VisM methods (including a comparison of the and methods)
- VisM events

This control is a wrapper for the C++ classes listed in a later chapter.

5.1 VisM Extended Connection String Syntax

The Server property, **Connect()** method, and **SetServer()** method can all use a connection string, which is a pieced string that uses a colon for the delimiter. Usually it has the following form (as described in Connection Strings and Connection Tags):

```
"CN_IPTCP:server[port]"
```

The first piece of this argument, `CN_IPTCP`, is the connection method, which is always TCP. The second piece is the server name or IP address and port where the Caché superserver is running. For example, you could use the following syntax to set the connection of a VisM named VisM1:

```
VisM1.Server = "CN_IPTCP:127.0.0.1[57772]"
```

5.1.1 Runtime Form of the Connection String

For historical reasons, the connection string can have a slightly different form at runtime. Specifically, if you are connected, the connection string has an odd number of pieces, because Caché Direct inserts a third piece to this property, after the superserver information, as follows:

```
"CN_IPTCP:server[port]:slaveserver[port]"
```

This new third piece indicates the slave server to which you are connected. It has the same form as the master server piece. If you are not currently connected, this property is empty.

Usernames and passwords cannot contain characters that are used as delimiters in the connection string. These include the colon (`":"`, the **$Piece** delimiter), and square brackets (`"["` and `"]"`, used to separate the port number).

5.1.2 Other Forms of the Connection String

The connection string can include the username and password; these are used only if you have enabled the Caché Direct login option. This login option has been rendered obsolete by Caché security and is thus not documented apart from this mention.

CAUTION: Use of this form is discouraged. If you include a password in the connection string, your Caché is susceptible to *any* simple attack.

The connection string can include the username and encrypted password as follows:

```
"CN_IPTCP:server[port]:username:password"
```

In this case, if you are connected, the connection string would have the following form at runtime:

```
"CN_IPTCP:server[port]:slaveserver[port]:username:password"
```

Alternatively, the connection string can include the username and unencrypted password. If you are connecting to a 5.2 or later server, note that Caché requires the password in unencrypted format.

```
"CN_IPTCP:server[port]:username:@password"
```

In this case, if you are connected, the connection string would have the following form at runtime:

```
"CN_IPTCP:server[port]:slaveserver[port]:username:@password"
```

Note: If your client is 5.2 or later, then the client automatically uses your Windows authorization information (username and password) for Kerberos validation if needed (see Connecting with an Indirect Reference for details on connections that use Kerberos authorization). This *supplements* the Caché Direct login (rather than replacing it). If Caché security is not turned on, then Caché Direct bypasses the Kerberos checking, but still passes the username/password if they are given. The server then does whatever is switched on at that end.

5.2 VisM Properties

The VisM control has a set of properties that are mirrored on the server, as well as other properties.

5.2.1 Mirrored VisM Properties

Caché Direct mirrors the values of certain VisM properties between the client and server, as described in the section Mirrored Properties in the chapter Basics of the VisM Control.

If you are using these properties for one-way communication – and especially if they are large – clear them before returning values from the server. Otherwise, your application will waste communication resources. See the chapter Best Practices. As with all other properties that the client sends to the server, the values must have only text characters; see the section Unicode and Locale Issues in that chapter.

P0, P1, P2, P3, P4, P5, P6, P7, P8, P9

These properties are mirrored on the client and server. On the client, they appear as properties of the VisM control; on the server, they appear as local variables, having the same values as the corresponding properties on the client.

PLIST

This property is mirrored on the client and server in a different manner. Because the client and Caché have different representations of arrays, Caché Direct uses this property to pass array-like values between the server and client. The property has a different form on the client and server. For complete details, see the section Using PLIST in the chapter Basics of the VisM Control.

VALUE

This property is mirrored on the client and server in the same way as P0, P1, and so on, with one addition: If the value of the Code property begins with a dollar sign or an equal sign, the server prepends “Set VALUE” or “Set VALUE =” to the start of the Code property. This means that the result is returned in the VALUE property in such a case.

5.2.2 Other VisM Properties

This section lists the other VisM properties (the properties that are not mirrored). Note that some of these properties are sent to the server.

Code

Contains the line of ObjectScript code that is sent to the server for execution. As with all other properties that the client sends to the server, this string must have only text characters; see the section Unicode and Locale Issues.

ConnectionState

This property always indicates the state of the connection. It is used in conjunction with the keep alive feature and tells an application whether the client has had a communication failure and, if so, when the connection was broken.

If ConnectionState is zero, the connection is OK or a successful disconnect has occurred. If the property is nonzero, then it indicates the time of day (in seconds since midnight) when the server was lost. (This is the same as the second piece of **$Horolog**. The day is not indicated; it is presumed to be recent.) This property is a long integer.

ConnTag

Runtime only. Indicates the tag of the CDConnect to which this VisM is connected. If you change this property, you change the tag of the associated CDConnect itself, rather than changing the connection. This property is mainly useful for informational purposes. Note that if this property is an empty string, either there is no connection or there is a connection but no tag is associated with it.

ElapsedTime

Indicates how long it took Caché Direct to process the last message; this is the time from when the client sent the message to the time when the client received a reply. This property is read-only at runtime.

Error

Contains an error number. If it is zero, no error has occurred. This property is read-only at runtime. See the description of the ErrorName property, next.

ErrorName

A string describing an error that has occurred. If it is empty, no error has occurred. This property is read-only at runtime.

The Error and ErrorName properties are set after every server call. If the call is successful, Error is set to 0 and ErrorName is cleared. If an error is reported from the server, the error number is set into Error and a short description is set into ErrorName. While these are not always fully distinctive or descriptive, they still allow the client portion of the application to inform the user that something has gone wrong and to take some action.

Note that errors reported at this level are errors noticed by the server, usually programming errors such as <SYNTAX> or <UNDEFINED>. Logical errors, inconsistencies, and others noticed by the application code should be reported by the application in its results. There are features in the server that allow an application to return error conditions through the Error and ErrorName properties.

ErrorTrap

Controls the handling of communication errors. There are two classes of errors that may occur in a Caché Direct application: errors in the communications process itself, and errors that occur in the application and are reported back to the client.

- Application errors are always reported through the Error and ErrorName properties and the OnError event.
- Communication errors are reported differently, depending on the value of ErrorTrap. If ErrorTrap is False, communications errors are handled with a message box, warning the user of a problem. If ErrorTrap is True, communications errors are reported through the Error and ErrorName properties and the OnError event. The application can then handle them any way you choose.

For historical reasons, the default value for ErrorTrap is False. You should usually set ErrorTrap to True before trying to connect to Caché from the VisM.

This matters only if the application is running without a user or if you want the application to handle such errors automatically.

ExecFlag

A switch that controls when the line of code in the Code property is executed. Its default value is 0, indicating that the client is idle and not sending messages to the server. Possible values:

- As soon as you set ExecFlag to 1, the server executes the line of code in the Code property once (in the context of the P0-P9, VALUE, and PLIST properties). After the server returns, it resets ExecFlag to 0. (You might find the **Execute()** method more convenient than this setting.)
- If ExecFlag is set to 2, this means "execute on reference." That is, any reference to the VALUE property is preceded by an automatic call to the server to execute the code in the Code property. This is useful if the Code property is an expression that represents the current state of something on the server and that you would like to execute again every time you need it. For example, if the Code property were "=$$GetNext^mydata", then the following Visual Basic code could be used to retrieve an array of data from the server:

  ```
  For i = 1 to 1000 array(I) = VisM1.VALUE Next i
  ```

- If ExecFlag is set to 3, this means "execute on interval timer." In this case, a timer (with the interval given by the Interval property), causes the Code to be executed each time the timer goes off.

A common use of the timer option is to do something periodically and use the Executed event to respond after each execution. To use the timer option, use the following overall flow:

1. Set ExecFlag to 0.
2. Set values for all the relevant mirrored properties and for the Interval property.
3. Then set ExecFlag to 3 to switch on the timer.

Remember that all communication with the server is synchronous. The client must receive the reply to the current message before sending the next message. Using timers can occasionally cause the client to try to send a nonsynchronous message. For example, a user might perform an operation that generates a message while a timer-generated message is in progress. In this case, the client will receive a "nonsynchronous message" error, and the message will be not be sent.

Interval

The number of milliseconds between automatic execution of the Code property. The default value is 1000 milliseconds (= 1 second). See the ExecFlag property for value 3.

KeepAliveInterval

Specifies the interval between automatic keep alive messages from the client. It is an integer number of seconds. These messages are sent whenever the client is otherwise idle and the interval has expired.

KeepAliveTimeOut

Specifies how long the client waits for a reply from the server, after sending a keep alive message to the server.

LogMask

Used for debugging on the client side. This is a 32-bit integer property, with each bit assigned as a flag for a particular type of logging. If logging is on at any time during the run of a process, a text log file will be created in the same directory from which the executable is run. Its name will be CDxxx.log, where xxx is the next available sequential number, starting at 000. (So, the first time a log is created, it will be CD000.log.) . The log is closed when the process exits.

To enable client-slide logging, turn all the bits on by setting the value of this property to 2,147,483,647 ($2^{31} - 1$). In Visual Basic, you can use `&H7FFFFFFF,` which is the hexadecimal representation of the same number. To turn off logging, set the value to 0.

The contents of the log are best interpreted by InterSystems personnel, but they include a trace of most of what happened to the VisM and a full dump of all the messages sent to and received from the server. It tends to err on the side of too much information rather than too little. If it is needed, it can be very helpful as a real-time record of what actually happened.

MServer

Has the same purpose as the Server property and can be set to any of the same values as that property. This property is provided only for backward compatibility and should not be used in new applications. See the appendix Notes for Users of the Previous Versions.

MsgText

A four-piece string that specifies the message to display when asking the user whether to cancel (see the PromptInterval property). This property must be a string of the form:

```
"prompt_message|title_text|OK_button_text|Cancel_button_text"
```

The message dialog box that is displayed has a window title (as given by *title_text*) and longer message (as given by *prompt_message*). The dialog box also has two buttons that have text labels.

- If the user clicks the button that is labeled with *OK_button_text*, the dialog box is closed and the query is not interrupted.
- If the user clicks the button that is labeled with *Cancel_button_text*, the dialog box is closed and the query *is* interrupted.

The default value of this property is as follows:

```
"This may take a while. Do you wish to wait?|
Communications|Wait for Reply|Cancel Wait"
```

When you set this property, you can omit any piece. The client uses the default for any string piece that you omit or that you set to a zero length string.

NameSpace

Establishes the namespace context of the routines and globals referenced by the application code. The default value is the empty string. In that case, the routines and globals are referenced in the namespace in which the server is running. When the execution message arrives at the server, if the NameSpace property is not empty and is different from the current namespace, the namespace is changed as indicated. This occurs before the code is executed.

Remember that there is a significant cost to changing the namespace; see the chapter Best Practices.

PDELIM

The delimiter string used with the PLIST property; see the section Mirrored VisM Properties. It is read/write at runtime, which means it cannot be set at design time. For historical reasons, the default value is the string $C(13,10). If it is set to the empty string, there is effectively no delimiter and PLIST is taken as a single element. Note that this property is not sent to the server.

PromptInterval

Specifies how long to wait, in seconds (not milliseconds), before asking whether the user wants to keep waiting or cancel the activity (which is typically a long query). This prompt is displayed only if the server has not yet replied. This prompt would give the user the option of waiting longer or of canceling the activity. The MsgText property specifies the text of the message to display in this case. If the property is zero or negative, the user is never prompted; the default is zero.

Server

This property serves two purposes.

- You set the property to connect to a particular server or to disconnect from the currently connected server.
- At runtime, you can get the property to see what server the client is connected to; in this case, the property value has a slightly different form.

You can set this property to a connection string, a connection tag, a quoted question mark, or an empty string. For details on connection strings, see the section VisM Connection String, earlier in this chapter.

Setting the Server property has the same effect as calling the **SetServer()** method. See the section Comparison of Connection Methods, later in this chapter.

Tag

Not used by Caché Direct. This property exists for compatibility with Microsoft Visual Basic conventions. You may use it any way you wish.

TimeOut

The integer number of milliseconds that the client will wait for a reply from the server. The time is measured from just after the message is sent through TCP to when Windows reports that data has been received. When the timer goes off, meaning that no reply has been received within the allowable time, the connection is broken, which normally causes a TCP error on the server, causing it to shut down. The client then regains control with an error

condition that can be handled. If the application wishes to proceed, it should create a new CDConnect, which will create a new slave server job, with access to the globals (naturally), but none of the local state of the old server job.

If this property is negative or zero, the client will wait forever for a return message. The default value is 60000 (60 seconds).

5.3 VisM Methods

The VisM control provides the following methods:

Connect

Connects this VisM to the specified Caché server, creating a new CDConnect if needed. Use any of the following syntaxes:

```
Connect(connection_string, tag)
Connect(connection_string)
Connect("?", tag)
Connect("?")
Connect(tag)
Connect("")
```

connection_string	A string of the form `CN_IPTCP:`*server_name*`[`*port*`]` where *server_name* is the DNS name or the IP address of the machine that is running Caché and *port* is the port that the Caché superserver is using. Also see the section VisM Connection String earlier in this chapter.
tag	An optional string that acts as the name for the newly created CDConnect object. Note that it is your responsibility to make sure that each tag is unique within a given client process, at any time.
`""`	In this case, VisM connects to most recently opened CDConnect, after first disconnecting if applicable. The original CDConnect is not changed.

For details on the behavior, see the subsection Comparison of Connection Methods.

DeleteConnection

Disconnects from and destroys the CDConnect connected to this VisM, and shuts down the server job.

Execute

This method is a shortcut way of setting the Code property and calling the server. It is exactly equivalent to saving the Code property, setting the Code property to the argument to the Execute method, setting the ExecFlag property to 1 to cause execution, and then restoring the Code property to what it was before the call. All error trapping and execution of the OnError and Executed events occurs in the same way.

LoadRtnFromFile

Obsolete. Do not use.

LoadGblFromFile

Obsolete. Do not use.

SetMServer

Has the same purpose as the **SetServer()** method. This method is provided only for backward compatibility and should not be used in new applications. See the appendix Notes for Users of the Previous Versions.

SetServer

Closes the existing connection for this VisM and creates a new connection, as specified. In contrast to the **Connect()** method, the **SetServer()** method can change the channel of an existing CDConnect. Use any of the following syntaxes:

```
SetServer(connection_string, tag)
SetServer(connection_string)
SetServer("?", tag)
SetServer("?")
SetServer(tag)
SetServer("")
```

connection_string	A string of the form `CN_IPTCP:`*server_name*`[`*port*`]` where *server_name* is the DNS name or the IP address of the machine that is running Caché and *port* is the port that the Caché superserver is using. Also see the section VisM Connection String earlier in this chapter.
tag	An optional string that acts as the name for the CDConnect object. Note that it is your responsibility to make sure that each tag is unique within a given client process, at any time.
`""`	In this case, VisM disconnects from the CDConnect, leaving its channel alone.

For details on the behavior, see the subsection Comparison of Connection Methods, next.

5.3.1 Comparison of Connection Methods

The following table describes the actions of the **SetServer()** and **Connect()** methods.

First Argument	SetServer Method	Connect Method
Connection string	Action depends on whether already connected: • If not yet connected, connect to this server • If already connected, change channel of the CDConnect (and shut down old server process)	Action depends on whether already connected: • If not yet connected, connect to this server • If already connected, disconnect from the original CDConnect and connect to a new CDConnect (original CDConnect is not changed)
Connection tag	Connect to specified CDConnect (disconnecting first if already connected; original CDConnect is not changed) [permitted as of Caché version 2007.1]	Connect to specified CDConnect (disconnecting first if already connected; original CDConnect is not changed)

First Argument	SetServer Method	Connect Method
Empty string	Disconnect from the CDConnect, leaving its channel alone	Connect to most recently opened CDConnect (disconnecting first if already connected; original CDConnect is not changed)
Quoted question mark	Prompt user for a server and then: • If not yet connected, connect to this server • If already connected, change channel of the CDConnect (and shut down old server process)	Prompt user for a server and then: • If not yet connected, connect to the specified server • If already connected, disconnect from the original CDConnect and connect to a new CDConnect (original CDConnect is not changed)

Setting the Server property has the same effect as calling the **SetServer()** method.

Note that multiple VisMs sharing a single CDConnect should not try to communicate simultaneously. If they do, you will get a "nonsynchronous communication error" message.

5.4 VisM Events

VisM has events that are fired after a line of code has been executed. You may attach code to these events to simplify the operation of the client.

Executed

This event is fired after every attempt to execute code on the server, whether successful or not. If an error occurred, the OnError event is fired before the Executed event.

OnError

This event is fired any time the server reports an error to the client (that is, whenever the Error and ErrorName properties are set to non-empty values). If the error occurred while trying to execute some code, OnError is fired before the Executed event.

ShutDown

This event is fired if any server message times out or if the server indicates it is in the process of shutting down. Your application can use this to inform the user, perform a graceful shut down, or attempt a reconnection. The integer argument to this event is the value of the ConnectionState property, the time when the server was lost.

Once a connection has been lost, further attempts to send a message result in a <ServerLost> error.

6

Using Caché Direct in Non-ActiveX Applications

This chapter describes how to use Caché Direct properties and methods from an application such as C++ that does not use ActiveX (where you do not have VisM.ocx). It assumes you have a good understanding of programming in other languages.

6.1 General Procedure

The overall procedure is as follows:

1. Get the Caché Direct header files (.h) from the InterSystems InterSystems Worldwide Response Center (WRC). These files define the C++ classes that provide Caché Direct capabilities.
2. **Include** these files in your application source code.
3. To reproduce the functionality of the VisM object, have your code create an instance of the CDLink class and an instance of the CDParms class. In general:
   - CDLink includes properties such as timeouts and error properties and methods for managing connections to Caché. These are the same as the corresponding properties of the VisM control.
   - CDParms contains all the properties that the client sends to the server. These properties are also the same as the corresponding properties of the VisM control.
4. Set values of properties and execute methods as needed. The same rules and considerations apply in this case as with the VisM control.

6.2 Recommendations

The header files provide a lot of information about the Caché Direct classes, but not all these classes are suitable for direct use. The first recommendation is to use only the CDLink and CDParms classes. These classes expose the appropriate methods of the lower-level classes and ensure that connections are managed correctly. You should ignore classes other than CDLink and CDParms.

The second recommendation is to use the **CtrlExec1()** method of CDLink, rather than the **CtrlExec()** method. The former method is designed for use with the CDParms class, which you use as a container for the mirrored properties.

6.3 Notes

For reference purposes only, you may find it helpful to understand the organization of the Caché Direct classes. First, the primary classes are as follows:

- The CDLink class represents the client. The VisM control is a wrapper around this class. This class exposes the common properties and methods that would be needed by any client. Properties include timeouts, callback function pointer, the CDMsg object, Error, ErrorName and ErrorTrap, keepalive, client tag. Methods include those to manage its own properties and those of its CDConnect object, which it creates as needed. Each CDLink may be attached to zero or one CDConnect.
- The CDConnect class represents the CDConnect object, as described earlier in this document. It includes linkages to the underlying communications DLLs, either the older NTI DLLs or the newer CConnect.dll that manages Kerberos security. It also provides the client side of the read/write hooks. It holds the basic static client information, such as the executable name and signature, and the user and client machine names. It handles the communications threading, timeouts, and the keep alive facility. It provides basic methods like Connect, Send and SendKeepAlive, Receive, and Disconnect, and exposes the Server property and other properties to its clients. It also exposes server properties like whether it is a Unicode server, and the address and port of the server. It keeps a list of its zero or more attached CDLink clients.
- There is also a helper class, CDMsg, that knows how to compose and decompose messages in the proper format. CDMsg provides methods like Init, AppendField, and GetField. The CDLink includes a CDMsg for building messages from its client. There is also a CDMsg in the CDConnect, used for messages which it constructs itself, like the Task and KeepAlive messages.

To support direct access, there are also some helper classes:

- The CDProp class defines some data types and provides conversion methods for them. It is used to represent a single mirrored property.
- The CDParms class is a container for all the VisM properties that the client sends to the server: the mirrored properties and a few additional properties. Internally, this class contains multiple instances of CDProp.

Each CDProp contains a single property. CDParms is a collection of all the CDProps for one client.

7

Logging

In rare situations, you may want to enable client or server logging, particularly to diagnose problems. The server also records all error traps in its error global.

7.1 Client Logging

You can enable client logging in two general ways:

- By setting the LogMask property of VisM.ocx. This property overrides any registry settings.
- By setting the registry switches. The registry settings act as defaults if the property is not set. The registry switches provide precise control over what client processes create logs. Specifically, the switches can now reside in either the HKEY_CURRENT_USER hive or the HKEY_LOCAL_MACHINE hive, and can be modified so that logging only occurs for a particular user, or application program, or combination of the two. The choices are searched in a hierarchy, from most specific to most general, so that general switches can be overridden by more specific ones for special purposes.

  In addition, Caché Direct provides routines to set and get these registry entries, so they can be manipulated easily under program control.

7.1.1 VisM LogMask Property

You can directly set the LogMask property of VisM.ocx. To enable logging, set this property equal to the integer value of `0x7FFFFFFF` (for C++) or `&H7FFFFFFF` (for Visual Basic). For example:

```
VisM1.LogMask = Val(&H7FFFFFFF)
```

The decimal integer corresponding to this hex value is `2147483647`.

To disable logging, set the property to 0. This property overrides any registry settings.

7.1.2 Registry Switches

The client logging switches are registry key values with the following names:

- LogMask is a bit mask that indicates whether logging should take place or not. It takes the same values as the LogMask property described in the previous section, VisM LogMask Property.

- LogFolder allows the logs to be directed to a specific folder. The default is the folder that contains the main executable for the application. The default applies if the value is not given or if it is blank. Also see the section Limiting the Size of the Log Files, later in this chapter.

The switches are located under the key *hivename*/Software/InterSystems/Cache, where *hivename* is either:

- HKCU (the HKEY_CURRENT_USER hive)
- HKLM (the HKEY_LOCAL_MACHINE hive).

In addition, for a specific user running a given application, three other versions of each value are available. They are named by concatenating the user and/or application name to the basic value name. For example, if the username is `Joe` and the application name is `MyApp.exe`, you could also have values named `LogMaskJoeMyApp`, `LogMaskJoe`, and `LogMaskMyApp`, which would be checked in that order. Each would be checked first in HKCU and then in HKLM. The first one found would be used. Similarly, there could be values named `LogFolderJoeMyApp`, `LogFolderJoe`, and `LogFolderMyApp`.

7.1.3 Getting and Setting the Registry Values

Four client-side routines are available to set and get these registry values. They are exposed by the ISLog.dll file. Their C signatures are as follows:

```
DWORD GetRegLogMaskEx(LPCSTR pUsername, LPCSTR pAppname);

DWORD SetRegLogMaskEx(HKEY hiveKey,
                      LPCSTR pUsername,
                      LPCSTR pAppname,
                      DWORD dwMask);

int GetRegLogFolderEx(LPCSTR pUsername,
                      LPCSTR pAppname,
                      LPSTR buf,
                      int buflen);

void SetRegLogFolderEx(HKEY hiveKey,
                       LPCSTR pUsername,
                       LPCSTR pAppname,
                       LPSTR buf);
```

In each case, if *pUsername* or *pAppname* are null or empty strings, those arguments are not used. If those arguments are given, the Get_ routines will search through the appropriate combinations, returning the first one found, if any.

For example, if you call **GetRegLogMaskEx(NULL, "Joe", NULL)**, it will first look for `LogMaskJoe` in HKCU and HKLM and then LogMask in HKCU and HKLM, returning the first one found, or 0 if none are found. If both *pUsername* and *pAppname* are given, then all four combinations will be searched, in the order given above.

For the **SetRegLogMaskEx** and **SetRegLogFolderEx** routines, if HKCU or HKLM is given for the first argument, then that hive will be set. If no hive is given, then HKLM, the default, will be set.

SetRegLogMaskEx returns any previous value of the key it is setting.

GetRegLogFolderEx returns the length of the folder name it finds, and 0 if none is found.

7.1.4 Limiting the Size of the Log Files

Two client-side registry settings (LogSizeLimit) can prevent the log files from becoming too large. The switches are located under the key *HK*/Software/InterSystems/Cache, where *HK* is either HKCU or HKLM as before.

This setting specifies the maximum size of any log file, while the client is writing to the log. When the log reaches that size, the client starts writing to a new log file. If there was a previous older log file, it is discarded, so that there are never more than two log files. When the client shuts down, the two existing log files are combined, and the resulting log can be up to two times the value of LogSizeLimit, but never more than that.

The start of the log file will contain basic information about the session, followed by a divider labeled with the string `Snip`, followed by the most recently logged activity.

7.2 Server Logging

If you enable server logging, the logs will contain a trace of the activity in the server portion of the slave server job, including all the messages that went between the client and server. To enable server logging, use the Terminal to enter the following command:

```
Set ^%CDLOG=1
```

After you enable logging, any new Caché Direct server job creates a text log file, usually in the mgr directory, named CD*xxx*.log, where *xxx* is the **$Job** of the job for which the log is created. Each new job gets its own log file. Once the switch is on, run whatever tests you like.

The switch is checked only as the slave server job starts. If the switch is changed after that, it has no effect on jobs that are already running. So logging cannot be turned either on or off after a job starts.

When the tests are over (or at least started), turn the switch off by setting it to an empty string or by killing it:

```
Kill ^%CDLOG
```

CAUTION: This switch is global and thus affects all new jobs. Also, the logs can get very large. Therefore, it is a good idea to perform logging as briefly as possible, especially if the server is not dedicated to the tests. You could automate the process from the client, running a Caché Direct job that turns on the switches just before the test itself starts and a similar job that turns them off just after the test starts. This could even be part of the test itself, setting the switches, reconnecting by resetting the Server property (which will start a new server job), and then turning off the switches as the first activity after reconnecting. For example, if the client is a Visual Basic application, it might start up with code like this:

```
VisM1.Execute "Set ^%CDLOG=1"
; reset to the same value as before then start a new job
VisM1.Server = VisM1.Server
VisM1.Execute "Kill ^%CDLOG"
```

7.3 Server Error Global

Independently of the server logging that is mentioned above, the server keeps an internal server error log in the global **%CDServer("Error")**. This contains all trapped errors.

Note that if this log includes a line that says “emergency brake,” that indicates that the server detected an infinite error loop (five or more errors within a second).

A

Installation and Upgrade

Caché Direct is installed automatically when you install Caché. Sometimes, however, you may want to upgrade the server without reinstalling Caché or you may want to install the client-side software on a machine where Caché is not installed. This appendix describes how to do both of these things.

A.1 Upgrading Your Caché Direct Server

Caché Direct uses a server that is automatically installed with the rest of the Caché installation, but you can upgrade it easily without performing a new Caché installation. The Caché Direct server code is built to be backward compatible with older clients and all platforms. This means that you can upgrade the Caché Direct server to correct problems or install new features without other major upgrades.

To upgrade your Caché Direct server:

1. Get the latest Caché Direct encrypted text file for your platform from the InterSystems InterSystems Worldwide Response Center (WRC) and place it on the Caché server machine. This file has the extension .enc.
2. Stop any running Caché Direct jobs.
3. In the Terminal, enter the following command:

   Do rload^%CDCrypt(<*path_to_.enc file*>[**,***password*]**)**

   For example:

   ```
   Do rload^%CDCrypt("C:\\temp\\CDCache.enc","SYS")
   ```

This decrypts the source, loads the resulting .int code, compiles it, and destroys the source. The optional password is for the case where the server source is to be preserved. It is usually left off. If the password is provided and is correct, the .int source is retained. If you need to perform some activity that requires the source (such as debugging or testing changes), you can get the password from the InterSystems InterSystems Worldwide Response Center (WRC).

A.2 Installing VisM on a New Machine

The easiest way to install VisM is to install the Caché client software as usual. You can, however, install the Caché Direct client software manually. To do so:

1. Make sure that you have installed the Microsoft Visual 2008 Redistributable Package, which is needed in order to register the VisM DLLs.
2. Copy the following files from a machine where the client is installed:
   - For Caché Direct before version 5.1 32–bit systems
     - NTI.dll
     - NTIIPTCP.dll
     - Cmvism32.dll and all files with similar names
     - ISLog.dll
     - VISM.ocx
   - For Caché Direct 5.1 and later 32–bit systems
     - cconnect.dll
     - Cmvism32.dll and all files with similar names
     - ISCDLink.dll
     - ISLog.dll
     - VISM.ocx
   - For Caché Direct 5.1 and later 64-bit systems
     - cconnect64.dll
     - Cmvism64.dll and all files with similar names
     - ISCDLink64.dll
     - ISLog64.dll
     - VISM64.ocx

   Copy the files from the directory C:\Program Files\Common Files\InterSystems\Cache and paste them into the same directory on the target machine.
3. Open a command prompt, navigate to this directory, and enter the following command to register the ActiveX control (must be run by a user with administator privileges):

   For 32-bit systems

   ```
   C:\Windows\SysWOW64\regsvr32 vism.ocx
   ```

   For 64-bit systems

   ```
   C:\Windows\System32\regsvr32 vism64.ocx
   ```

B

Notes for Users of the Previous Versions

Caché Direct is designed to be backward compatible, but in some cases there are behavior changes you should know about. Some of the architectural changes are worth highlighting for users of previous versions, who may want to adapt their applications to take advantage of the new features.

B.1 IPv6 Issues in Caché Direct

In the current release of Caché Direct, IPv6 address formats are fully supported. In particular, the Server/MServer property of VisM.ocx allows IPv6 addresses within the same general connection string format as previously.

However, depending on how they are used by an application, there may be some hazards that need to be confronted. Specifically, the connection string in Caché Direct is a colon-delimited expression of the following general form:

```
CN_IPTCP:server_address[port]
```

where *server_address* is the master server and can be an IP address, a server DNS name, or the special name `localhost`.

On the return from setting the Server property (which is the mechanism to request a connection to the given server), the property is reset to reflect the result of the connection attempt. If the attempt was successful, a last piece was added containing the slave server address. In recent releases (since the introduction of the superserver), this piece was always the same as the second piece, making it redundant. If the connection attempt was unsuccessful, the value of the property was changed to the empty string, `""`.

Potential confusion arises with IPv6 addresses, which contain colons in the *server_address* part. IPv6 also supports more than one form of loopback address. Caché Direct supports all of these format variations. But, depending on the assumptions made by application code, the new addresses could cause incorrect behavior. The following describes how Caché Direct handles IPv6 addresses:

1. The connection string is still in the same general format, given above. It recognizes the extent of the *server_address* by looking for the square brackets that enclose the port number.
2. IPv6 has its own loopback form (`::1`) and also supports IPv4-style and “mapped IPv4” address forms. These look like `::a.b.c.d` and `::FFFF.a.b.c.d`, where `a.b.c.d` are the decimal bytes of the IPv4 address and `FFFF` is the two-byte sequence of all ones. So IPv4 loopbacks can be expressed as `::127.0.0.1` or `::FFFF.127.0.0.1` or `::FFFF:007F:0100` (low bytes come first in the address). All of these loopbacks are equivalent and don’t represent real addresses. In order to make meaningful comparisons, Caché Direct converts them all, whether from the registry or explicitly from application code, to the name `localhost`, which is recognized by the connection software and handled appropriately.

3. Caché Direct no longer appends a third redundant piece to the connection string if the address is in IPv6 format (i.e., contains any colons). If a connection is successful, the Server property will contain the server to which a connection was made. For IPv6 format addresses, it will look like the simple form of the string it was given. For IPv4 or DNS names, or `localhost`, it will still contain the redundant third piece to maintain backward compatibility with versions of Caché prior to 5.1.
4. If the application code relies on colons only representing piece delimiters in an IPv6 environment, it will likely fail.

The general approach is that, if a human can easily recognize what is meant by the connection string, then the software can, too. The parsing rules for the connection string are:

1. The first part is the same as always, `CN_IPTCP:`.
2. The next piece (the server, in IPv4 or IPv6 form, and port) is terminated by the closing square bracket after the port number. This eliminates any confusion caused by colons in the address.
3. The inclusion of username and password in the connection string has been deprecated since the introduction of Cache security. However, both parameters are still recognized and handled correctly by Caché Direct in the extended IPv6 format.

B.2 Previous Shared Connection Behavior

The old VisM control was a single-threaded, shared connection mechanism. It had an MServer property, which was shared among all the VisM controls in the process. The property was actually a computed value that reflected the connection to the server. If any VisM changed it, it disconnected from any current server and made a new connection to a new server process. If it was set to an empty string, it disconnected from the server.

Communication occurred in a separate thread.

B.3 New Architecture

The new VisM control has a new internal structure with three layers: the VisM objects, the CDConnects to which they are connected, and below that, the server processes that are attached to the CDConnects. The CDConnect has several qualities that are particularly relevant to this discussion:

- The CDConnect can be connected to multiple callers. This means that multiple VisM clients can share the same server process.
- The CDConnect can be disconnected from one server process and connected to another, bringing along all the callers with it.
- The CDConnect does not have to be connected to any server process.
- There can be multiple CDConnects, each with its own server and VisM clients.

Because there are now three layers and because of how the CDConnect layer is designed, there is a great deal more flexibility of Caché Direct applications. You can create client/server relationships that were not previously possible. Also because there are simultaneous *separate* connections, you can have a multithreaded application. Each connection still has a separate communication thread that it manages. Note that each connection must still be synchronous, because the server job itself is single-threaded and can only handle one message at a time.

B.4 Behavior Notes

Given the changes described here, a backward compatibility problem arises for some situations. The new VisM tries to behave the same way for the most common cases. Where it can not, it tries to do the most expected thing.

This version of VisM provides a property called Server, which you use to connect to Caché. For backward compatibility, VisM still has a property with the older name, MServer. In most situations, these properties are interchangeable. These properties do behave differently in the case where the VisM control has not yet been connected to a server.

- If the application *gets* the MServer property and the VisM is not yet connected, it will share the most recent connection, if any, and return that as the MServer property.
- If the application gets the Server property in the same situation, the VisM will not be connected, and the Server property remains as an empty string.

If the application *sets* either the Server property or the MServer property, the behavior is as follows:

- If the VisM was not previously connected, a new connection is created, not affecting any existing VisMs or their connections. This behavior is different from that of previous versions of Caché Direct.
- If the VisM had previously been connected, the connection object that it is using (possibly shared with other VisMs) will disconnect from its server and, if the connection string is not empty, connect to a new one. Only the VisMs that are sharing the channel are affected. This is compatible with the older versions of Caché Direct, which would change the server for all other VisMs.

Also, this version of VisM provides new methods, which you can read about in the chapter Basics of Using the VisM Control. These methods are **Connect()**, **SetServer()**, and **DeleteConnection()**.

As of Caché version 2007.1, you can use a connection tag as the first argument to **SetServer()**. You can also now set the Server property (or the MServer property) equal to a connection tag.

B.5 Other Architecture Changes

- The Caché superserver now takes the place of the old Caché Direct master server routine. Formerly, the master server would receive a request for a connection, spawn a slave server process on a new port, and redirect the caller to the new port. The caller would disconnect from the master server and then connect to the slave server on the correct port.

  Now the superserver receives a request for a connection, spawns a slave server process, and forwards the caller to this process without any intermediate disconnection. No additional ports are used.
- If the Caché Direct server is a Unicode server, all messages are sent in Unicode, eliminating locale issues. (Note that the same is not true for 8–bit servers; see Unicode and Locale Issues.)
- Depending on the version of Caché Direct you are using, this version may have more hooks for message handling, controlling the client/server interaction, and so on.
- Caché Direct now provides transparent support for Caché security, so there is little need for the login hook (**^%CDSwitch("SecHook")**) in the NewTask processing. The hook is still supported, but is no longer documented.

C

Example: Visual Basic Printer Support

InterSystems can provide you with a Caché Direct sample that demonstrates callbacks by using the Windows default printer (the VB Printer object) from ObjectScript. As a by-product, the sample will provide a Visual Basic form that can be included into any project and used or expanded as necessary. The approach can also be easily extended to support the Screen and Clipboard Visual Basic objects.

The sample routine prints itself to the Windows default printer, in several fonts, with comments italicized and page headers and numbers in a different font. This is fairly simple to write and shows most of the printing capabilities you might need. The printing model is that of Visual Basic (and Windows), not Caché. (Slash parameters to the **Write** command could be implemented in a custom GBI device, but they would not provide any more capabilities or convenience than this sample.) The one concession to Caché conventions is in the parameters of the **Print** method, which follow the ObjectScript model, since it is both simpler and more familiar. Note that the arguments to the **Print()** method are quoted strings, because ObjectScript variable names have no meaning to Basic.

C.1 Overview

A common way to execute Basic code from an ObjectScript routine is to put the code into the Click event of a button and set the Value property of the button to 1. This causes the Click event to fire and the Basic code to execute. As a variation, we can exploit the fact that there are other events that fire in response to other actions. In particular, the text control fires a Change event in response to a change in the Text property. This is convenient because we can set the Text property to an appropriate string in a single call from Caché and have the Change event code interpret it. The result, if any, will be returned in the Tag property of the same text control. This approach can be used on a separate, hidden form, in a totally general way, and all in Basic.

C.2 Internal Command Syntax

The sample includes a few simple ObjectScript routines to format command strings and send them to the Printer object. The Text property contains a string consisting of a command and some number of arguments. The commands correspond to the operations to be performed with the Printer object: setting and getting properties and performing methods.

C.2.1 Setting Properties

To set properties, use **SetProp()** followed by one or more comma-separated pieces of the form *property_name=value*. This allows you to set multiple properties in a single call. For example, to access the Printer1 instance directly, use this:

```
Set _Printer1!txtPrint.Text="SetProp:FontName=Arial,FontSize=12,FontBold=1"
```

Or to access the printer through the helper routine, use this:

```
Do SetProp^%CDPrt("FontName=Arial,FontSize=12,FontBold=1")
```

C.2.2 Getting Properties

To get property values, use **GetProp()** followed by one or more comma-separated property names. The response to the query will be set into the Tag property, in the same form as the **SetProp()** arguments. For example, to access the printer directly, use this:

```
Set _Printer1!txtPrint.Text="GetProp:FontName,FontSize,FontBold"
```

Or to access it through the helper routine, use this:

```
Set prop=$$GetProp^%CDPrint("FontName,FontSize,FontBold")
```

The Tag property is then set to a string of the following form:

```
"FontName=Arial,FontSize=12,FontBold=1"
```

This property would also returned as the function result.

C.2.3 Executing Methods

The following methods are available in this sample: **NewPage()**, **Scale()**, **TextHeight()**, **TextWidth()**, **Print()**, and **EndDoc()**. Each of these methods has an equivalent command that you can execute through the helper routine.

The arguments to the **Print()** method follow ObjectScript format control conventions (as in the **Write** command), not Basic conventions. For example, to access the printer directly, use the following:

```
Set _Printer1!txtPrint.Text="NewPage"
Set _Printer1!txtPrint.Text="Scale:0,0,80,60"  or  "Scale"
Set _Printer1!txtPrint.Text="TextHeight:"12""
Set _Printer1!txtPrint.Text="TextWidth:"50""
Set _Printer1!txtPrint.Text="Print:!,?8,""Some text"""
Set _Printer1!txtPrint.Text="EndDoc"
```

To access it through the helper routine, use this:

```
Do NewPage^%CDPrint
Do Scale^%CDPrint(0,0,80,60)  or  Do Scale^%CDPrint( )
Set ht=TextHeight^%CDPrint("12")
Set wid=TextWidth^%CDPrint("50")
Do Print^%CDPrint("!,?8,""Some text""")
Do EndDoc^%CDPrint
```

The arguments to the **Print()** method must be quoted strings, because ObjectScript variable names have no meaning to Basic. See `quote^CDPrTest` in the sample code for a routine that properly quotes strings, doubling up internal quotes.