CORBA-BASED COMMUNICATIONS ARCHITECTURES FOR RESOURCE HOLONS IN HMS

BY

GABRIELA VARVARA

Abstract. The Holonic Manufacturing Systems (HMS) represent a pragmatic solution to model and design modern production processes intended to be flexible in terms of the volume and the diversity of the resulting consumer goods and adaptable to the availability of the processing resources. In order to increase the efficiency of HMS, the communication aspects like interoperability and reliability inside a genuine distributed system have to be carefully considered. This paper details the design aspects of a distributed service-oriented architecture based on CORBA standards intended to enhance the cooperation during the holarchies formation in a PROSA based architecture. It presents the communication architectural view and the basic rules to define the IDL generic interfaces and a client-server Java application that integrates a control system developed in JACK® multiagent environment for the resource holons.

Key words: CORBA, distributed programming, communication architecture, network robotics, holonic control.

2000 Mathematics Subject Classification: 68T42, 93C95.

1. Introduction

The modern manufacturing systems have to adapt to the rapid variations of the flow, volume and diversity of the customers’ demands in the current trend of globalization of production and sale processes. Features like flexibility and adaptability are critical and have to be provided at all the levels of the decision parts of the system. In order to achieve this objective, a new paradigm for the analysis and design was adopted namely the holon. As defined in [1], [2], the holon is an autonomous, cooperative, self-content entity that provide a common base to model hybrid systems including the manufacturing ones. The autonomy refers to the ability to identify and adopt strategies and plans and to control their
execution. If these ones are mutually acceptable for a group of holons, the cooperation has to be invoked. It will generate an aggregation process resulting in holarchies as a rational limit of the holons. On the other side, the same holarchy is built on the basis of the recursive structure of the holons that allows both horizontal and vertical aggregation. Finally, the self-similar internal structure provides for the holons of the same type the same behavior and communication interfaces. The HMS PROSA architecture is entirely based on the holon paradigm and contains at least three main holons namely:

- order holon (OH) to model the management of the manufacturing tasks in order to perform them correctly and on time;
- product holon (PH) to hold and process product knowledge to comply with the fabrication process and achieve sufficient quality for the results;
- resource holon (RH) that control the physical part of a production resource ranging from a factory or a shop floor to raw materials, tools personnel or even energy flow.

The basic elements of the reference architecture can be extended with additional staff holon that provides information and expertise to the above mentioned decision holons.

According to the PROSA model, there are three different kinds of relations among the above mentioned holon entities, namely:

- the aggregation to represent the connections inside a hierarchy of holons that are clustered together. It represents an appropriate solution for the management of the HMS complexity;
- the specialization relation intended to introduce more pragmatic differentiations inside the order, product or resource abstract holon categories. For instance, order holons are generic representation for both customer orders and maintenance tasks;
- the association to model the relations developed among holons, possible of different types, during a concrete manufacturing process. The relationships developed among the holons involved in an assembling manufacturing activity represent a good example.

It can be noticed that the structure elements of the architecture are separated from the decision entities. The control can be both hierarchical and heterarchical, the hemi-heterarchical schemes being used frequently. In order to implement this decisional part of the system with no alteration of any holon characteristic a multiagent programming solution is recommended [3]. The agent, as a software entity, will process the information inside the control system. Meanwhile, agent architectures, based on object-oriented paradigm, are limited in supporting the holon recursive structure, its affiliation to multiple or temporary groups or even specific time constraints or reliability specifications. According to [3], the solution was to consider the holon as a composite entity,
possibly distributed, that includes a software agent as the deliberative component and a structural recursive part with operational role. Consequently, the structural component can be either a physical device or an entire holarchy as depicted in Fig. 1. It exchanges information with the software agent through a communication interface. For a resource holon this internal architecture allows the integration of the physical device controller that cannot run complex applications, with the external software agent.

![Fig. 1 – The holonic system architecture.](image)

Referring to the implementation strategy, the presented solution adopted the JACK® multiagent platform that brings important benefits [4] such as a deliberative BDI agent architecture, a complete event driven execution model and a high level communication middleware necessary to handle the inter-agent information exchange during the inference mechanism. On the other hand, the global main characteristics of the HMS must be considered: the inherent distribution, the variable complexity, the unbalanced loading, or the hybrid nature. Consequently, the communication intra/inter agent has to be distributed, service-based, upon a client/server paradigm with a high degree of interoperability. This paper proposes a CORBA (Common Object Request Broker Architecture) based solution. This standard brings all the benefits required by the holonic integration. Further it has an open specification based on an architecture that does not depend on the vendor or the execution infrastructure. The goal is to implement robotic resource holon which can overcome the shortcomings of other components of a HMS. CORBA uses an ORB (Object Request Broker) as the middleware that establishes client-server relationships at the object level. It facilitates the invocation of a server object method across the network transparently without any knowledge about the
server location, programming language or operating system. CORBA-based systems allow interoperation between client and server objects on a large scale through services invocation. Referring to HMS that comprises different multiagent control systems that have to interoperate in a reliable manner, the CORBA middleware is able to offer build-in support for communication interfaces. For the proposed HMS a Java IDL (Interface Definition Language) development process was adopted, that offers a uniform framework for the control implementation made in JACK as a superset of Java and the communication programming. To test the developed communication architecture, an experimental manufacturing system was considered, consisting of two robotized cells, created around two 6 d.o.f industrial robots able to operate with a machine tool, a conveyor, a computer vision system and several storage devices. Simple assembling manufacturing scenarios were solved by making use of this equipment.

The rest of the paper is organized in 5 sections. Section II makes an overview of the main CORBA architectural components that will be further used in the proposed solution. Section III integrates the CORBA event services into the holonic communication model. Section IV defines the basic trader mechanism necessary for locating appropriate object services across a distributed holonic structure. Sections V is devoted to server/client side object distribution guidelines for the case of a robot resource holon and the physical execution part driver. Section VI concludes the paper.

2. The Architecture of the CORBA Based Communication Application

CORBA is a technical communication infrastructure for applications build upon distributed objects. It enables object-oriented RPC (Remote Procedure Call) through a few main constituent parts namely [5]:

- ORB (Object Request Broker) is the distributed service that implements the request to the remote object. The same request mechanism is used by the client and the CORBA object regardless of where the object is located with no restriction to certain implementation language. ORB is responsible for all the necessary translation between the programming languages. This kind of independence requires that for each computer architecture and operating system from the network at least a CORBA 2.0 compliant product exists (this version introduced the Interoperable Object References) and for all the programming languages the IDL mapping was defined.

- The Object Adapter – represents the connection between the CORBA objects specified by means of IDL and the proper object implementation. When the client-side ORB receives a client request, it routes this one on the server-side ORB that further transmits the request to an object adapter. The last one
has to determine to which object implementation the request has to be forwarded. During a concrete implementation of a distributed object one has to decide on a certain object adapter. Since the interface to the ORB core depends on the core, to create portable objects from one ORB to another, the new ORB requires the same object adapter.

- The IDL (Interface Definition Language) is a dedicated language specified by the OMG (Object Management Group) used according to [6] to define the object interfaces in a unified manner. The object IDL interface is programming language neutral. It indicates the operations the object supports and not how they are implemented. Consequently, IDL do not specify algorithms or object states. It represents only a contract between the code using the object (client) and the code implementing the object (server). IDL defines language bindings for different programming languages. It is used to describe modularized object interfaces, operations and attributes that an object supports, exceptions raised by an operation, data type of an operation return value and its parameters. An IDL compiler for every CORBA-compliant programming language exists; it converts the IDL code to stub and skeleton source files.

- The IR (Interface Repository) is a CORBA server used to store the metadata from the CORBA objects interface definitions. This information will be accessed at run-time in order to generate dynamic invocation using the Dynamic Skeleton Interface from CORBA 2.0. The IR can be located anywhere in the network, is a stand-alone component not available for all ORB products and has to be activated separately from the client and server processes. It is described as any CORBA object by a set of IDL interfaces.

- The Dynamic Skeleton Interface replace for CORBA version 2.0 and higher the Dynamic Invocation Interface and provides a run-time mechanism for server-side management of components whose IDL definition are not known during the implementation and compilation of the server. While the static invocation of the server operations are made easily through static interfaces generated by the IDL compiler, the dynamic method invocation must be generated by the programmer himself via the Dynamic Skeleton Interface. This interface supports synchronous communication modes initiated by the operation invoke(), deferred-synchronous communication initiated by the operations send() and get_response()/pull_response() and asynchronous communication initiated by the operation send_oneway(). In order to handle certain class of requests at run-time the server uses an ORB component named skeleton that analyses incoming messages with respect to target object and method and tries to determine the receivers of the messages.

- The Implementation Repository (IR) administrates different server implementations. It activates automatically the server code at the start of the client. Exceptions are the persistent servers that have to be started manually and
afterwards, can permanently wait for client requests. One drawback in developing IR based communications is that the vendor will decide whether and how the repository implementation will be made. As a consequence, the developers have to deal with the problem of possible crashed servers.

- The Servant refers to an invocation target containing methods for handling the remote method invocations. It represents a CORBA object in the server application. In the object oriented languages, both the remote CORBA Object and its Servant are objects from the viewpoint of the object oriented programming, the last one being written in an IDL supported language.

- The Client is a program entity that transparently invokes an operation on a server object implementation through an appropriate servant.

- The Stubs and the Skeletons serve as a connector between the client and server applications, respectively, and the ORB. They are generated by the IDL compiler. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler that reduces the potential for inconsistencies and increases opportunities for automated optimizations.

The developed application uses all of the basic CORBA components, with the remark that the dynamic implementation is just generic. It will be subsequently developed depending on the need to introduce this mechanism for individual cases. The communication application architecture is represented for both the client and the server in the following diagram:

Fig. 2 – The architecture of the CORBA based distributed communication application.
The next paragraphs will detail two specific architectural adaptations for holonic cases namely the Event Service to assure the event driven characteristics and the Trade Service for the service-oriented design.

3. The CORBA Event Service Integration. Event Channels Design

The CORBA Event Service allows asynchronous communication mechanisms to invoke operations on remote objects. The implemented event based communication model has the following characteristics:
- The supplier and consumer event messages are disengaged from each other;
- The communication is indirect and asynchronous;
- An arbitrary number of event senders distribute event notifications to an arbitrary number of recipients;
- Senders and recipients do not know their location or the IDL interfaces location during the communicating process.

To obtain these features the application was designed by means of a mediator type template. It works as an event channel and it is implemented by an object placed between the sender and the receiver, as depicted below:

![Fig. 3 – The CORBA Event service architectural concept.](image)

The communication through the event channel is indirect, asynchronous and anonymous. As a remark, the event-based communication is less effective than the classical client/server model. The channel produces a variety of events that cannot be filtered despite the fact they are not equally important for the recipient or cannot be interpreted. Because a mediator based weak coupling between sender and recipient is useful when the supplier cannot wait for the processing of sent messages, or a large number of recipients are served, the mechanism is used frequently. An event-based system is easier to spread than a classical client/server one. As a result it is recommended for the holonic based production management applications that have to integrate a large number of legacy and next-generation systems.
For the proposed holonic scenario as a complete event driven system, the suppliers/receivers, as depicted in Fig. 3, can be both physical devices of the resource holons or software JACK® agents. The physical devices will transmit their states to the coordinating agents and receive from them the action commands. This mechanism can be further used for agents’ communication during the interactions between different multiagent systems.

Referring to the communication process, there are several transmission-reception models. They differ by the involvement of communication objects during a distributed connection, as illustrated by the following diagram:

Fig. 4 – The CORBA Event Service transmission models.

The supplier and the receiver never interact directly. The supplier communicates with an event consumer proxy created on the provider host, and the receiver is connected to a supplier proxy defined on the consumer host. The manner in which messages are transmitted internally in the event channel is completely transparent to the user and structured in four different styles:

- Push style where the supplier actively pushes the data to the consumer proxy.
- Pull style where the consumer will play the active role in absorbing the messages from the event channel.
- Two mixed push/pull styles where a push supplier will be connected indirectly to pull consumer and vice versa.

There is also the possibility to transmit events using different model connections among several suppliers and more than one consumer all connected to a single channel. The service is designed so that programmers have a minimum intervention in this area.
The implementation of the communication application related to the Event Service starts with the IDL interfaces definition for both supplier/receiver side. After compilation the following class relationship structure will be generated:

![Diagram of class relationships](image)

Fig. 5 – The view of participating classes during the Event Service analysis phase (UML view obtained with Software Architect®).

The proxy objects from the class diagram inherit the supplier/consumer interfaces (arrows with solid line) being their perfect substitute in the communication process. The dotted arrows represent the circumstantial dependence relationships between classes in the sense that input parameters and/or returned values from different methods of a class refer directly to object references of the related class.

For the proposed holonic system the inter-agent communication is designed according to JACK® platform communication paradigm. In order to transmit the physical devices status but also the issued agent orders different hardware/software components have to interact. For this reason a solution based on the CORBA standard related to event management becomes effective. The communication mechanism used took account of the active role of the event provider specific to a holonic control system. Consequently a pure push style was adopted and it was implemented by the classes with gray hatchings from the Fig. 5.

For a push communication, the PushConsumer interface is implemented so that the consumer is receiving data actively transmitted by
the supplier. Thus, as depicted the diagram in Fig. 5, the provider proxy ProxyPushSupplier will communicate data by calling the push() method from consumer class that can receive data of any format. The disconnect_push_consumer() method has to be called by the provider whenever it wants the completion of the communication process and the release of the customer resources. Any subsequent event transmission will determine the proxy on the provider side to throw an exception object of type Disconnected, indicating that consumer is not available for new events.

On the provider side, the consumer proxy is represented by ProxyPushConsumer class that communicates with PushSupplier interface. The only call that has to be implemented by the programmer is disconnect_push_supplier(), as depicted in Fig. 5. It will cause the communication to breakdown and the release on the provider resources. As the supplier is the active part that will call the transmission operation on the consumer proxy, no further operation is necessary any longer.

It can be easily noticed that the events channel is the operational core of the Event Service. Its management is ensured through AdminConsumer objects, respectively SupplierAdmin. The IDL specification of their interfaces will be included in a single module called CosEventChannelAdmin.

In order to establish a provider-client connection and to forward events through the communication channel five steps are necessary for any chosen communication model:

1. It is created and delivered an object of EventChannel type. This process dependents on CORBA vendor and should be done to allow the provider and the consumer to be connected.

2. The call of for_suppliers() method from the object EventChannel to create an administration Supplier type object. Similarly, a ConsumerAdmin type object will be delivered to the consumer as result of for_consumers() operation call.

3. The providers and the consumers will get the appropriate proxy objects from Admin objects. For a push communication model, the supplier will call the obtain_push_consumer() function from SupplierAdmin class to create ProxyPushConsumer object. Similarly, the consumer will make the call obtain_push_supplier() from ConsumerAdmin class to get a ProxyPushSupplier object.

4. The suppliers/consumers must be added to the events channel by connecting calls from the proxy classes. These will create links between providers/consumers and their consumer/provider proxies.

5. The events are submitted through push()/pull()/try_pull() calls depending on chosen communication model.

Moreover, the destroy() operation from the EventChannel object will
destroy the communication channel when it is no longer necessary. As a result, the attached administrative objects will also be destroyed.

The IDL specifications for the proxy interfaces will be grouped into CosEventChannelAdmin module that also provides two additional exception objects: AlreadyConnected to when a connection attempt is made for an already connected proxy and TypeError that is relevant in order to detect a type mismatch for the applications that specify the event type.

The proxy objects work remotely on behalf of the supplier/customer and support push(), pull() and try_pull() operations on the opposite side of the channel they are working on.

For IDL proxy interfaces the definition of connect type operations, as described in the fourth step of the above procedure, is representative. For example, the operation connect_push_supplier() allows the connection of a push supplier to the proxy of a push consumer.

The exchange of messages for a normal communication startup process is illustrated in Fig. 6 for the provider side and in Fig. 7 for the consumer side.

In order to design the communication part for the proposed holonic control system it has to be considered that both the driver of the resource execution part and the software control agent can be equally supplier and consumer. The execution part receives commands from the JACK® agent and sends states and the results of executions. Consequently, all the connections for data input/output have to be created and activated.

![Fig. 6 – The startup of the communication process with the events channel (supplier side).](image-url)
Fig. 7 – The startup of the communication process with the events channel (consumer side).

4. The CORBA Trade Service Integration. Service-Oriented Implementation

For a distributed environment, the interactions generated by the CORBA Trade Service are based on the client/server paradigm as outlined in the next architectural view:

Fig. 8 – The Trade Service mechanism according to CORBA standard.

As depicted in Fig. 8, the server starts the trading process by registering in a specified repository the service types that could be offered in the future. For
a particular service the stored data refer to the names and properties. In the second step the server offer will be exported as object reference to the Trader Service. Then the server will wait for a client request. On the other side, the client will query the Trader Service any time it is looking for a certain object reference that was previously registered by the server. After an appropriate processing, the client will use this reference to directly access the server and benefit from its service.

For the integration of CORBA Trading facilities, the Java code was implemented in four successive steps:

1. The connection of the server to the Trader Service:

```java
//a generic reference for a CORBA object was created
org.omg.CORBA.Object obj=orb.resolve_initial_references(“TradingServer”);
//and converted to Java reference
if(obj!=null){
    org.omg.CosTrading.Lookup trader =
    org.omg.CosTrading.LookupHelper.narrow(obj);
}
```

For the assumed holonic system five basic services where identified:

a) the robot – product transfer, product assembling;
b) the machine tool – raw part processing;
c) the conveyor – part/product transfer;
d) the storage device – part/product location, part/product identifying;
e) the computer vision system.– raw checking.

Each identified service can be offered by an arbitrary number of the above mentioned holons. In order to become effective, these services have to be first registered. Then the servers will declare the types of offer objects that will be instantiated the next step.

2. Adding a new service to the Trader:

```java
//obtaining the generic reference of the Trader service
org.omg.CORBA.Object obj=trader.type_repos();
//the conversion to a real reference
org.omg.CosTradingRepos.ServiceTypeRepository trader_repos_obj=
        org.omg.CosTradingRepos.ServiceTypeRepository Helper.narrow(obj);
//defining the properties of the registered service
prop[0]=new org.omg.CosTradingRepos.ServiceTypeRepository.PropStruct();
prop[0].name=“name”;// the name property of the registered service
prop[0].value_type=orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string);
//the name property can only be readed
prop[0].mode=PropertyMode.PROP_MANDATORY_READONLY;
prop[1].name=“location”;// defining the location property
prop[1].value_type=orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string);
```
// the location is a mandatory property
propr[1].mode=PropertyMode.PROPERTYMANDATORY;
// add_type() calling will produce the service registration
add_type(   // add_type call with the current type parameters
    "Product transfer", // service type
    "IDL:TraderApp/ Product transfer", // the IDL name of the type
    propr,     // offered properties
    superTypes);    // there are no parent types)

The Trader Service repository will manage all the offered object references.

3. Exporting the service offers:
Whenever a server wants to make some offers available, it will register
them in the Trader Service through an export offer process. For the "Product
Transfer" service the Java code becomes:

// the server of the robot equipment driver will instantiate the Product Transfer service
DriverRobot1_Impl r1_impl=new DriverRobot1_Impl();
DriverRobot1 r1=r1_impl._this(orb);
// DriverRobot1 connects to the Trader Service as in the first step and receive the
// variable //trader_lookup_var used to access the registration component responsible for
// offer export
org.omg.CORBA.object trader=orb.resolve_initial_reference("TradingService");
org.omg.CosTrading.Register register=lookup.register_if();
// the robot1 server gives the offer properties the current values
propr[0]=new org.omg.CosTrading.Property();
propr[0].name="name";
propr[0].value=orb.create_any();
propr[0].value.insert_string("robot1");
propr[1].name="location";
propr[1].value=orb.create_any();
propr[1].value.insert_string("AB zone");
// the DriverRobot1 server will call the export() method to register the offer
String id=reg.export(DriverRobot1, "Product transfer", propr);

4. Registered services client querying:
As soon as an offer has been exported to the Trader service the clients can
use the lookup interface to invoke the associated service. The implementation
sequence for a basic query where the Trader Service reference is accessible from
the second step, involves the setup of the following offer criteria:

trader query(   // client querying sequence
    "Product transfer", // service type
    "AB zone",     // offer constraint
    "random",      // results sort criteria
    policy,        // standard policy file reference
In a more specialized context, as a holonic scenario with self-organizing of the manufacturing processes/resources, the decisions are based on heuristic elements. It will be necessary to add other negotiation mechanisms as the Contract Network one to long term system optimization.

5. The Application Execution and the Objects Deployment

The CORBA communication application between JACK multiagent environment and the driver of the physical robot execution part for an assembling/transportation manufacturing scenario will start up with the IDL interface definition. The IDL modules compilation will automatically generate the stubs and the skeleton for both client and server side as depicted in Fig. 9. The IDL compiler has to be specific to the working hardware/software platform and to the object implementation as depicted in Fig. 2. As a direct consequence the number of generated files differ from one platform to another. For the proposed system the agents and driver for the robot execution part were created in Java or an equivalent superset and the access to the controller of the physical equipment made through a Windows service already existent in the communication infrastructure of the laboratory ABB robots and accessible through a specialized InterLink module. It is worthwhile to notice that after compilation on the client side the server object will be represented by client side implementation references despite the fact that the servant objects are implemented in possible different language.

![Diagram](image-url)

Fig. 9 – The object deployment for the communication between the robot and its execution part.
The main application class, the stub/skeleton code along with additional classes related to the adopted CORBA services will be structured on both the client and the server side. They will accept the client invocations and will redirect them through ORB. If the methods will be transmitted with call arguments, the stub will assure their correct conversion into an ORB compatible format. If the server object will produce a return value the server proxy object will be responsible for client format conversion. The entire communication system is software. Consequently, it is not relevant where the objects are deployed. The IDL operation realized on the server side will be used as services on the client side. Through the compilation process, the skeleton file will generate by an inheritance process most of the current service implementations.

The ORB functionalities are sufficient to assure the interoperability for no matter hardware/software platform and implementation language of the communicating parts. But, if a distributed application will use two different ORB products, it is necessary to adopt a TCP/IP message exchange protocol with inter-ORB connections as IIOP. For this case a unique IOR (interoperable reference) will be used to uniquely identifying of objects and their address spaces. It will specify a unique profile containing the host address, the port number and the key object identifier. Referring to time visibility, IOR can be either:

- temporary – it is always invalidated at the end of the application and restarted on the later activation.
- persistent (does not imply the persistence of associated CORBA objects) – it is delivered once to the client application. For every subsequent server startup the host address must remain unchanged.

6. Conclusions

The presented paper makes an overview of the main characteristics of a holon based manufacturing model, details the interoperability problems and offer a portable solution for a laboratory holonic scenario applied to resource robot holons.

The dynamic behaviour and the self-organizing property of the holonic systems can be preserved through a CORBA based implementation. This standard offers large sense interoperability with a minimum programming effort.

The adopted client-server communication solution fits the intrinsic control mechanism of the proposed system. Moreover none of the basic properties of the holon, namely autonomy and cooperation is altered by CORBA mechanism.

As the multiagent control system is totally event-driven the paper details a CORBA solution based on the Event Service that integrates with JACK events management for the adopted push transmission model.
In order to adapt the communication to complex heterarchical control mechanism the CORBA trader service was included in the proposed architecture. The use of a unique object identifying mechanism generates simple and robust implementation. For the adopted laboratory setup the communication between the robot agents and their execution parts was effective and made in real-time. Future work will develop the application for the remaining resource holons of the proposed laboratory holonic setup and will make experiments to real manufacturing scenarios.

Acknowledgments. This work was supported by the research project SOFHICOR, Contract no. 11-042/2007.

Received: May 26, 2010

Gheorghe Asachi” Technical University of Iași, Department of Automatic Control and Applied Informatics

e-mail: gvarvara@ac.tuiasi.ro

REFERENCES

Sistemele holonice reprezintă o soluție în modelarea și proiectarea sistemelor moderne de fabricație ce aduce beneficii legate de creșterea flexibilității în raport cu volumul producției, diversitatea produselor obținute, adaptabilitatea și disponibilitatea resurselor de procesare aferente. În vederea creșterii eficacității acestor sisteme, aspecte legate de comunicarea între componente cu structură diferită, precum interoperabilitatea și fiabilitatea în context distribuit trebuie analizate cu atenție.

Lucrarea detaliază aspecte de proiectare ale unei aplicații de comunicare într-un sistem holonic destinat îmbunătățirii procesului de formare a holografurilor de tip PROSA. Soluția adoptată are o arhitectură bazată pe servicii CORBA și elemente generice IDL ce aparțin standardului CORBA, cu respectarea paradigmii de programare client-server. Implementarea a fost realizată în mediul Java în vederea unei integrări rapide cu aplicația de control a sistemului holonic studiat ce a fost dezvoltată în mediul multiagent JACK®, dar în principiu poate fi ales orice limbaj compatibil CORBA. Niciuna din caracteristicile de bază ale holonului, cum ar fi autonomia sau cooperarea, nu este alterată prin folosirea mecanismelor CORBA în comunicare. Mai mult, deoarece sistemul de control pentru procesul analizat este implementat pe o platformă complet condusă de evenimente, s-au adăugat funcționalităților de bază ale interoperabilității facilități legate de lucrul cu serviciul CORBA de evenimente, pentru care un model push pur a fost ales și implementat. Acestea au fost completeate cu elemente ce defineșc arhitectura serviciului CORBA de schimb, pentru îmbunătățirea mecanismelor de control pentru structuri holonice complexe, de tip heterarhic.

S-au exemplificat rezultatele obținute pentru cazul comunicării dintre holonii resursă de tip robot și driverele echipamentelor de execuție aferente. Soluția obținută este simplă și robustă având la bază un mechanism unic de identificare a obiectelor.