Implementing Distributed Controlled Objects with CORBA

Mark Plesko, J. Stefan Institute, Ljubljana, Slovenia
e-mail: mark.plesko@ijs.si

Abstract

The heart of the control system implementation of the light source ANKA is an object model of devices. It is the Accelerator Control Interface (ACI), a language independent collection of interfaces based on network distributed objects using the CORBA standard. All common accelerator components such as power supplies, vacuum, RF, position and current monitors are defined by means of functions and parameters. The devices are described according to CORBA with the Interface Definition Language (IDL), which presents a language-independent way of defining object interfaces. Each controlled parameter, called device property, is an object by itself, implementing atomic actions such as get/set, increment/decrement, etc. All constants related to a property such as min/max, name, description, etc. are obtained from the property directly by means of remote methods - no direct database access is necessary. Values of the properties are updated asynchronously by means of monitor objects. The ACI is meant to be a standardised interface so that applications and pieces of control systems can be hooked to it from either side. The ACI does not replace existing control system architectures and frameworks but rather tries to use their features in order to be as compatible as possible to those systems.

1. INTRODUCTION

The Accelerator CORBA Interface (ACI) [1,2] defines controlled devices (e.g. power supply, current monitor, vacuum pump, etc.) as network objects that are remotely accessible from any computer through the established client-server paradigm. The underlying communication mechanism is based on the Common Object Request Broker Architecture (CORBA), the state-of-the-art standard for remote objects. This paper assumes that the reader has basic knowledge of CORBA and the concepts behind it, which can be found best on the many links on the Web[3,4].

The ACI is meant to be a standardised interface so that applications and pieces of control systems can be hooked to it from either side. The ACI does not replace existing control system architectures and frameworks. The ACI attempts to build a model of devices that are commonly used in all accelerators. It is the largest common denominator that can be found among different types of accelerator facilities. The ACI is only a definition of device interfaces with the use of IDL, not a definition of an API. An API with more powerful or sophisticated features can be built atop of the IDL interfaces, or even replace the CORBA protocol with a proprietary scheme. However, the IDL interfaces have been defined such that it is possible to perform all necessary control actions just by direct CORBA connections to the ORB that exports the IDL device interfaces. The idea behind this approach is that it is not necessary for the client to load any special API library – any CORBA ORB can find devices on the net and communicate with them.

The main design goals of ACI are:

- Rely on pure CORBA only: don't be language or system specific; don't assume extra functionality in an API library.
- Enforce strong type checking wherever possible. Illegal commands should be discovered already during compile time. Run-time parsing of commands through constructs like send("command") must be avoided. Generic applications can use the introspection capabilities of CORBA (e.g. Interface Repository, Dynamic Interface Invocation, etc.) instead.
- Exploit the object paradigm: the object itself is responsible to provide all data that is relevant to it. Avoid therefore direct access to database servers; leave this to the implementation.
- Don't try to define a generic interface for any possible control system. Specialise on the definition of accelerator objects with the functionality that is common to all accelerators.
- Define object interfaces; don't prescribe their implementation and don't provide client-side functionality. The ACI is merely a hook to the underlying control system.
- Don't allow the client to manipulate control system behaviour. Assume rather that reasonable default values are provided by the system managers through control system configuration tools.
- Use well-proven concepts from existing accelerator control systems.
- Base data transfer on asynchronous calls assuming that all client and server host operating systems are multithread capable as is necessary for GUI-based applications. Keep synchronous calls just for compatibility with legacy systems.
- Use only the core CORBA without CORBA services, because existing ORB implementations do not cover all services yet. But keep to the CORBA interface names for methods that implement the same interface contract.
- Encourage site-specific additions through interface inheritance instead of providing generic bypasses

to strong type checking. However, all interfaces that are defined in the ACI must be implemented at a given site, because client applications from other sources rely on them. Clients written on site can still use the added functionality without penalty.

- A technical issue inspired by Java: pass all parameters to methods by value; in IDL this means that all parameters are declared as “in”. In order to save space, the “in” keyword is omitted in all definitions in this text.

Great care has been taken to be as close as possible to existing control system frameworks like EPICS, CDEV, ACOP, TACO, DOOCS, etc. Many concepts were actually taken directly from one or several of those frameworks.

2. DEFINITIONS

A ***device*** is a CORBA object that corresponds to the model of a physical device, e.g. power supply, vacuum pump, current monitor, etc. The device is the basic entity of the ACI, because it is the most natural concept for modelling physical entities in an accelerator. ***Commands*** that are executed on a device, like on, off or reset are referred to as methods of the device. Each device has a number of ***device properties*** that are controlled, e.g. electric current, status, position, etc. Properties, which are also defined as objects in the ACI, are referred to as IDL attributes of the device. Properties are distinguished by type (integer, double, etc.) and by being read-only or read-write objects. Each such “property object” has specific ***characteristics***, e.g. the value, the minimum, etc. The methods of a property allow to retrieve or modify these characteristics: get(), set(), minVal(), etc.

Finally, a device has ***resources***, which are implementation dependent static key-value pairs, like “position”-“sector 3”, “interface”-“analog 16 bits”, etc. Resources are not used by the ACI and by generic applications. However, they are provided as a generic way to retrieve information that is not covered by the ACI from a database.

Most of the device commands and property methods are executed asynchronously by the remote object. The results of the operations are communicated to the client by means of a ***callback***. A callback is an object interface that must be implemented by the client, so that it can be invoked by the remote object. During this process, the remote object functions as a client and the client performs as a server.

A ***device server*** is a CORBA ORB that implements one specific device interface. A device server usually communicates with the hardware via VME, fieldbus, or similar. Note that devices of the same type, i.e. having the same IDL interface, can exported by several different device servers, residing on different hosts. A typical example is an accelerator complex with an injector and a storage ring. The device server for the injector exports the power supplies of the injector, while the device server for the storage ring exports the power supplies of the storage ring. Both types of power supplies have the same IDL.

3. DESIGNING THE OBJECT MODEL

When designing the object model, i.e. defining our objects and their relations, we have to decide which things from the real world our objects correspond to. Furthermore, we have to determine the way to communicate with objects.

These are design decisions, which often depend on taste. However, a proper and consistent design allows for efficient coding and for seamless interconnection among the individual parts of the control system. An object model that is close to the real world is much more intuitive to programmers, thus requires less learning and results in fewer mistakes.

3.1 Devices and Their Properties are Objects

We believe that the most human-oriented way is to represent each accelerator device with a respective object. Such a concept is seen best in a graphical control panel (figure 1).

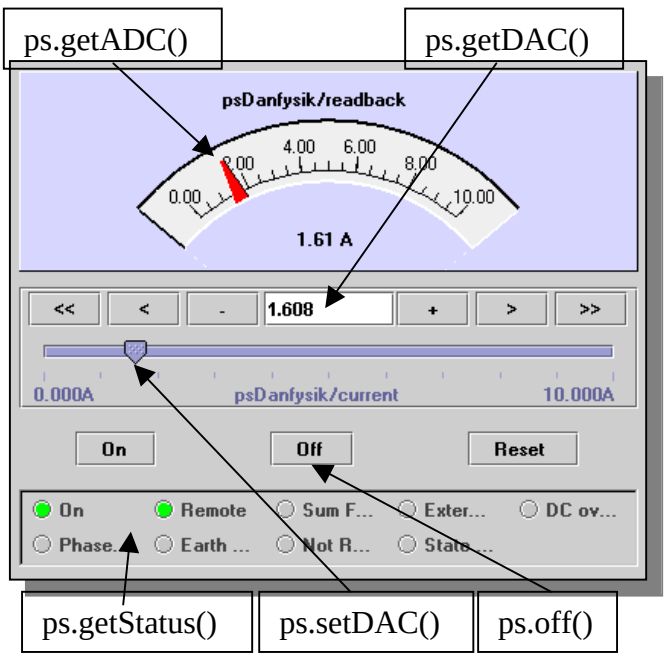

Figure 1: The elements of a control panel relate to the properties of one accelerator device, a magnet power supply.

CORBA allows very fine-grained definition of objects and their properties. With a control system we control accelerator devices, such as power supplies, beam position monitors, etc. So it is natural that we represent those devices in the computer with objects. In the network, they become CORBA objects.

The following example analysis of a power supply shows how a device object looks and what kind of properties it has.

Table 1: properties and commands of a power supply:

Property	**Type**	**Access**
ADC:	double	Read only
DAC:	double	Read/write
Status	bits	Read only
Command	**Input type**	**Return value**
on():	none	void
off():	none	void

We see that properties have different types and also differ in whether they are read-only or also can be written to. But there is more to properties. A property, such as power supply current, has properties of its own: a minimum, a maximum, a description, units, etc. To avoid confusion, the properties of properties are called characteristics.

In most cases, all properties have the same type of characteristics. There is a difference between read-only and read-write properties, as the former have no command to write. Another difference is between properties of type double and type bits (actually an unsigned integer containing a bit pattern), as the latter have no minimum or maximum. But all these differences can be systematized into a matrix of few classes:

Table2: A matrix of primitive types of control properties

TYPE	**double**	**bit-pattern**	**enumerated**
readonly	ROdouble	ROpattern	ROenum
readwrite	RWdouble	-	RWenum

We can add another dimension to the table by adding arrays, or sequences, respectively, of those primitive types. Sequences are used to transmit arrays of data. Usually they do not correspond to a single property but rather to a collection of properties of the same type.

The type char is not provided on purpose. It is not necessary in most cases, anyway. Most often, chars are used in serial communications, where values or status are read/written in ASCII mode, but need to be converted at some point. Thus by avoiding the char type, we force the implementers of the control system that chars are immediately converted at the device server level or even below. The purpose, why char is not supported, becomes now obvious: the property type should match the device property and not any low-level I/O protocol such as a serial link. If it is unavoidable to send single characters to a device then they can sent either via an enum property and then converted at the server level into strings or via a dedicated command with a parameter of type char.

We see that the concept of properties is quite complex. It is again only natural, that properties are modeled through objects and not as simple values. The value of a property is accessed or changed via the methods get() and set(). If the property is read-only, then it only has the method get(). The minimal allowed value is obtained via the method minVal(), etc.

In first approximation, the properties of different type differ only in the type of the value (RWdouble.get() returns a double, ROpattern.get() returns an unsigned int), but are the same otherwise, e.g. all properties have a method description() which returns a string with the description of the property – the property current of the device power supply could have the description "PS_Q3A_SR/current".

Read-only and read-write properties of the same type have actually more differences, although the value types are the same. Still it makes sense to define a common superclass. As example the superclass Pdouble and its subclasses RWdouble and ROdouble, are shown in table 3. Some methods that use concepts introduced later in section 3.3 have been omitted for clarity.

Table 3: The IDL definitions for double properties

```
interface Pdouble {
       void get(CBdouble cb);//via callback
       double getSynchronous(...);
       long getHistory(...);
       ... // monitors, see section 3.3
       double defaultValue();
       double graphMin();
       double graphMax();
       double minStep();
       unsigned int resolution();
       string description();
       string format();
       string units();
}
```

```
interface ROdouble : Pdouble {
       ... // alarms, see section 3.3
       double alarmLowOn();
       double alarmLowOff();
       double alarmHighOff();
       double alarmHighOn();
}
```

```
interface RWdouble : Pdouble {
       void set(double value, CBvoid cb) ;
       oneway void setNonBlocking(double v)
       Completion setSynchronous(double v);
       void increment(...);
       void decrement(...);
       double minValue();
       double maxValue();
}
```

Once the properties have been defined, it is rather straightforward to define the individual devices. Table 4 shows an example of a magnet power supply: it has three properties and three explicit commands. Some basic functionality such as device name, position, etc. is inherited from a superclass called Device, which can keep also system-related data such as security. For brevity's sake, the Device class is not discussed here.

Table 4: The IDL description of a power supply:

```
interface PowerSupply : Device {
       // properties
       readonly attribute RWdouble current;
       readonly attribute RODouble readbck;
       readonly attribute ROpattern status;
       // commands
       void on(CBvoid);
       void off(CBvoid);
       void reset(CBvoid);
}
```

A simple program that switches the power supply on, checks for status errors and sets a current is demonstrated in table 5, where the CORBA error handling has been omitted for clarity.

Table 5: A simple program using CORBA

```
ps.on(new CBvoid()) ;
... // wait for callback to return;
if (ps.status().getSynchronous() == 0)
     ps.current().setNonBlocking(100.0);
else printf ("error in switching on\n");
```

To conclude the discussion on the object model, let's just demonstrate how we use inheritance to model a power supply, which has additional functionality to the simple one that we have used in our example. Consider a power supply, which can ramp according to a pre-loaded ramping curve. The corresponding interface definition is shown in table 6.

Table 6: Using inheritance with IDL

```
RampedPS : PowerSupply {
     void loadCurve(sequence double vals,
                    sequence int Nsteps)
     void start();
     void stop();
}
```

The basic functionality is directly inherited from the base power supply object.

3.2 The Application Programmer's Interface

All the IDL objects that have been defined in the previous section together form the application programmer's interface (API), because all of the objects talks to the device server. As there are many classes, it is considered a "wide" interface in contrast to the "narrow" one, which is more common in control systems:

```
ctrlObj.remote(device,"msg",dataIn,dataOut)
```

where the object named `ctrlObj` keeps a few methods that encapsulate all communication with the server. Here, `device` represents a generic device (in C/C++ systems, it is usually a pointer to a structure describing the device) and the parameters `dataIn` and `dataOut` are some generic containers that keep any type of data (in C/C++ a pointer of type void). The string `"msg"` is a command that the device server understands – unless the programmer has mistyped it. Based on the meaning of the command string, the device server interprets the input data from the `dataIn` container and packs data into the `dataOut` container.

A significant advantage of the wide interface is, that the explicit method call on objects is safer, as many errors are discovered at compile-time by the compiler automatically. Table 7 shows some examples of successful error detection, none of which would be detected in the case of the narrow interface.

Table 7: An example of error detection in the wide interface.

```
// first definitions of interfaces
interface A {
       void command(typeX x) ;
}
interface B {
       typeZ request(typeY y)
}
// second variable assignment
typeX x;
typeY y;
typeZ z;
// The compiler generates errors on:
A.request(y);     // command does not exist
z = B.request(x); // invalid parameter type
x = B.request(y); // assigned type mismatch
```

It may be rightly argued that a narrow interface is compact and generic. While the API is compact, the programmer must learn or look up in the device manuals all device types and their corresponding commands. In the wide interface, the API is itself the device manual. A simple look at the object definitions provides all insight of device commands. By generic, it is meant that the same application can control different device, because the user determines the devices and commands at run-time. However, CORBA also has a way to create command invocations at run-time. The dynamic invocation interface (DII) of CORBA allows to dynamically create an request object, even when the interfaces are not known and the interface libraries not linked to the application. The procedure is actually very similar to the use of the narrow interface. In addition, CORBA keeps a central repository of interfaces, which can be queried during run-time to discover available interfaces.

Adding it all up and considering the advantages of object oriented concepts such as inheritance, it is clear that the wide interface is natural for CORBA. Each device becomes a named CORBA object. All devices are arranged by interface type, so equal devices are grouped through the interface. Each device interface inherits from a basic Device, which defines basic data such as name, position, security/access, etc.

Advanced CORBA features and services further empower our concept of the wide interface. CORBA's reflection or introspection, respectively, allows run-time

discovery of interfaces and methods. It is possible and straightforward to find all interfaces that are supported by running device servers in the LAN. It is equally straightforward to find first all devices of a given interface, and then find all methods of a given device. The clean structure of objects in the ACI makes the interpretation easy.

We have written an application that uses all the above mentioned features and allows the operator to invoke any command on any device through a series of lists and menus. This application, called Object Explorer is completely generic through use of CORBA's dynamic invocation interface. Any future device that will be modelled according to the ACI will be discovered and controlled the moment its device server exports it to the network.

3.3 The Communication Model

Preferably, the communication between the object and the client program is asynchronous in order to benefit from the advantage of distributed processes running independently. In addition, the client's graphical user interface remains responsive even when a remote task takes long to execute.

We could have used the CORBA event service, but there is a major disadvantage to it. A CORBA event must be generic and can not be related to a given type. Therefore the Any type is used by CORBA to pack event data. Such an approach would thwart our efforts to discover all typing errors at compile time.

In order to retain strong type checking we have defined our own callback classes for each property type. So whenever a request is sent from the client to the object, the clients passes a callback object as a parameter of the requesting method. When the result of the request is known, either a return value or just a confirmation of the action, the object on the server side invokes a method of the callback object. To obtain a value of type <type>, the following callback is used:

```
interface CB<type> {
       oneway void execute (<type> value,
Completion c);
}
```

The completion object is composed of a POSIX-like time stamp and two short integers representing the type/code of a possible error (zero stands for no errors).

Asynchronous communications provide also the means for monitors: the client registers once with the object that it wants to receive callbacks at regular time intervals or when the value of a parameter changes above a threshold. There is no need for the client to regularly poll the object for values. Callbacks are even more efficient for alarms. A callback is invoked only when alarm conditions occur or disappear.

There are two types of monitors: alarms and property monitors. Alarms are represented by the SimpleMonitor interface. A simple monitor can only be created and later destroyed. Property monitors, represented by the Monitor interface, can be customised for different repetition rates and/or to trigger callbacks whenever a value changes by a certain amount. This amount, however, can not be defined by the user, but is pre-set in the configuration data of the device. It is also defined in raw binary values and not in engineering units, as it can not be subject to non-linear conversions.

4. CONCLUSIONS

The presented accelerator control interface (ACI) is successfully being implemented for the control system of ANKA [5], a 2.5 GeV synchrotron radiation light source being built in Karlsruhe, Germany.

The interfaces for most of the devices (power supplies, vacuum components, RF components) have been defined and the corresponding device servers written. As it turned out that an enumerated type would be needed only for two cases, it was not implemented and the only two property types used for ANKA are double and pattern.

The device servers have been implemented using C++ under Windows NT 4.0. They take data from the LonWorks fieldbus, which is employed at ANKA. However, the interface to the fieldbus has been kept small so in principle the device servers could be attached to an EPICS-based system with little effort.

On the client side, the CORBA objects are wrapped into JavaBeans [6], which are then connected with commercial data-manipulation and visualisation Beans using visual tools or programmatically. As the CORBA objects and hence the wrapper-Beans are generic models of controlled data, they can be used at any other control system. The Java applications are based on those objects only and can thus be run without change on any other accelerator, which implements servers that export ACI devices.

5. REFERENCES

[1] For a complete description of ACI see http://kgb.ijs.si/Clanki/ACI_draft_3.html

[2] M.Plesko, The CORBA IDL Interface for Accelerator Control, Proc. EPAC98, Stockholm, June 1998

[3] All CORBA related information can be found on the web pages of the Object Management Group and the links therein: www.omg.org.

[4] A brief introduction to CORBA concepts is at http://www.infosys.tuwien.ac.at/Research/Corba/OMG/arch2.htm#446864

[5] M. Plesko et al, A Control System Based on Web, Java, CORBA and Fieldbus Technologies, PCaPAC99 workshop, Tsukuba, January 1999.

[6] G. Tkacik et al., Java Beans of Accelerator Devices for Rapid Application Development, PCaPAC99 workshop, Tsukuba, January 1999.