Evaluation of CORBA communication models for the development of a robot control framework

Eric Colon
Unmanned Ground Vehicles Centre
Royal Military Academy
Avenue de la Renaissance 30
B-1000 Brussels, Belgium

ABSTRACT

This paper reports on ongoing work in the specification and development of a modular software control framework for mobile robots. In order to allow flexibility, a systematic analysis of requirements has been conducted and their consequences on the framework architecture are summarised in this paper. We show that the free CORBA implementation library ACE_TAO fulfils the communication requirements of the framework. We present in details the communication models defined by CORBA and compare results obtained by applying each of them on a typical distributed application.

Keywords: software modularity, distributed control, control framework, CORBA

1 INTRODUCTION

Many researchers in robotics are confronted with the same problem: they have at their disposal many excellent algorithms but due to the lack of standard it is almost impossible to easily reuse those bricks into new applications. Existing programs have to be modified, translated, ported, or simply (!) completely rewritten from scratch when changing/updating the robotic platform. What is needed is a software framework that enables agile, flexible, dynamic composition of resources and permits their use in a variety of styles to match present and changing computing needs and platforms. Since a couple of years, some researchers have begun to work in this way.

At the Unmanned Ground Vehicle Centre (UGV-C), we are dealing with command and control applications including tele-monitoring, tele-operation (including shared, traded and supervised control), collaboration (between users), for single and multi-robots (of the same or different models). The present efforts are justified by the fact that most of the tools developed by the research community deals with autonomous robots (MCA, DCA, MIRO, GeNoM,...). Note that we are not dealing with hard real-time control like the Orocos project (www.orocos.org). It addresses higher-level components like planning and user interaction.

In the next section, the requirements of the control framework are summarized and related to the properties of the ACE_TAO library. Afterwards, we examine the communication models defined by the CORBA specifications and relate them to application needs. Finally we conclude with some guidelines about the communication model selection.

2 FRAMEWORK REQUIREMENTS AND CORBA

The requirements for the control framework have been presented in [1]. These requirements have been inferred from a systematic analyse of typical tele-robotic applications [2]. In the first reference we also justify the choice of CORBA and more particularly the ACE_TAO CORBA implementation among other middleware technologies for our future developments. We show here that the communication requirements of the framework are fully compatible with the free CORBA implementation library ACE_TAO.

- Integration of different robotic systems: use of native libraries (C/C++): ACE_TAO is mainly written in C++, which is the de facto language of most robot libraries.

- Concurrent control of several robots: distributed and multi-threaded processes, easy communication and process synchronisation. This is the reason of the existence of ACE.

- Universal GUI, flexible programming language and implementation solution: the choice of ACE_TAO does not restrict the GUI developments. GUI can be based on C++ toolkits or on interpreted language. Furthermore, CORBA communication is straightforward in Java.

- Shared control between several users requires management of access and use policy as well as coordination between control modules: this is not directly linked to the choice of ACE_TAO. However, the network capabilities could facilitate the implementation of such functions.

- Integration of user algorithms: requires run-time configuration capabilities, portability. By modifying the server registration data in the naming/trading services different servers (functionalities) can be selected at run-time.

- Flexibility:
  - Distribution: This is the core function of CORBA.
  - Modularity: by using Object Oriented programming and interfaces using the Interface Definition Language, capabilities can be divided among several components.
  - Configurability: the naming/trading services could contribute to solve the configurability issues.
  - Portability: ACE_TAO is a universal library that can be used on almost all platforms.
  - Scalability: ACE_TAO provides many components suited for applications involving many processes.
  - Maintainability: ACE_TAO is based on many standard design-patterns. CORBA is an industry standard.

- Performance and efficiency: ACE_TAO has been developed for time-critical applications used in aviation and medicine. Many years of development and improvement have lead to very efficient software implementation. ACE_TAO provides many RT components (RT Event communication, RT-CORBA,...).

3 CORBA COMMUNICATION MODELS

CORBA offers different methods to implement communication and data transfer between objects. The basic communication models provided by CORBA are synchronous two-way, one-way and deferred synchronous[1]. To alleviate some drawbacks of these models Asynchronous Method Invocation (AMI) has been introduced. The Event Service and the Notification Service provide additional communication solutions. The following of this section briefly describes all these models and discusses their benefits and drawbacks.

Synchronous two-way

In this model, a client sends a two-way request to a target object and waits for the object to return the response. The fundamental requirement is that the server must be available to process the client's request.

While it is waiting, the client thread that invoked the request is blocked and cannot perform any other processing. Thus, a single-threaded client can be completely blocked while waiting for a response, which may be unsatisfactory for certain types of performance-constrained applications.

The advantage of this model is that most programmers feel comfortable with it because it conforms to the well-know method-call on local objects.

One-way

A one-way invocation is composed of only a request, with no response. One-way is used to achieve "fire and forget" semantics while taking advantage of CORBA's type checking, marshalling/unmarshalling, and operation demultiplexing features. They can be problematic, however, since application developers are responsible for ensuring end-to-end reliability.

The creators of the first version of CORBA intended ORBs (Object Request Broker) to deliver one-way over unreliable transports and protocols such as the UDP. However, most ORBs implement one-way over TCP, as required by the standard Internet Inter-ORB Protocol (IIOP. This provides reliable delivery and end-to-end flow control. At the TCP level, these features collaborate to suspend a client thread as long as TCP buffers on its associated server are full. Thus, one ways over IIOP are not guaranteed to be non-blocking. Consequently, using one-way may or may not have the desired effect. Furthermore, CORBA states that one-way operations have "best-effort" semantics, which means that an ORB need not guarantee their delivery. Thus, if you need end-to-end delivery guarantees for your one-way requests, you cannot portably rely on one-way semantics.

Deferred synchronous

In this model, a client sends a request to a target object and then continues its own processing. Unlike the way synchronous two-way requests are handled, the client ORB does not explicitly block the calling thread until the response arrives. Instead, the client can later either poll to see if the target object has returned a response, or it can perform a separate blocking call to wait for the response. The deferred synchronous request model can only be used if the requests are invoked using the Dynamic Invocation Interface (DII).

[1] All specification documents over CORBA are available on the OMG web site: http://www.omg.org.

The DII requires programmers to write much more code than the usual method (Static Invocation Interface or SSI). In particular, the DII-based application must build the request incrementally and then explicitly ask the ORB to send it to the target object. In contrast, all of the code needed to build and invoke requests with the SII is hidden from the application in the generated stubs. The increased amount of code required to invoke an operation via the DII yields larger programs that are hard to write and hard to maintain. Moreover, the SII is type-safe because the C++ compiler ensures the correct arguments are passed to the static stubs. Conversely, the DII is not type-safe. Thus, the programmer must make sure to insert the right types into each `Any` or the operation invocation will not succeed.

Of course, if one can't afford to block waiting for responses on two-way calls, he needs to decouple the send and receive operations. Historically, this meant the programmer was stuck using the DII. A key benefit of the CORBA Messaging specification is that it effectively allows deferred synchronous calls using static stubs (automatically generated communication methods hiding CORBA complexities), which alleviates much of the tedium associated with using the DII.

CORBA Messaging

The CORBA Messaging specification introduces the Asynchronous Method Invocation (AMI) model. As we saw in the preceding section, the standard CORBA doesn't define a truly asynchronous method invocation model using the SII. A common workaround for the lack of asynchronous operations is to use separate threads for each two-way operation. However, the complexity of threads makes it hard to develop portable, efficient, and scalable multi-threaded distributed applications. Moreover, since support for multi-threading is inadequately defined in the CORBA specification there is significant diversity among ORB implementations.

Another common workaround to simulate asynchronous behaviour in CORBA is to use one-way operations. For instance, a client can invoke a one-way operation to a target object and pass along an object reference to itself. The target object on the server can then use this object reference to invoke another one-way operation back on the original client. However, this design incurs all the reliability problems with one-way operations described in previous section. To address these issues, CORBA Messaging defines the AMI specification that supports a polling and a callback model. Only the callback model of the CORBA Messaging specification has been implemented in ACE_TAO.

The internal mechanism is actually based on two normal synchronous invocations in both directions. Remarkably, adding asynchrony to the client generally does not require any modifications to the server since the CORBA Messaging specification treats asynchronous invocations as a client-side language mapping issue.

The Events Service

There are many situations where the standard CORBA (a)synchronous request/response model is too restrictive. For instance, clients have to poll the server repeatedly to retrieve the latest data values. Likewise, there is no way for the server to efficiently notify groups of interested clients when data change.

The OMG COS Events Service provides delivery of event data from suppliers to consumers without requiring these participants to know about each other explicitly. A **Supplier** is an entity that produces events, while a **Consumer** is one that receives event notifications and data. The central abstraction in the COS Events Service is the Event Channel, which plays the role of a *mediator* between Consumers and Suppliers and supports decoupled communication between objects. Events are typically represented as messages that contain optional data fields.

Suppliers and Consumers can both play an active or a passive role. A PushSupplier object can actively *push* an event to a passive PushConsumer object. Likewise, a PullSupplier object can passively wait for a PullConsumer object to actively *pull* an event from it.
By combining the different possible roles for consumers and producers, we obtain the four canonical models of component collaboration in the OMG COS Events Service architecture.

While Push type streams are preferred when the supplier/consumer work at the same pace, Pull type streams are best suited when data processing is slower than possible data production or when it is requested at random.

Benefits:

- Producers do not receive callback registration invocations. Therefore, it need not maintain any persistent storage for such registration
- The event channel ensures that each event is distributed to all registered Consumers.
- The symmetry underlying the Events Service model might also be considered as a benefit. It simplifies application development and allows Event channels to be chained together for bridging or filtering purposes.

Drawbacks:

- A complicated consumer registration (multiple interfaces, bi-directional object reference handshake,...),
- The lack of persistence that can lead to events and connectivity information lost,
- The lack of filtering that leads to increased system network utilisation especially when multiple suppliers are involved.

The Notification Service

Two serious limitations of the event channel defined by the OMG Event Service are that it supports no event filtering capability and no ability to be configured to support different qualities of service. Thus, the choice of which consumers connected to a channel receive which events, along with the delivery guarantee that is made to each supplier, is hard-wired into the implementation of the channel. Most Event Service implementations deliver all events sent to a particular channel to all consumers connected to that channel on a best-effort basis.

A primary goal of the Notification Service is to enhance the Event Service by introducing the concepts of filtering, and configurability according to various quality of service requirements. Clients of the Notification Service can subscribe to specific events of interest by associating filter objects with the proxies through which the clients communicate with event channels. These filter objects encapsulate constraints which specify the events the consumer is interested in receiving, enabling the channel to only deliver events to consumers which have

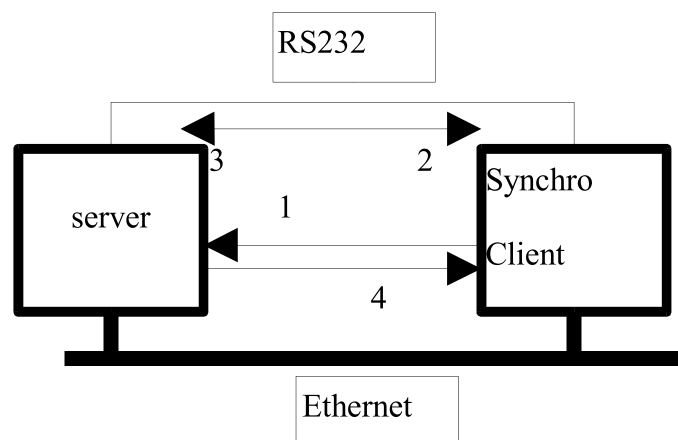
expressed interest in receiving them. Furthermore, the Notification Service enables each channel, each connection, and each message to be configured to support the desired quality of service with respect to delivery guarantee, event aging characteristics, and event prioritisation. The Notification Service attempts to preserve all of the semantics specified for the OMG Event Service, allowing for interoperability between basic Event Service clients and Notification Service clients. The Notification Service supports all of the interfaces and functionality supported by the OMG Event Service.

The TAO implementation does not support Pull interfaces and Typed Event style communication. Work is underway to implement the TAO Real-Time Notification Service. This is an extension to TAO's CORBA Notification Service with Real-Time CORBA support. Dead or unresponsive consumers and suppliers are detected and automatically disconnected from the Notification Service.

4 EVALUATION OF CORBA COMMUNICATION MODELS

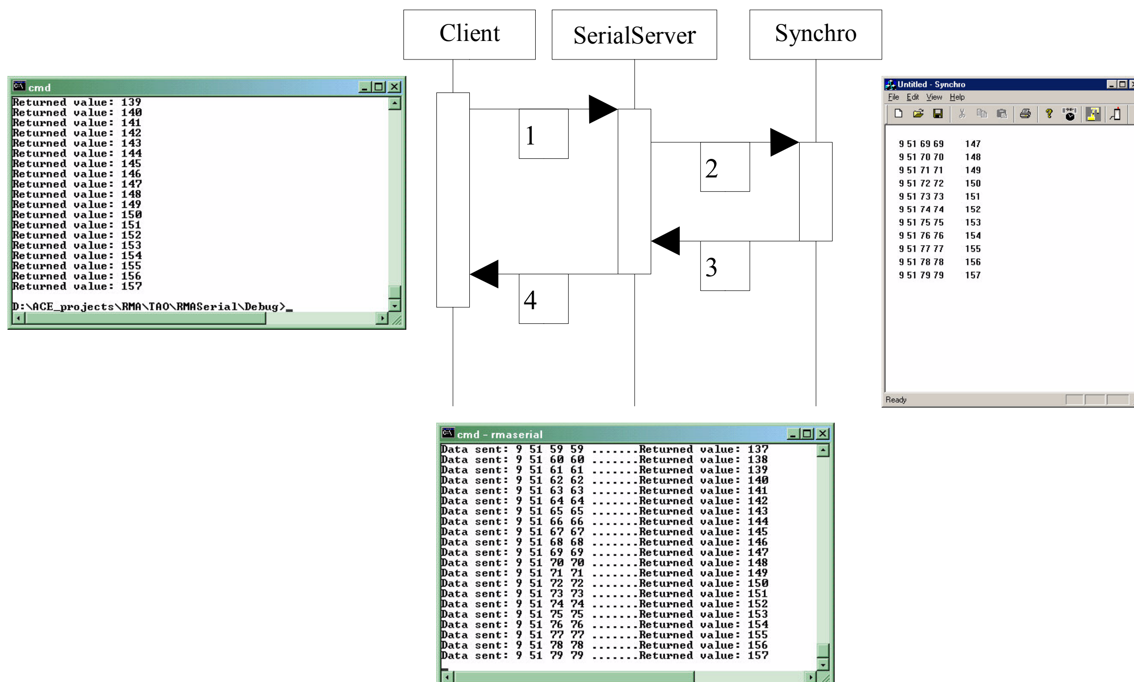
A comparative example using different communication models has been implemented. It provides CORBA wrapping to serial communication.

Standard two-way communication model
A serial server has been developed using the TAO library. A client program reads inputs from the console and sends corresponding commands to the serial server. This server communicates through the serial port to another program simulating a micro-controller controlling a robot (*Synchro*). It reads translation and rotation inputs and adapts these values to the robot kinematics. The right and left speeds values preceded by the command code are sent to the serial server. The next figure illustrates this application.



The time sequence and screen captures are shown in the next figure.

It has been verified that the serial server can process method calls and transfer data to/from the serial port concurrently. TAO provides several concurrency models and in our tests the default configuration has been used, namely a single-threaded, reactive model. One thread handles requests from multiple clients via a single Reactor [3].



It is appropriate when the requests take a fixed, relatively uniform amount of time and are largely compute bound. This is the case in this application where the response of the remote serial client program is immediate and is determined by a 50 ms timer. The single thread processes all connection requests and CORBA messages. Application servants need not be concerned with synchronizing their interactions since there is only one thread active with this model. [4]

AMI callback model

AMI helps solve the problems with waiting efficiently for long latency calls to complete. With this technique, long-running calls don't interfere with other calls. It allows single-threaded applications to avoid blocking while waiting for responses.

One of the advantages of this model is that existing CORBA servers need not be changed at all to handle AMI requests. Furthermore, the TAO IDL compiler automatically generates methods that implement the AMI communication model. A reference to a handler object is passed as an additional parameter to the invocation method and the response is received in this handler object. The class handler declaration and implementation are also automatically generated.

The serial server has been modified to generate a one second timeout. Requests are sent in a loop and responses are arriving in the right order (counter value) with a time difference of 1 second.

The AMI client callback model requires client programmers to write more code than with the synchronous model. In particular, client programmers must write the Handler servant, as well as the associated client event loop code to manage the asynchronous replies.

Events and notification model
While above communication methods are only based on CORBA core implementation, events and notification communication models rely on additional services. They allow the creation of events or notification channels that will are used by producers and consumers.

In these models, producer and consumer are now decoupled. If we keep the same data flow, the client plays now the role of producer pushing data and the serial server the role of consumer. The data are pushed to the consumer by the event channel.

The data flow throughput is in this case limited by the serial communication speed that is far slower than the TCIP one. In case of slow data production (manual input), the application behaves like the original one. But if the data production runs faster, this could rapidly leads to buffer overflow and lost data. If the communication is broken or the application on the other side is slow or response times are variable, we cannot absorb the data flow.

In the previous implementation, if there is no response from the system connected to the serial port, the caller blocks and returns after a timeout of 1sec. It means that if we use a push model, the push period should be larger than 1 sec. Consequently, it should be better for the serial server to pull the data (PullConsumer) and for the client to provide it on request (PullSupplier).

In the case of a mobile robot a joystick could produce motion commands at regular intervals (typically 50 ms) and consequently a push model is best suited for the Supplier. So we get a Push/Pull hybrid model and the Event Channel act as a queue. Another solution is to modify the serial server implementation by using non-blocking (serial) communication.

What happens if a pushconsumer blocks on the call of the push() invocation?
If we use a single threaded Event Channel, all communications will also be blocked. So, it could perhaps be better to use a thread by connection or thread by client model for the ORB.

One of the advantages of the synchronous two-way communication model is to return the state of the communication on the serial port to the client. By using an event model, we lose this capability. It means that we need to use another method if we want to monitor the serial communication state.

We see that using models with higher capabilities also requires more effort to program and to maintain and also more resources from the system.

Which model for which task?
According to the task to be accomplished, different communication models can be selected. From model properties and tests described above, we derive the following guidelines.

Configuration
Synchronous two-way is best suited for light operations. It can be used for configuration tasks or to evaluate the availability of resources.

Control
An Event model is best suited for components which continuously produce/consume data and for processes requiring RT capabilities.
Motion commands are generated continuously (periodic or not): mouse click, joystick, manual commands, file, path generator, ... and are best propagated as events.

Data processing
Asynchronous Method Invocation can be used for components requiring long computation times (planning, localisation, stereovision,...).

Visualisation
Data are continuously produced (converted by intermediate components and forwarded to the next one) to finally arrive with the consumers. An event model (Notification) is best suited in this case.

7 CONCLUSION

Developing modular control software requires a systematic and detailed analysis of the applications requirements. Furthermore, adopting programming standards like CORBA and the choice of open-source software is the only way, according to us, to reach this software modularity.

CORBA provides different communication models that suit different users' needs. By using more sophisticated models, like the Events model, we can develop more flexible software. On the other hand, it generally requires to write more code and to modify data flow models.

If using simple CORBA synchronous model is not more complicated than writing sockets based programs, other models require assimilating new communication paradigms and thinking differently. CORBA has certainly a steep learning curve but offers many benefits for writing heavy distributed applications.

REFERENCES

[1] *Software Modularity for Mobile Robotic Applications*, Eric Colon, Hichem Sahli, Clawar conference, September 2003, Catania, Italy.

[2] *Telematics Applications in Automation and Robotics - TA2001*, July 2001, Weingarten, Germany

[3] *Pattern Languages of Program Design*, Jim Coplien and Douglas C. Schmidt, Addison-Westley, 1995, ISBN 0-201-6073-4

[4] Configuring TAO's components, Douglas C. Schmidt, http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/configurations.html.