CONTROL SYSTEM FOR CRYRING

E. Westlin, M. Engström, Manne Siegbahn Laboratory, Stockholm, Sweden

Abstract

A new modular control system architecture using COM/DCOM for the CRYRING synchrotron/storage ring in Stockholm is described. The purpose of the system is to provide access to control system parameters in a generic sense. The current communication hardware consists of a CAMAC bus, serial modbus and socket connections. Client programs communicate with servers using DCOM[1]. A basic server controls access to one hardware communication port. Each client connects to one or more servers.

1 SYSTEM ENVIRONMENT

The CRYRING, a 52 m circumference synchrotron and storage ring at the Manne Siegbahn Laboratory in Stockholm, is in operation since 1990 for research mainly in atomic and molecular physics. The facility includes, beside the ring, a number of ion sources, an RFQ pre-accelerator, and beam transport lines to a number of experimental stations using the low-energy ion beams directly from the ion sources.

The number of parameters to be handled by the control system is presently around 300. Out of these, around 250 are of the simple type having a static output value, and around 50 are of the time-varying type being controlled by autonomous function generators. The system is expanding with the accelerator facility. The present data distribution infrastructure is based on serial CAMAC and local G-64 controllers, but other types of communication and local control will be used for the future installations.

2 PARAMETER MODEL

2.1 General

Basically a parameter is the object of interest in any control system. Since this new control system is to replace an old one with as little inconvenience as possible and maybe also because it is a good way to structure the system, the old parameter concept has been kept intact.

The system also has, apart from the old parameters, to handle new types of equipment the old system didn't or couldn't handle. Because of these requirements a more generic parameter concept has been adopted, resulting in a hierarchy of parameter types, where some functionality is common and other is specific to each parameter type.

Two types of access methods have been implemented, one structured and fixed, the other unstructured and flexible.

2.2 Structured Access

This method of access uses COM/DCOM versions of the parameter access methods. This design gives convenient, efficient and robust access methods. These methods use scalars, vectors or structures as values. The main drawback is that they are predefined and can handle only one type of parameter. So either one squeezes new parameter types into one already existing, or write new dedicated interface functions.

2.3 Unstructured Access

However to facilitate a more generic access to all parameters, regardless of type, a set of generic access methods has been developed. These methods have adopted the concept of a path to a data point. The value to transport to or from this data point could be of any type (within certain limits of course) as far as the client/server communication concerns. It's the responsibility of the sender/receiver to serialize/unserialize the data. The basic building block for data transfer is an encapsulated union not unlike the VARIANT datatype. To aid the development of client/server serialization a special archiver, derived from the MFC CArchive class, is being developed, enabling a technique similar to the one deployed in ConSys [2], the new control system of the ASTRID ring in Aarhus.

Typically a data point corresponds to an attribute of a parameter but it could also be an attribute of a module driver to be used when adjusting the behavior of the control system. It could also be a more abstract point such as a 'SCRIPT' point as explained below. The basic requirement on the path is that it should begin with a parameter or module name, the rest of the path is parsed by the intended parameter or module implementation, so this could be quite free formed, similar to an OPC item definition [3].

3 CLIENT/SERVER COMMUNICATION

3.1 General Operation

The communication is based on DCOM. Clients interested in some parameter value have two choices of reading values, either they use the forward on demand interface or register a subscription and at a later time gets called back by the server. The callbacks could be conditional as ordered by the client. The subscription/callback functionality is in the following referred to as the alarm system

3.2 Clients

A rather fat client library communicate with one or more servers. The client hide most of the DCOM system calls. A database object/server lets the client know which parameter belongs to which server. The same database object is also accessed by the servers. End user programs either link with the client, as is the case with some legacy programs, or communicate with the client via ActiveX/OLE.

3.3 Server Structure

The system is built to solve specific needs and there has been no deliberate attempt to build a system handling every conceivable situation. However when using the adopted method, which is influenced by the Unix STREAMS modular system [4], the situations encountered have found straightforward solutions.

The basic building block of a server is a Module Driver, a concept inherited from the old control system. In the new system this has been made into a more generic term. The end user client communication over the net via DCOM connects to a client proxy object in the server process. Internal to the server is a head module driver which is the conceptual module driver object. This object persists in the system database and keeps record of the parameters accessible through the server. The head module driver in turn connects to other module drivers implementing embedded protocols.

A module driver is typically implemented in a separate DLL. It is accessed through a COM in-process server interface. Each module driver has a unique name and several module drivers can be instances of the same COM class. Which module driver instance to select is determined with a special login method in the COM interface. All clients on all levels, also inside a server, are required to first call this method.

3.4 Server Execution

All access through a hardware port is controlled by a main work thread, residing in the head module. When a client makes a request the server converts this into a job object, similar to a process instance in a regular operating system. This job object is placed into a queue where the main work thread, awaken by the system heartbeat or by the arrival of a new job, later picks it up, executes it, and then typically wakes the client proxy thread, which can return the result to the client process.

The queue is actually implemented as two separate queues, one sleep queue and one run queue. When the work thread runs, it first checks which jobs in the sleep queue has timed out, and moves those to the run queue. Next the work thread takes the highest priority job from the run queue and executes it. This execution preferably should not contain any wait states, but it's up to the implementation of the job object to exit and return since there is no preemption. To improve concurrence, job execution should be short, and if longer jobs are needed, they could be divided into several invocations of the job. When the jobs execute method finishes to the main work thread loop, it returns a code which is interpreted as follows

JOB_DONE: The job was successfully executed. Signal the client proxy.
JOB_MORE: A longer job yields to give other jobs some time to run. The job is reinserted into the run queue.
JOB_SLEEP: A job has a wait state. It is placed in the sleep queue.
JOB_ABORT: There was an error. Signal the client proxy.
JOB_BLOCKED: The current job has a wait state and the module driver can not execute other jobs. This causes the module driver to enter a blocked state, and to be placed in the sleep queue of it's parent module.

3.5 Batch Processing and Server Internal Jobs

The inherent scheduler can be used to implement batch processing. This has been used to implement caching and the alarm event handling in the system. It could also be used to implement a scriptable scheduler. This batch processor could accept sleep commands and parameter set/get requests. Using the alarm system one could implement triggers similar to those found in database managers.

An example of what a trigger might look like:

```
When C9CAMSM.ACQ > 100.0 do
        C9CAMSM.CCV=0.3
```

Other module drivers combine several parameters into new ones. And using the scheduling, closed loop regulators can be built. An example of this is a simple PID regulator.

REFERENCES

[1] Randy Abernethy, COM/DCOM Unleashed.

[2] ConSys. http://www.isa.au.dk/consys/

[3] OPC, OLE for Process Control. Data Access Standard Version 1.0A September 11, 1997.

[4] Dennis M. Ritchie, A Stream Input-Output System, AT&T Bell Laboratories Technical Journal 63, No. 8 Part 2 (October, 1984).