Global Knowledge®

Expert Reference Series of White Papers

Understanding the AIX Object Data Manager

Understanding the AIX Object Data Manager

Iain Campbell, UNIX/Linux Open Systems Architect, eLearning Specialist

Introduction

AIX was first announced as a product in 1986 as the operating system (OS) for the PC RT, IBM's first commercial RISC processor system. At that time UNIX as an operating system was in its childhood—SUN Microsystems had released SunOS in 1982, and HP-UX debuted in 1984. But at that point there was little standardization across these UNIX variants—the Portable Operating System Interface (POSIX) standards group within the IEEE were working on a set of standards for UNIX-like operating systems, but the first of those would not be released until 1988, and it dealt largely with kernel internals, not issues of interest to the systems administrator like input/output (I/O) device management, network configuration, or package management.

The architects of AIX felt that including a simple database engine as an integral part of the OS and using that database as a tool to organize configuration data would be a benefit to the system administrator. Hence, the Object Data Manager (ODM) was born. If you are coming to AIX from another UNIX system, the ODM will be new to you. Fortunately it is not so very complicated, and once you understand it, you will find that the ODM meets its objectives, giving the AIX administrator a simple and consistent interface to manage key areas of configuration data.

We will first look at how the ODM is structured, and then we will examine the range of information the ODM manages (and also what it doesn't manage). Finally, we will consider how we can interact with these databases in order to meet the goals the architects had for the ODM, making AIX administration easier.

Structure of the ODM

Strictly speaking, the ODM itself is not a database; rather, it is a database engine that operates on data stored in a database. It is an object-oriented (OO) design, consisting of about 50 or so object classes. As is the case with OO databases, each class has a name and a list of attributes for which each object in that class will possess its own set of values.

Device Management

As device management is one of the key functions of the ODM, let's examine a device example. If a system has access to a disk, then we might want to know where that disk is located in the system, what type of disk it is, if the disk driver is loaded and working, and what name we will use to reference the disk.

To do that, an OO database might have an object class called devices, and that class might have attributes like location, type, status, and name. Then, for each actual device in the system, there would be a corresponding object entered into that class.

So, consider a working SAS disk located in slot three of the internal disk cage; its object entry might look something like this:

```
devices:
        Name = disk01
        Location = slot3
        Type = SAS
        Status = Up
```

A Fibre Channel (FC) attached LUN provisioned from a storage area network (SAN) might look like this:

```
devices:
        Name = disk04
        Location = SAN
        Type = fc_LUN
        Status = Up
```

And, if the type attribute is not restricted to disks, then an Ethernet card might look like this:

```
devices:
        Name = ethernet01
        Location = slot4
        Type = Ethernet
        Status = Up
```

Now, it may be that our SAS disk driver has some optional values that could be adjusted to optimize the performance of the drive. For example, disk drivers might allow multiple I/O requests to queue up for the drive, and we might want to adjust the length of the queue. So, we could add another attribute to our devices class called queue, and our disk01 object might now look like this:

```
devices:
        Name = disk01
        Location = slot3
        Type = SAS
        Status = Up
        Queue = 10
```

This works fine for one disk, but what happens if this queue doesn't work the same way for an FC attached LUN? Perhaps LUNs do queuing a different way, and need a different parameter, the value of which will not be relevant for the SAS disk. So now our disk objects might end up looking like this:

```
devices:
        Name = disk01
        Location = slot3
        Type = SAS
        Status = Up
        Queue = 10
        SAN_Queue = null
```

devices:
Name = disk04
Location = SAN
Type = fc_LUN
Status = Up
Queue = null
SAN_Queue = 100

Now, what about our Ethernet card? Neither of the queue parameters matters to it, so its device entry ends up looking like this:

devices:
Name = ethernet01
Location = slot4
Type = Ethernet
Status = Up
Queue = null
SAN_Queue = null

The problem is that each type of device has attributes specific to that device, and if we try to implement all the possibilities in one object class—it doesn't work very well, specifically, it isn't very scalable. What we could do is to create another object class that we can use to store data about device attributes and their values.

If we want to use this approach to handle our disk queuing parameters, then our new attributes object class might have objects like this:

attributes:
Type = SAS
Parameter = Queue
Value = 10
attributes:
Type = fc_LUN
Parameter = SAN_Queue
Value = 100

This can be easily extended to support a parameter value for an Ethernet card as well. So, for example, if I have a Gigabit card:

attributes:
Type = Ethernet
Parameter = Speed
Value = 1000

Now we have some scalability. When the OS is working with disk01, its utilities can be coded to scan the devices class to find the location, name, and other device attributes—and they can also scan the attributes class for all objects having Type = SAS for performance parameters.

Some problems still remain, however. If we have several SAS disks and some are newer than others, then the newer disks might perform best with a queue attribute larger than the value best suited to the older disks. We cannot necessarily assume that because a disk is a SAS disk, it should use the same value for any given attribute.

This can be solved by replacing the attributes class with two classes, which could be called default and custom. For a SAS disk named disk01 needing a queue length of 10, these objects would be relevant:

devices:
    Name = disk01
    Location = slot3
    Type = SAS
    Status = Up

defaults:
    Type = SAS
    Parameter = Queue
    Value = 10

For a newer SAS disk named disk02 needing a queue length of 20, an additional object is created in the custom class:

custom:
    Type = SAS
    Name = disk02
    Parameter = Queue
    Value = 20

For this to work, it is only necessary to add code to specify that where it is possible to obtain conflicting values for the same combination of device and parameter value then the value obtained from the custom class will take precedence.

Now, what about that Ethernet card? Let's assume it is a multi-port card (and most are). How can we handle two sets of parameters using only one device object? We could add an attribute for number of ports, but then how could we address the situation in which each port might need different parameters, such as speed or frame size?

In this case, it makes more sense just to consider each port as a separate device. So, for our two-port Ethernet card, we would have two devices objects:

devices:
    Name = ethernet01
    Location = slot4
    Type = Ethernet
    Status = Up
devices:
    Name = ethernet02
    Location = slot4
    Type = Ethernet
    Status = Up

We might have these Defaults and Custom objects if we want one of the ports to use jumbo frames:

defaults:
Type = Ethernet
Parameter = Jumbo_Frames
Value = no
custom:
Type = Ethernet
Name = ethernet02
Parameter = Jumbo_Frames
Value = yes

Package Management

So far, we have been using I/O device management examples. Let us now consider a different application— package management. Most operating systems have some sort of package manager, which is a system that allows each of the data files that make up the OS to be organized into groups so that they can be selectively installed as desired and patched as necessary. How would we start to lay out our object classes?

We will want to start with a class called files, whose objects might have attributes such as name, size, or permissions, like this:

files:
Name = "/etc/passwd"
Size = 2344
Owner = root
Permissions = 644

We will also want an object class for packages, the groupings we will use to allow groups of files to be managed as a single entity, which is really what package managers are all about. So, we might have an object like this:

packages:
Name = "User Management"
Package_ID = 211

Now the goal is to organize files into packages, and also to add patch management so that our package needs to have a version number. After adding this, our objects would be:

files:
Name = "/etc/passwd"
Size = 2344
Owner = root
Permissions = 644
Package_ID = 211

packages:
Name = "User Management"
Package_ID = 211
Version = 4.0.4

We could go on to add information about prerequisites, installation history, file checksums, and so forth—but this simple example illustrates how the ODM concept can be applied as equally to package management as it can to device management.

Now that we have presented some simple examples of a hypothetical ODM, we will examine how the real AIX ODM is structured.

Structure of the AIX ODM

The actual AIX ODM consists of some 50 or more object classes. The object data for each class is stored in binary formatted files, each file containing the data for one class and named after the class whose data it contains.

Most of these ODM data files are located in `/etc/objrepos` (objrepos is a contraction of object repository). Some of the entries in `/etc/objrepos` are links to data files that reside in `/usr/lib/objrepos`—these are classes that contain data that is read-only. Most of these read-only classes are device related; the equivalent of the defaults class in our example database.

Data Managed by the ODM

It is useful to note that the ODM does not manage all AIX configuration data. The ODM is used to manage data in seven distinct functional areas. Within these areas, all relevant data is managed by the ODM (outside of these areas the ODM is not involved). These are the bodies of data managed by the ODM:

Devices and Network Configuration

Data about all I/O devices and also network configuration data (such as IP configuration) are held primarily in these core classes:

- PdDv (Predefined Devices) — the list of all devices supported by this release of AIX
- PdAt (Predefined Attributes) — default values for all device attributes
- CuDv (Customized Devices) — the devices present on this machine
- CuAt (Customized Attributes) — non-default attribute values

Referring to the example database we developed earlier, the CuDv class would be the equivalent of the devices class, PdAt would be the defaults class, and CuAt would be the custom class.

AIX adds the PdDv class to support plug and play device configuration. Each AIX-supported I/O device will report a unique identifying string to the kernel device configuration code at boot time. The kernel code looks up that ID in the PdDv class to see if it finds a match. If it does, then the device is supported, and the data in the matching object contains information about the driver module to be loaded by the kernel to support that device.

If a device does not get recognized at boot, it could be because it is physically broken and unable to provide the ID string when requested. Or, the ID string provided has no match in the PdDv class, which implies that the device is not supported by the version of AIX that the machine is running. New device support is added to AIX via the release of new Technology Levels (TLs), which will result in changes to the PdDv class to support new devices.

There are more classes relating to data such as parent/child relationships (e.g., multi-port adapters), paths to SAN LUNs, or Vital Product Data (VPD)—the term used by IBM to include hardware-specific data such as manufacturer, serial number, or firmware version. Feel free to explore these on your own systems using the commands we will show you.

Installed Software (Package Management)

The classes associated with the AIX package manager are:

- lpp — description and install state of each package
- product — pre requisite and version information for each package
- inventory — individual file data, one object per each file belonging to a package
- history — dates of most recent install or update events for packages

System Management Interface Tool (SMIT)

Located in `/usr/lib/objrepos`, the set of classes beginning with the string `sm_` contain all the data necessary for SMIT to function. This includes all of the menu/submenu structure, the questions in all the dialog screens, the commands used to generate lists, and so forth. These classes cannot be viewed or edited.

It is interesting to note that as the whole structure of SMIT is database-driven, it makes it easy for IBM to maintain it as new functionality requiring additions and changes to the SMIT menus dictate.

Network Installation Management (NIM)

Should you choose to build a NIM server (and in any AIX environment, you should!)—all the data required to manage network installation, backup, recovery, and software maintenance is stored in a group of classes beginning with `nim_`.

System Resource Controller (SRC)

The SRC is an AIX utility that brings the ordered structure of the ODM to bear on the task of organizing daemon processes. The classes beginning with "SRC" contain the list of all known daemons and the details of the commands needed to manage them. The master SRC daemon (`srcmstr`) is started from `/etc/inittab` at boot and uses the SRC object class data to give the administrator a consistent way to manage daemons using the `startsrc`, `stopsrc`, `lssrc`, and `refresh` commands.

Error Logging and Kernel Dump Configuration

There are some very small classes that contain information concerning the configuration of the kernel log and also kernel dump configuration.

It is worthwhile to note that OS configuration data that does ***not*** fall into these categories is ***not*** managed in any way by the ODM. For example, no user account data, such as login information, password management, and so forth is stored in the ODM. Also, the ODM holds no storage management data, so no information regarding Logical Volume Management (LVM), file systems or mount tables is stored in the ODM.

Finally, let us look at some commands we can use to interact with the ODM.

The ODM from the Command Line

ODM class data can be manipulated directly, although this is very rarely necessary. Normally higher- level commands such as `lsattr` or `chdev` are used, as we will see. However, you might want to do some investigation to get a better understanding of the structure of the actual data a command (like `lsdev`, for example), is operating on. How this can be done is outlined next.

Examining ODM Data

In order to examine the structure of a class:

```
# odmshow CuDv
```

This will give you a C syntax-like definition of each of the data types and names in the class specified as the argument to the command, in this example CuDv, the customized devices class. Note that this is describing only the *structure* of the class—not any actual object *data*.

In order to examine actual objects, do this:

```
# odmget CuDv
```

This will return all the objects in the class in stanza format, similar to the format used in the example database earlier.

In many cases you will not want all the objects in a class; in order to select only the objects of interest, you can add a query term, for example:

```
# odmget -q name=hdisk0 CuDv
```

More complex queries are possible using an SQL-like syntax:

```
# odmget -q "parent like 'scsi*'" CuDv

# odmget -q "parent = scsi0 and connwhere='8,0'" CuDv
```

Deleting Objects

It is also possible to modify objects directly, although that is rarely necessary and should be done with great caution, as there is essentially no error checking. If you make a mistake, it can be very difficult to correct, so be very careful!

For example, consider the following:

```
# odmdelete -o CuDv
```

That will immediately delete all objects in the CuDv class, effectively crippling the device configuration database! Fortunately this can be repaired by retrieving a copy of the file `/etc/objrepos/CuDv` from a backup. Most AIX operations would continue in the meanwhile, although changing a device would likely fail with strange errors until you can recover those lost objects.

An even more dangerous command might be:

```
# odmdelete -o CuDv -q "name like 'en*'"
```

That will delete only objects that match the query. If you know in advance which objects those are, then you have a chance of getting them back. But if you don't really know how many objects the query matched it is too late to find out now because they're gone. Again, the fix here would be to recover the entire CuDv object class data file.

Creating and Changing Objects

If you want to create a new object or objects you can edit an ASCII text file containing the definition of the desired objects in the same stanza format as you would see in the output of `odmget`. If you name that file `NewObjects.asc`, for example, then:

```
# odmadd NewObjects.asc
```

will add the objects detailed in the file. One input file can create multiple objects, potentially in multiple classes according to the stanza data in the input file.

Finally, you can also change object data:

```
# odmchange -o CuDv ChangedObject.asc
```

where the input file is again in the stanza format. Normally, an `odmget` command is used with output redirection to a file to obtain the current values. That file can then be edited to reflect the desired changes and fed back using the `odmchange` command.

These commands that directly manipulate the ODM are, as I mentioned, rarely (if ever) necessary to know for everyday administration purposes, but if you want to get a better idea of how the ODM works at the raw data level this is how to do it.

Let us conclude then by looking at the commands that you do use on an everyday basis, and see how they interact with the underlying ODM.

AIX High Level ODM Dependant Commands

There are five key AIX commands that exclusively interact with the device management component of the ODM. They are `lsdev, chdev, lscfg, lsattr` and `cfgmgr`. If you were to issue the command:

```
# lsattr -El hdisk0
```

and then these ODM commands:

```
# odmget -q name-hdisk0 CuDv
# odmget -q name=hdisk0 CuAt
# odmget -q uniquetype="disk/scsi/osdisk" PdAt
```

it should be clear where the `lsattr` command is getting its information. You will see that `lsattr` is selective; it doesn't report all of the data stored for `hdisk0` in the ODM, just the parameters that are the ones most useful in practice.

The format of the `lsattr` output is also considerably more human friendly than the verbose stanza output format of the raw ODM data. Try this with `lsdev` and `lscfg` and you will see that these commands choose a different subset of the raw ODM data to report.

If there are few objects returned by the `odmget` against the CuAt class, use `chdev` to change one of the attributes of the device. You should now see a new object created in the CuAt class as a result, if the parameter value you entered for your change was a non-default value.

If you wonder what the E and D flags for the `lsattr` command mean, a bit of experimenting will show that using the D flag causes `lsattr` to look only in the predefined classes such as PdDv and PdAt—the default values—while the `E` flag causes `lsattr` to instead report the union of the default and customized class data, giving you the currently effective (hence E for effective) parameter values for the device in question.

The PdDv class is central to AIX plug and play device configuration, driven by the `cfgmgr` command. When you issue `cfgmgr` the ODM device database is rebuilt, and any drivers that need loading into the kernel will be loaded and started. So, if you issue the command:

```
# rmdev -l hdisk1
```

and then look at the ODM objects that relate to `hdisk1`, you should find that they are all still there; all that `rmdev` without the `d` flag does is to unload the kernel driver. Check the CuDv object for `hdisk1` and you should find its status attribute has the value 0, where it would formerly have had the value 1. Issue `cfgmgr`, the driver will be reloaded and the status parameter will return to a value of 1.

If, on the other hand, we add the `d` option to the `rmdev` command:

```
# rmdev -dl hdisk1
```

then in addition to the kernel driver being unloaded the CuDv object for that device is deleted, as well as any CuAt objects referencing that device. Issuing `cfgmgr` will get the device back and the driver loaded, but any non default attributes will revert to default values, meaning that no new CuAt objects will be created as a result of the `cfgmgr` command (except in the case of a disk that has a Physical Volume ID [PVID], as that is stored permanently on the disk and is read into a CuAt object when the disk is configured, indicating as mentioned earlier that storage configuration is not primarily managed in the ODM; the ODM merely holds copies of the data that exists primarily on disk itself).

Package Management

The key high level commands for package management include `installp` and `lslpp`. Try something like:

```
# lslpp -l bos.rte
```

and then:

```
# odmget -q name="bos.rte" lpp

# odmget -q lpp_name="bos.rte" product
```

and compare what you see. Note the lpp_id parameter value from the lpp class and use that as a query against the other package management classes:

```
# odmget -q lpp_id=1 inventory
# odmget -q lpp_id=1 history
```

Conclusion

So, what does all this mean for the AIX administrator? Here are five key things to take away:

1. The ODM operates primarily in the background. The AIX administrator rarely (if ever) has to interact directly with the ODM.
2. The ODM brings structure to the management of key bodies of data needed to configure AIX. This allows for consistency in commands that can be built to implement device and package management, among others.
3. The ODM does *not* manage data relating to user administration or file system management. It does have objects relating to Logical Volume Management; however, the objects in these classes are derived from LVM metadata whose primary and authoritative storage location is located in LVM-specific data structures located on the disk storage itself.
4. Devices in AIX are all essentially plug and play devices. The list of devices supported being determined by the Technology Level (TL) of the AIX release which in turn dictates the data content of the predefined device object classes.
5. The System Resource Controller (SRC) is an offshoot of the ODM which brings consistency to the management of system daemons by adding classes to the ODM to hold data about all daemon processes supported by the release of AIX in play.

Learn More

Learn more about how you can improve productivity, enhance efficiency, and sharpen your competitive edge through Power Systems training.

AIX, UNIX
IBM i
Linux
System Software

Visit **www.globalknowledge.com** or call **1-800-COURSES (1-800-268-7737)** to speak with a Global Knowledge training advisor.

About the Author

Iain Campbell is a mechanical engineer by profession. While responsible for managing several production automation labs at Ryerson Polytechnic University in Toronto, he became distracted by UNIX operating systems. His first experience of AIX was a PC RT used in the lab as a machine cell controller. Iain has been teaching and consulting in AIX and Linux since 1997. He is the author of *Reliable Linux* (Wiley, New York, 2002), as well as several technical papers and curriculum material. He holds LPI and Novell certifications in Linux administration, and is an IBM certified AIX Specialist.