THE CERN PS/SL CONTROLS JAVA APPLICATION PROGRAMMING INTERFACE

F. Di Maio, P. Charrue, J. Cuperus, I. Deloose, K. Kostro, M. Vanden Eynden, CERN, Geneva, Switzerland

W. Watson, TJNAF, Newport News, USA

Abstract

The PS/SL Convergence Project was launched in March 1998. Its objective is to deliver a common controls infrastructure for the CERN accelerators by year 2001. In the framework of this convergence activity, a project was launched to develop a Java Application Programming Interface (API) between programs written in the Java language and the PS and SL accelerator equipment. This Java API was specified and developed in collaboration with TJNAF. It is based on the Java CDEV [1] package that has been extended in order to end up with a CERN/TJNAF common product.

It implements a detailed model composed of devices organised in named classes that provide a property-based interface. It supports data subscription and introspection facilities.

The device model is presented and the capabilities of the API are described with syntax examples. The software architecture is also described.

1 THE JAVA API PROJECT

The purpose of the Java API project is to develop a Java package or packages that will provide an interface between controls applications written in the Java language and the CERN accelerators operated by the PS or SL divisions. One aspect of this work is to develop a language-independent object-oriented model of the accelerator devices. By moving to this new language in collaboration, the PS and SL controls groups expect to eliminate duplication of effort and also to produce a better product in the process [2].

This project is a collaboration between CERN and TJNAF, W.Watson being a member of this project since its foundation. At the end of the specification phase, the decision was made to base the implementation on a Java version of the CDEV library provided by TJNAF [3]. The Java API in use at CERN is now based on a new version of the Java CDEV library provided by TJNAF and the two laboratories maintain a collaboration on this product.

2 THE DEVICE/PROPERTY MODEL

The Java API depends on a device-oriented view of the control system. In this view the system consists of named **devices.** A device may represent a physical device in the control system (such as a magnet or a beam position monitor), but it may also represent an abstraction of a control entity (e.g. a ring with associated tune and orbit measurements). Conceptually, devices have **device properties**, which constitute the state of the device. By **getting** the value of a device property the device state can be read. Accordingly, a device can be controlled by **setting** one of its properties with the required value. For instance a magnet may have a "current" property. The getting or setting of this property respectively delivers the actual value of the magnet current or brings the current to the required level. Although devised independently, this model is very similar to the Java beans model with its get/set attribute accessors.

Devices are organised in **device classes** that describe the device interface namely the available properties and their type. Devices of the same class have the same set of properties. Device classes constitute the meta-description of devices. An example is given in Fig. 1.

Properties may have **characteristics** such as units, resolution, etc. A property's characteristic is either a single value, common to all members of the same device class, or a reference to another property, which holds the value. Characteristics may be used to describe the purpose of a property or to indicate relations between properties.

Magnet
status: int command: int current: double currentAcq: double

Figure 1: Sample Properties for a Magnet device

3 THE CAPABILITIES

Here the essential capabilities of the Java API are described. It should be kept in mind that some of these capabilities such as subscription or time stamping rely on the underlying control system capabilities.

3.1 I/O Methods

The I/O methods of the device objects implement the following capabilities:
- get or set a property (*get, set*),
- activate or stop the monitoring of a property (*monitorOn, monitorOff*),

- execute a named operation accepting input parameters and returning results (*send*)
- get or set the reference value of a property (*getReference, setReference*).

The CDEV's **Data** and **DataEntry** objects are used to exchange data. A *Data* object is a container for one or many *DataEntry* objects that encapsulate data and implement conversion methods from the internal data representation to any supported data type (numeric values, strings and arrays). *DataEntry* objects also have a tag (*String*) so that a *Data* object can be composed of many *DataEntry* objects with different tags.

When an error is detected in the interface library or is reported by the underlying control system, a **DeviceError** object is returned. A *DeviceError* object encapsulates a message, a category and a numeric code. It is also a *java.lang.Exception* object that can be thrown.

3.2 Synchronous and Asynchronous Methods

Synchronous methods block the user thread until the I/O is completed. They take *Data* objects as parameters and returns *DeviceError* objet on failure. The following code fragment is an example of a synchronous *get* invocation.

```
Device dev = new Device("BTP.DVT10");
Data = new Data();
DeviceError err;
err = dev.get("CurrentAcq", data);
if (err != null)  throw err; // simplest usage
double acqCurrent = data.getDoubleValue();
```

Asynchronous methods do not block the user thread and a separate thread will execute a user-specified method when the I/O is completed. Asynchronous methods are based on the Java event model and are essential to exploit the power of Java. They require a class implementing the **DeviceListener** interface as a parameter. The **DeviceListener** interface defines a *deviceChanged* method, which allows receiving **DeviceEvent** objects. A *DeviceEvent* object provides the information about the completion of an I/O operation including the *Data* or *DeviceError*, the *Device* and all the I/O parameters.

The following code fragment illustrates the implementation of a class to receive a magnet's current.

```
class MagCurrentHandler implements DeviceListener {
void deviceChanged (DeviceEvent event)  {
  String devName = event.getDevice.getName();
  double acq_current =
    event.getValue().getDoubleValue();
 ...
```

3.3 Timing System Support

Many parts of the CERN accelerators are controlled by a timing system, which sends events on a dedicated network as well as information about the process, like the cycle description for a cyclical accelerator.

As a result, a device is usually linked to a **timing system**. Consequently, I/O operations may require some timing system specific parameters such as a **cycle type** or an **event identifier**. It must also be possible to have the execution of an I/O operation synchronised with a timing systems event.

For this purpose, a *Device* object can be associated with a **DeviceContext** object that can be used to specify such parameters. The following code fragment illustrates how to specify a timing-event as well as a cycle-type that will control the monitoring of a device.

```
DeviceContext ctxt = new DeviceContext();
ctxt.setCycleType("CPS.PARTY.PROTON");
ctxt.setTimingEvent ("END_CYCLE");
dev.setContext(ctxt);
```

3.4 Acquisitions

In addition to values, acquisitions can include a **time-stamp**, a Java double (64 bits float) giving the acquisition time in seconds since the usual Posix origin, and a **cycle-stamp**, a Java long integer (64 bits) used to identify every executed machine cycle.

All these data are enclosed in the same *Data* object. In addition to the *DataEntry* object(s) returning the value, there can be *DataEntry* objects tagged "timeStamp" and "cycleStamp".

3.5 Device and Device Class Discovery

In most cases the application program knows which properties are supported by a device class, what is the native type of the property, etc. But there is a class of programs like generic displays and browsers, which need device class information to construct their I/O requests. This information is made available through the directory service, which also fulfils the role of naming service and device querying engine.

Information about devices includes their parameters, the timing system they belong to and their relation to other devices. Information about the device classes includes properties and their attributes and characteristics. This information is provided by means of dedicated classes, such as *DeviceData*, *DeviceClass* and *DeviceProperty* which are described in [4].

4 THE SOFTWARE ARCHITECTURE

4.1 The core CDEV part

Many CDEV concepts map directly the CERN requirements: the device concept, asynchronous I/O, connectivity to different control systems and support for publish/subscribe. As a result, the project team decided to use the CDEV's Java development for the CERN Java API implementation.

A new version of the CDEV's Java implementation was designed and has been provided by TJNAF. It offers a better connectivity to local control systems, an extended support for property based I/O and it allows extension with a directory service. The public interface has also been reviewed in order, for instance, to introduce event listeners and error objects that have been mentioned above.

The software architecture is illustrated in Fig.2. The core CDEV part includes the interfaces that define the capabilities, which are implemented by the directory service and by the I/O services.
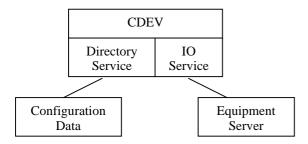

Figure 2: Software Architecture

4.2 The directory service

The directory service, described in detail in [4] is an addition to the original CDEV implementation, which adds all capabilities that rely on configuration data, such as the discovery of devices and device classes. A directory service is used at CERN but it is still possible to use CDEV without such a service.

The CERN I/O services also use the directory service to set up their device calls. This includes getting the network parameters of the devices and checking whether the properties are implemented for the device and what their attributes are.

4.3 The I/O services

An I/O service implements the connection with the underlying control system. There can be many variants and two distinct devices may be served by two distinct I/O services.

There are two possible architectures for implementing an I/O service. In a 2-tier architecture, the Java process has a direct connection to the front-end server tasks. In a 3-tier architecture, the Java process connects to a middle tier server that communicates with local equipment servers.

The 2-tier architecture is simpler to implement and is the more efficient one for synchronous I/O (blocking calls). The first CERN implementations of the API are based on this architecture: a JNI connection to C/C++ libraries is used for communicating with the PS accelerator devices while a Java RPC package (SUN protocol) is used for the SL accelerator devices. The first solution is not "pure Java" (native libraries are required)

but did not require new server tasks in the front-end computers. It is also worth mentioning that JNI connections to C libraries that are not thread-safe impose the usage of semaphores that reduce parallelism.

The 3-tier architecture shall also be used in the future. This comes from two major constraints: a) it became clear that the Java code running in the client's virtual machine must be reduced in order to cope with performance limitation on some platforms, b) the equipment server tasks are running on real-time systems where resources are limited. The current CDEV Java distribution includes an I/O service that can communicate with EPICS and all other CDEV C++ supported systems in a 3-tier architecture. At CERN, a dedicated PS/SL convergence sub-project is working on a new middleware architecture. For now, prototypes have been made using home-made protocols, RMI or CORBA.

5 CONCLUSIONS

The primary goal of the Java API project was to deliver a common API for PS and SL controls as a major component of a common architecture. The results of this project are 1) a common model, 2) a common API based on CDEV, 3) a common configuration management facility. These products are a good base for a common software architecture and will certainly evolve.

The collaboration with TJNAF is very valuable in this process and up to now a strong collaboration is maintained. CERN and TJNAF use the same CDEV version. Each site had to introduce local extensions but by continuous communications and regular merges, we stayed on the same track.

Some Java applications that use this API have already been delivered to operators. In the process of producing Java application, we have now two aspects to cover. The first one is to have an enhanced communication infrastructure that will allow reducing the weight of the client part. The second one is to provide programmers with high-level components that will facilitate the production of application software. There is hope for sharing such components with TJNAF and other CDEV users.

6 REFERENCES

[1] Jie Chen, Graham Heyes, Walt Akers, Danjin Wu and William Watson III, "CDEV: An Object-Oriented Class Library for Developing Device Control Applications", Proceedings of ICALEPCS 95, Chicago, U.S.A. October29-Nov. 3, 1995 , p 97.

[2] http://hpslweb.cern.ch/pssl/projects/javapi/javapi.html CERN PS/SL Java API Project home page.

[3] http://www.jlab.org/cdev/ - CDEV home page

[4] J. Cuperus, P. Charrue, F. Di Maio, K. Kostro and W. Watson, "A Directory Service for the CERN PS/SL Java Programming Interface", this Conference.