DOOCS: a Distributed Object Oriented Control System

O.Hensler, K.Rehlich
Deutsches Elektronen-Synchrotron, DESY, Hamburg, Germany

Abstract

DOOCS is a distributed control system that was developed for the DESY accelerator HERA and mainly for the Tes-la Test Facility TTF. 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 server, communication objects and display compo-nents. 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. Through its multiprotocol client API DOOCS talks to all EPICS server. The TTF control system consists of contributions from Fermi-Lab, INFN Frascati and Milan, LAL, CAE Paris and DESY groups with DOOCS integrating the various different systems in order to access them from a single display program.

The server part usually runs on VME based SPARC boards to allow simple access to any kind of VME-hardware like fast ADCs and digital I/O. On the fieldbus side we support the DESY standard SEDAC and the German indus-trial standard PROFIBUS to get access to our distributed PLC environment supplied by SIEMENS.For displaying (client side) we can use the industrial package LabView or our own DOOCS Data Display (DDD) which permits creation of applications in the style of a usual drawing program.

1. INTRODUCTION

DOOCS is developed by K. Rehlich and O. Hensler with contributions from M. Bhnert, D. Hoppe, G. Grygiel, S. Goloborodko, P. Shevtsov. Some of the design ideas are based on the HERA proton vacuum control system that already used the object definition paradigm of devices. To be able to control all kinds of devices of an accelerator the whole system was redesigned and is now written in C++. The system is used for some part in the HERA vac-uum system and for TTF (TESLA Test Facility).

DOOCS programming is done on SUN SPARCstations under the Operating Systems SunOS and Solaris 2.x, and is ported so far to Linux/x86 with the GNU Compiler. The Communication is based on Remote Procedure Calls (ONC/RPCs) with the e**X**ternal **D**ata **R**epresentation (XDR) network format, which is a widely used industrial standard.

The device servers in the system are located on SPARCstations, VME-SPARC processors and PC's with the LINUX operating system. Most server processes are running on embedded SPARC CPU's in VME crates and are able to talk to various VME - cards via memory mapping. As fieldbus we support the DESY standard SEDAC and the German industrial standard Profibus. The Linux PC's operate with an analog/digital data acquisition system (Cobra) via an AT-bus.

DOOCS is build as a set of class libraries for device servers, the communication interface and the client programs. It provides some standard applications and an Application

Programming Interface (API) to create special client application programs. Device servers which need to read from other servers use the same API library. On top of the API we provide a Virtual Instrument library to allow LabVIEW programs to read and write all device data. The API allows multiprotocol accesses. So far EPICS calls are implemented. A rich set of standard functions is imple-mented in the API library.

2. DESIGN IDEA

The main design idea of DOOCS is to look in a device oriented manner to a control system or speaking from a pro-grammers point of view: every hardware device is an object. This leads to a concept of device servers which han-dle all properties of a particular device. These servers then create as many as needed device instances (locations) of its type. Because the design is already object orientated, the choice of C++ as programming language was obvious.In our terms a device server is an independent program that completely controls a number of devices and provides any kind of data to the network.A client is an independent program which receives that data and sends control mes-sages to the servers.

It is a distributed system, because the device definition is done on the server side only and is transparent to the cli-ents. 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.

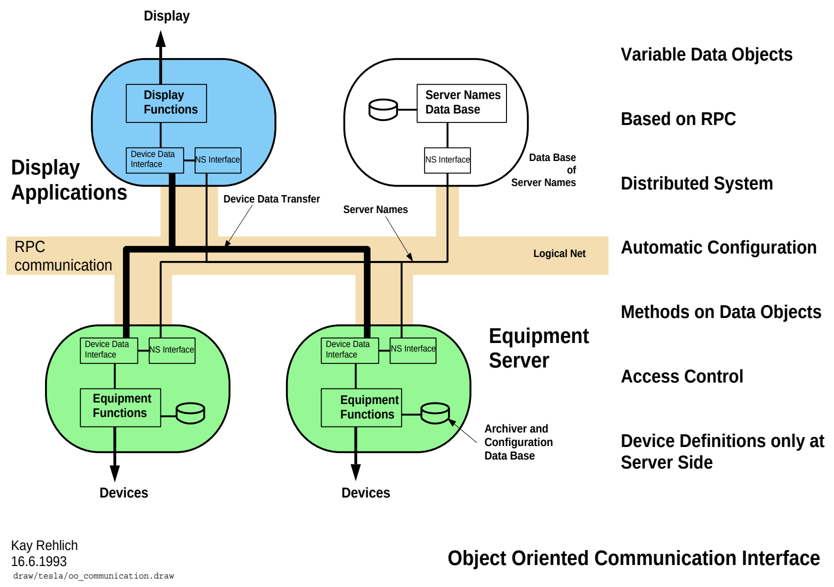
Many different servers can run on the same hardware or these server can run on different CPUs. Even the same type of device server with different locations are able to run on many CPUs. The names (IP-Address) for the clients are resolved by an Equipment Name Server (ENS), which again can run multiple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

Due of the very good experience at the HERA Proton vacuum control system, another obvious choice for us was to use the ONC/RPCs 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 inde-pendent 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 to boot and run without network. Because we have a harddisk, archiving of history data is done locally as well. This helps to reduce network load and allows to analyse problems that happened during network breakdown. So no history data will be lost.

In other words, a server provides the complete service for a device class without the need of a network conection. 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 this data base is a file on the local disk a server does not depend on the net-work.

The communication separates completely the client programs from the device servers. All device specific informa-tion is hold in the servers. A modification in the 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 addi-tion to the DOOCS-RPC protocol transparent to the user. So far the EPICS calls are also supported from within the API.

This drawing shows the possible three layers of DOOCS. On the bottom are the device servers which are con-nected to the hardware and providing the network access.

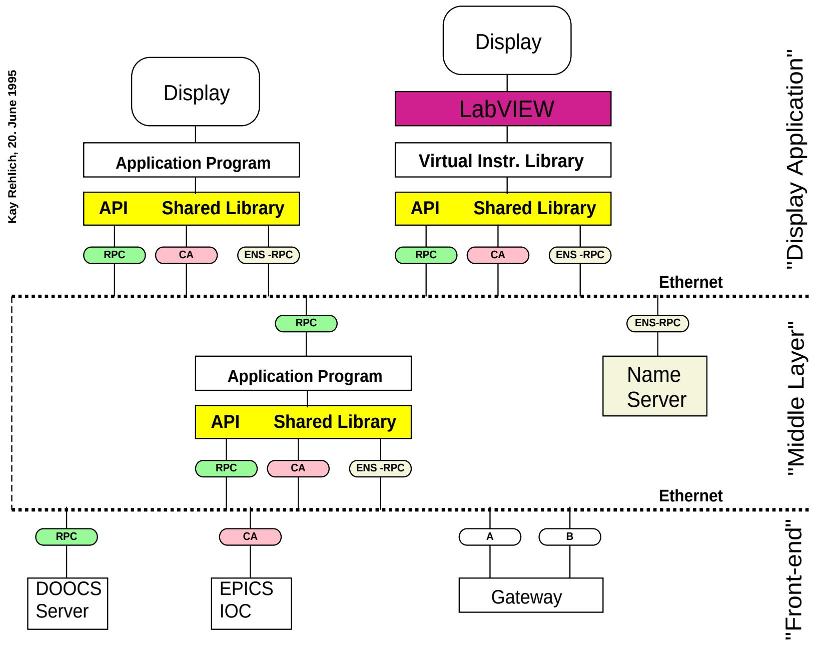
The middle layer of the control system is used for servers with no direct hardware connection. These servers may run simulations or calculations or combine different servers with hardware connection to compose higher functions on groups of devices. From a client point of view there is no difference in the communication to a server in the middle and front-end layer. All of them provide the information in an unique way. The Equip-ment Name Server (ENS) is another server in the middle layer. Several name servers with the same informa-tion are running on different CPU's for redundancy reasons. They provide the necessary information about the servers in the system to the clients.

On the display layer one can see two client applications displaying the data coming from the network. The right one uses Virtual Instruments (VI) designed on top of the DOOCS API to use the data in LabView. All cli-ent programs use the same API library.

Device servers normally run in front-end computers. They are build from a object oriented library as a toolbox to compose this servers. This library contains the basic class to create a server, all data types as network objects, functions to access hardware, configuration file I/O and archiving.

3. CLIENT INTERFACE

The client part consists of a communication class, an address class and a data class as the client interface plus some internal classes. The data class library has a lot of methods to access, change and convert data and con-trol 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

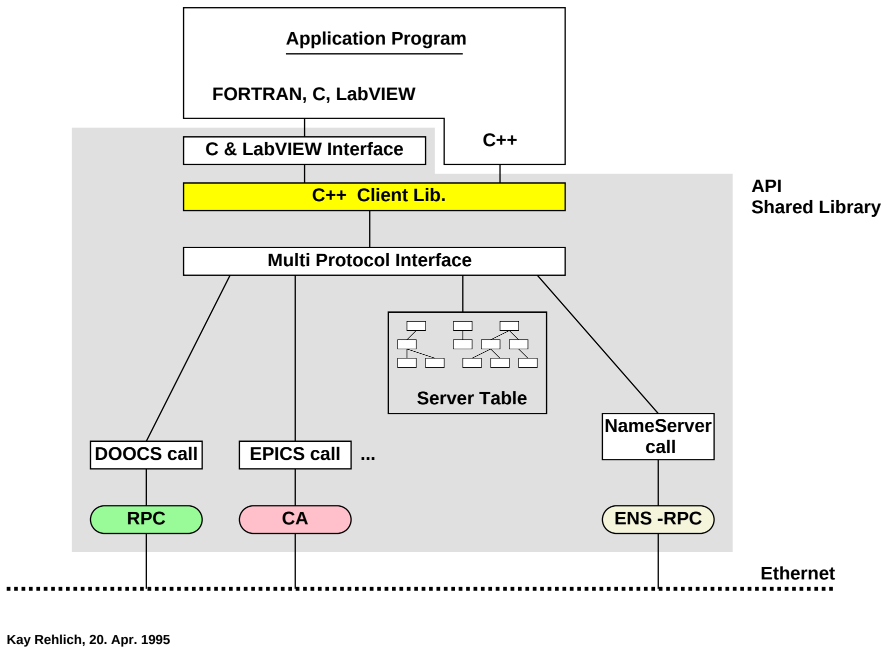


added by defining a C structure of the data and adding new methods to the data class. The conversion of a new structure into different number representations of different CPU architecture is automatically done by the rpcgen compiler which generates the eXternal Data Representation (XDR) routines.

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 communica-tion inter-face has 4 different calls only. 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.

4. CLIENT LIBRARY (API)

The next drawing shows the internal structure of the DOOCS Application Program Interface (API). In the cen-tre one can see the Multi Protocol Interface, which easily allows to add additional control protocols. So far DOOCS supports its own RPCs and the EPICS channel access (CA). The API has access to a name server (ENS), which resolves the necessary information of the servers. These data are stored inside the server table so that just one call to the name server is needed. Because of the object orientated structure



of DOOCS, the gen-eral interface is in C++. To incorporate C and Labview applications a C interface was designed on top of the C++ interface.

As data analysing programs in physics are available in Fortran, an interface to that language is provide as well.

A read call specifies an address and a data object to be sent to the server and gets a result object from the server (the sending data in a read call is used in some calls to select a reading range for instance):

```
#include 'eq_client.h'
//
// create the required objects
//
EqAdr ea;
EqData send_data, *result;
EqCall* eq = new EqCall;
//
// fill the address class
//
ea.adr ('TTF.VAC/MASS_SPECTR/VERT/RANGE');
//
// do the call and analyse the result
//
result = eq− > get(&ea, &send_data);
if(result− > error())
printf ('Error');
else
```

printf('Data is %s',
$(char*)result- > get\_string());$

This example demonstrates some methods on data objects also. The $'result- > error()'$ returns the error code of the call and the $'result- > get\_string()'$ gives the result in an ASCII string. There are a lot of other methods on data objects to get a reading as a float or int or to step through arrays and so on. The data transferred is always in the native format of the sender and converted on the receiver side into the needed format, e.g. a server that provides a float value is readable as a char string or int on the receiver, but the data on the network is trans-ferred as a float.

5. CLIENT APPLICATIONS

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 sequenzing programs. The programming in LabVIEW is done just with the mouse by connecting graphic symbols. The Virtual Instrument (vi) library to access device data is described in an additional paper by Serguei Goloborodko.

As a second operators interface we have a **DOOCS Data Display** program (ddd) which is an graphical editor to create and run control panels. ddd allows to create 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. This program is described in a further manual.

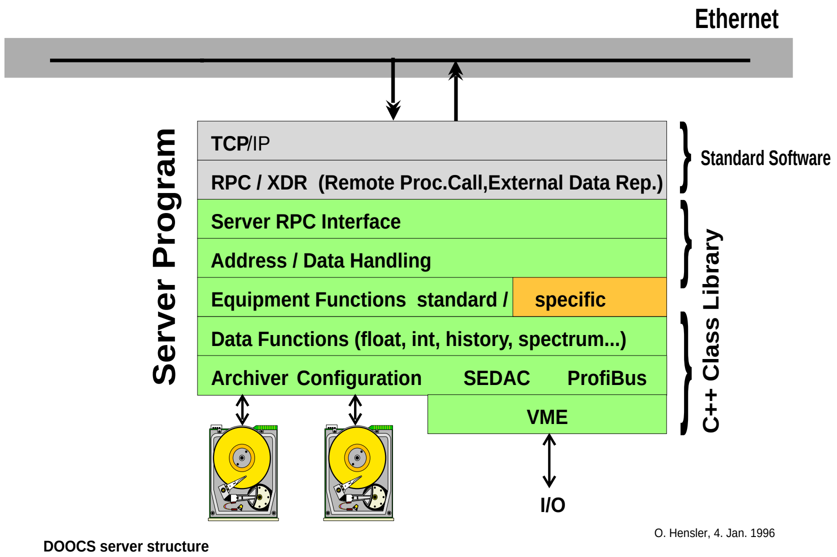
rpc_test is a generic program to display and modify all available device data. It reads the actual list of device servers from a name server and the available properties of a certain server from the process itself. Because the program gets all informations from the network, all parameters in the system 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 to show 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 commandline interfaces to read or write device data.


Ethernet
Server Program
TCP/IP
RPC / XDR (Remote Proc.Call,External Data Rep.)
Server RPC Interface
Address / Data Handling
Equipment Functions standard / specific
Data Functions (float, int, history, spectrum...)
Archiver Configuration SEDAC ProfiBus
VME
I/O
Standard Software
C++ Class Library
O. Hensler, 4. Jan. 1996
DOOCS server structure

6. DEVICE SERVER

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 communi-cation network and are implemented as data objects.

The server library defines a basic class for device servers. This basic class provides the common communica-tion objects like the name, the error status and on/off-line information. An actual device server inherits this and adds device specific code and data objects. Data objects are also part of the library and are defined for sta-tus words and bits, correction polynomials, 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.

7. SERVER LIBRARY

The server library is a collection of various classes to provide the network access and the access to the hard-ware like VME, Profibus and SEDAC. In addition there are classes to read/write the configuration file and to archive various data types to disk.

The server library provides a standard main() function, which is inside the file eq_rpc_server.cc. main() reads the configuration file, opens the RPC communication, sets up some signal handling functions (e.g. timer for update()) and then goes into the permanent running function called svc_run(). This function waits for incom- ing RPC calls.

In order to configure the main() function for the needs of a specific server, one needs to set two define varia-bles in the Makefile.

The first one is KEY_GEN, which sets a unique number for the RPC communication.

The second one is CONF_FILE, which sets the name of the configuration file for this server. The name usually should be similar to the server name. If one do not set this variable the default name is "eq_config.conf".

A device server reads the configuration file on start-up. It defines the number of instances a server has and all parameters for the device properties. If a property field is undefined in the configuration, a default value is used. The tags in the file have the same name as the property on the network. A running server can be config-ured to update the configuration file in a certain time period or after a number of modifications. This configu-ration is also done from the normal server network interface. Each server therefore provides a server instance (_SVR) on the network. This belongs to the standard part in the EqFct class.