How To...

Implementing to the new

Management Central Java Framework

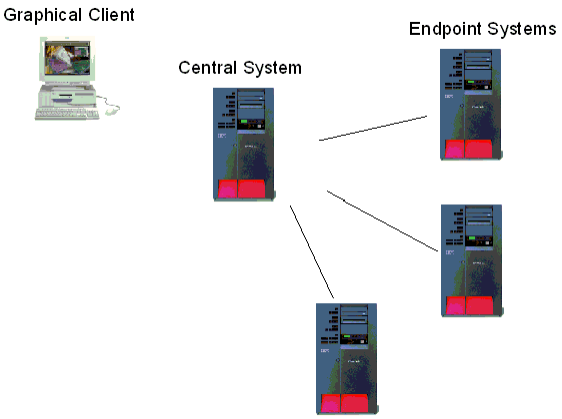
VeeFiveAreOneEmZero

Final Version

Document last changed: June 27, 2001 11:19am

Preface

"Managing multiple systems as easy as managing a single system."



Management Central Documentation

This *Management Central How To...* document is a bridge between the code you need to write for your application and the JavaDoc provided by the Management Central Java Framework. To assist in your application development, you should first find the documentation listed below and take a quick look at it. Next, read this document to get a high level understanding of what is available for your application development with tips on design and implementation. When you are finished reading this book, refer to the JavaDoc for the details about the classes and interfaces discussed in this book that will apply to your application.

Management Central Web Site:

http://www.as400.ibm.com/sftsol/MgmtCentral.htm

Java Class Documentation (JavaDoc):

AS/400 Toolbox for Java:

http://www-1.ibm.com/servers/eserver/iseries/toolbox/

What This How To Book Contains

This book is organized with three different intended readers in mind with different points of view:

- Application Designer
- GUI Developer
- Application Developer

The Application Designer is the person who understands the "big picture" of the project, the customer value, and the functions needed to be performed. The application designer influences the user interface design, performs object oriented analysis and design, and determines the structure and flow of the application.

The Application Developer uses the design from the application designer and implements the design. Some design decisions will need to be made during this process, but the application designer should be made aware of any significant changes. The application developer primarily develops code that runs on the AS/400 system, but also needs to provide some interfaces and functions for the GUI Developer.

The GUI Developer also uses the design from the application designer and implements the design. Some design decisions will need to be made during this process, but usually the User Interface Designer will influence the overall design to provide the best user experience to the customer. Any significant changes should be reviewed by the application designer. The GUI developer primarily develops code that runs on the client, but also needs to provide good requirements and feedback to the application developer.

Each major section or chapter contains the follow information:

- Overview - a brief description of the topic being discussed
- Interfaces and Flows - Application Designer, Application Developer and GUI Developer points of view
- Terms, Classes, and Interfaces - a table of Java terms, classes, and interfaces that are used
- Scenarios - commonly used development scenarios with code snippets
- Hints and Tips - technique hints and tips when designing and developing your application
- Program Examples - full source code examples with comments

Conventions Used in this Cookbook

This book uses the following typographical conventions:

This style....	Is used for...
`Fixed width font`	Code elements such as classes and methods.
`Fixed width font underline`	Emphasis for code elements.
Bold	Management Central Classes and Interfaces.
Underlined	Management Central Methods.

Table of Contents

Introduction

If you are new to Management Central, here are a some basic principles, behaviors, and a brief history of Management Central.

Principles of Management Central:

- Make the management of multiple systems as easy as managing a single system
- Provide this management capability in the base AS/400 operating system
- Provide an easy-to-use graphical user interface to management functions

Management Central Application behaviors:

- Asynchronous
- Unattended
- Scheduled
- Multiple System
- Short or Long running

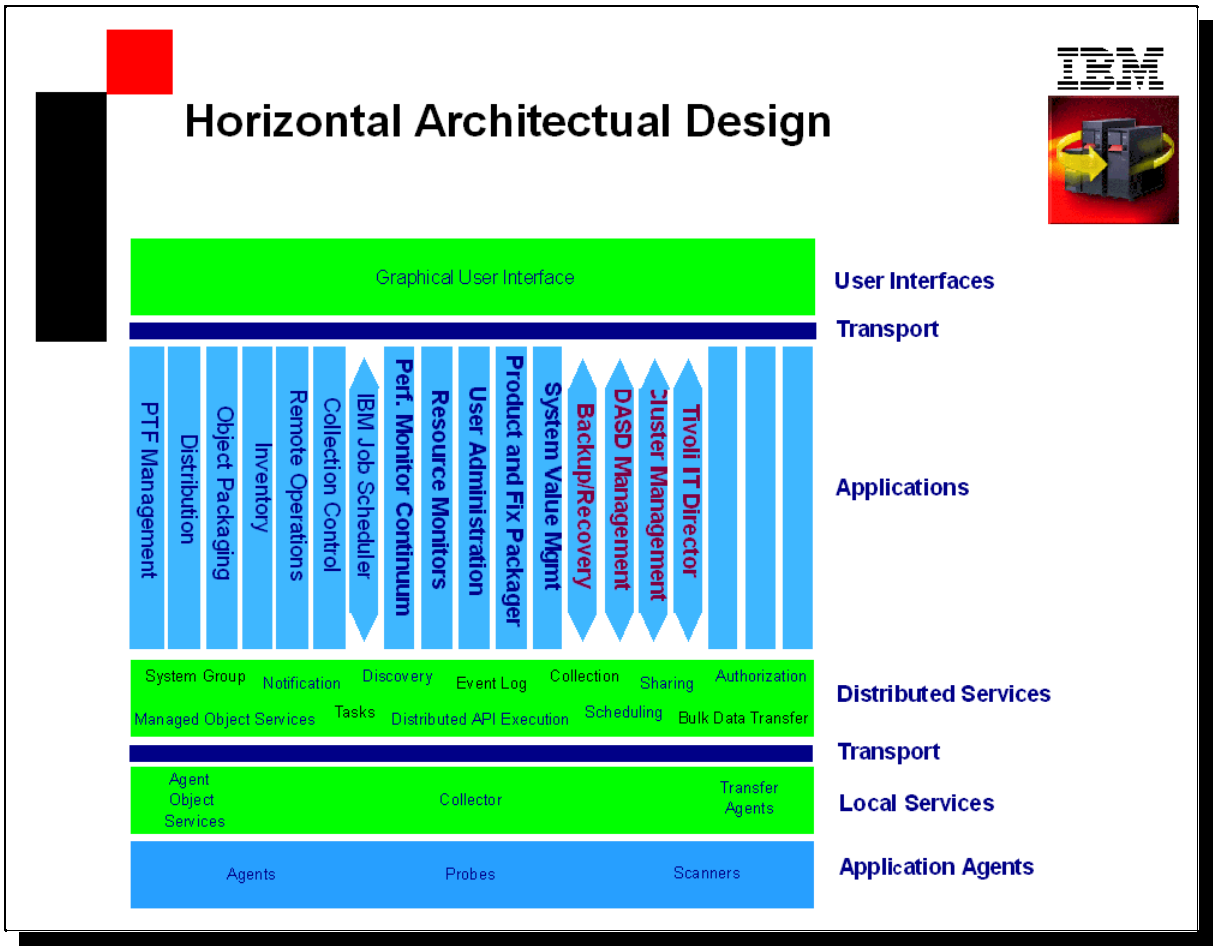
A little bit of history...
Management Central is a suite of integrated systems management applications that began to appear in V4R3 with Client Access for Windows (5763-XD1) V3R2. When installing Client Access for Windows, you can select to perform a Custom installation and optionally choose to install Management Central along with Operations Navigator.

With V4R3, Management Central provided the base for multiple system management with the introduction of the Management Central C++ infrastructure. This infrastructure provided a horizontal architectural approach to software development when developing AS/400 system management solutions. This horizontal approach separates the user interface from the transport mechanism, the application from common service components, etc. AS/400 Endpoint Systems, AS/400 System Groups, Event Log, and a Monitor application provided an intuitive graphical interface to real-time performance information with simple automation and notification for management of multiple AS/400 systems.

In V4R4, Management Central added a number of new integrated graphical applications to help manage AS/400 systems: Inventory Collection, Software Fix(PTF) Management, Remote Operations, Package and Object Distribution, and Performance Collection Services. This extended the Management Central C++ horizontal infrastructure with additional common services like Bulk Data Transfer, Discovery, and Collection Services. Management Central is now an integrated part of AS/400 Operations Navigator in V4R4. The Operations Navigator tree hierarchy has been enhanced to include Task Activity, Scheduled Tasks, Definitions, Monitors, AS/400 Endpoint Systems, and AS/400 System Groups.

In V5R1, Management Central continues to extend its AS/400 management control with additional applications like enhancing historical Monitor capabilities, Product and Fix Packager, Job Resource Monitors, Message Monitors, User Profile Management, and System Value Management. With all the interest in Management Central, areas within IBM and external to IBM are looking at using Management Central to implement their management applications and solutions. For this reason, we have also developed the Management Central Java Framework(jMC).

The Management Central Java Framework is an extendible and pluggable infrastructure for areas within IBM and external system management solution partners to use to their advantage when developing their suite of applications. Areas within IBM such as DASD Management, Backup Recovery and Media



Services (BRMS), and Logical Partitioning (LPAR) are using the jMC to build functions that balance disk drives, backup and restore data, and build system groups based on hardware configurations.

This document helps in the development of new such applications that want or need to have the behaviors and capabilities that the jMC offers.

Designing a Distributed Application

Overview

This chapter gives you an overview and a little insight on what is available when designing your application. The Management Central Java Framework provides the building blocks and tools to assist you in the construction of your management applications. There are a number of different concepts that you will want to take a look at to see if your application can benefit from them. Some management application will use all these concepts while others will only use a couple. You should take a look at all of them to see what best fits your needs. At a high level these concepts include:

- Definitions
- Activities
  - Tasks
  - Services
- Descriptors
  - Definition, Task, and Service
  - Public and Private
- Views and List Views

Definitions allow you to store information persistently on the Central System to be used or reused at a later time. Once stored on the Central System, it can be optionally viewed, changed, removed, and shared by all the operators and administrator connecting to that Central System. The Definition will remain on the Central System, even across system IPL's, until it is explicitly removed. A Package Definition is one example of a Definition. The Package Definition is used in an existing MC application that distributes packages of files or AS/400 objects to multiple systems. The Package Definition contains a list of AS/400 objects, files, folders or libraries which are grouped together for distribution. The operator can add or remove files to the package definition and store it persistently on the central system. Once the operator thinks the package contains the right files, they can then make a request to distribute the package to one or more System Groups or Endpoint Systems. This distribution request would then take the information from the package definition and create a new activity called a Task.

Activities are operations that have one or more of the following characteristics: can be short or long running, asynchronous, unattended, scheduled, and can run on multiple systems. There are two types of activities: Tasks and Services. Parameters and properties about how a task or service should run may come from a definition, or can be retrieved from an intuitive user interface (e.g. property pages, wizard, or dialogs). The definition may be a placeholder of information before the task or service needs to be created or started.

Tasks have all the characteristics of an activity but run only a single time to completion and finish with some form of status(i.e. Completed Successfully or Failed). When a task is initiated, the client will pass the request for the operation to occur to the Central System. The Central System will then take the request and fan it out, or broadcast it, to all the Endpoint Systems that are supposed to perform the

activity. After the request completes on the Endpoint System, status and results flow back to the Central System where it is stored. Once the task request has been delivered to the Central System, the client can disconnect from the Central System. At a later date or time, the client can reconnect to the Central System and view the status and results for the task. The System Values Compare-and-Update function is an example of a task. With this task, a system administrator is able to update any number of AS/400 System Values, on any number of endpoint systems based on a single model system. The System Values to be changed and the endpoint systems on which the compare and update should occur is stored as a definition on the central system. Whenever the administrator wishes to execute the compare and update, a task can be created from the stored definition, and executed on the endpoints.

Services, like tasks, have all the characteristics of an activity, but unlike tasks, are turned on and run indefinitely until turned off. When you use Services, you start them and stop them and something meaningful happens in between. Unlike tasks, a service can be run multiple times, where each time the service is turned on and turned off. A performance monitor is an example of a Service. If you write a Service to monitor CPU percentage, you would start the monitor, graph the data being monitored, and then stop the monitor when you no longer want to graph the data. A trace or collection facility would be another example where you intend to turn the function on, have something meaningful happen, and then turn it off with the ability to turn it on and off whenever necessary.

Know Your Management Central Objects

Within the Management Central Java Framework there are several categories of objects you'll need to become familiar with in order to implement your own definition or activity. Some classes from each category were brushed upon in the preceding section; they and others will be classified and described here.

There are three main categories to be concerned with when dealing with the jMC. The first, classified as the **Action**, is the real meat of the activity. It's the server side code that provides the behavior and actions that define the application. It resides on the endpoint system, and is the workhorse that performs the application's goals on each individual system. For instance, in the Distributed Command Call application supplied by the jMC, **McEndpCommandAction** is the class that actually provides the mechanism to execute or cancel the command on each endpoint system. When implementing your own activity, the Application Developer implements the **McActionIfc** and either the **McSwitchable** or **McExecutable** interfaces. These interfaces will be explained in detail in their respective sections, but in general, these classes provide the interface to start and stop the activities. The McSwitchable interface defines methods on and off that allow the user to start a service, allow it to run as long as they wish, and then end it when they are ready. The McExecutable interface defines methods execute and cancel, which allow a user to start a task and let it run to completion, or cancel it in mid-execution, if they choose.

The second category of Objects in the jMC, and the most important for the GUI programmer, is the **View**. Views provide a bridge between the graphical client and the server-side application. By creating and maintaining a reference to a View, many of the complexities involved with maintaining a remote reference to the server are abstracted from the GUI programmer. For instance, the View handles all connection details with the Central System. The AS/400 that the user has specified as the central system is stored within Operations Navigator, and upon construction of any View object, this data is retrieved, and used to connect to the system. The interfaces that exist on the View objects are there to propagate data from the client to the server, and to maintain the integrity of that data; that is, when data is changed on the client, that change must be sent to the server to ensure the endpoint action is executed correctly. They, when paired with a descriptor, give you access to create, change, and delete tasks and services, and provide listener actions to attach, detach, and be notified when elements change. The list view bridges all the interfaces and methods that the GUI developer needs to manipulate lists of elements. It gives you access to get lists of definition, task, and service views, and also listener actions to attach, detach, and be notified when elements are added, changed, or removed from the list.

Views come in a multitude of flavors, but at the topmost level resides the interface all Views must implement, **McManageableViewIfc**. Key methods from this interface are:

addManageable()	Allows the jMC to manage the data associated with this View on the Central Site. "Management" consists of (among other things) caching the data into memory, persistently storing it in a database, updating created and changed dates, and distributing the data to the endpoint systems when instructed to do so.
changeManageable()	Updates the state of a presently managed object. If, after calling addManageable(), the data stored within the View is changed, the jMC on the Central Site must be updated with the changed data. This method provides an interface to notify the jMC of the change.
getManageable()	Allows for retrieval of managed objects from the Central Site into the View object. Techniques for selecting which managed objects should be returned will be discussed in depth in a later section.
removeManageable()	Once a managed object is no longer needed (because the associated action has completed, for instance), this method must be called on the object to allow the jMc to clean up any data associated with the managed object.

In addition, class **McManageableListView** exists to manage a list of Views. It allows the user to retrieve an entire list of qualifying View objects with a single call to the Server. Each View in the list can then be managed independently or as part of the ListView. Key methods from McManageableListView are:

getManageableViews()	Returns a list of qualifying View objects from the server. The list of Views is based on criteria you specify.

removeManageableList()	Removes manageability of each View that's part of this list from the server. The jMC will no longer manage any of the elements.

It's important to note that none of the specific details behind View objects need be known by the GUI developer. That is, you don't need to know how to manage objects within the Management Central Java Framework; rather, you simply need to tell the Framework which objects to manage.

The third crucial category of objects in the jMC is the actual managed object. A managed object is technically any object implementing the McManageable interface, but more specifically, a category of objects called **Descriptors**, provided by the jMC, has already extended the McManageable interface. The Descriptor is basically the definition part of the activity. It contains the system group on which to distribute the activity, as well as the information about how and where to perform the activity. This descriptor must be created on the client, usually by gathering information from the user, and then placed within a View object. When addManageable is called on the View, the Descriptor is passed along to the Central System, and, based on data within the Descriptor, the Action can then be performed.

Distributed descriptors, which are created on the client, are used to store data about how the activity should behave. They supply system groups, and interfaces to create the specific endpoint data. Endpoint descriptors, which are created on each endpoint system, are used to direct the specific activity that should be performed. These descriptors store the class name of the action class (the class implementing execute and cancel for Tasks, or on and off for Services), and the application attributes, or data, that the action will need.

An important detail that is abstracted from the application developer is the distributed nature of activities within the jMC. When the descriptor is created on the client, a system group is defined within it. Then, when the View object is created, it sends the descriptor to the central system (the central system is retrieved from Operations Navigator - a detail that will be explained in more depth later) where information about the activity can be managed. When you tell your task to exeute, or turn your service on, the jMC handles all communication details to distribute and start the activity on the endpoint systems. Similarly, the endpoints maintain a reference to central system, and the central system maintains a reference to the client, and therefore can propagate status and results about the endpoint activity back to the client. (See the figure on the following page for details).

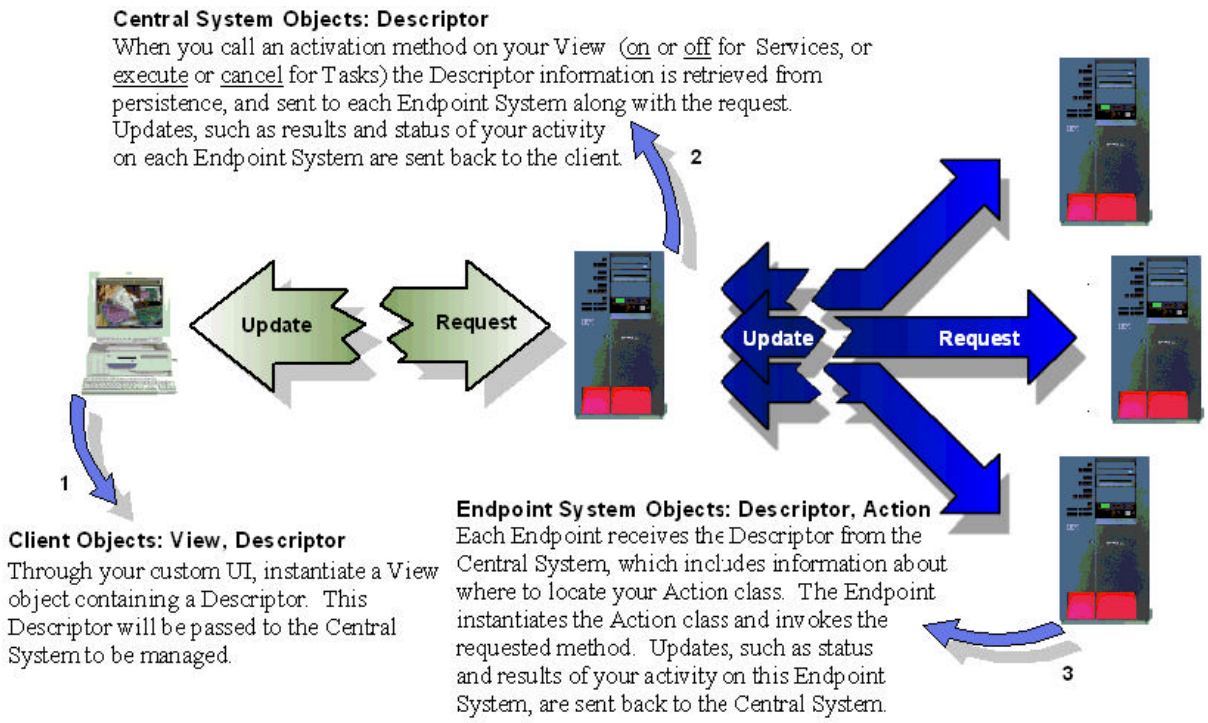


Figure: Objects and actions within the Management Central Java Framework.
When you call methods on your View object, the jMC interacts with the Central System to fulfill the request. If the request applies to the Endpoint Systems, as it would if, for instance, you asked to start a service, the Central System sends requests to each Endpoint System on your behalf. Updates from each request flow back to the client. While the Descriptor flows from client to Central System to Endpoint Systems, other Objects have their appointed place.

In designing your application, you'll most likely want to extend one of two distributed descriptor implementations, **McDistributedTaskDescriptor** or **McDistributedServiceDescriptor**, and one of two endpoint descriptor implementations, **McEndpointTaskDescriptor** or **McEndpointServiceDescriptor**.

Terms, Classes, and Interfaces

Common Management Central Java classes and interfaces that you need to be familiar with when working with Definitions, Tasks, and Services.

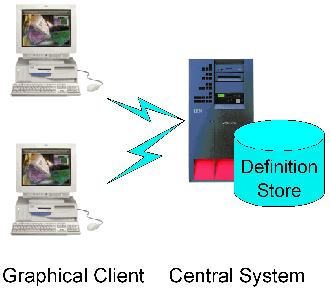
Thing	Type	What to do with it	Purpose
McManageableView	Class	Use	Views help bridge the client portion of your application to the Management Central Java Framework, making programming easier for the developer. They provide the interfaces to manage your activities on the endpoint.
McManageableListView	Class	Use	List Views provide you with a way to retrieve multiple Views, using a single call to the server.

			This retrieval is based on criteria you set up with objects called selection criteria.
McManageableSelectionCriteria	Class	Use	Use this class in conjunction with McManageableListView to specify the type of manageables to retrieve. The selection criteria allows you to specify Owner, Type, Category, Sharing, etc.
McDistributedDescriptor	Class	Extend	You'll extend either the Task or Service subclass of McDistributedDescritptor.
McEndpointDescriptor	Class	Extend	You'll extend either the Task or Service subclass of McEndpointDescriptor
McActionIfc	Interface	Implement	This interface defines methods that must be implemented by your Action class. They allow the jMC to associate an instance of your Descriptor with your activity.
McActivityIfc	Interface	Use	This is the highest level interface that all distributed activities must support. It provides methods necessary for the jMC to manage your activity. In creating your activity, you'll extend an implementation of McActivityIfc.
McExecutable	Interface	Implement	This interface defines the methods that you may call on a distributed task. Use it when you want to begin the execution of a task or to cancel an executing task.
McSwitchable	Interface	Implement	This interface defines the methods that you may call on a distributed service. Use it when you want to start (turn on) a service or to stop (turn off) a running service.

In addition to what's available from the Management Central Java Framework, the ***AS/400 Toolbox for Java*** will play a key role in helping you develop your activity. For accessing everything from commands and programs, to data queues and user spaces, or job logs and message queues, the classes provided by the Toolbox can help your client or server code access objects specific to the AS/400 environment. Many of the examples in this document make use of the Toolbox, and a URL to their JavaDoc has been provided in the preface of this document.

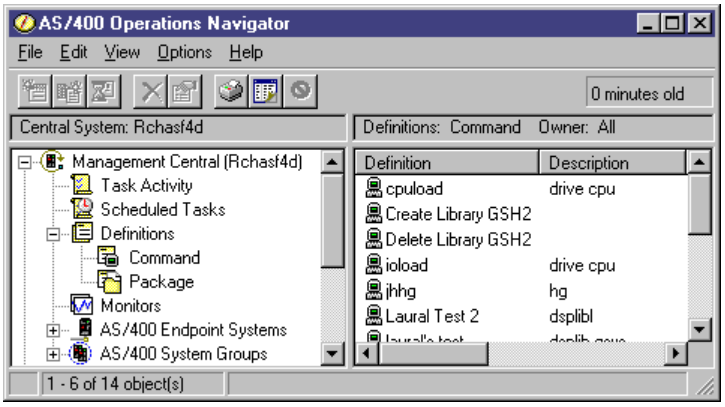
Management Central Definitions

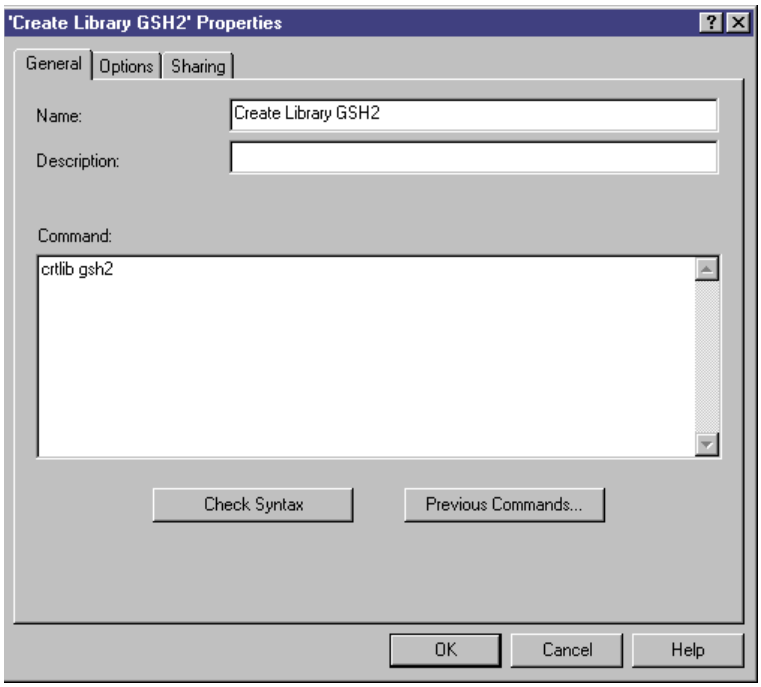
Overview



In this chapter, you will learn how to build your own application specific Definitions using the Management Central Java Framework. Definitions allow your application to store commonly used information on the Central System that you can share between users.

With V4R4, Management Central delivered two types of Definitions: Command and Package. As you can see from the AS/400 Operations Navigator window, there is a container called **Definitions** within **Management Central** that applications can plug their own definition containers into. In the case of command definitions, storing a command definition on the central system allows you to share commonly used or complex commands with other users. Depending on the definition sharing value, multiple users can select the command definition at any time, modify it, and then use and reuse the information to create a Task or Service and either run it immediately or schedule it to run at a later time.

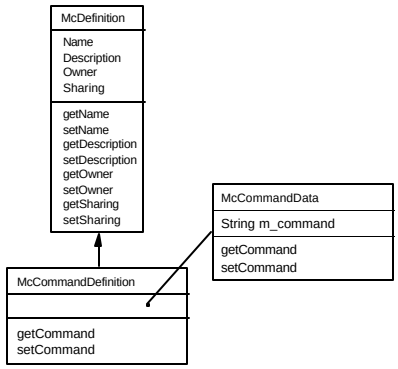




The information stored in the definition is up to the application designer. In the simple case of a command definition, we store the name, description, command, and a few other options related to running a command on an AS/400. When implementing definitions using the Management Central Java Framework, many of the attributes necessary for the definition, such as Name, Description, Owner and Sharing, have already been implemented by the jMC, and therefore are inherently free to the application developer.

Interface and Flow

Application Designer



As the application designer for the new definition, you need to determine what information needs to be saved with your definition. When you extend the **McDefinition** class, you automatically get Name, Description, Owner, and Sharing attributes along with other advanced features. In the case of this command definition example, you need to create a class to contain the command data member with getter and setter methods and a class to represent your definition. This data class (referred to as "application attributes") should be contained within your definition class. In this way, the jMC can easily manage the data associated with your new definition.

For the more advanced users, the application developer will also need to look at the default handling provided with the **McDefaultDefinitionContainer** and **McDefaultDefinitionPersistence** classes and interfaces. If the definition store on the AS/400 needs to be very specific to your application or interface with an existing application, then creating new classes that implement the **McContainerIfc** and/or **McPersistentIfc** interfaces would be needed. This would be the case if you need to control table definitions, support unique queries, or if you want to store your information in something other than a database.

Classes and Interfaces:

- com.ibm.mc.client.definition.McDefinition
- com.ibm.mc.server.container.McContainerIfc
- com.ibm.mc.server.container.McDefaultDefinitionContainer
- com.ibm.mc.server.persistence.McPersistentIfc
- com.ibm.mc.server.persistence.McDefaultDefinitionPersistence

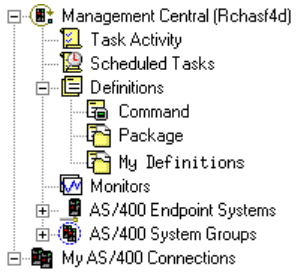
As the application designer, you will want to identify a number of different application specific design points when designing a new type of definition. Use the following checklist to assist you in your design.

Design checklist:

- ✓ What data or attributes do you need to store in your specific definition?
  Example: Name, Description, Owner, Sharing, and a Command

  Guidelines:
  - Think of data and information you want to store persistently on the Central System.
  - Don’t include System Groups or Endpoint Systems information in your definition, or information available via jMC base implementation, such as Name and Owner.

✓ Where in the Operations Navigator hierarchy do you want to see the list of your application's Definitions?



Guidelines:

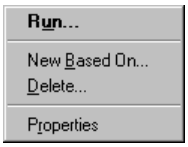
- See your UI Designer for help.

✓ What context menu options do you want on your definition Container?

Guidelines:

- If integrating into Operations Navigator, you will want the same behavior as other definition containers. This includes context menu options for *Explore*, *Open*, and *Create Shortcut*.
- See your UI Designer for help.

✓ What context menu options do you want on each of your Definitions?



Guidelines:

- If you choose to use the information stored in your definition to perform a task, then context menu options like *Run* or *Distribute* may be appropriate. This context menu action would then take the information out of the definition and use it as input when constructing the task.
- See your UI Designer for help.

Application Developer

The application developer will take the design specification and first create a class to hold all the application data or attributes, and second, extend the **McDefinition** class to create your new application definition class. From the design, the application developer may also need to extend the **McDefaultDefinitionContainer** and/or **McDefaultDefinitionPersistence** or may choose to implement the corresponding interfaces **McContainerIfc** and **McPersistentIfc**. This would only be necessary if the persistent storage and/or caching mechanisms provided by the Management Central Java Framework was not adequate.

Classes and Interfaces:

- com.ibm.mc.client.definition.McDefinition
- com.ibm.mc.server.container.McContainerIfc
- com.ibm.mc.server.container.McDefaultDefinitionContainer
- com.ibm.mc.server.persistence.McPersistentIfc
- com.ibm.mc.server.persistence.McDefaultDefinitionPersistence

Scenario 1: Defining a new class that contains all your application data

In this scenario you create a new Java class that represents the data or attributes for your application's definition. This class is commonly referred to as Application Attributes. This class needs to implement the java.io.Serializable interface so it can be sent to the central system and stored in a persistent manner.

```
package com.ibm.as400.opnav.McDefinitionSample;

import java.io.Serializable;

public final class MyCommandData implements Serializable
{
    private String m_command;

    public MyCommandData(String name) {
        m_command = name;
    }

    public String getCommand() {
        return m_command;
    }
```

Hints and Tips: All your application's definition data should be present in this data class, referred to as "Application Attributes". The Management Central Java Framework can easily manage your definition's data when you treat it as application attributes. See the JavaDoc for **com.ibm.mc.client.definition.McDefinition** for details. This data will be contained within your definition class that you'll create in the next step.

Scenario 2: Defining your Application's Definition

In this scenario you create a new Java class that represents your application's definition. After looking at the Management Central Java class **McDefinition** and it's base classes, you will see that Name, Description, Owner, and Sharing are all provided for you. When you extend **McDefinition**, all you need to add is public getter and setter methods hiding the containment relationship to your application data class. The *applicationData* field used in the getCommand method below, is defined by McDefinition's superclass, **McManageable**. It refers to the object you set as your application attributes. In this case, it's an instance of MyCommandData instantiated in the class constructor.

```
package com.ibm.as400.opnav.McDefinitionSample;

import java.io.Serializable;

import com.ibm.mc.client.definition.McDefinition;
import com.ibm.as400.opnav.McDefinitionSample.MyCommandData;

public final class MyCommandDefinition extends McDefinition
{

  public MyCommandDefinition(String name,         // Name for the
definition
                             String description,  // Brief description
                             int sharing,         // Sharing attributes
                             String command)      // AS/400 Command to
run
  {
    super(name, description, sharing, new MyCommandData(command));
```

Scenario 3: Advanced Features

Overriding the Default Persistence Mechanism for a Definition

By default, a new type of definition will have its instances stored in a similar manner to most other definitions. This storage mechanism is defined in the class:
com.ibm.mc.server.persistence.McDefaultDefinitionPersistence

In some cases, special circumstances may exist where the default persistence mechanism for definitions is not appropriate for a particular type of definition, e.g.:

- You require that your definitions be stored in a manner such that programs besides the Management Central Java Framework may access them. For example, your definitions are actually stored in a database table that is used by programs other than your Management Central application.
- Your type of definition is already stored persistently. The definition maps to a system object or some other persistent entity. Because of this persistence, you don't need the Management Central Java Framework to store your definition instances for you.

In cases like these, you need to replace the default persistence mechanism provided by Management Central and provide a persistence mechanism that you design and develop. To provide your own persistence mechanism for a new type of definition, you must do two things:

1. Create a new class that implements the **com.ibm.mc.server.persistence.McPersistenceIfc** interface. This class is your new persistence mechanism, and must map the functions provided in the McPersistenceIfc to the specifics of your persistence environment.
2. Override the getDefinitionPersistenceClass method in your new definition class. Your implementation of this method must return the fully-qualified name of the class you created in step 1. (The default implementation of this method returns the class "**com.ibm.mc.server.persistence.McDefaultDefinitionPersistence**"). An example override is shown below:

```
public final class MyCommandDefinition extends McDefinition
{
    .
    .
    .
    public static String getDefinitionPersistenceClass() throws
McException
    {
        return "com.mycompany.mypackage.MyDefinitionPersistence";
    }
```

Overriding the Default Container Behavior for a Definition

A definition object may exist in two forms on the server. The first form is its persistent form. A persistent representation of every definition will always exist on the central system. The second form is a transient Java object that represents the definition. The transient form of the object may go in and out of existence over time, as Management Central applications access it. There is some amount of overhead associated with the instantiation of the transient Java object from its persistent form. First the persistent form must be retrieved from the persistent store, then a transient Java object is created from this persistent representation.

By default, no caching is done of transient definition instances. This means that it will **always** access the persistent representation of the definition when performing queries or updates, and it will **always** have to instantiate transient Java object(s) when it returns one or more instances to an application. This default container behavior is defined in the class **com.ibm.mc.server.container.McNoCacheContainer**.

If performance becomes a concern when accessing your definitions, you have two choices. You can use the MC Java Framework's container implementation that automatically caches transient versions of your definition objects, or you may want to implement a more intelligent container mechanism of your own that provides a caching mechanism. This will eliminate the need to go to the persistent store and instantiate new transient objects for some operations. The jMC's caching container is defined in the class **com.ibm.mc.server.container.McCacheContainer**. To use it, skip step 1 below and specify

this class in step 2. If you don't want to use the McCacheContainer, you will need to provide a new container mechanism for your definition. To accomplish this, you must do two things:

1. Create a new class that implements the **com.ibm.mc.server.container.McContainerIfc** interface. This class is your new container mechanism, and must map the functions provided in the McContainerIfc to the specifics of your caching approach.
2. Override the getDefinitionContainerClass method in your new definition class. Your implementation of this method must return the fully-qualified name of the class you created in step 1. An example override is shown below:

```
public final class MyCommandDefinition extends McDefinition
{
    .
    .
    .
    public static String getDefinitionContainerClass() throws
McException
    {
        return "com.mycompany.mypackage.MyDefinitionContainer";
    }
```

GUI Developer

The GUI developer puts all the pieces together for the end user. A possible solution could be to add a new container in Operations Navigator under the Definitions branch of the Management Central tree, add a context menu option on this new container to create new definitions, and add context menu choices on each definition to view it's properties and to perform actions (e.g. *New Based On...* and *Delete*).

To perform this development you will need to know how to create an Operations Navigator Plug-in using **ListManager** and **ActionManager** interfaces and how to work with Management Central Java Framework classes **McDefinition**, **McDefinitionView**, and **McDefinitionListView**. Details on creating an Op-Nav plugin should be attained from the Operations Navigator project; the rest will be presented here.

Classes and Interfaces:

- com.ibm.mc.client.definition.McDefinition
- com.ibm.mc.client.definition.McDefinitionView
- com.ibm.mc.client.definition.McDefinitionListView

Scenario 1: Creating Definition Instances

In Scenario 1, you take your new application's data and definition classes, created by the Application Developer, and explore how to create instances of them. This would be the underlying function when a New Definition menu option is selected. Basically there are three steps in creating a new definition:

1. Create an instance of your application's definition. This could be triggered from the user interface where the user specifies different properties.
2. Create an instance of a Definition View to manage the instance of your definition
3. Tell the Definition View to persistently store the instance of your definition to the definition database on the Central System using the addManageable method.

```
 public void newDefinition() throws McException
{
   MyCommandDefinition thisDefinition     = null;
   McDefinitionView    thisDefinitionView = null;

   // Step 1: Create an instance of the MyCommandDefinition class
    thisDefinition = new MyCommandDefinition(
                        "MyDef",                                  // Name
                        "MyDef Description",                      // Description
                        McManageable.NONE,                        // Sharing
                        "SNDMSG MSG(Hi) TOUSR(poochie)");         // Command

   // Step 2: Create a Definition View object
   thisDefinitionView = new McDefinitionView(thisDefinition);
```

Scenario 2: Get a list of your Definitions

If you need to retrieve a list of the Definitions that you created in Scenario 1, this next scenario will give you the basics. There are only a few steps needed here:

1. Set up the selection criteria to only get definitions that match the class of your application's definition **MyCommandDefinition**.
2. Create an instance of a Definition List View to manage your list.
3. Ask the Definition List View to return you a list of definitions that matches the selection criteria you specified in step 1.

```
  public Vector listDefinitions() throws McException
  {
    McManageableSelectionCriteria selCriteria = null;
    McDefinitionListView          viewList    = null;
    Vector                        myDefViews  = null;

    // Step 1: Define selection criteria for a list of your definitions
    selCriteria = new McManageableSelectionCriteria(
                        "com.ibm.as400.opnav.McDefinitionSample.MyCommandDefinition",
// Class
                        McManageable.ALL,  // Category
                        null,              // Owner list only used when next parm is
true)
                        false,             // use sharing
                        0);                // Last date changed

    // Step 2: Create a new Definition List View object to manage your definitions
    viewList = new McDefinitionListView(selCriteria);
```

Scenario 3: Get asynchronous status updates for Lists of Definitions

By implementing the **McManageableListener** interface, you can be notified when a new definition has been created, changed, updated, or deleted. This is most useful when maintaining a list of definitions, and you wish to be notified whenever they are added, removed, updated, or changed. This list is defined using selection criteria so that you are not notified when just any definition is created, but only those that meet your selection subset. This code is identical to the code used to retrieve a list of definitions based on a selection subset, but we add a fourth step here to attach the current class as the ManageableListener. This interface, along with the implemented update, change, and remove methods, allows the Management Central Java Framework to send you a notification when an activity has been updated.

```
public class MyDefinitionList implements McManageableListener
{
  public void getList() {
    McManageableSelectionCriteria selCriteria    = null;
    McDefinitionListView          viewList       = null;
    Vector                        retrievedTasks = null;

    // Step 1: Define selection criteria to get a list of definitions
    selCriteria = new McManageableSelectionCriteria(
                        "com.ibm.as400.opnav.McDefinitionSample.MyCommandDefinition",
// Class
                        McManageable.ALL,  // Category
                        null,              // Owner list (only used when next parm is
true)
                        false,             // use sharing
                        0);                // Last date changed


    // Step 2: Create a new Definition List View to manage your tasks
    viewList = new McDefinitionListView(selCriteria);

    // Step 3: Ask the Distributed Definition Manager for a list of Distributed
Command Tasks
    retrievedTasks = viewList.getManageableViews();

    // Step 4: Attach this class, which implements the McManageableListener interface,
    //         to handle any notifications.
    viewList.attachManageableListener(this);
  }

  // The following methods are required as an implementation of McManageableListener

  public void manageableAdded(McManageableEvent event) throws McException {
    // Insert code to handle when a new definition has been created
  }

  public void manageableChanged(McManageableEvent event) throws McException {
    // Insert code to handle when a definition has changed.  A "change" is a
    // user-directed property change, such as changing the definition's description.
  }
```

Scenario 4: Deleting Definition Instances

If you need to delete a definition instance, or a list of them, the Definition View and List View classes provide the methods for you. Deleting a definition removes the definition from the Management Central databases and is no longer a managed definition. When working with the View object, you can simply call removeManageable on the instance of the object itself; for a List of Views, you can simply call removeManageableList on the ListView instance.

```
// Step 1: Tell the Definition View to remove the instance of your task
view.removeManageable();

// Or, for a Distributed Task List View
```

Scenario 5: Changing an existing Definition Instance

To change an existing definition, the View again provides the method for you to use. Prior to calling change, you would have a reference to a definition that you previously created or retrieved from a list. With the reference to the definition, you may have the end user modify it by displaying a set of property pages and using the appropriate set methods to update the instance of the definition. When you have the definition instance updated, you can tell the View to store the changes.

```
// Step 1: Call the set methods to update the Definition
thisDefinition.setCommand("SNDBRKMSG MSG('message') TOMSGQ(QPADEV000H)");

// Step 2: Tell the Definition View to change your Definition with the updates
```

Hints and Tips: Remember that the Management Central Java Framework can easily clone and manage your definition's data when you treat it as Application Attributes. See the JavaDoc for **McDefinition** for details. If you have any data members in your application's definition class, it will not be changed or saved.

NOTE: In addition to all the definition features documented above, there are additional advanced features that apply to definitions that you should be aware of. Since they are features that also apply to activities (and not just definitions), they have been documented in their own chapter, affectionately entitled *Advanced Features*, beginning on page 54. Please note that while all advanced scenarios apply to activities, only scenarios 2-4 apply to definitions.

Terms, Classes, and Interfaces

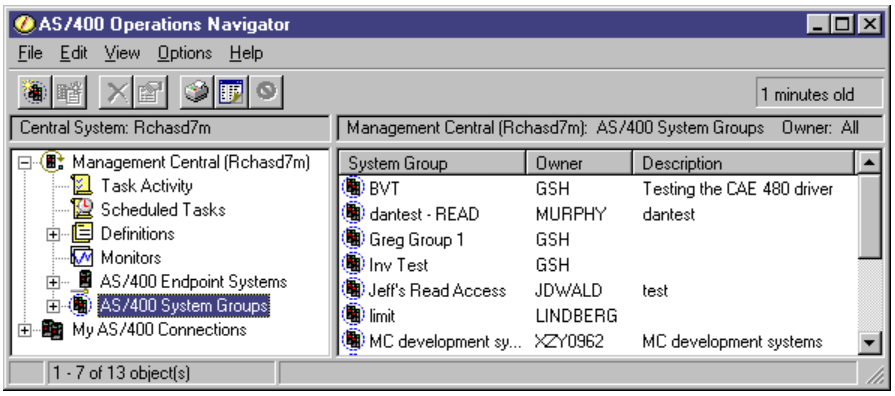
Thing	Type	What to do with it	Purpose
McDefinition	Class	Extend	When you create a new type of definition, you must inherit from this class to get the data attributes and method implementations that all definitions need.
McDefinitionView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate definitions and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when the definition has changed.
McDefinitionListView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate lists of definitions and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a definition has been created, changed, updated, and deleted.
McContainerIfc	Interface	Implement	When your application defines a new type of definition, you may want to implement a special container for your definition objects. A container controls the policy of accessing objects that are persistently stored and could support more sophisticated queries. Within the container, you can decide when it is appropriate to go to the persistent store and when it may be appropriate to use an "in memory" definition object. By default, new types of definitions use a default container implementation called McDefaultDefinitionContainer. If you write your own container, you must identify it to the Management Central Java Framework by providing a static method on your definition class called getDefinitionContainerClass. See the McDefinition class for the signature of the static method.
McPersistentIfc	Interface	Implement	When your application defines a new type of definition, you may want to implement a special persistence object to store your definitions. Your own persistence object allows you to control the persistence mechanism for your definition objects. This would allow you to store them in their own database or some other kind of persistence. By default, new types of definitions use a default persistence implementation called McDefaultDefinitionPersistence. This stores your definitions in the same database as other types of definitions and only allows you to perform operations that are defined in this interface. If you write your own persistence class, you must identify it to the Management Central Java Framework by providing a static method on your definition class called getDefinitionPersistenceClass. See the McDefinition class for the signature of the static method.

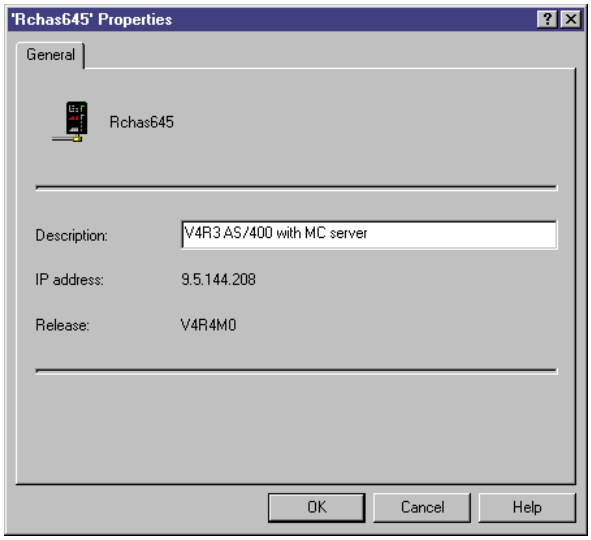
Hints and Tips

This space has been included so that you can document your own special hints and tips:

Management Central Endpoint Systems and System Groups

One example where **McDefinition** classes are used is in the construction of Endpoint Systems and System Groups. These can be viewed using the existing AS/400 Endpoint Systems and AS/400 System Groups containers.

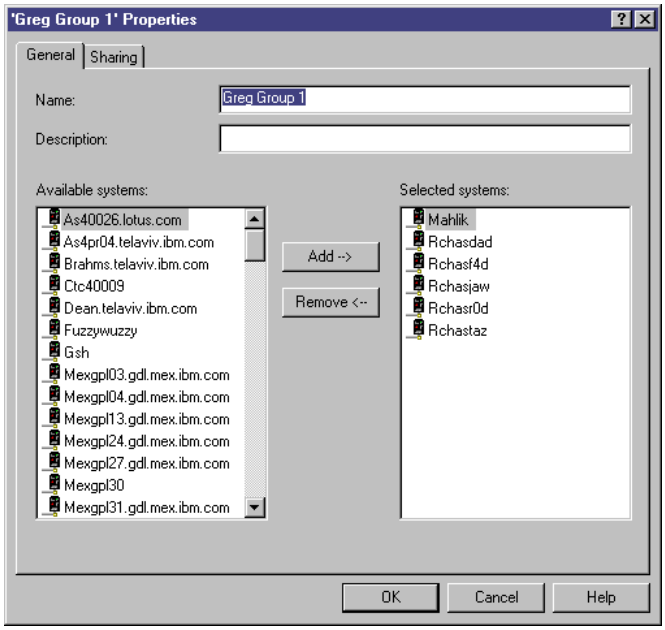




The **McEndpointSystem** class extends the **McDefinition** class and adds attributes, like Protocol, IP Address, Release, and Operating System, that further define an AS/400 system. Since **McDefinition** extends **McManageableData** (which contains Name, Description, Owner, and Sharing), the **McEndpointSystem** class doesn't need to provide any additional code to support these fields. They can simply be used when constructing an instance of a **McEndpointSystem**.

The **McSystemGroup** class is a little different from the **McEndpointSystem** class in that it extends the **McCompositeDefinition** class. This class in turn extends **McDefinition** and implements **McComposeable**. Implementing the **McComposeable** interface gives a system group the ability to add, remove, find, and get children objects.

System Groups are used when executing a task or service. A System Group may contain one or more Endpoint Systems, and, because it implements **McComposeable**, can also contain other System Groups.



Programming Example

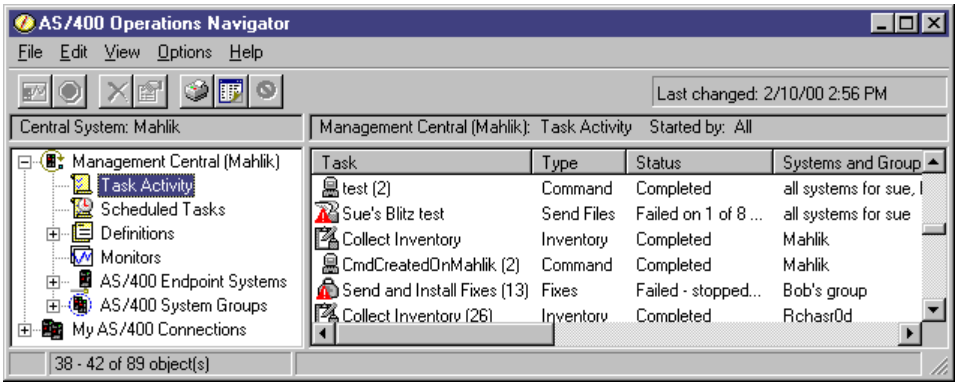
The following programming example provides a snippet of how you could create a Management Central System Group when passed an AS/400 Java Toolbox AS400 object. Once you have the System Group containing the one Endpoint System specified in the AS400 object, you can use it to start a Service or execute a Task.

```
public McSystemGroup toMcSystemGroup(AS400 as400Obj)
{
  McEndpointSystem s1      = null;
  McSystemGroup    group   = null;

  try {
    s1 = new McEndpointSystem(as400Obj.getSystemName(), // Name
                              "",                       // Description
                              McManageable.NONE,        // Sharing
                              "");                      // IP Address

    // Create a new System Group specifying the Endpoint System
    group = new McSystemGroup("Temporary System Group",  // System Group Name
                              "Temporary System Group",  // Description
                              McManageable.NONE,         // Sharing
                              s1);                       // Endpoint System
  }
  catch (McException e) {
    System.out.println("McException:" + e);
    System.exit(1);
```

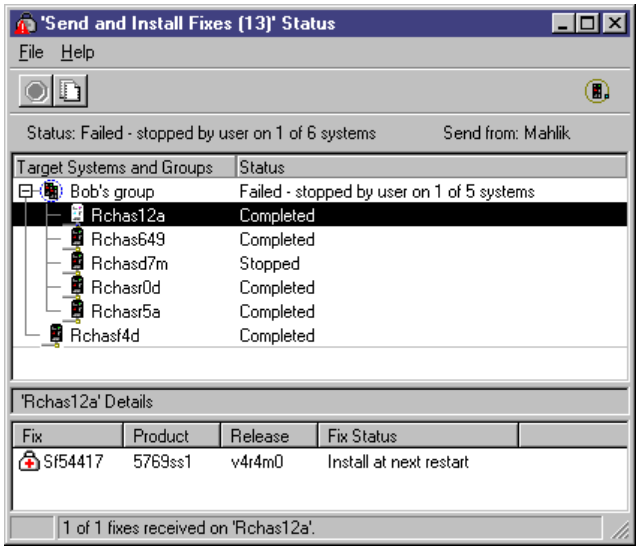
You can also expand the above example to allow a Vector of AS400 objects to be specified, loop through the vector to get each Endpoint System and add it to a vector of systems, and return the System Group containing multiple Endpoint Systems.

Management Central Distributed Tasks

Overview



Tasks are long running asynchronous operations that can be scheduled and run unattended on multiple remote systems. The operator or administrator generally selects an action to perform from the graphical user interface, selects which systems to perform the action, and then determine whether to run the action now or schedule it to be run at a later date and time. After the action completes at the endpoint system, it sends status information back to the central system and can be later viewed on the workstation.



One example of a task is the Fixes application that was delivered with Management Central in V4R4. One function of this application allows the operator to distribute and install Program Temporary Fixes(PTFs) on other systems in the network. The operator would select a list of fixes to install and then pick which system groups or systems to install them on. During the distribution and installation process, status information would be sent back to the central system. If and when the operator wanted to check on the task, she could simply click on the Task Activity container and view the detailed status information for that task. Tasks will always have some final status whether it completed successfully, failed, or was ended by the user.

Currently the Management Central Java Framework provides three different ways for applications to implement tasks. The first method, and simplest, is to use the Distributed Command Call Application provided with Management Central. The Distributed Command Call Application allows you to execute an AS/400 CL command on a group of systems. If all or part of your application can be performed simply by sending a command to the remote system all you need to do is use the **McDistCommandDescriptor** class. This class takes a CommandCall object which you construct from the AS/400 Java Toolbox and a **McSystemGroup** to indicate where to execute the AS/400 CL command. For more details see section Management Central Command Call Application on page 65.

If instead of calling a command you need to call an AS/400 program or service program, you can use the second way to implement tasks by using the Distributed API Application. By constructing Program Call Markup Language(PCML) statements and using the ProgramCallDocument from the AS/400 Java Toolbox, you can create tasks using the **McDistApiDescriptor** class. The **McDistApiDescriptor** class also accepts a ProgramCall class or ServiceProgramCall class from the AS/400 Java Toolbox. For more details see section Management Central Distributed API Application on page 74.

A benefit of using either of the two aforementioned applications is that you won't need to write any server code; rather, you'll make use of classes implemented by the jMC, and simply write a user interface to interact with the server. However, if neither of the provided applications meet your needs, then you can create your own task by extending the Management Central Java Framework. The following descriptions and scenarios will help to explain how to create your own task.

Interfaces and Flows

Application Designer

As the application designer, you will need to determine what functions or actions your application needs to perform on the endpoint systems and what data or information is needed to perform that action. The **McDistributedTaskDescriptor** is provided to give you a way to describe exactly how you want your distributed task to behave. When you extend the **McDistributedTaskDescriptor** class, you create a new kind of distributed task which will inherit Name, Description, Owner, and Sharing data members and methods. You only need to supply the data specific to your application.

Classes and Interfaces:

- com.ibm.mc.client.activity.task.McDistributedTaskDescriptor
- com.ibm.mc.client.activity.task.McEndpointTaskDescriptor
- com.ibm.mc.client.activity.task.McExecutable
- com.ibm.mc.server.activity.McActionIfc

Design checklist:

- ✓ What data do you need to send to the remote endpoint systems?
  Example: Name, Description, Owner, Sharing, and a Command

  Guidelines:
  - Think of data and information you need in order to perform your task. This information is considered the “Application Attributes” of the task.
  - Don’t include System Groups, Endpoint Systems, Status, or Results information in your task. This is already built into the task objects. Also, omit information available via jMC

base implementation, such as Name and Owner.

- ✓ Once you have the data at the endpoint system, what action to do want to perform?

  Guidelines:
  - In your class that implements the McExecutable interface you will need to decide what function to perform in the execute method and the cancel method. This is where your custom logic goes.

- ✓ Where in the Operations Navigator hierarchy do you want to see the list of your application's tasks? Also, if your application creates multiple type of tasks do you want to filter out some or do you want to show all of them?

  Guidelines:
  - See your UI Designer for help.

- ✓ What context menu options do you want on your Task Container?

  Guidelines:
  - If integrating into Operations Navigator, you will want the same behavior as other task containers. This includes context menu options for *Explore*, *Open*, and *Create Shortcut*.
  - See your UI Designer for help.

- ✓ What context menu options do you want on each or your Application's Task?

  Guidelines:
  - If you choose to use the information stored in your definition to perform a task, then context menu options like *Run* and *Distribute* may be specified. This context menu action would then take the information out of the definition and use it as input when constructing a distributed task.
  - See your UI Designer for help.

Application Developer

The application developer takes the design specification and will need to create at least four classes for a distributed task. First, you need to create a class to contain all your application data and attributes. This class will be termed the "Application Attributes" of your activity. The jMC will automatically store these attributes persistently on the Central System. Second, you need to extend the **McDistributedTaskDescriptor** class to create your application specific Distributed Task Descriptor class. Next, you need to extend the **McEndpointTaskDescriptor** class to create your application specific Endpoint Task Descriptor. This class will have the ability to set and retrieve your Application

Attributes. And finally, you need to create a class which implements the **McActionIfc** and **McExecutable** interfaces for your application specific action that will execute on the endpoint system.

Classes and Interfaces:

- com.ibm.mc.client.activity.task.McDistributedTaskDescriptor
- com.ibm.mc.client.activity.task.McDistributedTaskDescriptorIfc
- com.ibm.mc.client.activity.task.McEndpointTaskDescriptor
- com.ibm.mc.client.activity.task.McEndpointTaskDescriptorIfc
- com.ibm.mc.client.activity.task.McExecutable
- com.ibm.mc.server.activity.McActionIfc

In the following scenarios, you will create the basic classes that represent your application's task.

Scenario 1: Create a new class that contains all your application data

In this scenario you create a new Java class that represents the data or attributes for your application's task. This class needs to implement the java.io.Serializable interface so it can be sent to the central system and stored in a persistent manner.

> **Hints and Tips:** Note the following MyCommandData class was the same you used in the definition examples. When you create a task based on a definition, it is easy to reuse the Application Attributes class to supply all the data from the definition to the task.

```
package com.ibm.as400.opnav.MyTaskSample;

import java.io.Serializable;

public final class MyCommandData implements Serializable
{
    private String m_command;

    public MyCommandData(String name) {
        m_command = name;
    }

    public String getCommand() {
        return m_command;
    }
```

Hints and Tips: All your application's task data should be present in this data class and should not be placed as local data members within the Distributed Task Descriptor class. The Management Central Java Framework can easily manage your application's task data when you treat it as Application Attributes.

Scenario 2: Create a new class that extends the McDistributedTaskDescriptor class

By extending the **McDistributedTaskDescriptor** class, you inherit data members and methods for Name, Description, Owner, etc.. In this example, all you need to provide is your public getter and setter methods and implement the createEndpointData method defined in the **McDistributedActivityDescriptorIfc** interface class. The createEndpointData method is where you create an instance of your Endpoint Task Descriptor and give it to the Management Central Java Framework. Also, since endpoint descriptors are for the specific execution of the task, you need to call setPrivate(true) on your descriptor before returning from the method. Doing so will prevent anyone else from sharing your endpoint descriptor, and it will notify the MC Java Infrastructure that the descriptor should not be stored persistently. The *coordinator* field used to add event listeners is defined by McDistributedTaskDescriptor's superclass, McActivityDescriptor. It refers to an event coordinator that processes events for this descriptor.

Hints and Tips: Remember you only need to include the data that is unique to your task. Information like System Group, Status, and Results are already provided for you in the Distributed Task Descriptor and associated classes.

```
package com.ibm.as400.opnav.MyTaskSample;

public class MyDistributedCommandCallDescriptor extends McDistributedTaskDescriptor
{
  // Constructors
  public MyDistributedCommandCallDescriptor(
                                String theName,
                                String theDescription,
                                int theSharing,
                                McSystemGroup theSystemGroup,
                                CommandCall applicationData) throws McException
  {
    super(theName, theDescription, theSharing, theSystemGroup, applicationData);
    coordinator.addEventListener(new McStatusAspect(), new
McDefaultStatusEventHandler());
    coordinator.addEventListener(new McResultAspect(), new
McDefaultResultEventHandler());
  }

  // Implement methods from the McDistributedActivityDataIfc interface
  public McEndpointActivityDescriptorIfc createEndpointData() throws McException
  {
    MyEndpointCommandCallDescriptor eptData =
                new MyEndpointCommandCallDescriptor(getName(),
                                                    getDescription(),
                                                    getSharing(),
                                                    getCommand());
    eptData.setPrivate(true);
    return eptData;
  }
```

NOTE: You may have noticed that in the code example above you're using a **MyEndpointCommandCallDescriptor** class which you will create next.

Scenario 3: Create a new class that extends the McEndpointTaskDescriptor class

This next piece of code is to define the Endpoint Task Descriptor. Like the Distributed Task Descriptor, this class also contains the data for your application task. In this case, the data is used on the endpoint system and needs to contain any information that your execute and cancel action methods might need. Like the previous example, you start out by defining any private data members you need with getter and setter methods. Next, you need to implement the getActivityActionClass method defined in the **McActivityDescriptorIfc** interface. As you can see, here is where you tell the Management Central Java Framework the class which implements the **McExecutable** interface which contains the execute and cancel methods.

```
package com.ibm.as400.opnav.MyTaskSample;

public class MyEndpointCommandCallDescriptor extends McEndpointTaskDescriptor
{
  // Constructors
  public MyEndpointCommandCallDescriptor(
                                String theName,
                                String theDescription,
                                int theSharing,
                                CommandCall applicationData) throws McException
  {
    super(theName, theDescription, theSharing, applicationData);
    coordinator.addEventListener(new McStatusAspect(), new
McDefaultStatusEventHandler());
    coordinator.addEventListener(new McResultAspect(), new
McDefaultResultEventHandler());
  }

  // Implement methods from the McActivityDescriptorIfc interface
  public String getActivityActionClass()
  {
```

Scenario 4: Create a new class that implements the McExecutable interface

The fourth part involves implementing methods for the **McActionIfc** interface and the **McExecutable** interface. The **McActionIfc** interface includes the setObserver, setDescriptor, setThread, descriptorChange, and descriptorRemove methods. The **McExecutable** interface includes the execute and cancel methods. For details on these methods, see the JavaDoc.

```
package com.ibm.as400.opnav.MyTaskSample;

public class MyEndpointCommandCallAction implements McActionIfc, McExecutable
{
  private McRemoteListener           observer = null;
  private McEndpointTaskDescriptorIfc data     = null;
  private McMethodThreadIfc          thread   = null;

  // Implement methods from the McActionIfc interface
  public void setObserver(McRemoteListener observer) throws McException
  { // set observer to local member variable
    this.observer=observer;
  }

  public void setDescriptor(McActivityDescriptorIfc data) throws McException
  { // set data to local member variable
    if (data instanceof MyEndpointCommandCallDescriptor)
      this.data = (MyEndpointCommandCallDescriptor)data;
    else
      ; // handle Error
  }

  public void setThread(McMethodThreadIfc thrd) throws McException
  { // set thrd to local member variable
    if( (this.thread == null) && (thrd != null) )
      this.thread = thrd;
    else
      ; // handle Error
  }

  public void descriptorChange() throws McException
  {}

  public void descriptorRemove() throws McException
  {}
```

GUI Developer

The GUI developer connects the user interface to the Distributed Task Descriptor class created by the application developer. One possible solution could be to add a new task container in Operations Navigator under the Task Activity branch of the Management Central tree, adding a context menu option on this new container to create new tasks, and adding context menu choices on each task to view it's properties and to perform actions (i.e. New Based On, View Status, etc.).

To perform this development you will need to know how to create an Operations Navigator Plug-in using ListManager and ActionManager interfaces, how to work with jMC classes **McDistributedTaskDescriptor**, **McDistributedTaskView**, and **McDistributedTaskListView**, and how to use the GUI helpers provided by the Management Central Java Framework to display properties, select systems and groups, and to delete your application.

Classes and Interfaces:

- com.ibm.mc.client.activity.task.McDistributedTaskDescriptor
- com.ibm.mc.client.activity.task.McDistributedTaskView
- com.ibm.mc.client.activity.task.McDistributedTaskListView

Scenario 1: Create and Execute an instance of your new Application Task

In scenario 1, you take your new set of application task classes and explore how to create instances of them and to execute your task. This would be the underlying function when a New Task menu option is selected. Basically there are four steps in creating and executing a new task.

1. Create an instance of your application's distributed task descriptor specifying the task name, task description, sharing, system group, and task data information. The properties of the task would be retrieved from GUI dialogs or wizards prompting the user for the information. Note that the getSystemGroup and getCommand methods used in the Descriptor's constructor must be supplied by the user to retrieve the list of endpoint systems on which to execute the command, and get the application attributes for the task.
2. Create an instance of a Distributed Task View specifying the application's Distributed Task Descriptor created in step 1.
3. Tell the Distributed Task View to add the instance of your task so that it can be managed.
4. Call the execute method to distribute and execute your application task on all the endpoint systems specified in the System Group.

```
public void newTask() throws McException
{
  McDistributedTaskDescriptorIfc distCommandDesc = null;
  McDistributedTaskView          distTaskView    = null;

  // Step 1: Create an instance of your Application Distributed Task Descriptor
  distCommandDesc = new MyDistributedCommandCallDescriptor(
                        "MyAppTask",             // Name
                        "MyAppTask Description", // Description
                        McManageable.NONE,       // Sharing
                        getSystemGroup(),        // User method to get System
Group
                        getCommand());           // User method to get command

  // Step 2: Create an instance of a Distributed Task View
  distTaskView= new McDistributedTaskView(distCommandDesc);

  // Step 3: Tell the Management Central Java Framework Task Manager to add this
```

Scenario 2: Get a list of your Application Task Instances

If you need to retrieve a list of the tasks that you created in Scenario 1, this next step will show you how. There are only a few steps needed here.

1. Set up the selection criteria to only get tasks of the class **MyDistributedCommandCallDescriptor**
2. Create an instance of the **McDistributedTaskListView** to manage the list of tasks
3. Ask the Distributed Task List View to return to you a list of your application's Distributed Tasks

```
public Vector listTasks() throws McException
{
  McManageableSelectionCriteria selCriteria  = null;
  McDistributedTaskListView     distTaskView = null;
  Vector                        myTaskList   = null;

  // Step 1: Define selection criteria to get a list of your application
  selCriteria = new McManageableSelectionCriteria(

"com.ibm.as400.opnav.MyTaskSample.MyDistributedCommandCallDescriptor",
                      McManageable.ALL, // Category
                      null,             // List of owners (only used if next parameter
is true
                      false,            // Include shared tasks
                      0);               // Check the last changed date of the task

  // Step 2: Create a new Task List View to manage your application tasks
```

Scenario 3: Get asynchronous status and results of an Activity

Once you've created and started your activity, it will run asynchronously with other activities. If you want to monitor the status of the request, you will want to attach a class to handle when status and results are returned from the endpoint system to you. This class will implement the **McStatusDetailListener** and **McResultDetailListener** interfaces. Note: In the example shown below, the same class implements both these interfaces so we pass in this as the object to handle status and result updates.

```
      // Previously you would have retrieved a list of task views and
    selected the
      // one task view that you want to monitor status and results

      // Attach to be notified when a Status or Result object is received
      view.attachStatusDetailListener(this);
      view.attachResultDetailListener(this);

      // Display a status window or dialog, or turn service on




      public void statusUpdate(McStatusEvent event) throws McException
      {
        // Get the status object out of the event information
        McStatusIfc status = event.getStatus();

        // If the overall status value indicates the task has finished
        if ( status.getLevel() == McStatusIfc.DistributedAct &&
    status.isFinalized() )
        {
```

```
public void resultUpdate(McResultEvent event) throws McException
{
  // Get the result object out of the event information
  McResultIfc result = event.getResult();

  // Since the result object is a hierarchy of results for each Endpoint System
  // specified in the task, you need to get the results for the specific
Endpoint
  // System to see its details.
  McResultIfc childResult = (McResultIfc)(result.findChild("system1"));

  // Be sure that the result object is an instance of ProgramCall before
  // performing ProgramCall type methods.
  if(childResult != null && childResult.getResultData() instanceof ProgramCall)
  {
    // Get the ProgramCall object our of the result object
    ProgramCall pgm = (ProgramCall)childResult.getResultData();

    if ( (pgm.getMessageList() != null) && (pgm.getMessageList().length > 0) )
    {
      // Retrieve list of AS/400 messages
      AS400Message[] msgs = pgm.getMessageList();
```

Hints and Tips: The resultUpdate method above is expecting result data of type ProgramCall, but the getResultData method generically returns an Object of type Serializable. You could have easily cast the result data to whatever kind of an object your specific application expects.

The MC Java Framework also allows you to easily implement your own Status and Result objects to use for your application. All you need to do is create an object that extends either **McTaskStatus**, **McResult**. If you don't implement your own status and result objects, these default types will be used, but if you do decide you need extensions of the functionality available, you only need to extend what's applicable. For instance, if you're implementing a task, and only need your own status type, you only need to extend McTaskStatus, and can use the default McResult implementation.

Once you've created your objects, you'll need to override the createStatus and/or createResult methods in your descriptor objects (both endpoint and distributed) to return an object of your new status or result type. The jMC will automatically invoke these create methods when it needs a status or result object.

Scenario 4: Get asynchronous status updates for Lists of Tasks

By implementing the **McManageableListener** interface, you can be notified when a new task has been created, changed, updated, or deleted. This is most useful when maintaining a list of activities, and

you wish to be notified whenever they are added, removed, updated, or changed. This list of tasks is specified using selection criteria so that you are not notified when just any activity is created, but only those tasks that meet your selection subset. This code is identical to the code used to retrieve a list of task based on a selection subset, but we add a fourth step here to attach the current class as the ManageableListener. This interface, along with the implemented update, change, and remove methods, allows the Management Central Java Framework to send you a notification when an activity has been updated.

```
public class MyTaskList implements McManageableListener
{
  public void getList() {
    McManageableSelectionCriteria selCriteria    = null;
    McDistributedTaskListView     viewList       = null;
    Vector                        retrievedTasks = null;

    // Step 1: Define selection criteria to get a list of Distributed Command
Tasks
    selCriteria = new McManageableSelectionCriteria(

"com.ibm.mc.client.activity.task.command.McDistCommandDescriptor",
                  "McManageable.ALL",    // category
                  null,           // Owner List (only used when next parm is true)
                 false,           // use sharing
                  0);                    // last changed date

    // Step 2: Create a new Task List View to manage your tasks
    viewList = new McDistributedTaskListView(selCriteria);

    // Step 3: Ask the Distributed Task Manager for a list of Distributed Command
Tasks
    retrievedTasks = viewList.getManageableViews();

    // Step 4: Attach this class, which implements the McManageableListener
interface,
    //         to handle any notifications.
    viewList.attachManageableListener(this);
  }

  // The following methods are required as an implementation of
McManageableListener

  public void manageableAdded(McManageableEvent event) throws McException {
    // Insert code to handle when a new task has been created
  }

  public void manageableChanged(McManageableEvent event) throws McException {
    // Insert code to handle when a definition has changed.  A "change" is a
    // user directed property change, such as changing the definition's
```

Scenario 5: Delete an Application Task Instance

If you need to delete a task, or a list of task, the Distributed Task View and List View classes provides the methods for you. Deleting a task removes the task from the Management Central databases and is no longer a manageable task. When working with the Distributed Task View object, you can simply

call removeManageable method on the instance of the object itself; for a List of Views, you can simply call removeManageableList on the ListView instance.

```
// Step 1: Tell the Distributed Task View to remove the instance of your
service
distServiceView.removeManageable();
```

Scenario 6: Change an Application Task Instance

To change an existing task, the Distributed Task View provides the method for you to use. Prior to calling change, you would have a reference to a task that you previously created or retrieved from a list of tasks. With the reference to the task, you may have the end user modify it by displaying property pages and using the appropriate set methods to update the task instance. Now when you have the task instance up to date, you can tell the Distributed Task View to store the changes.

```
// Step 1: Change the Distributed Task Descriptor locally
distCommandDesc.setDescripton( "New Descripton" );

// Step 2: Tell the Distributed Task View to change the Descriptor on the
```

Scenario 7: Schedule your task

It is very easy for the GUI developer to use the Descriptor and View classes to create and schedule a distributed task. This scenario is very similar to distributing a command call to run on multiple endpoints. The difference is instead of calling execute, you now need to gather scheduling information as the fifth step and call schedule as an additional step.

1. Create an instance of a Distributed Command Descriptor specifying the Task Name, Task Owner, Task Description, Sharing, System Group and command.
2. Create an instance of a Distributed Task View object specifying the Distributed Command Descriptor created in step 1.
3. Tell the Distributed Task View to add the instance of your task so that it can be managed.
4. Construct a **McScheduleInfo** object using a description of your activity, and "execute" as the scheduled method, and set the schedule information by prompting the user with the Management Central Schedule Dialog or a supported Business Partner Scheduler.
5. Call the schedule method to schedule the task on the Central System.

```
McDistributedTaskDescriptorIfc distCommandDesc = null;
McDistributedTaskView          distTaskView    = null;

// Step 1: Create an instance of your Application Distributed Task Descriptor
distCommandDesc = new MyDistributedCommandCallDescriptor(
                      "MyAppTask",             // Name
                      "MyAppTask Description", // Description
                      McManageable.NONE,       // Sharing
                      getSystemGroup(),        // User method to get System Group
                      getCommand());           // User method to get command

// Step 2: Create an instance of a Distributed Task View
distTaskView = new McDistributedTaskView(distCommandDesc);

// Step 3: Tell the Management Central Java Framework Task Manager to add this
//         Application Disributed Task Descriptor to manage.
distTaskView.addManageable();

// Step 4: Set the Schedule Information
```

Scenario 8: Retrieve your scheduled tasks

If you need to retrieve a list of previously Scheduled Distributed Tasks, this next step will show you how. Refer to the previous section where you Scheduled your task to execute at a later date. If that task is currently managed by the Management Central Java Infrastructure, it will be returned in your list of scheduled tasks below. There are only a few steps needed here.

1. Set up the selection criteria to only get scheduled tasks of the class **MyDistributedCommandCallDescriptor**. This time you needed to use the **McActivityDescriptorSelectionCriteria** class instead of the **McManageableSelectionCriteria**. The activity selection criteria extends the capabilities of the manageable selection criteria to include status information. This allows you to indicate that you only want to receive activities that are in a particular status, such as active, completed, or in this case scheduled.
2. Create an instance of the **McDistributedTaskListView** to manage the list of tasks
3. Ask the Distributed Task Manager to return to you a list of Distributed Command Call Tasks

```
McActivityDescriptorSelectionCriteria selCriteria = null;
Vector retrievedSchedTasks = new Vector();
int[] statusList = {McStatusIfc.Scheduled};

// Step 1: Define selection criteria to get a list of  Scheduled Distributed
//         Command Call Tasks
selCriteria = new McActivityDescriptorSelectionCriteria(

".ibm.as400.opnav.MyTaskSample.MyDistributedCommandCallDescriptor",
                      McManageable.ALL,  // Category
                      null,              // OwnerList (only valid if next parm is
true)
                      false,                    // useSharing
                      0,                 // last changed date
                      statusList);       // StatusList - ours contains only
```

NOTE: In addition to all the task features documented above, there are additional advanced features that apply to tasks that you should be aware of. Since they are features that apply to activities in general (and some apply to definitions), they have been documented in their own chapter, affectionately entitled *Advanced Features*, beginning on page 54.

Terms, Classes, and Interfaces

Thing	Type	What to do with it	Purpose
McDistributedTaskDescriptor	Class	Extend	An application will create one (or more) subtype(s) of this class to define how their multi-system task will execute, what data will be associated with it, and how it will handle events on the central system. Your subtype(s) of this class will describe most of the parts of your application.
McDistributedTaskDescriptorIfc	Interface	Use	The interface that all distributed task descriptors will support.
McDistributedTask	Class	Use	Objects belonging to this class represent the actual, executing task. You may operate on distributed task objects using the McExecutable interface. All distributed tasks have a distributed activity descriptor associated with them. The task refers to its descriptor to get information about how it should execute. You are not responsible for instantiating this object. A distributed task object can be obtained from a distributed task descriptor object that is being managed by the Management Central Java Framework.
McExecutable	Interface	Implement	This interface defines the methods that you may call on a distributed task. Use it when you want to begin the execution of a task or to cancel an executing task.
McEndpointTaskDescriptor	Class	Extend	An application will create one (or more) subtype(s) of this class to define how their task will execute, what data will be associated with it, and how it will handle events. on the endpoint.
McEndpointTaskDescriptorIfc	Interface	Use	The interface that all endpoint task descriptors will support.
McActionIfc	Interface	Implement	You must provide a class that implements this interface to perform the endpoint action that is part of your task. Your object will perform the

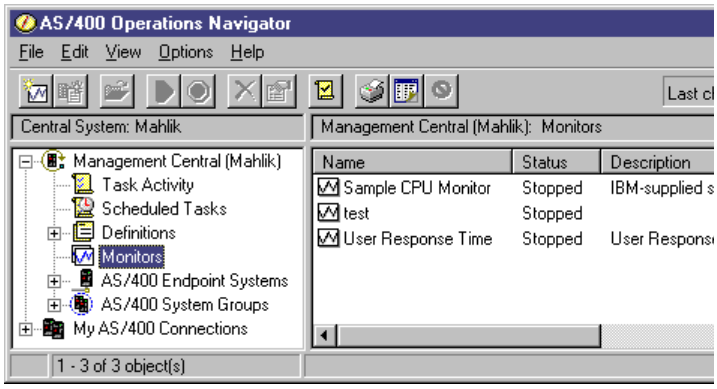
			"real work" of making the task execute on the endpoint, and the Management Central Java Framework will call the object at the appropriate time.
McDistributedTaskView	Class	Use	This class bridges your application to the MC Java Framework functions to manipulate tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a status has changed. Task Actions: - execute - cancel Distributed Task Manager: - addManageable - getManageable - changeManageable - removeManageable Event Listeners: - attachStatusDetailListener - detachStatusDetailListener - attachResultDetailListener - detachResultDetailListener - attachConnectionListener - detachConnectionListener
McDistributedTaskListView	Class	Use	This class bridges your application to the MC Java Framework functions to manipulate lists of tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a new task has been added, changed, or removed from a list of tasks. Distributed Task Manager: - getManageableViews Event Listeners: - attachManageableListener
McManageableSelectionCriteria	Class	Use	Use this class in conjunction with the McDistributedTaskListView to specify the type of tasks to retrieve. The selection criteria allows you to specify Type, Category, Sharing, etc.

Hints and Tips

This space has been included so that you can document your own special hints and tips:

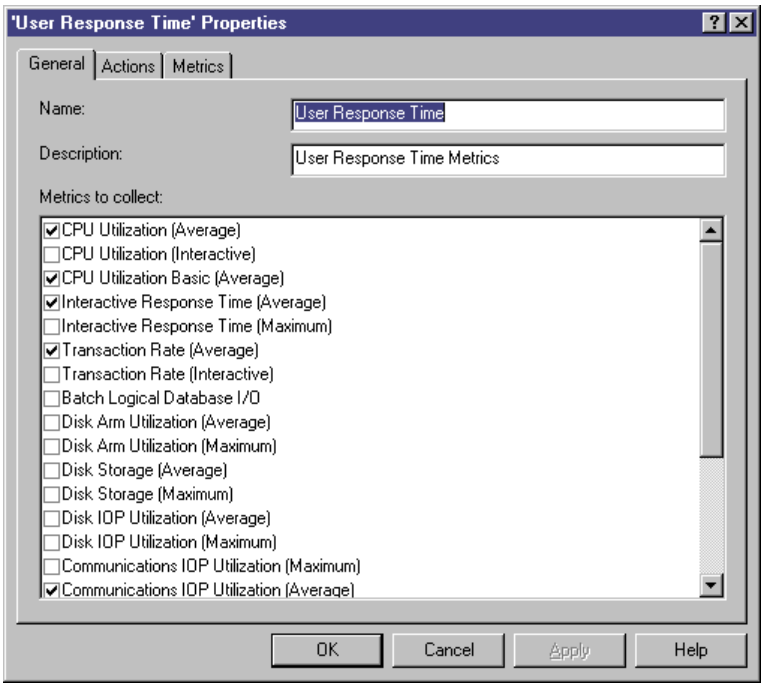
Management Central Distributed Services

Overview



Services are long running asynchronous operations that perform a continuous action. One type of service is a monitor that can watch over a device or resource on the system. Others could include trace-like or collector-type functions. Unlike tasks, services never really end but are instead turned on and off by the operator or administrator.

One example of a service is the Monitor application that was delivered with Management Central in OS/400 V4R3. These system monitors allowed the operator to watch over different attributes of the system. These attributes or metrics included CPU Utilization, Interactive Response Time, and Transaction Rates as just a few examples. After the operator selected which metrics to watch, she could then select the interval to poll the resource, how to graph the information, set thresholds when a value got too high or too low, and also perform some automation when a threshold was reached.



When developing your own service, you will want to think about a number of different things. First, you will need to determine what continuous operation to perform. If it fits the monitor type of service, then what resources do you want to watch or monitor? What happens when the user turns the service on, or off? What information do you need to send back to the Central System and GUI to keep the user informed of the progress?

No matter what the answers are to these questions for your particular application, the Management Central Java Framework provides you with the infrastructure to help you in your implementation.

Interfaces and Flows

Application Designer

As the application designer, you will want to address a lot of the questions that were posed on the previous page:

- What continuous action is being performed?
- What happens when the user selects to turn the service on? off?
- What resources to watch or monitor?
- Multiple metrics or should each be different monitor types?
- How often should the information be polled or gathered?
- Does the resource information need to be graphed?
- Does the user want to set thresholds if a value gets too high, too low, or maybe when it changes states?
- Allow the user to perform some action or automation when a threshold has been reached?

Use the following checklist to assist you in your design.

Design checklist:

- ✓ What data do you need to send to the remote endpoint systems?
  Example: Name, Description, Owner, Sharing, and Monitor details

  Guidelines:
  - Think of data and information you need in order to perform your service. This information is considered the "Application Attributes" of the service.
  - Don't include System Groups, Endpoint Systems, Status, or Result information in your service. This is already built into the service objects. Also, omit information available via jMC base implementation, such as Name and Owner.

- ✓ Once you have the data at the endpoint system, what action to do want to perform?

  Guidelines:
  - In your class that implements the McSwitchable you will need to decide what function to perform in the on method and the off method. This is where your custom logic goes.

- ✓ Where in the Operations Navigator hierarchy do you want to see the list of your application's Services? Also, if your application creates multiple type of services do you want to filter out some or do you want to show all of them?

  Guidelines:

- See your UI Designer for help.

✓ What context menu options do you want on your Service/Monitor Container?

Guidelines:
- If integrating into Operations Navigator, you will want the same behavior as other Monitor containers. This includes context menu options for *Explore*, *Open*, and *Create Shortcut*.

✓ What context menu options do you want on each Service/Monitor?

Guidelines:
- See your UI Designer for help.

Application Developer

The application developer takes the design specification and will need to create at least four classes for a distributed service. First, you need to create a class to contain all your application data and attributes. This class will be termed the "Application Attributes" of your activity. The jMC will automatically store these attributes persistently on the Central System. In addition, you will need to determine what functions or actions your service needs to perform on the endpoint systems and what data or information is needed to perform that action.

The **McDistributedServiceDescriptor** is provided to give you a way to describe how you want your distributed service to behave. By extending the **McDistributedServiceDescriptor** class, you create a new kind of distributed service which inherits Name, Description, Owner, and Sharing data members and methods. You only need to supply information specific to your application.

The **McEndpointServiceDescriptor**, like the Distributed Service Descriptor, is used to describe and contain the data for your service. In this case, the Endpoint Service Descriptor contains the data that is used on the endpoint system and is used when the operator requests to turn on and off your service. Again, the framework provides all the basic information (e.g. *Name*, *Description*, etc.). All you have to add are any private data members you need with getter and setter methods.

The fourth class used is the piece that performs all the service or monitor function on the endpoint system. Creating a new class which implements the **McActionIfc** and **McSwitchable** interface will allow your distributed service to be notified when to turn on and when to turn off.

Now, lets walk through the process of creating a new distributed service application. There are three steps or scenarios that we will look at:

1. Create a new class that contains your application data. This data will be referred to as "Application Attributes".

2. Create a new class that extends the McDistributedServiceDescriptor class
3. Create a new class that extends the McEndpointServiceDescriptor class
4. Create a new class that implements the McSwitchable interface

Scenario 1: Create a new class that contains all your application data

In this scenario you create a new Java class that represents the data or attributes for your application's task. This class needs to implement the java.io.Serializable interface so it can be sent to the central system and stored in a persistent manner.

```
package com.ibm.as400.opnav.MyServiceSample;

import java.io.Serializable;

public final class MyMonitorData implements Serializable
{
    private String m_program;

    public MyMonitorData(String name) {
        m_program = name;
    }

    public String getProgram() {
        return m_program;
    }
```

Scenario 2: Create a new class that extends the McDistributedServiceDescriptor class

The **McDistributedServiceDescriptor** class is used to store your application service data on the Central System. By extending the **McDistributedServiceDescriptor** class, you inherit data members and methods for Name, Description, Owner, etc. In this example, all you need to provide is your private data members with getter and setter methods that defines your application service. Then, implement the createEndpointData method defined in the **McDistributedActivityDescriptorIfc** interface, a parent of **McDistributedServiceDescriptor**. This method is where you create an instance of your Endpoint Service Descriptor and give it to the Management Central Java Framework. The *coordinator* field used to add event listeners is defined by McDistributedTaskDescriptor's superclass, McActivityDescriptor. It refers to an event coordinator that processes events for this descriptor.

Hints and Tips: Remember you only need to include the data that is unique to your service. Information like System Group, Status, and Results are already provided for you in the Distributed Service Descriptor and associated classes.

```
package com.ibm.as400.opnav.MyServiceSample;

public class MyDistRemoteServiceDescriptor extends McDistributedServiceDescriptor
{
  // Constructors
  public MyDistRemoteServiceDescriptor(
                                String theName,
                                String theDescription,
                                int theSharing,
                                McSystemGroup theSystemGroup,
                                MyMonitorData applicationData) throws McException
  {
    super(theName, theDescription, theSharing, theSystemGroup, applicationData);
    coordinator.addEventListener(new McStatusAspect(), new
McDefaultStatusEventHandler());
    coordinator.addEventListener(new McResultAspect(), new
McDefaultResultEventHandler());
  }

  // Implement methods from the McDistributedActivityDataIfc interface
  public McEndpointActivityDescriptorIfc createEndpointData() throws McException
  {
    MyEndpRemoteServiceDescriptor eptData = new MyEndpRemoteServiceDescriptor(
                          getName(),                  // Name
                          getDescription(),           // Description
                          getSharing(),        // Sharing
                          getMonitorData()            // Application attributes
                          );

    eptData.setPrivate(true);
    return eptData;
  }
```

NOTE: You may have noticed that in the code example above you're using a **MyEndpRemoteServiceDescriptor** class which you will create next.

Scenario 3: Create a new class that extends the McEndpointServiceDescriptor class

Like the Distributed Service Descriptor, this class also contains the data for your application service. In this case, the data is used on the endpoint system and needs to contain any information that your on and off action methods might need. Like the previous example, you start out by defining any private data members you need with getter and setter methods. Next, you need to implement the getActivityActionClass method defined in the **McActivityDescriptorIfc** interface. Here is where you tell the Management Central Java Framework the class which implements the **McSwitchable** interface which contains the on and off methods.

```
package com.ibm.as400.opnav.MyServiceSample;

public class MyEndpRemoteServiceDescriptor extends McEndpointServiceDescriptor
{
  // Constructors
  public MyEndpRemoteServiceDescriptor(
                                String theName,
                                String theDescription,
                                int theSharing,
                                MyMonitorData applicationData) throws McException
  {
    super(theName, theDescription, theSharing, applicationData);
    coordinator.addEventListener(new McStatusAspect(), new
McDefaultStatusEventHandler());
    coordinator.addEventListener(new McResultAspect(), new
McDefaultResultEventHandler());
  }
```

Scenario 4: Create a new class that implements the McSwitchable interface

This scenario involves implementing methods for the **McSwitchable** interface and the **McActionIfc** interface. The **McActionIfc** interface includes the setObserver, setDescriptor, setThread, descriptorChange, and descriptorRemove methods. The **McSwitchable** interface includes the on and off methods.

```
package com.ibm.as400.opnav.MyServiceSample;

public class MyEndpRemoteServiceAction implements McSwitchable, McActionIfc
{
  private McRemoteListener               observer = null;
  private McEndpointServiceDescriptorIfc data     = null;
  private McMethodThreadIfc              thread   = null;

  public MyEndpRemoteServiceAction() throws McException
  {} // Must implement a no-argument constructor for dynamic instantiation

  private void doUpdate(McEvent event, String method) throws McException
  {
    try { observer.update(event); }
    catch(RemoteException e) {
      if (e instanceof McRemoteException)
        ; // Handle MC generated exception. Will most likely want
          // to construct a new McException(e) and throw to caller.
    }
  }

  public void setObserver(McRemoteListener observer) throws McException
  {  this.observer=observer;  }

  public void setDescriptor(McActivityDescriptorIfc data) throws McException
  {
    if (data instanceof McEndpointServiceDescriptorIfc)
      this.data = (McEndpointServiceDescriptorIfc)data;
    else
      ; // handle error
  }

  public void setThread(McMethodThreadIfc thd)
  {
    if( (this.thread == null) && (thd != null) )
      this.thread = thd;
  }

  public void descriptorChange() throws McException
  {}

  public void descriptorRemove() throws McException
  {}
```

GUI Developer

The GUI developer connects the user interface to the Distributed Service Descriptor class defined by the application developer. One possible solution could be to add a new Monitor container in Operations Navigator under the Monitors branch of the Management Central tree, adding a context menu option on this new container to create new monitors, and adding context menu choices on each monitor to view it's properties and to perform actions (i.e. New Based On, View Status, etc.).

To perform this development you will need to know how to create an Operations Navigator Plug-in using ListManager and ActionManager interfaces, how to work with jMC classes **McDistributedServiceDescriptor**, **McDistributedServiceView**, and **McDistributedServiceListView**, and how to use the GUI helpers provided by the Management Central Java Framework to display properties, select systems and groups, and to delete your application services.

Classes and Interfaces:

- com.ibm.mc.client.activity.service.McDistributedServiceDescriptor
- com.ibm.mc.client.activity.service.McDistributedServiceView
- com.ibm.mc.client.activity.service.McDistributedServiceListView

Scenario 1: Creating Instances of your new Application Service

In scenario 1, you take your new set of service classes and explore how to create instances of them. This would be the underlying function when a New Service or New Monitor menu option is selected. Basically there are three steps in creating a new service.

1. Create an instance of your application's distributed service descriptor. The properties of the service would be retrieved from possibly GUI dialogs or wizards prompting the user for the information. Note that the getSystemGroup and getMonitorData methods used in the Descriptor's constructor must be supplied by the user to retrieve the list of endpoint systems on which to execute the command, and get the application attributes for the service.
2. Create an instance of a Distributed Service View object specifying your application service descriptor.
3. Tell the Distributed Service View to add the instance of your service that you created in the first step.

```
public void newService() throws McException
{
  // Step 1: Create an instance of the Distributed Service Descriptor
  McDistributedServiceDescriptorIfc data = new MyDistRemoteServiceDescriptor(
                                "MyAppTask",           // Name
                                "Description",  // Description
                                McManageable.NONE,     // Sharing
                                getSystemGroup(),      // User method to get
System Group
                                getMonitorData());     // User method to construct
data

  // Step 2: Create an instance of a Distributed Service View object
```

Scenario 2: Turn On/Off your Application Service Instance; receive Status and Results

Once you've created your service instance, you will want to have the user start it (turn it on) and be able to stop it (turn if off). This is accomplished by calling the on and off methods on the distributed service view.

```
// Start your Service
view.on();

// ... (further application processing - display a status dialogue, etc)

// Stop your Service
```

Scenario 3: Get a list of your Application Service Instances

If you need to retrieve a list of the services that you created in Scenario 1, this next step will show you how. There are only a few steps needed here.

1. Set up the selection criteria to only get services of class **MyDistRemoteServiceDescriptor**
2. Create an instance of **McDistributedServiceListView** to manage the list of services
3. Ask the Distributed Service List View to return to you a list of your application's Distributed Services

```
public Vector listServices() throws McException
{
  McManageableSelectionCriteria selCriteria     = null;
  McDistributedServiceListView  distServiceView = null;
  Vector                        myServiceList   = null;

  // Step 1: Define selection criteria to get a list of your application
  selCriteria = new McManageableSelectionCriteria(

"com.ibm.as400.opnav.MyServiceSample.MyDistRemoteServiceDescriptor",
                      McManageable.ALL, // Category
                      null,             // List of owners (only used if next parameter
is true
                      false,            // Include shared tasks
                      0);               // Check the last changed date of the task

  // Step 2: Create a new Task List View to manage your application tasks
```

Scenario 4: Get asynchronous status and results of a Service

Once you've created and started your service, it will run asynchronously with other activities. If you want to monitor the status of the request, you will want to attach a class to handle when status and results are returned from the endpoint system to you. In Scenario 1, where you turned your service on and off, you would probably also want to attach a class to handle status and results before turning the service on, and detach the class after turning the service off.

This class will implement the **McStatusDetailListener** and **McResultDetailListener** interfaces. In the example shown below, the same class implements both these interfaces so we pass in this as the object to handle status and result updates.

```
    // Previously you would have retrieved a list of service views and
selected
    // the one view that you want to monitor status and results

    // Attach to be notified when a Status or Result object is received
    view.attachStatusDetailListener(this);
    view.attachResultDetailListener(this);

    // Display a status window or dialog, or turn service on



   // Implementing the McStatusDetailListener interface
   public void statusUpdate(McStatusEvent event) throws McException
   {
       try
       {
           McStatusIfc status = event.getStatus();
           System.out.println("McTestService: statusUpdate "
                           + "Status ID: " + status.getId()
                           + " Level: " + status.getLevel()
                           + " Value: " + status.getIntValue());
           if(status.getLevel() == McStatusIfc.DistributedAct &&
              (status.getIntValue() == McStatusIfc.On ||
               status.getIntValue() == McStatusIfc.Off ||
               status.getIntValue() == McStatusIfc.Failed))
           {
               // handle status update
           }
       }


  // Implementing the McResultDetailListener interface
  public void resultUpdate(McResultEvent event) throws McException
  {
      try
      {
          // Get the result object out of the event information
          McResultIfc result = event.getResult();

          // Since the result object is a hierarchy of results for each Endpoint

          // System specified in the task, you need to get the results for the
          // specific Endpoint System to see its details.
          McResultIfc childResult = (McResultIfc)(result.findChild("system1"));


          System.out.println("McTestService: resultUpdate, system1 "
                          + "Result ID: " + result.getId()
                          + " Level: " + result.getLevel()
```

Hints and Tips: The getResultData used in the previous example generically returns an Object of type Serializable. You can easily cast the result data to whatever kind of an object your specific application expects.

The MC Java Framework also allows you to easily implement your own Status and Result objects to use for your application. All you need to do is create an object that extends either **McServiceStatus** or **McResult**. If you don't implement your own status and result objects, these default types will be used, but if you do decide you need extensions of the functionality available, you only need to extend what's applicable. For instance, if you're implementing a service, and only need your own status type, you only need to extend McServiceStatus, and can use the default McResult implementation.

Once you've created your objects, you'll need to override the createStatus and/or createResult methods in your descriptor objects (both endpoint and distributed) to return an object of your new status or result type. The jMC will automatically invoke these create methods when it needs a status or result object.

Scenario 5: Get asynchronous status updates for Lists of Services

By implementing the **McManageableListener** interface, you can be notified when a new service has been created, changed, updated, or deleted. This is most useful when maintaining a list of activities, and you wish to be notified whenever they are added, removed, updated, or changed. This list of services is specified using selection criteria so that you are not notified when just any activity is created, but only those services that meet your selection subset. This code is identical to the code used to retrieve a list of activities based on a selection subset, but we add a fourth step here to attach the current class as the ManageableListener. This interface, along with the implemented update, change, and remove methods, allows the Management Central Java Framework to send you a notification when an activity has been updated.

```
public class MyServiceList implements McManageableListener
{
  public void getList() {
    McManageableSelectionCriteria selCriteria    = null;
    McDistributedServiceListView  viewList       = null;
    Vector                        retrievedTasks = null;

    // Step 1: Define selection criteria to get a list of Distributed Services
    selCriteria = new McManageableSelectionCriteria(

"com.ibm.as400.opnav.MyServiceSample.MyDistRemoteServiceDescriptor",
                  "McManageable.ALL",    // category
                  null,           // Owner List (only used when next parm is true)
                 false,           // use sharing
                  0);                    // last changed date

    // Step 2: Create a new Service List View to manage your services
    viewList = new McDistributedServiceListView(selCriteria);

    // Step 3: Ask the Distributed Service Manager for a list of Distributed
Services
   retrievedTasks = viewList.getManageableViews();

    // Step 4: Attach this class, which implements the McManageableListener
interface,
    //         to handle any notifications.
    viewList.attachManageableListener(this);
  }

  // The following methods are required as an implementation of
McManageableListener

  public void manageableAdded(McManageableEvent event) throws McException {
    // Insert code to handle when a new task has been created
  }

  public void manageableChanged(McManageableEvent event) throws McException {
    // Insert code to handle when a definition has changed.  A "change" is a
    // user-directed property change, such as changing the definition's
```

Scenario 6: Delete an Application Service Instance

If you need to delete a service, or a list of services, the Distributed Service View and List View classes provides the methods for you. Deleting a service removes the service from the Management Central databases and is no longer a manageable service. When working with the Distributed Service View object, you can simply call removeManageable method on the instance of the object itself; for a List of Views, you can simply call removeManageableList on the ListView instance.

```
  // Tell the Distributed Service View to remove the instance of your service
  distServiceView.removeManageable();

  // Or, for a Distributed Service List View
```

Scenario 7: Change an Application Service Instance

To change an existing service, the Distributed Service View provides the method for you to use. Prior to calling change, you would have a reference to a service that you previously created or retrieved from a list of services. With the reference to the service, you may have the end user modify it by displaying property pages and using the appropriate set methods to update the service instance. Now when you have the service instance up to date, you can tell the Distributed Service View to store the changes.

```
// Step 1: Change the Distributed Service Descriptor locally
distServiceDesc.setDescripton( "New Description" );

// Step 2: Tell the Distributed Service View to change the Descriptor on the
server
```

NOTE: In addition to all the services features documented above, there are additional advanced features that apply to services you should be aware of. Since they are features that apply to activities in general (and some apply to definitions), they have been documented in their own chapter, affectionately entitled *Advanced Features*, beginning on page 54.

Terms, Classes, and Interfaces

Thing	Type	What to do with it	Purpose
McDistributedServiceDescriptor	Class	Extend	An application will create one (or more) subtype(s) of this class to define how their multi-system service will run, what data will be associated with it, and how it will handle events on the central system. Your subtype(s) of this class will describe most of the parts of your application.
McDistributedServiceDescriptorIfc	Interface	Use	The interface that all distributed service descriptors will support.
McDistributedService	Class	Use	Objects belonging to this class represent the actual, running service. You may operate on distributed service objects using the McSwitchable interface. All distributed services have a distributed activity descriptor associated with them. The service refers to its descriptor to get information about how it should run. You are not responsible for instantiating this object. A distributed service object can be obtained from a distributed service descriptor object that is being managed by the MC framework.

McSwitchable	Interface	Implement	This interface defines the methods that you may call on a distributed service. Use it when you want to start (turn on) a service or to stop (turn off) a running service.
McEndpointServiceDescriptor	Class	Extend	An application will create one (or more) subtype(s) of this class to define how their service will run, what data will be associated with it, and how it will handle events on the endpoint.
McEndpointServiceDescriptorIfc	Interface	Use	The interface that all endpoint service descriptors will support.
McActionIfc	Interface	Implement	You must provide a class that implements this interface to perform the endpoint on and off action that is part of your service. Your object will perform the "real work" of making the service run on the endpoint, and the MC framework will call the object at the appropriate time.
McDistributedServiceView	Class	Use	This wrapper class extends the capability of the McDistributedService class to provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI databeans can be notified directly when a status has changed. Service Actions: - on - off Distributed Service Manager: - addManageable - getManageable - changeManageable - updateManageable - removeManageable Event Listeners: - attachStatusDetailListener - detachStatusDetailListener - attachResultDetailListener - detachResultDetailListener
McDistributedServiceListView	Class	Use	This wrapper class extends the capability of the McDistributedServiceManager class to provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a new service has been added, changed, or removed from a list of services. Distributed Service Manager: - getManageableList Event Listeners:

			- attachManageableListener

Hints and Tips

This space has been included so that you can document your own special hints and tips:

Advanced Features

The following scenarios are provided to supplement the information in each of the preceding sections. They are considered advanced features only because they aren't *necessary* to use the definitions or activities provided by the Management Central Java Framework. However, they are quite useful, and are provided here as a reference.

All examples are shown using the Distributed Task as the base, but keep in mind that all are valid for any activity, whether tasks or services. Also, scenarios 2-4 also apply to definitions.

Scenario 1: Receiving connection updates

Similar to receiving status or results updates by *attaching* a class that implements a certain interface, clients can also attach for connection updates by implementing the **McConnectionListener** interface and implementing the connectionUpdate method. This method will return to any attached listeners a **McConnectionEvent** update whenever the status of the connection changes. The event can then be polled to determine the nature of the update. The following example will help get you started.

First, you need to attach a class as a connection listener. You do this by first creating a **McConnectionAspect** object to tell the MC Java Framework which systems you wish to receive updates from. You should specify the special String constant McManageable.ALL so even if the central system is changed by the user, you'll still receive connection updates from the new central system. Then, invoke attachConnectionLister through its static interface on class **McClientConnectionManager**, supplying the aspect and the listener instance. Since this class will implement McConnectionListener, specify *this* as the listener instance.

```
// Attach class as the connection listener
McConnectionAspect aspect = new McConnectionAspect(McManageable.ALL);
McClientConnectionManager.attachConnectionListener(this, aspect);

// ... custom logic for processing your application goes here

// When you no longer want connection updates, detach this
// instance as a connection listener
```

Then, implement the connectionUpdate method. Notice that you'll need to supply the getCentralSystemName method used to get the current central system to compare to the connection event. Use the **McClientConnectionManager** class to help you.

```
public void connectionUpdate (McConnectionEvent event) throws McException
{
    // For instance, could check if event represents a broken connection
    if (event.contains (getCentralSystemName()) &&
       (type == McConnectionEvent.MCJavaConnectionFree  ||
        type == McConnectionEvent.MCJavaConnectionDrop  ||
        type == McConnectionEvent.MCCPPConnectionDrop))
    {
        // custom logic to close windows and notify users
        // that connection to central site has been lost
```

Scenario 2: Private Descriptors

By default, all Descriptors are public. Mainly, this means that once the manageable is managed by the jMC, it will be stored persistently in a database on the server. Any user that has appropriate authority can retrieve that descriptor off the server by setting up a SelectionCriteria object that meets some criteria of the descriptor, creating a list view with the selection criteria, and calling `getManageableViews` on the listView.

Private descriptors, on the other hand, are never stored persistently, and may not be retrieved off the server once they're created, even by the owner. Even if a user specifies a selection criteria that exactly matches the private descriptor, it will not be returned in their list. This is why you let the jMC default to use the public Distributed Task Manager when creating your Distributed Task Descriptor, but you need to specify setPrivate(true) when creating the Endpoint Task Descriptor within your Distributed Task Descriptor implementation. Since the endpoint descriptor is only applicable to that single execution, you wouldn't want to grant others the ability to retrieve it off the server.

In the following example, the descriptor is created as normal, but Step 1a is added to make the descriptor private. Without this step, anyone with the appropriate authority could retrieve the task from the server, and possible update it.

```
McDistributedTaskDescriptorIfc distCommandDesc = null;
McDistributedTaskView          distTaskView    = null;

// Step 1: Create an instance of a Distributed Task Descriptor
distCommandDesc = new MyDistributedCommandCallDescriptor(
                          "Task Name",            // Name
                          "Task Description",     // Description
                          McManageable.NONE,      // Sharing
                          getSystemGroup(),       // System Group
                          getCommand() );  // Application attributes

// Step 1a: Make descriptor private
distCommandDesc.setPrivate(true);

// Step 2: Create an instance of a Distributed Task View object
//         specifying the Distributed Task Descriptor.
distTaskView = new McDistributedTaskView(distCommandDesc);
```

Scenario 3: Public Descriptor Sharing

In this scenario, we added Step 1a to change the Descriptor's sharing value. Sharing lets the owner specify whether other users can view or change the contents of the descriptor. Only public descriptors can be shared.

```
    McDistributedTaskDescriptorIfc distCommandDesc = null;
    McDistributedTaskView          distTaskView    = null;

    // Step 1: Create an instance of a Distributed Task Descriptor
    distCommandDesc = new MyDistributedCommandCallDescriptor(
                              "Task Name",            // Name
                              "Task Description",     // Description
                              McManageable.NONE,      // Sharing
                              getSystemGroup(),       // System Group
                              getCommand() );  // Application attributes

    // Step 1a: Enable full sharing
    //          Alternatively, this could have been specified on the constructor
above
    distCommandDesc.setSharing(McManageable.FULL);

    // Step 2: Create an instance of a Distributed Task View object
```

By default, there is no sharing of descriptors, but by setting the sharing value of this **McDistCommandDescriptor** to *McManageable.FULL*, all users will be able to retrieve the descriptor using the listView's getManageableViews method, and will also be able to make and store changes to the server using the changeManageable method on the View, or even delete it via a call to removeManageable on the View.

Valid sharing values and their meanings are:

McManageable.NONE	This is the default sharing value. No users will be able to view the descriptor.
McManageable.READ	All users will be able to view but not change, the descriptor.
McManageable.USE	All users will be able to view and change, but not delete the descriptor. **Only activities (tasks and services) may utilize this sharing value.**
McManageable.FULL	All users will be able to view, change or delete the descriptor. **Only definitions may utilize this sharing value.**

Scenario 4: Auto Increment

In this scenario, we added Step 1a to set auto increment to true. Auto increment allows the user to create multiple instances of a task without worrying about a name conflict. The first time this is run, it

will create a task with a name specified by the parameter theName (ex. “MyTask”). By adding step 1a, the second time this is run the Management Central Java Framework will automatically identify that the name”MyTask” exists and increment the name to “MyTask(2)”. The third time it will be “MyTask(3)” and so on. Without adding step 1a, the second time the newTask method is run an exception would be signaled when performing the addManageable method call.

```
McDistributedTaskDescriptorIfc distCommandDesc = null;
McDistributedTaskView          distTaskView    = null;

// Step 1: Create an instance of a Distributed Task Descriptor
distCommandDesc = new MyDistributedCommandCallDescriptor(
                          "Task Name",            // Name
                          "Task Description",     // Description
                          McManageable.NONE,      // Sharing
                          getSystemGroup(),       // System Group
                          getCommand() );  // Application attributes

// Step 1a: Enable auto-increment
distCommandDesc.setAutoIncrement(true);

// Step 2: Create an instance of a Distributed Task View object
//         specifying the Distributed Task Descriptor.
```

Scenario 5: Categories

When the same task class needs to be used for multiple purposes, categories can be used to distinguish between them. In this command task example, maybe you have the need for both Backup-type tasks and Restore-type tasks. Since both types of tasks use the same descriptor class the jMC will not be able to distinguish between both types of tasks given only the class name. The use of Categories will help distinguish between your different types of tasks. When you create an instance of your task, you can specify a Category to use like “Backup-type” instead of creating a separate task class for each. This is done in Step 1a in the following example.

```
McDistributedTaskDescriptorIfc distCommandDesc = null;
McDistributedTaskView          distTaskView    = null;

// Step 1: Create an instance of a Distributed Task Descriptor
distCommandDesc = new MyDistributedCommandCallDescriptor(
                          "Task Name",            // Name
                          "Task Description",     // Description
                          McManageable.NONE,      // Sharing
                          getSystemGroup(),       // System Group
                          getCommand() );  // Application attributes

// Step 1a: Set category
distCommandDesc.setCategory("Backup-type");

// Step 2: Create an instance of a Distributed Task View object
//         specifying the Distributed Task Descriptor.
distTaskView = new McDistributedTaskView(distCommandDesc);
```

If you specify a category when you create an instance of your application's task, you can specify that same category in your selection criteria on the Manageable Selection Criteria interface to only receive tasks that match. This allows you to retrieve a list of only Backup-type tasks or Restore-type tasks even though they both share the same com.ibm.as400.opnav.MyTaskSample.McDistributedCommandCallDescriptor class.

```
public Vector listTasks()
{
  McManageableSelectionCriteria selCriteria  = null;
  McDistributedTaskListView     distTaskView = null
  Vector                        myTaskList   = null;

  // Step 1: Define selection criteria to get a list of your application
  selCriteria = new McManageableSelectionCriteria(

"com.ibm.as400.opnav.MyTaskSample.MyDistributedCommandCallDescriptor",
                       "Backup-type",    // Category
                       null,             // List of owners (only used if next parameter
is true
                       false,            // Include shared tasks
                       0);               // Check the last changed date of the task

  // Step 2: Create a new Task List View to manage your application tasks
```

If you do *not* specify a category when constructing your descriptor, the Management Central Java Framework will create a default category for you, and use that category when storing your descriptor on the central system. You can explicitly use the default category by specifying the **McManageable.DEFAULT** special category value.

If your descriptor is created and stored with the default category, it can only be retrieved using the default category (remember, if you omit a category specification, the default category will be used), or by specifying the **McManageable.ALL** special category value. This special value tells the jMC to disregard the category value when retrieving descriptors based on your selection criteria. In the above example, if your `listTasks` method specifies **McManageable.ALL** as its category value, both Backup-type and Restore-type tasks will be retrieved.

Scenario 6: Logging activity events

Upon execution of your application's Action on the server, you may find that you want to log events persistently, so that a user interface can retrieve these events at some later date and be able to see what has historically occurred with their application. This would be the case with long running services, where you would like to have the ability to start a service and then detach your user interface, only to reattach later and view the events of the service. The MC Java Framework's Event Log classes can help you with this.

To implement your own events and have them stored persistently on the central site, you'll need to extend the **McLoggableEventDetail** class. This class helps you define the data specific to your application, as well as a format for the data, so your formatted data can be retrieved at a later date. As an implementation of McLoggableEventDetail, you need to write the getEventSpecificBytes method to return the actual data for the event, the getFormatBytes method to return the format of the actual event data, and the getFunctionCodeBytes method to provide a function code to help the user interface to retrieve the events from the central system.

In addition to the methods defined in the example, you'll need public constructors and/or getter and setter methods to set and retrieve the data to and from your event class.

```
public class MyMonitorEventDetail extends McLoggableEventDetail
{
 public byte[] getEventSpecificBytes() throws McException
  {
    byte[] eventData = null;

    eventData = BinaryConverter.intToByteArray(messageInfo.getMessageSeverity());

    eventData = McUtilities.merge(eventData,
                  McUtilities.stringToByteArray(getAS400().getCcsid(),
                  McUtilities.correctSize(m_messageInfo.getMessageType(), 2)));

    // ... process all information you want associated with the eventData
    return eventData;
  }

  public byte[] getFormatBytes() throws McException
  {
    return McUtilities.stringToByteArray(getAS400().getCcsid(), format);
  }
```

Hints and Tips: The *format* and *function code* data members used in the preceding example cannot be specified by the application developer creating their own event type. They, as well as the event-specific bytes, are defined by a Management Central API that must be adhered to. If you are using this example to create your own event type, contact a member of the Management Central Java Framework to help update the API to accommodate your new kind of event.

Then, in your code that's monitoring for certain conditions in which you wish to generate an event, you need to create a **McLogEvent**, containing a **McLoggableEvent**, containing an instance of your implementation of **McLoggableEventDetail**, **MyMonitorEventDetail**. That's a lot of objects to handle, but it provides the jMC with a generic interface to update your central system with events specific to your application.

```
public class MyLogActionEvent implements Serializable
{
  McRemoteListener observer = null;
  String guiClassName = "com.ibm.as400.opnav.MyEventSample.MyGuiClassEventHandler";

  public MyLogActionEvent(McRemoteListener theObserver)
  {
    this.observer = theObserver;
  }

  public void logEvent(MyEndpRemoteServiceDescriptor endpDescriptor)
                                          throws McException, RemoteException
 {
    McLoggableEventDetail eventDetail = new MyMonitorEventDetail();
    int eventType = 0;

    // ... set data in your event detail instance

    McLoggableEvent event = new McLoggableEvent(
                        endpDescriptor, // endpoint service descriptor for this
endpoint
                        eventType    // int, indicating the kind of event
```

Hints and Tips: The *observer* data member passed into the class constructor can be retrieved from your Action class (the one that implements **McActionIfc**). The class' setObserver method will be called by the MC Java Framework to give you an instance of McRemoteListener. This observer is simply a managed reference to your endpoint activity.

Finally, you'll need to update both your endpoint and distributed descriptor constructors to listen for loggable events. The endpoint descriptor will be notified when the Action class calls update on the observer, and it will forward the update on to your distributed descriptor on the central system. There, the MC Java Framework will persistently store the event.

```
// The Aspect filters the logged events based on class name
McAspect aspect = new McAspect("com.ibm.mc.client.notification.McLogEvent");

// The coordinator, defined in your descriptor's superclass, coordinates events

// In your endpoint descriptor, tell the event handler to forward the event
// to the central system (true), and NOT to log the event on this system (false)
```

Alternatively, on the central system, you'll want to toggle the boolean values for the McLoggableEventHandler, because you *don't* want to update any other observer, but you *do* want to log the event on the current system.

```
coordinator.addEventListener(aspect, new McLoggableEventHandler(false, true));
```

Now that the event is logged on the central system, the user interface designer will need to provide the implementation to retrieve the events from the central system. Since loggable events are definitions, you'll use the **McDefinitionListView** class to manage your list of events, and the **McEventSelectionCriteria** class to specify which events you want. There are only a few steps needed here:

```
  String[] classTypes =
{"com.ibm.as400.opnav.MyEventSample.MyGuiClassEventHandler"};

  // Step 1: Specify which events to retrieve
  McEventSelectionCriteria criteria = new McEventSelectionCriteria(
                            null,         // category list
                                   null,         // owner list (valid if next parm
is true)
                            false,               // include sharing
                            0,            // last changed
                            null,         // originator list
                            null,         // system list
                            classTypes,   // application type list
                            null);               // eventTypeList

    // Step 2: Create a new Definition List View object to manage your events
```

Many of the parameters of the **McEventSelectionCriteria** constructor are similar to the **McManageableSelectionCriteria** constructor discussed in earlier sections. The most important for this example is the *application type list* parameter. This specifies fully qualified class names for classes that know how to decode the event detail that is part of the McLoggableEvent. This is the same class name as was used in the McLoggableEvent constructor when the event was logged. In the example shown above, a McLoggableEvent was created with the class name `"com.ibm.as400.opnav.MyEventSample.MyGuiClassEventHandler"`, so if, in the example above, this String was specified in the McEventSelectionCriteria, the event will be returned to you from the call to getManageableViews . With the event, you can dynamically instantiate this class, send the event over for decoding of the loggable event detail, and show the loggable event in a gui panel.

Call To Action

You may find that you need to periodically invoke methods directly on your endpoint activities other than those that are managed by the jMC. For instance, if you're implementing a service, the jMC will handle the execution of the on and off methods at the appropriate times, based on actions of the user, but what if you need to allow the user to reset some triggered metric for that service? The Call To Action methods on **McActivityView** are your answer.

Two implementations exist, and they vary slightly. The callToActionAsync method is used to push the method call onto a queue. The MC Java Framework will invoke the method in due time as it's

de-queued from the server, and status and/or results will be propagated back to the caller via registered status, results or callToAction listeners. The callToActionSync method, on the other hand, will directly invoke the method, allowing the requester to wait on that execution thread until the method is complete. Any status or results will be returned directly to the caller through return values or exception objects. This method invocation will follow the same execution model as the rest of the framework, that is, it will be run under the user profile of the owner of the activity.

In either case, you can only call methods defined in your Action class; that is, the class that implements **McActionIfc** (and either the McSwitchable interface for services, or the McExecutable interface for tasks).

In this example, we'll implement a method called manualTriggerReset to be invoked from the user interface using a call to action method. On the server, simply define the method to be invoked in your McActionIfc class. This is the same class that defines the actions of your application.

```
package com.ibm.as400.opnav.MyServiceSample;

public class MyEndpRemoteService implements McSwitchable, McActionIfc
{
  private McRemoteListener               observer = null;
  private McEndpointServiceDescriptorIfc data     = null;
  private McMethodThreadIfc              thread   = null;

  public MyEndpRemoteService() throws McException
  {} // Must implement a no-argument constructor for dynamic instantiation

  // define interface methods:
  //     on, off from McSwitchable
  //     setDescriptor, setObserver, setThread,
  //     descriptorChange, descriptorRemove from McActionIfc

  // Step 1: define your call to action method
  public void manualTriggerReset(Vector jobIDs,boolean runCommand,String
```

On the client, use these steps as guidelines for invoking an activity action method via a call to action method:

1. Set up a view containing your implemented distributed service descriptor, and use the view to addManageable on the descriptor.
2. Attach the current class as the class that's listening for call to action updates. This class must implement the **McCallToActionListener** interface.
3. Set up the list of parameters and parameter types required for the method you wish to call. For the sake of simplicity, assume these parameters were created earlier in the code.
4. Invoke the view's call to action method, specifying the name of the action method you need to call, the parameter list, the parameter type list, and the system group on which to run. Note that the getSystems method used must be supplied by the user to retrieve the list of endpoint systems on which to execute the method.
5. When you're through processing callToAction updates, detach the current instance as a callToAction listener.

```
  // Step 1: Create an instance of the Distributed Service Descriptor
  McDistributedServiceDescriptorIfc data = new MyDistRemoteServiceDescriptor(
                                "MyAppTask",           // Name
                                "Description",  // Description
                                McManageable.NONE,     // Sharing
                                getSystemGroup(),      // User method to get
System Group
                                getMonitorData());     // User method to
construct data

  // Create an instance of a Distributed Service View object
  // specifying your application service descriptor.
  McDistributedServiceView view = new McDistributedServiceView(data);

  // Tell the Distributed Service View to add the instance of your service
  view.addManageable();

  // Step 2: Attach the current class as the listener
  view.attachCallToActionListener(this);

  // Step 3: Set up parm list and parm types for the call to action.
  Object[] parms = {getJobIds(), new Boolean(true), getOwner()};

  String[] parmTypes = {"java.util.Vector",
```

Finally, as part of the McCallToActionListener interface, you must supply a resultUpdate method to handle the callToAction result event:

```
public void resultUpdate(McCallToActionEvent event) throws McException
{
    // Retrieve results from the method invocation
    Object data = event.getData();

    // process results
```

Hints and Tips: Instead of implementing the CallToActionListener interface to receive asynchronous updates as shown above, you could have alternatively called the callToActionSync method, which returns an object of type McResultIfc, which would contain results returned by the invoked method.

Query Manager

Query Manager is an added utility for directly accessing database data on the Central System. This interface allows the developer to create a query statement using SQL syntax and use it to retrieve data directly from a server table. The data retrieval will be processed using the Management Central Java

Framework's user profile "QYPSJSVR", and therefore this interface will only have access to databases where permission has been explicitly granted to the profile.

Any client code that uses this interface will be SQL dependent and will be dependent on the actual table names, fields and formats on the central system. In addition, this interface should not be used to access definitions, tasks, services, or any other Manageable objects that are stored persistently on the central system. This interface should only be used to query non-Manageable data on the central system that is stored in an SQL database.

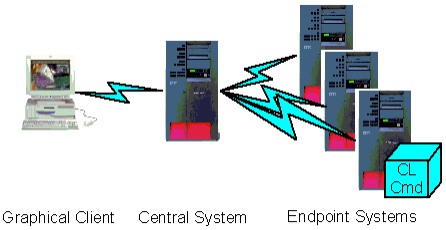
In the example below, an inventory application stores information about hardware and software on the target system. The application creates their own table QAYIVSYS in which this data is stored. The example shows how to query this table using the Query View utility. The results returned from the query come in the form of a Vector, where each element of the Vector is a row matching the selection criteria. Since each row has multiple columns, each element of the Vector is another Vector, where each element is a column from the table. When extracting data from each row, extract it as an Object, as opposed to primitive types (e.g., extract text data as type String, but extract numerical values as Integer, instead of int).

To use Query Manager, you'll need only one class, **com.ibm.mc.client.McQueryView**, and its only static method, performSqlQuery.

```
  // Step 1: Build SQL select statement
  String select = "SELECT * FROM QUSRSYS.QAYIVSYV WHERE SYSTEM_KEY = " +
key;

  // Step 2: Execute the query.
  Vector rows = McQueryView.performSqlQuery(select);

  // Step 3: Retrieve results data
  for(int i = 0; i < rows.size(); i++)
  {
      // get first row
      Vector singleRow = (Vector)rows.elementAt(i);

      // 2nd field in DB "SYS_VALUE"
      String wrkValueName = (String)singleRow.elementAt(1);

      // 3rd field in DB "DATA_TYPE"
      Short dbDataType = (Short)singleRow.elementAt(2);

      if (dbDataType.intValue() == 1)
      {
          String valueData = " ";

          // get length of string
```

Management Central Distributed Command Call Application

Overview



In this chapter you will learn about classes provided by the Management Central Java Framework that allow you to run an AS/400 CL Command on multiple remote AS/400 endpoint systems. The **McDistCommandDescriptor** class, with the help from the **McDistributedTaskView** and **McDistributedTaskListView** classes, provide the Management Central Distributed Task functions allowing a Java program to execute a non-interactive AS/400 command on multiple systems and return status and results back to the Central System and the graphical client.

To make this happen, the jMC uses classes from the AS/400 Java Toolbox. The CommandCall class is used to construct the AS/400 CL command so that the jMC can send and execute the request to the endpoint systems. The CommandCall class is also used to store any AS400Message objects that are returned as the result of executing the command.

Interfaces and Flows

Application Designer

You will first want to determine whether the **McDistCommandDescriptor** class contains the functionality that meets your application needs. Your application would use this function if the interfaces you are calling on the AS/400 are CL commands and you require only minimal status and results about the execution of the command.

Some of the specifications of the **McDistCommandDescriptor** are:

- Can run a single AS/400 CL command at a time
- Runs asynchronously, meaning a once the task is distributed to the endpoints, each endpoint runs the task in parallel and reports status back upon completion of the task
- Returns a limited defined set of status values
- Returns AS/400 messages within the CommandCall object
- The command will run under the profile of the owner

GUI Developer

The following scenarios describe how to use the CommandCall, **McDistCommandDescriptor**, **McDistributedTaskView**, and **McDistributedTaskListView** classes.
Classes and Interfaces:

- com.ibm.mc.client.activity.task.command.McDistCommandDescriptor
- com.ibm.mc.client.activity.task.McDistributedTaskView
- com.ibm.mc.client.activity.task.McDistributedTaskListView
- com.ibm.mc.client.activity.McActivityDescriptorSelectionCriteria
- com.ibm.mc.client.McManageableSelectionCriteria
- com.ibm.as400.access.CommandCall

Scenario 1: Create and Execute a Distributed Command Call Task

It is very easy for the GUI developer to use the **McDistCommandDescriptor** class to create and execute a distributed command call task. Here are the steps to get you started:

1. Create an instance of an AS/400 Java Toolbox CommandCall object and set the command.
2. Create an instance of a Distributed Command Descriptor specifying the Task Name, Task Owner, Task Description, Sharing, and a System Group; then set the command using the CommandCall object created in step 1. Note that the getSystemGroup method used in the Descriptor's constructor must be supplied by the user to retrieve the list of endpoint systems on which to execute the command.
3. Create an instance of a Distributed Task View object specifying the Distributed Command Descriptor created in step 2.
4. Tell the Distributed Task View to add the instance of your task so that it can be managed.
5. Call the execute method to distribute and execute the command on all the endpoint systems specified in the System Group.

```
        McDistCommandDescriptor distCommandDesc = null;
        McDistributedTaskView   distCommandView = null;

        // Step 1: Create an instance of a CommandCall object and set the command
        CommandCall cmdToRun = new CommandCall();
        cmdToRun.setCommand("CRTLIB USRLIB");

        // Step 2: Create an instance of a Distributed Command Descriptor
        distCommandDesc = new
                        McDistCommandDescriptor("Command Task Name",        // Name
                                                "Command Task Description", //
    Description
                                                McManageable.NONE,          // Sharing
                                                getSystemGroup(),           // System
    Group
                                                cmdToRun,                   //
    CommandCall
                                                null);                      //
    Category
```

> **Hints and Tips:** If you look at the Toolbox documentation for CommandCall, you will see a constructor that accepts an AS400 Object. In the example above, if you supply a CommandCall containing an AS400 Object that is already connected to some AS/400 endpoint system, the jMC will accept it, but will overwrite the Object. Since the AS400 is used to execute native calls on the Endpoint, the data stored within the Object must correspond with the current system. If it does not, the jMC will construct a new AS400 on the endpoint system and use that Object for command processing.

Scenario 2: Get list of Distributed Command Call Tasks

If you need to retrieve a list of Distributed Command Call tasks that would include the task you created in Scenario 1, this next step will show you how. There are only a few steps needed here.

1. Set up the selection criteria to only get tasks of the class **McDistCommandDescriptor**
2. Create an instance of the **McDistributedTaskListView** to manage the list of tasks
3. Ask the Distributed Task Manager to return to you a list of Distributed Command Call Tasks

```
      McManageableSelectionCriteria selCriteria    = null;
      Vector                        retrievedTasks = new Vector();

      // Step 1: Define selection criteria to get a list of Distributed Command Tasks
      selCriteria = new McManageableSelectionCriteria(
                  "com.ibm.mc.client.activity.task.command.McDistCommandDescriptor", //
  Class
                  McManageable.ALL,  // Category
                  null,              // List of owners (only used if next parameter is
  true)
                  false,             // Include shared activities
                  0);                // Last changed date of the activity
```

Scenario 3: Delete a Distributed Command Call Task

If you need to delete a Distributed Command Call task, or a list of them, the Distributed Task View and List View classes provide the methods for you. Deleting a task removes the task from the Management Central databases and is no longer a managed task. When working with the View object, you can simply call removeManageable on the instance of the object itself; for a List of Views, you can simply call removeManageableList on the ListView instance.

```
// Step 1: Tell the Distributed Task View to remove the instance of your task
view.removeManageable();

// Or, for a Distributed Task List View
```

Scenario 4: Change a Distributed Command Call Task

To save changes of an existing task on the Central System, the Distributed Task View provides the method for you to use. Prior to calling change, you would have a reference to a task that you previously created or retrieved from a list of tasks. With the reference to the task, you may have the end user modify it by displaying a property page and using the appropriate set methods to update the task instance. When you have the task instance up to date, you can tell the Distributed Task View to store the changes. When working with the task view object you can simply call changeManageable on the instance of the object itself.

```
// Step 1: Change the Distributed Task Descriptor locally
distCommandDesc.setDescripton( "New Descripton" );

// Step 2: Tell the Distributed Task View to change the instance of your task
```

Scenario 5: Get asynchronous status and results of a Task

Once you've created your task and called execute, the task will run asynchronously with other activities. If you want to monitor the status of the request, you will want to attach a class to handle status and results that are returned from each endpoint system. This class will implement the **McStatusDetailListener** and **McResultDetailListener** interfaces. Note: In the example shown below, the same class implements both these interfaces so we pass in this as the object to handle status and result updates.

```
 // Previously you would have retrieved a list of task views and selected the
 // one task view that you want to monitor status and results

 // Attach to be notified when a Status or Result object is received
 view.attachStatusDetailListener(this);
 view.attachResultDetailListener(this);

 // Display a status window or dialog

 // When the user is done with this window or dialog, detach the status and
 // result listeners before closing the window


public void statusUpdate(McStatusEvent event) throws McException
{
  // Get the status object out of the event information
  McStatusIfc status = event.getStatus();

  // If the overall status value indicates the task has finished
  if ( status.getLevel() == McStatusIfc.DistributedAct && status.isFinalized()
)
  {
    // Handle status update


public void resultUpdate(McResultEvent event) throws McException
{
  // Get the result object out of the event information
  McResultIfc result = event.getResult();

  // Since the result object is a hierarchy of results for each Endpoint
System
  // specified in the task, you need to get the results for the specific
Endpoint
  // System to see its details.
  McResultIfc childResult = (McResultIfc)(result.findChild("system1"));

  // Be sure that the result object is an instance of CommandCall before
  // performing CommandCall type methods.
  if(childResult != null && childResult.getResultData() instanceof
CommandCall)
  {
    // Get the CommandCall object our of the result object
    CommandCall cmd = (CommandCall)childResult.getResultData();

    if ( (cmd.getMessageList() != null) && (cmd.getMessageList().length > 0) )
    {
      // Retrieve list of AS/400 messages
      AS400Message[] msgs = cmd.getMessageList();
```

Hints and Tips

What exactly does the execute do for a Distributed Command Call?

The execute tells the Management Central Java Framework to distribute the task to every system specified in the system group. Once delivered to the endpoint system, the CommandCall object will be extracted from the task and run. If the return code value from the run() method indicates an error (false), then McStatusIfc.Failed will be returned in the status event. If the return code value indicates success (true), then a value of McStatusIfc.Completed will be returned in the status event.

In either case, results are also constructed and returned to the Central System and available to the client. After the CommandCall object is run, any messages are placed in the CommandCall object. In your resultUpdate method, you can interrogate the result information, extract the CommandCall, and check to see if there are any messages that have been returned.

The actual execution of the command will occur in a Client Access Server job. This job will run under the user profile of the owner of the task. For more details, see the JavaDoc for CommandCall.

What exactly does the cancel do for a Distributed Command Call?

When the CommandCall object is requested to run on the endpoint system, it will start a new Client Access server job. The Distributed Command Call application will remember this job name. When the cancel request is received on the endpoint system, an ENDJOB immediate command will be executed to end the Client Access server job processing the execute request. If the ENDJOB command was executed a McStatusIfc.Canceled status will be returned in the status event. If the execute request had already completed, then the cancel request will be disregarded.

Terms, Classes, and Interfaces

Thing	Type	What to do with it	Purpose
McDistCommandDescriptor	Class	Use	An application will create one of these objects to execute an AS/400 CL command on multiple systems.
McDistributedTaskView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a status has changed or when results have been received. Task Actions: - execute - displayScheduleDialog - schedule - cancel

			Distributed Task Manager: - addManageable - getManageable - changeManageable - removeManageable Event Listeners: - attachConnectionListener - detachConnectionListener - attachStatusDetailListener - detachStatusDetailListener - attachResultDetailListener - detachResultDetailListener
McDistributedTaskListView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate lists of tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a task has been created, changed, updated, and deleted. Distributed Task Manager: - getManageableViews - removeManageableList Event Listeners: - attachManageableListener - detachManageableListener
McManageableSelectionCriteria	Class	Use	Use this class in conjunction with the McDistributedTaskListView to specify the type of tasks to retrieve. The selection criteria allows you to specify Type, Category, Sharing, etc.
McActivityDescriptorSelectionCriteria	Class	Use	Like McManageableSelectionCriteria, you use this class in conjunction with the McDistributedTaskListView to specify the type of tasks to retrieve. This class extends the base to include selection criteria to subset activities based on their status values.
CommandCall	Class	Use	Provided by the AS/400 Java Toolbox: Contains the AS/400 CL command to execute on all the remote systems. This object will also be used to return AS/400 messages if the execution of the command resulted in any joblog messages.

Programming Examples

In CMVC there are a number of test programs available at:
as400a\v5r1m0t.ss03\int\cmvc\java.pgm\yps.ss03\com\ibm\app\client

- TestCmdCall
- TestCmdCallAttach
- TestCmdCallCancel
- TestCmdCallCreate
- TestCmdCallExecute
- TestCmdCallRemove
- TestCmdCallSchedule

The **TestCmdCall** Java program is an all inclusive test program that will create a new task, attach for status and result notifications, execute the task, process status and result events, and remove the task when completed.

The rest of the test programs break the entire test into controllable pieces. The **TestCmdCallCreate** Java program will create a new task. The **TestCmdCallAttach** Java program will associate itself to the task so when the task executes it can receive status and result notifications. The **TestCmdCallExecute** Java program will kick off the execution of the task and will also receive status and result notifications. The **TestCmdCallSchedule** Java program will schedule the task to execute at a later date and time. The **TestCmdCallRemove** Java program will delete the task from the Management Central Task data base on the AS/400. The **TestCmdCallCancel** Java program will attempt to cancel the running task. The **TestCmdCallChange** Java program will change the name of the task.

In these examples it is important to understand the **McKey** concept. When a new task is created, a key is created to uniquely identify the task This key is made up of three parts: the task class, the task name, and the user who owns the task. In the **TestCmdCallCreate** Java program the key is created and assigned when you create a new instance of your task. This happens when you instantiate a new **McDistCommandDescriptor** and perform an addManageable.

```
    // Create an instance of a Distributed Command Descriptor
    McDistCommandDescriptor distCommandDesc = new
                    McDistCommandDescriptor("Command Task Name",        // Name
                                            "Command Task Description", //
Description
                                            McManageable.NONE,          //
Sharing
                                            getSystemGroup(),           // System
Group
                                            cmdToRun,                   //
CommandCall
                                            null);                      //
Category
```

Notice that when you construct a new **McDistCommandDescriptor**, you specified two out of the three essential parts of the key:

1. Task Class = McDistApiDescriptor
2. Task Name = "DistApiDesc-TestTask

The owner is determined by the Management Central Java Framework.

Now in the **TestCmdCallAttach**, **TestCmdCallExecute**, **TestCmdCallCancel**, **TestCmdCallChange**, and **TestCmdCallRemove** Java programs, you can get the task again by constructing the **McKey** and creating a new Distributed Task View.

```
McKey tempKey = new McKey(

"com.ibm.mc.client.activity.task.command.McDistCommandDescriptor",
                    "Command Task Name");
```

Be Aware: The test programs referenced above were created for the purpose of testing the Management Central Java Framework, and no attention has been paid to quality GUI programming concepts. You should not use these tests as guides on exactly how to set up your client, but only on how to interact with **Distributed Descriptor** and **View** Objects within the jMC.

For instance, while the jMC provides asynchronous status and results from each endpoint specified in the system group, there is no alternative method for receiving synchronous status or results. After calling the view's execute method, these test programs suspend the main thread until a status update has arrived, after which the main thread is resumed, and execution completes. What this means is that if you use more than one endpoint system, you will lose all status and results information from every system in your system group *except* the one that finishes first.

So, while the test cases will show you how to send and receive data from your central site in the distributed environment of the Management Central Java Framework, it does not give advice as to how to handle that data.

See the section on Plugging Into Operations Navigator for a more robust implementation of handling status and result updates within this asynchronous environment.

Management Central Distributed API Application

Overview

In this chapter you will learn about classes that allow you to call an AS/400 Application Programming Interface(API) on multiple remote AS/400 endpoint systems. You will use classes provided by the AS/400 Java Toolbox in conjunction with classes provided by the Management Central Java Framework. The **McDistApiDescriptor** class with help from the **McDistributedTaskView** and **McDistributedTaskListView** classes provide the Management Central Distributed Task functions allowing a Java program to run a program or service program API on multiple groups of systems and return status and results back to the Central System and the graphical client workstation.

You may choose to use the **ProgramCall**, **ServiceProgramCall**, or **ProgramCallDocument** classes from the AS/400 Java Toolbox to define and construct your API request. Passing one of these objects to the Management Central Distributed API Application, Management Central can send and run the API on the endpoint systems. These classes use the **AS400Message** class to return messages that may have been logged in the job log as a result of the API execution. This message array will be returned in the result for each endpoint system receiving the request.

Interfaces and Flows

Application Designer

As the application designer, you will want to determine whether the **McDistApiDescriptor** class contains the functionality your application needs. Your application would use this function if the interface you are calling on the AS/400 is an Application Programming Interfaces(API) and you require only minimal status and results about the execution of the API.

Some of the specifications of the **McDistApiDescriptor** are:

- Can run a single AS/400 API at a time
- Runs asynchronously. Meaning a task is created and status needs to be checked for completion of the task
- Returns a limited defined set of status. It will return Completed when no messages are returned and Failed when any message is returned.
- Returns any output parameters within the ProgramCall, ServiceProgramCall, or ProgramCallDocument resulting object
- Returns AS/400 messages within the ProgramCall, ServiceProgramCall, or ProgramCallDocument resulting object
- The API will run under the user profile of the owner.

Gui Developer

The following scenarios describe how to use the **ProgramCall**, **McDistApiDescriptor**, and **McDistributedTaskView** classes. Processing is very similar when using the **ServiceProgramCall** or **ProgramCallDocument** AS/400 Java Toolbox classes.

Classes and Interfaces:

- com.ibm.mc.client.activity.task.api.McApiData
- com.ibm.mc.client.activity.task.api.McDistApiDescriptor
- com.ibm.mc.client.activity.task.api.McEndpApiDescriptor
- com.ibm.mc.server.activity.task.api.McEndpApiAction
- com.ibm.mc.client.activity.task.McDistributedTaskView
- com.ibm.mc.client.activity.task.McDistributedTaskListView
- com.ibm.mc.client.activity.McActivityDescriptorSelectionCriteria
- com.ibm.mc.client.McManageableSelectionCriteria
- com.ibm.as400.access.AS400Message
- com.ibm.as400.access.ProgramCall
- com.ibm.as400.access.ServiceProgramCall
- com.ibm.as400.data.ProgramCallDocument

Scenario 1: Create and Execute a Distributed API Application Task

It is very easy for the GUI developer to use the **McDistApiDescriptor** class to create and execute a distributed API task. Here are the steps to get you started:

1. Create an instance of an AS/400 Java Toolbox **ProgramCall** class and associated parameters.
2. Create an instance of a Distributed API Descriptor specifying the Task Name, Task Owner, Task Description, Sharing, System Group, and the **ProgramCall** object created in step 1. Note that the getSystemGroup method used in the Descriptor's constructor must be supplied by the user to retrieve the list of endpoint systems on which to execute the command.
3. Create an instance of a Distributed Task View specifying the Distributed API Descriptor created in step 2.
4. Tell the Distributed Task View to add the instance of your task so that it can be managed.
5. Call the execute method to distribute and call the API on all the endpoint systems specified in the System Group.

```
    McDistApiDescriptor   distApiDesc = null;
    McDistributedTaskView distApiView = null;

    // Step 1: Create an instance of a ProgramCall object and set associated
parameters
    // Create and/or retrieve AS400 object
    AS400 as400System = getSystem();

    // Create the path to the program.
    QSYSObjectPathName programName = new QSYSObjectPathName("QSYS", "QWCRSSTS",
"PGM");

    // Create the program call object.  Associate the object with an AS400
object.
    ProgramCall apiSystemStatus = new ProgramCall(as400System);

    // Create the program parameter list.  This program has five
    // parameters that will be added to this list.
    ProgramParameter[] parmlist = new ProgramParameter[5];

    // The AS/400 program returns data in parameter 1.
    parmlist[0] = new ProgramParameter( 64 );

    // Parameter 2 is the buffer size of parm 1.
    AS400Bin4 bin4 = new AS400Bin4( );
    Integer iStatusLength = new Integer( 64 );
    byte[]  statusLength = bin4.toBytes( iStatusLength );
    parmlist[1] = new ProgramParameter( statusLength );

    // Parameter 3 is the status-format parameter.
    byte[] format = McUtilities.stringToByteArray(as400System.getCcsid(),
"SSTS0200");
    parmlist[2] = new ProgramParameter(format);

    // Parameter 4 is the reset-statistics parameter.
    byte[] reset = McUtilities.stringToByteArray(as400System.getCcsid(), "*NO
   ");
    parmlist[3] = new ProgramParameter( reset );

    // Parameter 5 is the error info parameter.
    byte[] errorInfo = new byte[32];
    parmlist[4] = new ProgramParameter( errorInfo, 0 );

    // Set the program to call and the parameter list to the program call
object.
    apiSystemStatus.setProgram( programName.getPath(), parmlist );

    // Step 2: Create an instance of a Distributed API Descriptor
    //         Create the Management Central Distributed API Descriptor task
```

Note: Step 1 above requires you to supply your own AS400 Object via the getSystem method. The only thing it is used for in this example is in converting the Strings into byte array representations. The AS400 Object is necessary so that the conversion routine knows which character set ID (CCSID) to use on the conversion. When executing on the endpoint, the AS400 Object contained within the ProgramCall will be replaced with an object representing the current system. However, the CCSID issue leads to some subtle complexities when dealing with this text conversion.

First, it means that each AS/400 endpoint system in your system group MUST have the same CCSID as the AS400 you use to construct the ProgramParameter list. If it does not, the ProgramParameters may not be interpreted correctly on the endpoint system, causing your program to fail.

Second, it means that a connection must be established to retrieve the CCSID value for some AS400. You can do this either by retrieving the current central system from Operations Navigator, providing an AS/400 system name, user profile, and password with which to connect, or by creating an empty AS400 object, and allowing it to prompt the user for the appropriate information.

Of course, none of these issues arise if you have no need for text-to-byte array conversion as part of your ProgramParameter setup.

Scenario 2: Get a list of Distributed API Application Tasks

If you need to retrieve a list of Distributed API Tasks that would include the task you created in Scenario 1, this next step will show you how. There are only a few steps needed here.

1. Set up the selection criteria to only get tasks of the class **McDistApiDescriptor**
2. Create an instance of the McDistributedTaskListView to manage the list of tasks
3. Ask the Distributed Task List View to return to you a list of Distributed API Tasks

```
  public Vector listTasks()
  {
    McManageableSelectionCriteria selCriteria    = null;
    McDistributedTaskListView     viewList       = null;
    Vector                        retrievedTasks = new Vector();

    // Step 1: Define selection criteria to get a list of Distributed API Tasks
    selCriteria = new McManageableSelectionCriteria(
                "com.ibm.mc.client.activity.task.api.McDistApiDescriptor", // Class
                McManageable.ALL, // Category
                null,             // List of owners (only used if next parameter is
true)
                false,            // Include shared activities
                0);               // Check the last changed date of the activity

    // Step 2: Create a new Task List View to manage your tasks
    viewList= new McDistributedTaskListView(selCriteria);
```

Note: This example will retrieve all the tasks of type **McDistApiDescriptor** and return them in a Vector. The **McDistApiDescriptor** class is the same class used in Scenario 1 step 2 when you created the task.

Scenario 3: Delete a Distributed API Task

If you need to delete a Distributed API task, or a list of them, the Distributed Task View and List View classes provide the methods for you. Deleting a task removes the task from the Management Central databases and is no longer a manageable task. When working with the Distributed Task View object, you can simply call removeManageable method on the instance of the object itself; for a List of Views, you can simply call removeManageableList on the ListView instance.

```
// Step 1: Tell the Distributed Task View to remove the instance of your task
distApiView.removeManageable();

// Or, for a Distributed Task List View
```

Scenario 4: Change a Distributed API Task

To save changes of an existing task on the Central System, the Distributed Task View provides the method for you to use. Prior to calling change, you would have a reference to a task that you previously created or retrieved from a list of tasks. With the reference to the task, you may have the end user modify it by displaying property pages and using the appropriate set methods to update the task instance. When you have the task instance up to date, you can tell the Distributed Task View to store the changes.

```
// Step 1: Change the Distributed Task Descriptor locally
distCommandDesc.setDescripton( "New Descripton" );

// Step 2: Tell the Distributed Task View to change the instance of the task
```

Terms, Classes, and Interfaces

Thing	Type	What to do with it	Purpose
McDistApiDescriptor	Class	Use	An application will create one of these objects to run an AS/400 Application Programming Interface (API) on multiple systems.
McDistributedTaskView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI databeans can be notified directly when a status has changed or when results have been received. Task Actions: - execute - displayScheduleDialog - schedule - cancel View Actions: - addManageable - getManageable - changeManageable - removeManageable Event Listeners: - attachConnectionListener - detachConnectionListener - attachStatusDetailListener - detachStatusDetailListener - attachResultDetailListener - detachResultDetailListener
McDistributedTaskListView	Class	Use	This class bridges your application to MC Java Framework functions to manipulate lists of tasks and provide additional capabilities that make GUI programming easier. By using the attach and detach capabilities, GUI classes can be notified directly when a task has been created, changed, updated, and deleted. Distributed Task Manager: - getManageableViews - removeManageableList Event Listeners: - attachManageableListener - detachManageableListener
McManageableSelectionCriteria	Class	Use	Use this class in conjunction with the

			McDistributedTaskListView to specify the type of tasks to retrieve. The selection criteria allows you to specify Type, Category, Sharing, etc.
McActivityDescriptorSelectionCriteria	Class	Use	Like McManageableSelectionCriteria, you use this class in conjunction with the McDistributedTaskListView to specify the type of tasks to retrieve. This class extends the base to include selection criteria to subset activities based on their status values.
ProgramCall	Class	Use	Provided by the AS/400 Java Toolbox: Contains the AS/400 program API and parameters to call on all the remote systems. Output parameters will be returned in a resulting ProgramCall object. This object will also be used to return AS/400 messages if the execution of the API resulted in any messages.
ServiceProgramCall	Class	Use	Provided by the AS/400 Java Toolbox: Contains the AS/400 service program API and parameters to call on all the remote systems. Output parameters will be returned in a resulting ServiceProgramCall object. This object will also be used to return AS/400 messages if the execution of the API resulted in any messages.
ProgramCallDocument	Class	Use	Provided by the AS/400 Java Toolbox: Contains the AS/400 program API or service program API and parameters to call on all the remote systems. Output parameters will be returned in a resulting ProgramCallDocument object. This object will also be used to return AS/400 messages if the execution of the API resulted in any messages.

Programming Examples

In CMVC there are a number of test programs available at:
as400a\v5r1m0t.ss03\int\cmvc\java.pgm\yps.ss03\com\ibm\app\client

- TestApiPgm
- TestApiPgmCallCreate
- TestApiPgmCallAttach
- TestApiPgmCallExecute
- TestApiPgmCallRemove

The **TestApiPgm** Java program is an all inclusive test program that will create a new task, attach for status and result notifications, execute the task, process status and result events, and remove the task when completed.

The rest of the test programs break the entire test into controllable pieces. The **TestApiPgmCallCreate** Java program will create a new task. The **TestApiPgmCallAttach** Java program will associate itself to the task so when it executes it can receive status and result notifications. The **TestApiPgmCallExecute** Java program will kick off the execution of the task and will also receive status and result notifications. The **TestApiPgmCallRemove** Java program will delete the task from the Management Central Task data base on the AS/400.

In these examples it is important to understand the **McKey** concept. When a new task is created, a key is created to uniquely identify the task This key is made up of three parts: the task class, the task name, and the user who owns the task. In the **TestApiPgmCallCreate** Java program the key is created and assigned when you create a new instance of your task. This happens when you instantiate a new **McDistApiDescriptor** and perform an addManageable.

```
distApiDesc = new McDistApiDescriptor("McDistApiTask_Name",       // Name
                                      "McDistApiTask_Descriptor", // Description
                                      McManageable.NONE,          // Sharing
                                      getSystemGroup(),           // System Group
                                      apiSystemStatus,            // ProgramCall
                                      null);                      // Category
distApiView = new McDistributedTaskView(distApiDesc);
distApiView.addManageable();
```

Notice that when you construct a new **McDistApiDescriptor**, you specified two out of the three essential parts of the key:

1. Task Class = McDistApiDescriptor
2. Task Name = “DistApiDesc-TestTask”

The owner is determined by the Management Central Java Framework.

Now in the **TestApiPgmCallAttach**, **TestApiPgmCallExecute**, and **TestApiPgmCallRemove** Java programs, you can get the task again by constructing the **McKey** and creating a new Distributed Task View.

```
      McKey tempKey = new
McKey("com.ibm.mc.client.activity.task.api.McDistApiDescriptor",
                                "DistApiDesc-TestTask");
      distApiView= new McDistributedTaskView(tempKey);
```

Be Aware: The test programs referenced above were created for the purpose of testing the Management Central Java Framework, and no attention has been paid to quality GUI programming concepts. You should not use these tests as guides on exactly how to set up your client, but only on how to interact with **Distributed Descriptor** and **View** Objects within the jMC.

For instance, while the jMC provides asynchronous status and results from each endpoint specified in the system group, there is no alternative method for receiving synchronous status or results. After calling the view's execute method, these test programs suspend the main thread until a status update has arrived, after which the main thread is resumed, and execution completes. What this means is that if you use more than one endpoint system, you will lose all status and results information from every system in your system group *except* the one that finishes first.

So, while the test cases will show you how to send and receive data from your central site in the distributed environment of the Management Central Java Framework, it does not give advice as to how to handle that data.

See the section on Plugging Into Operations Navigator for a more robust implementation of handling status and result updates within this asynchronous environment.

Advanced Features

All the advanced features (documented in the Advanced Features section on page 54) plus all the actions that can be taken on Tasks (documented in the Tasks sections on page 34) also apply to the distributed command and distributed API applications. For instance, the DistributedCmdCall can be scheduled, or, the DistributedApiDescriptor object can be constructed to auto-increment the name. All of the flexibility of Tasks can be applied to these two Task implementations.

Utilities

The following scenarios are provided to supplement the information in each of the preceding application sections. They are considered "utilities" only because they aren't core functions provided by the Management Central Java Framework, and are not *necessary* to develop a distributed activity. However, they are quite useful, and most distributed activities wouldn't be very useful without them. Therefore, they are provided here as a reference.

Handling Exceptions

In this scenario, a **com.ibm.mc.client.util.McException** is caught and interrogated to determine the cause of an error. If, for whatever reason, an Exception is thrown during processing, the Management Central Java Framework will always attempt to catch the Exception, whether it was thrown initially by some jMC method or by any other Java method, and package it into a McException. This McException may then be caught and re-thrown with additional information from the caller of the errant message, and so on until the Exception is finally re-thrown remotely to the client. Therefore, when this McException is returned to the client, it may have multiple nested Exception objects within it.

In your catch block, you may interrogate the McException with the containsErrorID method of McException to determine if a particular error ID. This identifier must be either an ID defined in the **McService** class, or a class name of a predefined Java Exception class. (McService refers to the logging of service messages, or job logging on the AS/400, and is not to be confused with the services we've defined as activities in the jMC). Alternatively, the error can be output to the client for informational purposes with the printStackTrace method. Consult the JavaDoc on McException for further information on how to use the McException class, and McService to view predefined error ID strings.

```
try
{
    view.addManageable();
}
catch( McException e )
{
    if( e.containsErrorId("java.sql.MCJS_MGBL_DUPKEY") )
       return "Key Error";
    else if( e.containsErrorId("java.io.IOException") )
       return "I/O Error";
    else
    {
```

Tracing Messages

The Management Central Java Framework provides a default tracing mechanism to make it easier for you to trace messages. Using class **McTrace** in package com.ibm.mc.client.util, tracing messages to a file becomes a one-step process. For instance, when retrieving instances of your Distributed Command Tasks off the server (as you did in the Distributed Command Call Application Section of this document, Scenario 2), you may want to trace certain elements of the execution. Only a few steps are needed here:

1. Initialize trace with the file name you wish to trace to, and the level of data you would like to trace. Valid values for level are Error, Warning, Information, and Diagnostic. If trace level is stet to Error (the default), only messages with Error severity will be logged; if trace level is Information, all Informational, Warning, and Error messages will be logged.
2. Execute your Management Central function.
3. Check trace level, and trace appropriate messages

```
    McManageableSelectionCriteria selCriteria    = null;
    Vector                        retrievedTasks = new Vector();

    // Step 1: Initialize trace
    String fName = "C:\\MGTC.Java.Service.Log";

    // specify a descriptive component name here. You'll be able
    // to filer trace messages based on your component name.
    McTrace.setFileName("MySoftwareComponent", fName);
    McTrace.setTraceLevelOn(McTraceable.INFORMATION);

    // Step 2: Execute Management Central Function
    // Define selection criteria to get a list of Distributed Command Tasks
    selCriteria = new McManageableSelectionCriteria(
                "com.ibm.mc.client.activity.task.command.McDistCommandDescriptor", //
Class
                McManageable.ALL,  // Category
                null,              // List of owners (only used if next parameter is
true)
                false,             // Include shared activities
                0);                // Last changed date of the activity

    // Create a new Task List View to manage your tasks
    McDistributedTaskListView viewList = new McDistributedTaskListView(selCriteria);

    // Step 3: Tracing an informational message
    if( McTrace.isTraceInformationOn() )
        McTrace.logInformation( getClass().getName(), "Executing getList from
server" );

    try {
        retrievedTasks = viewList.getManageableViews();

    } catch( McException mce ) {
```

This will trace the number of Distributed command Call Tasks that were found on the server, and that match your selection criteria, or alternatively, if an exception occurs, then the exception will be traced.

Service Log

The Management Central Java Framework also provides you with a mechanism to joblog messages on the server. This scenario makes use of the **McService** class to log the message, and the**McException** classe to build the data to be logged. Documented steps are as follows:

1. Build an Object array that contains substitution Strings for the message you wish to log. The substitution is defined for the specific McService message that is being logged. If you supply your own messages, you'll need to determine if substitution text is required for that type of message.
2. Build a McException using your specific McService or supplied message, and substitution text.
3. Tell the MC Java Framework to log the data. The message will be logged to the job that the jMC server is executing in. This job will have a unique system number, but will always have a job name of QYPSJSVR and will be running under user QYPSJSVR.

```
try
{
  // execute code here
}
catch( McException e )
{
  // Step 1: Build a McException with substitution data
  Object[] subs = {"evaluate", getClass().getName()};

  // Step 2: Create the McException object to log using the
  //         MCJS_METHOD_INVOKEFAIL message from McService
  McException e1 = new McException(McService.getMessage(
                       McService.MCJS_METHOD_INVOKEFAIL,
                       McService.DIAGNOSTIC, subs), e);
```

As a general rule, anything that gets service logged should also be traced, checking if trace is on as in the previous example. Additionally, you don't need to log messages only when exceptions occur. Any time you feel is appropriate, you can generate a **McMessage** using McService's getMessage method, or by constructing your own, and logging the McMessage with the log method.

Additional Utilities

Many convenience classes and methods exist to allow you to do common tasks within the Management Central Java Framework. All classes discussed here reside in the com.ibm.mc.client.util package. If you find you're writing your own convenience methods for tasks you need to execute in multiple places within your code, consult the JavaDoc for these classes - chances are you'll find exactly what you're looking for.

McUtilities	Contains data management utilities. Some make byte array manipulation easier for the user; some simplify serialization and deserialization; some are for data conversion.
McProcess	This class, located in com.ibm.mc.server.util, is dependent on the AS400 operating system, and can be used to configure a process external to the Management Central Java environment
McMethodThread	Useful for applications that that wish to process method requests on a privately maintained thread but wish to abstract the details of thread management. Class can be used to queue, de-queue and invoke methods.
McMethodQueue	Provides a default method queuing mechanism. Used alone, it only provides standard queue functionality, but when used in conjunction with a McMethodThread, queued methods can be automatically invoked by the jMC.

AS/400 Deployment

Now that you've created your application plug-in for running within the Management Central Java Framework, how will the framework find your classes? Easy. Just create or update a System level environment variable, named specifically QYPSJ_APP_CLASSPATH with the IFS path to your classes. When starting the Management Central Java server, it will append the value of this new variable to it's preexisting classpath, allowing it to find your classes. Contact your system administrator for instructions on how to set the environment variable.