CORBA FOR CONTROL SYSTEMS

Ricardo Sanz

Universidad Politécnica de Madrid

Abstract: The Common Object Request Broker Architecture (CORBA) is a middleware specification for the development of interoperable, distributed object systems. Object technology is of extreme importance in complex control system development besides its progressive use in real-time systems. This paper tries to provide a general overview of the topics in distributed object systems, focusing on CORBA aspects that are critical for control systems engineering. Some sample applications are presented. *Copyright ©2000 IFAC.*

Keywords: CORBA, object-oriented systems, distributed systems, real-time systems, development methodologies.

1. INTRODUCTION

1.1 *Information Technology in industry*

The process of incorporation of information technology (IT) into industrial processes is making profound modifications in production systems. Control and monitorization technology is leaving the *islands-of-automation phase*, entering a new phase of complete systems integration. While enterprise integration architectures (EAI) are hot topics in advanced business engineering, at the production level where controllers live, the incorporation of new technology and designs is confronting difficult problems.

In most cases the problems are mainly due to classical barriers posed to innovation in production systems: lack of predictability, need for non-stop operation, lack of reliability and availability, less than ideal market maturity, exploitation managers resilience, etc.

Two main objectives are being pursued in this effort, namely Complete Horizontal Integration (CHI) and Complete Vertical Integration (CVI). CHI deals with the integration of business units, business-to-business integration or supply chain integration.

In his speech we will address more the topic of CVI. It is time to start thinking in plant-wide integration reaching even the lowest levels in production plants: sensors, actuators and basic controllers.

Distributed object computing (DOC) is gaining an increased audience in information technology and, in new tech sectors, it is the technology of election for new system implementation. From global experience in last years it is pretty clear that -besides other advantages- DOC technology enhances systems integrability, making easier the construction of complex information applications. We will see in this paper how a DOC technology, namely CORBA, can supply us with some tools needed for better development of complex, integrated control systems.

1.2 *Control engineering processes*

Control systems complexity is increasing at a very fast pace in this days. New needs and new capabilities (nobody knows who come first) are driving control systems development into mainstream systems engineering. Integration capabilities are getting progressively critical as system size increments, because modular development is the only known practical way for complex systems engineering.

From this perspective it is surprising, to some extent, the limited role that software technologies play in control engineering journals and symposia. It looks like this technology does not have relevance enough to be considered a research discipline for control engineers (only small-scoped real-time topics are addressed in control engineering places).

This paper does not contain equations, nor feedback loops or control algorithms ... Can we say it that it is relevant for control engineering? The answer is *Yes, it's very relevant*. Let's take a closer loop at the the *real structure of a control problem*.

The typical development process of a control system can be decomposed, like any other engineering process, in a series of phases that go from the identification of the need to the decommission of the control system.

An example of phasing can be:

(1) Problem identification
(2) Plant Modeling
(3) Control design
(4) Control implementation
(5) Commissioning
(6) Operation
(7) Decommissioning

Research in control systems has been mainly focused in the second and third phases, because the first is considered an *a priori* for control engineering (*i.e.* it is always given) and from fourth to seventh they can be left to implementors (*i.e.* to raw work force). The separation between the control laboratory and the real plant is too wide for real engineering.

The basic technology used today to implement control systems is software technology. But, beyond a classical view of digital implementation of controllers (Åström and Wittenmark, 1997), software technologies are the basis of modern complex control systems, from SCADAs and DCSs to intelligent controllers based on soft computing (Gupta and Singh, 1996).

Software is the main *implementation* tool and this has relegated software technologies from the core *theoretical* control discipline. Any real control engineer can see this problem taking a view on *"consolidated"* control engineering magazines as *Automatica* or the *IEEE Transactions on Automatic Control*. No paper about a software topic will find a place in any of them; it will be relegated to the fellow *implementations* journal. This is a big mistake. Not only big, but critical for the discipline.

Control engineering is about *systems performance*; this means that the knowledge of the controlled system must involve not only the target system but the controller itself, and when controllers are software-based, giving a guarantee on global performance means a clear analysis and deep understanding of software issues. When controller complexity increases there are no available formal methods to guarantee behavior. Only good development processes can provide an statistically predictable quality. Good processes involve all controller life-cycle; from the problem identification phase to the operation phase and even decommission).

When complexity increases due to software flexibility the probability of failure increases. Systems that were manually operated are now operated by computers and this leads to a critical computer dependence of many artificial systems. The case of the USS Yorktown is paradigmatical. The ship had to go back to the harbor due to a software failure.

While software is becoming a real problem, it is also providing some solutions. For example, advanced research topics on systems fault-tolerance are strongly based in information processing capabilities that are used to detect the fault, isolate it, and devise alternative control strategies that can overcome the fault (Blanke *et al.*, 2000).

1.3 *Complex Software for Control*

Software systems can range from a small shoe shop database to Star Trek's *USSS Enterprise* control software. In a quick effort we can make a quick and dirty classification of software systems based on factors that induce systems complexity:

- Conventional: the shoe shop database.
- Real-time: meeting deadlines.
- Embedded: run within limited resources.
- Fault Tolerant: good behavior under faults.
- Distributed: run on several interacting computers.
- Intelligent: solving ill-posed problems.
- Large: millions of lines of code.
- Integrated: interoperate with alien systems.
- Heterogeneous: run on heterogeneous platforms.

Complexity factors affect negatively the systems development process. Development effort grows with complexity much more than linearly and there are even systems we cannot build; examples are 24x365 systems (total availability), one-shot systems (should work at the first try)[1] or HP-LC (High Performance and Low Cost).

[1] Star wars was an example of this problem.

Software engineers have always been raiders of the silver bullet (Brooks, 1992) looking to solutions to software development problems. Complex software engineering is just an emerging discipline, that is slightly appearing in frontier areas between those complexity topics mentioned before.

2. INTRODUCTION TO COMPLEX CONTROL SYSTEMS

A typical control system in a modern plant is composed by a heterogeneous collection of hardware and software entities scattered over a collection of heterogeneous platforms (operator stations, remote units, process computers, programmable controllers, intelligent devices) and communication systems (analog cabling, serial lines, fieldbuses, LANs or even satellite communications). This HW/SW heterogeneity is a source of extreme complexity in the control system regarded as a whole.

Apart from the platforms that provide support to the different control system components, the technologies used in control system implementation are quite heterogeneous and provide functionalities that go well beyond the classical sensing-calculating-acting triad.

Examples of this heterogeneity is the use of software systems for controller autotuning, advanced monitorization, filtering and estimation, adaptation and learning, plant-wide optimization, or real-time, in-the-loop simulation. Interception software systems are playing a wide collection of intelligent roles in complex controllers fitting as interfaces between pre-existent systems (pants, controllers and humans). Examples of these roles are data/action filters and monitors.

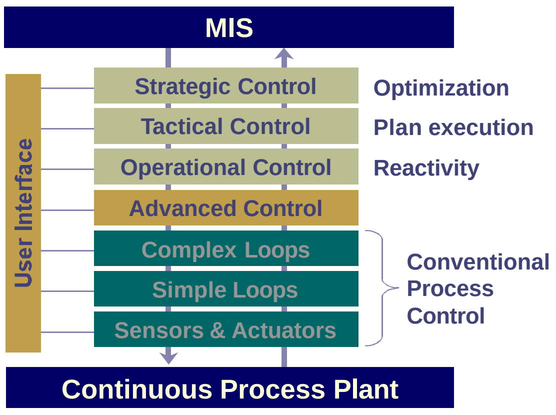


Fig. 1. A classical layering of control entities in a complex continuous process control system. Layer quantity and labeling is somewhat field-dependent, but layer roles can be easily mapped from domain to domain.

Classical hierarchical layering overcomes some of the difficulties of complex systems construction. An example of layering is shown in Figure 1 where some *intelligent* layers are added atop classical control layers in process control systems.

While hierarchies encapsulate low level behavior, simplifying the deployment of higher level controllers, they do not necessarily solve the problem of the *conceptual integrity* of the system. Layers can be difficult to match if they lack a common view of structure and responsibility distribution.

Conceptual integrity –an elusive, difficult to define property– is seen as the core factor affecting systems constructability. Conceptual integrity manifests in several system properties (some of them functional and some non-functional) that are considered extremely important in systems construction. These properties are the basic design principles of systems architecture(Shaw and Garlan, 1996) (See Table 1).

Table 1. Architecture design principles

Conforming	Scalable
Suitable	Simple
Composable	Standard
Modular	Proven
Extensible	Performing
Fast	Efficient

We will see in the next section how object technology can provide us with some ideas and tools to approximate this ideal of system conceptual integrity.

3. THE ROLE OF OBJECT TECHNOLOGY

The very nature of control systems is object-oriented (OO) because a control system couples virtual entities with real ones. A controller correlates control design issues and software implementations –that are very conceptual in its nature– with sensors, actuators and external world entities –that are very physical objects.

Control software makes a continuous mapping between external an internal entities and hence, object-oriented software is a natural way to build these systems. During the last decade OO technology was relegated from mainstream real-time software because OO implementations introduce computational overhead to support some aspects of OO computation (for example, dynamic binding). While this is usually the case, today computational power makes less important this overhead, and OO technology is becoming the technology of election to build complex real-time systems *because it provides better mechanisms for complexity handling*. An example of big importance for us is the case of real-time distributed systems, where OO technology is a clear winner (Shokri and Sheu, 2000).

Industrial plants are *Seas of Objects* and software-intensive controllers for them reflect this nature. The natural plant-modeling mechanisms are object-oriented (Rodríguez and Sanz, 1999) and dealing with preexisting software systems –for example legacy controllers– is best done using object wrapping. Advanced controllers are designed using clear responsibility distribution between control objects (CRC cards are a good approach to distributed controller analysis). This approach enables the development of architectures that exhibit some of the properties of Table 1 (Rushby, 1999).

3.1 *Objects, components and agents*

This discussion about responsibilities lead us to a concept of control systems as collections of interacting agents that match the most classical object-oriented view: objects have inner life and interact by means of message interchange.

Old days' object passivity, like the Smalltalk's approach to OO, do not fit well with our distributed controller model because it employs only one thread of control that leads to a computation model based on the sequentiation of method request and execution. This approach does not fit our needs because activities in the world are *naturally concurrent* and not sequential.

The need of going out of mono-threading was clear very soon and mechanisms for dealing with it were promptly added to OO systems. Exception handling extensions to classical environments and special multi-threading environments were developed for support this concurrent model; but true concurrence is only possible in multiprocessing environments: multiprocessors and distributed systems.

The extension of multi-threading support in operating systems provided a fake but very effective environment for concurrence. The simplified and less resource consuming model of thread interaction has demonstrated a benefit for complex systems development.

This support from the operating systems has brought new life to object systems. Object are no longer passive entities that become active only upon request from other objects. We can distinguish two types of activity:

Re-Activity: Objects are active in response to other objects' requests.

Pro-Activity: Objects are active pursuing object's own goals.

Objects are no longer passive, they can initiate activity by their own *will*. Beyond the big philosophical discussions in artificial intelligence circles about the meaning of the term *autonomy*, this step to object pro-activity has been the first true step to real, practical autonomy. Pro-activity was obvious in past control systems but not from a perspective of inner will of the entity. Only from the fusion of pro-activity and responsibility a true advance to autonomy has been achieved.

We are then reaching fields that go beyond simple, single activity and we get immersed in a process of agentification. Agents' technology provide models of autonomy in limited scopes; where agent interaction is a central issue. There is a lot of wasted words and paper in relation with agency, and, as a result, "agent" has almost lost its meaning (Sanz, 2000). Reading a dictionary, two concepts of agent emerge that fit our purpose: a. Those who act b. Those who act on behalf of others. This last meaning is the common interpretation in the Internet-related agencies (mail filters, web crawlers, etc). The first sense is best suited for our view of complex control systems.

Distributed control systems are agencies, where each agent pursues an objective. The operational cycle of each agent is based on three interrelated activities: Sensing, Reasoning and Acting (*i.e.*a control loop).

Agent-based models of software development are focused on the partial autonomy of agents in relation with an specific task (Sanz *et al.*, 2000). The resulting agency is a community of communicating entities that perform a global, rational decision making process by means of negotiation and collaboration. This involves in many cases policies for resource sharing, creation of markets and contract signing.

Models of objects, agents and components (object and agent materializations) are evolving to a common view. This view fits the control engineering view of a component for a distributed control system. This means that modern complex distributed controllers are being built as a collection of reusable components that implement agencies to achieve a final objective that is shared-by or emergent-from a collection of agents. In this paper I will use the term *agent* or *object* to refer to the same type of *"active object"* entity. *Component* will be the term used to refer to concrete implementations of the agents.

Agent componentization is a good foundation for a research "product line", because it offers a foundation for easy, component-based development of new systems using proven compo-

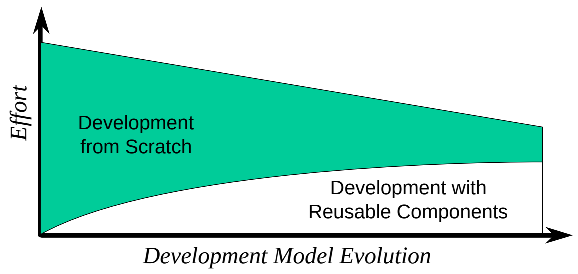


Fig. 2. Resource allocation to custom development and component reuse evolve with the development model.

nents. This let developers concentrate the efforts in new components they are developing and not in well known components that need to be rebuilt to fit in a new schema. Companies moving to a component-based development process have seen that the effort needed for new applications is reduced as is reduced the effort put in custom, application-specific developments (this is due to the product-line focus of most companies).

3.2 *Distributed Object Computing*

Distributed Object Computing (DOC) or Object-Oriented Distributed Processing (OODP) is a software model based in the use of services provided by objects that are running in different hosts. Distribution means true concurrence even when most distributed applications serialize behavior of the application using some form of centralized controller.

DOC can be considered a generalization of the client/server model. In DOC, client and server roles are relative to a specific request and not to the whole life-cycle of the object (an object can be the client in a request and the server in the next one).

DOC is a "natural" way of modeling distributed systems because it hides implementation details (OS, protocols, languages) behind "interfaces". Encapsulation, abstraction and inheritance are valid and very useful concepts to model distributed control systems.

There are many benefits of using DOC for control systems engineering. In many cases they are the same as for any other type of system, but in most situations they are of critical importance for control software due to the special requirements posed to control systems. Examples of these benefits are:

- Object collaboration through connectivity and interworking;
- performance through parallel processing;
- reliability and availability through replication;
- scalability and portability through modularity;
- extensibility through dynamic configuration and reconfiguration;
- cost effectiveness through resource sharing and open systems;
- maintainability through hot swapping and
- design flexibility through transparency.

DOC is an extremely valuable model for control software development.

3.3 *Integration*

DOC technology addresses particularly well one of the main problems of complex systems construction: *integration*.

If we consider the interaction between two pieces of code (let's call them the *client* and the *server)* we can identify four relative positions (*i.e.*four coarse types of integration mechanisms:

In-Thread: Client and server are parts of the same thread. Interaction is done by method call. This means serialization (no concurrence) and a simple integration vehicle (programming language routine invocations). This is easy to use, extremely fast and reliable. It is strange to have client and server in a different state –from a reliability perspective– due to external factors.

In-Process: Client and server are parts of the same process but in different threads. We have inter-thread requests usually based on ITC [2] mechanisms provided by the operating system. This is relatively complex but is very fast and reliable.

In-Host: This situation is similar to the previous, but in this case client and server are in different processes. Inter-process requests are based on operating systems IPC [3] . This is also a fast and reliable mechanism.

In-Net: Client and server are in different hosts. The basic integration mechanism is some form of remote procedure call (RPC) [4] Inter hosts requests rank lower in speed and reliability because it is easier to have different host states in client, server or even communication channel. Distribution means in many cases unpredictability and unreliablility.

Middleware is a generic name used to refer to a class of software whose sole purpose is to serve as glue between separately built systems.

[2] Inter-Thread Communication.
[3] Inter-Process Communication.
[4] Lower level mechanisms can also be used but in most cases it is not worth the effort.

Object-oriented middleware is used to simplify the development and use of ubiquitous objects. Middleware tries to simplify the implementation of clients and servers for different relative locations; for example making possible the implementation of clients that are unaware of server locations.

A big simplification is achieved using the same interface to be used by client and servers independently of the base integration mechanism; *i.e.*the same interface is used to wrap an IPC and an RPC (see Figure 3).

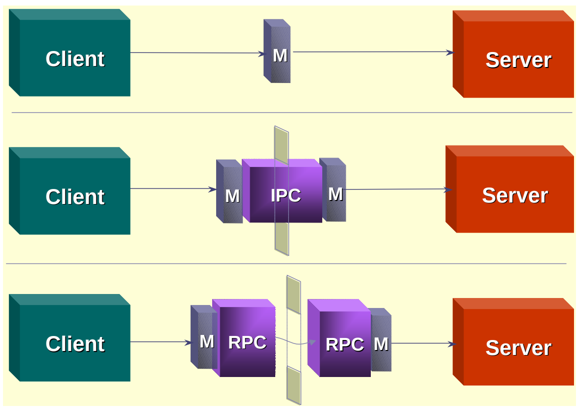


Fig. 3. A great simplification is achieved using the same interface to be used by client and servers to use/provide the service.

But the real big step is when this interface is independent of the relative location of the opposite object (see Figure 4).

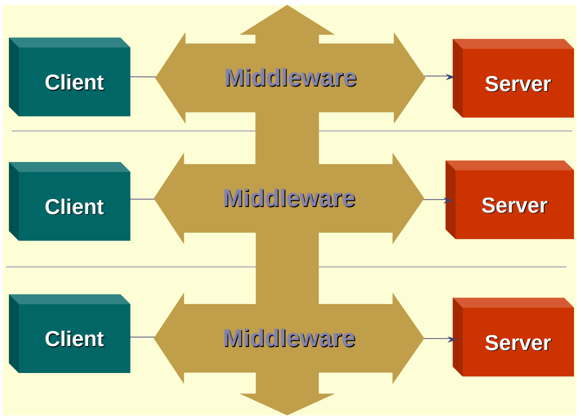


Fig. 4. Middleware hides underlying mechanisms to provide an homogeneous platform for ubiquitous computing.

Brokering middleware is based on the use of an intermediary entity between the client and the server: the broker (See Figure 5). The process of remote invocation is decomposed in eight steps:

1. The client makes a call to the client stub (the client plug to the broker).
2. The client stub packs the call parameters into a request message and invokes a wire protocol.
3. The wire protocol delivers the message to the server side stub(the server plug to the broker).
4. The server side stub then unpacks the message and calls the actual method on the object.

5-8. The response -if any- uses the same process to reach the client.

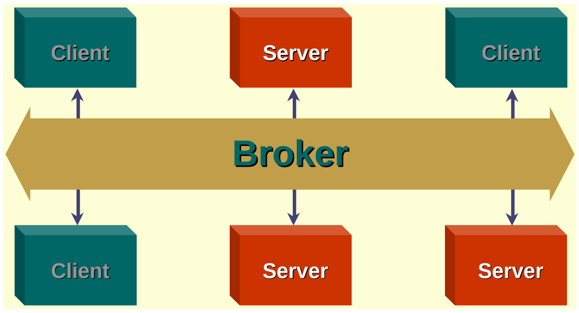


Fig. 5. Brokering middleware is based on the use of an "intelligent" intermediary between clients and servers.

3.4 *Middleware and muddleware*

There are many contenders in the object-oriented middleware arena. The three main technologies are Microsoft's COM+, Sun Microsystems' Java RMI and Object Management Group CORBA.

There are big discussions about what "is the best technology" but –as is the usual case– there is no clear winner. If we try to understand all the arguments we easily get into the muddle (of terminology, of arguments, of policies, of money and lost business opportunities, etc.).

From my point of view there are some clear –although partial– criteria to follow:

- Homogeneous-platform applications on MS desktop machines: COM+
- Internet-wide heterogeneous platforms: Java
- Fully heterogeneous and special requirements: CORBA

But what are those "special requirements"? The answer to this question is pretty long but extremely interesting for control engineers: Real-time behavior, fault tolerance, small memory footprint, pervasive heterogeneity (hardware, operating system and programming language), platform resource control, vendor independence, open specification process, modularity, embedability, etc.

Does it mean that you cannot use COM or Java for control applications? Not so, it only means that you cannot use them if you have some of these requirements unless you want to put a huge effort to fulfill them. It is simpler to build applications with tough requirements using CORBA[5]. There are many good books on CORBA (many of them published by OMG Press/Wiley) but the book of Jon Siegel is particularly important (Siegel, 2000).

Things are somewhat changing for Java. While it was originally developed for the embedded market it reaches the public recognition in website and Internet programming. Now, after the approval of the Real-time Java specification, perhaps it can regain the embedded and real-time markets.

4. OMG, UML AND CORBA TECHNOLOGY

OMG stands for Object Management Group[6], an organization created to foster object technology by means of the creation of a software marketplace for object technology. Using OMG's object technology any organization can leverage previous efforts in building control systems. Two of the main components of this technology spectrum are CORBA and UML.

The OMG is an *standardization organization* with an open, vendor-neutral, international, widely recognized and rapid standardization process based on demonstrated technology. It is composed by more than 800 members (for profit and not for profit organizations), with tens of concurrent technology processes, ranging from networking infrastructure to air traffic control or human genome data management. It maintains a strong liaison with other organizations as ISO, ITU-T, W3C, TINA-C, Meta Data Coalition, etc. OMG's object technology is the object technology of reference: IDL, UML, MOF, XMI[7], etc.

The main contribution of OMG to the OO world is the Object Management Architecture (OMA). This is an specification for the construction of open distributed object systems based on brokering and a collection of predefined services (OMG, 1998*c*).

The technology provided by the OMA can be grouped in:

Object Request Broker (ORB): is the run-time integration vehicle that forwards requests and responses.

Interface Definition Language (IDL): is the interface definition mechanism for implementation independence.

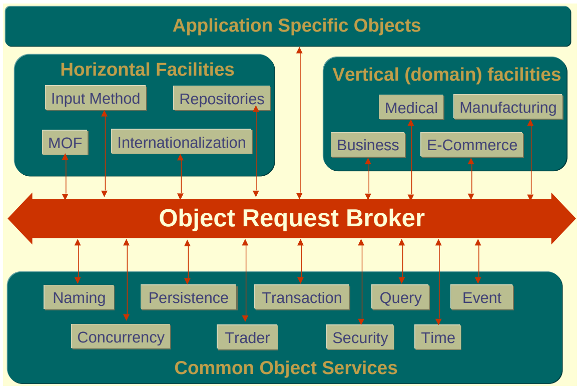


Fig. 6. OMA Overview. The main parts are shown but only some of the services are detailed.

Language Mappings: are the mappings from IDL to several programming languages for the implementation of client an servers.

Repositories: are stores that provide run-time information about interfaces and implementations.

Interoperability: between CORBA systems and with external entities (like Microsoft COM).

Most of these specifications are contained in the main CORBA document: *Common Object Request Broker Architecture and Specification* (OMG, 1998*a*)[8].

The OMG provides extensions and profiles over the base specifications as separate documents that specify the points of departure from the main specification. Of special importance for control systems engineering are the Minimum-CORBA specification; the Real-time CORBA specification and the Fault-tolerant CORBA specification (see Section 5).

4.1 *CORBA IDL*

OMG IDL (Interface Definition Language) is an implementation independent language used to specify CORBA object interfaces. It is now an ISO standard and has several interesting characteristics: it supports multiple-inheritance (not so common in OO technology); it is –obviously– strongly typed; it is independent of any particular language and/or compiler and can be mapped to many programming languages (some mappings are specified by the OMG and others are contributed specifications); it enables interoperability because it isolates interface from implementation.

[5] Even while being knowledgeable in CORBA is a daunting task.
[6] Find it at `http://www.omg.org/`.
[7] Interface Definition Language, Unified Modeling Language, MetaObject Facility, XML-Based Metadata Interchange.

[8] In release 2.3, the language mapppings were taken out to constitute separate documents with own evolution.

4.2 *OMG Structure and Activity*

The OMG technical activity is organized around three bodies:

- the Architecture Board, responsible of the OMA and the verification that new specifications are compliant with it;
- the Platform Technology Committee, responsible of CORBA core technology, and
- the Domain Technology Committee, responsible of specifications in vertical domains.

The work is performed by a collection of working groups in the different areas; from core technology like the interoperability protocols to domain specific activities like data acquisition or financial security.

The OMG specification process is based on the submission of specifications from private organizations in accordance with Request For Proposals (RFPs) done by the OMG. This means that the specification elaboration process is not done by an standardization committee (ISO C++ took more than eight years) but by an –usually– small group of OMG members *based on their own criteria and previous developments.*

This means that if a company possess a technology that fits an RFP, the company can send the specification of that technology as a proposal to the OMG and it has a good chance of getting it approved as an OMG specification. This has been the case of UML proposed by Rational or the Fault-tolerance specification proposed by Sun.

If there are several proposals, the different submitters try to find a consensus and deliver a single, consolidated version, supported by all them. This is usually called a Joint Revised Submission.

5. CORBA ASPECTS FOR REAL-TIME CONTROL

Apart from the importance of having a platform for integration and development of modularized controllers, there are some new issues in CORBA that are specially relevant for distributed control systems engineering. These issues are: predictable behavior, fault tolerance and embeddability.

The Real-time PSIG [9] (Platform Special Interest Group) is addressing all these topics because they have focused their activities on real-time systems, and most real-time systems are also embedded and have some fault tolerance requirements.

[9] They can be found at `http://www.omg.org/realtime/`.

The Real-time PSIG goal is the recommendation of adoption of technologies that can ensure that OMG specifications enable the development of real-time ORBs and applications.

To achieve this goal, the Real-time PSIG gathers real-time requirements from industry, organize workshops and other activities and involve real-time technology manufactures to elaborate Requests For Information and Requests For Proposals for these technologies.

The main results of this work an be organized in the three categories:

Real-time CORBA: The Real-Time CORBA specification in addition to the Messaging specification provides mechanisms for controlling resource usage to enhance application predictability (OMG, 1999*c*; OMG, 1998*b*).

Fault-tolerant CORBA: The specification provides mechanisms for fault tolerance based on entity redundancy (OMG, 1999*b*).

Minimum CORBA: Addresses the construction of CORBA applications on systems with little resources like embedded computers (OMG, 1999*c*; OMG, 1999*a*; OMG, 1998*b*). This specification eliminates most dynamical interfaces that are not necessary in frozen applications (most embedded applications are ROMmed applications).

5.1 *Real-time CORBA*

RT-CORBA standardizes the the mechanisms for resource control (memory, processes, priorities, threads, protocols, bandwidth, etc.) and handling of priorities in a distributed sense (for example forwarding client priorities to the server).

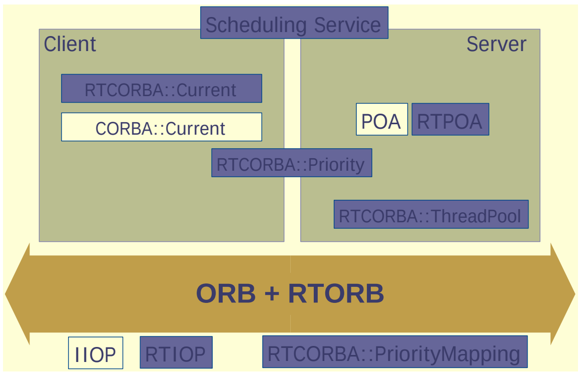


Fig. 7. Real-time CORBA extensions to provide strong control of resources to both clients and servers.

Using these mechanisms, the Real-time CORBA developer can control:

- Request time-outs
- Resource allocation and sharing
- Priority control and propagation

- Priority inversion
- Method invocation blocking
- Routing
- Transport selection

5.2 *Fault-tolerant CORBA*

Fault-tolerant CORBA tries to enhance application fault tolerance reducing to a minimum the impact to the application (computing overheads and increase of complexity). Fault tolerance is increased by means of entity replication: cold passive replication, warm passive replication, active replication or active replication and majority voting.

5.3 *Minimum CORBA*

Embedded CORBA applications reduce memory footprint by means of elimination of some features (dynamic interfaces and repositories), the use of standardized operating system services or special transports. The elimination of a specific service from the specification does not mean that the application cannot use it, only that it will not be necessarily provided by a compliant CORBA implementation.

5.4 *Bridging Domains*

While the Minimum CORBA specification reduces the requirements posed to the ORB, the *Real-time CORBA* and *Fault Tolerant CORBA* specifications can increase the size an complexity of the application.

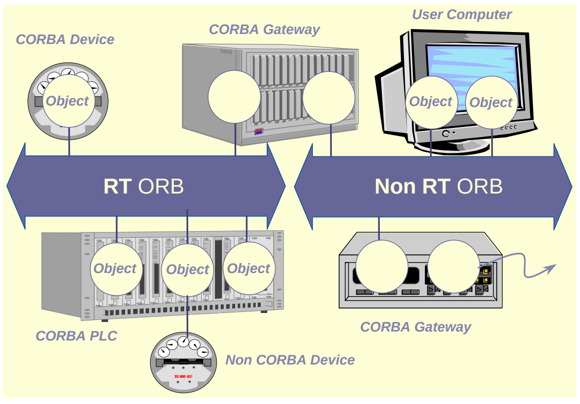


Fig. 8. A CORBA Gateway can bridge between two worlds of different protocols or ORBs.

Thanks to interoperability, it is not necessary at all to have all the application running atop he same ORB. It is possible to have the critical part of an application running over a Real-time ORB and the rest over a more conventional one. Figure 8 shows an example of using a CORBA gateway to bridge between two different worlds in a control application.

6. APPLICATION EXAMPLES

In this section I will present succinctly some examples on the use of CORBA technology in control systems. The broker used is all of them is the ICa Broker(Sanz *et al.*, 1999*b*), a broker specially tailored for control applications that was developed by us during the ESPRIT DIXIT project [10] .

6.1 *Robot Teleoperation*

As a laboratory experiment we have used CORBA to build a robot teleoperation application (see Figure 9. The application contains three CORBA objects: a six DoF [11] full force feedback master, a seven DoF robot slave and a coordinate space mapper (transforms master axis space into robot axis space).

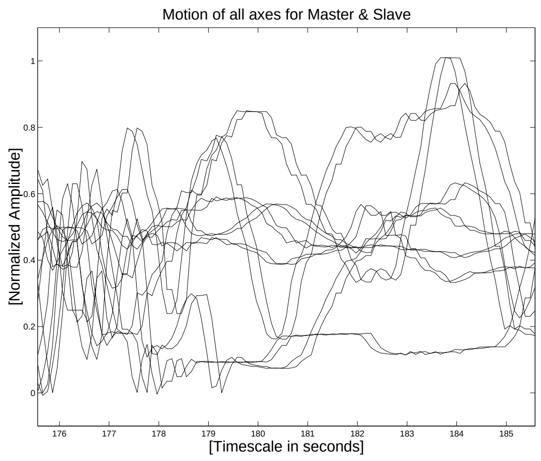


Fig. 9. Axis position evolution in master and robot during a test.

A big delay is appreciated ($\approx$ 250 ms) but with a small jitter. The test was done using the common laboratory 10baseT network in normal state (about 20% load).

6.2 *Risk Management*

Another application of interest is RiskMan. This is a system for emergency management in a chemical complex with nine plants (see Figure 10). The system supports the whole life-cycle of emergencies: prevention, detection, firing, diagnosis, handling, follow-up & cancellation.

[10] Now it is a commercial product distributed by an UPM spin-off company called SCILabs.
[11] Degrees of Freedom.

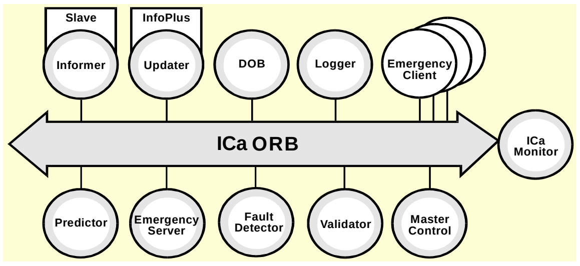


Fig. 10. Some of the CORBA objects that compose the RiskMan application. Informer and Updater are wrappers of external systems.

The application is composed by a collection of CORBA objects running on heterogeneous platforms (VAX/VMS, Alpha/UNIX, x86/Windows NT) performing an heterogeneous collection of functions: expert systems, user interfaces, wrappers of real-time plant databases, data filters based on fuzzy rules, predictors based on neural networks, etc.

6.3 *HydraVision*

HydraVision is a real-time video system for the support of remote operation of hydraulic power plants. It uses a country-wide fiber optics WAN network of a electric company to integrate a collection of objects that wrap physical entities in the system (see Figure 11).

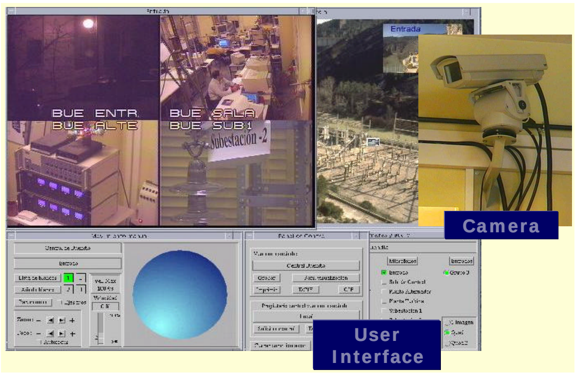


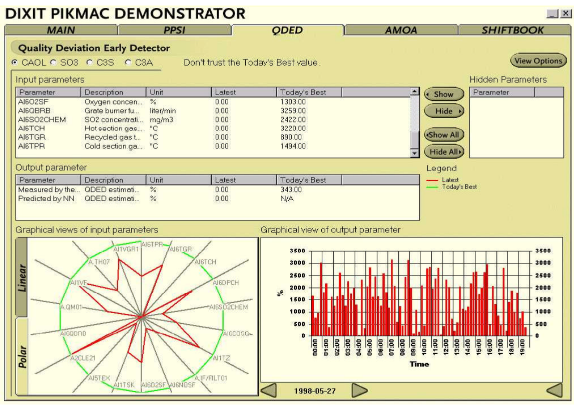
Fig. 11. The HydraVision main user interface and one of the wrapped cameras.

The physical systems that are wrapped as CORBA objects are: cameras, MPEG compressors, image/audio multiplexers, microphones, loudspeakers, video monitors, video stores, still image printers, etc.

The system is supports multicasting and bidirectional streaming. It used by human operators to: get a visual confirmation of the status of the remote plant, video-conference, faking human presence, remote diagnosis, etc..

6.4 *PICMAK*

PIKMAC is an operator support system designed to address plant-wide strategic decision making in a cement plant. The system is used by operators specially in night and weekend shifts when there is only one one person in the plant(see Figure 12).


DIXIT PIKMAC DEMONSTRATOR
MAIN
PPSI
QDED
AMOA
SHIFTBOOK
Quality Deviation Early Detector
View Options
Show
Hide
Show All
Hide All
Legend
Latest
Today's Best
Graphical views of input parameters
Graphical view of output parameter
Linear
Polar
Time
1998-05-27

[✓ CAOL] SO3 C3S C3A Don't trust the Today's Best value.

Input parameters

Parameter	Description	Unit	Latest	Today's Best
AI6O2SF	Oxygen concen...	%	0.00	1303.00
AI6QBRB	Grate burner fu...	liter/min	0.00	3259.00
AI6SO2CHEM	SO2 concentrati...	mg/m3	0.00	2422.00
AI6TCH	Hot section gas...	"C	0.00	3220.00
AI6TGR	Recycled gas t...	"C	0.00	890.00
AI6TPR	Cold section ga...	"C	0.00	1494.00

Hidden Parameters

Parameter

Output parameter

Parameter	Description	Unit	Latest	Today's Best
Measured by the...	QDED estimati...	%	0.00	343.00
Predicted by NN	QDED estimati...	%	0.00	N/A

Fig. 12. Part of the user interface that shows the results of the on-line quality estimator QDED. It uses neural networks to estimate present clinker quality because it is not possible to have an direct real-time measure of it.

The system is composed by a collection of interacting CORBA objects that provide four top level functionalities:

- Clinker quality estimation using neural network technology.
- Instantaneous cost estimator using deep models.
- Alarm management using expert systems.
- Inter-shift communication using multimedia technology.

6.5 *Present work*

Our present work is focused in lowering the requirements posed to the platforms to run CORBA objects. We are working in the development of embedded modular systems based on this technology.

As examples, we are using CORBA in mobile robotics and also in the development of standardized implementations of electric substation automation systems (SAS).

In this last case, we try to build CORBA automation objects embedded in field devices based on the Electric Utilities IEC 61850 Draft standard. To do this work, we have funding from the European Comission through the IST DOTS project.

7. STATE AND FUTURE OF THE TECHNOLOGY

CORBA technology is impressive but perhaps too impressive for normal control systems developers. It suffers what is called a second system effect, trying to address all possible functionality or requirement. We must identify our own needs and determine if the CORBA way fits our needs. If not, we are still in time to modify it.

Perhaps the main question is *Why we need integration ?*. Beyond many obvious answers (to build TotalPlants, to achieve total safety, to be the first in the market, to spend less money, etc.) I would like to stress one door that this approach opens for us: The modular approach fostered by CORBA will let us develop true modular control systems, and this will eventually lead to reach human-like complexity levels in artificial minds. For sure CORBA will not be the integration technology for future C3POs, but it will open that way. If you remember the movie 2010, HAL 9000 goes back to life (or conscience) when Dr. Chandra re-connect the modules that encapsulate high-level mental functions using an integrational backbone.

The second point I want to mention is *design freedom*. Design freedom is necessary in the complex control systems domain to explore alternative controller designs. Excessively restrictive technologies will collapse –unnecessarily– dimensions of the controller design space (Shaw and Garlan, 1996). This is, for example, the case of some fieldbus technologies that support several slaves but only *one* master. While design restrictions (in the form of prerequisite design decisions) simplify development they sacrifice flexibility.

Can we get both, simple development and flexibility ? The key are no-compromises frameworks, *i.e.*frameworks where design dimensions are still open even when pre-built designs are available. To continue the example of the fieldbus, the one-master/several-slaves approach is one type of pre-built, directly usable, design; but the underlying field bus mechanism should allow for alternative, multi-master designs. This can be done by means of the development of agent libraries that provide predefined partial designs in the form of design patterns (Sanz *et al.*, 1999*c*), and a transparent object-oriented real-time middleware.

This approach will let developers construct their own agencies to support their own designs because it is impossible to fight the *not-invented-here* syndrome. Let the people do what they think they need. Do not define *ultimate* solutions. Provide reusable assets that can be adapted to any problem in a domain progressively focused (Sanz *et al.*, 1999*a*).

8. CONCLUSIONS

Software technology is of extreme relevance in any area of engineering activity. In the case of automatic control systems, we can say that it is not only relevant but a mandatory technology in a wide variety of control system implementations. Control engineers must know more about software because it is as basic as differential equations for the proper construction of control systems.

While there are many research developments in DOC, three are the main contenders in the DOC wars: COM, CORBA and Java. But it is pretty clear that the only widely available technology that is addressing -more or less- the full range of topics in our automatic control business is CORBA. The three main objectives we are searching in a software technology are embedded, real-time and robust. All they are being addressed by CORBA specifications: Minimum CORBA, Real-time CORBA, CORBA Messaging and Fault-Tolerant CORBA. It is worth note however that most commercial products are ignoring the real-time market because they think it is a very small market.

It should be clear however, that selection of any one of these technologies does not hamper the application of the others. In fact, in relation with Java and CORBA, it is worthy note that both distributed object models are pretty the same, and evolution of distribution for Java applications is being fostered over CORBA compliant brokers. On the other side, CORBA interoperability specifies mappings to COM and OLE Automation, making possible the integration of COM based applications in broad-class CORBA systems.

While CORBA has been developed for distributed applications, the transparent integration mechanism it offers serves also for non-distributed applications. Some broker implementations provide local transports that do not use network protocols and hence do not induce a big overhead. There are even broker implementations that can reduce overhead for local invocations to zero (Sanz *et al.*, 1999*b*). This means that the programmer can transparently decide where to put the objects and CORBA can do it with a minimal impact in performance.

Real-time CORBA is very new (we have only release 1.0 and some errata corrections) and it has been developed as a compromise usable in many fields and hence it has only a fixed priority model (instead of other dynamic priority models better suited for control applications). Dynamic scheduling service will appear eventually but it is suffering a painful specification process.

We should mention the strong bias in RT-CORBA to telecoms, that make it sacrifice strong predictability. Real-time ORBs are being deprecated by mainstream ORB vendors and hard real-time ORBs are far in the future. Next major issues for our business will be pluggable transports (that are not politically correct because they are not IIOP) and real-time services; like the mentioned scheduling service, real-time events or transactions. Some of them have been demonstrated by OMG contributors. Other relevant issues are combinations of specifications; for example Real-time + Fault-tolerant or Real-time + Minimum. Stay tuned.

Remember: Ignoring software topics in control systems research is a big mistake. Not big, but critical for the discipline. Control engineering is not only a discipline of mathematical modeling and differential equation solving. Control engineering is the discipline of *artificial behavior* and software is what makes the behavior. CORBA is a good tool to support our designs, but we must work hard to make CORBA more oriented towards control systems engineering.

9. REFERENCES

Åström, Karl Johan and Björn Wittenmark (1997). *Computer Controlled Systems*. third ed.. Prentice-Hall. New York, NJ.

Blanke, Mogens, Christian Frei, Franta Kraus, Ron J. Patton and Marcel Staroswiecki (2000). Fault tolerant control systems. In: *Control of Complex Systems* (Karl Åström, Alberto Isidori, Pedro Albertos, Mogens Blanke, Walter Schaufelberger and Ricardo Sanz, Eds.). Springer. In Press.

Brooks, Fred (1992). No silver bullet. *Computer*.

Gupta, M.M. and N.K. Singh (1996). *Intelligent Control Systems*. IEEE Press. Piscataway, NJ.

OMG (1998*a*). Common object request broker architecture and specification. Technical Report 2.3. Object Management Group.

OMG (1998*b*). Corba messaging. Technical Report `orbos/98-05-05`. Object Management Group.

OMG (1998*c*). A discussion of the object management architecture. Technical report. Object Management Group.

OMG (1999*a*). Errata for real-time corba joint revised submission. Technical Report `orbos/99-03-29`. Object Management Group.

OMG (1999*b*). Fault tolerant corba. Technical Report `orbos/99-10-05`. Object Management Group.

OMG (1999*c*). Real-time corba. Technical Report `orbos/99-02-12`. Object Management Group.

Rodríguez, Manuel and Ricardo Sanz (1999). HOMME: A modeling environment to handle complexity. In: *IASTED Modeling and Simulation Conference*.

Rushby, John (1999). Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical Report NASA/CR-1999-209347. NASA.

Samad, Tariq (1998). Complexity management: Multidisciplinary perspectives on automation and control. Technical Report CON-R98-001. Honeywell Technology Center. Minneapolis, MI.

Samad, Tariq and Weyrauch, John, Eds.) (2000). *Automation, Control, and Complexity: New Developments and Directions*. John Wiley and Sons.

Sanz, Ricardo (2000). *Agents for Complex Control Systems*. in Samad and Weyrauch (2000).

Sanz, Ricardo, Fernando Matía and Santos Galán (2000). Fridges, elephants and the meaning of autonomy and intelligence. In: *Proceedings of ISIC'2000*. Patras, Greece.

Sanz, Ricardo, Idoia Alarcón, Miguel J. Segarra, Angel de Antonio and José A. Clavijo (1999*a*). Progressive domain focalization in intelligent control systems. *Control Engineering Practice* **7**(5), 665–671.

Sanz, Ricardo, Miguel J. Segarra, Angel de Antonio and José A. Clavijo (1999*b*). ICa: Middleware for intelligent process control. In: *International Symposium on Intelligent Control*. Cambridge, MA.

Sanz, Ricardo, Miguel J. Segarra, Angel de Antonio, Fernando Matía, Agustín Jiménez and Ramón Galán (1999*c*). Patterns in intelligent control systems. In: *Proceedings of IFAC 14th World Congress*. Beijing, China.

Shaw, Mary and David Garlan (1996). *Software Architecture. An Emerging Discipline*. Prentice-Hall. Upper Saddle River, NJ.

Shokri, Eltefaat and Phillip Sheu (2000). Real-time distributed object computing: An emerging field. *IEEE Computer* pp. 45–46.

Siegel, Jon (2000). *CORBA 3: Fundamentals and Programming*. second ed.. OMG Press/Wiley. New York.