USING CDEV AS MIDDLEWARE IN VACUUM EQUIPMENT CONTROLS

I.Laugier, N.N.Trofimov, CERN, Geneva, Switzerland

Abstract

Recent experience when developing vacuum control applications at CERN made evident that a significant amount of work was induced by the variety of Application Programming Interfaces (API) in different parts of the huge accelerator vacuum control system. A natural solution to the problem is to provide an isolation layer of software that will effectively decouple applications from any dependencies on the specific interfaces by presenting its own unique API to application programmers. The CDEV [1] package developed at TJNAF has been used as an implementation framework for this layer of software. The implementation follows a 3-tier architecture where vacuum equipment servers, based on the CDEV "generic server engine", act as intermediaries between applications and subsystem specific controls. The paper describes in detail this client-server architecture and presents experience with using CDEV in the CERN accelerator control environment.

1 INTRODUCTION

For the time being, all three main accelerator control systems at CERN (LEP, SPS and PS) use significantly different protocols and interfaces for the vacuum equipment access. As a consequence, in each system application programmers have to write completely different code to read, for example, a pressure value from a vacuum gauge.

The software presented in this paper aims at hiding these differences to the application programmer. The basic idea is that, despite the diversity of existing low level equipment controls, at the application software level devices of the same kind can be represented by a common control model reflecting their prime operational purpose rather then specific implementations.

The Common Control Model (CCM) for the vacuum equipment [2] presents a vacuum system to the applications as a collection of logical devices. Each logical device contains a number of functional components corresponding to physical variables in the underlying vacuum equipment and belongs to a device class. Each functional component is uniquely identified by a name within its device. All devices belonging to the same class have the same nomenclature of functional components.

Several classes of the functional components have been defined in the CCM; each class provides a standard interface to a certain type of physical device variables and, depending on the type (analog or discrete, input or output), specifies a number of attributes which applications can observe and, in some cases, modify. For example, components of the Measurement class have *value*, *validity*, *timestamp*, *units*, *minValue*, *maxValue*, *resolution* and *format* attributes and serve to model analog input channels.

This model can be implemented on top of the existing control facilities and, along with a common API, serve as a basis for portable application software.

2 OVERVIEW

The prototype implementation of the model on LEP and PS accelerators at CERN is based on the CDEV class library developed at TJNAF [1]. Basic concepts of the CDEV software are very close to the control model, so the mapping is rather simple and straightforward. Functional components map to CDEV device attributes, and attributes of the functional components map to CDEV properties.

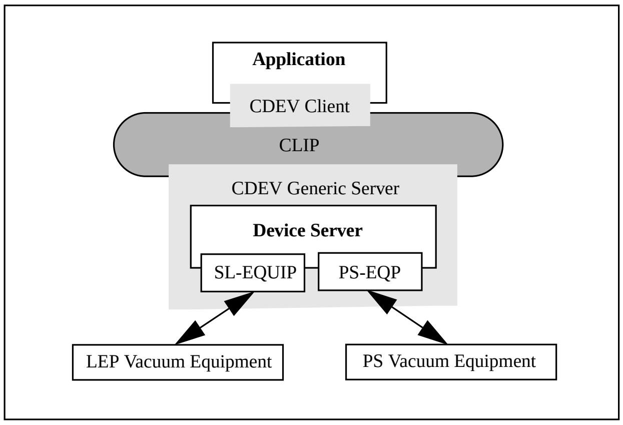

Figure 1: Device server.

Applications linked to the CDEV client library communicate with Device Servers using the CDEV networking protocol (CLIP, Figure 1). The Device Servers are implemented within the CDEV Generic Server framework and their primary function is to map logical devices to the underlying physical equipment. For each logical device the servers provide the “stub” functions which link the device to physical I/O points in the equipment. The physical I/O is performed using the PS [3] and LEP [4] specific equipment access interfaces and protocols.

Since the physical I/O is relatively slow, the servers maintain a cache for the most recent data read from the equipment. Applications read device attribute values from the cache; the cache data is periodically updated so that it always reflects with a required precision current state of the physical equipment attached to the server. Upon each cache update, the applications that subscribed to (monitorOn) values of device attributes can be notified on changes if the data received from equipment is different from the cached data.

3 CONFIGURATION DATA

One of the main goals in the Device Server design was to make the server software, as much as possible, data driven. In particular, it shall be possible to introduce a new device class in the control model or a new device in the system without recompilation, just by entering the device or the class descriptor into a configuration database. In the prototype implementation, the general accelerator (Oracle) database was used to store the configuration data.

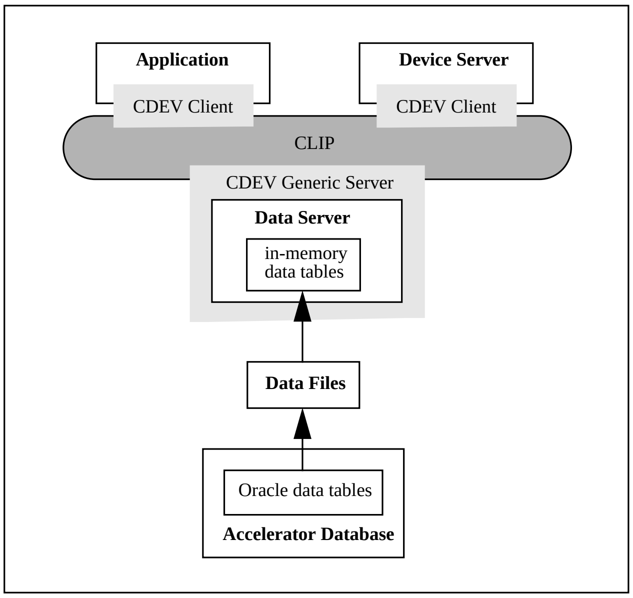

Figure 2: Data Server.

Since access to the accelerator database is relatively slow and not always guaranteed, the two level data extraction scheme is used (Figure 2). Information from Oracle data tables is dumped into files that are read at startup by the Data Server which then acts as an actual data source for the Device Servers and applications.

The Data Server is also implemented using the CDEV Generic Server framework and is seen by its clients as a special "well known" database device that responds to the *select*, *query* and *get* messages. The first two messages clients can use to locate a required descriptor in the database. When the descriptor is found, its contents can be obtained from the database using the *get* message.

With this approach, we practically do not use the cdevDirectory device and DDL files. The DDL file in use only describes interfaces to the CDEV name server and the Data Server. All directory services are provided by the database device.

4 DEVICE SERVER SOFTWARE

The Device Server software is written in C++. Classes shown in Figure 3 constitute the server framework; altogether, they allow to create, according to the database description, the software objects which represent logical devices and “export” them over the network to client applications.
.

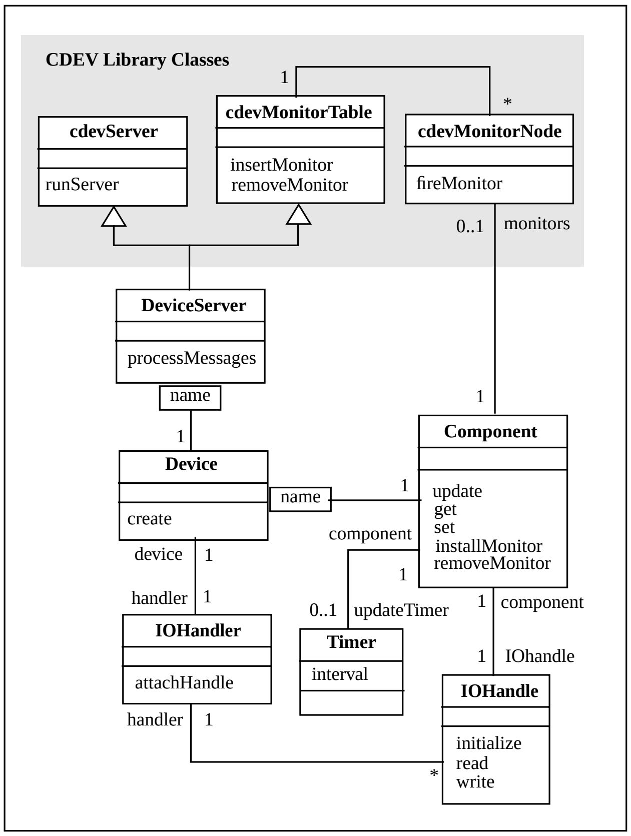

Figure 3: Object model for the Device Server.

The server main function simply creates a DeviceServer object giving the server name as an argument to the constructor.

The constructor locates the server description in the database using the name as a key and extracts from the database a list of all devices attached to the server. It then iterates through the list calling for each device the static *create* method of the Device class.

The *create* method in turn extracts from the database a class description for the specified device and for each functional component defined in this device class creates a corresponding Component object. At this point, it also creates a Timer and an IOHandle for the Component.

The Timer will periodically activate the Component *update* method which refreshes the Component attribute values with new data obtained from the equipment. If the values change during the update, the method can fire monitors installed on the Component and hence report the changes to all interested clients. The Timer *interval* value is set from the database according to the sampling rate required to follow changes of the corresponding physical variable.

The physical I/O during the Component update procedure is performed using the *read* method of the associated IOHandle object. The IOHandle is an abstract class which just defines a common interface to the I/O operations. A concrete implementation must be provided for each specific I/O connection.

Logical devices, even of the same class, can map to many different hardware "boxes" connected to the control system in different ways. The IOHandler is a collection of the IOHandles required to connect a logical device to some type of the physical equipment via certain I/O interface. For each logical device, the configuration database specifies the name of its IOHandler. The *create* method of the Device class extracts this information from the database and dynamically attaches appropriate IOHandles to the device Components.

Component attributes can be dynamic if their values change during the operation of the vacuum device (the pressure value, for example) or static otherwise. The static attributes, such as units of measurement, typically get their values from the configuration database during the Component creation. In some cases, they can be initialized from the device hardware using the *init* method of the IOHandle class.

When the server is populated with Device objects, the main function invokes the *runServer* method of the cdevServer class and the server starts to accept client connections and process incoming messages. The communications are handled by the CDEV library classes; incoming messages from connected clients are buffered in the input queue and then dispatched to the *processMessages* method of the DeviceServer class.

The method first locates the Device and the Component to which a message is targeted and then, depending on the message type (*get*, *set*, *monitorOn*, *monitorOff*), invokes one of the Component accessor functions (*get*, *set*, *installMonitor*, *removeMonitor*).

The *get* function packs values of the Component attributes specified in the message context into a cdevData object and send this object back to the client in the response message. The *set* function assigns new values to the Component attributes specified in the inbound cdevData object and then downloads the values to the equipment using the *write* method of the IOHandle class. The *install*/*removeMonitor* functions call corresponding methods of the cdevMonitorTable class to register/unregister a monitor on the Component attributes specified in the message context.

The message processing and the regular update of Component attribute values are performed in different (POSIX) threads to allow the server to handle client requests during relatively long I/O operations. In each Component, a mutex prevents simultaneous access to the Component data from the message handling and update threads.

5 RESULTS

Several device servers providing access to more then a thousand vacuum devices of the PS complex are in operation for over a year now. The device server for the vacuum equipment of the new Antiproton Decelerator has been successfully used in the commissioning of the machine [5]. The server software has proved to be reliable and showed performance adequate for vacuum control applications.

6 REFERENCES

[1] Jie Chen et al., 'CDEV: An Object-Oriented Class Library for Developing Device Control Applications', ICALEPCS'95, Chicago, 1996.

[2] I.Laugier, P.M.Strubin, N.N.Trofimov, 'A Common Control Model for Vacuum Equipment at CERN', ICALEPCS'99, Trieste, 1999.

[3] F. Di Maio, A.Risso, 'The CERN-PS Equipment Access Library', CERN PS/CO Note 93-87, 1994.

[4] P.Charrue et al., 'The Equipment Access Software for a Distributed UNIX-Based Accelerator Control System', ICALEPCS'93, Berlin, 1993.

[5] P.M.Strubin et al., 'First Experience with Control and Operational Models for Vacuum Equipment in the AD Decelerator', PAC'99, New York City, 1999.