DOOCS: a Distributed Object Oriented Control System on PC's and Workstations

G. Grygiel, O.Hensler, K. Rehlich, DESY

Abstract

DOOCS is a distributed control system that was developed for HERA and TTF applications. It is an object oriented system design from the device server level up to the operator console. Class libraries were developed as building blocks for device servers, communication objects and display components. The whole system is written in the programming language C++ and runs mainly on Solaris, SunOS and PC's with the LINUX operating system. In the mixed environment of PC (x86) and Sun workstation (SPARC) hardware the communication is realized by Remote Procedure Calls (RPC) which handle the conversion of data structures between the client and server processes. For the three different PC and workstation operating systems the programs are compiled from a single source. Other platforms may easily be added.

Introduction

The main design idea of DOOCS is to look in a device oriented manner at a control system, or speaking from a programmers point of view: every hardware device is an object. This leads to a concept of device servers which handle all properties of a particular device. These servers then create as many as needed device instances (locations) of its type.

In DOOCS terms, a device server is an independent program that completely controls a number of devices and provides the data to the network.A client is an independent program which receives that data and sends control messages to the servers.

It is a distributed system, because the device definition is done on the server side only and is transparent to the clients. Whenever a server is started, new device instances are created or new properties added, these changes are immediately available on the network; there is no central database to hold these items. Client programs may request an actual on-line list of all devices and properties by means of a query request.

Many different servers can run on the same hardware or these servers can run on different CPU's. Even the same type of device server with different locations are able to run on many CPU's. The names (IP-Address) for the clients are resolved by an Equipment Name Server (ENS), which again can run multiple times on various computers.

The ONC/RPCs (Open Network Computing/Remote Procedure Calls) are used for the network communication. These Remote Procedure Calls were developed by SUN and are available on every common computer platform. Many industrial software standards like Network File System (NFS) are based on this protocol. The RPC library converts all transferred data types into a computer architecture independent representation by means of the XDR technic. Therefore, data exchange between different computers is no problem.

In order to be independent from the Ethernet, we attached a harddisk to every server station. This allows booting and runing without the network. Because we have a harddisk, archiving of history data is done locally as well. And every device server keeps a local data base of its actual configuration and state in order to restart the system unchanged after a power failure. Since these two data bases are in a file on the local disk, a server does not depend on the network and network load is reduced for the archiving.

The communication separates completely the client programs from the device servers. All device specific information is held in the servers. A modification in the

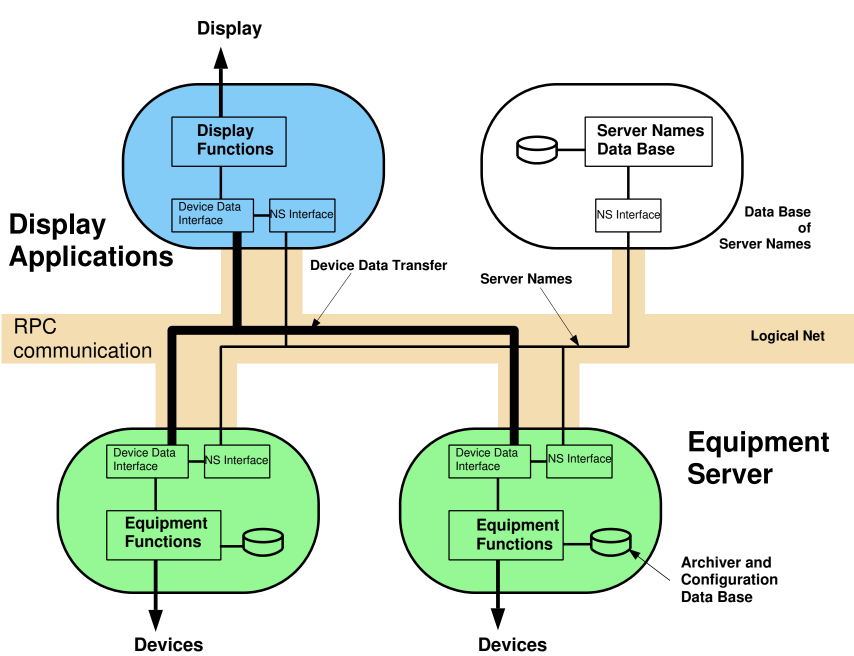

Fig.4: DOOCS Client-Server Model

layout of a server program has no impact on the client program because of the symbolic access of data.
The DOOCS client Application Programming Interface (API) is able to handle different network protocols in addition to the DOOCS-RPC protocol. So far, the EPICS Channel Access calls are supported as a second protocol within the API. A further protocol (the so called “Classic protocol” from Fermilab) is implemented in a server that operates as a gataway with additional archiving capabilities.

The Application Program Interface

The client part consists of a communication class, an address class and a data class as the client interface plus some internal classes (Fig.2). The data class library has a lot of methods to access, change and convert data and control informations. The transferred data also contain type, error and length information. A data item can be a simple integer, float, string or a complex structure of archived spectra or histories or error messages or a list of available services. So far 16 data types are defined. New types can easily be added by defining a C structure of the data and adding new methods to the data class. The conversion of a new structure into different number representations of different CPU architecture is automatically done by the rpcgen compiler which generates the eXternal Data Representation (XDR) routines.
With the address class the client program specifies a facility, a location, a device and a property of a device. The 4 parts are ASCII names with up to 16 characters each; the property part can be up to 32 characters long. The addressing of the servers and the handling of the server links is done in the library. A server could be on any computer in the internet. Killing and restarting of services are handled in the communication libraries by automatic creation or recreation of links to the services. The implementation of our communication protocol also allows reading a value from all objects of a certain device server class with just one call. This feature reduces the network traffic a lot and gives a program the chance to read all error flags of one server, for instance, without knowing the actual names or number of objects. There is also provision in the protocol for generalized programs to get the names and properties of all objects in the net. The client communication interface has only 4 different calls : two to read data, one to set data and a third one to read the actual list of devices and properties. The two read calls implement a blocking synchronous read call and a monitor call.

Server Implementation

Device server processes are build from a modular library of C++ classes. An actual server consists of several entries of a device type at different locations in the system. It may contain different device types also. Every instance of a device defines a set of properties. These named properties are the access points on the communication network and are implemented as data objects.
The server library defines a basic class for device servers. This basic class provides the common communication objects like the name, the error status and on/offline information. An actual device server inherits this and adds device specific code and data objects. Data ob-

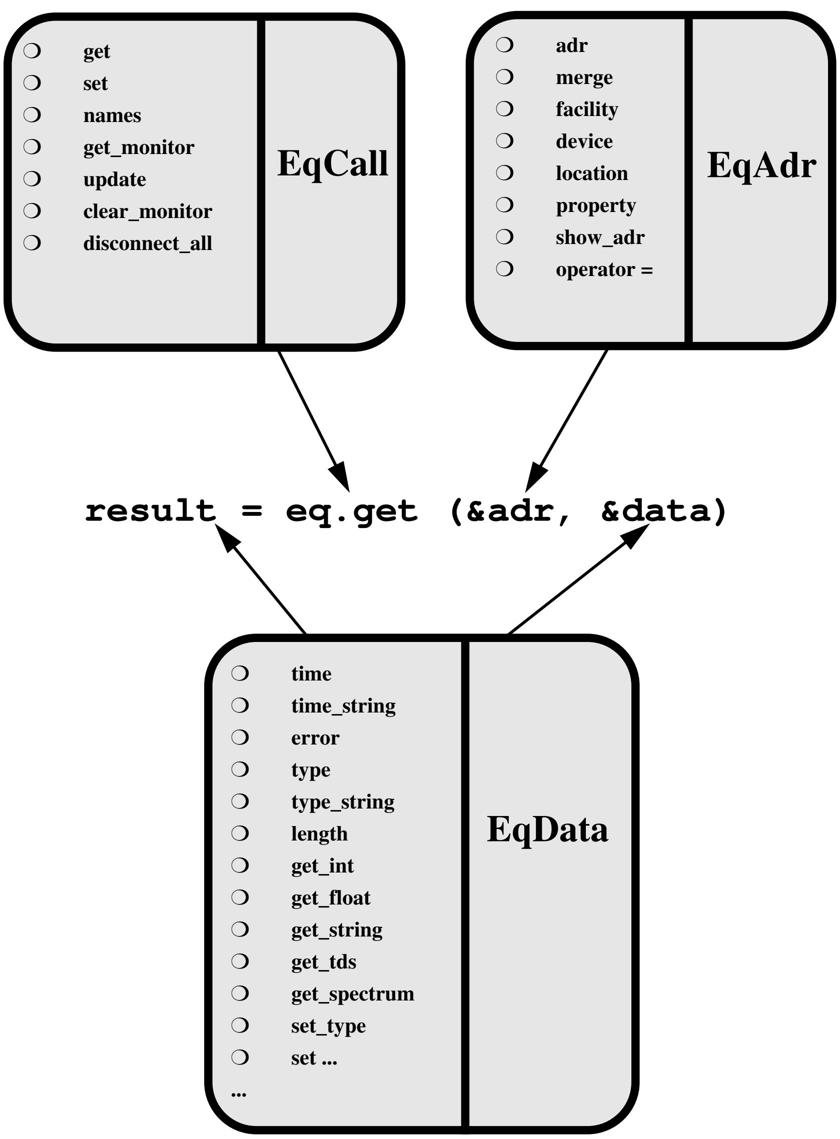

Fig.2: Classes used in the Application Program Interface

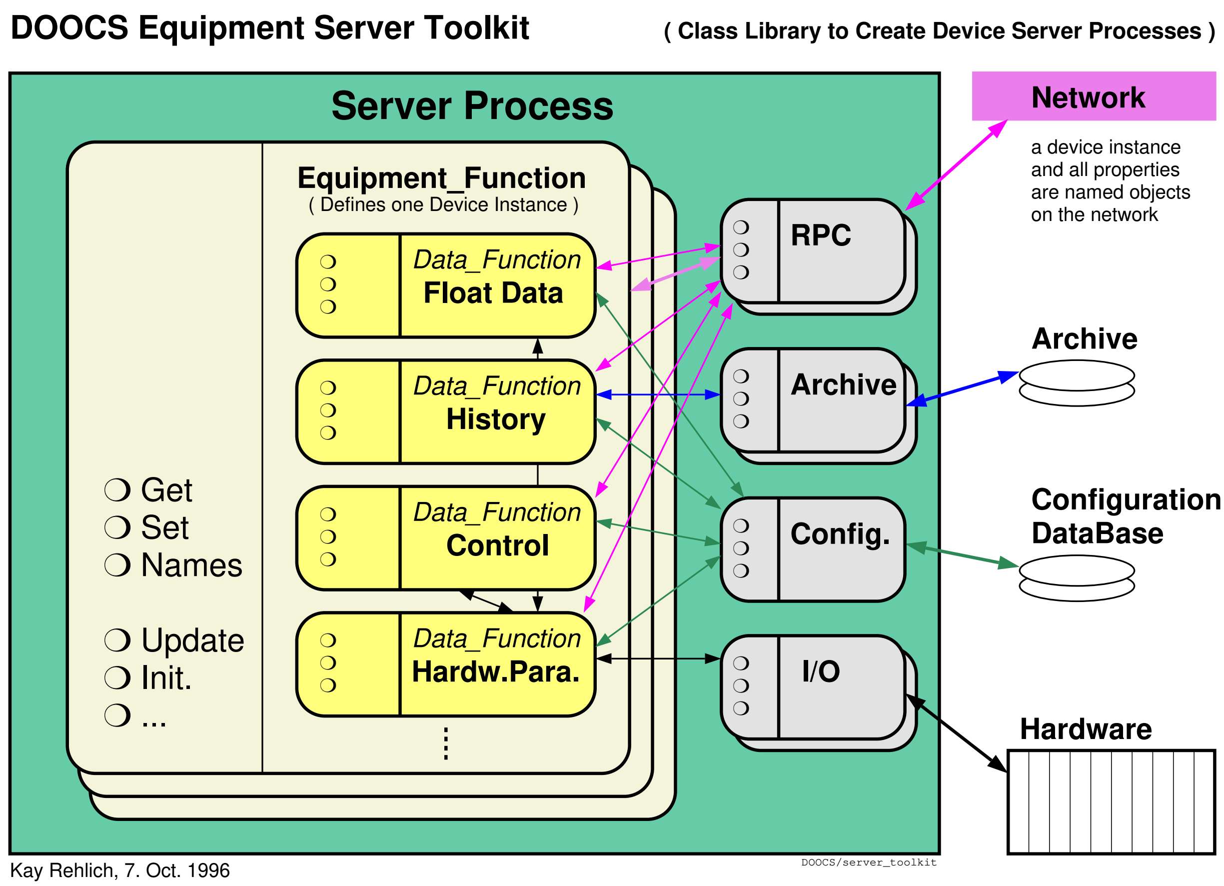

Fig.3: The Equipment Server Toolkit

jects are also part of the library and are defined for status words and bits, correction polynomials, digital filters, errors, float values, archiver and spectra readings and so on. By declaring a data object in a device server it is automatically inserted into the list of named device properties and accessible on the network.
The server library is a collection of various classes to provide the network access and the access to the hardware like VME, Profibus and SEDAC. In addition there are classes to read/write the configuration file and to archive the data types on disk.
The Eq_fct class is the base class from which all device server instances are derived. It provides all the basic functionality that every device instance of each server should contain. One gets the programming interface for all needed methods of the derived device classes, because Eq_fct provides virtual functions like update(), init() which every device server has to supply.
It defines for all device instances a name string, which is seen from the client side as the location name. And it provides error handing for every device automatically.
The first class always derived from Eq_fct is the EqFctSvr-class, which is the first instance (location) of every server. This instance provides special properties like update-rate, last update, log-file name.

The D_fct class is the data interface from the device server to the network. A declaration of a member of the D_fct class is similar to a declaration of a float in C or C++. But, a float of the D_fct class is an object that is readable and setable from the network. In other words, the data function class D_fct makes local variables available to client programs. It internally handles all network stuff. D_fct also implements the functions to read/write the configuration file. If used for the definition of hardware addresses, the D_fct class provides the address parameters on the network and implements the methods to access the devices from a server routine. That is, a client may change the device address and the server uses the on-line updated address in further calls to the device. Often the get and set routines of D_fct are overloaded in actual servers to program special device operations.

Client Programs

This chapter gives a brief description of some commonly used client programs. All programs use the same shared library with the application program interface to access device data.

LabVIEW from National Instruments is used for applications which need to display and program device data. It is useful for measurements and all kind of sequencing programs. The programming in LabVIEW is done with the mouse by connecting graphic symbols. A Virtual In-

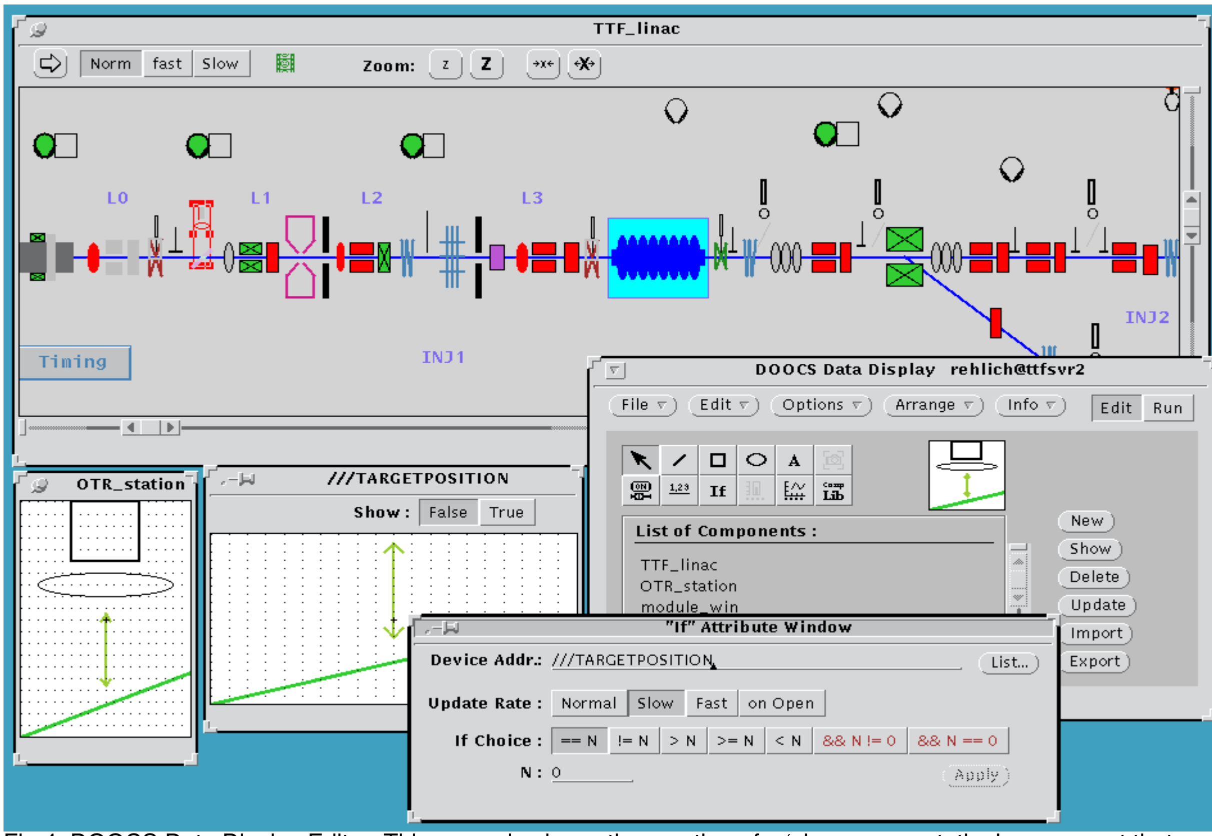

Fig.4: DOOCS Data Display Editor. This example shows the creation of a 'view screen station' component that may be used at several places in the linac display (top window) for different stations. The displayed position of the mirror is animated by the position reading from the device.

strument (vi) library implements the access to device data.

As a second operators interface we have a **DOOCS Data Display (Fig.4)** program (ddd) which is a graphical editor to create and run control panels. ddd allows the creation of component libraries in a hierarchical way. The synoptic displays are animated by the status of the devices; subwindows with detailed information or plots are activated by a single mouse click.

rpc_test is a generic program to display and modify all available device data. It reads the actual list of device servers from a name server and the available properties of a certain server from the process itself. Because the program gets all informations from the network, all parameters in the system can be read and written without any change in the rpc_test program.

plot_test is a further generic program to display all data from all devices which are plotable. Since it reads the names from the network, as rpc_test does, it allows showing all available data types with historical data or snapshots from scopes, for instance. This data may come from actual readings or from an archive.

save_and_restore is a tool to read and modify several device properties on one page. The tool is configured from files and stores device data in files.

xerror is a tool to display all errors of device servers including the logfiles of the devices. It continuously scans a list of device servers and displays any error. xerror is configured from a file.

doocsget and **doocsput** are command line interfaces to read or write device data.

Conclusions

All server and client programs run on PC's, Sun workstations and VME based SPARC CPU's with the LINUX/x86, SunOS 4.x or Solaris 2.4/2.5 operating systems. The programs are compiled from a single source by means of the GNU or Sun C++ compilers. A few 'ifdef' statements in the source files and slightly different 'make' files cope with the differences of the operating systems and allow the generation of the executables with the UNIX 'make' command. Sources and libraries are managed for a multi user development by the Concurrent Versions System (CVS). Since the hardware differences of PC's and SPARC processors (little-endian/big-endian number representations) are handled by the Remote Procedure Calls (RPC) and the eXternal Data Representation system, client and server programs running on different architectures can be mixed. Also the X-Window system helps to build a homogeneous environment based on the UNIX operating system.