Process Client Programming Guide

iPlanet™ Integration Server

Version 3.0

August 2001

Contents

List of Figures

List of Procedures

List of Code Examples

Preface

The *iIS Process Client Programming Guide* describes how to write a client application for an iPlanet UDS enterprise application using a programming language corresponding to one of the iIS client APIs. The iIS client APIs are TOOL, C++, CORBA/IIOP, JavaBeans, and ActiveX.

This preface contains the following sections:

Product Name Change

Forte Fusion has been renamed the iPlanet Integration Server. You will see full references to the new name, as well as the abbreviation iIS.

Audience for This Guide

The intended audience for this manual is programmers who have experience developing end user software that presents a graphical user interface and who understand how to use the type of API required by their programming environments. Users of the TOOL API should be familiar with TOOL and the iPlanet UDS application development environment.

Before reading this book, it would be helpful to read the first chapter of the *iIS Process Development Guide*, which explains general iIS application development concepts and the iIS application architecture. It also explains how process definitions, the process engine, and client applications work together.

If you are new to iIS or want to familiarize yourself with the components of iIS and how they interact in an iIS system, refer to the *iIS Conceptual Overview*.

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	**Description**
Chapter 1, "Building an iIS Process Client Application"	General guidance on developing an iIS process client application. Read this chapter before reading the chapter for the API you are using.
Chapter 2, "Building a TOOL Client"	Provides guidance on writing a TOOL client application.
Chapter 3, "Building a C++ Client"	Provides guidance on writing a C++ client application.
Chapter 4, "Building a CORBA/IIOP Client"	Provides guidance on writing a CORBA/IIOP client application.
Chapter 5, "Building a JavaBeans Client"	Provides guidance on writing a JavaBeans application.
Chapter 6, "Building an ActiveX Client"	Provides guidance on writing an ActiveX application.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (iIS, JSP, iMQ).
	Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, there are additional documentation resources, which are listed in the following sections. The documentation for all iIS products can be found on the iIS CD. Be sure to read "Viewing and Searching PDF Files" on page 22 to learn how to view and search the documentation on the iIS CD.

iIS documentation can also be found online at http://docs.iplanet.com/docs/manuals/iis.html.

The titles of the iIS documentation are listed in the following section.

iPlanet Integration Server Documentation

iIS Adapter Development Guide

iIS Backbone Integration Guide

iIS Backbone System Guide

iIS Conceptual Overview

iIS Installation Guide

iIS Process Client Programming Guide

iIS Process Development Guide

iIS Process System Guide

Online Help

When you are using an iIS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iIS distribution: `FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as FNscript or Cscript, type help from the script shell for a description of all commands, or help *<command>* for help on a specific command.

Documentation Roadmap

A roadmap to the iIS documentation can be found in the *iIS Conceptual Overview* manual.

iIS Example Programs

iIS example programs are shipped with the iIS product and installed in two locations, one for process development (using the process engine) and one for application integration (using the iIS backbone).

Process Development Examples Process development examples are installed at the following location:

```
FORTE_ROOT/install/examples/conductr
```

The PDF file, `c_examp.pdf`, describes how to install and run the examples in this directory. The Appendix to the *iIS Process Development Guide* also describes how to install and run the examples.

Application Integration Examples Process integration examples are installed at the following location:

```
FORTE_ROOT/install/examples/fusion
```

Each example has its own sub-directory, which contains a README file that explains how to install and run the example.

Viewing and Searching PDF Files

You can view and search iIS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iIS distribution. You may want to consolidate your iIS documentation with the documentation for your iPlanet UDS distribution.

2. Set up a directory structure that keeps the `iisdoc.pdf` and the `iis` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ **To view and search the documentation**

1. Open the file `iisdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.

3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

   NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	**Keyboard Command**
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `iisdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `iisdoc.pdf` home page or select Edit > Search > Results.

Chapter 1

Building an iIS Process Client Application

Before reading this chapter, read the beginning chapters of the *iIS Process Development Guide* to get an overview of the entire iIS process management system to see how a client application fits into that system.

This chapter covers some general concepts about developing an iIS process client application that uses one of the iIS process client APIs. Topics covered include:

- what a process client application is and how it relates to an iIS process management system
- some typical uses of the process client APIs
- what an iPlanet UDS environment is
- general aspects of iIS process objects, process attributes, attribute accessors, and the effects of lock requests
- what timers are and how to access one

After you read this chapter, go to the chapter on the client API for the language or protocol your application is using (TOOL, C++, CORBA/IIOP, JavaBeans, or ActiveX). For detailed API reference information, see the iIS online help.

What Is an iIS Process Client Application?

An iIS process client application is an application that communicates directly with the iIS process engine using one of the process client APIs (TOOL, C++, CORBA/IIOP, JavaBeans, and ActiveX). A process client provides an interface to an iIS enterprise application. You can have one large client application that provides a graphical interface, or multiple client applications that perform either robotic or interactive activities. Process client applications that support a similar set of processes typically communicate with the same iIS process engine. For more information on the iIS process management system, refer to the introduction of the *iIS Process Development Guide*.

NOTE The iIS Backbone also allows you to incorporate legacy applications into an iIS enterprise application. For more information, refer to the *iIS Backbone System Guide*.

An iIS process client application:

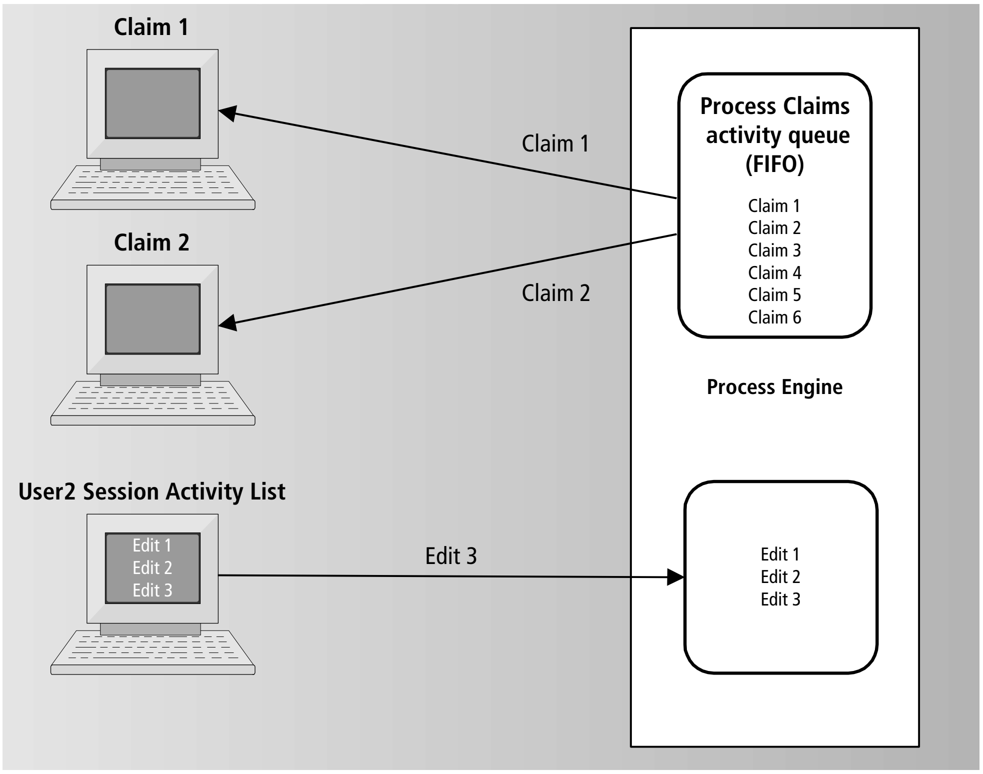
- communicates with the iIS process engine using a client application program interface (API)
- provides access to activities and attributes defined in running process instances
- performs the work indicated by an activity's application dictionary entry, possibly by calling service objects, starting modules in the client application, or executing commercial software
- can run on client or server machines

The user can be a person that the client application communicates with through its user interface. The user can also be a robotic or other automated client that the client application communicates with through a programmatic interface. In either case, the client application can provide a *heads-up interface*, which presents a list of work that a user can choose from, or a *heads-down interface*, which presents only one item of work at a time that a user must perform.

Heads-up and Heads-down Models

Depending on the design of the interface to an iIS enterprise application and the requirements of each process, you can use a heads-up or a heads-down model, as shown in Figure 1-1:

Figure 1-1 Heads-up and Heads-down Client Models



In the *heads-up model*, the client application requests lists of work, and the client selects the work to perform next. This model is used in environments where the workers can choose what to do based on their own schedule requirements and priorities.

In the *heads-down model*, the client application requests only one activity at a time for its user. The iIS process engine picks the next activity on the queue of activities of that type and sends it to the client application. This model is appropriate for environments where work is procedure-oriented and is handled by a pool of workers, such as a clerical pool.

Robotic clients are processes that do not require human intervention. Robotic clients can use offered activities, queued activities, or automatic activities.

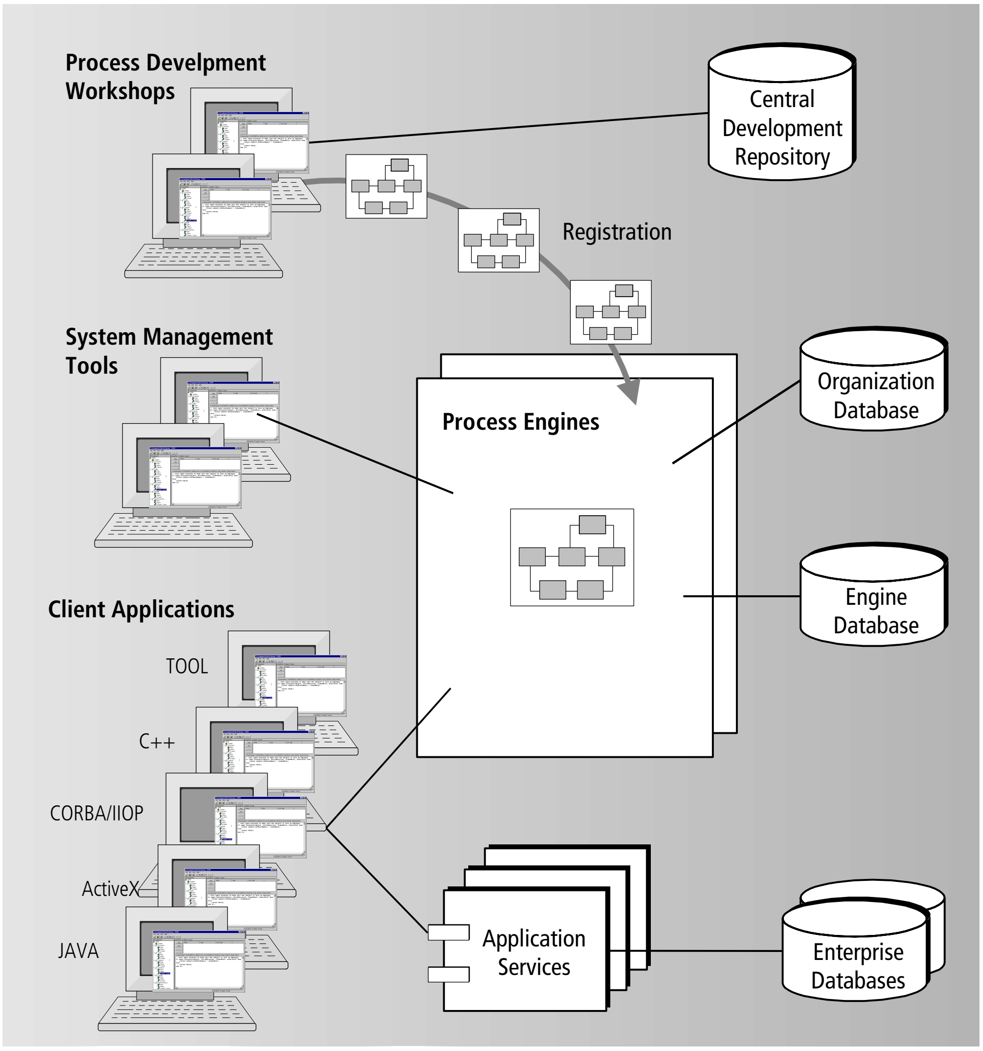
Client Applications and iIS Process Development

As described in the first chapter of the *iIS Process Development Guide*, an iIS process client application is written in a programming language corresponding to one of the process client APIs. This can be the iPlanet UDS scripting language TOOL, C++, a CORBA/IIOP-compliant language, JavaBeans, or a language capable of using ActiveX, such as Visual Basic or Visual C++.

An iIS process engine must be running on a server accessible to your client application (either in the current iPlanet UDS environment or in one connected to the current environment). The process definitions that your client application is designed to access must be registered on that engine.

Figure 1-2 shows how pieces of the iIS development system fit together. Process definitions are created and edited in the Process Development workshops and registered with a running process engine. These client applications communicate both with iIS process engines and with your shared application services.

Figure 1-2 Client Applications and the iIS Process Development System



Performing Operations with a Client API

A client application uses one of the process client APIs to communicate with the iIS process engine. Generally, a client application supports the following operations in the following order:

1. Establish a session with an engine.
2. For applications that provide a heads-up interface, maintain a list of activities offered by the engine to the session (a work list).
3. Select an item in a work list (heads-up interface) or take an item from the top of a queue in the engine (heads-down interface).
4. Invoke the correct applet or application to perform the activity—whether it be a word processor, a simple window, or a custom application using shared application services.
5. Inform the engine when work on an item has been completed by the end user.

In principle, these tasks are the same in any client application language. However, the details of how your application performs them is closely tied to your development environment. The remainder of this chapter presents an overview of some of these tasks, but does not cover them in the detail that the API chapters do. To see how to perform these tasks with a particular API, see the chapter on that API.

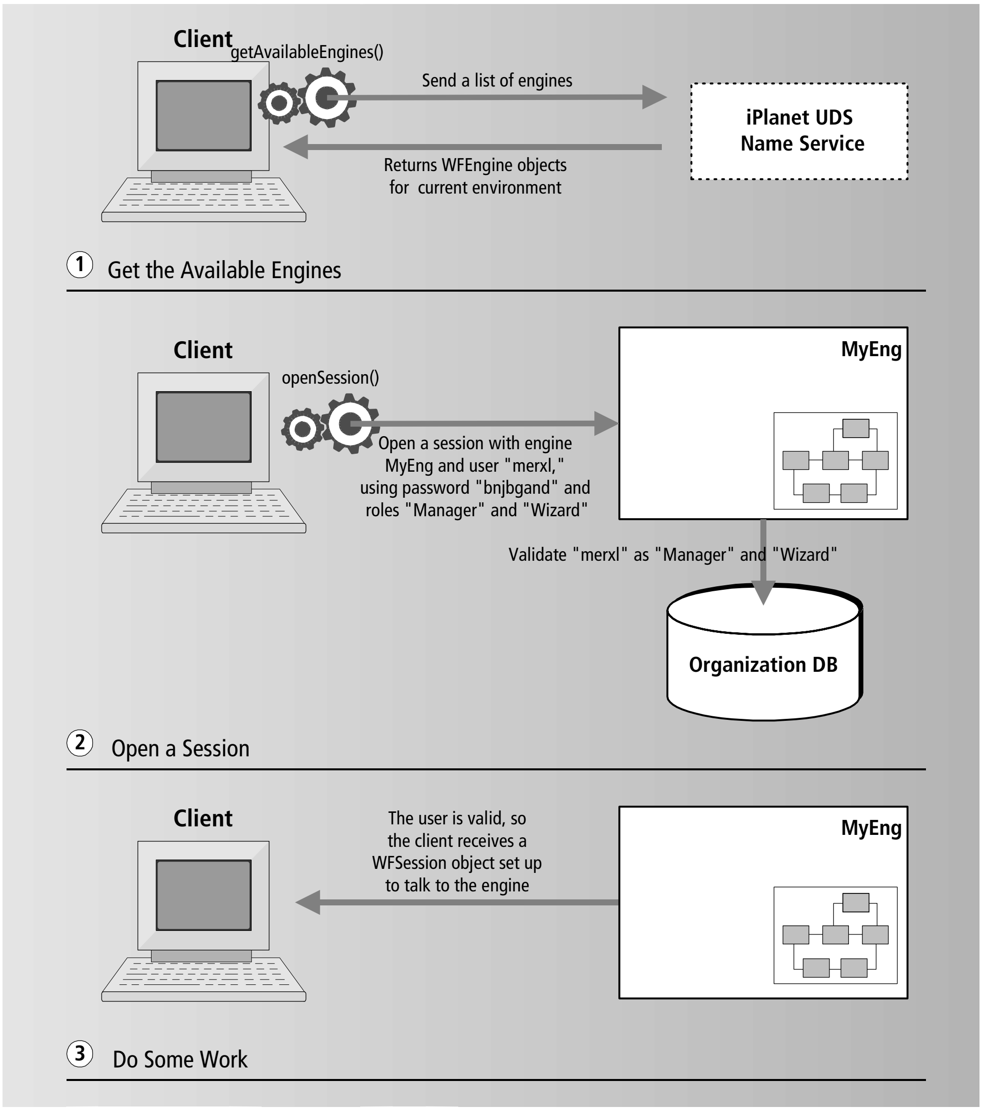
Establishing a Session With the Engine

The client application must establish a session with an engine before it can do any work with the engine. The client maintains at least one WFSession object for communicating with the engine.

To establish a session and begin displaying work to the user, the client must make a series of API calls that contact the iIS process engine. These services respond by passing objects back to the client that it can use to communicate with the engine (WFEngine and WFSession objects).

Figure 1-3 represents this series of calls and responses as a conversation between the client application and the engine.

Figure 1-3 Starting a Session


Client
getAvailableEngines()
Send a list of engines
iPlanet UDS Name Service
Returns WFEngine objects for current environment
1 Get the Available Engines
Client
openSession()
MyEng
Open a session with engine MyEng and user "merxl," using password "bnjbgand" and roles "Manager" and "Wizard"
Validate "merxl" as "Manager" and "Wizard"
Organization DB
2 Open a Session
Client
The user is valid, so the client receives a WFSession object set up to talk to the engine
MyEng
3 Do Some Work

Displaying a Work List and Choosing Work

After opening the session, the client session object must either get a list of activities for the current user to perform, or a single queued activity if the user is working heads-down.

When the client application gets a list of activities from the engine, it typically displays them to the user in a work list, from which the user can choose the next work item to perform.

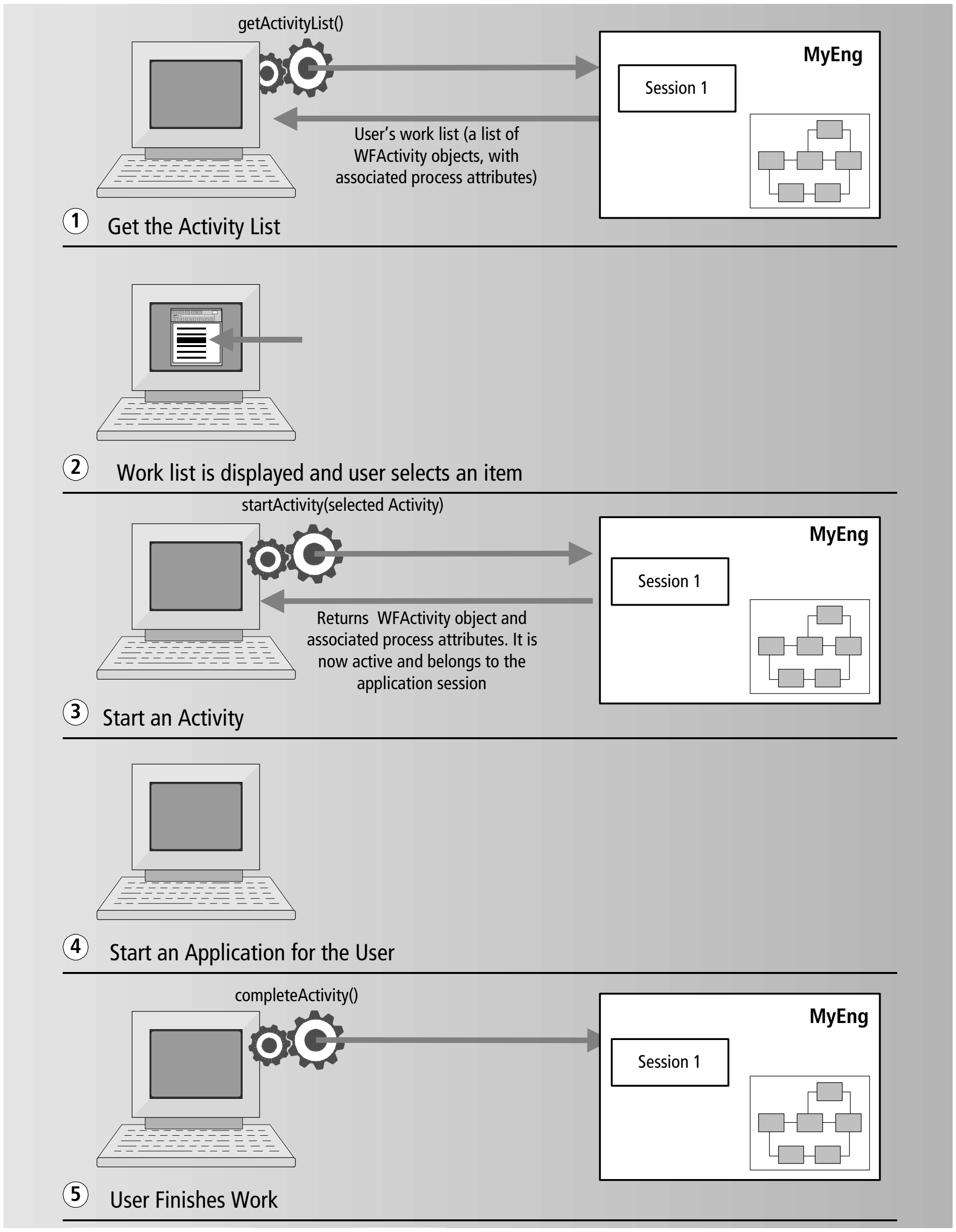
When the user chooses an item of work, the client application sends a request to the engine asking that the associated activity in the engine be made ACTIVE and that it be assigned to this session. The engine checks to see if the activity is still available, and, if it is, assigns it to this session. The client application can then start whatever applications the user needs to perform the work.

When the user is finished, the client application tells the engine that work on this activity is completed. The engine changes the state of the activity to COMPLETED, and changes the state of other activities in the process instance, according to the process definition.

The client application then starts over by updating the work list, if necessary. The user can then select another work item.

Figure 1-4 illustrates this series of actions.

Figure 1-4 Updating a Work List and Performing Work on an Activity


getActivityList()
MyEng
Session 1
User's work list (a list of WFActivity objects, with associated process attributes)
1 Get the Activity List
2 Work list is displayed and user selects an item
startActivity(selected Activity)
MyEng
Session 1
Returns WFActivity object and associated process attributes. It is now active and belongs to the application session
3 Start an Activity
4 Start an Application for the User
completeActivity()
MyEng
Session 1
5 User Finishes Work

Closing a Session

When the user is completely done, the client application closes the session with the engine and then does its own cleanup and shuts down.

Working with iIS Objects

This section provides conceptual information on some of the iIS objects your application must work with and covers the following topics:

- synchronizing client objects with engine objects
- process client two-phase commit
- iPlanet UDS environments and WFEngine objects
- process attributes and WFAttributeAccessor objects
- timers and WFTimer objects

For an API-specific walkthrough of how to start and end a session, how to handle work lists, how to start and end activities, and so on, see the chapter for the API you are using.

Synchronizing Client Objects with Engine Objects

The process engine returns objects of the following classes to client applications:

- WFActivity
- WFAttributeAccessor
- WFAttributeDesc (CorbaWFAttributeDesc for the CORBA/IIOP API)
- WFEngine (CorbaWFEngine for the CORBA/IIOP API)
- WFDeadlineTimer
- WFElapsedTimer
- WFProcess,
- WFSession (CorbaWFSession for the CORBA/IIOP API).

For more information on these classes, see the online help for the API you are using. Each class description has a *Using <Classname>* section, at the end of which is a note about client objects being simply copies of engine objects. There is also a table that shows which methods contact the engine and update the client object and which methods contact only a client object.

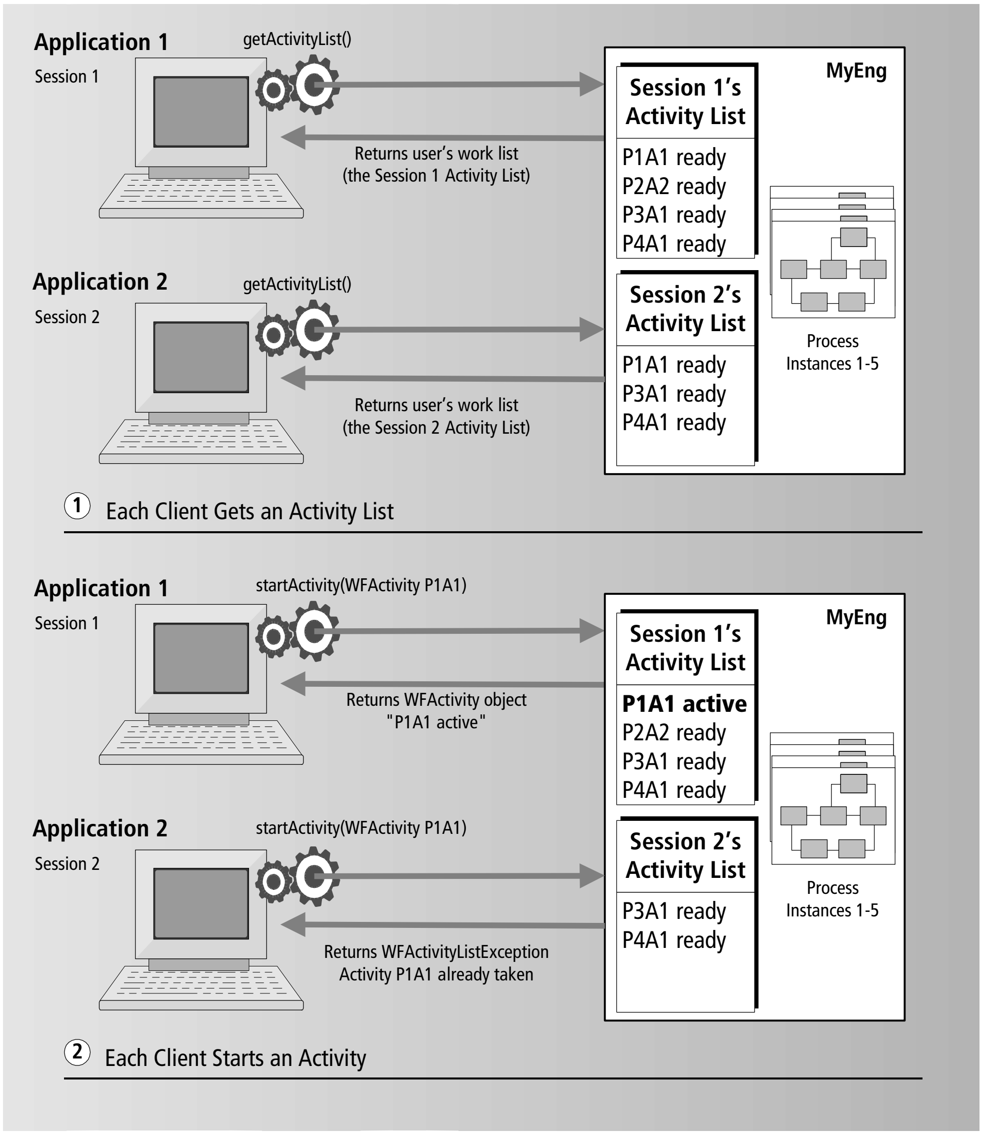
All these objects are maintained by the engine. The object passed to the client application is a representation of the engine object—it is not the engine object itself, nor is it a pointer to the engine object. This client-side object contains a copy of information about the engine object, such as the engine object's unique identifier, its state, possibly its value, and so on. The client version of the object also provides methods that give the client a way to work with the engine object.

For example, a client with a heads-up interface, Client2, calls the GetActivityList method of WFSession, which returns a list of WFActivity objects available to the user of the client application. In the engine, these objects are all activity instances that are part of process instances that are running in the engine. At the time the engine sends the list to the client, these activities are all in READY state, waiting to be performed.

Between the time Client2 displays an activity to the user on a work list and the time the client chooses it, the user of another client application, Client1, has already chosen the same activity, and that application now has it. (Client1 would have called the StartActivity method of WFActivity and gotten back the WFActivity object set to ACTIVE state.) The state of the engine's activity object has changed to ACTIVE, but the copy that Client2 has is still in READY state. It is only when Client2 calls StartActivity that it discovers that the WFActivity object is no longer available. (When event handling is used to update the activity list, it is likely that the first application would have received an event notifying it of this state change, but it is also possible that the two StartActivity calls were sent close together, and the first one got there just before the second one.)

Figure 1-5 shows this interaction between the two clients and the engine:

Figure 1-5 Client Objects as Copies of Engine Objects: WFActivity Example


Application 1
Session 1
getActivityList()
Returns user's work list
(the Session 1 Activity List)
Application 2
Session 2
getActivityList()
Returns user's work list
(the Session 2 Activity List)
MyEng
Session 1's Activity List
P1A1 ready
P2A2 ready
P3A1 ready
P4A1 ready
Session 2's Activity List
P1A1 ready
P3A1 ready
P4A1 ready
Process Instances 1-5
1 Each Client Gets an Activity List
Application 1
Session 1
startActivity(WFActivity P1A1)
Returns WFActivity object
"P1A1 active"
Application 2
Session 2
startActivity(WFActivity P1A1)
Returns WFActivityListException
Activity P1A1 already taken
MyEng
Session 1's Activity List
P1A1 active
P2A2 ready
P3A1 ready
P4A1 ready
Session 2's Activity List
P3A1 ready
P4A1 ready
Process Instances 1-5
2 Each Client Starts an Activity

Activity Lists

Figure 1-5 shows another engine feature that has an analog on the client: the activity list. The engine keeps a list of READY and ACTIVE activities for each heads-up client session, called an *activity list*. Each heads-up client session maintains a list of current work, called a *work list*. These lists can get out of synch if the client does not keep its objects up-to-date with the objects on the engine.

ActivityListUpdate Event

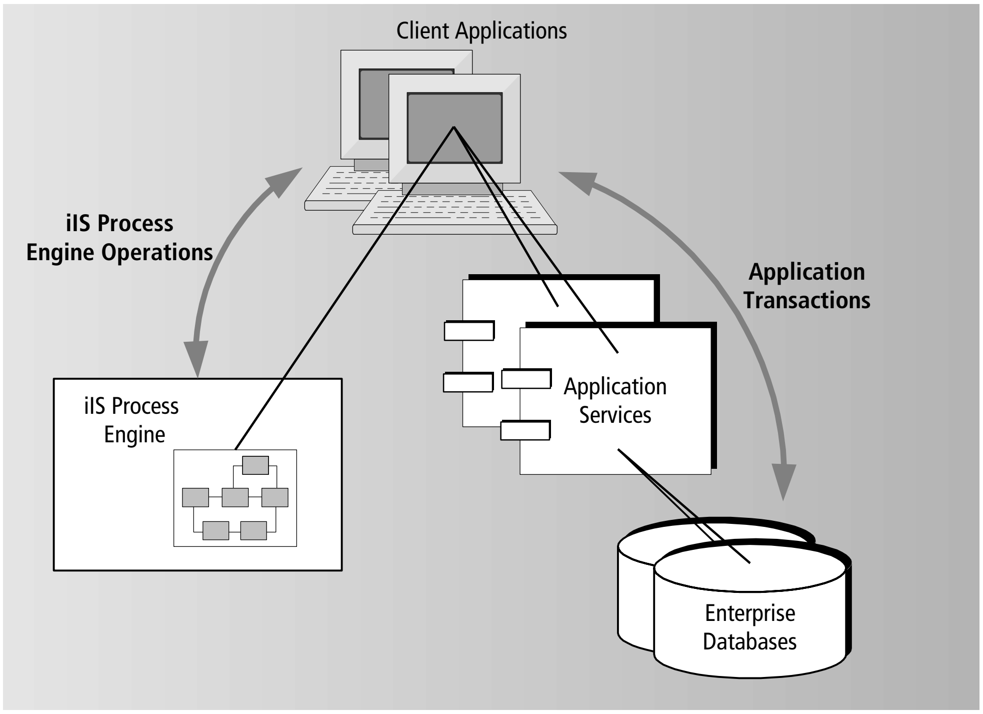
A client can synchronize with the engine using the WFSession object, which receives events from the engine notifying it when a change has been made that affects its work list. This event is called ActivityListUpdate.

Only TOOL client applications have direct access to iPlanet UDS events returned by the engine. Client applications written in any language can implement *activity update caching* to respond to iPlanet UDS ActivityListUpdate events. For details on activity update caching, see the chapter for the API you are using.

Two-Phase Commit

iIS provides a two-phase commit protocol that allows client applications to easily synchronize iIS process state changes with application database updates. As illustrated in Figure 1-6, client applications typically perform operations that change process state maintained by the engine, while also making application database updates.

Figure 1-6 Client Applications Change Both iIS Engine Process State and Application Data



For example, a client application typically interacts with the engine to start an activity, performs the activity by updating application data, and then interacts with the engine again to complete the activity. If for some reason, the update of application data fails, the CompleteActivity operation should not succeed. Or, similarly, if for some reason the CompleteActivity operation fails, the update of application data should not succeed.

iIS's two-phase commit protocol facilitates the synchronization of operations on the iIS process engine with client application transactions by letting you place engine operations in a transactional context. As with application database transactions, iIS's two-phase commit protocol lets you split the processing of iIS engine operations into two phases: the first phase—*preparing*—guarantees that the operation can either commit or roll back; the second phase actually performs the commit (or rollback).

NOTE "Rollback" in this usage is equivalent to "undo." For example, rolling back a CompleteActivity operation cancels the request to complete the activity.

Because of this two-phase commit capability, an iIS engine transaction and an application transaction can be combined into a higher level distributed transaction. The constituent transactions are placed in a PREPARE phase before either can commit. If either of the constituent transactions cannot be placed in a PREPARE phase, then the other is rolled back to its original state. In other words, the distributed transaction only commits if both constituent transactions commit, guaranteeing that application state and process state are synchronized.

iIS's two-phase commit protocol also facilitates synchronization between different activities—or sessions—in a given process instance. For example, if Client Application A (session A) has already prepared an AbortProcess for the process instance, and Client Application B (session B) tries to complete or abort an activity belonging to that process instance, an exception is raised on session B indicating that session A is going to abort the same process instance.

iIS's two-phase commit protocol makes it possible to include the following operations on a process engine in a transactional context:

- CreateProcess
- AbortProcess
- StartActivity
- CompleteActivity
- AbortActivity

When included in a transactional context, these operations are automatically placed in a PREPARE phase before being explicitly committed or rolled back. This allows them to be synchronized, as described above, with application update transactions.

The client application must ensure that all objects prepared for two-phase commit are resolved (committed or rolled back). In the event of a failure, the iIS engine may still have transactions in the prepared state. The system administrator should then check for and resolve these transactions, as explained in Chapter 6, "Managing Process Execution," of the *iIS Process System Guide*.

Two-phase commit can only be implemented from the TOOL API. Refer to Chapter 2, "Building a TOOL Client," for information on implementing two-phase commit transactions in client applications.

About iPlanet UDS Environments

Before you can start a session, you must obtain a WFEngine object that represents a currently running iIS process engine. In most cases you do not need to be concerned with the environment in which the engine is located. You simply call the GetAvailableEngines method (in TOOL, provided by the WFDirectoryService service object) and have it return a list of WFEngine objects representing the engines available to your client application. If the engine your client application needs to access is not on that list, either the engine has not been started, or it is running in an environment that is not connected to the current one. In the latter case, it might be helpful to know a little more about iPlanet UDS environments.

Environment Manager and Name Service

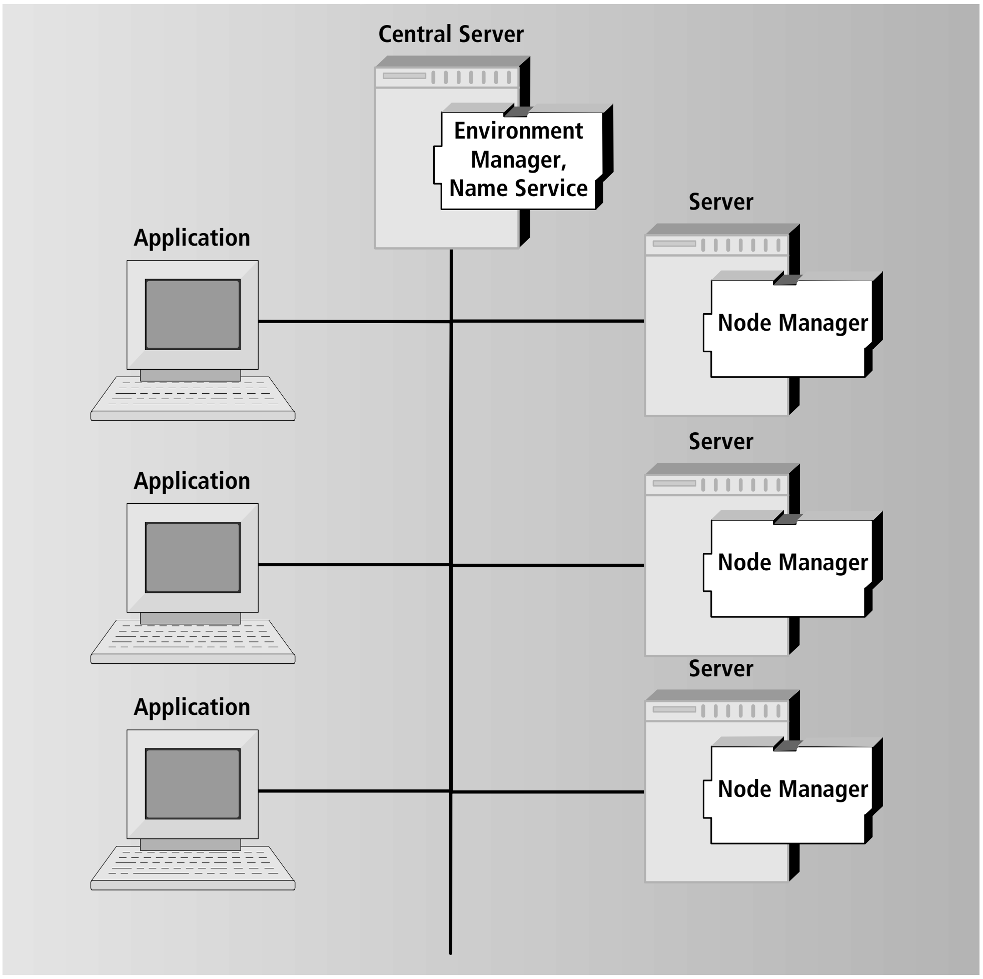
An iPlanet UDS environment is a collection of machines running iPlanet UDS processes that communicate with each other and with a common name service. You set up the environment and name service when you install iIS.

Any iPlanet UDS process using a specific name service address (contained in the FORTE_NS_ADDRESS environment variable) is said to participate in the iPlanet UDS environment covered by the name service at that address. The name service is part of the Environment Manager process (started with the nodemgr -e command), and there is a single Environment Manager per iPlanet UDS environment. By definition, the Environment Manager process runs on a machine designated as the *central server*.

Node Managers

In a single environment, you can also have other participating server machines, each running a node manager process (started with the nodemgr command). The Environment Manager process is a special kind of node manager that manages its own machine (the *central server* node), coordinates all the node managers on all the other server machines in the environment, and maintains the name service.

Figure 1-7 An iPlanet UDS Environment



Connected Environments

If you have the iPlanet UDS development environment, you (or your system administrator) can use the iPlanet UDS Environment Console to connect several environments and create a *global name space* in which applications and the Environment Manager in one environment know about named objects in other environments to which the Environment Manager is connected. Each Environment Manager manages the part of this global name space corresponding to its environment. Any iPlanet UDS partition (such as your client application) in these connected environments can access service objects (or iIS engines) in any part of the global name space. Each Environment Manager manages all named objects created by the servers directly connected to it.

For details on the relationship of iIS to iPlanet UDS environments, see the *iIS Process System Guide*. For more information on iPlanet UDS environments, see the *iPlanet UDS System Management Guide*.

Working with Process Attributes

Process attributes are defined in the Process Definition Workshop. They store information important to the process definition (for example, for making routing decisions), information that is stored nowhere else, or information that is needed often enough to require quick access (for example, a purchase order number).

In the Application Dictionary Workshop, you (or the process designer) can define an application dictionary entry that specifies a set of process attributes. Specifying these attributes allows the client application to automatically obtain an attribute accessor (an object of type WFAttributeAccessor for accessing process attributes, checking their values, and updating them) when it retrieves an activity. In the Process Definition Workshop, the process designer associates this application dictionary entry with an activity, making these attributes available to each session that is eligible to work on the activity.

For example, your client application gets a list of activity objects when it calls the GetActivityList method of WFSession (see "Getting and Handling Offered Activities" on page 61). Each activity whose application dictionary item has a list of process attributes specified comes back to the client with an attribute accessor for those attributes.

Client applications need process attributes for various reasons, such as:

- providing specific information about an activity instance on a work list, such as a purchase order number, in addition to the activity's general description
- updating information for the process instance when the user works on an activity
- picking up a database key value so a relevant database record can be accessed

Attribute Locks

More than one client can access a process attribute at one time. For example, in a process where two activities run in parallel, if the same process attribute is associated with both activities, two client applications could try to change the attribute at the same time. To avoid these kinds of conflicts, attributes can be locked for reading (allowing multiple clients to read the attribute's value, and preventing changes to the attribute) or writing (exclusively locking the attribute so a single client can change its value).

As mentioned earlier, in the process definition, the process designer can associate a set of attributes with an activity. At the same time, the process designer also designates a lock request setting for each of these attributes, which the engine applies when the client session starts the activity (calls StartActivity or StartQueuedActivity).

There are five types of lock requests you can make. The lock request is specified when you obtain the accessor. Lock requests are shown in the following table in TOOL syntax:

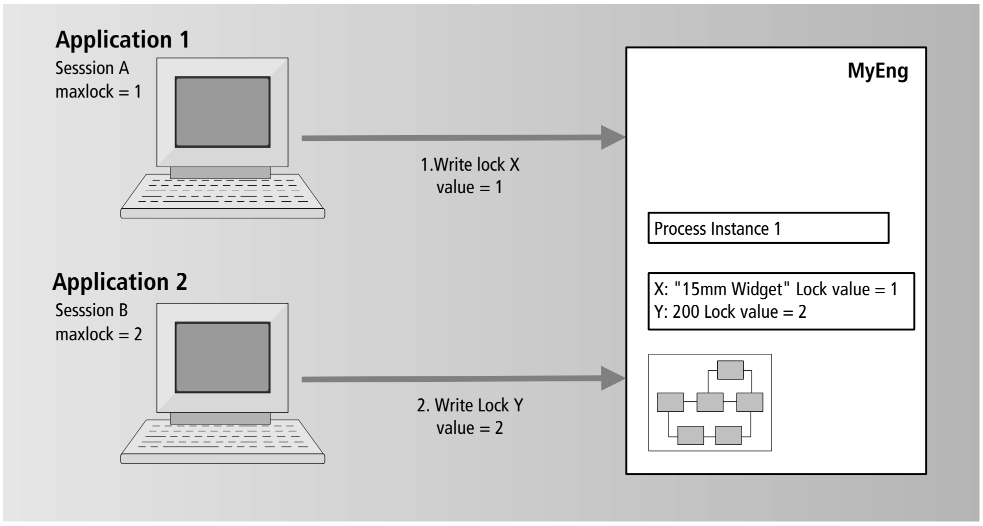
Lock Request	Description
WFAttributeDesc.NO_LOCK	The attribute is not locked, and a copy is made of its current value (ignoring any pending updates to its value), regardless of any locks that might currently be set on this attribute. The purpose is just to obtain its current value, with the understanding that the value might change at any time.
WFAttributeDesc.READ	The attribute is to be locked for reading, preventing other accessor objects from gaining a WRITE lock to it, but allowing other accessor objects to read it. If the attribute is currently WRITE locked, this request fails.
WFAttributeDesc.READQ	This request is the same as READ, except that the request is queued if the attribute is currently WRITE locked. (See "Understanding Deadlock Avoidance.")
WFAttributeDesc.WRITE	The attribute is to be locked in preparation for getting or changing its value or both. This lock prevents another accessor from obtaining a READ or WRITE lock (but an accessor using NO_LOCK can still read it). If the attribute currently has a either a READ or WRITE lock, this request fails.
WFAttributeDesc.WRITEQ	This request is the same as WRITE, except that the request is queued if the attribute is currently locked. (See "Understanding Deadlock Avoidance" below.)

Understanding Deadlock Avoidance

If the client application makes a queued lock request (READQ or WRITEQ), the engine raises an exception if it detects that adding this request to the queue might cause deadlock. This situation can occur if two sessions hold locks on process attributes of the same process instance, and one of those sessions requests a queued lock on attributes held by the other session. This situation does not necessarily lead to deadlock, but it might.

For example, a process instance has attributes X and Y. Session A has a write lock on attribute X and session B has a write lock on attribute Y. If session A makes a READQ lock request on attribute Y, there is no problem with deadlock, so the engine grants it and session A waits for session B to release the lock. However, if session B subsequently makes a READQ lock request on attribute X, if the engine allowed it, both sessions would be deadlocked waiting for the other one to relinquish a lock.

Figure 1-8 Avoiding Deadlock



As shown in the preceding diagram, the engine implements deadlock avoidance by assigning an ordinal number to each attribute as it is locked, increasing it by one for each subsequent attribute that is locked. The session that gets a lock receives the attribute's lock number, and the engine keeps track of the highest lock number assigned to each session. If a session attempts to lock an attribute with an ordinal number that is less than the session's current lock number, the lock request is denied.

In the previous example, when session A locks attribute X, X gets a lock value of 1 and A is assigned a maximum lock value of 1. When session B locks attribute Y, Y gets a lock value of 2 and B's maximum lock value becomes 2. When session A requests a READQ lock on attribute Y, because A's maximum lock value is 1 and Y's lock value is 2, A is granted the request and put in the read lock queue for Y. When session B requests a lock on attribute X, the engine denies the request because session B's maximum lock value is 2, but the lock value of X is 1.

If session A had not requested a lock on attribute Y, session B would still have been denied the lock request on X because its maximum lock value, 2, was greater than attribute X's lock value, 1. Therefore, there are situations in which a session could logically get a lock without causing deadlock, but the engine's deadlock avoidance scheme prevents the lock.

The other situation in which the engine will deny a lock request is if a session attempts to obtain a WRITE lock on an attribute on which it already holds a READ or WRITE lock.

The easiest way for a client application to avoid deadlock is for it to have only one attribute accessor open on a process instance at a time. If your application opens more than one attribute accessor on a process instance, you must ensure that if other sessions lock attributes from this process instance, then all the sessions either lock different attributes or they lock the same attributes in the same order.

NOTE Because a NO_LOCK request is always granted, regardless of the locks on an attribute, your client sessions can have as many accessors that use NO_LOCK as you like.

Getting an Attribute Accessor

When you work with activities, whether you are displaying them on the work list or starting an activity the user wants to work on, you often need to work as well with process attributes.

To work with process attributes, your client application must obtain an attribute accessor. The following list shows some common ways to get attribute accessors and indicates the lock requests that are applied in each case:

- When a heads-up client application is setting up its work list, it calls GetActivityList and, with each WFActivity object returned, gets an accessor with limited, NO_LOCK access to the attributes associated with the activity. The client must call the GetAttributeAccessor method of each WFActivity object to get its accessor.

  Regardless of the lock settings defined in the process definition, the engine gives the client NO_LOCK access to each of these attributes. (If the client application is resuming a suspended session in which it had one or more ACTIVE activities, it gets the same locks for those activities that it had when the session was suspended.)

- When the user is ready to work on an activity, a heads-up client application calls StartActivity, which by default returns an accessor for the activity's defined set of attributes (the ones specified in its application dictionary entry) and causes the engine to set the locks defined for the attributes.

  A heads-up application can *override* the default set of attributes defined for the activity by passing StartActivity an array of WFAttributeDesc in its attributeList parameter, indicating the lock request setting for each attribute. The engine passes back an attribute accessor for the designated list of attributes, first setting their locks accordingly.

- A client application can call the OpenAttributes method of an activity's parent process to get access to additional attributes. The activity must be on the session's activity list. In the attributeList parameter to this method, the client application specifies the list of attributes and their lock request settings, just as with StartActivity.

- When the user is ready to work on an activity, a heads-down client application calls the StartQueuedActivity method of WFSession, which by default causes the engine to set the locks defined for the attributes in its application dictionary entry. However, this method does not give the client an attribute accessor. The client application must call GetAttributeAccessor to get an accessor for the attributes.

As with StartActivity, a heads-down application can indicate which attributes it wants in the StartQueuedActivity call, passing it an array of WFAttributeDesc in a parameter that overrides its application dictionary entry's specified set of attributes and indicates the lock request setting for each attribute. The engine passes back an accessor for that list of attributes, first setting their locks accordingly.

- When a TOOL client application receives a WFSession.ActivityListUpdate event, the activity object is returned with an attribute accessor that can be obtained by calling the activity object's GetAttributeAccessor method. If the type parameter has the value WFSession.ACTIVITY_ADDED, the activity has just become READY, and the client application might need to add the activity to its work list and display some associated attribute values to the user. If the type parameter has the value WFSession.ATTRIBUTE_ACCESSOR_UPDATE, indicating that one or more process attributes associated with an activity on its activity list have changed value, the client application might need to change some associated attribute values that it displays to the user. In both cases, the client session calls the activity's GetAttributeAccessor method and gets NO_LOCK access to the associated attributes.

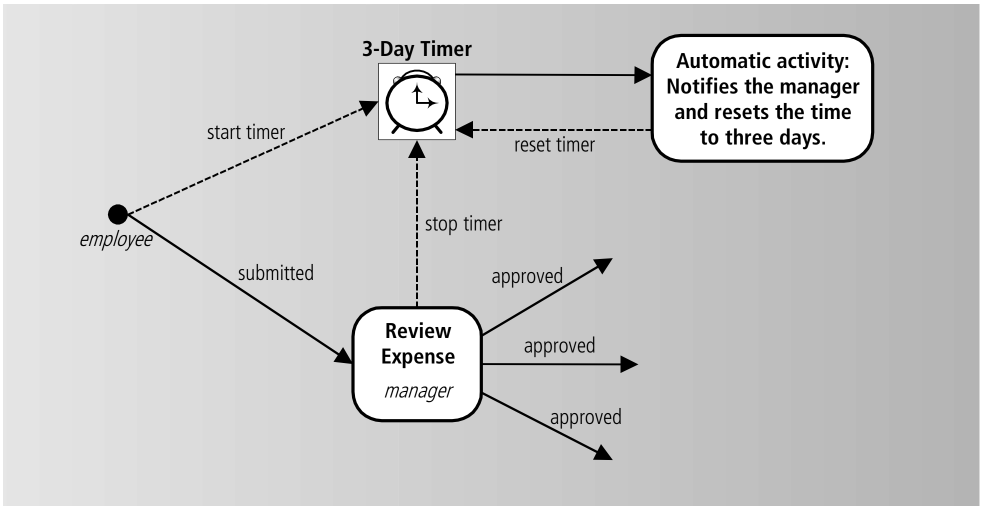
Working with Timers

Timers, which are defined in the Process Definition Workshop, have enough built-in logic to operate self sufficiently, usually without any intervention from the client application. While you should not need to use the timer API to access timers from the client application, the API is available if intervention is required.

A timer is an object that can be set for a period of time, like a kitchen timer (an *elapsed timer*), or to a date and time, like an alarm clock (a *deadline timer*). If the time period expires or the date and time is reached, the timer performs an appropriate action in the process instance, usually starting an activity to handle the timeout condition.

For example, in an insurance claims processing process, claims come in to a central dispatcher, who opens a new file for each claim and then starts a new process instance so the claim can be handled. When the process instance starts up, it both starts an elapsed timer and makes the first activity ready. The first activity in the process instance is for a claims processor to start initial processing on the claim, and there is a time limit of three days for this work to start. If the work does not start in three days, the timer expires and the dispatcher is notified to increase the priority of the claim and restart the timer.

Figure 1-9 An Elapsed Timer in a Process Definition



Getting Access to a Process Timer

A process timer is represented in the client application by either a WFDeadlineTimer object or a WFElapsedTimer object. Your application gets a representation of a timer object by calling the GetDeadlineTimer method or the GetElapsedTimer method of the timer's parent process instance. Your application can get the timer's WFProcess object by calling the GetProcess method of a WFActivity object it has received from the engine. Of course, you must choose a WFActivity object that is in the same process instance as the timer.

For example, in Figure 1-9, when the timer expires, the "Dispatcher raises priority" activity becomes READY, and a WFActivity object is sent to the client application that the dispatcher is running.

When the dispatcher's client application receives the WFActivity object, it goes through the following steps:

1. The client application displays in its work list the "Dispatcher raises priority" activity description that comes back in the activity's application dictionary entry.

2. The dispatcher chooses this activity in the work list, and the client application calls the StartActivity method of WFActivity to make the activity ACTIVE.

3. When StartActivity returns the WFActivity object (indicating that the activity is now ACTIVE and owned by this client session), the client application calls the GetProcess method of WFActivity to get the activity's (and the timer's) parent process.
4. With that parent process, the client application calls the GetElapsedTimer method of WFProcess to get the timer, indicating the name of the timer in the name parameter. (The client application programmer must know the name of the timer as defined in the process definition.)
5. When GetElapsedTimer returns the timer, the client application can then use the timer object to call the SetTimer method of WFElapsedTimer and reset the timer to a duration of three days.

In some situations, the process designer might be able to make efficient use of a feature of process execution called *timer links*, which allow changes to the timer state and value based on progression through the process. Timer links are set in the Process Definition Workshop, described in the *iIS Process Development Guide*.

Understanding the Process Client APIs

You can build an iIS client application using any of the process client APIs, as appropriate for your organization's system requirements, resources and standards. Each API provides the same services for managing engine sessions, activities, work lists, and so on, as explained in "Performing Operations with a Client API" on page 30. This section highlights basic considerations and features for each API.

TOOL

TOOL, the iPlanet UDS scripting language, was designed to support enterprise application development. As well as standard object-oriented features, TOOL supports event-driven processing, transactions, multitasking, exception handling and other enterprise functionality which makes it ideal for production systems requiring high performance and reliability. A TOOL client calls directly to the iIS process engine.

C++

The C++API interacts with the iPlanet UDS runtime system, which is installed on your client. C++ API calls to the iIS process engine are handled by an iPlanet UDS client partition. Specialized data types enable platform independence for your C++ application. Event handling is available for updating activity lists.

CORBA/IIOP

The CORBA/IIOP API is represented in Interface Definition Language (IDL) so that you can build a client application using Java or any CORBA/IIOP language. The CORBA/IIOP API interacts with the iPlanet UDS runtime system, which is installed on your client. This type of client communicates with the iIS process engine by means of ORBs (Object Request Brokers). This API includes some special data types used for process attributes. Event handling is available for updating activity lists.

JavaBeans

The JavaBeans API provides a set of non-visual JavaBeans to allow your Java application to interact with the iIS process engine. No iPlanet UDS installation is required, and data type translation is automatic. Event handling is available for updating activity lists.

ActiveX

The ActiveX API, based on Microsoft's COM and DCOM standard, can be used by various languages, such as C++ and Visual Basic. (It does not support VBScript or Visual Basic for Applications.) It provides a set of non-visual controls to allow your ActiveX application to interact with the iIS process engine. No iPlanet UDS installation is required, and data type translation is automatic. Event handling is available for updating activity lists.

Chapter 2

Building a TOOL Client

This chapter describes how to use the iPlanet UDS TOOL language and the iIS TOOL process client API to write a client application for an iIS enterprise system.

This chapter assumes that you understand TOOL programming concepts, as described in the *TOOL Reference Guide*, and that you know how to use the iPlanet UDS workshops, as described in *A Guide to the iPlanet UDS Workshops*.

The Client API

An iIS process client application uses the client application program interface (API) to communicate with the iIS process engine and execute processes registered with that engine. Some of the things a client application does with the API are:

- open a session with the engine
- get activities for the user, either as a work list (heads-up) or a single work item (heads-down) at a time
- process events posted about activity list changes and update the work list displayed to the user as necessary
- enhance the work list display for easier management of activities
- take control of an activity ("start" an activity) so the user can perform it
- set process attributes related to the work the user wants to do
- notify the engine when an activity is complete
- start instances of processes
- close sessions
- handle exceptions

This section explains how these tasks are accomplished using the TOOL client API, with sample TOOL code included.

Setting Up Your Project

You can create an iIS process client application in the iPlanet UDS Workshops.

When you create the project for your application, ensure that you have the iIS library WFClientLibrary in your repository, because it contains the client API needed by your client application project. If you are running iIS (as opposed to running iPlanet UDS without iIS), you should have this library in your repository.

If necessary, add WFClientLibrary to your workspace. (In the Repository Workshop, choose Plan > Include Public.) With this library in your workspace, add it as a supplier plan to your client application project (in the Project Workshop, choose File > Supplier Plans and pick WFClientLibrary from the list of available plans.) With this library as a supplier plan to your project, you can use its classes where appropriate in your client application.

In addition, you might want to include the process manager API if users of your client application will have the authority to abort arbitrary processes or activities. The name of this library is WFProcMgrLibrary. (For more information, see "Aborting an Activity" on page 73 and the iIS online help for the TOOL client API.)

If your client application uses the XMLData process attribute to supply XML documents to an iIS proxy, then you need to add the WFCustomIF library to your workspace, and then add it as a supplier plan to your client application project.

Opening a Session With the Engine

To perform useful work, your client must establish a session with an engine that is running in an iPlanet UDS environment. In your client application, you call either the FindEngine method or the GetAvailableEngines method of the WFDirectoryService service object to obtain a WFEngine object representing the engine with which you want to start a session. You then call this WFEngine object's OpenSession method, passing various parameters to establish your session.

Before opening obtaining an engine object, you need to determine the following information:

- the name of the engine that executes the process definitions your application will work with and whether the engine is running
- the environment where the engine is running if it is not the current, active environment or a connected environment

After obtaining an engine object, but before opening a session with the engine, you need to obtain the following information to pass to WFEngine.OpenSession:

- the user name and password of the user opening the session
- the name of the session
- how you intend to handle various conditions, like an unexpected termination of the session or the user suspending the session

When you have obtained the information you need and determined what you want to do with the session, you call WFEngine.OpenSession with the appropriate parameters.

Getting an Engine Object

You must either already know the name of the engine with which you want to open a session, or your application must be able to obtain it (for example, from a system variable).

In addition, the engine must be in the current environment or in an environment connected to the current one. (See "About iPlanet UDS Environments" on page 40 for more information.)

If you want to retrieve a single engine object, call WFDirectoryService.FindEngine. For example, the following TOOL code gets a WFEngine object for the engine named Claims in the current environment:

```
myEngine:WFEngine;
myEngine = WFDirectoryService.FindEngine(name='Claims');
```

If you want a list of all engines registered in the current environment, call the WFDirectoryService.GetAvailableEngines method. This method finds all the engines registered with the name service for the active environment. You can then find the engine you want from the array of engine objects this method returns. You find the engine by calling the GetName method of each engine object in the array and testing the return value. For example, the following TOOL code would find the engine named Claims in the list of engines either in the active environment or in environments connected to the active environment:

```
engineName : string = ‘Claims’;
myEngine : WFEngine;
engineList : Array of WFEngine;
engineList = WFDirectoryService.GetAvailableEngines( );
for engine in engineList do
  if engine.GetName = engineName then
    myEngine = engine;
    exit;
  end if;
end for;
```

NOTE Both methods contact the iPlanet UDS name service on the server. FindEngine verifies that the engine is registered. (If the engine is not registered, the method returns a NIL object.) GetAvailableEngines returns all registered engines.

It is possible that the engine your program chooses will not be running when it makes the subsequent WFEngine.OpenSession call, in which case the call will fail. However, your program can retry the call if you know that the engine will eventually be started.

Opening an Engine Session from Multiple Partitions

The WFClientLibrary in the TOOL API provides a user visible service object, WFDirectoryService, which is used in the following manner to locate an engine and open a session:

```
engine:WFEngine = WFDirectoryService.FindEngine('myengine');
session:WFSession = engine.OpenSession(....);
```

However, because WFDirectoryService is a *user visible* service object, the iPlanet UDS partitioning system places it in only one partition of an iPlanet UDS application. If you want to open a session with the engine from two partitions (say, from both the client and a server partition), then the partition that does not have WFDirectoryService assigned to it needs to explicitly create a WFDirectory object and use that object to create a session.

In other words, you write code to instantiate a local copy of the WFDirectory class and use that instance to create a session. For example:

```
localWFDir:WFDirectory = new;
engine:WFEngine = localWFDir.FindEngine('myengine');
session:WFSession = engine.OpenSession(....);
```

Setting Up User Profile Information

When you start a new session, you must supply user profile information about the user to the engine. To do so, you must have the name of a user profile that is registered with the engine, and you might need to know something about its structure if the user profile has been customized. (Some coordination with the person who creates user profiles in the User Profile Workshop is necessary. For information on the User Profile Workshop, see the *iIS Process Development Guide*.)

Validating the User

The engine uses the user profile name that comes in from your application to find a registered user profile that it can populate with your user information. When the engine has successfully created the user profile object and populated it, the engine passes this information to the site-defined Validation.ValidateUser method, which verifies it against the information about the user in the organizational database. (For more information, see the description of the Validation Workshop in the *iIS Process Development Guide*.)

The purpose of this validation is to ensure that the user is entitled to the roles and any other user attributes they specify in the client application, because the engine compares this information to the assignment rules of activities to determine which sessions get which activities. For example, if the user tries to sign on as a manager but does not have that role listed with their name in the organizational database, the engine detects the mismatch, and the call to OpenSession fails with an exception.

Supplying User Information

Before it passes a user profile name to the engine, the client application must determine the user name and password of the user, both of which are probably entered by the user when starting the client application. The client application can also prompt for any roles or other specially defined user information required to work with the activities this application will display, or it can let the ValidateUser method populate the user profile with this information from the organization database.

NOTE Because ValidateUser must be developed and registered before the client application is developed and it must work with all client applications, it is important to make a design decision early on as to how much information to send the engine.

When your client application has the necessary information from the user, you create a new WFUserProfileDesc object and fill in its attributes as necessary. For example, the following code creates the object *myUser* and initializes it with the user name *jfleuri*, the user profile name *AcmeUserProfile1* (for a user profile registered with the engine as AcmeUserProfile1), and the roles *Manager* and *Claims Processor*:

```
myUser:WFUserProfileDesc = new;
myUser.SetProfileName('AcmeUserProfile1');
myUser.SetUserName('jfleuri');
myUser.AddRole('Manager');
myUser.AddRole('Claims Processor');
```

NOTE The user name is case-sensitive.

Customizing a User Profile

The default definition of UserProfile includes only a name and a set of roles. However, if the user description is more complicated (for example, it indicates a specific level of budget approval authority), someone at your site must create a new definition of UserProfile in the User Profile Workshop, distribute the new user profile library, register it with the appropriate engines, and notify you so you can use the appropriate profile name and attributes in your WFUserProfileDesc object.

For example, users of a client application at Acme, Inc., in addition to roles, have a maximum budget authority, which is defined in AcmeUserProfile2 as the integer attribute *maxBudget*. The following code would initialize an object of this class for the user with user name *hgretel* in the role *manager* with a maximum budget authority of $5,000:

```
myUser:WFUserProfileDesc = new;
myUser.SetProfileName('AcmeUserProfile2');
myUser.SetUserName('hgretel');
myUser.AddRole('manager');
myUser.SetInteger('maxBudget',5000);
```

The Engine and the WFUserProfileDesc Object

When you pass a WFUserProfileDesc object as a parameter to WFEngine.OpenSession, the engine accesses the appropriate user profile definition and passes the object to the Validation.ValidateUser method that was defined in the Validation Workshop. This method is written by someone at your site to interpret those attributes correctly, and, like the user profile, is registered with the engine.

Suspended and Reconnected Sessions

If the session is interrupted by a network or engine failure (because the engine has not been set up to be fully fault-tolerant) or by intervention from a system administrator, you can have the engine either suspend the session (because the user might be in the middle of some work) or terminate it (thus losing whatever work might be in progress). Allowing a session to be suspended means that any work the user was doing can be resumed at a later date. Requiring that a session be terminated when it loses contact with the engine means that the work the user was doing will be lost, any process attributes that were locked will be unlocked, and the corresponding activity will be made READY again, available for other users to perform.

When a session is put in a SUSPENDED state, the following actions occur:

- Any activities that are currently ACTIVE are processed according to each activity's defined suspend action (or the overriding value specified in the suspendAction parameter of WFActivity.StartActivity or WFProcess.StartQueuedActivity). ACTIVE means that the activity was started by the client application's making a call to either WFActivity.StartActivity or WFSession.StartQueuedActivity.

  If the defined action in the process definition is *remove* (or the suspendAction parameter specified WFActivity.REMOVE), the activity is removed from the session's activity list. Depending on the specification in the process definition, the activity is either aborted or rolled back to the READY state (rolling back includes closing the activity's attribute accessor with ROLLBACK.) The default action is to abort the activity when the session is suspended.

  If the defined action is *retain* (or the parameter specified WFActivity.RETAIN), the activity is retained on the session's activity list in the ACTIVE state.

- Any activities that were on the session's activity list that were not chosen by the user are in READY state. The engine takes these activities off that session's activity list because the user can no longer choose any of them.

Disconnect Action Values

To determine whether a session is suspended or terminated if the connection to the engine is unexpectedly interrupted, set the controls parameter of OpenSession with one of the Disconnect Action values. The settings are WFEngine.SUSPEND_ON_DISCONNECT or WFEngine.TERMINATE_ON_DISCONNECT.

If you do not set one of these Disconnect Action values, the default action is to terminate the session if the WFSession object determines that the engine is disconnected and cannot be reconnected.

When you start a session, you can indicate if you want the engine to reconnect to a previously suspended session, if there was one, or to discard any previously suspended sessions and start fresh. If you reconnect to a suspended session, the user can gain access to work that was in progress when the session was suspended.

Reconnect Action Values

To determine whether a session reconnects to a suspended session or starts up as a new session, set the controls parameter of OpenSession with one of the Reconnect Action values. The settings are WFEngine.RECONNECT_ALLOWED or WFEngine.RECONNECT_PROHIBITED.

If you do not set one of these Reconnect Action values, the default action is to reconnect to a suspended session.

For more information, see "Handling the Connection With the Engine" on page 81.

Offered Activity Notifications

To save engine resources, a client opening a heads-down session should set the controls parameter of OpenSession with the Activity Offers value WFEngine.OFFERED_IGNORED. If this value is not set in the controls parameter, the engine by default will evaluate the session whenever an offered activity becomes READY.

Receiving ActivityListUpdate Events

Before a heads-up session displays a work list, it gets its initial list of activities by calling WFSession.GetActivityList. Thereafter, the engine notifies the session whenever there is a change to the list by sending it ActivityListUpdate events, because WFSession.GetActivityList by default instructs the engine to start sending ActivityListUpdate events to the session.

However, when the session first starts, there is often a lag between the time it starts and when it gets the list. During this lag time, the engine might start sending ActivityListUpdate events before the session even has a work list to update. To avoid this timing issue, by default an OpenSession call refuses ActivityListUpdate events. You can explicitly control reception of these events by setting the controls parameter of OpenSession with one of the ListUpdate Event values, WFEngine.AL_EVENTS_ON or WFEngine.AL_EVENTS_OFF.

For more information on ActivityListUpdate and maintaining work lists, see "Getting and Handling Offered Activities" on page 61.

Opening a Session

When you have determined the conditions for your client program's session with the engine, you can open the session by calling the WFEngine object's OpenSession method. The following code starts a new session with the engine Claims. It uses the sample code from the previous sections to set the parameters and starts up with the following session controls set:

- If the engine disconnects, the session is suspended.
- The session is allowed to reconnect to a suspended session.
- ActivityListUpdate events are not sent (the default).
- The session is evaluated whenever an offered activity becomes READY (the default).

Code Example 2-1 Opening a session (TOOL)

```
EstablishSession(NameForSession:string) : WFSession
-- Construct the user profile descriptor.
tmpUser : WFUserProfileDesc = new;
tmpUser.SetProfileName('UserProfile');
tmpUser.SetUserName(userName);

-- Check which 'SignOn' radio list item is selected.
case self.SignOn is
  when 1 do  -- Employee item selected
    tmpUser.AddRole('Employee');
  when 2 do  -- Reviewer item selected
    tmpUser.AddRole('Manager');
  when 3 do  -- Accountant item selected
    tmpUser.AddRole('Accountant');
  when 4 do  -- Auditor items selected
    tmpUser.AddRole('Auditor');
end case;

-- Open session.
tmpSession : WFSession;
-- The connection to the engine has been established in
-- the ConnectToEngine method in this class.
tmpSession = self.Engine.OpenSession (
          sessionName = NameForSession,
          user = tmpUser,
          userPassword = password,
          controls = WFEngine.RECONNECT_ALLOWED +
          WFEngine.SUSPEND_ON_DISCONNECT);
openStatus:integer = tmpSession.GetOpenStatus();
return (tmpSession);
```

Project: ExpenseReportClient • **Class:** LogonWindow • **Method:** EstablishSession

If the session is unable to start, the OpenSession call throws an exception. Otherwise, the session object is created and immediately usable.

To determine if the session reconnected to an existing suspended session or started as a new session, the last line of code calls GetOpenStatus. The next thing to do is to test if the returned value was WFSession.RECONNECTED_SESSION or WFSession.NEW_SESSION. If the session has reconnected to a previously suspended session, there might be one or more activities in the ACTIVE state that the client application must display to the user. If not, the session is a new one, and the client application can simply continue to get activities and display them.

Whether the session reconnected or is a new session, the client application must indicate what work is available to the user, as described in the next section.

Indicating What Work Is Available

Once the session has been established with the engine, the client application must determine what work is available to it. How it does this depends on whether the client application is a heads-up or a heads-down application.

- If the client application is a heads-up application, it is likely to display a work list to the user, as described next in "Getting and Handling Offered Activities."
- If the client application is a heads-down application, it retrieves the next queued activity available to it, as described in "Getting and Handling Queued Activities" on page 68.

NOTE The following sections have examples that use attribute accessors (WFAttributeAccessor) to work with process attributes. For an introduction to process attributes and a description of how WFAttributeAccessor objects can be obtained, see "Working with Process Attributes" on page 42.

Getting and Handling Offered Activities

For each session, the engine maintains a *session activity list*. This list has on it activities that are in READY state that the engine has offered to the session (because the session user profile matches an assignment rule for each of the activities). In addition, this list includes any activities that the session has made ACTIVE. A call to WFSession.GetActivityList returns this list of activities.

To get this list of activities from the engine, you declare an object to receive the return value of the method call, then call WFSession.GetActivityList, as shown in the following code sample:.

```
activityList:Array of WFActivity;
activityList = session.GetActivityList();
```

Calling this method automatically causes the engine to start sending ActivityListUpdate events to the session, allowing it to update its work list whenever there is a change to the session's activity list.

Building a Work List

Each activity object returned by GetActivityList has associated with it an application dictionary entry composed of an activity description, an application code, and, if the process designer defined them, a set of process attributes. You can construct a work list by extracting this information from each activity. Using the WFActivity methods GetName and GetActivityDescription, you can get the name and description of the activity.

It is likely that the work list you display for the user will include supplemental information for each entry (such as a customer name, purchase amount, order number, and so on), which you can obtain by calling each activity's GetAttributeAccessor method to obtain a WFAttributeAccessor. You can then use the attribute accessor's GetNames and GetValue methods to get the names and values of these process attributes.

NOTE If the return value of GetAttributeAccessor is NIL, there is no attribute accessor associated with the activity.

The following example shows a method that builds a work list display from an array of WFActivity. It uses objects such as ExpenseReportListView that are defined elsewhere and assumes that certain attributes, Status, ExpenseRptID, and Priority, are associated with each returned activity.

Code Example 2-2 Building a work list display (TOOL)

```
BuildList(session:WFSession)
-- Initialize listview field. This field will display the
worklist.
ExpenseReportListView = new;
ExpenseReportListView.IsFilled = TRUE;
ExpenseReportListView.IsFolder = TRUE;
ExpenseReportListView.IsOpened = TRUE;
node : ExpenseReportDisplayNode;

-- Get the activity list.
activityList : array of WFActivity =
        session.GetActivityList();

for activity in activityList do

  attribAccessor : WFAttributeAccessor =
    activity.GetAttributeAccessor();

  -- Get the values of the process attributes.
  statusNum : integer = attribAccessor.GetInteger('Status’);
  erid : integer = attribAccessor.GetInteger('ExpenseReportID');
  priorityVal : boolean = attribAccessor.GetBoolean('Priority');

  -- Use the expense report id attribute to get the expense report
  -- from the service object.
  eReport : ExpenseReport =
    ExpenseReportMgrSO.GetExpenseReport(erid);

  -- Supply values for this row in the worklist.
  node = new;
  node.ExpenseReportID = erid;
  node.DateSubmitted = new;
  node.DateSubmitted.SetValue(eReport.DateSubmitted);
  node.EmployeeName = eReport.EmployeeName;
  node.TotalAmount = eReport.TotalAmount;
  node.Priority = priorityVal;
  node.DVNodeText = self.GetStatusString(statusNum);
  node.WorkItem = activity;
  node.Parent = ExpenseReportListView;

end for;
```

Project: ExpenseReportClient • **Class:** ReviewerWindow • **Method:** BuildList

Maintaining a Work List

You can use one of the following two approaches for keeping a session work list display updated:

- poll the engine periodically
- respond to events posted by the engine

Polling the Engine

The polling approach involves periodically performing a GetActivityList request to obtain the entire activity list for the session. You can use this list either to completely refresh the display or you can merge the changed elements of the list with the corresponding displayed elements.

If your client application uses this technique, it should turn off reception of ActivityListUpdate events in every call to GetActivityList, including the first one, as shown in the following code sample:

```
worklist = session.GetActivityList(
               eventControl = WFSession.AL_EVENTS_OFF);
```

Responding to Events

If your application is written using the TOOL API, it has access to events returned by the engine. The one posted by the engine to your session object is ActivityListUpdate, which indicates a change to the session activity list and returns a WFActivity object. The recommended technique for updating the work list is to pick up the activity object from the event, then go through a series of method calls for the activity to get the new information about it.

NOTE You can call GetActivityList to update the entire list in response to an event, but this technique could result in your updating the work list twice for each changed activity. If your application has ActivityListUpdate events waiting in its queue when you call GetActivityList, the engine would already have posted the changes to the activities it returns to you. Your application would then update the work list with all the activities received from GetActivityList, and then process the events in the queue, unnecessarily updating the work list again for each event.

➤ **To use the returned WFActivity object to update a single work list item**

1. Call WFActivity.GetName to get the activity's name.
2. Call WFActivity.GetState to get its current state.
3. If you display attributes with the work item, call WFActivity.GetAttributeAccessor, then get their names with WFAttributeAccessor.GetNames, and pick up each attribute's value with WFAttributeAccessor.GetValue.
4. Index into the array in which you keep your work list and update each value of the appropriate work item.

NOTE Event handling of activity lists is also possible with the other APIs described in this manual. If you are using an API other than TOOL, see the section on maintaining a work list in the chapter for that API.

Displaying and Managing a Work List

You can enhance the client application work list display so that work items are removed when no longer needed, new items are added in their proper place, and other items are updated.

If you have implemented event handling, the session activity list is kept up-to-date and your application is informed when it needs to update the work list. The WFActivityListMgr class handles these tasks. To enhance the work list display you need to subclass WFActivityListMgr and implement the methods that perform display operations.

➤ To implement work list display operations

1. Create a subclass of WFActivityListMgr that provides implementations for the following methods, which are invoked by the WFActivityListMgr service methods when they detect that the display needs to be updated:

Status	Method	Action
Required methods	DisplayClear	Clear the work list from the display.
	DisplayNewItem	Use data from an activity object to put a new line at position in the work list, making room for it by moving the old line and all lines below it down one line.
	DisplayUpdateItem	Replace the old line in the work list indicated by the position parameter with information from its activity parameter.
	DisplayRemoveItem	Delete the line indicated by the position parameter and move the work list items below it up one line.
Optional methods	DisplayGroupStart	Start saving screen updates until the subsequent invocation of DisplayGroupEnd.
	DisplayGroupEnd	Display all screen updates saved since DisplayGroupStart was last invoked.
	RedundantUpdateEvent	Can be used when your program must perform special processing if the list manager ignores an update event (for example, because the event is a duplicate of one sent already).

2. Create a list manager object whose type is the subclass.

3. Invoke the Initialize method for the list manager object in the event loop for the relevant window after the session has been successfully opened.

   The Initialize method registers the list manager's event handler, invokes WFSession.GetActivityList to get the session's current activity list, and displays the work list. The method repeatedly invokes DisplayNewItem to display the entire work list.

You can control whether new items are inserted at the top of the work list (above line 1) or added to the end by setting the attribute ItemAddedPolicy to either WFActivityListMgr.ADD_AT_TOP or WFActivityListMgr.ADD_AT_BOTTOM. If you do not set this attribute, items are added at the bottom of the work list by default.

The attribute LastItem indicates the highest numbered position being managed (usually the last line in your window.)

NOTE See the Expense Report example application to get an idea of how to implement the display methods. This example application is described in detail in Appendix A of the *iIS Process Development Guide*.

NOTE WFActivityListMgr does not implement sorting. If the work list must be sorted (for example, ordering it according to expense report amount or purchase order priority), the client application must implement the sort. For a complete description of WFActivityListMgr, see the iIS online help.

Starting an Activity

In a heads-up application, when the user chooses a work item to work on, the client application calls WFActivity.StartActivity for the associated activity. Call this method synchronously (wait for it to return) to ensure that the client application has gotten the activity, because it is possible that another client application has simultaneously chosen it. In this case, the WFActivityListException exception will be raised by the engine. If the engine can change the activity's state to ACTIVE and give it to your session, it does so, and the method returns successfully. The engine also sends a WFSession.ActivityListUpdate event to all sessions that have this activity on their work lists, indicating that the activity is now ACTIVE.

The preferred way of working with activities is to specify their behavior in the process definition and not override these settings when you start them from a client application. (The less specific information about a process definition the client application must carry, the freer the process designer is to make design changes.) Therefore, it is usually best to call StartActivity without specifying parameters and pick up the default set of process attributes associated with the activity in its application dictionary entry.

If you must specify parameters, you can specify the suspendAction (what happens to the activity when the session is suspended), or the attributeList (attributes to be returned with the activity and the lock request for each), or both.

Getting and Handling Queued Activities

A heads-down application presents the user with the next item of work—it does not allow the user to choose from a work list as a heads-up client application does. This *next item of work* is represented in a process definition as a queued activity. To support heads-down applications, the engine maintains prioritized queues of queued activity instances. Each queue is composed of "like" activities, queued activities that have the same name and process definition. The highest priority activity is at the top of the queue, ready to be assigned to the next eligible session that asks for it.

To determine which sessions are eligible, the engine attaches the activities' assignment rules to the queue. A session requesting an activity in the queue must have a user profile matching the requirements of the queue's assignment rules. If so, the session gets access to the queue. (For more information on assignment rules, see the description in the *iIS Process Development Guide*.)

In a heads-down application, when the user is ready for the next item of work, you call your session object's StartQueuedActivity method. For example, the following code sample retrieves an activity called DoAccounting that is in the ExpRptActng process. If there is no activity in the queue, the method returns immediately:

```
curWorkItem:WFActivity;
curWorkItem = mySession.StartQueuedActivity(
        processName = 'ExpRptActng',
        activityName = 'DoAccounting',
        emptyAction = WFSession.IMMEDIATE);
```

This code sample does not use two parameters of the method, suspendAction and attributeList, for a reason: if you use these two parameters, they override behavior that should be defined in and controlled by the process definition.

NOTE Unlike GetActivityList, this method does not cause the engine to start sending the session ActivityListUpdate events, because a heads-down client has no session activity list.

If there is an activity on the queue, the engine retrieves the highest priority activity from the queue, marks it ACTIVE, adds it to the session's activity list, and returns a WFActivity object that describes the activity. (See "Dealing With An Empty Activity Queue" on page 70 for information on how to handle a queue with no activities in it.)

A WFAttributeAccessor object is built as part of the StartQueuedActivity operation if one is specified by the activity definition or if a list of WFAttributeDesc objects is provided in the method's attributeList parameter. If no process attributes are defined for the activity and none are specified in the StartQueuedActivity call, the attribute accessor returned is NIL. The following code retrieves the attribute accessor for an activity that has been returned to the client application:

```
curAttribs:WFAttributeAccessor;
curAttribs = curWorkItem.GetAttributeAccessor();
```

You can use the attribute accessor's GetName and GetValue methods to get the names and values of the attributes associated with the activity. Use GetInteger rather than GetValue if the data type of the attribute you want to access is of type integer.

The following sample code starts a queued activity and tests the return value to see if an activity was retrieved. If so there's an attribute accessor that can be used to obtain the associated process attribute information. (See the previous section for a code sample that gets attribute values from an accessor.)

Code Example 2-3 Starting a queued activity (TOOL)

```
GetWork(session:WFSession)
begin
-- Activity is an attribute in this class of type WFActivity
  self.Activity =
     session.StartQueuedActivity
        (processName = 'ERPD',
         activityName = 'DoAccounting',
         emptyAction = WFSession.IMMEDIATE);

  if self.Activity = NIL then
    self.Window.MessageDialog (
      'No expense reports available.');
  else
    accessor : WFAttributeAccessor =
      self.Activity.GetAttributeAccessor();
```

Code Example 2-3 Starting a queued activity (TOOL) *(Continued)*

```
    erID : integer = accessor.GetInteger('ExpenseReportID');

    eReport : ExpenseReport =
      ExpenseReportMgrSO.GetExpenseReport(erID);
    self.ExpenseReportGrid = eReport;
  end if;
exception
  when e : GenericException do
    self.Window.MessageDialog ((ErrorDesc)(e).Message);
end;
```

Project: ExpenseReportClient • **Class:** AccountantWindow • **Method:** GetWork

NOTE The code fragment above is from the Expense Reporting example, which is described in the Appendix to the *iIS Process Development Guide*.

Dealing With An Empty Activity Queue

If there is no activity on the queue, your client application can either wait until one is available or return immediately and try again later. The previous code sample returns immediately. If it had specified an emptyAction of WFSession.WAIT, your client application would wait until a DoAccounting activity instance was added to the queue.

You can control how long your client session waits by starting a separate timed task to handle the call to StartQueuedActivity. If the method call does not return before the timeout, your client application can call the CancelStartQueuedActivity method to cancel the StartQueuedActivity call. In this case, StartQueuedActivity returns with a NIL object.

For example, the following code sample shows (in an overly simplistic way) how you might set a timer in an event queue to time out at five minutes and cancel an asynchronous task that calls StartQueuedActivity. The user calls StartTimedQueuedActivity, which uses the helper method StartTheQueuedActivity to do the work while monitoring the time:

Code Example 2-4 Cancelling a task (TOOL)

```
method StartTimedQueuedActivity(
            session : WFSession,
            processName : string,
            queueName : string) : WFActivity;
begin
  waitTimer : Timer(IsActive = FALSE);
  timedOutBefore : boolean;
  event loop
    postregister
      start task StartTheQueuedActivity(session,
            processName,
            queueName);
      waitTimer.TickInterval = 300000;  // Wait five minutes
      timer.IsActive = TRUE;

    when ActivityStarted(activity) do
      timer.IsActive = FALSE;
      return activity;
    when waitTimer.Tick() do
      timer.IsActive = FALSE;
      if session.CancelStartQueuedActivity(processName,
                 queueName) then
        // successful cancel
        return NIL;
      end if;
      // Cancel attempt made after engine actually started
      // (assuming that nothing really weird happened.)
      if timedOutBefore then
        // We’ve waited twice as long as expected
        return NIL;
      end if;
      timedOutBefore = TRUE;

      // We’ll give it one more timeout period
      timer.IsActive = TRUE;
    end event;
  timer.IsActive = FALSE;
  return;
end method;

method StartTheQueuedActivity(session : WFSession,
        processName : string,
        queueName : string)
begin
  activity : WFActivity;
```

Code Example 2-4 Cancelling a task (TOOL) *(Continued)*

```
  activity = session.StartQueuedActivity(processName,
             queueName,
             WFSession.RETAIN,
             WFSession.WAIT);
  post ActivityStarted(activity);
  return;
end method;
```

Multitasking Your StartQueuedActivity Calls

If your heads-down client needs to check several activity queues, you can use the multitasking feature in iPlanet UDS. For each different kind of queued activity you want your client to get work from, start a task. For each WFActivity object that is returned, call its GetAppCode method to differentiate it from the other queued activities you have requested. See the iIS online help for details on this method.

You can either have each task block on the StartQueuedActivity call or have each task periodically poll for work in an event loop. Then you can use events to notify the parent task that a piece of work has been found and needs to be worked on.

Completing Work On an Activity

When the user has finished working on the work item, set any process attributes that need setting with WFAttributeAccessor.SetXXX methods (for example, SetInteger), then call WFActivity.CompleteActivity to tell the engine you are done. The engine commits all changes and the process continues as indicated by the process definition.

Rolling Back an Activity

If work was not completed and the user wants to quit without applying updates, you can call WFActivity.RollbackActivity to roll back changes the user has made and reset the activity's state to READY, making it available to other users. What the engine rolls back are changes to process attributes associated with the activity. Any changes that your client application has made to site files and any transactions that it has started with site databases must be explicitly undone or rolled back by your client application.

Another option is to call WFActivity.AbortActivity to abnormally terminate the activity. Do not exercise this option unless something drastic has happened and you want to risk aborting the process. Unless the process definition explicitly handles aborting activities, it will abort the entire process. Aborting an activity is described in the following section, "Aborting an Activity".

Aborting an Activity

As described in "Completing Work On an Activity" on page 72, you can allow the user to abort the current activity from the client application if the user wants to back out the work done so far. (It is more likely that you will allow the user to roll back an activity rather than aborting it, since aborting the activity can also abort the entire process.)

You can also use the Process Management API to allow a user other than the one who typically performs the activity to abort an activity. Because aborting an activity is a drastic measure that can have widespread consequences, you should limit this capability to users with the appropriate authority, such as a manager who is responsible for a process. For details on the Process Management API, see the iIS online help.

A user might want to abort an activity for a variety of reasons. One possibility is that the person working on the activity is out of town and the process instance cannot continue until the activity is aborted. It might be reasonable to abort the activity, let it abort the process instance, and start the entire process again. If you do not want the process to abort, you must define an abort router for the activity, probably one that does not let the activity abort, but makes it READY again so another user can work on it. (Abort routers are defined in the Process Definition Workshop. See the *iIS Process Development Guide* for more information.)

Before using this API, you must include the library WFProcMgrLibrary in your application. Choose Plan > Include Public, then in the Include Public Plan dialog box, select WFProcMgrLibrary and click OK. If this library is not on the list, you must install iIS.

Your user can abort an activity only if it appears on the client application's activity list.

In general, to support aborting an activity, your code should:

- instantiate a WFPMgrProcess object and an array of WFPMgrActivity objects
- initialize the WFPMgrProcess object with the process object of the activity you want to abort
- obtain the READY and ACTIVE activities of that process object
- find the activity to be aborted on that list, test its state, and abort it if it is in the state you expect it to be in
- if the activity is not on the list (because it is not in READY or ACTIVE state), abort the process instead (optional, depending on the purpose of the abort)

For example, assume that you just received an ActivityListUpdate event, that the WFActivity object passed is called *act*, and that foundAct is not NIL, meaning the instance of activity *VerifyClaim* has been found. In the following example, the management task is to abort the activity:

Code Example 2-5 Aborting an activity (TOOL)

```
proc : WFPMgrProcess = new;
allActs : Array of WFPMgrActivity;

-- Initialize process management objcet and get activity list
proc.Initialize(act.GetProcess());
allActs = proc.GetActivities();

foundAct : WFActivity;
for a in allActs do

    -- look for activity named VerifyClaim
    if a.GetName() = ’VerifyClaim’ then
      foundAct = a;
      exit;
    end if;
end for;

-- Abort process if activity not found
if foundAct = NIL then
  proc.Abort();
else

  -- Abort associated activity only if it’s in ACTIVE state
  foundAct.AbortActivity(WFActivity.ACTIVE,
    WFActivity.NO_ROUTING);
end if;
```

Implementing Two-Phase Commit

As discussed in "Two-Phase Commit" on page 38, a client application can use iIS's two-phase commit protocol to synchronize application database updates with iIS process state changes. This protocol allows the following operations to participate in a two-phase commit transaction:

- WFActivity.AbortActivity
- WFActivity.CompleteActivity
- WFActivity.StartActivity
- WFPMgrProcess.AbortProcess
- WFSession.CreateProcess

NOTE This capability is supported only by the TOOL client API. For a given session, only one two-phase commit operation can take place at a time.

How the Two-Phase Protocol Works

The two-phase commit protocol is implemented by placing an iIS process engine session in two-phase commit mode. The engine uses a unique *transaction ID* to monitor operations during a two-phase transaction. You can specify your own transaction ID. Otherwise, the engine generates a transaction ID (based on a time stamp, session name, and the session operation sequence number).

NOTE The transaction ID can be useful for the client application (or the iIS system administrator) to track transactions.

Use either of the following to place a session in two-phase commit mode:

WFSession.OpenSession When opening the session; the engine generates a transaction ID consisting of a time stamp, session name, and a session operation sequence number.

WFSession.SetControl When explicitly passing a unique transaction ID to an existing session.

For example:

```
mySession = self.Engine.OpenSession (
      sessionName = NameForSession,
      user = someUser,
      userPassword = somePassword,
      controls = WFEngine.RECONNECT_ALLOWED +
                WFEngine.TWO_PHASE_COMMIT_ON);
```

When two-phase commit mode is enabled for a session, if a client application invokes any of the iIS engine operations that can participate in two-phase commit, the operation is automatically placed in a PREPARE phase. At this point the engine guarantees that it can either commit or abort the operation. For example, any StartActivity or CompleteActivity operation is automatically placed in a PREPARE phase.

All iIS objects associated with a prepared transaction are unavailable for further operations until the client application explicitly commits or aborts the transaction.

When a client application has placed a session in two-phase commit mode, it typically does the following:

- prepares any application (database update) transactions it needs to make.
- invokes an iIS engine operation that can participate in a two-phase commit transaction (for example, CompleteActivity)
- commits or otherwise resolves the application transaction.
- commits or rolls back the iIS engine operation.
    - If both the iIS engine operation and application transaction are successfully placed in a PREPARE phase, then each is explicitly committed.
    - If either the iIS engine operation or application transaction is not successfully placed in a PREPARE phase, the other is rolled back.

For example:

Code Example 2-6 Two-phase commit (TOOL)

```
CurrentActivity.StartActivity(WFActivity.RETAIN);

-- Enable two-phase commit
mySession.SetControl (mySession.TWO_PHASE_COMMIT_ON);

--Prepare client DB transactions here
. . .

-- Prepare the iIS engine
CurrentActivity.CompleteActivity;

-- Resolve client DB transactions here
-- (in this example, commit)
. . .

--Resolve iIS operation
mySession.Commit();

-- Disable two-phase commit (This call is optional)
mySession.SetControl (MySession.TWO_PHASE_COMMIT_OFF);
```

Creating a Process Instance

To start a new instance of a process definition, you call WFSession.CreateProcess, indicating the name of the process definition in the processName parameter. The user must be in a role that can create a new instance of the process for this method to succeed. (In other words, the process definition must have an assignment rule that gives a user with the same role as the user of this client application permission to create a new instance of the process.)

If you also want to initialize process attributes when you start the process instance, you can put the list of process attributes and their values in an array of WFAttributeDesc and pass this array in the attributeValues parameter of CreateProcess.

NOTE When the engine gets a request to create a new process instance, it ignores the LockType attribute of any WFAttributeDesc objects in the attributeValues parameter. For that reason, you need to set only the Name and Value attributes of each WFAttributeDesc object in the array.

The following method illustrates how to build an array of WFAttributeDesc:

```
BuildAttributeDesc() : Array of WFAttributeDesc
tmpArray : Array of WFAttributeDesc = new;

tmpArray[1] = new(Name = 'ExpenseReportID',
  Value = IntegerData(
      value = ExpenseReportGrid.ExpenseReportID));
tmpArray[2] = new (Name = 'TotalAmount',
    Value = DoubleData(
      value = ExpenseReportGrid.TotalAmount));
tmpArray[3] = new (Name = 'Status',
    Value = IntegerData(
      value = ER_SUBMITTED));

return tmpArray;
```

Project: ExpenseReportClient • **Class:** ExpenseReportWindow • **Method:** BuildAttributeDesc

The method below illustrates how to create a process instance, using the method shown above to supply the array of attributes:

```
Display(session:WFSession)
...
    -- Set attributes and create process instance.
    ERProcess = session.CreateProcess(
        processName = 'ERPD',
        attributeValues = BuildAttributeDesc());

    self.Window.MessageDialog('Expense Report submitted.');
...
```

Project: ExpenseReportClient • **Class:** ExpenseReportWindow • **Method:** Display

NOTE The preceding code fragments are from the Expense Reporting example, which is described in the appendix to the *iIS Process Development Guide*.

Setting Process Recovery Level

By default all current process state information is recovered for a process in the event of engine or system failure. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. If appropriate, the recovery level can be altered so that the process instance is simply restarted with no recovery of process state information; another alternative you have is specifying no recovery of the process.

The default behavior of full recovery applies under the following conditions:

- no other value is specified for the process recovery level in the Process Definition property inspector
- you do not override the default in the client application code
- the engine is configured for state recovery

If logging options for the engine are turned off, the process definition default does not override the engine setting. For details on setting the recovery level in the Process Definition Workshop, see the *iIS Process Development Guide*. For details on engine configuration, see the *iIS Process System Guide*.

You can use the recoveryInfo parameter of the CreateProcess method to control whether or not the recovery level value specified in the process definition is used when creating the process. This method returns a WFProcess object for each newly created process instance. If either no value or the value WFProcess.RCVR_NORMAL is passed in for recoveryInfo, the process is created using the recovery level specified in the process definition. If another value is passed in, that value overrides the recovery level value specified in the Process Definition property inspector. The possible other values are WFProcess.RCVR_FULL, WFProcess.RCVR_NONE, or WFProcess.RCVR_PROCESS_ONLY.

Closing a Session

When the user is ready to log off, your client application must close the session with the engine. You call the WFSession.CloseSession method and, depending on whether or not the user has completed work, indicate with the newState parameter that the session is to be either terminated or suspended.

Terminating the Session

If the user has completed all work and the engine has successfully closed any associated activities, you can terminate the session by calling CloseSession with the newState parameter set to WFSession.TERMINATED. Terminating the session causes the engine to remove all READY activities from the session's activity list.

The following code fragment illustrates an appropriate time to close a session in a Display method:

```
Display(session:WFSession)
...
self.Open();
event loop
  BuildList(session);
  when ...
end event;
self.Close();
session.CloseSession(WFSession.TERMINATED);
```

Project: ExpenseReportClient • **Class:** EmployeeWindow • **Method:** Display

NOTE The preceding code fragment is from the Expense Reporting example, which is described in the appendix to the *iIS Process Development Guide*.

Another effect of terminating the session is that the engine aborts any ACTIVE activities. If the user has any work in progress, it is represented by at least one ACTIVE activity, and all the work is lost at this point. If the user indicates that the session is to be terminated and there is an ACTIVE activity, it would be good to let them know the consequences and suggest suspending the session or finishing all work as alternatives.

Suspending the Session

If the user has any work in progress, you typically would take whatever steps are necessary to save the current work and then suspend the session by calling CloseSession with the newState parameter set to WFSession.SUSPENDED. Suspending the session causes the engine to remove all READY activities from the session's activity list, but leave any ACTIVE activities on the suspended session's activity list. When the user logs in at a later time, you can open the new session and tell the engine to reconnect to the suspended session by indicating WFEngine.RECONNECT_ALLOWED in WFSession.OpenSession's controls parameter.

Handling the Connection With the Engine

As described in "Suspended and Reconnected Sessions" on page 57, when the client application starts a session (by calling WFSession.OpenSession), it is connected to an engine through the client application's WFSession object. If you call this object's SetPingInterval method, it monitors the client application's connection to the engine in the background, sending ping messages to ensure that the connection is still up, and handling reconnection if the connection goes down (either the network connection is lost or the engine fails). By default, pinging is off and the default interval is set to 2 seconds. You can turn pinging on and change the interval by calling WFSession.SetPingInterval or WFSession.SetRetry. For details on these methods, see the iIS online help.

Losing the Connection

If pinging is on and the connection to the engine is lost, the client detects the disconnection and posts an EngineDisconnected event.

If pinging is off, the WFSession object does not test the connection with pings. However, if the client calls a method that contacts the engine and the engine is down, the session object detects the disconnection and posts an EngineDisconnected event.

When the session object posts an EngineDisconnected event, it automatically attempts to reestablish the connection. If it is able to do so within the parameters specified with the WFSession.SetRetry method (by default, 20 retries at intervals of 2 seconds), the session object posts an EngineReconnected event and the session is reestablished.

If the session object is unable to reestablish the connection within this time period (for example, the engine has failed and there is no backup engine that can be automatically restored, or the engine takes an especially long time to come back up), it raises a WFException with LostContact as the reason. At this point, the client application must disconnect and reestablish the session later.

It is possible that the connection with the engine could go down and come back up again, and your client application would never be notified because the user is working solely on the client side, performing the work associated with an activity. What happens in this instance is that your client application's WFSession object detects that the engine connection has gone away because the engine did not respond to one of its periodic ping messages. The WFSession object then begins to try to restore the connection and does so successfully before it receives any method calls that connect with the engine.

It is also possible that the connection with the engine could go down, but not come back up again before the client application makes a call that connects to the engine. In this case, when the client session makes the method call (for example, WFActivity.StartActivity) WFSession sends a WFSession.EngineDisconnected event to the client application, indicating that it has detected that contact has been lost with the engine, but it is trying to reestablish the connection. One way for your client application to handle the event is to wait for an EngineReconnected event and make the call again. (For a description of these events, see the iIS online help.)

What does the engine do in these circumstances? It maintains its own internal representation of each session and tracks the activities offered to or being used by each session. It handles disconnections as follows:

- If the engine goes down and then recovers, after it recovers, it marks its session objects SUSPENDED until the clients' session objects reestablish connection. Any activities in an ACTIVE state remain that way and remain connected to their sessions. Any attributes with locks remain locked.
- If the engine does not go down, but it loses contact with a session object (for example, the network node for that session goes down, or the client application crashes), the engine marks its corresponding session object either SUSPENDED or TERMINATED, depending on the setting of the autoDisconnect parameter to the WFSession.OpenSession method. (See "Suspended and Reconnected Sessions" on page 57. That section also describes how active and offered activities are handled.)

Reestablishing the Connection

If the engine goes down and your client application does not make any method calls that contact the engine during the period that it is down, you do not need to do anything: the WFSession object reestablishes the connection on its own as long as the engine is not disconnected for longer than the timeout period.

If the duration of the disconnection exceeds the timeout period, the next time your client application makes a method call contacting the engine, it receives a LostContact exception. At that point, you must terminate the client application and reestablish the connection later, when you start up again.

NOTE If you indicated TERMINATED in the autoDisconnect parameter to WFSession.OpenSession (the session is to be terminated if connection is lost with the engine), the session is terminated only if the engine loses contact with your session. If the engine goes down and recovers, the session is always suspended when the engine comes back up, regardless of the autoDisconnect setting.

As with starting a new session, the client calls WFEngine.OpenSession to restart a suspended session. The controls parameter include the value WFSession.RECONNECT_ALLOWED. When the session is open, the next thing the client does depends on whether it is a heads-up or a heads-down application.

For a heads-up application, on the first call to GetActivityList, if there are any ACTIVE activities in the returned list and the return value of OpenSession was WFSession.SUSPENDED_SESSION, the user was working on these activities previously when the session was suspended. The client application can test the current values of attributes and prompt the user for any information needed to continue work on the activities.

Normal behavior for a heads-down client application after it starts a completely new session is to do a StartQueuedActivity call to get its first activity. However, when reconnecting to a suspended session, a heads-down client must first call WFSession.GetActivityList to determine if it already has an ACTIVE activity left over from when its session was suspended. If there is an ACTIVE activity on the list, the client application can deal with it in the same manner as a heads-up client does (see previous paragraph).

If you do not want to reestablish connection with the suspended session, you set the reconnectAction parameter to WFSession.RECONNECT_PROHIBITED. The engine terminates the suspended session, aborting any ACTIVE activities, and starts a new session for your client application.

Recovering Attribute Accessors

After you reconnect a session to the engine, you must call WFSession.GetAtrributeAccessors to see which accessors your recovered session has open, if any. This method returns an array of attribute accessors. You can call the WFAttributeAccessor methods GetName, GetID, and GetProcessID to determine which attributes belong to which process instances.

Handling Exceptions

iIS reports client API exceptions with three exception classes: UsageException, WFException, and WFActivityListException.

UsageException

UsageException is the standard iPlanet UDS exception class defined in the Framework Library online Help. UsageException is used in iIS to indicate incorrect use of an API.

WFException

WFException is defined in WFClientLibrary and is the generic exception class for the iIS process engine. It is a subclass of Framework.GenericException. As such, the Message virtual attribute is defined and refers to a TextData object that describes the error.

All WFException objects are preceded on the error stack by a stack dump to help the user locate the programming problem that led to the exception.

Sometimes informational entries are left on top of the stack to help the user identify the source of the error. The information provided is usually the name of the activity and the process that was the subject of the session’s request. If so, the exception that is of interest is raised, and an information entry is added to the top of the stack.

WFActivityListException

The WFActivityListException exception class is a subclass of WFException that is generated only during WFActivity.StartActivity processing. It tells the calling code that the request was rejected because the session activity list maintained by the engine did not contain the activity. Since this event is a subclass of WFException, the message text and stack trace are available as described for WFException.

The most likely cause is that some other session has already activated a READY activity that used to be on the session's activity list. This situation can arise because the requesting session has not yet had an opportunity to process the relevant WFSession.ActivityListUpdate event that the engine posted on the session object.

The normal action for the client application to take in response to this event is to inform the user that the selected activity is not available and repaint the display with the activity removed. The client must be prepared to ignore the related WFSession.ActivityListUpdate event that is (or will be) waiting in the session object's event queue. Some clients might prefer to defer the display repaint until this event actually appears.

This exception can also be raised if the client attempts to start a WFActivity object that is stale. For example, the client may have processed the related ActivityListUpdate event, but somehow retained an old (now out-of-date) WFActivity object. The engine cannot differentiate this situation from the expected (client is slightly behind in event processing) circumstances. Hence the two are combined into one event.

The following code sample illustrates an event handler that registers for the ActivityListUpdate event. After the activity is started, it checks for exceptions generated when the activity is no longer on the activity list maintained by the engine:

```
Display (erDisplayNode:ExpenseReportDisplayNode, userName:string)

-- Get handle for current activity
activity : WFActivity = erDisplayNode.WorkItem;
begin
  activity.StartActivity(WFActivity.REMOVE);
exception
  when e : WFActivityListException do
    self.Window.MessageDialog (
       'The activity you selected is no longer available.');
    task.ErrorMgr.Clear();
    return;
end;
```

Project: ExpenseReportClient • **Class:** ReviewExpenseWindow • **Method:** Display

NOTE The preceding code fragment is from the Expense Reporting example, which is described in the appendix to the *iIS Process Development Guide*.

Chapter 3

Building a C++ Client

This chapter describes how to use the iIS C++ process client API to write a client application for an iIS enterprise system. It assumes that you understand C++ programming concepts.

This chapter will be more useful to you if you have read the first chapter of the *iIS Process Development Guide*, which provides an overview of the iIS process management system. It is also helpful to read this manual's Chapter 1, "Building an iIS Process Client Application" which describes what a client application is and provides overviews of starting and ending sessions, retrieving activity lists and starting activities, and retrieving and updating client objects, process attribute accessors, and timers.

This chapter starts with an explanation of concepts helpful in understanding how to use C++ with iIS and describes tasks you need to perform to set up your system and perform C++ call-in to the iIS process engine. (For detailed information on using C++ call-ins, see the iPlanet UDS manual *Integrating with External Systems*.)

After this general information, the chapter continues with a description of how to use the C++ client API with iIS that covers typical tasks your process client application is likely to perform, such as opening a session and getting a list of available activities.

Using the C++ API with the iPlanet UDS Runtime System

Because the iIS process engine is developed using iPlanet UDS and uses the iPlanet UDS runtime system, the C++ API must interact with that runtime system. Your C++ client application cannot call directly into the iIS engine, as an iPlanet UDS TOOL application can. Instead, the C++ API calls must be handled by an iPlanet UDS client partition, a process running on your client platform that makes the calls to the engine.

The C++ API classes you use in your client application have a reference to an underlying iIS class (have a *handle* for the iIS class) in the iIS client partition.

In your client application, the first thing you must do is start the iPlanet UDS client partition (described in detail in "Starting the Client Partition" on page 95). The part of the client partition that your program accesses is called a *task*, and the C++ object you create to reference it is called a *task handle*. To start the partition, you use the global function WFStartup, which returns the task handle created when the partition is started.

When you first instantiate an iIS object in your C++ program, you must pass the class's constructor the handle to the iPlanet UDS task that was returned by the call to WFStartup. The constructor uses this task handle to ensure that the corresponding iPlanet UDS object is created in the iPlanet UDS partition and that your C++ handle object is hooked up to this iPlanet UDS object. From then on, you can use the C++ object normally. (All the C++ API class constructors require a task handle. Not using one causes a compilation error.)

When your iIS C++ object goes out of scope, the iPlanet UDS runtime system manages the iPlanet UDS object. When there are no other handles for this object, the iPlanet UDS runtime system deallocates the object and reclaims the memory it used.

When your C++ client application has finished using iIS, you can use the WFShutdown global function to close the client partition.

Defining iIS C++ API Objects

A handle class requires that there be an underlying iIS object to pass its method calls to the iIS process engine. Therefore, when you declare an object that is an instance of the handle class (a *handle object*), you must pass its class constructor the task handle returned in the call to WFStartup. The following example shows how to create a WFActivity object in your client application. It uses the task handle cndTask created in the code sample in "Starting the Client Partition" on page 95:

```
WFActivity MyActivity (cndTask);
```

If you have already created an iIS object for a handle object, you can assign another handle object to the existing one, as shown in the following code sample:

```
WFActivity MyActivityRef (cndTask);
MyActivityRef = MyActivity;
```

Understanding the C++ API Data Types

In the C++ API, some of the parameters and return values use specialized data types starting with wf_. The purpose of these data types is to provide platform independence for your C++ client application.

The following table shows the C++ and iPlanet UDS TOOL equivalents of these data types. (TOOL data types are described in the *TOOL Reference Guide* and in iIS online help, and are the types used in process definitions.)

Type in C++ API	C++ equivalent	TOOL type
wf_bool	int	boolean
wf_i4	long int (4-byte integer)	i4
wf_ui4	unsigned long (4-byte integer)	ui4

Handling iIS Exceptions

The iIS client API provides the following macros, which you can use in your C++ client application to catch client API exceptions:

qqhTRY(*task*) starts a try block

qqhCATCH(*class*, *var*) starts a catch block, which catches exceptions of the specified handle class and uses the var parameter as an instance of that class

qqhELSE_CATCH(*class*, *var*) statement in a catch block that catches exceptions of the specified handle class

qqhELSE statement in a catch block that catches unexpected exceptions of the specified handle class

qqhELSE_ONLY catches all iIS exceptions without being part of a catch block

qqhEND_TRY ends a try block

The following code shows the general syntax of the try and catch statements:

```
qqhTRY(task){
  ... code interacting with Conductor;
} qqhCATCH(class, var) {
  ... code handling the exception of the specified class
} qqhELSE_CATCH(class, var) {
  ... code handling the exception of the specified class;
} qqhELSE_CATCH(class, var) {
  ... code handling the exception of the specified class;
... any other qqhELSE_CATCH statements;
} qqhELSE {
  ... code handling any other Conductor or runtime exceptions;
} qqhEND_TRY;
```

You can use the qqhELSE_ONLY macro to catch any iIS exceptions without first defining a catch block, as shown in the following code sample:

```
qqhTRY(task) {
  ... code interacting with Conductor;
} qqhELSE_ONLY {
  ... code handling any raised Conductor exceptions;
qqhEND_TRY;
```

The following example shows how you could use the qqhTRY, qqhCATCH, and qqhELSE_CATCH macros to catch iIS exceptions. After the activity is started, it checks for exceptions generated when the activity is no longer on the activity list maintained by the engine:

```
Display (erDisplayNode, userName) {
  WFActivity activity (cndTask);

  // Get chosen activity from work list
  activity = erDisplayNode.WorkItem;
  qqhTRY(cndTask) {
    activity.StartActivity(WFActivity.REMOVE);
  } qqhCatch(WFActivityListException, activityNotFound) {
      printf('The activity you selected is no longer
available.');
  } qqhElseCatch(WFException, unknownExcept) {
      printf
(unknownExcept.GetMessage(cndTask).AsCharPtr(cndTask);
  }
  qqhEND_TRY;
}
```

Handling iIS and C++ Exceptions

You can nest qqhTRY blocks within C++ TRY blocks; however, you cannot overlap the qqhTRY blocks and TRY blocks, and you cannot nest C++ TRY blocks within qqhTRY blocks.

Setting up Your System and Compiler to Use the C++ API

To ensure that all the correct files can be located by the compiler and linker when you compile and build your C++ client application, you must set up some paths.

NOTE In the following paths, FORTE_ROOT refers to the directory where you installed iIS. On a Windows NT system, for example, the default is c:\forte.

➤ To set up your system and compiler

1. Set your INCLUDE path to specify the directories containing the shared library file for the C++ API and the header files for iIS runtime and library classes (conductr.h is located in FORTE_ROOT\install\inc\conductr):

   - FORTE_ROOT\install\inc\cmn
   - FORTE_ROOT\install\inc\ds
   - FORTE_ROOT\install\inc\handles
   - FORTE_ROOT\install\inc\os
   - FORTE_ROOT\install\inc\conductr
   - FORTE_ROOT\install\inc\conductr\ofcustom
   - FORTE_ROOT\install\inc\conductr\wfclient

2. Using an environment variable or the make file, specify the libraries needed for linking. The library file extensions will vary depending on your platform.

   On Windows NT, Windows 2000, and Windows 95\98, the LIBPATH contains a list of .lib files. You need to specify the following paths:

   - FORTE_ROOT\userapp\wfclient\cl0 \wfclient.lib
   - FORTE_ROOT\userapp\ofcustom\cl0\ofcustom.lib
   - FORTE_ROOT\userapp\wfclien1\cl0\wfclie0.lib
   - FORTE_ROOT\install\lib\qqcm.lib
   - FORTE_ROOT\install\lib\qqdo.lib
   - FORTE_ROOT\install\lib\qqfo.lib
   - FORTE_ROOT\install\lib\qqhd.lib
   - FORTE_ROOT\install\lib\qqkn.lib
   - FORTE_ROOT\install\lib\qqsh.lib
   - FORTE_ROOT\install\lib\qqsm.lib

On UNIX platforms, the path that specifies libraries for linking contains a list of shared library files. You need to include the following shared libraries (where *xxx* stands for the file extension, depending on the platform, as described in the table following the list):

- FORTE_ROOT/userapp/wfclient/cl0/wfclient.*xxx*
- FORTE_ROOT/userapp/ofcustom/cl0/ofcustom.*xxx*
- FORTE_ROOT\userapp\wfclien1\cl0\wfclie0.*xxx*
- FORTE_ROOT/install/lib/qqcm.*xxx*
- FORTE_ROOT/install/lib/qqdo.*xxx*
- FORTE_ROOT/install/lib/qqfo.*xxx*
- FORTE_ROOT/install/lib/qqhd.*xxx*
- FORTE_ROOT/install/lib/qqkn.*xxx* (named qqknpthrd on some UNIX platforms)
- FORTE_ROOT/install/lib/qqsh.*xxx* (not on some UNIX platforms)
- FORTE_ROOT/install/lib/qqsm.*xxx*
- The following table describes the file extensions for each platform:

Platform	**Shared Library Extension**
Alpha OpenVMS	.exe
Alpha OSF/1	.so
Compaq VAX OpenVMS	.exe
DG Intel DG/UX	.so
HP/UX	.sl
IBM AIX	.a
OS/390	.so
RS/6000 AIX	.a
Sequent DYNIX/ptx	.so
Siemens/Nixdorf Reliant UNIX	.so
Sun Solaris	.so
VAX OpenVMS	.exe

The Client API

A client application uses the client application program interface (API) to communicate with the engine and execute processes registered with that engine. Some of the things a client application does with the API are the following:

- start an iIS process client partition
- open a session with the engine
- display activities (work items) to the user, either as a work list (heads up) or a single work item (heads down)
- update the work list displayed to the user as necessary
- take control of an activity ("start" an activity) so the user can perform it
- set process attributes related to the work the user wants to do
- notify the engine when the work is done
- create instances of processes
- close sessions
- handle exceptions

This section explains how these tasks are accomplished using the iIS process client C++ API, with sample C++ code included.

Using the C++ API with the Client Partition

If your C++ client application needs to perform any start-up tasks independently of iIS, it would be faster to perform them prior to starting the iIS client partition. When it is time to interact with the iIS process engine, start the client partition (with WFStartup), then interact with iIS through the C++ API. When your client application has finished with iIS, stop the client partition (with WFShutdown).

Because the resources required to start up and shut down a client partition can be considerable, it is recommended that you start the client partition once and leave it running as long as your C++ client application needs it.

Starting the Client Partition

For your client program to be able to use the iIS client API, you must include the conductr.h header file.

Before your client application can work with iIS, it must start an iIS client partition that runs in the same process as your C++ client application. First you define an instance of WFTaskHandle, the handle class for a task handle, to receive the handle returned from the start-up function. The next step is to use the global function WFStartup to start the client partition and return a handle to the main task accessible in the client partition. See the iIS online help for details on this function.

The following example shows how to define a task handle and start the client partition:

```
#include <conductr.h>
...
int main(int argc, char** argv)
{
WFTaskHandle cndTask;
  printf(
    "Starting the C++ client for the Expense Reporting
process.\n");
  cndTask = WFStartup();
  ...
}
```

Closing the Client Partition

When your iIS process client application has finished using the iIS engine, you can use the WFShutdown global function to close your iIS client partition and automatically dereference the iIS task, releasing the memory for the client partition. See the iIS online help for details on this function.

The following sample function declaration uses the global task handle cnd

Task (obtained on the call to WFStartup) to shut down the client partition:

```
int StopClientPartition()
{
//. . . Perform clean up functions.
  WFShutdown(cndTask);
}
```

Opening a Session With the Engine

To perform useful work, your client must establish a session with an engine that is running in an iPlanet UDS environment. In your client application, you call either the WFFindEngine global function or the WFGetAvailableEngines global function to obtain a WFEngine object representing the engine you want to start a session with. You then call this WFEngine object's OpenSession method, passing various parameters to establish your session. See the iIS online help for more details.

Before opening obtaining an engine object, you need to determine the following information:

- the name of the engine that executes the process definitions your application will work with and whether the engine is running.
- the environment where the engine is running if it is not the current, active environment or a connected environment

After obtaining an engine object, but before opening a session with the engine, you need to obtain the following information to pass to WFEngine::OpenSession:

- the user name and password of the user opening the session
- the name of the session
- how you intend to handle various conditions, like an unexpected termination of the session or the user suspending the session

When you have obtained the information you need and determined what you want to do with the session, you call WFEngine::OpenSession with the appropriate parameters.

Getting an Engine Object

You must either already know the name of the engine with which you want to open a session, or your application must be able to obtain it (for example, from a system variable).

In addition, the engine must be in the current environment or in an environment connected to the current one. (See "About iPlanet UDS Environments" on page 40 for more information.)

If you want to retrieve a single engine object, call the WFFindEngine global function with the appropriate parameters. For example, the following sample code gets a WFEngine object for the engine named Claims in the current environment:

```
WFEngine myEngine (cndTask);
myEngine = WFFindEngine('Claims');
```

If you want a list of all the engines registered in the current environment, call the WFGetAvailableEngines global function. This function finds all the engines registered with the name service for the active environment. You can then find the engine you want from the array of engine objects this method returns. You find the engine by calling the GetName method of each engine object in the array and testing the return value. For example, the following sample code would find the engine named *Claims* in the list of engines either in the active environment or in environments connected to the active environment. The code tests to see if an engine object was returned. If so, it tests whether the engine object name is "Accounting". If so, there's a match and the counter is set to a value that ends the for loop. If an engine is NIL, it's the first one in the array and there's no engine available:

```
WFEngineArray availEngines (cndTask);
WFEngine myEngine (cndTask);
char* engineName = "Accounting";
int engineCount;
availEngines = WFGetAvailableEngines();
engineCount = availEngines.Items();
for (i=0;i<(int)engineCount;i++){
   if (!availEngines[i].IsNil()){
      if (strcmp(availEngines[i].GetName(),engineName) == 0){
          myEngine = availEngines[i];
          i = (int)engineCount;
      } // end if GetName
  } // end if !IsNil
  else {
      printf('No engine available');
      i = (int)engineCount;
  } // end else
} // end for loop
```

NOTE Both functions contact the iPlanet UDS name service on the server. WFFindEngine verifies that the engine is registered. (If the engine is not registered, the method returns a NIL object.) WFGetAvailableEngines returns all registered engines.

It is possible that the engine your program chooses will not be running when it makes the subsequent WFEngine::OpenSession call, in which case the call will fail. However, your program can retry the call if you know that the engine will eventually be started.

Setting Up User Profile Information

When you start a new session, you must supply user profile information about the user to the engine. Before you can do so, you must have the name of a user profile that is registered with the engine, and you might need to know something about its structure if the user profile has been customized. (Some coordination with the person who creates user profiles in the User Profile Workshop is necessary. For information on the User Profile Workshop, see the *iIS Process Development Guide*.)

Validating the User

The engine uses the user profile name that comes in from your application to find a registered user profile that it can populate with your user information. When the engine has successfully created the user profile object and populated it, the engine passes this information to the site-defined Validation.ValidateUser method, which verifies it against the information about the user in the organizational database. (For more information, see the description of the Validation Workshop in the *iIS Process Development Guide*.)

The purpose of this validation is to ensure that the user is entitled to the roles and any other user attributes they specify in the client application, because the engine compares this information to the assignment rules of activities to determine which sessions get which activities. For example, if the user tries to sign on as a manager but does not have that role listed with their name in the organizational database, the engine detects the mismatch, and the call to OpenSession fails with an exception.

Supplying User Information

Before it passes a user profile name to the engine, the client application must determine the user name and password of the user, both of which are probably entered by the user when starting the client application. The client application can also prompt for any roles or other specially defined user information required to work with the activities this application will display, or it can let the ValidateUser method populate the user profile with this information from the organization database. Because ValidateUser must be developed and registered before the client application is developed and it must work with all client applications, how much information to send the engine is a design decision that must be agreed upon in advance.

When your client application has the necessary information from the user, you create a new WFUserProfileDesc object and fill in its attributes as necessary. For example, the following code creates the object *myUser* and initializes it with the user name *jfleuri*, the user profile name *AcmeUserProfile1* (for a user profile registered with the engine as AcmeUserProfile1), and the roles *Manager* and *Claims Processor*:

```
WFUserProfileDesc myUser (cndTask);
myUser.SetProfileName(‘AcmeUserProfile1’);
myUser.SetUserName(‘jfleuri’);
myUser.AddRole(‘Manager’);
myUser.AddRole(‘Claims Processor’);
```

NOTE The user name is case-sensitive.

Customizing a User Profile

The default definition of UserProfile includes only a name and a set of roles. However, if the user description is more complicated (for example, it indicates a specific level of budget approval authority), someone at your site must create a new definition of UserProfile in the User Profile Workshop, distribute the new user profile library, register it with the appropriate engines, and notify you so you can use the appropriate profile name and attributes in your WFUserProfileDesc object.

For example, users of a client application at Acme, Inc., in addition to roles, have a maximum budget authority, which is defined in AcmeUserProfile2 as the integer attribute *maxBudget*. The following code would initialize an object of this class for the user with user name *hgretel* in the role *Manager* with a maximum budget authority of $5,000:

```
WFUserProfileDesc myUser (cndTask);
myUser.SetProfileName('AcmeUserProfile2');
myUser.SetUserName('hgretel');
myUser.AddRole('Manager');
myUser.SetInteger('maxBudget',5000);
```

The Engine and the WFUserProfileDesc Object

When you pass a WFUserProfileDesc object as a parameter to WFEngine::OpenSession, the engine accesses the appropriate user profile definition, passing the object to the Validation.ValidateUser method. This method is written by someone at your site to interpret the WFUserProfileDesc attributes correctly, and, like the user profile, is registered with the engine.

Suspended and Reconnected Sessions

If the session is interrupted by a network or engine failure (because the engine has not been set up to be fully fault-tolerant) or by intervention from a system administrator, you can have the engine either suspend the session (because the user might be in the middle of some work) or terminate it (thus losing whatever work might be in progress). Allowing a session to be suspended means that any work the user was doing can be resumed at a later date. Requiring that a session be terminated when it loses contact with the engine means that the work the user was doing will be lost, any process attributes that were locked will be unlocked, and the corresponding activity will be made READY again, available for other users to perform.

When a session is put in a SUSPENDED state, the following actions occur:

- Any activities that are currently ACTIVE are processed according to each activity's defined suspend action (or the overriding value specified in the suspendAction parameter of StartActivity or StartQueuedActivity). ACTIVE means that the activity was started by the client application's making a call to either WFActivity::StartActivity or WFSession::StartQueuedActivity. For more information on these functions, see the iIS online help.

If the defined action in the process definition is *remove* (or the suspendAction parameter specified WFActivity::REMOVE), the activity is removed from the session's activity list. Depending on the specification in the process definition, the activity is either aborted or rolled back to the READY state (rolling back includes closing the activity's attribute accessor with ROLLBACK). The default action is to abort the activity when the session is suspended.

If the defined action is *retain* (or the parameter specified WFActivity::RETAIN), the activity is retained on the session's activity list in the ACTIVE state.

- Any activities that were on the session's activity list that were not chosen by the user are in the READY state. The engine takes these activities off that session's activity list because the user can no longer choose any of them.

Disconnect Action Values

To determine whether a session is suspended or terminated if the connection to the engine is unexpectedly interrupted, you set the controls parameter of OpenSession with one of the Disconnect Action values. The settings are WFEngine::SUSPEND_ON_DISCONNECT or WFEngine::TERMINATE_ON_DISCONNECT.

If you do not set one of these Disconnect Action values, the default action is to terminate the session if the WFSession object determines that the engine is disconnected and cannot be reconnected.

When you start a session, you can indicate whether you want the engine to reconnect to a previously suspended session, if there was one, or to discard any previously suspended sessions and start fresh. If you reconnect to a suspended session, the user can gain access to work that was in progress when the session was suspended.

Reconnect Action Values

To determine whether a session reconnects to a suspended session or starts up as a new session, you set the controls parameter of OpenSession with one of the Reconnect Action values. The settings are WFEngine::RECONNECT_ALLOWED or WFEngine::RECONNECT_PROHIBITED.

If you do not set one of these Reconnect Action values, the default action is to reconnect to a suspended session.

Offered Activity Notifications

Whenever an offered activity becomes READY, the engine compares its assignment rules to the user profile objects of all active sessions and adds the activity to the activity lists of the sessions that match. However, if a session is heads-down, it works only with queued activities, and allowing the engine to test the session's user profile object against the assignment rules of offered activities is a waste of system resources, because the session never gets any offered activities.

To save system resources, an application logging on as a heads-down session should set the controls parameter of OpenSession with the Activity Offers value WFEngine::OFFERED_IGNORED. If this value is not set in the controls parameter, the engine by default will evaluate the session whenever an offered activity becomes READY.

Opening a Session

When you have determined the conditions for your client program's session with the engine, you can open the session by calling the WFEngine object's OpenSession member function. The following code starts a new session with the engine Claims. It uses the sample code from the previous sections to set the parameters and starts up with the following session controls set:

- If the engine disconnects, the session is suspended.
- The session is allowed to reconnect to a suspended session.
- ActivityListUpdate events are not sent (the default; see "Using Activity Update Caching" on page 106 for details on using event handling).
- The session is evaluated whenever an offered activity becomes READY (the default).

Code Example 3-1 Opening a session (C++)

```
#include <conductr.h>
...
int main(int argc, char** argv)
{
  WFTaskHandle cndTask;
  printf(
    "Starting the C++ client for the Expense Reporting
process.\n");
  cndTask = WFStartup();
  ...
  WFEngine myEngine (cndTask);
  myEngine = WFFindEngine(‘Claims’);
  WFUserProfileDesc myUser (cndTask);

  // Set up user profile info
  myUser.SetProfileName(‘AcmeUserProfile1’);
  myUser.SetUserName(‘jfleuri’);
  myUser.AddRole(‘Manager’);
  myUser.AddRole(‘Claims Processor’);
  char * password = ‘D4xiijepl’;
  char * nameForSession = ‘MySession’;
  WFSession mySession (cndTask);

  // Open the session
  mySession = myEngine.OpenSession (
          nameForSession,
          myUser,
          password,
          WFEngine::RECONNECT_ALLOWED +
              WFEngine::SUSPEND_ON_DISCONNECT);
  wf_i4 openStatus = mySession.GetOpenStatus();
  ...
}
```

If the session is unable to start, the OpenSession call throws an exception. Otherwise, the session object is created and immediately usable.

To determine if the session reconnected to an existing suspended session or started as a new session, the last line of code calls GetOpenStatus. The next thing to do would be to test if the returned value was WFSession::RECONNECTED_SESSION or WFSession::NEW_SESSION. If the session has reconnected to a previously suspended session, there might be one or more activities in the ACTIVE state that the client application must display to the user. If not, the session is a new one, and the client application can simply continue to get activities and display them to the user.

Whether the session reconnected or is a new session, the client application must indicate what work is available to the user, as described in the next section.

Indicating What Work Is Available

Once the session has been established with the engine, the client application must determine what work is available to it. How it does this depends on whether the client application is a heads-up or a heads-down application.

- If the client application is a heads-up application, it is likely to display a work list to the user, as described next in "Getting and Handling Offered Activities."
- If the client application is a heads-down application, it retrieves the next queued activity available to it, as described in "Getting and Handling Queued Activities" on page 109.

NOTE The following sections have examples that use attribute accessors (WFAttributeAccessor) to work with process attributes. For an introduction to process attributes and a description of how WFAttributeAccessor objects can be obtained, see "Working with Process Attributes" on page 42.

Getting and Handling Offered Activities

For each session, the engine maintains a *session activity list*. This list has on it activities that are in READY state that the engine has offered to the session (because the session user profile matches an assignment rule for each of the activities). In addition, this list includes any activities that the session has made ACTIVE. A call to WFSession::GetActivityList returns this list of activities.

To get this list of activities from the engine, you declare an object to receive the return value of the method call, then call WFSession::GetActivityList, as shown in the following code sample:

```
WFActivityArray activityList (cndTask);
activityList = mySession.GetActivityList();
```

Building a Work List

Each activity object returned by GetActivityList has associated with it an application dictionary entry composed of an activity description, an application code, and, if the process designer defined them, a set of process attributes. You can construct a work list by extracting this information from each activity. Using the WFActivity member functions GetName and GetActivityDescription, you can get the name and description of the activity.

It is likely that the work list you display for the user will include supplemental information for each entry (such as a customer name, purchase amount, order number, and so on), which you can obtain by calling each activity's GetAttributeAccessor member function to obtain a WFAttributeAccessor. You can then use the attribute accessor's GetNames and GetValue member functions to get the names and values of these process attributes.

NOTE If the return value of GetAttributeAccessor evaluates to WF_TRUE when you call the IsNil function, there is no attribute accessor associated with the activity.

The following example shows a method that builds a work list display from a WFActivityArray. It uses objects such as ExpenseReportListView that are defined elsewhere and assumes that certain attributes, Status, ExpenseRptID, and Priority, are associated with each returned activity. After getting the list of activities, it gets the number of activities in the list; it then loops through the array if there are any members:

Code Example 3-2 Building a work list display (C++)

```
void BuildList(WFSession session) {

// Initialize listview field.
// This field will display the worklist.
ExpenseReportListView listView;
listView.IsFilled = TRUE;
listView.IsFolder = TRUE;
listView.IsOpened = TRUE;
// Get the activity list.
WFActivityArray activityList (cndTask);
int maxActs;
WFAttributeAccessor attribAccessor (cndTask);
activityList = mySession.GetActivityList();
maxActs = (int) activityList.Items();
for (i=0;i<maxActs;i++) {
  attribAccessor = activityList[i].GetAttributeAccessor();
```

Code Example 3-2 Building a work list display (C++) *(Continued)*

```
    // Get values for this row in worklist
    // from attribute accessor
    listView[i].Status = attribAccessor.GetInteger("Status");
    listView[i].ExpenseReportID =
          attribAccessor.GetInteger("ExpenseReportID");
    listView[i].Priority = attribAccessor.GetBoolean("Priority");
    listView[i].WorkItem = activityList[i].GetName();
}
```

Maintaining a Work List

You can use one of the following two approaches for keeping a session work list display updated:

- poll the engine periodically
- respond to events posted by the engine

Polling the Engine

Using the polling approach, you periodically call WFSession::GetActivityList to obtain the entire activity list for the session. You can use this list either to completely refresh the display or you can merge the changed elements of the list with the corresponding displayed elements, depending on which is faster.

Using Activity Update Caching

A feature called *activity update caching* enables a C++ application to respond to iPlanet UDS events without having to repeatedly poll the engine. To use activity update caching, you start a session as you normally do, and then turn on activity update caching in the client session. You then create a client variable of type WFSessionUpdateArray and use it when you call WFSession::GetSessionUpdates to receive lists of activity updates.

Update caching works best if WFSession::GetSessionUpdates is called in a thread, allowing the main client thread to continue with other processing. However, on systems that do not have threading, it is possible to use this member function, but call it without waiting for a return value so the member function never gets blocked.

➤ **To use activity update caching**

1. Connect to an engine and start a session (for sample code, see “Opening a Session With the Engine” on page 96).

2. Get the initial list of activities, turn on update caching, and display the work list. For example:

```
WFActivityArray myActivityList (mainTask);
myActivityList = mySession.GetActivityList (
    WF_AL_EVENTS_ON | WF_SAVE_UPDATES_ON);
```

   Calling GetActivityList with these two settings turns on activity update caching. The default number of updates that can be cached is 20. If you want to change the default number, you can call WFSession::SetUpdateLimit, described in the iIS online help.

   When you are doing activity update caching, whenever you call GetActivityList, you must specify both that you want events turned on and that the engine is to cache them.

3. Declare a variable of type WFSessionUpdateArray and start a thread that uses the variable to pick up the latest group of activity updates. The code in the thread calls WFSession::GetSessionUpdates, as shown in the following code fragment:

```
WFSessionUpdateArray updateList (mainTask);
int i;

// Start a thread if supported
. . .
updateList = mySession.GetSessionUpdates(WF_TRUE);
if (!updateList.IsNil())
{
    for (i = 0; i < updateList.Items(); i++)
    {
        // Do something with each change
        // Could update worklist here
    }
}
```

   In this code fragment, GetSessionUpdates indicates that it will wait until the engine has activity updates. Having the member function wait requires that the call be imn a separate thread if the main thread is to continue processing.

You can cancel a waiting GetSessionUpdates call by calling WFSession::CancelGetSessionUpdates from another thread of the same session.

There is no code in the code fragment's for loop. One thing you could do here is to update the work list with each activity encountered.

If you cannot create a separate thread for the GetSessionUpdates call, call it with wait set to WF_FALSE. You must then periodically call it again to get the latest session updates.

Update Limit

The session starts with *update limit*, the number of activity updates saved for this session, set to a default value of twenty. You can set the update limit to a different value by calling WFSession::SetUpdateLimit.

If the total number of updates for this session exceeds this limit, the client session enters an *overrun* state, adds an entry to the update list indicating that the limit has been exceeded (a WFSessionUpdate object with SessionChange value of WFSession::SESSION_WORK_OVERRUN), and removes all activity updates from the list. Until you reset the client session's state with a call to WFSession::GetSessionUpdates, the client session saves no activity updates for the session.

If your code detects that the update limit has been exceeded (by encountering a WFSessionUpdate object in the WFSessionUpdateArray with a SessionChange value of WFSession::SESSION_WORK_OVERRUN), your client application must call WFSession::GetActivityList to get all activity changes, then clear the work list and redisplay it.

When you call WFSession::GetSessionUpdates and specify that the member function will not wait (the parameter is false), the member function returns even if there have been no updates. If there have been no updates, the returned WFSessionUpdateArray object is null (WFSessionUpdateArray::IsNil returns true.)

If you specify that the member function will wait until there is something in the list (the parameter is true), the member function could return a null WFSessionUpdateArray object if one of the following events occurs:

- the connection to the engine is lost
- the client application's main thread suspends or closes the session
- the client application's main thread calls WFSession::GetActivityList
- the client application's main thread calls WFSession::CancelSessionUpdateWaiting

Starting an Activity

In a heads-up application, when the user chooses a work item to work on, the client application calls WFActivity::StartActivity for the associated activity (see the iIS online help for more details on this function). Call this function synchronously (wait for it to return) and use it in a qqhTRY() block to ensure that the client application has gotten the activity, because it is possible that another client application has already chosen it. If the engine can change the activity's state to ACTIVE and give it to your session, it does so, and the function returns successfully. If the activity has already been taken, a WFActivityListException will be raised by the engine.

The preferred way of working with activities is to specify their behavior in the process definition and not override these settings when you start them from a client application. (The less specific information about a process definition the client application must carry, the freer the process designer is to make design changes.) Therefore, it is usually best to call StartActivity by passing a NULL pointer to the attributeList parameter to pick up the default set of process attributes associated with the activity in its application dictionary entry.

Getting and Handling Queued Activities

A heads-down application presents the user with the next item of work—it does not allow the user to choose from a work list as a heads-up client application does. This *next item of work* is represented in a process definition as a queued activity. To support heads-down applications, the engine maintains prioritized queues of queued activity instances. Each queue is composed of "like" activities, queued activities that have the same name and process definition. The highest priority activity is at the top of the queue, ready to be assigned to the next eligible session that asks for it.

To determine which sessions are eligible, the engine attaches the activities' assignment rules to the queue. A session requesting an activity in the queue must have a user profile matching the requirements of the queue's assignment rules. If so, the session gets access to the queue. (For more information on assignment rules, see the description in the *iIS Process Development Guide*.)

In a heads-down application, when the user is ready for the next item of work, you call your session object's StartQueuedActivity function. For example, the following code sample retrieves an activity called DoAccounting that is in the ExpRptActng process. If the session gets suspended, the activity remains on the session's activity list. If there is no activity in the queue, the method immediately returns a NIL object:

```
WFActivity curWorkItem (cndTask);
curWorkItem = mySession.StartQueuedActivity(
         "ExpRptActng",
         "DoAccounting",
         WFSession::IMMEDIATE);
```

This code sample does not use two parameters of the function, suspendAction and attributeList, for a reason: if you use these two parameters, they override behavior that should be defined in and controlled by the process definition.

If there is an activity on the queue, the engine retrieves the highest priority activity from the queue, marks it ACTIVE, adds it to the session's activity list, and returns a WFActivity object that describes the activity. (See "Dealing With an Empty Activity Queue" for information on how to handle a queue with no activities in it.)

A WFAttributeAccessor object is built as part of this operation if specified by the activity definition or if a list of WFAttributeDesc objects is provided in the method's attributeList parameter. If no process attributes are defined for the activity and none are specified in the StartQueuedActivity call, the attribute accessor returned evaluates to WF_TRUE when you call its IsNil member function. The following code sample retrieves the attribute accessor for an activity that has been returned to the client application:

```
WFAttributeAccessor curAttribs (cndTask);
curAttribs = curWorkItem.GetAttributeAccessor();
```

The following sample code starts a queued activity and tests the return value to see if an activity was retrieved. If so there's an attribute accessor that can be used to obtain the associated process attribute information. (See the previous section for a code sample that gets attribute values from an accessor.)

```
WFActivity qAct (cndTask);
qAct = mySession.StartQueuedActivity (
   "ERPD",
   "DoAccounting",
   WFSession::IMMEDIATE);
WFAttributeAccessor accessor (cndTask);
if (qAct.IsNil())
  printf ("No expense reports available.\n");
else
  accessor = qAct.GetAttributeAccessor();
```

Dealing With an Empty Activity Queue

If there is no activity on the queue, your client application can either wait until one is available or return immediately and try again later. The previous code sample specifies an immediate returns. If it had specified an emptyAction of WFSession::WAIT, your client application would wait until a DoAccounting activity instance was added to the queue.

You can control how long your client session waits by starting a separate timed task to handle the call to StartQueuedActivity. If the method call does not return before the timeout, your client application can call the CancelStartQueuedActivity method to cancel the StartQueuedActivity call. In this case, StartQueuedActivity returns with a NIL object.

For more information on multithreading C++ applications with iPlanet UDS, see the iPlanet UDS manual *Integrating with External Systems*.

Completing Work On an Activity

When the user has finished working on the work item, set any process attributes that need setting with the WFAttributeAccessor::Set*Type* member functions (for example, SetInteger), then call WFActivity::CompleteActivity to tell the engine you are done. The engine commits all changes and the process continues as indicated by the process definition.

Rolling Back an Activity

If work was not completed and the user wants to quit without applying updates, you can call WFActivity::RollbackActivity to roll back changes the user has made and reset the activity's state to READY, making it available to other users. What the engine rolls back are changes to process attributes associated with the activity. Any changes that your client application has made to site files and any transactions that it has started with site databases must be explicitly undone or rolled back by your client application.

Aborting an Activity

Another option is to call WFActivity::AbortActivity to abnormally terminate the activity. Do not exercise this option unless something drastic has happened and you want to risk aborting the process. Unless the process definition explicitly handles aborting activities, it will abort the entire process.

Creating a Process Instance

To start a new instance of a process definition, you call WFSession::CreateProcess, indicating the name of the process definition in the processName parameter. The user must be in a role that can create a new instance of the process for this function to succeed. (In other words, the process definition must have an assignment rule that gives a user with the same role as the user of this client application permission to create a new instance of the process.)

If you also want to initialize process attributes when you start the process instance, you can put the list of process attributes and their values in a WFAttributeDescArray and pass this object in the attributeValues parameter of CreateProcess.

NOTE When the engine gets a request to create a new process instance, it ignores the LockType attribute of any WFAttributeDesc objects in the attributeValues parameter. For that reason, you need to set only the Name and Value attributes of each WFAttributeDesc object in the array.

The following code sample illustrates how to build a WFAttributeDescArray:

Code Example 3-3 Initializing process attributes (C++)

```
// Declare the array object
WFAttributeDescArray attrDescArray (cndTask);

// Declare each descriptor object
WFAttributeDesc desc1(cndTask);
WFAttributeDesc desc2(cndTask);
WFAttributeDesc desc3(cndTask);

// Make each descriptor an object the engine can recognize,
// then initialize it
desc1.New();
desc1.SetName ("ExpenseReportID");
desc1.SetInteger (SetID());
desc2.New();
desc2.SetName ("TotalAmount");
desc2.SetInteger (GetTotalAmount());
desc3.New();
desc3.SetName ("Status");
desc3.SetInteger (EXP_RPT_SUBMITTED);

// Put each descriptor in the descriptor array
attrDescArray.AppendRow(desc1);
attrDescArray.AppendRow(desc2);
attrDescArray.AppendRow(desc3);
```

The following code creates a process instance by using the previously built attribute accessor array:

```
WFProcess ERProcess (cndTask);
ERProcess = mySession.CreateProcess("ERPD", attrDescArray);
```

Setting Process Recovery Level

By default all current process state information is recovered for a process in the event of engine or system failure. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. If appropriate, the recovery level can be altered so that the process instance is simply restarted with no recovery of process state information; another alternative you have is specifying no recovery of the process.

The default behavior of full recovery applies under the following conditions:

- no other value is specified for the process recovery level in the Process Definition property inspector
- you do not override the default in the client application code
- the engine is configured for state recovery

If logging options for the engine are turned off, the process definition default does not override the engine setting. For details on setting the recovery level in the Process Definition Workshop, see the *iIS Process Development Guide*. For details on engine configuration, see the *iIS Process System Guide*.

You can use the recoveryInfo parameter of the CreateProcess member function to control whether or not the recovery level value specified in the process definition is used when creating the process. This member function returns a WFProcess object for each newly created process instance. If either no value or the value WFProcess::RCVR_NORMAL is passed in for recoveryInfo, the process is created using the recovery level specified in the process definition. If another value is passed in, that value overrides the recovery level value specified in the Process Definition property inspector. The possible other values are WFProcess::RCVR_FULL, WFProcess::RCVR_NONE, or WFProcess::RCVR_PROCESS_ONLY.

Closing a Session

When the user wants to log off, your client application must close the session with the engine. You call the WFSession::CloseSession method and, depending on whether the user has completed work or not, indicate with the newState parameter that the session is to be either terminated or suspended.

Terminating the Session

If the user has completed all work and the engine has successfully closed any associated activities, you can terminate the session by calling CloseSession with the newState parameter set to WFSession::TERMINATED. Terminating the session causes the engine to remove all READY activities from the session's activity list. (For a description of CloseSession, see the iIS online help.)

Another effect of terminating the session is that the engine aborts any ACTIVE activities. If the user has any work in progress, it is represented by at least one ACTIVE activity, and all the work is lost at this point. If the user indicates that the session is to be terminated and there is an ACTIVE activity, it would be good to let them know the consequences and suggest either suspending the session or finishing all work as alternatives.

Suspending the Session

If the user has any work in progress, you typically would take whatever steps are necessary to save the current work and then suspend the session by calling CloseSession with the newState parameter set to WFSession::SUSPENDED. Suspending the session causes the engine to remove all READY activities from the session's activity list, but leave any ACTIVE activities on the suspended session's activity list. When the user logs in at a later time, you can open the new session and tell the engine to reconnect to the suspended session by indicating WFEngine::RECONNECT_ALLOWED in the controls parameter of WFSession::OpenSession.

Handling the Connection With the Engine

As described in "Suspended and Reconnected Sessions" on page 100, when the client application starts a session (by calling WFSession::OpenSession), it is connected to an engine through the client application's WFSession object. If you call this object's SetPingInterval method, it monitors the client application's connection to the engine in the background, sending *ping* messages to ensure that the connection is still up, and handling reconnection if the connection goes down (either the network connection is lost or the engine fails). By default, however, pinging is off.

The default interval is set to 2 seconds. You can turn pinging on and change the interval by calling WFSession.SetPingInterval or WFSession.SetRetry. Set the length of this interval to a value appropriate for your network connection. For more details, see the iIS online help.

Losing the Connection

If pinging is off, the WFSession object does not test the connection with pings. However, if the client calls a function that contacts the engine and the engine is down, the session object does detect the disconnection. If pinging is off, the client application must disconnect and reestablish the session later.

If pinging is on and the connection to the engine goes down, the session object automatically attempts to reestablish the connection. If it is able to do so within a time period that you can set with the WFSession.SetRetry method (by default, 20 retries at intervals of 2 seconds), the session object automatically reconnects with the engine, and the session is able to continue.

If the session object is unable to reestablish the connection within this time period (for example, the engine has failed and there is no backup engine that can be automatically restored, or the engine takes an especially long time to come back up), it raises a WFException with LostContact as the reason. At this point, the client application must disconnect and reestablish the session later.

Reestablishing the Connection

As with starting a new session, the client calls WFEngine::OpenSession to restart a suspended session. The controls parameter must include the value WFSession::RECONNECT_ALLOWED. When the session is open, the next thing the client does depends on whether it is a heads-up or a heads-down application.

For a heads-up application, on the first call to GetActivityList, if there are any ACTIVE activities in the returned list and the return value of OpenSession was WFSession::SUSPENDED_SESSION, the user was working on these activities previously when the session was suspended. The client application can test the current values of attributes and prompt the user for any information needed to continue work on the activities.

Normal behavior for a heads-down client application after it starts a completely new session is to do a StartQueuedActivity call to get its first activity. However, when reconnecting to a suspended session, a heads-down client must first call WFSession::GetActivityList to determine if it already has an ACTIVE activity left over from when its session was suspended. If there is an ACTIVE activity on the list, the client application can deal with it in the same manner as a heads-up client does (see previous paragraph).

If you do not want to reestablish connection with the suspended session, you set the reconnectAction parameter to WFSession::RECONNECT_PROHIBITED. The engine terminates the suspended session, aborting any ACTIVE activities, and starts a new session for your client application.

Recovering Attribute Accessors

After you reconnect a session to the engine, you must call WFSession.GetAtrributeAccessors to see which accessors your recovered session has open, if any. This method returns an array of attribute accessors. You can call the WFAttributeAccessor methods GetName, GetID, and GetProcessID to determine which attributes belong to which process instances.

Chapter 4

Building a CORBA/IIOP Client

This chapter describes how to use the iIS CORBA/IIOP process client API to write a client application for an iIS enterprise system. Because the CORBA/IIOP client API is represented in Interface Definition Language (IDL), it is not language specific and can be used by various languages, such as C++ and Java. Examples in this section and in the iIS online help for the CORBA/IIOP client API are written in Java.

This chapter will be more useful to you if you have read the first chapter of the *iIS Process Development Guide*, which provides an overview of the iIS process management system. It is also helpful to read this manual's Chapter 1, "Building an iIS Process Client Application," which describes what a client application is and provides overviews of starting and ending sessions, retrieving activity lists and starting activities, and retrieving and updating client objects, process attribute accessors, and timers.

This chapter starts by describing tasks you need to perform to set up your system and perform CORBA/IIOP call-ins to the iIS process engine. After this general information, the chapter continues with a description of how to use the CORBA/IIOP API with iIS, covering typical tasks your client application is likely to perform, such as opening a session and getting a list of available activities.

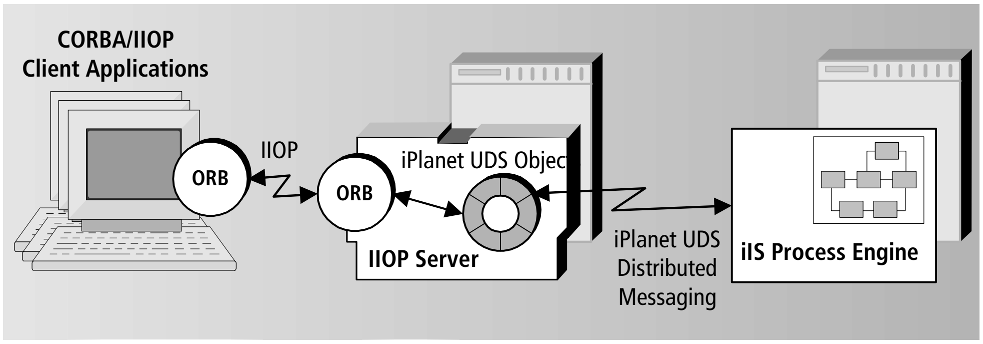
Using the CORBA/IIOP Client API

The CORBA/IIOP client API must be used by client applications that conform to the CORBA (Common Object Request Broker) 2.0 and IIOP (Internet Inter-ORB Protocol) standards. You typically create these applications with an Object Request Broker application development environment like Visigenic Visibroker for Java or Iona OrbixWeb, which translates the CORBA/IIOP API from IDL (Interface Definition Language) to whatever language you are using to develop the client.

To be able to communicate with the iIS process engine, both the CORBA/IIOP client application and the iIS engine must have their own ORBs (Object Request Brokers). Your client development environment creates the ORB used by your client application while iIS provides the ORB for the process engine. Use the Conductor Script command, IIOPserver, to start up the ORB for the iIS environment and create an IOR file. Your client application uses the IOR file to access the iIS process engine IIOP API.

The following diagram illustrates communication between a CORBA/IIOP process client application and the iIS environment containing an engine.

Figure 4-1 Communication Between a CORBA/IIOP Client and the iIS Environment



See the *iIS Process System Guide* for more information on Conductor Script and the IIOPserver command.

Understanding the CORBA/IIOP API Data Types

In the CORBA/IIOP client API, some of the data types, like string and integer, are native to IDL and some, like ui4 and DateTimeData, are translations of iPlanet UDS data types used in iIS process attributes.

The following table shows these iPlanet UDS data types as declared in this API and what they mean in CORBA/IIOP IDL. For the more complex data types, there is a longer explanation following the table.

iPlanet UDS data type	CORBA/IIOP IDL equivalent
DateTimeData	string (described in detail after table)
IntervalData	string (described in detail after table)
TextData	string
ui4	unsigned long (4-byte integer)--applies to method return values that represent identifiers used by the iIS process engine, such as a process ID

DateTimeData in CORBA/IIOP IDL

The DateTimeData type is represented as a string in IDL It represents a specific date and time and consists of the day in DD-Mmm-YYYY format followed by a space and the time in HH:MM:SS format. The hours are in 24-hour clock format. For example, the following code sample sets a DateTimeData variable to 2:35 PM on January 31, 2001:

```
Framework.DateTimeData myDate = "31-Jan-2001 14:35:00";
```

IntervalData in CORBA/IIOP IDL

The IntervalData type is represented as a string in IDL It represents an interval of time, such as 1 hour and 37 seconds or 2 days, and consists of a series of integer fields separated by colons in the following format:

"*years*:*months*:*days*:*hours*:*minutes*:*seconds*"

You can use any integer value between the colons. Because the string is read from left to right, if you do not need fields to the right, you do not have to fill them in.

For example, the interval of 6 years, 4 months, and 23 days could be entered as follows:

```
"6:4:23"
```

The values you enter are not limited by the type to the left of the field. In other words, you can enter more than 12 months, more than 30 days, more than 24 hours, more than 60 minutes, and so on. All the fields are summed to get a final interval value. If the value for a field is zero, but it must precede another field to make its units the correct ones, enter “0” for that field. For example, the previous interval of 6 years, 4 months, and 23 days could also be written as 72 months, 88 days, and 36 hours, as follows:

```
"0:72:88:36"
```

The following example shows how to declare an IntervalData variable and set its value to 2 hours, 17 minutes, and 39 seconds:

```
Framework.IntervalData myInterval = "0:0:0:2:17:39";
```

Error Handling

If you use Java try/catch blocks around code that invokes iIS process engine methods you can catch exceptions from the iIS process engine or the CORBA server. If your method does not catch the exceptions generated by the iIS engine (or the CORBA server), the method must declare that it throws those exceptions.

The following code sample attempts to open a session with an engine based on the parameters given. The catch block at the end handles all normal runtime errors that iIS, CORBA, or Java may generate. The code uses the CorbaWFEngine object to create user profile objects:

```
public static WFSession openSession( CorbaWFEngine eng,
String userProfileName, String userName, String userRole,
String userPassword, String sessionName )
{
    CorbaWFSession theResult = null;

    try {
      WFUserProfile user = eng.GetUserProfile();

      user.SetProfileName( userProfileName );
      user.SetUserName( userName );
      user.AddRole( userRole );

      theResult = eng.OpenSession( sessionName, user,
        userPassword,
        CorbaWFEngine.OFFERED_ACCEPTED  +
        CorbaWFEngine.RECONNECT_ALLOWED +
        CorbaWFEngine.SUSPEND_ON_DISCONNECT  );

    } catch( Exception ex ) {
        showError( "Cannot open session: " +
        ex.getMessage() );
    }

    return theResult;
}
```

Setting Up Your System to Use the CORBA/IIOP API

The CORBA/IIOP API is in the file FORTE_ROOT/userapp/wfcorbaa/cl1/conductr.idl.

NOTE FORTE_ROOT refers to the directory where you installed iIS. On a Windows NT system, for example, the default location is c:\forte.

Use your application environment's conversion command to convert this file to classes in the language in which you are developing.

For example, if you are using Visigenic Visibroker, the command is:

```
idl2java conductr.idl
```

This command generates Java classes and saves them in subdirectories in your CLASSPATH so that your Java compiler can access them.

Starting the iIS IIOP Server

When you are ready to test your program, you must have the iIS IIOP server running. To start the iIS IIOP server, start Conductor Script, then enter the following command:

```
IIOPserver start
```

Entering this command starts the iIS IIOP server and generates the IOR (Interoperable Object Reference) *conductr.ior file* in the directory FORTE_ROOT/etc/iiopior/. This file contains the node name, the TCP/IP port, and any other information necessary to find the ORB. The full path to this file is:

```
%FORTE_ROOT%/etc/iiopior/conductr.ior
```

Using the CORBA Naming Service

The CORBA Naming Service provides a mechanism for client applications to locate a CORBA server. A CORBA server can register itself by name with a name server. Client applications can then use the name server to locate the CORBA server, bind to the server object, and call methods on it. For more information on the CORBA Naming Service interface, refer to your CORBA specification or your CORBA vendor's documentation.

The iIS ORB uses an *ns.ior file* to locate a name server to register with. The ns.ior file must contain the IOR for the CORBA name server you are using. When the iIS ORB starts, it looks for a CORBA name server to register with by examining the ns.ior file at the following location.

```
%FORTE_ROOT%/etc/iiopior/ns.ior
```

If the file exists and contains a valid IOR, then the iIS ORB registers with that name server using the name "ConductorServer." The section "Multiple CORBA Servers" on page 126 shows how you can modify the name used by the iIS ORB to register with a naming service.

If you want the iIS ORB to register with a naming service, you must first create the ns.ior file before starting the iIS ORB. One way to create the ns.ior file is to write a simple client application to a name server that obtains the server's IOR string and writes it to a file. The following Java example shows how to create an ns.ior file for the Java name service tnameserv:

```
initNCRef = orb.resolve_initial_references ("NameService");
rootContext = NamingContextHelper.narrow (initNCRef);
FileWriter theout;
theout = new FileWriter("ns.ior");
PrintWriter iorout = new PrintWriter(theout);
iorout.println(orb.object_to_string(rootContext));
iorout.close();
```

To use the CORBA Naming Service in your iIS process client, obtain the Conductor Directory object from the name server (instead of reading the conductr.ior file), as illustrated in the following Java example:

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

// Convert the IOR string into an object reference
org.omg.CORBA.Object NSobj;
NamingContext rootContext=null;
try {
      NSobj = orb.resolve_initial_references("NameService");
      rootContext=NamingContextHelper.narrow(NSobj);
      if (rootContext == null) {
        // error
      }
}
catch (org.omg.CORBA.ORBPackage.InvalidName ie) {
      // ...
}
org.omg.CORBA.Object objRef = null;
NameComponent[] NC = new NameComponent[1];
NC[0] = new NameComponent("ConductorServer","");

try {
      objRef = rootContext.resolve(NC);
}
catch (org.omg.CosNaming.NamingContextPackage.CannotProceed e)
{ ... }
```

NOTE The iIS example JExpenseNS shows how to use Sun's Java implementation of the CORBA Naming Service.

Multiple CORBA Servers

If you want to have multiple CORBA servers running you typically start each server with a different *advertise name*. Each server then registers with the name server using its advertise name. Do either of the following to set the advertise name:

- Set the environment variable CONDUCTOR_IIOPSERVER_NAME

  By default, this is set to ConductorServer.

- Use the following command line to start a server with the specified AdvertiseName, which overrides the name specified by CONDUCTOR_IIOPSERVER_NAME

```
ftexec -fi bt:%FORTE_ROOT%/userapp/wfcorbaa/cl1/wfcorb1 \
-n AdvertiseName
```

Using the IOR File to Access the iIS Environment

Your client application must be able to reference the iIS ORB by reading the IOR file. Since the IOR file is regenerated each time the iIS ORB is started, unless you want to copy it each time to another location, it is easiest to reference it in the default location.

For example, the full path to the default IOR file location could be a command line startup parameter for the client application. On the Windows NT command line, it would like something like this:

```
java myclient %FORTE_ROOT%/etc/iiopior/conductr.ior
```

Your program would read this command-line input and find the IOR file, and then create an object from the information read in the file. The final step to enable your client application to establish a session and find an iIS process engine would be to cast this object as a CORBA/IIOP API object of type CorbaWFDirectory.

The following code shows how to set up the connection through the IOR file for a Visibroker Java program that is started with the IOR file as a command-line option. The GetIOR method reads the IOR file by using the file name entered on the command line and creates an object to be used to initialize the connection to the iIS process engine:

Code Example 4-1 Setting up a connection using the IOR file (CORBA)

```
  public static String GetIOR(String fileName) {
    DataInputStream input = null;
    string ior;

    try {
      FileInputStream x = new FileInputStream(fileName);
      input = new DataInputStream(x);
    }
    catch(FileNotFoundException ex) {
      return null;
    }
    try {

      // Read contents of IOR file
      ior = input.readLine();
    }
    catch(IOException ex){
      ior = null;
    }
    try {
      input.close();
    }
    catch(IOException ex) {
    return ior;
  }
. . .
//
// "main" method
public static void main(String args[]) {
  if (args.length <= 0) {
    System.out.println("Please specify IOR-file");
    System.exit(1);
  }

  // Put contents of IOR file into ior object
  String ior = GetIOR(args[0]);
  if (ior == null) {
    System.out.println("IOR-file " + args[0] +
                " does not exist or is empty");
    System.exit(1);
  }

    // Initialize Visigenic ORB for iIS client application
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

    // Convert IOR string into object
    org.omg.CORBA.Object obj = orb.string_to_object(ior);

    // Cast Visigenic object into the iIS
    // object that retrieves engines
    CorbaWFApi.CorbaWFDirectory DirectorySO =
        CorbaWFApi.CorbaWFDirectoryHelper.narrow(obj);
. . .
```

Accessing the API

After setting up your system and application to access the iIS Environment through CorbaWFDirectory, you can access the iIS engine by using the API. The remainder of this chapter discusses how to use the iIS CORBA/IIOP process client API in a client application.

NOTE Although the API is in generic IDL format, all examples are given in Visigenic Java.

The Client API

A client application uses the client application program interface (API) to communicate with the engine and execute processes registered with that engine. Following are some of the things a client application does with the API:

- set it up to talk to the iIS ORB (see the previous section, "Using the IOR File to Access the iIS Environment" on page 126)
- open a session with the engine
- display activities (work items) to the user, either as a work list (heads up) or a single work item (heads down)
- take control of an activity ("start" an activity) so the user can perform it
- set process attributes related to the work the user wants to do
- notify the engine when the work is done
- start instances of processes
- close sessions
- handle exceptions

This section explains how these tasks are accomplished using the iIS CORBA/IIOPprocess client API, with sample Java code included.

Opening a Session With the Engine

To perform useful work, your client must establish a session with an engine that is running in an iPlanet UDS environment. In your client application, you call either the CorbaFindEngine method or the CorbaGetAvailableEngines method of the CorbaWFDirectory object to obtain a CorbaWFEngine object representing the engine you want to start a session with. You then call this CorbaWFEngine object's CorbaOpenSession method, passing various parameters to establish your session.

Before opening obtaining an engine object, you need to determine the following information:

- the name of the engine that executes the process definitions your application will work with and whether the engine is running.
- the environment where the engine is running if it is not the current, active environment or a connected environment

After obtaining an engine object, but before opening a session with the engine, you need to obtain the following information to pass to CorbaWFEngine.CorbaOpenSession:

- the user name and password of the user opening the session
- the name of the session
- how you intend to handle various conditions, like an unexpected termination of the session or the user suspending the session

When you have obtained the information you need and determined what you want to do with the session, you call CorbaWFEngine.CorbaOpenSession with the appropriate parameters

Getting an Engine Object

You must either already know the name of the engine you want to open a session with, or your application must be able to obtain it (for example, from a system variable).

In addition, the engine must be in the current iIS environment or in an environment connected to the current one. (See "About iPlanet UDS Environments" on page 40 for more information.)

If you want to retrieve a single engine object, call CorbaWFDirectory.CorbaFindEngine. For example, the following Java code gets a CorbaWFEngine object for the engine named *Claims* in the current environment:

```
CorbaWFEngine myEngine =
        DirectorySO.CorbaFindEngine(name="Claims");
```

If you want a list of all the engines registered in the current environment, call the CorbaWFDirectory.CorbaGetAvailableEngines method. This method finds all the engines registered with the name service for the active environment. You can then find the engine you want from the array of engine objects this method returns. You find the engine by calling the GetName method of each engine object in the array and testing the return value. For example, the following code would find the engine named Claims in the list of engines either in the active environment or in environments connected to the active environment.

The code tests to see if an engine object was returned. If so, it tests whether the engine object name is "Accounting". If so, there's a match and the counter is set to a value that ends the for loop. If an engine is NIL, it's the first one in the array and there's no engine available:

```
String engineName = "Claims";
CorbaWFEngine myEngine;
CorbaWFEngine[] availEngines =
        CorbaWFDirectory.CorbaGetAvailableEngines();
integer engineCount = availEngines.length;

for (i=0;i<engineCount;i++){
   if (availEngines[i] != null){
      if equalsIgnoreCase(availEngines[i].GetName(),engineName){
          myEngine = availEngines[i];
          i = engineCount;
      } // end if GetName
  } // end if != null
  else {
      System.out.println ("No engine available");
      i = engineCount;
  } // end else
} // end for loop
```

NOTE Both methods contact the iPlanet UDS name service on the server. CorbaFindEngine verifies that the engine is registered. (If the engine is not registered, the method returns a null object.) CorbaGetAvailableEngines returns all registered engines.

It is possible that the engine your program chooses will not be running when it makes the subsequent CorbaWFEngine.CorbaOpenSession call, in which case the call will fail. However, your program can retry the call if you know that the engine will eventually be started.

Setting Up User Profile Information

When you start a new session, you must supply user profile information about the user to the engine. Before you can do so, you must have the name of a user profile that is registered with the engine, and you might need to know something about its structure if the user profile has been customized. (Some coordination with the person who creates user profiles in the User Profile Workshop is necessary. For information on the User Profile Workshop, see the *iIS Process Development Guide*.)

Validating the User

The engine uses the user profile name that comes in from your application to find a registered user profile that it can populate with your user information. When the engine has successfully created the user profile object and populated it, the engine passes this information to the site-defined Validation.ValidateUser method, which verifies it against the information about the user in the organizational database. (For more information, see the description of the Validation Workshop in the *iIS Process Development Guide*.)

The purpose of this validation is to ensure that the user is entitled to the roles and any other user attributes they specify in the client application, because the engine compares this information to the assignment rules of activities to determine which sessions get which activities. For example, if the user tries to sign on as a manager but does not have that role listed with their name in the organizational database, the engine detects the mismatch, and the call to CorbaOpenSession fails with an exception.

Supplying User Information

Before it passes a user profile name to the engine, the client application must determine the user name and password of the user, both of which are probably entered by the user when starting the client application. The client application can also prompt for any roles or other specially defined user information required to work with the activities this application will display, or it can let the ValidateUser method populate the user profile with this information from the organization database. Because ValidateUser must be developed and registered before the client application is developed and it must work with all client applications, how much information to send the engine is a design decision that must be agreed upon in advance.

When your client application has the necessary information from the user, you instantiate a new WFUserProfileDesc object and fill in its attributes as necessary. For example, the following code instantiates the object *myUser* and initializes it with the user name *jfleuri*, the user profile name *AcmeUserProfile1* (for a user profile registered with the engine as AcmeUserProfile1), and the roles *Manager* and *Claims Processor*:

```
WFUserProfileDesc myUser = theEngine.GetUserProfile();
myUser.SetProfileName("AcmeUserProfile1");
myUser.SetUserName("jfleuri");
myUser.AddRole("Manager");
myUser.AddRole("Claims Processor");
```

NOTE The user name is case-sensitive.

Customizing a User Profile

The default definition of the user profile includes only a name and a set of roles. However, if the user description is more complicated (for example, it indicates a specific level of budget approval authority), someone at your site must create a new definition of the user profile in the User Profile Workshop, distribute the new user profile library, register it with the appropriate engines, and notify you so you can use the appropriate profile name and attributes in your WFUserProfileDesc object.

For example, users of a client application at Acme, Inc., in addition to roles, have a maximum budget authority, which is defined in AcmeUserProfile2 as the integer attribute *maxBudget*. The following code would initialize an object of this class for the user with user name *hgretel* in the role *manager* with a maximum budget authority of $5,000:

```
WFUserProfileDesc myUser = theEngine.GetUserProfile();
myUser.SetProfileName("AcmeUserProfile2");
myUser.SetUserName("hgretel");
myUser.AddRole("manager");
myUser.SetInteger("maxBudget",5000);
```

The Engine and the WFUserProfileDesc Object

When you pass a WFUserProfileDesc object as a parameter to CorbaWFEngine.CorbaOpenSession, the engine finds the appropriate user profile definition and passes the description to the Validation.ValidateUser method defined in the Validation Workshop. This method is written by someone at your site to interpret those attributes correctly, and, like the user profile, is registered with the engine.

Suspended and Reconnected Sessions

If the session is interrupted by a network or engine failure (because the engine has not been set up to be fully fault-tolerant) or by intervention from a system administrator, you can have the engine either suspend the session (because the user might be in the middle of some work) or terminate it (thus losing whatever work might be in progress). Allowing a session to be suspended means that any work the user was doing can be resumed at a later date. Requiring that a session be terminated when it loses contact with the engine means that the work the user was doing will be lost, any process attributes that were locked will be unlocked, and the corresponding activity will be made READY again, available for other users to perform.

When a session is put in a SUSPENDED state, the following actions occur:

- Any activities that are currently ACTIVE are processed according to each activity's defined suspend action (or the overriding value specified in the suspendAction parameter of WFActivity.StartActivity or WFSession.StartQueuedActivity). ACTIVE means that the activity was started by the client application's making a call to either WFActivity.StartActivity or CorbaWFSession.StartQueuedActivity. For more information, refer to the iIS online help.

  If the defined action in the process definition is *remove* (or the suspendAction parameter specified WFActivity.REMOVE), the activity is removed from the session's activity list. Depending on the specification in the process definition, the activity is either aborted or rolled back to the READY state (rolling back includes closing the activity's attribute accessor with ROLLBACK).

  If the defined action is *retain* (or the parameter specified WFActivity.RETAIN), the activity is retained on the session's activity list in the ACTIVE state.

- Any activities that were on the session's activity list that were not chosen by the user are in READY state. The engine takes these activities off that session's activity list because the user can no longer choose any of them.

Disconnect Action Values

To determine whether a session is suspended or terminated if the connection to the engine is unexpectedly interrupted, set the controls parameter of CorbaOpenSession with one of the following Disconnect Action values:

CorbaWFSession.SUSPEND_ON_DISCONNECT
CorbaWFSession.TERMINATE_ON_DISCONNECT

If you do not set one of these Disconnect Action values, the default action is to terminate the session if the CorbaWFSession object determines that the engine is disconnected and cannot be reconnected.

When you start a session, you can indicate if you want the engine to reconnect to a previously suspended session, if there was one, or to discard any previously suspended sessions and start fresh. If you reconnect to a suspended session, the user can gain access to work that was in progress when the session was suspended.

Reconnect Action Values

To determine whether a session reconnects to a suspended session or starts up as a new session, set the controls parameter of CorbaOpenSession with one of the following Reconnect Action values:

CorbaWFSession.RECONNECT_ALLOWED
CorbaWFSession.RECONNECT_PROHIBITED

If you do not set one of these Reconnect Action values, the default action is to reconnect to a suspended session.

For more information, see "Handling the Connection With the Engine" on page 149.

Offered Activity Notifications

Whenever an offered activity becomes READY, the engine compares its assignment rules to the user profile objects of all active sessions and adds the activity to the activity lists of the sessions that match. However, if a session is heads-down, it works only with queued activities, and allowing the engine to test the session's user profile object against the assignment rules of offered activities is a waste of system resources, because the session never gets any offered activities.

To save system resources, an application logging on as a heads-down session should set the controls parameter of CorbaOpenSession with the Activity Offers value CorbaWFEngine.OFFERED_IGNORED. If this value is not set in the controls parameter, the engine by default will evaluate the session whenever an offered activity becomes READY.

Opening a Session

When you have determined the conditions for your client program's session with the engine, you can open the session by calling the CorbaWFEngine object's CorbaOpenSession method. The following code starts a new session with the engine *Claims*. It uses the sample code from the previous sections to set the parameters and starts up with the following session controls set:

- If the engine disconnects, the session is suspended.
- The session is allowed to reconnect to a suspended session.

- ActivityListUpdate events are not sent (see "Using Activity Update Caching" on page 140 for a code sample that uses ActivityListUpdate).
- The session is evaluated whenever an offered activity becomes READY.)

Code Example 4-2 Opening a session (CORBA)

```
import java.io.*;
import CorbaWFApi.*;
import WFClientLibrary.*;
import OFCustomIF.*;
...
public static void main(String args[]) throws
                                            InterruptedException
{

  // Read IOR file indicated by first command line option
  String fileName = args[0];
  FileInputStream x = new FileInputStream(fileName);
  DataInputStream input = new DataInputStream(x);
  String ior = input.readLine();

  // Initialize Visigenic client ORB
  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

  // Convert IOR string to object
  org.omg.CORBA.Object obj = orb.string_to_object(ior);

  // Cast Visigenic object into iIS object
  CorbaWFApi.CorbaWFDirectory DirectorySO =
        CorbaWFApi.CorbaWFDirectoryHelper.narrow(obj);

  // Get the Claims engine
  CorbaWFEngine myEngine =
        DirectorySO.CorbaFindEngine("Claims");

  // Set up profile information
  WFUserProfileDesc myUser = myEngine.GetUserProfile();
  myUser.SetProfileName("AcmeUserProfile1");
  myUser.SetUserName("jfleuri");
  myUser.AddRole("Manager");
  myUser.AddRole("Claims Processor");
  String nameForSession = "MySession";

  // Open the session
  CorbaWFSession mySession = myEngine.CorbaOpenSession (
          NameForSession,
          myUser,
          password,
          CorbaWFSession.RECONNECT_ALLOWED +
              CorbaWFSession.SUSPEND_ON_DISCONNECT);
  ui4 openStatus = mySession.GetOpenStatus();
  ...
}
```

If the session is unable to start, the CorbaOpenSession call throws an exception. Otherwise, the session object is created and immediately usable.

To determine if the session reconnected to an existing suspended session or started as a new session, the last line of code calls GetOpenStatus. The next thing to do would be to test if the returned value was CorbaWFSession.RECONNECTED_SESSION or CorbaWFSession.NEW_SESSION. If the session has reconnected to a previously suspended session, there might be one or more activities in the ACTIVE state that the client application must display to the user. If not, the session is a new one, and the client application can simply continue to get activities and display them to the user.

Whether the session reconnected or is a new session, the client application must indicate what work is available to the user, as described in the next section.

Indicating What Work Is Available

Once the session has been established with the engine, the client application must determine what work is available to it. How it does this depends on whether the client application is a heads-up or a heads-down application, as follows:

- If the client application is a heads-up application, it is likely to display a work list to the user, as described next in "Getting and Handling Offered Activities."
- If the client application is a heads-down application, it retrieves the next queued activity available to it, as described in "Getting and Handling Queued Activities" on page 143.

NOTE The following sections have examples that use attribute accessors (WFAttributeAccessor) to work with process attributes. For an introduction to process attributes and a description of how WFAttributeAccessor objects can be obtained, see "Working with Process Attributes" on page 42.

Getting and Handling Offered Activities

For each session, the engine maintains a *session activity list*. This list has on it activities that are in READY state that the engine has offered to the session (because the session user profile matches an assignment rule for each of the activities). In addition, this list includes any activities that the session has made ACTIVE.

To get this list of activities from the engine, you declare an object to receive the return value of the method call, then call CorbaWFSession.GetActivityList, as follows.

```
WFActivity[] activityList =
        mySession.GetActivityList(CorbaWFSession.AL_EVENTS_OFF);
```

NOTE Calling this method with the parameter set to CorbaWFSession.AL_EVENTS_OFF prevents the engine from sending ActivityListUpdate events to the session, which, since the client is a CORBA/IIOP client, cannot receive iPlanet UDS events anyway. Turning these events off reduces engine overhead and lets it run faster.

Building a Work List

Each activity object returned by GetActivityList has associated with it an application dictionary entry composed of an activity description, an application code, and, if the process designer defined them, a set of process attributes. You can construct a work list by extracting this information from each activity. Using the WFActivity methods GetName and GetActivityDescription, you can get the name and description of the activity.

It is likely that the work list you display for the user will include supplemental information for each entry (such as a customer name, purchase amount, order number, and so on), which you can obtain by calling each activity's GetAttributeAccessor method to obtain a WFAttributeAccessor. You can then use the attribute accessor's GetNames and GetValue methods to get the names and values of these process attributes.

NOTE If the return value of GetAttributeAccessor is null, there is no attribute accessor associated with the activity.

The following code sample shows a method that builds a work list display from an array of WFActivity. It uses objects such as ExpenseReportListView that are defined elsewhere and assumes that certain attributes, Status, ExpenseRptID, and Priority, are associated with each returned activity. After getting the list of activities, it gets the number of activities in the list; it then loops through the array if there are any members:

Code Example 4-3 Building a work list display (CORBA)

```
void BuildList(CorbaWFSession mySession) {
// Initialize listview field.
// This field will display the work list.
ExpenseReportListView listView;
listView.IsFilled = true;
listView.IsFolder = true;
listView.IsOpened = true;

// Get the activity list.
WFActivity[] activityList;
integer maxActs;
WFAttributeAccessor attribAccessor;
activityList = mySession.GetActivityList();
maxActs = myActivityList.length;
for (i=0;i<maxActs;i++) {
  attribAccessor = myActivityList[i].GetAttributeAccessor();

  // Get values for this row in worklist
  // from attribute accessor
  listView[i].Status = attribAccessor.GetInteger("Status");
  listView[i].ExpenseReportID =
        attribAccessor.GetInteger("ExpenseReportID");
  listView[i].Priority = attribAccessor.GetBoolean("Priority");
  listView[i].WorkItem = myActivityList[i].GetName();
}
```

Maintaining a Work List

You can use one of the following two approaches for keeping a session work list display updated:

- poll the engine periodically
- respond to events posted by the engine

Polling the Engine

Using the polling approach, you periodically call CorbaWFSession.GetActivityList to obtain the entire activity list for the session. You can use this list either to completely refresh the display or you can merge the changed elements of the list with the corresponding displayed elements, depending on which is faster.

Using Activity Update Caching

A feature called *activity update caching* enables a CORBA-compliant language application to respond to iPlanet UDS events without having to repeatedly poll the engine. To use activity update caching, you start a session as you normally do, and then turn on activity update caching in the client session. You then create a client variable of type WFSessionUpdate_struct[] and use it when you call CorbaWFSession.GetSessionUpdates to receive lists of activity updates.

Update caching works best if CorbaWFSession.GetSessionUpdates is called in a thread, allowing the main client thread to continue with other processing. However, on systems that do not have threading, it is possible to use this method, but call it without waiting for a return value so the method never gets blocked.

➤ **To use activity update caching**

1. Connect to an engine and start a session (for sample code, see "Opening a Session With the Engine" on page 129).
2. Get the initial list of activities, turn on update caching, and display the work list. For example:

```
WFClientLibrary.WFActivity[] myUpdateList;
myUpdateList = mySession.GetActivityList (
    CorbaWFSession.AL_EVENTS_ON |
    CorbaWFSession.SAVE_UPDATES_ON);
```

Calling GetActivityList with these two settings turns on activity update caching. By default, the number of updates that the engine will cache is twenty. If you want to change the default number, you can call CorbaWFSession.SetUpdateLimit, described in the iIS online help.

When you are doing activity update caching, whenever you call GetActivityList, you must specify both that you want events turned on and that the engine is to cache them.

3. Declare a variable of type WFClientLibrary.WFSessionUpdate_struct[] and start a thread that uses the variable to pick up the latest group of activity updates. The code in the thread calls CorbaWFSession.GetSessionUpdates, as shown in the following sample code:

```
WFClientLibrary.WFSessionUpdate_struct[] updateList;
int i;

// Start a thread, if supported
. . .
updateList = mySession.GetSessionUpdates(true);
if (updateList != null)
{
    for (i = 0; i < updateList.length; i++)
    {
        // Do something with each change
        // Could update worklist here
    }
}
```

In this sample code, GetSessionUpdates indicates that it will wait until the engine has activity updates. Having the method wait requires that the call be in a separate thread if the main thread is to continue processing.

You can cancel a waiting GetSessionUpdates call by calling CorbaWFSession.CancelGetSessionUpdates from another thread of the same session.

There is no code in the sample code's for loop. One thing you could do here is to update the work list with each activity encountered.

If you cannot create a separate thread for the GetSessionUpdates call, call it with wait set to false. You must then periodically call it again to get the latest session updates.

Update Limit

The session starts with *update limit*, the number of activity updates saved for this session, set to a default value of twenty. You can set the update limit to a different value by calling CorbaWFSession.SetUpdateLimit.

If the total number of updates for this session exceeds this limit, the client session enters an *overrun* state, adds an entry to the update list indicating that the limit has been exceeded (a WFSessionUpdate object with SessionChange value of CorbaWFSession.SESSION_WORK_OVERRUN), and removes all activity updates from the list. Until you reset the client session's state with a call to CorbaWFSession.GetSessionUpdates, the client session saves no activity updates for the session.

If your code detects that the update limit has been exceeded (by encountering a WFSessionUpdate object in the WFSessionUpdate array with a SessionChange value of CorbaWFSession.SESSION_WORK_OVERRUN), your client application must call CorbaWFSession.GetActivityList to get all activity changes, then clear the work list and redisplay it.

When you call CorbaWFSession.GetSessionUpdates and specify that the method will not wait (the parameter is false), the method returns even if there have been no updates. If there have been no updates, the returned WFSessionUpdate array is null.

If you specify that the method will wait until there is something in the list (the parameter is true), the method could return a null WFSessionUpdate array if one of the following events occurs:

- The connection to the host is lost.
- The client application's main thread suspends or closes the session.
- The client application's main thread calls CorbaWFSession.GetActivityList.
- The client application's main thread calls CorbaWFSession.CancelSessionUpdateWaiting.

Starting an Offered Activity

In a heads-up application, when the user chooses a work item to work on, the client application calls WFActivity.StartActivity for the associated activity (see the iIS online help for more information on this method). Call this method synchronously (wait for it to return) and use it in a try block to ensure that the client application has gotten the activity, because it is possible that another client application has already chosen it. If the engine can change the activity's state to ACTIVE and give it to your session, it does so, and the method returns successfully. If the activity has already been taken, a WFActivityListException will be raised by the engine.

The preferred way of working with activities is to specify their behavior in the process definition and not override these settings when you start them from a client application. (The less specific information about a process definition the client application must carry, the freer the process designer is to make design changes.) Therefore, it is usually best to call StartActivity without specifying process attributes and pick up the default set of process attributes associated with the activity in its application dictionary entry.

Getting and Handling Queued Activities

A heads-down application presents the user with the next item of work: It does not allow the user to choose from a work list as a heads-up client application does. This *next item of work* is represented in a process definition as a queued activity. To support heads-down applications, the engine maintains prioritized queues of queued activity instances. Each queue is composed of “like” activities, queued activities that have the same name and process definition. The highest priority activity is at the top of the queue, ready to be assigned to the next eligible session that asks for it.

To determine which sessions are eligible, the engine attaches the activities’ assignment rules to the queue. A session requesting an activity in the queue must have a user profile matching the requirements of the queue’s assignment rules. If so, it gets access to the queue. (For more information on assignment rules, see the description in the *iIS Process Development Guide*.)

In a heads-down application, when the user is ready for the next item of work, you call your session object’s StartQueuedActivity method. For example, the following code sample retrieves an activity called DoAccounting that is in the ExpRptActng process. If the session gets suspended, the activity remains on the session’s activity list. If there is no activity in the queue, the method returns immediately:

```
CorbaWFAttributeDesc[] attribList = new CorbaWFAttributeDesc[0];
WFActivity curWorkItem = mySession.StartQueuedActivity(
        "ExpRptActng", "DoAccounting",
        CorbaWFSession.USE_DEFINED_SUSPENDED_ACTION,
        CorbaWFSession.IMMEDIATE,
        attribList);
```

NOTE This code sample sets the parameters *suspendAction* to `CorbaWFSession`.USE_DEFINED_SUSPENDED_ACTION and *attributeList* to an empty array of CorbaWFAttributeDesc to prevent them from overriding behavior that should be defined in and controlled by the process definition.

If there is an activity on the queue, the engine retrieves the highest priority activity from the queue, marks it ACTIVE, adds it to the session's activity list, and returns a WFActivity object that describes the activity. (See "Dealing With An Empty Activity Queue" on page 145 for information on how to handle a queue with no activities in it.)

Just as for StartActivity, a WFAttributeAccessor object is built as part of the StartQueuedActivity operation if one is specified by the activity definition or if a list of CorbaWFAttributeDesc objects is provided in the method's attributeList parameter. If no process attributes are defined for the activity and none are specified in the StartQueuedActivity call, the attribute accessor returned is null. The following code retrieves the attribute accessor for the activity curWorkItem that has been returned to the client application:

```
WFAttributeAccessor curAttribs;
curAttribs = curWorkItem.GetAttributeAccessor();
```

You can use the attribute accessor's GetName and Get*Type* methods (such as GetInteger) to get the names and values of the attributes associated with the activity. For example, use GetInteger if the data type of the attribute you want to access is of type integer.

The following sample code starts a queued activity and tests the return value to see if an activity was retrieved. If so there's an attribute accessor that can be used to obtain the associated process attribute information. (See the previous section for a code sample that gets attribute values from an accessor.)

```
CorbaWFAttributeDesc[] attribList = new CorbaWFAttributeDesc[0];
WFActivity qAct = mySession.StartQueuedActivity (
           "ERPD", "DoAccounting",
           CorbaWFSession.USE_DEFINED_SUSPENDED_ACTION,
           CorbaWFSession.IMMEDIATE, attribList);
WFAttributeAccessor accessor;
if (qAct == null)
  println ("No expense reports available.");
else
  accessor = qAct.GetAttributeAccessor();
```

Dealing With An Empty Activity Queue

If there is no activity on the queue, your client application can either wait until one is available or return immediately and try again later. The previous code sample returns immediately. If it had specified an emptyAction of CorbaWFSession.WAIT, your client application would wait until a DoAccounting activity instance was added to the queue.

You can control how long your client session waits by starting a separate timed task to handle the call to StartQueuedActivity. If the method call does not return before the timeout, your client application can call the CancelStartQueuedActivity method to cancel the StartQueuedActivity call. In this case, StartQueuedActivity returns with a null object.

Completing Work On an Activity

When the user has finished working on the work item, set any process attributes that need setting with WFAttributeAccessor.Set*Type* methods (for example, SetInteger), then call WFActivity.CompleteActivity to tell the engine you are done. The engine commits all changes and the process continues as indicated by the process definition.

Rolling Back an Activity

If work was not completed and the user wants to quit without applying updates, you can call WFActivity.RollbackActivity to roll back changes the user has made and reset the activity's state to READY, making it available to other users. What the engine rolls back are changes to process attributes associated with the activity. Any changes that your client application has made to site files and any transactions that it has started with site databases must be explicitly undone or rolled back by your client application.

Aborting an Activity

Another option is to call WFActivity.AbortActivity to abnormally terminate the activity. Do not exercise this option unless something drastic has happened and you want to risk aborting the process. Unless the process definition explicitly handles aborting the activity, the process will be aborted.

Creating a Process Instance

To start a new instance of a process definition, you call CorbaWFSession.CreateProcess, indicating the name of the process definition in the processName parameter. The user must be in a role that can create a new instance of the process for this method to succeed. (In other words, the process definition must have an assignment rule that gives a user with the same role as the user of this client application permission to create a new instance of the process.)

If you also want to initialize process attributes when you start the process instance, you can put the list of process attributes and their values in an array of CorbaWFAttributeDesc and pass this array in the attributeValues parameter of CreateProcess. This parameter is of type WFAttributeDesc, but accepts an array of CorbaWFAttributeDesc because this class is a subclass of WFAttributeDesc.

NOTE When the engine gets a request to create a new process instance, it ignores the LockType attribute of any process attributes sent to it in the attributeValues parameter. For that reason, you need to set only the Name and Value attributes of each CorbaWFAttributeDesc object in the array.

The following method illustrates how to build an array of CorbaWFAttributeDesc:

```
CorbaWFApi.CorbaWFAttributeDesc[] attrs =
         new  CorbaWFApi.CorbaWFAttributeDesc[3];
    attrs[0] = myEngine.GetCorbaWFAttributeDesc(
                   "ExpenseReportID",
CorbaWFAttributeDesc.NO_LOCK);
    attrs[0].SetInteger(1001);
    attrs[1] = myEngine.GetCorbaWFAttributeDesc(
                   "TotalAmount", CorbaWFAttributeDesc.NO_LOCK);
    attrs[1].SetDouble(333.33);
    attrs[2] = myEngine.GetCorbaWFAttributeDesc(
                   "Status", CorbaWFAttributeDesc.NO_LOCK);
    attrs[2].SetInteger(1);
```

The following code creates a process instance by using the previously built attribute accessor array:

```
String procName = "ExpenseReport";
WFClientLibrary.WFProcess myExpRpt = mySession.
        CreateProcess(procName, attrs);
```

Setting Process Recovery Level

By default all current process state information is recovered for a process in the event of engine or system failure. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. If appropriate, the recovery level can be altered so that the process instance is simply restarted with no recovery of process state information; another alternative you have is specifying no recovery of the process.

The default behavior of full recovery applies under the following conditions:

- no other value is specified for the process recovery level in the Process Definition property inspector
- you do not override the default in the client application code
- the engine is configured for state recovery

If logging options for the engine are turned off, the process definition default does not override the engine setting. For details on setting the recovery level in the Process Definition Workshop, see the *iIS Process Development Guide*. For details on engine configuration, see the *iIS Process System Guide*.

You can use the recoveryInfo parameter of the CreateProcess method to control whether or not the recovery level value specified in the process definition is used when creating the process. This method returns a WFProcess object for each newly created process instance. If either no value or the value WFProcess::RCVR_NORMAL is passed in for recoveryInfo, the process is created using the recovery level specified in the process definition. If another value is passed in, that value overrides the recovery level value specified in the Process Definition property inspector. The possible other values are WFProcess::RCVR_FULL, WFProcess::RCVR_NONE, or WFProcess::RCVR_PROCESS_ONLY.

Closing a Session

When the user wants to log off, your client application must close the session with the engine. You call the CorbaWFSession.CloseSession method and, depending on whether the user has completed work or not, indicate with the newState parameter that the session is to be either terminated or suspended.

Terminating the Session

If the user has completed all work and the engine has successfully closed any associated activities, you can terminate the session by calling CloseSession with the newState parameter set to CorbaWFSession.TERMINATED. Terminating the session causes the engine to remove all READY activities from the session's activity list.

Another effect of terminating the session is that the engine aborts any ACTIVE activities. If the user has any work in progress, it is represented by at least one ACTIVE activity, and all the work is lost at this point. If the user indicates that the session is to be terminated and there is an ACTIVE activity, it would be good to let them know the consequences and suggest suspending the session or finishing all work as alternatives.

Suspending the Session

If the user has any work in progress, you typically would take whatever steps are necessary to save the current work and then suspend the session by calling CloseSession with the *newState* parameter set to CorbaWFSession.SUSPENDED. Suspending the session causes the engine to remove all READY activities from the session's activity list, but leave any ACTIVE activities on the suspended session's activity list. When the user logs in at a later time, you can open the new session and tell the engine to reconnect to the suspended session by indicating CorbaWFSession.RECONNECT_ALLOWED in CorbaWFEngine.CorbaOpenSession's *controls* parameter.

Handling the Connection With the Engine

As described in "Suspended and Reconnected Sessions" on page 133, when the client application starts a session (by calling CorbaWFSession.CorbaOpenSession), it is connected to an engine through the client application's CorbaWFSession object. If you call this object's SetPingInterval method, it monitors the client application's connection to the engine in the background, sending ping messages to ensure that the connection is still up, and handling reconnection if the connection goes down (either the network connection is lost or the engine fails). By default, the pinging is off. The default interval is set to 2 seconds. You can turn pinging on and change the interval by calling CorbaWFSession.SetPingInterval or CorbaWFSession.SetRetry. For more details, see the iIS online help.

Losing the Connection

If pinging is off, the WFSession object does not test the connection with pings. However, if the client calls a function that contacts the engine and the engine is down, the session object does detect the disconnection. If pinging is off, the client application must disconnect and reestablish the session later.

If pinging is on and the connection to the engine goes down, the session object automatically attempts to reestablish the connection. If it is able to do so within a time period that you can set with the WFSession.SetRetry method (by default, 20 retries at intervals of 2 seconds), the session object automatically reconnects with the engine, and the session is able to continue.

If the session object is unable to reestablish the connection within this time period (for example, the engine has failed and there is no backup engine that can be automatically restored, or the engine takes an especially long time to come back up), it raises a WFException with LostContact as the reason. At this point, the client application must disconnect and reestablish the session later.

It is possible that the connection with the engine could go down and come back up again, and your client application would never be notified because the user is working solely on the client side, performing the work associated with an activity. What happens in this instance is that your client application's WFSession object detects that the engine connection has gone away because the engine did not respond to one of its periodic ping messages. The WFSession object then begins to try to restore the connection and does so successfully before it receives any method calls that connect with the engine.

It is also possible that the connection with the engine could go down, but not come back up again before the client application makes a call that connects to the engine. In this case, when the client session makes the method call (for example, WFActivity.StartActivity) WFSession sends a WFSession.EngineDisconnected event to the client application, indicating that it has detected that contact has been lost with the engine, but it is trying to reestablish the connection. One way for your client application to handle the event is to wait for an EngineReconnected event and make the call again. (For a description of these events, see the iIS online help.

What does the engine do in these circumstances? It maintains its own internal representation of each session and tracks the activities offered to or being used by each session. It handles disconnections as follows:

- If the engine goes down and then recovers, after it recovers, it marks its session objects SUSPENDED until the clients' session objects reestablish connection. Any activities in an ACTIVE state remain that way and remain connected to their sessions. Any attributes with locks remain locked.
- If the engine does not go down, but it loses contact with a session object (for example, the network node for that session goes down, or the client application crashes), the engine marks its corresponding session object either SUSPENDED or TERMINATED, depending on the setting of the autoDisconnect parameter to the WFSession.OpenSession method. (See "Suspended and Reconnected Sessions" on page 133. That section also describes how active and offered activities are handled.)

Reestablishing the Connection

As with starting a new session, the client calls CorbaWFEngine.CorbaOpenSession to restart a suspended session. The controls parameter must include the value CorbaWFSession.RECONNECT_ALLOWED. When the session is open, the next thing the client does depends on whether it is a heads-up or a heads-down application.

For a heads-up application, on the first call to GetActivityList, if there are any ACTIVE activities in the returned list and the return value of CorbaOpenSession was CorbaWFSession.SUSPENDED_SESSION, the user was working on these activities previously when the session was suspended. The client application can test the current values of attributes and prompt the user for any information needed to continue work on the activities.

Normal behavior for a heads-down client application after it starts a completely new session is to do a StartQueuedActivity call to get its first activity. However, when reconnecting to a suspended session, a heads-down client must first call CorbaWFSession.GetActivityList to determine if it already has an ACTIVE activity left over from when its session was suspended. If there is an ACTIVE activity on the list, the client application can deal with it in the same manner as a heads-up client does (see previous paragraph).

If you do not want to reestablish connection with the suspended session, you set the reconnectAction parameter to CorbaWFSession.RECONNECT_PROHIBITED. The engine terminates the suspended session, aborting any ACTIVE activities, and starts a new session for your client application.

Recovering Attribute Accessors

After you reconnect a session to the engine, you must call CorbaWFSession.GetAtrributeAccessors to see which accessors your recovered session has open, if any. This method returns an array of attribute accessors. You can call the WFAttributeAccessor methods GetName, GetID, and GetProcessID to determine which attributes belong to which process instances.

Chapter 5

Building a JavaBeans Client

This chapter describes how to use the iIS JavaBeans process client API to write a client application for an iIS enterprise system.

This chapter starts with an architectural overview and then explains how to install and configure your system to use the iIS JavaBeans client API. Some general guidance on using the JavaBeans API is also provided.

The rest of this chapter describes typical tasks your iIS process client application is likely to perform with the JavaBeans API, such as opening a session and getting a list of available activities.

See the first chapter of the *iIS Process Development Guide* and Chapter 1, "Building an iIS Process Client Application" of this guide for more information.

The JavaBeans Client API

The iIS JavaBeans process client API enables rapid development of Java applications that access the iIS process engine.

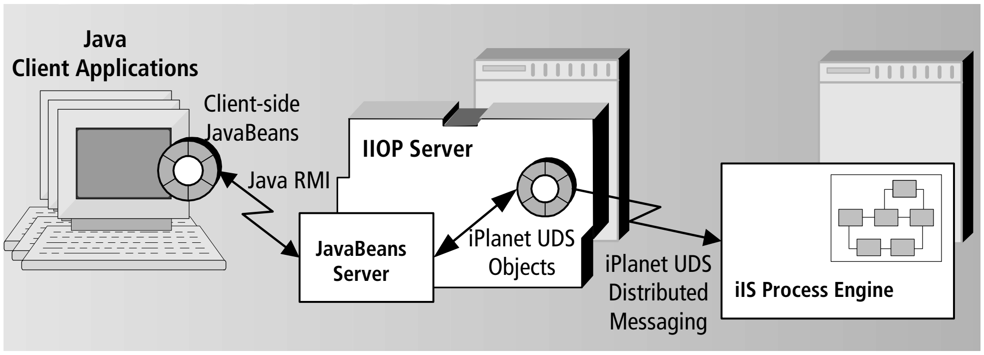
The JavaBeans client API provides a set of non-visual JavaBeans to allow your Java application to interact with the iIS engine. Each iIS JavaBean supplies a variety of methods and properties that you use to send requests to and receive information from the engine. You either manipulate these methods and properties programmatically or, in some cases, use the event handling capabilities of your Java application builder.

The JavaBeans client API components provide access to the entire iIS process engine API, and adhere to JavaSoft standards and conventions.

Architectural Overview

The following diagram illustrates the architecture of the iIS JavaBeans client API.

Figure 5-1 iIS JavaBeans API Architecture



The API provides both client and server beans. Client beans send client requests to the server task using RMI (Remote Method Invocation), locating the JVM using RMI directory services.

The Java server application, running its own JVM, uses server beans and the iIS CORBA/IIOP client API to relay client requests to the iIS IIOP server, with appropriate data type conversion. The IIOP server communicates with the iIS process engine, and returns responses to the client with appropriate data type conversion and exception handling.

Installing and Configuring the JavaBeans Client API

The iIS JavaBeans client API requires both client and server installations. A typical iIS JavaBeans configuration has several clients using a single server. Larger configurations require more than one server.

Setting Up a Server

An iIS JavaBeans server must have iIS, JDK (Java Development Kit) 1.2, and an ORB (Object Request Broker) installed.

- For details on installing iIS, see the *iIS Installation Guide*.
- For information on obtaining and installing the JDK, see the JavaSoft web site (www.javasoft.com/products/jdk/1.2/).
- Information on supported ORBs is maintained on the platform matrix on the Support/Service page, which is available from (www.sun.com/forte).

Working with Third Party ORBS

After installing iIS, the JDK, and a third party ORB, you must generate the appropriate Java code for the ORB and then compile the code. This step is not necessary if you are using Java IDL from Sun.

VisiBroker

The following procedure explains how to set your CLASSPATH and generate Java code using VisiBroker.

➤ **To generate VisiBroker Java code**

1. Set the CLASSPATH environment variable on the server node. For example, on UNIX:

```
$VBROKER/lib/vbjorb.jar:.:$VBROKER/lib/vbjtools.jar:\
$VBROKER/lib/vbjapp.jar:$FORTE_ROOT/install/lib/java/forte.zip:\
$JAVA_HOME/lib/classes.zip:\
$FORTE_ROOT/install/lib/java/conductor.jar
```

NOTE The CLASSPATH is one continuous string. Do not type the trailing backslash indicated on the first three lines of the example.

VBROKER and JAVA_HOME are environment variables representing the roots of your VisiBroker and JDK installations.

2. Generate the ORB code from the iIS IDL files. In a working directory for the server, do the following:

```
cd /jbserver
cp $FORTE_ROOT/install/inc/idl/framewor.idl .
cp $FORTE_ROOT/userapp/wfcorbaa/cl1/conductr.idl .
$JAVA_HOME/bin/java com.visigenic.vbroker.tools.idl2java \
  -incl_files_code -no_comments -no_tie -no_examples conductr.idl
```

   This generates the Java code needed by VisiBroker.

OrbixWeb

The following procedure explains how to set your CLASSPATH and generate Java code using OrbixWeb.

➤ **To generate OrbixWeb Java code**

1. Set the CLASSPATH environment variable on the server node. For example, on Windows platforms:

```
%JAVA_HOME%\lib\classes.zip;\
%FORTE_ROOT%\install\lib\java\conductor.jar;\
%FORTE_ROOT%\install\lib\java\forte.zip;\
%ORBIXWEB_HOME%\classes
```

NOTE The CLASSPATH is one continuous string. Do not type the trailing backslash indicated on the first three lines of the example.

   JAVA_HOME and ORBIXWEB_HOME are environment variables representing the root of your OrbixWeb and JDK installations.

2. Generate Java code for the ORB from the iIS IDL files. In a working directory for the server, do the following:

```
cd \jbserver
copy %FORTE_ROOT%\install\inc\idl\framewor.idl
copy %FORTE_ROOT%\userapp\wfcorbaa\cl1\conductr.idl
%ORBIXWEB_HOME%\bin\idl -N -jO . conductr.idl
```

   This generates the Java code needed by OrbixWeb.

Compiling Java ORB Code

This section shows how to compile the Java ORB code generated by either VisiBroker or OrbixWeb. This step is not necessary if you are using Java IDL.

➤ **To compile the Java ORB code**

1. Use your Java compiler to compile the generated VisiBroker or OrbixWeb code. The following example is for Windows platforms:

```
cd \jbserver\Framework
cd Framework
%JAVA_HOME%\bin\javac -J-mx32m -d .. *.java
cd ..\OFCustomIF
%JAVA_HOME%\bin\javac -J-mx32m -d .. -classpath \jbserver; \
%CLASSPATH% *.java

cd ..\WFCustomIF
%JAVA_HOME%\bin\javac -J-mx32m -d .. -classpath \jbserver; \
%CLASSPATH% *.java

cd ..\WFClientLibrary
%JAVA_HOME%\bin\javac -J-mx32m -d .. -classpath \jbserver; \
%CLASSPATH% *.java

cd ..\WFCorbaApi
%JAVA_HOME%\bin\javac -J-mx32m -d .. -classpath \jbserver; \
%CLASSPATH% *.java

cd ..
```

   JAVA_HOME is an environment variable representing the root of your JDK installation.

Starting the iIS JavaBeans Server

Before starting the iIS JavaBeans server, you must first start the iIS IIOP server on the node running the JavaBeans server. You then start the JavaBeans server, implementing a security policy defined in a separate Java security policy file, called java.policy in these examples. A simple policy file is listed below:

```
grant
{
  Permission java.security.AllPermission;
};
```

NOTE This policy is useful in development, but inappropriate for production environments. For more information on Java security, refer to the JDK program, policytool.

➤ To start the JavaBeans server

1. If you are using the Java IDL from Sun, modify your CLASSPATH, as indicated in the following example for Windows platforms:

```
%FORTE_ROOT%\install\lib\java\forte.zip:\
%JAVA_HOME%\lib\classes.zip:\
%FORTE_ROOT%\install\lib\java\conductor.jar
```

NOTE This step is not necessary if you are using a third party ORB.

The CLASSPATH is one continuous string. Do not type the trailing backslash indicated at the end of the first two lines of the example.

JAVA_HOME is an environment variable representing the root of your JDK installation.

2. Start the iIS IIOP server partition on the JavaBeans server node:

```
cscript
iiopserver start
```

3. Start the JavaBeans server:

```
cd \jbserver
java -Djava.security.policy=java.policy             \
  com.forte.conductor.server.WFServer               \
  $FORTE_ROOT\etc\iiopior\conductr.ior [PORT_NUM]
```

This starts a Java virtual machine that functions as your JavaBeans server.

NOTE The -D option indicates that the Java security policy property is java.policy, a local file.

PORT_NUM represents the port number the server is listening on. If this option is not specified, then the port number defaults to 1099.

Setting Up an iIS JavaBeans Client

You must install the Java 1.2 runtime environment on the client. If the client is used for Java application development, the JDK is required. No iPlanet UDS installation is required on the client.

➤ **To set up a client**

1. Obtain the following Java archive from the JavaBeans server:

   `$FORTE_ROOT/install/lib/java/conductorClient.jar`

2. Copy this archive to your standard location for Java archives.

3. Modify the CLASSPATH environment variable to point to the conductorClient JAR. The following example CLASSPATH is for Windows platforms:

   `%JAVA_HOME%\lib\classes.zip;c:\myjars\conductorClient.jar`

Introducing the iIS JavaBeans Client API

This section provides some general guidance on using the JavaBeans client API.

The complete iIS JavaBeans client API reference is available online in Javadoc format. You can view this documentation with a standard Web browser. Access the file from the iIS CD-ROM or from the following directory on the iIS engine node:

```
%FORTE_ROOT%\install\doc\java\api\com\forte\conductor\package-summary.html
```

Using Methods and Constants

There are no special considerations for invoking iIS JavaBean methods. The following code sample shows a simple method invocation to terminate a workflow session:

```
public class SessionHandler {

// WFSession is already set up
private WFSession    mySession;

public void terminate() {
  try {

      // Use constant value as parameter
      mySession.closeSession( WFSession.TERMINATED );
  } catch( Exception ex ) {
      showError( "Error terminating session: " +
        ex.getMessage();
  }
}
```

The line that invokes mySession.closeSession illustrates a method invocation with a single parameter that takes a constant value. If the member mySession is null, Java generates the appropriate runtime error exception.

Several methods in the JavaBeans API require the use of constants, as indicated in the JavaBeans API online reference information. To give symbolic names to what otherwise would be simple integer values, iIS provides final static fields that define constant values in the JavaBeans API classes. In the preceding code sample, the WFSession class defines TERMINATED as a final static field that represents a constant integer value.

Creating JavaBeans Instances

In most cases, you do not directly construct iIS JavaBean instances. Instead, they are created for you when you invoke the appropriate JavaBeans API methods. The only iIS bean instance you must construct directly is the WFDirectoryService bean. The following code sample shows how to create the directory service bean:

```
WFDirectoryService dirSvc = new WFDirectoryService();
```

If you are using a Java application builder that supports JavaBeans, an alternative is to place an invisible (non-visual) WFDirectoryService bean in a container. The application builder constructs the directory service bean instance when you instantiate the container.

Using Properties

You use JavaBean properties to read or write data items associated with a bean instance. The behavior of the data items depends on the particular property. Many iIS properties can only be read. Some can be both read and written to, and, in one case, a property can only be written to but not read. The following code sample reads some iIS properties:

Code Example 5-1 Reading iIS properties (JavaBeans)

```
public class SessionHandler {

// WFSession is already set up
private WFSession    mySession;

public void onSessionOpened() {
    try {

        // Read session's IsNil property
        if ( mySession.getIsNil() ) {
            showError( "Session is nil!" );
        } else {

        // Read session's Name property
            String name = mySession.getName();
        // Read session's OpenStatus property
            int openStatus = mySession.getOpenStatus();
            // ...
        }
    } catch( Exception ex ) {
        showError( "Error reading session information: " +
            ex.getMessage();
    }
}
```

Writing to a property is similar to reading one, except that the setProperty method is used, as shown in the following code sample. This code uses an engine bean to create an attribute descriptor bean, and invokes the set method to write a property named StringValue:

```
// myEngine is already set up
private WFEngine     myEngine;
// ...
WFAttributeDesc desc = myEngine.createWFAttributeDesc(
    "DocName", WFAttributeDesc.WRITE );
desc.setStringValue( "Winnie the Pooh" );
```

Using Arrays

The iIS JavaBeans client API uses arrays of various iIS bean types. These are used in the same manner as regular Java arrays, as shown in the following code sample:

Code Example 5-2 Using arrays of iIS bean types (JavaBeans)

```
public class DirSvcHandler {
private WFDirectoryService dirSvc = new
    WFDirectoryService();
public void listEngines() {
    try {

        // Declare array of WFEngine beans
        // and fill in the array
        WFEngines[] engines;
        engines = dirSvc.getAvailableEngines();
        int nEngs = engines != null ? engines.length : 0;

        for ( int iEng = 0; iEng < nEngs; iEng++ ) {

            // Get item from the array
            WFEngine eng = engines[ iEng ];
System.out.println( eng.getName() );
        }
    } catch( Exception ex ) {
        showError( "Error getting engine list: " +
            ex.getMessage();
    }
}
```

Creating Arrays

The iIS JavaBeans client API requires that you create arrays of bean instances for various method invocations. A typical array is an array of WFAttributeDesc beans, for filling an attribute list. The following code sample, which declares and fills in an attribute list containing two attribute descriptors, shows how to create this kind of array. The code creates an empty array with space for two WFAttributeDesc beans, and uses the engine bean to create an attribute descriptor bean:

Code Example 5-3 Creating an array of bean instances (JavaBeans)

```
public static WFAttributeDesc[]
fetchList( WFEngine eng ) {
    WFAttributeDesc[] theResult;

    // Create empty array
    theResult = new WFAttributeDesc[ 2 ];
    try {

        // Create attribute descriptor bean
        theResult[ 0 ] = eng.createWFAttributeDesc(
            "DocName",  WFAttributeDesc.NO_LOCK );
        theResult[ 1 ] = eng.createWFAttributeDesc(
            "DocPages", WFAttributeDesc.NO_LOCK );
    } catch( Exception ex ) {
        showError( "Error creating attribute list: " +
            ex.getMessage() );
    }

    // return initialized array
    return theResult;
}
```

Error Handling

If you use Java try/catch blocks around code that invokes iIS JavaBeans methods you can catch exceptions from the iIS engine or the JavaBeans server. If your method does not catch the exceptions generated by the iIS engine, the method must declare that it throws the exceptions WFException and RemoteException.

The following code sample attempts to open a session with an engine based on the parameters given. The catch block at the end handles all normal runtime errors that iIS or Java may generate. The code uses the WFEngine bean to create user profile beans:

Code Example 5-4 Error handling (JavaBeans)

```
public static WFSession openSession( WFEngine eng, String
userProfileName, String userName, String userRole, String
userPassword, String sessionName ) {
    WFSession theResult = null;
    try {

        // Create user profile beans
        WFUserProfile user = eng.createUserProfile();

        user.setProfileName( userProfileName );
        user.setUserName( userName );
        user.addRole( userRole );
        theResult = eng.openSession( sessionName, user,
            userPassword,
            WFSession.OFFERED_ACCEPTED  +
            WFSession.RECONNECT_ALLOWED +
            WFSession.SUSPEND_ON_DISCONNECT  );
    } catch( Exception ex ) {
        showError( "Cannot open session: " +
            ex.getMessage() );
    }
    return theResult;
}
```

iIS Data Types and Java Data Types

iIS process definitions use native iPlanet UDS data types; these data types are converted to Java data types as needed. Data to be sent from a Java client to the iIS engine is also automatically converted. The following table shows the iPlanet UDS data types and the corresponding Java data types:

iIS Data Type	Java Data Type	Comments
TextData	string	Java uses UTF-8 encoding
DoubleData	double	8-byte IEEE floating point
BooleanData	boolean	
DateTimeData	java.util.Date	
IntervalData	long	
IntegerData	int	

Using The JavaBeans API

A client application uses the client application program interface (API) to communicate with the engine and execute processes registered with that engine. A client application typically uses the API to perform the following tasks:

- open a session with the engine
- display activities (work items) to the user, either as a work list (heads up) or a single work item (heads down)
- update the user's work list as necessary
- take control of an activity ("start" an activity) so the user can perform it
- set process attributes related to the work the user wants to do
- notify the engine when the work is done
- create instances of processes
- close sessions
- handle exceptions

This section explains how these tasks are accomplished using the JavaBeans client API with sample Java code included.

Opening a Session With the Engine

To perform useful work, your client must establish a session with an iIS process engine.

➤ **To open a session**

1. Designate which JavaBeans server the client application will use.
2. Determine the name of the engine to which your session will connect.
3. Connect to that engine.
4. Establish a session with that engine.

Designating the JavaBeans Server

The client communicates with the server using Java RMI. Before any interaction with the iIS engine can occur, each client must designate which JavaBeans server it wants to use.

The server is designated by its network name (as known to dns), and the RMI registry port number. These values are determined by which node the server is running on, and the port number provided to the server, if any. The default port number is 1099.

The following code sample designates the JavaBeans server at node titan, port 1147:

```
public class DirSvcHandler {
private WFDirectoryService dirSvc = new
    WFDirectoryService();

public DirSvcHandler() {
    dirSvc.setServerName( "titan" );
    dirSvc.setServerPort( 1147 );
} }
```

Determining the Engine Name

An engine name has two parts: the name of the engine, and the name of the environment to which that engine is connected. You can determine the engine name and environment in a variety of ways. The simplest approach is to hard-code the engine name. If you want your program to be more flexible, you can get this information from environment variables or configuration files, or by prompting the user for the information at startup.

The following code sample lists the iIS engines available at runtime:

Code Example 5-5 Listing available process engines (JavaBeans)

```
public class DirSvcHandler {
private WFDirectoryService dirSvc = new
    WFDirectoryService();
public void listEngines() {
    try {
        WFEngine[] engines;

        // Get an array of available iIS engines
```

Code Example 5-5 Listing available process engines (JavaBeans) *(Continued)*

```
        engines = dirSvc.getAvailableEngines();

        int nEngs = engines != null ? engines.length : 0;
        for ( int iEng = 0; iEng < nEngs; iEng++ ) {
            WFEngine eng = engines[ iEng ];
            System.out.println( eng.getName() );
        }
        if ( nEngs == 0 )
            System.out.println(
                "Sorry ... no engines available" );
        } catch( Exception ex ) {
            showError( "Error getting engine list: " +
              ex.getMessage();
        }
}
```

The previous code sample uses the WFDirectoryService bean instance named dirSvc to get a list of all available engines. This list could serve to create a pick list for the user.

Connecting to an Engine

After you establish the engine name, you create the connection to that engine by invoking the WFDirectoryService bean's wfFindEngine method, which returns a WFEngine bean instance. The following code sample gets a WFEngine bean and passes that bean to another method:

```
public class DirSvcHandler {
private WFDirectoryService dirSvc = new
    WFDirectoryService();
public WFEngine openEngine( String engName ) {
    WFEngine theResult = null;
    try {

        // Use the directory service bean to find the engine
        theResult = dirSvc.wfFindEngine( engName );
    } catch( Exception ex ) {
        showError( "Error opening " + engName + ": " +
            ex.getMessage() );
    }
    return theResult;
}
```

Establishing a Session with the Engine

By invoking wfFindEngine, you obtain a WFEngine bean instance. You use this bean instance to establish an engine session. In addition to the WFEngine bean, you need a WFUserProfileDesc bean (a user profile descriptor) that you create to describe the user who is connecting to the engine.

The following code sample creates a user profile descriptor by invoking the engine's createUserProfile method with the appropriate name. It uses the descriptor bean in the openSession invocation; if successful, openSession returns a bean referencing a WFSession bean. (In a more realistic program, the user profile data would not be hard-coded.)

Code Example 5-6 Opening a session (JavaBeans)

```
public static WFSession openSession( WFEngine eng ) {
    WFSession theResult = null;
    try {

        // Create user profile descriptor instance
        WFUserProfileDesc user = eng.createUserProfile();
        user.setProfileName( "FmsMaintProfile" );
        user.setUserName( "Berton, Pierre" );
        user.setOtherInfo( "Other user information" );
        user.addRole( "Electrical Repair" );
        user.addRole( "Hydraulic Repair" );
        user.addRole( "PLC Programming" );

        theResult = eng.openSession( "FmsTroubleTicket",
            user, "passwordShuhh!",
            WFSession.OFFERED_ACCEPTED  +
            WFSession.RECONNECT_ALLOWED +
            WFSession.SUSPEND_ON_DISCONNECT );
    } catch( Exception ex ) {
        showError( "Error opening session: " +
            ex.getMessage() );
    }
    return theResult;
}
```

Suspended and Reconnected Sessions

If the session is interrupted by a network or engine failure (because the engine has not been set up to be fully fault-tolerant) or by intervention from a system administrator, you can have the engine either suspend the session (because the user might be in the middle of some work) or terminate it (thus losing whatever work might be in progress). Allowing a session to be suspended means that any work the user was doing can be resumed at a later date. Requiring that a session be terminated when it loses contact with the engine means that the work the user was doing will be lost, any process attributes that were locked will be unlocked, and the corresponding activity will be made READY again, available for other users to perform.

When a session is put in a SUSPENDED state, the following actions occur:

- Any activities that are currently ACTIVE are processed according to each activity's defined suspend action (or the overriding value specified in the suspendAction parameter of startActivity or startQueuedActivity). ACTIVE means that the activity was started by the client application's invoking either WFActivity.startActivity or WFSession.startQueuedActivity, as described in the online iIS JavaBeans API reference.

  If the defined action in the process definition is *remove* (or the suspendAction parameter specified WFActivity.REMOVE), the activity is removed from the session's activity list by rolling it back to the READY state (rolling back includes closing the activity's attribute accessor with ROLLBACK). The default action is to abort the activity when the session is suspended.

  If the defined action is *retain* (or the parameter specified WFActivity.RETAIN), the activity is retained on the session's activity list in the ACTIVE state.

- Any activities that were on the session's activity list that were not chosen by the user are in READY state. The engine takes these activities off that session's activity list because the user can no longer choose any of them.

Disconnect Action Values

To determine whether a session is suspended or terminated if the connection to the engine is unexpectedly interrupted, you set the controls parameter of openSession with one of the Disconnect Action values. The settings are WFEngine.SUSPEND_ON_DISCONNECT or WFEngine.TERMINATE_ON_DISCONNECT.

If you do not set one of these Disconnect Action values, the default action is to terminate the session if the WFSession bean determines that the engine is disconnected and cannot be reconnected.

When you start a session, you can indicate whether you want the engine to reconnect to a previously suspended session, if there was one, or to discard any previously suspended sessions and start fresh. If you reconnect to a suspended session, the user can gain access to work that was in progress when the session was suspended.

Reconnect Action Values

To determine whether a session reconnects to a suspended session or starts up as a new session, you set the controls parameter of OpenSession with one of the Reconnect Action values. The settings are WFEngine.RECONNECT_ALLOWED or WFEngine.RECONNECT_PROHIBITED.

If you do not set one of these Reconnect Action values, the default action is to reconnect to a suspended session.

Offered Activity Notifications

Whenever an offered activity becomes READY, the engine compares its assignment rules to the user profile objects of all active sessions and adds the activity to the activity lists of the sessions that match. However, if a session is heads-down, it works only with queued activities, and allowing the engine to test the session's user profile object against the assignment rules of offered activities is a waste of system resources, because the session never gets any offered activities.

To save engine resources, an application logging on as a heads-down session should set the controls parameter of OpenSession with the Activity Offers constant WFEngine.OFFERED_IGNORED. If this constant is not set in the controls parameter, the engine by default will evaluate the session whenever an offered activity becomes READY.

Getting Work Information From the Engine

After you have established a session with the engine, you have a WFSession bean that you use to get work information from the engine. You can request two kinds of work:

- a list of offered activities (used by an application that offers a list of work items for the user to choose from, a "heads-up" application)
- a queued activity (used by an application that gives the user no choice, but starts a single task for the user, a "heads-down" application)

Requesting a Queued Activity

If the client application is a heads-down application, it gets one activity at a time from the engine and starts it for the user. This type of activity is called a *queued activity*, because the engine uses a queue to manage it for client sessions that can request one.

The startQueuedActivity method of the WFSession bean lets you request a queued activity. When you make the request, you can specify that the method either wait for an activity (specifying WFSESSION.WAIT) or return. The method can return regardless of whether the engine has an activity for the session (specifying WFSESSION.IMMEDIATE).

The following code sample returns immediately, whether or not a queued activity is available:

```
// The session sess has previously been opened
public static WFActivity getWork( WFSession sess ) {
    WFActivity theResult = null;
    try {
        theResult = sess.startQueuedActivity(
            "StampingPressTroubleTicket", // Process name
            "Diagnose",                  // Activity name
            WFSession.RETAIN,            // Suspend action
            WFSession.IMMEDIATE,         // Empty action
            null );                      // Process attribute list
    } catch( Exception ex ) {
        showError( "Error starting next work activity: " +
            ex.getMessage() );
    }
    return theResult;
}
```

As the name implies, startQueuedActivity not only gets a queued activity from the engine, but starts the activity as well. You still must complete the activity when the client is done, as described under “Completing an Activity” on page 181.

Requesting a List of Offered Activities

If your client application is a heads-up application, it requests a list of activities from the engine. It then displays the list of activities to the user as a work list, from which the user can choose any work item. You obtain the list of activities by invoking the WFSession bean’s getActivityList method, which returns an array of WFActivity beans. You then turn the list of activities into a list of work items that you can display to the user (see “Displaying a Work List” on page 173).

The following code sample uses the array theResult to store the activity list returned from the invocation to sess.getActivityList. This list could be used to create a pick list for the user:

```
// Session sess has been previously opened
public static WFActivity[] getWorkList( WFSession sess ) {
    WFActivity[] theResult = null;
    try {

        // Get list of activities offered to this session
        theResult = sess.getActivityList(
             WFSession.AL_EVENTS_ON );
    } catch( Exception ex ) {
        showError( "Error getting work list: " +
            ex.getMessage() );
    }
    return theResult;
}
```

Displaying a Work List

As described above, the client application obtains an array of WFActivity beans from the engine and turns it into a work list to display to the user. The user can select an item from the work list to start work on an activity.

The following code sample invokes the showWorkList method to display the contents of the work list:

```
// Assume the worklist has been previously filled in
public static void showWorkList( WFActivity[] workList ) {
    try {
        int nActs = workList!=null ? workList.length : 0;

        // A GUI application would use the for loop
        // to fillin a list box
        for ( int iAct = 0; iAct < nActs; iAct++ ) {
            System.out.println( String.valueOf( iAct ) +
            ": " + workList[ iAct ].getName() );
        }
    } catch( Exception ex ) {
        showError( "Error showing work list: " +
            ex.getMessage() );
    }
}
```

Maintaining the Work List

You can use one of the following two approaches for keeping a session work list display updated:

- poll the engine periodically
- respond to events posted by the engine

Polling the Engine

Using the polling approach, you periodically call WFSession.getActivityList to obtain the entire activity list for the session. You can use this list either to completely refresh the display or you can merge the changed elements of the list with the corresponding displayed elements, depending on which is faster.

Using Activity Update Caching

To use activity update caching, start a session and then turn on activity update caching in the client session. Invoke WFSession.getSessionUpdates to receive lists of activity updates as an array of WFSessionUpdate beans.

The event update feature uses the following methods and properties of WFSession JavaBean class:

- The eventControl parameter of getActivityList allows the user to turn event caching off and on. The values, WFSession.SAVE_UPDATES_ON and WFSession.SAVE_UPDATES_OFF, are used in combination with (added with) the values WFSession.AL_EVENTS_ON or WFSession.AL_EVENTS_OFF.
- setUpdateLimit allows the user to reset the default number of updates saved (the default is 20).
- getSessionUpdates returns an array of session updates (type WFSessionUpdate).
- cancelSessionUpdateWaiting cancels a getSessionUpdates invocation that is waiting for an engine response.

Update caching works best if you invoke WFSession.getSessionUpdates in a separate thread, allowing the main client thread to continue with other processing.

➤ To use activity update caching

1. Connect to an engine and start a session (see "Opening a Session With the Engine" on page 166).

2. Get the initial list of activities, turn on update caching, and display the work list. For example:

```
WFActivity[] actList;
WFSession    sess;
// ...
// Assume the session is opened
actList = sess.getActivityList(
    WFSession.AL_EVENTS_ON + WFSession.SAVE_UPDATES_ON );
```

   Calling getActivityList with these two settings turns on activity updates caching. The default number of updates that can be cached is 20. To change the default, you invoke WFSession.setUpdateLimit.

   Whenever you invoke getActivityList for activity updates caching, you must specify that you want events turned on, and that the engine is to cache them.

3. Periodically invoke WFSession.getSessionUpdates, as shown in the following code sample:

```
WFActivity[] actList;
WFSession    sess;
// ...
// Assume the session has previously been opened
WFSessionUpdate[] updateList;

// Do not wait for updates
updateList   = sess.getSessionUpdates( false );
int nUpdates = updateList != null ? updateList.length : 0;
for ( int idx = 0; idx < nUpdates; idx++ ) {
    // process the update, so that our current work list
    // is updated.
}
```

   In this sample code, the false parameter in the call to getSessionUpdates indicates that the method will not wait until the engine has activity updates. (Having the method wait requires a call in a separate thread if the main thread is to continue processing.)

You can cancel a waiting getSessionUpdates by calling WFSession.cancelGetSessionUpdates from another thread of the same session.

There is no code in the sample code's for loop. One thing you could do here is to update the work list with each activity according to the contents of each update on the list.

If you do create a separate thread for getSessionUpdates, invoke it with the wait parameter set to true. See the Java Developer's Kit documentation for details on multi-threading.

A simple example of a listener thread that is started and then cancelled is shown in the following code sample:

```
// clear existing updates
updates = sess.getSessionUpdates( false );

// Construct and start a listener thread class bean
ListenerThread  listener = new ListenerThread( sess );
new Thread( listener ).start();
// ... Do a bunch of work (or Thread.sleep( 15000 );)

// Ask session to cancel waiting for updates;
// wait for listener thread to find out about cancellation
sess.cancelSessionUpdateWaiting();
Thread.sleep( 15000 /* ms */ );
if ( listener.isCancelled )
    System.out.println( "Listener cancelled okay." );
else
    System.out.println( "Listener not cancelled!" );
```

The listener thread class for the above example is shown in the following code sample.

NOTE A more realistic listener would execute this repeatedly and update the work list rather than wait for new session updates.

Code Example 5-7 Listener thread class for this example (JavaBeans)

```
private class ListenerThread implements Runnable {

    // Constructor
    public ListenerThread( WFSession sess ) {
        this.sess = sess;
    }

    // Thread startup method
    public void run() {
        WFSessionUpdate[]   updates = null;
        try {

            // Wait for new session updates
            updates = sess.getSessionUpdates( true );
        } catch( Exception ex ) {
            showError( "Error waiting for updates: " +
                ex.getMessage() );
            return;
        }
    this.isCancelled = updates        == null ||
                       updates.length == 0;
    }

    // If some updates appeared before cancellation,
    // isCancelled will be false
    public  boolean   isCancelled = false;
    private WFSession sess        = null;
}
```

Update Limit

The session starts with *update limit*, the number of activity updates saved for this session, set to a default value of 20. You can change the update limit by invoking WFSession.setUpdateLimit.

If the total number of updates for the session exceeds this limit, the client session enters an *overrun* state. It adds an entry to the update list indicating that the limit has been exceeded (a WFSessionUpdate bean with sessionChange value of WFSession.SESSION_WORK_OVERRUN). It also removes all activity updates from the list.

Until you reset the client session's state by calling WFSession.getActivityList, the client session saves no activity updates for the session.

If your application code encounters a WFSessionUpdate bean instance with its sessionChange property set to WFSession.SESSION_WORK_OVERRUN, the session is in the overrun state. You must reset the overrun state and fully resynchronize the application's work list (invoking WFSession.getActivityList).

When you use WFSession.getSessionUpdates and specify that it will not wait (the parameter is false), getSessionUpdates returns even if there have been no updates. If there have been no updates, the returned WFSessionUpdate array is null.

If you specify that getSessionUpdates is to wait until there is something in the list (the wait parameter is true), the getSessionUpdates could get a null WFSessionUpdate array if one of the following events occurs:

- the connection from the JavaBeans server to the engine is lost
- the client application's main thread suspends or closes the session
- the client application's main thread invokes WFSession.getActivityList
- The client application's main thread invokes WFSession.cancelSessionUpdateWaiting

Closing a Session

When the user wants to log off, your client application must close the session with the engine. To close a session, invoke the WFSession.closeSession method and, depending on whether the user has completed work or not, indicate with the newState parameter that the session is to be either terminated or suspended.

Terminating the Session

If the user has completed all work and the engine has successfully closed any associated activities, you can terminate the session by invoking closeSession with the newState parameter set to WFSession.TERMINATED. Terminating the session causes the engine to remove all READY activities from the session's activity list.

Another effect of terminating the session is that the engine aborts any ACTIVE activities. If the user has any work in progress, it is represented by at least one ACTIVE activity, and all the work is lost at this point. If the user indicates that the session is to be terminated and there is an ACTIVE activity, it would be helpful to let them know the consequences, and suggest the alternatives of suspending the session, finishing all work, or rolling back the active activities.

Suspending the Session

If the user has any work in progress, you typically would take whatever steps are necessary to save the current work and then suspend the session by invoking closeSession with the newState parameter set to WFSession.SUSPENDED. Suspending the session causes the engine to remove all READY activities from the session's activity list, but leave any ACTIVE activities on the suspended session's activity list. When the user logs in at a later time, you can open the new session and tell the engine to reconnect to the suspended session by indicating WFEngine.RECONNECT_ALLOWED in the controls parameter of WFSession.openSession.

Working with Activities

As described under "Requesting a List of Offered Activities" on page 172, a heads-up client application obtains activities from the engine and then must indicate to the engine that this user will be working on one of the activities by starting that activity. (A heads-down application starts the activity at the same time it requests it with StartQueuedActivity.)

Regardless of how the activity was started, when the user has completed work on the activity, the client application must indicate when it is finished with the activity by invoking a method on the WFActivity bean.

The sections that follow describe how to do the following:

- start an offered activity
- complete work on an activity
- roll back work on an activity
- abort an activity

Starting an Offered Activity

When the user chooses an item from the work list, the client application must notify the engine that the user wants to work on the activity. It does so by calling the selected WFActivity bean's startActivity method, as shown in the following code sample. If another session has already started the activity, the engine throws an exception and control transfers to the catch block.

Code Example 5-8 Starting an activity (JavaBeans)

```
public static WFAttributeAccessor startPartAct(
    WFActivity act, WFEngine eng, String partName ) {
    WFAttributeAccessor theResult = null;
    WFAttributeDesc[]   attrList  = null;
    try {
        attrList = new WFAttributeDesc[ 1 ];
        attrList[ 0 ] = eng.createWFAttributeDesc(
            "PartType", WFAttributeDesc.WRITE );
        attrList[ 0 ].setStringValue( partName );
        theResult = act.startActivity( WFActivity.REMOVE,
            attrList ); // REMOVE is the suspended action
    } catch ( Exception ex ) {
        showError( "Error starting activity: " +
            ex.getMessage() );
    }
    return theResult;
}
```

The startActivity method returns a WFAttributeAccessor bean instance that you retain for use by the client application, allowing it to read and write process attributes in the engine.

Completing an Activity

When the user has completed work on an activity, the client application must notify the engine by invoking WFActivity.completeActivity, as shown in the following code sample:

```
public static void completePartAct( WFActivity act ) {
    try {
        act.completeActivity();
    } catch( Exception ex ) {
        showError( "Error completing activity: " +
            ex.getMessage() );
    }
}
```

Rolling Back an Activity

If the user cannot complete an activity that the client application previously started, the client application can roll back the activity to its previous (READY) state, freeing it so the engine can queue it or offer it to qualified open sessions.

To roll back an activity, you invoke WFActivity.rollbackActivity, as shown in the following code sample:

```
public static void rollbackPartAct( WFActivity act ) {
    try {
        act.rollbackActivity();
    } catch( Exception ex ) {
        showError( "Error rolling back activity: " +
            ex.getMessage() );
    }
}
```

A typical application would require logic to roll back the client application program's state as well.

Aborting an Activity

If something drastic has happened while the user was working on an activity, you might have to abort it. Since aborting an activity might also abort its entire process instance, do so only if necessary. The actual meaning of aborting an activity depends on its process definition.

To abort an activity, you invoke WFActivity.abortActivity, as shown in the following code sample:

```
public static void abortPartAct( WFActivity act ) {
    try {
        act.abortActivity();
    } catch( Exception ex ) {
        showError( "Error aborting activity: " +
            ex.getMessage() );
    }
}
```

Creating a Process Instance

If the user of your client application has the appropriate privileges, you might have to create a new instance of a process definition. To do so, call WFSession.createProcess. Typically, you also need to initialize some of the process's attributes by supplying this method with an attribute list, using an array of WFAttributeDesc bean instances.

The following code sample builds a WFAttributeDesc array:

```
public static WFAttributeDesc[] getUnitAttrList(
    WFEngine eng ) {
    WFAttributeDesc[] theResult = null;
    try {
         theResult = new WFAttributeDesc[ 3 ];
         theResult [ 0 ] = eng.createWFAttributeDesc(
            "UnitOperational", WFAttributeDesc.WRITE );
         theResult [ 0 ].setBooleanValue( false );
         theResult [ 1 ] = eng.createWFAttributeDesc(
            "UnitNumber", WFAttributeDesc.WRITE );
         theResult [ 1 ].setIntegerValue( 41 );
         theResult [ 2 ] = eng.createWFAttributeDesc(
            "TroubleArea", WFAttributeDesc.WRITE );
         theResult [ 2 ].setStringValue( "Ram 1B" );
    } catch ( Exception ex ) {
        showError( "Error creating unit attribute list: "
            + ex.getMessage() );
    }
    return theResult;
}
```

The following code sample creates a new process instance:

```
public static WFProcess createRepairProcess( WFEngine eng,
    WFSession sess ) {
    WFProcess         theResult = null;
    WFAttributeDesc[] attrList  = getUnitAttrList( eng );
    try {
        theResult = sess.createProcess(
            "HydraulicRepairTicket", attrList );
    } catch( Exception ex ) {
        showError( "Error creating repair process: " +
            ex.getMessage() );
    }
    return theResult;
}
```

The createProcess method returns a WFProcess bean instance, which can be used to access the process instance’s timers, attributes, name, ID, and state data.

Setting Process Recovery Level

By default all current process state information is recovered for a process in the event of engine or system failure. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. If appropriate, the recovery level can be altered so that the process instance is simply restarted with no recovery of process state information; another alternative you have is specifying no recovery of the process.

The default behavior of full recovery applies under the following conditions:

- no other value is specified for the process recovery level in the Process Definition property inspector
- you do not override the default in the client application code
- the engine is configured for state recovery

If logging options for the engine are turned off, the process definition default does not override the engine setting. For details on setting the recovery level in the Process Definition Workshop, see the *iIS Process Development Guide*. For details on engine configuration, see the *iIS Process System Guide*.

You can use the recoveryInfo parameter of the CreateProcess method to control whether or not the recovery level value specified in the process definition is used when creating the process. This method returns a WFProcess object for each newly created process instance. If either no value or the value WFProcess.RCVR_NORMAL is passed in for recoveryInfo, the process is created using the recovery level specified in the process definition. If another value is passed in, that value overrides the recovery level value specified in the Process Definition property inspector. The possible other values are WFProcess.RCVR_FULL, WFProcess.RCVR_NONE, or WFProcess.RCVR_PROCESS_ONLY.

Chapter 6

Building an ActiveX Client

This chapter describes how to use the iIS ActiveX process client API to write a client application for an iIS enterprise system. ActiveX, based on Microsoft's COM and DCOM standard, can be used by various languages, such as C++ and Visual Basic. Examples in this chapter are written in Visual Basic.

This chapter starts by explaining how to set up your system to use iIS ActiveX controls. After this general information, the chapter describes typical tasks your iIS client application is likely to perform with ActiveX API, such as opening a session and getting a list of available activities.

For more information, refer to Chapter 1, "Building an iIS Process Client Application" on page 25 of this manual and the introduction of the *iIS Process Development Guide*.

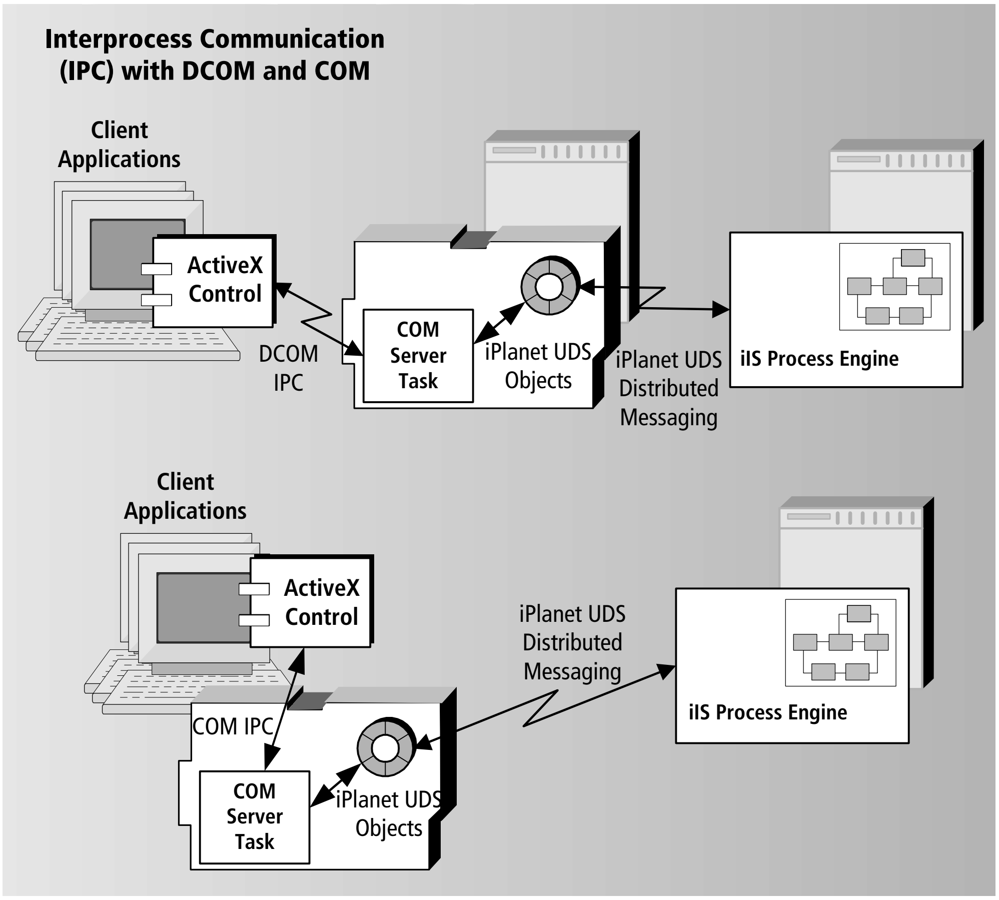
Using the ActiveX API with iIS

The iIS ActiveX process client API uses a series of invisible ActiveX controls to allow your Visual Basic client application to interact with the iIS engine. Each control supplies a variety of functions and properties that you use to send requests to and receive information from the engine. Since the controls are invisible (have no visual component that displays at runtime), you interact with them by using Visual Basic code.

Setting Up DCOM

ActiveX uses the Component Object Model (COM) to provide services within a single computer. However, you can set up ActiveX to use the Distributed Component Object Model (DCOM) to provide these services from remote servers on a network. The following figure shows the differences in the way interprocess communication (IPC) is handled by COM and DCOM.

Figure 6-1 Interprocess Communication with DCOM (top) and COM



From the client perspective, the system operation of ActiveX is the same for both COM and DCOM configurations. With the DCOM configuration you can avoid installing iIS on each client machine, which can provide considerable memory savings for large installations. However, as with any system using remote access, there can be a significant performance impact when using DCOM, depending on your network.

DCOM Installation and Configuration

When configuring ActiveX for DCOM, you need to install the ActiveX controls on the client machine and provide a means for the client application to determine which remote server to use. You can specify a server on the client by:

- configuring the client's registry
- setting an environment variable (FORTE_WF_REMOTE_COM_SERVER) for the client

Modifying the registry provides more flexibility for setting security features.

NOTE If both configuration techniques for DCOM are used on the same client machine, the server specified by FORTE_WF_REMOTE_COM_SERVER is used.

Refer to the *iIS Installation Guide* for detailed information on configuring a client to use DCOM.

Installing and Configuring the ActiveX Client API

iPlanet UDS must be installed on the same machine as your Visual Basic development environment and your runtime environment. See the *iIS Installation Guide* for more information.

After installing iIS, you can configure your Visual Basic project to use the iIS client library, which has the iIS ActiveX API controls in it.

➤ **To configure your Visual Basic project**

1. Open or create a project.
2. Select Project > References.
3. In the Available References list, select iPlanet UDS Conductor: Client ActiveX Controls Module and click OK.

➤ **If iPlanet UDS Conductor: Client ActiveX Controls Module item does not appear on the Available References list**

1. Click the Browse button.
2. Change the Files of Type field to ActiveX Controls (*.ocx).
3. Browse to your main Windows directory.
4. Open the following folders: System32 (on NT) or System (on Windows95), then Forte, then Conductr.
5. Open the file CONCLIEN.OCX so you can browse the controls in it.

Browsing the iIS Controls

As with any set of Visual Basic objects, you can inspect the iIS ActiveX controls. To do so, select View > Object Browser and change All Libraries to CONDUCTORCLIENTLib. You can then browse the list of iIS controls and constants.

NOTE For backward compatibility, Visual Basic references to iIS controls retain the "Conductor" naming convention used in a previous release.

Using Control Methods and Constants

You call iIS process engine functions as usual, by using the controls as Visual Basic objects. As mentioned earlier, you find these functions by using Object Browser. The following code sample shows a simple function call used to terminate a session:

Code Example 6-1 Using a Visual Basic control object (ActiveX)

```
 ' WFSession has been previously set up
Dim mySession As Object

Private Sub Terminate_Click()
  OnError GoTo ErrHandler

   ' Use constant value argument
  mySession.CloseSession (WFSESSION_TERMINATED)
  Exit Sub
```

Code Example 6-1 Using a Visual Basic control object (ActiveX) *(Continued)*

```
ErrHandler:
  MsgBox ("Error terminating session: " + Err.Description)
EndSub
```

The line that calls mySession.CloseSession illustrates a function call with a single argument that takes a constant value. If the object mySession is not of the correct type or is NIL, Visual Basic will generate the appropriate error exception.

The iIS process engine API requires the use of constants in various function calls. To give symbolic names to what would otherwise be simple integer values, iIS provides enumerations of constant values in the ActiveX control module. You browse constants with the object browser, as described previously under "Browsing the iIS Controls."

Using Control Properties

You use control properties to read or write data items associated with a control. The items can be of various types, depending on the particular property. Many iIS process engine API properties can only be read. Some can be both read and written to, and, in one case, a property can only be written to and not read. The following code sample reads some iIS process engine properties:

```
 ’ WFSession has been previously set up
Dim mySession As Object

Private Sub Form_Activate()

   ’ Display session properties in text boxes on the form
  If mySession.IsNil Then
    IsNil_TB = "TRUE"
  Else
    IsNil_TB = "FALSE"
  End If
  OpenStatusTB = mySession.OpenStatus
  ID_TB = mySession.ID
  SessionNameTB = mySession.SessionName
EndSub
```

Writing to a property is similar to reading one, except that the property is on the left of the assignment operator, as shown in the following code sample:

```
Dim Desc0 As Object
'... Set Desc0 to be a WFAttributeDesc. Then...
Set Desc0 =
   CreateObject("CONDUCTORCLIENT.WFAttributeDescCtrl1.1")
Call Desc0.New

 ' Write to AttributeName
Let Desc0.AttributeName = "DocName"

 ' Write to StringValue
Let Desc0.StringValue = "Winnie the Pooh"
```

The sample code uses Let statements to write to the properties AttributeName and StringValue. If these properties were read-only, as most iIS process engine properties are, Visual Basic would generate a runtime error.

To read control properties at runtime, use the Visual Basic debugger watch window.

Using Arrays of iIS Objects

The ActiveX API has arrays of various iIS object types. These arrays are typical Visual Basic variant arrays. For example:

```
Private Sub Command2_Click()
List1.Clear
Dim engines

 ' DirSvcl is a WFDirectory Service control
Let engines = DirSvc1.WFGetAvailableEngines
If IsNull(engines) or IsEmpty(engines) Then
  MsgBox ("No engines")
Else

   ' engObj is a WFEngine object
  For Each engObj in engines
    List1.AddItem (engObj.EngineName)
  Next engObj
EndIf
EndSub
```

As you can see, you iterate through variant arrays like engines conventionally, by using a For/Next loop.

Creating Arrays of iIS Objects

The iIS ActiveX client API requires that you create arrays of iIS objects for various function calls. An attribute list, an array of WFAttributeDescCtrl objects, is a typical array you create in client applications. The following code, which declares and fills in an attribute list containing two attribute descriptors, shows how to create this kind of array:

```
Dim attrList2(2) As Object

Set attrList2(0) =
  CreateObject("CONDUCTORCLIENT.WFAtrributeDescCtrl1.1")
attrList2(0).New
attrList2(0).LockType = WFATTRIBUTEACCESSOR_NO_LOCK
attrList2(0).AttributeName = "PartType"

Set attrList2(1) =
  CreateObject("CONDUCTORCLIENT.WFAtrributeDescCtrl1.1")
attrList2(1).New
attrList2(1).LockType = WFATTRIBUTEACCESSOR_NO_LOCK
attrList2(1).AttributeName = "PartRevision"
```

Referencing iIS Controls

When you need to reference an iIS control in your code, you declare an object of type object and initialize it as required by the iIS process engine API. The following code sample, which gets and handles an object that references a deadline timer control, illustrates how to set a variable to reference a control, and how to pass it to functions (and forms) at runtime:

Code Example 6-2 Referencing an iIS control (ActiveX)

```
Dim myProc As Object
'... set it to reference a WFProcess control elsewhere ...
Private Sub GetDeadlineTimerCmd_Click()
  On Error GoTo ErrHandler
  Dim timer As Object

   ' Note use of set
```

Code Example 6-2 Referencing an iIS control (ActiveX) *(Continued)*

```
  Set timer = myproc.GetDeadlineTimer(TimerName)
  Dim frm As New DeadlineTimerForm

   ’ Pass timer without parens
  frm.Initialize timer
  frm.Show
  Exit Sub

ErrHandler:
  MsgBox ("Error getting deadline timer: " + Err.Description)
End Sub
```

The code sample illustrates how a variable is set to refer to a control, and how it is passed to other functions (and forms) in the runtime environment.

Creating iIS Controls

In most cases, you do not need to create iIS controls yourself. Instead, you simply declare them as objects of type Object, and they are created for you when you call a function or access a property. There are three types of controls you must create yourself, however: WFDirectoryService, WFUserProfileDesc, and WFAttributeDesc.

For example, to create an attribute descriptor (WFAttributeDesc) control, you use the Visual Basic CreateObject function as follows:

```
  Dim Desc0 As Object
  Set Desc0 = CreateObject("CONDUCTORCLIENT.WFAttributeDescCtrl.1")
```

NOTE The iIS ActiveX client API must be properly installed on your system for the call to CreateObject to succeed.

Error Handling

Some of the code samples in this section have used the statement On Error GoTo at the beginning of subroutines and functions. This statement allows your code to catch exceptions from the iIS process engine. For example, attempting to create an instance of a process type that does not exist will cause such an error exception. If you do not catch the error (by having an On Error statement active in your subroutine, or in one of its callers) the program is terminated.

The following code sample attempts to open a session on an engine based on the contents of the subroutine's form. The error handler at the end of the subroutine catches all runtime exceptions generated by Visual Basic and the iIS process engine. The error handler uses a message box to tell the user what failed (clicking on the Open Session button), and why it failed (the error description).

Code Example 6-3 Error handling (ActiveX)

```
Private Sub OpenSession_Click()
  On Error GoTo ErrHandler
  Dim user
  Set user =
    CreateObject("CONDUCTORCLIENT.WFUserProfileDescCtrl.1")
  Call user.New

  Set user.ProfileName = ProfileNameTB
  Set user.UserName = UserNameTB
  Set user.OtherInfo = OtherInfoTB
  user.AddRole (RoleTB)

  Dim mySess As Object
  Set mySess = myEng.OpenSession(SessNameTB, user, PasswdTB, 0)
  If IsNull(mySess) Or IsEmpty(mySess) Then
    Call MsgBox("NULL or Empty result")
  Else
    Dim frm As New Form4
    Call frm.SetSession(mySess)
    Call frm.Show
  End If
  Set user = Nothing
  Exit Sub

ErrHandler:
  Call MsgBox("Cannot open session: " + Err.Description)
  Set user = Nothing
End Sub
```

iIS Data Types and Visual Basic Data Types

Type conversions are handled automatically between Visual Basic data types and the Visual Basic data types used by iIS. The following table shows the iIS data types and the corresponding Visual Basic data types:

iIS Data Type	Visual Basic Data Type	Comments
TextData	String	Windows NT uses UNICODE strings, while Windows 95 uses ANSI strings.
DoubleData	Double	8-byte floating point
BooleanData	Boolean	
DateTimeData	Date	Days are relative to midnight on 1899.12.30 00:00
IntervalData	Double	Visual Basic's interval is a floating point value based on days, while iPlanet UDS's is a value that is set with a string of the form "years:months:*days*:*hours*:*minutes*:*seconds*:*milliseconds*". The translation between the two is done automatically.
IntegerData	Long	4-byte integer

The ActiveX API for Visual Basic Programmers

A client application uses the client application program interface (API) to communicate with the engine and execute processes registered with that engine. Some of the things a client application does with the API are the following:

- open a session with the engine
- display activities (work items) to the user, either as a work list (heads up) or a single work item (heads down)
- update the work list displayed to the user as necessary
- take control of an activity ("start" an activity) so the user can perform it
- set process attributes related to the work the user wants to do
- notify the engine when the work is done
- create instances of processes

- close sessions
- handle exceptions

This section explains how these tasks are accomplished using the iIS ActiveX process client API, with sample Visual Basic code included.

Opening a Session With the Engine

To perform useful work, your client must establish a session with an iIS process engine.

➤ **To open a session**

1. Determine the name of the engine your session will connect to.
2. Connect to that engine.
3. Establish a session with that engine.

Determining the Name of the Engine

An engine name has two parts: the name of the actual engine, and the name of the environment to which that engine is connected. You can determine the engine name and environment in a variety of ways. The simplest approach is simply to hard-code the engine name. If you want your program to be more flexible, you can get this information from environment variables or configuration files or by prompting the user for the information at startup.

The following code sample lists the iIS engines available at runtime in a list box of engine names on the form:

```
Private Sub ListEngines_Click()
  List1.Clear
  Dim engines
  Let engines = DirSvc1.WFGetAvailableEngines

  If IsNull(engines) Or IsEmpty(engines) Then
    MsgBox ("Sorry. . . no engines are available")
  Else
    For Each engObj In engines
      List1.AddItem (engObj.EngineName)
      Next engObj
  End If
```

The previous code sample uses the WFDirectoryService control named DirSvc1 (an invisible control on the form) to get a list of all available engines. Alternatively, you can create the control as an object and use that object instead, as shown in the following code sample:

```
Dim DirSvc
Set DirSvc =
  CreateObject("ConductorClient.WFDirectoryServiceCtl.1")
Let engines = DirSvc.WFGetAvailableEngines
```

Connecting to an Engine

After you establish the engine name, you create the connection to that engine by using the WFDirectoryService control's WFFindEngine function, which returns an object that references a WFEngine control. The following code sample gets the object referencing the engine control and passes that Object to another form.

```
Private Sub OpenEngine_Click()
  Dim myEng As Object
  Dim nom As String
  Dim env As String

  Set myEng = DirSvc1.WFFindEngine(Text1, Text2)

  If IsNull(myEng) Or IsEmpty(myEng) Then
    MsgBox ("No such engine")
  Else

     ' Show the form for opening a session on the engine
    Dim frm As New Form2
    frm.SetEngine myEng
    frm.Show
  End If
End Sub
```

Establishing a Session with the Engine

To establish a session with the engine (the WFEngine control) that you obtained by calling WFFindEngine, you use the engine control to create a WFSession object. In addition to the WFEngine object, you need a WFUserProfileDesc object (a user profile descriptor) that you set up to describe the user who is connecting to the engine.

The code sample that follows creates a user profile descriptor by calling Visual Basic's CreateObject function with the appropriate name. It uses the descriptor object in the OpenSession call; if successful, OpenSession returns an object referencing a WFSession control.

Code Example 6-4 Opening a session (ActiveX)

```
Private Sub OpenSession_Click()
   On Error GoTo ErrHandler
   Dim user
   Set user =
    CreateObject("CONDUCTORCLIENT.WFUserProfileDescCtrl.1")
   Call user.New

   ' In a more realistic program, user profile
   ' data would come (in part) from Form fields
   Set user.ProfileName = "FmsMaintProfile"
   Set user.UserName = "Berton, Pierre"
   Set user.OtherInfo = "Other information about the user";
   user.AddRole ("Electrical Repair")
   user.AddRole ("Hydraulic Repair")
   user.AddRole ("PLC Programming")

   ' NameOfSession and Password are form fields
```

Code Example 6-4 Opening a session (ActiveX) *(Continued)*

```
   Dim mySess As Object
   Set mySess = myEng.OpenSession(NameOfSession, user, Password,
0)
   If IsNull(mySess) Or IsEmpty(mySess) Then
      Call MsgBox("Unexpected error opening session!")
   Else
      Dim frm As New Form4

     Call Form4.SetSession(mySess)
     Call Form4.Show
   End If
Exit Sub

ErrHandler:
 Call MsgBox("Cannot open session: " + Err.Description)
End Sub
```

Suspended and Reconnected Sessions

If the session is interrupted by a network or engine failure (because the engine has not been set up to be fully fault-tolerant) or by intervention from a system administrator, you can have the engine either suspend the session (because the user might be in the middle of some work) or terminate it (thus losing whatever work might be in progress). Allowing a session to be suspended means that any work the user was doing can be resumed at a later date. Requiring that a session be terminated when it loses contact with the engine means that the work the user was doing will be lost, any process attributes that were locked will be unlocked, and the corresponding activity will be made READY again, available for other users to perform.

When a session is put in a SUSPENDED state, the following actions occur:

- Any activities that are currently ACTIVE are processed according to each activity's defined suspend action (or the overriding value specified in the suspendAction argument of StartActivity or StartQueuedActivity). ACTIVE means that the activity was started by the client application's making a call to either WFActivity.StartActivity or WFSession.StartQueuedActivity. For more information, see the iIS online help.

  If the defined action in the process definition is *remove* (or the suspendAction argument specified WFActivity_REMOVE), the activity is removed from the session's activity list by rolling it back to the READY state (rolling back includes closing the activity's attribute accessor with ROLLBACK).

If the defined action is *retain* (or the argument specified WFActivity_RETAIN), the activity is retained on the session's activity list in the ACTIVE state.

- Any activities that were on the session's activity list that were not chosen by the user are in READY state. The engine takes these activities off that session's activity list because the user can no longer choose any of them.

Disconnect Action Values

To determine whether a session is suspended or terminated if the connection to the engine is unexpectedly interrupted, you set the controls argument of OpenSession with one of the Disconnect Action values. The settings are WFEngine_SUSPEND_ON_DISCONNECT or WFEngine_TERMINATE_ON_DISCONNECT.

If you do not set one of these Disconnect Action values, the default action is to terminate the session if the WFSession object determines that the engine is disconnected and cannot be reconnected.

When you start a session, you can indicate whether you want the engine to reconnect to a previously suspended session, if there was one, or to discard any previously suspended sessions and start fresh. If you reconnect to a suspended session, the user can gain access to work that was in progress when the session was suspended.

Reconnect Action Values

To determine whether a session reconnects to a suspended session or starts up as a new session, you set the controls argument of OpenSession with one of the Reconnect Action values. The settings are WFEngine_RECONNECT_ALLOWED or WFEngine_RECONNECT_PROHIBITED.

If you do not set one of these Reconnect Action values, the default action is to reconnect to a suspended session.

Offered Activity Notifications

Whenever an offered activity becomes READY, the engine compares its assignment rules to the user profile objects of all active sessions and adds the activity to the activity lists of the sessions that match. However, if a session is heads-down, it works only with queued activities, and allowing the engine to test the session's user profile object against the assignment rules of offered activities is a waste of system resources, because the session never gets any offered activities.

To save engine resources, an application logging on as a heads-down session should set the controls argument of OpenSession with the Activity Offers flag WFEngine_OFFERED_IGNORED. If this flag is not set in the controls argument, the engine by default will evaluate the session whenever an offered activity becomes READY.

Getting Work Information From the Engine

After you have established a session with the engine, you have a WFSession control that you use to get work information from the engine. You can request two kinds of work:

- a list of offered activities (used by an application that offers a list of work items for the user to choose from, a "heads-up" application)
- a queued activity (used by an application that gives the user no choice, but starts a single task for the user, a "heads-down" application)

Requesting a Queued Activity

If the client application is a heads-down application, it gets one activity at a time from the engine and starts it for the user. This type of activity is called a *queued activity*, because the engine uses a queue to manage it for client sessions that can request one.

You use the WFSession object's StartQueuedActivity function to request a queued activity. When you make the request, you can have the function either wait for an activity (specifying WFSESSION_WAIT) or return whether or not the engine has an activity for the session (specifying WFSESSION_IMMEDIATE).

The following code sample returns immediately, whether or not an a queued activity is available. It tests for a NIL activity by calling the WFActivity.IsNil function:

Code Example 6-5 Requesting a queued activity (ActiveX)

```
Private Sub GetQueuedAct_Click()
  On Error GoTo ErrHandler
  Dim Act As Object

   ' Leave as NIL list
  Dim AttrList As Variant
  Set Act =
    MySession.StartQueuedActivity(
      "StampingPressTroubleTicket", "Diagnose",
      WFSESSION_RETAIN, WFSESSION_IMMEDIATE, AttrList)
  If Not IsNull(Act) then
    Dim DiagForm As New DiagStampingPressForm
    DiagForm.Initialize Act
    DiagForm.Show
  Else
    MsgBox( "No diagnostic tasks available." );
  End If
  Exit Sub
  ErrHandler:
    MsgBox ("Error getting diagnostic task: " + Err.Description)
End Sub
```

As the name implies, StartQueuedActivity not only gets a queued activity from the engine, but starts it as well. You still must complete the activity when the client is done, as described under "Completing an Activity" on page 209.

Requesting a List of Offered Activities

If your client application is a heads-up application, it requests a list of activities from the engine. It then displays the list of activities to the user as a work list, from which the user can choose any work item. You obtain the list of activities by calling the WFSession object's ActivityList function, which returns an array of WFActivity objects. You then turn the list of activities into a list of work items that you can display to the user (see "Displaying a Work List").

The following code sample uses the array actlist to store the activity list returned from the call to MySession.ActivityList, and then calls a function of the form actForm to turn the activity list into a work list that it can display:

Code Example 6-6 Requesting an offered activity (ActiveX)

```
Private Sub GetWorkList_Click()
  On Error GoTo ErrHandler
  Dim actlist
  Let actlist = MySession.ActivityList(WF_AL_EVENTS_OFF)

  If IsNull(actlist) Or IsEmpty(actlist) Or UBound(actlist) < 0 Then
    MsgBox ("No activities")
  Else
    Dim actForm As New OfferedActsForm
    actForm.Initialize actlist
    actForm.Show
  End If
  Exit Sub
  ErrHandler:
    MsgBox ("Err getting work list: " + Err.Description)
End Sub
```

Displaying a Work List

As described under "Requesting a List of Offered Activities," the client application obtains an array of WFActivity objects from the engine and turns it into a work list that it displays to the user. The user can select an item from the work list to start work on.

In the following code sample, the Initialize function, called from the code sample in the previous section, creates a form that displays the work list:

```
 ' ----------------
 ' OfferedActsForm:
 ' ----------------
Dim MyActList As Variant

  Public Sub Initialize(actList As Variant)

   ' Clear ListBox control
  List1.Clear

   ' Remember list for whole form
  MyActList = actList
  For Each Activity In MyActList
```

```
    ' Iterate through the whole worklist
    ' populating the list box as you go
    Dim item As String
    item = Activity.ActivityName
    List1.AddItem item
   Next Activity

   End Sub
```

Maintaining the Work List

You can use one of the following two approaches for keeping a session work list display updated:

- poll the engine periodically
- respond to events posted by the engine

Polling the Engine

Using the polling approach, you periodically call WFSession.GetActivityList to obtain the entire activity list for the session. You can use this list either to completely refresh the display or you can merge the changed elements of the list with the corresponding displayed elements, depending on which is faster.

Using Activity Update Caching

To use activity update caching you start a session as you normally do, and then turn on activity update caching in the client session. You then call WFSession.GetSessionUpdates to receive lists of activity updates as a variant array of WFSessionUpdate objects.

The event update feature uses the following methods and properties of WFSession:

- The eventControl argument of ActivityList allows the user to turn event caching off and on. The values, WF_SAVE_UPDATES_ON and WF_SAVE_UPDATES_OFF, are used in combination with (added with) the values WF_AL_EVENTS_ON and WF_AL_EVENTS_OFF.
- UpdateLimit allows the user to reset the default number of updates saved (currently 20).
- GetSessionUpdates returns a variant array of session updates (type WFSessionUpdate).
- CancelSessionUpdateWaiting cancels a GetSessionUpdates call that is waiting for an engine response.

In addition, there is a control, WFSessionUpdate, which is used by the client application to request activity list updates and save any updates that come back from the engine.

Update caching works best if WFSession.GetSessionUpdates is called in a thread, allowing the main client thread to continue with other processing. However, on systems that do not have threading, it is possible to use this member function, but call it without waiting for a return value so the member function never gets blocked.

➤ To use activity update caching

1. Connect to an engine and start a session (see "Opening a Session With the Engine" on page 195).

2. Get the initial list of activities, turn on update caching, and display the work list. For example:

```
Dim actList
  Let actList =
   MySession.ActivityList(WF_AL_EVENTS_ON+WF_SAVEUPDATES_ON)
```

   Calling ActivityList with these two settings turns on activity update caching. The default number of updates that can be cached is 20. If you want to change the default number, you can call WFSession.UpdateLimit. For more details, see the iIS online help.

   When you are doing activity update caching, whenever you call GetActivityList, you must specify both that you want events turned on and that the engine is to cache them.

3. Periodically call WFSession.GetSessionUpdates, as shown in the following sample code:

```
Dim updateList
  Let updateList = mySession.GetSessionUpdates(FALSE)
  if not IsNull(updateList) then
    for each Update in updateList
         'Do something with each change
         'Could update worklist here
            Next Update
End If
```

In this sample code, GetSessionUpdates indicates that it will not wait until the engine has activity updates. (Having the function wait requires that the call be in a separate thread if the main thread is to continue processing.)

You can cancel a waiting GetSessionUpdates by calling WFSession.CancelGetSessionUpdates from another thread of the same session.

There is no code in the sample code's for loop. One thing you could do here is to update the work list with each activity encountered.

If you do create a separate thread for the GetSessionUpdates, call it with wait set to TRUE. See the Visual Basic documentation for details on multi-threading.

Update Limit

The session starts with *update limit*, the number of activity updates saved for this session, set to a default value of twenty. You can set the update limit to a different value by calling WFSession.SetUpdateLimit.

If the total number of updates for this session exceeds this limit, the client session enters an *overrun* state, adds an entry to the update list indicating that the limit has been exceeded (a WFSessionUpdate object with SessionChange value of WFSession_SESSION_WORK_OVERRUN), and removes all activity updates from the list. Until you reset the client session's state with a call to WFSession.GetSessionUpdates, the client session saves no activity updates for the session.

If your code detects that the update limit has been exceeded (by encountering a WFSessionUpdate object in the WFSessionUpdate array with a SessionChange value of WF_SESSION_WORK_OVERRUN), your client application must use WFSession.ActivityList to get all activities offered, then clear the work list and redisplay it.

When you use WFSession.SessionUpdates and specify that it will not wait (the argument is false), SessionUpdates returns even if there have been no updates. If there have been no updates, the returned WFSessionUpdate array is null.

If you specify that SessionUpdates will wait until there is something in the list (wait is true), the SessionUpdates could get a null WFSessionUpdate array if one of the following events occurs:

- the connection to the host is lost
- the client application's main thread suspends or closes the session
- the client application's main thread calls WFSession.ActivityList
- The client application's main thread calls WFSession.CancelSessionUpdateWaiting

Closing a Session

When the user wants to log off, your client application must close the session with the engine. You call the WFSession.CloseSession function and, depending on whether the user has completed work or not, indicate with the newState argument that the session is to be either terminated or suspended.

Terminating the Session

If the user has completed all work and the engine has successfully closed any associated activities, you can terminate the session by calling CloseSession with the newState argument set to WFSession_TERMINATED. Terminating the session causes the engine to remove all READY activities from the session's activity list. For a description of CloseSession, see the iIS online help.

Another effect of terminating the session is that the engine aborts any ACTIVE activities. If the user has any work in progress, it is represented by at least one ACTIVE activity, and all the work is lost at this point. If the user indicates that the session is to be terminated and there is an ACTIVE activity, it would be good to let them know the consequences and suggest either suspending the session or finishing all work as alternatives.

Suspending the Session

If the user has any work in progress, you typically would take whatever steps are necessary to save the current work and then suspend the session by calling CloseSession with the newState argument set to WFSession_SUSPENDED. Suspending the session causes the engine to remove all READY activities from the session's activity list, but leave any ACTIVE activities on the suspended session's activity list. When the user logs in at a later time, you can open the new session and tell the engine to reconnect to the suspended session by indicating WFEngine_RECONNECT_ALLOWED in the controls argument of WFSession.OpenSession.

Working with Activities

As described under "Requesting a List of Offered Activities" on page 201, a heads-up client application obtains activities from the engine and then must indicate to the engine that this user will be working on one of the activities by starting that activity. (A heads-down application starts the activity at the same it requests it with StartQueuedActivity.)

Regardless of how the activity was started, when the user has completed work on the activity, the client application must indicate when it is finished with the activity by calling a function of the WFActivity object itself.

The sections that follow describe how to do the following:

- start an offered activity
- complete work on an activity
- roll back work on an activity
- abort an activity

Starting an Offered Activity

When the user chooses an item from the work list, the client application must notify the engine that the user wants to work on the activity. It does so by calling the selected WFActivity object's StartActivity function, as shown in the code sample that follows. If another session has already started the activity, the engine throws an exception and control transfers to the function's ErrHandler. For a complete description of StartActivity, see the iIS online help.

```
Private Sub StartCmd_Click()
  On Error GoTo ErrHandler
  Dim Act As Object
  Dim acc As Object
  Dim attrList(1) As Object

  Set attrList(0) =
      CreateObject("CONDUCTORCLIENT.WFAttributeDescCtrl.1")
  attrList(0).New
  attrList(0).LockType = WFATTRIBUTEACCESSOR_NO_LOCK
  attrList(0).AttributeName = "PartType"

' Use ListBox index as an array index to get the selected
activity:
  Set Act = MyActList(List1.ListIndex)
' Attempt to start the activity (could cause an
' error if someone else has already started it.)
  Set acc = Act.StartActivity(WFACTIVITY_REMOVE, attrList)
  Dim actActForm As New ActiveActivityForm
    actActForm.Initialize acc, act
    actActForm.Show
  Exit Sub
  ErrHandler:
    MsgBox ("Error starting activity: " + Err.Description)
End Sub
```

The StartActivity function returns a WFAttributeAccessor object that you retain for use by the client application, allowing it to read and write process attributes in the engine.

Completing an Activity

When the user has completed work on an activity, the client application must notify the engine by calling WFActivity.CompleteActivity, as shown in the following code sample (for a complete description, see the iIS online help):

```
Private Sub CompleteCmd_Click()
  On Error GoTo ErrHandler
' MyAct is a form variable for the active activity.
  Set acc = MyAct.CompleteActivity
  Exit Sub

  ErrHandler:
    MsgBox ("Error completing activity: " + Err.Description)
End Sub
```

Rolling Back an Activity

If the user cannot complete an activity that the client application previously started, the client application can roll back the activity to its previous (READY) state, freeing it so the engine can queue it or offer it to qualified open sessions.

To roll back an activity, you call WFActivity.RollbackActivity, as shown in the following code sample (for a complete description, see the iIS online help):

```
Private Sub RollbackCmd_Click()
  On Error GoTo ErrHandler
' MyAct is a form variable for the active activity.
  Set acc = MyAct.RollbackActivity
  Exit Sub

  ErrHandler:
    MsgBox ("Error rolling back activity: " + Err.Description)
End Sub
```

NOTE Your client application must handle undoing changes it has made to site files or rolling back transactions it has started with site databases.

Aborting an Activity

If something drastic has happened while the user was working on an activity, you might have to abort it. Since aborting an activity might also abort its entire process instance, do so only if necessary. The actual meaning of aborting an activity depends on its process definition.

To abort an activity, you call WFActivity.AbortActivity, as shown in the following code sample (for a complete description, see the iIS online help):

```
Private Sub AbortCmd_Click()
  On Error GoTo ErrHandler
' MyAct is a form variable for the active activity.
  Set acc = MyAct.AbortActivity
  Exit Sub

  ErrHandler:
    MsgBox ("Error aborting activity: " + Err.Description)
End Sub
```

Creating a Process Instance

If the user of your client application has the appropriate privileges, you might have to create a new instance of a process definition. To do so, you call WFSession.CreateProcess (for a complete description, see the iIS online help). Typically, you also need to initialize some of the process's attributes by supplying this function with an attribute list, an array of WFAttributeDesc objects.

The following code builds a WFAttributeDesc array:

Code Example 6-7 Initializing process attributes (ActiveX)

```
Public Function SetdefaultAttributes() As Variant
  On Error GoTo SetDefaultAttributesErr

  Dim i As Integer
  Dim defAttrs(2) As Object

  For i = 0 To 2
    Set defAttrs(i) = CreateObject(
        "CONDUCTORCLIENT.WFAttributeDescCtrl.1")
    defAttrs(i).New
    defAttrs(i).LockType = WFATTRIBUTEACCESSOR_WRITE
  Next
```

Code Example 6-7 Initializing process attributes (ActiveX) *(Continued)*

```
  defAttrs(0).AttributeName = "UnitOperational"
  defAttrs(0).BooleanValue = False

  defAttrs(1).AttributeName = "UnitNumber"
  defAttrs(1).IntegerValue = 41

  defAttrs(2).AttributeName = "TroubleArea"
  defAttrs(2).StringValue = "Ram 1B"

  Let SetdefaultAttributes = defAttrs

  Exit Function

  SetDefaultAttributesErr:
    MsgBox ("Error during SetDefaultAttributes" +
Err.Description)
End Function
```

The following code creates a process instance:

```
Private Sub CreateProcess1_Click()
  On Error GoTo ErrHandler
  Dim proc As Object

' Empty(NIL) array
  Dim AttribValues As Variant
  AttribVals = SetdefaultAttributes()
  Set proc = MySession.CreateProcess(
      "HydraulicRepairTicket", AttribVals)

  Dim frm As New ProcessForm
  frm.Initialize proc
  frm.Show
  Exit Sub
  ErrHandler:
    MsgBox ("Error creating process: " + Err.Description)
End Sub
```

The CreateProcess function returns a WFProcess object, which can be used to access the process instance’s timers, attributes, name, ID, and state data.

Setting Process Recovery Level

By default all current process state information is recovered for a process in the event of engine or system failure. This includes the state of each process, activity, timer, and process attribute lock that is created in the course of process execution. If appropriate, the recovery level can be altered so that the process instance is simply restarted with no recovery of process state information; another alternative you have is specifying no recovery of the process.

The default behavior of full recovery applies under the following conditions:

- no other value is specified for the process recovery level in the Process Definition property inspector
- you do not override the default in the client application code
- the engine is configured for state recovery

If logging options for the engine are turned off, the process definition default does not override the engine setting. For details on setting the recovery level in the Process Definition Workshop, see the *iIS Process Development Guide*. For details on engine configuration, see the *iIS Process System Guide*.

You can use the recoveryInfo argument of the CreateProcess2 (not the CreateProcess) method to control whether or not the recovery level value specified in the process definition is used when creating the process. This method returns a WFProcess object for each newly created process instance. If either no value or the value WFProcess_RCVR_NORMAL is passed in for recoveryInfo, the process is created using the recovery level specified in the process definition. If another value is passed in, that value overrides the recovery level value specified in the Process Definition property inspector. The possible other values are WFProcess_RCVR_FULL, WFProcess_RCVR_NONE, or WFProcess_RCVR_PROCESS_ONLY.

Index