Controlling the ESRF Accelerators

W.-D.Klotz on behalf of the ESRF Controls Group*
European Synchrotron Radiation Facility, BP 220, 38043 Grenoble Cedex, France

1 System Overview

Architecture: The ESRF control system is based on a multilevel architecture of distributed hard- and software processing units[1]. *Logically* the system is structured into four levels, which we call: **Console Level** (Presentation), **Process Level** (Application), **Group Level** (Device Servers), and **Field Level** (Equipments). On the lowest level equipments are interfaced. Equipments are logically grouped by similar functionality on the group level. The group level is responsible for hardware specific- and real time I/O-operations. On the process level practically all higher level control tasks and physics applications are processed. Powerful multitasking capabilities and fast processing are mandatory on this level. The console level deals with the man-machine interface[1]. *Physically* the system is split into 2.5 levels. All nodes of the presentation and process level consist of workstations and file/compute servers interconnected by Ethernet. The group level nodes are realised by VMEbus crates. Every process level server masters a private Ethernet segment onto which group level nodes it is in charge of are connected. The physical boder line between group level and field level is fuzzy. Some dumb devices are directly interfaced to VME I/O-boards, but most dumb devices are interfaced by means of G64 crates. Groups of G64 crates, that interface classes of similar devices, are connected to ESRF propriatary multidrop highways that are mastered by group level crates.

Networks: Apart from the multidrop highway all computer connections are based on the IEEE 802.3 LAN standard and the TCP/IP protocol suite. The network is constructed using $50/125\mu m$ multimode graded index optic fibres for cabling, that will allow a later migration to FDDI. Four networking centers are located around the storage ring tunnel, a center comprising a "NODE" and a wiring "HUB". For the active stars at the NODES, the "Lannet Multinet II" system was chosen for its ability to operate in a single chassis and support up to four independent backbone bus's. In addition all backbone fibre optic links are run in a *synchronous* Ethernet mode. The wiring HUB is the central point for passive network components, i.e. the *backbone optical fibres* that link all the HUBs together in a circular structure around the storage ring tunnel. It also acts as the termination point for all star wired fibres that are attached to the VME systems. When installing the circular

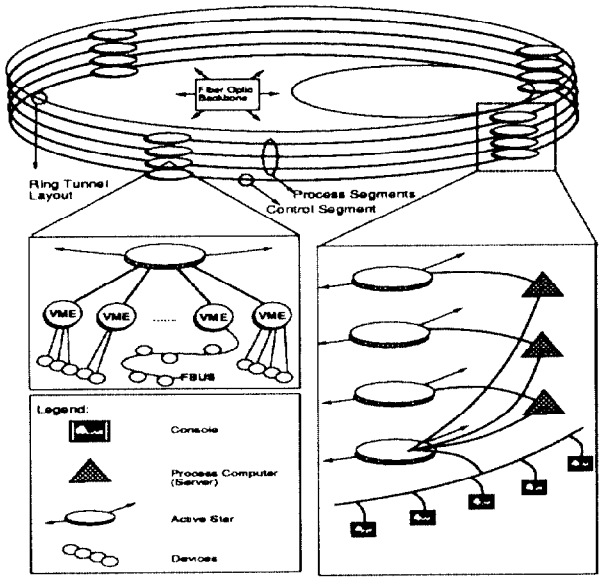

Figure 1: Overall Layout of Control System and Networks

backbone, provision for 10 independent rings has been made, out of which four will be used initially. One ring functions as the main *control segment*, to which all upper layer processors are connected. There are three dedicated *process servers* that operate as network gateways to the other three rings. The latter are used as *process segments* to which all middle layer VME systems are connected.

Computers: The consoles in the control room are HP Apollo 9000 Model 720/CRX[2] workstations. Their local disks hold the bootable image of the operating system and provide local swap space. File systems containing control system and physics applications software are remotely mounted[3] from the process servers. About 7 to 10 consoles will be needed in the main control room. As process-, file-, and data-base servers the HP Apollo 9000 Model 750 servers were selected. Four process servers are equipped with 128Mbytes of RAM and 1.3GBytes of disk space. One central file server is equipped with a disk striped mass storage system providing 8.3GBytes of capacity. Another HP 9000/750 is running as a *hot spare*. All VME systems have an identical base configuration. This comprises a

*A.Götz, D.Carron, J.M.Chaize, M.Fazio, C.Hervé, M.Karhu, P.Mäkijärvi, J.Meyer, P.Pinel, Y.Nicolas, B.Regad, V. Rey-Bakaikoa, M.Schofield, B.Scaringella, A.Smith, E.Taurel, P.Vines, R.Wilcke

[1] MMI

[2] 32Mbyte RAM, 600MByte disk

[3] with SUN NFS

Motorola 68030 CPU @ 20 MHz, 4 Mbyte RAM, and on-board Ethernet adapter. All systems are running in a *diskless* configuration for ease of maintenance and reliability[4]. System start-up, -reboot, and remote crate diagnostics are accomplished by a dedicated VMEboard that is operated through a front panel FBUS connection.

Interfacing; Most of the VME systems drive multidrop FBUS highways. This FBUS is not a general purpose network but implements a low cost remote input/ouput facility. It relies on a master-slave relationship, where a controller (VME based module) drives a large number[5] of slave nodes. The nodes comply with the G64 standard, so that full advantage can be taken of existing interface boards from industry. The transmission medium is a flexible shielded twisted-pair cable. Physical implementation uses a noise resistant Manchester encoding with transformer isolation. FBUS can still be safely operated at a speed of $1Mbit/sec$ on distances of up to $1km$ and 30 nodes without repeater. G64 and FBUS interfacing has been used for control of main magnet power supplies, beam position monitors, magnet interlocks, corrector magnet power supplies, and injection/extraction elements. Other significant subsystems that include G64 crates are the system to distribute the slow timing pulses, the video cross point switch, and the video multiplexors for fluorescent screen monitors. The rest of devices is directly interfaced to the VME systems; either by asynchronous serial lines[6] or digital I/Os.

Software: The workstations/servers run **HP-UX**; an AT&T System V Rel 3.0, and BSD 4.3 compliant implementation of the UNIX operating system. On the VME systems we use Microware's **OS9** realtime kernel/operating system. Microware's TCP/IP **Internet Support Package** provides Berkely sockets, and SUN's Network File System on the OS9 systems. The Man-Machine-Interface is based on the **X11-window** system and the **OSF/Motif** technology. The control system data are stored in relational databases which manage two logical parts: *Resource* data, and *Runtime* data. The implementation of the resource database uses **ORACLE** and its powerful set of development tools. Programming language is **C** and where available **ANSI C**.

2 Design Phases

PHASE 0: Phase "0" covered the period from project start until the beginning of storage ring commissioning. It was felt essential, that commissioning should be started with a set of *first generation application* programs, specifically tailored to this purpose, and that device interfacing and access are fully implemented for all machine parts. To reach this goal, a task sharing between the controls group and the machine theory group was defined. The controls group was in charge of installing the control system hardware and implementing all device servers, the machine theory group concentrated on writing the application programs. *A well defined, early available, and stable application programmer's interface*[7]*, played a fundamental role in this task sharing.* Using strictly the API device access interface and the X11 and Motif standards for interactive graphical

[4] remote booting is based on Network Boot PROMs using the tftp protocol

[5] up to 64 on one highway

[6] either RS422 or RS232

[7] API

	NFS/RPC		API
	UDP	TCP	UDP
open connection	20-25ms	35-45ms	55-65ms
close connection	0.1-0.2ms	0.3-0.5ms	10-20ms
RPC with 100bytes	10-15ms	15-20ms	15-20ms
RPC with 8kbytes	25-35ms	55-65ms	30-40ms
RPC with 40kbytes		220-250ms	

Table 1: Performance figures for RPC and API

I/O, an impressive number of physics applications have been developed in parallel with the basic control system software, and are available for commissioning now. The controls group's major priorities during this phase were on:

Installation of the control system hardware: A survey of field- and group-level hardware in terms of numbers of installed boards and system crates is given below (note that this excludes four subcontracted control systems for LINAC and three RF transmitters):

VME Crates	52
VME CPU 68030 @ 20MHz, 4MB RAM	53
VME Digital 64 in/32 out TTL	9
VME Serial 12 RS 422 channels	104
VME Analog 16 out, 12 bit	41
VME Video Mux 15:1	3
VME Image Processor	2
VME Stepper Motor Controler	19
VME hex prog. Delay Unit	14
VME Timing Master	1
VME Master Clock Divider	1
VME Remote Diagnostic Controler	50
VME FBUS Master Nodes	37
G64 Crates	112
G64 FBUS Slave Nodes	194
G64 Digital 16 in/out TTL	161
G64 Analog 8 in, 12 bit	190
G64 Analog 8 out, 12 bit	12
G64 Timing Pulse Driver, 2 ch.	74
Stepper Motor Power Driver	64

Development of a network transparent API: Access to the Device Servers is provided by three C calls. These calls allow the users to develop their applications in peace without being affected by what goes on in the Device Server software. Network transparency is achieved by using Remote Procedure Calls[8].

Development and installation of device servers: A unified model (called the *device server model*) has been developed to solve the problem of device access and -control. It uses a RPC-based Client/Server technology, which is a simple mechanism to distribute software tasks across any number of processors. Each device is an object created at start-up which has its own *data* and *behaviour*. Each device has a *unique name* for identification in network name space. Devices are configured via *resources* which are stored in a *database*. Devices are organized in *classes*, each device belonging to a single class. Classes are implemented in C using a technique called *Objects In C*. This technique is similar to the "widget" model from the X11 Intrinsics Toolkit of MIT. All classes are derived from one *root*

[8] RPC and XDR from SUN

class. The root class contains a generic description of the device and the basic client/server communication facilities. Subclasses inherit attributes and methods from parent classes. Actions on a device are made available via *commands*. Commands can be executed *locally* or *remotely*. Applications access a device and its commands using the API. Currently 81 Device Classes have been implemented, covering $\sim$ 80% of the total system. This reflects roughly: 8800 analog inputs, 7700 analog outputs, 2Kbyte digital inputs, 1.5Kbyte digital outputs, and 2.4Mbytes of released (tested!) C source code. The latter numbers do not include the large amount of devices for beam line front ends and insertion devices!

Development of the static data base: The resource database keeps device server resources and static data. Examples are: start-up resources, calibrations, equipment definitions, installation- & maintenance data, bookkeeping and archive data, etc... Presently it deals with:

Devices defined	4737
Devices exported	4206
out of which:	
for the LINAC domain	82
for the Transf. Line 1 domain	41
for the Synchrotron domain	457
for the Transf. Line 2 domain	73
for the Storage Ring domain	3023
for the Front End domain	97
for the Beam Port domain	429
Resources defined	27144
out of which:	
for the LINAC domain	461
for the Transf. Line 1 domain	196
for the Synchrotron domain	2711
for the Transf. Line 2 domain	572
for the Storage Ring domain	16186
for the Front End domain	1099
for the Beam Port domain	5312
for the Device Server Classes	590

PHASE 1: We currently enter this phase that will last the next two years at least. During this phase the control system will be upgraded from its initial commissioning state to a system that fully supports all facets of machine operation. Apart from small hardware amendments it is mainly development effort in software that will take place:

API: Currently the API is based on a blocking (**synchronous**) RPC. The calling client waits until the call returns from the server before continuing. Eternal waits are avoided by setting a *timeout* when calling the server. The API will be extended by a nonblocking (**asynchronous**) version which will dispatch the command and then return immediately. The response from the server will be queued and returned to the client when it is ready to receive it.

Runtime Database: Until now clients access devices through device servers on a command/response principle. Buffering or caching process data is the responsability of clients. We will add a runtime database as an independent entity to the system. This database is not a medium for permanent storage nor for tunnelling I/O requests to the device driver level. It is a buffering/caching *front for permanent storage*. Only *memory resident* database systems can meet the demands for sufficiently short transaction times. A prototype of the runtime database is operational and uses a *Real-Time Database Base Management System*[9] available on HPs. This database can be used to alleviate congestion problems. Multiple processes can update data asynchronously in the database. Clients can retrieve this information aynchronously without blocking the process doing the updating. The runtime database's prime source is a so-called *Update Daemon* that updates machine parameters periodically. Clients can issue booking requests for parameter updating at runtime. Streams of on-line data can be archived continously. Only a time window of predefined size is kept in memory by RTDB, the rest of the data is dumped into the disk-based ORACLE database. RTDB can also be used by applications as a means for interprocess communication. Applications dynamically allocate "tables" of formatted data, that can then be piped or multiplexed to other applications.

Security: Security is a key issue for a distributed system, and will be added on the device server level. The solution adopted here has been modelled on the Unix **access control lists** as described under ACL(5) in the Unix manual. An ACL consists of sets of **(user.group, mode)** entries associated with a file that specify permissions. We will extend this idea to define permissions for devices. Each entry specifies for one user-ID/group-ID combination a set of access permissions. The permissions a user must have to execute a command on a device are stored in the command list of the device server The set of permissions that we will implement for devices will be:

Read	command will only read the device
Write	command will write to the device
Single_Write	command will write but only in single user mode to device
Special	command can only be executed by priviledged user
Single_Special	command can only be excuted by priviledged user in single user mode

Man-Machine Interface: Both X11 and Motif, are extremely helpful but their libraries are complex to learn and to use for programming. User Interface Management Systems (sometimes called interface builders) are the tools which help the application programmer to design the user interface part of the application interactively. ESRF selected a UIMS that generates stand-alone C code and/or a combination of Motif-compliant C and UIL code[10]. This UIMS drastically eases now the design of Motif-based user interfaces. Synoptic drawings with selectable objects are scarcely supported by Motif. We therefore implemented a Motif compliant widget that uses vectorial drawings generated by PHIGS[11]. In addition to that, powerful X11/Motif compatible tools become available now that can be integrated into the MMI. At ESRF we started to use the spreadsheet WINGz, which has its own control language "HyperScript" that can be extended by user-defined C functions. Complete control applications can be written by using WINGz's presentation- and button tools that create interactive worksheets.

References

[1] W.-D. Klotz, "The ESRF Control Ssytem; Status and Highlights" in *Proc. ICALEPS* , Tsukuba, Japan, Nov. 1991.

[9] called RTDB

[10] Builder Xcessory from ICS

[11] Programmer's Hierarchical Graphics System