

# **The Potential Benefits of CORBA-Based Solutions for the Radar Data Acquisition System (RADAC)**

by  
Tim Turner  
Turner Engineering  
(919) 929 6851  
[tturner@intrex.net](mailto:tturner@intrex.net)

for  
Real-Time Software Engineering Branch  
NASA Wallops Flight Facility  
Wallops Island VA

and  
Computer Sciences Corporation  
Wallops Island VA

6 January 2000

## Table of Contents

List of Figures.....	2
Summary.....	3
1 Introduction .....	3
2 RADAC and CORBA .....	3
3 Demo system description.....	8
4 Results and recommendations.....	12
Appendix A: Experiences with TAO (the ACE ORB).....	16
References.....	17

## List of Figures

Figure 1. A CORBA interface.....	7
Figure 2. Map display from CORBA demo.....	8
Figure 3. Control and data flow .....	9
Figure 4. CORBA interfaces.....	10
Figure 5. Control states .....	11
Figure 6. Filter testbed concept.....	13
Figure 7. Proposed implementation of filter testbed.....	14

## Summary

This report describes the benefits of the CORBA (Common Object Request Broker Architecture) standard for the proposed RADAC (Radar Data Acquisition) system at NASA Wallops Flight Facility, and recommends using CORBA to implement data and control interfaces in the RADAC. The report also describes demonstration programs that were written to explore the use of CORBA for the RADAC system. These programs are suitable for use as a starting point for the development of the RADAC system in-house, if desired.

### 1 Introduction

NASA Wallops Flight Facility is considering the implementation of a new RADAR data acquisition system to replace the current Real-Time Computer System. This effort is currently in the Requirements Definition Phase.

Turner Engineering was hired in June 1999 to review the Requirements Document. That work evolved into the current effort to explore the potential benefits of CORBA for the RADAC system.

#### 1.1 RADAC system

The RADAC system will acquire data from multiple tracking sources, process the data including filtering and best source determination, and provide graphical displays. It can be assumed that the RADAC system will form a heterogeneous network involving a mix of computer architectures, operating systems, and programming languages. Some of the data formats will be mission-dependent. A goal of the new RADAC design will be to facilitate the rapid adaptation of the system for new missions, in order to provide competitive launch services in the future.

#### 1.2 CORBA standard

CORBA is an international, open-systems standard for inter-process communication. In CORBA, interfaces are viewed as distributed objects, containing both data and functions (methods). CORBA translates between differing machine architectures, operating systems, and programming languages, and provides a high-level way to define and implement new interfaces.

#### 1.3 In this document

Section 2 outlines some of the benefits of CORBA for the RADAC system. In Section 3, we describe our experiences with CORBA in a proof-of-concept demonstration. Section 4 presents our results and recommendations for future work. Finally, Appendix A relates our experience using TAO, a free CORBA implementation from Washington University.

### 2 RADAC and CORBA

CORBA offers numerous advantages to the RADAC system:

- ◆ rapid turn-around for new missions
- ◆ easy to extend system functionality
- ◆ keep hardware and software options open
- ◆ reliable, respectable real-time performance
- ◆ additional CORBA capabilities

#### 2.1 Rapid turn-around

In order to offer competitive launch services, Wallops must be able to tool up quickly for new missions or for evolving mission requirements. Frequently, the content and format of telemetry data changes between missions. In the current system, all values are converted to 32-bit integers, whose meaning, format, and units are described in tables. Programmers are responsible for encoding the data in the telemetry processor, then decoding the data

downstream, including possibly swapping bytes if the receiving processor uses a different byte order than the sending processor.

With CORBA, it is possible to express data using a variety of data types, including integer, floating point, enumerated types, characters, strings, arrays, and structs. Values can be stored directly in engineering units. CORBA handles the translation of data representations between differing computer architectures. The impact on turn-around is that the CORBA specification of the data format can be read and interpreted by machines and humans, and many of the programming steps for handling new data formats are completely automated.

There is potential for further productivity increases. Some CORBA implementations (called Object Request Brokers, or ORB's) offer editors for defining data interfaces, effectively eliminating another programming step. It is also possible to develop an automated logging system using CORBA services, providing a platform-independent file format with no additional programming required for new data formats.

For example, here is the nominal data specification for the demo system, written in CORBA's Interface Definition Language (IDL):

```
struct Nom_info
{
    float pgm_time;    // program time (seconds)
    float speed;      // nominal speed (m/s)
    float pos_lat;    // nominal pp latitude (degrees)
    float pos_long;   // nominal pp longitude (degrees)
    float pos_alt;    // nominal pp altitude (feet)
    float flt_el;     // nominal body elevation (degrees)
    float flt_az;     // nominal body azimuth (degrees)
    float iip_lat;    // IIP latitude (degrees)
    float iip_long;   // IIP longitude (degrees)
};
```

Data types can be as elaborate as you wish. Here's a refinement of the above:

```
struct LLA_point
{
    float lat; // latitude (degrees)
    float long; // longitude (degrees)
    float alt; // altitude (feet)
};

struct LL_point
    float iip_lat; // IIP latitude (degrees)
    float iip_long; // IIP longitude (degrees)
};

struct Nom_info
{
    float pgm_time; // program time (seconds)
    float speed; // nominal speed (m/s)
    LLA_point pp; // present position
    float flt_el; // nominal body elevation (degrees)
    float flt_az; // nominal body azimuth (degrees)
    LL_point iip; // IIP
};
```

## **2.2 Extending system functionality**

It is easy to define new interfaces using CORBA, as seen from the example above. If the system is properly implemented, it is also easy to create new code modules to plug into the system. In the demo system, the code has been modularized so that the programmer needs only to code the functionality provided by the new module. The control interface, control logic, and data interface code is all inherited from parent classes in C++.

Specifically, to implement a new module using the demo system code, you supply the actions for the control interface routines such as `init()`, `start()`, and `shutdown()`, and for the processing steps to be called from data interface routine `push()`, which receives the data.

Note that CORBA provides language bindings for a number of high-level languages, including C, C++, Ada, Smalltalk, and Java. A particular ORB will support one or more of these. We used TAO from Washington University, which only supports C++.

## **2.3 Keeping hardware and software options open**

Real-time launch systems tend to last a long time. In the end, the system may be running on hardware that is impossible to maintain because the manufacturers have gone out of business. The operating systems and programming languages used may also become impediments to maintenance, much less further development of the system.

The hardware, operating system, and computing language(s) comprise the platform for a computer program. A major advantage of CORBA is that it is platform-independent. For a particular computing language, the code interface to a CORBA object is immutable for all operating systems and hardware architectures. Also, the server can use a different ORB computer language than its clients.

If five years out, you want to switch to a different computer architecture, architecture, or language, if there is an ORB available for that platform, your interfaces will be up and running. If you decide to switch ORB's, the clients will all work without modification, and the servers should be readily adaptable. There are currently dozens of commercially available ORB's on the market.

## **2.4 Real-time performance**

CORBA delivers respectable real-time performance, with guaranteed delivery. If a transmission fails for some reason, the problem can be resolved by an exception handler.

Using CORBA's interface protocol, you can specify whether you want data transfers to block, meaning the server will wait until the client has received the data, or not. There are no blocking transfers in the demo system, so that a slow or failed process cannot cause another process to hang.

Using a free ORB from Washington University, and a 300 MHz PC, we measured latencies of approximately 1 ms per hop. That is two orders of magnitude faster than the time interval of the program clock.

If one prefers not to use the CORBA protocols, it's possible to convert the data to CORBA's binary format, then transmit the data using some other protocol, such as broadcast sockets.

## **2.5 Other CORBA capabilities**

CORBA will be useful to the RADAC system simply for its clean handling of interfaces and communication protocols. There are, however, many more capabilities, whose utility may only show up later on. These are the hidden advantages of adopting a mature, open-systems standard with broad industry support. I'll point out a few of these advantages here. Non-programmers may want to skip over the fine points.

Using CORBA, it's easy for your program to respond to many different input sources within a single thread of control. That means that you don't have to create an additional process or task for each simultaneous input source. For example, the demo system employs both control and data interfaces. Every other process in the system responds to commands from a controller process, which in turn responds to user inputs. All the processes besides the controller receive commands in the form of CORBA messages, so that their data interfaces are under control of their control interfaces. The extreme case is a map display process, which handles control and data inputs, keyboard commands, and mouse clicks simultaneously.

CORBA has many additional options in the form of CORBA services. One of these, the naming service, is used extensively during start-up of the demo system. The Naming Service provides a distributed table look-up. It allows you to store information, accessed by strings. In the demo, the controller process and the data server process register with the Naming Service, the other processes, which need these two in order to run, simply look up their location using the Naming Service. In this way, there is no prior information in the system about where processes are located, not even the Naming Service itself.

Other services provide additional support roles. For example, there is a service to convert interface data to and from the internal form CORBA uses. This makes it possible to generate file writers and readers automatically, or to use your own protocol for passing data defined by CORBA.

In addition, should the need arise for, e.g., real-time, distributed database with queries, CORBA provides services for that as well.

## **2.6 CORBA: how do you use it?**

As a programmer, using CORBA involves the following steps:

- ◆ you define interfaces in Interface Description Language (IDL)
- ◆ IDL compiler produces code for the client and server sides of the interface
- ◆ you write code to implement the server actions
- ◆ you write the client application

Figure 1 shows a schematic of a CORBA interface. The process initiating a transaction is the client. The answering process is the server. The code that handles the transfer of data across the interface belongs to the ORB. Both the client and server view the interface as a set of subroutines. The client initiates a request by calling an interface routine, which causes the server to enter the corresponding routine on the server side. As with normal subroutines, data can be passed in either direction.

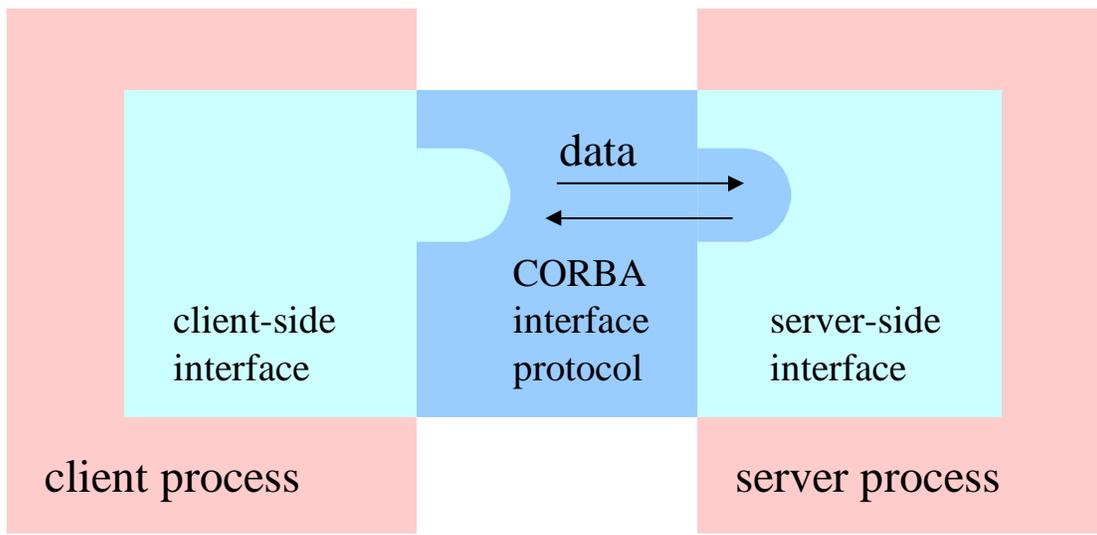


Figure 1. A CORBA interface

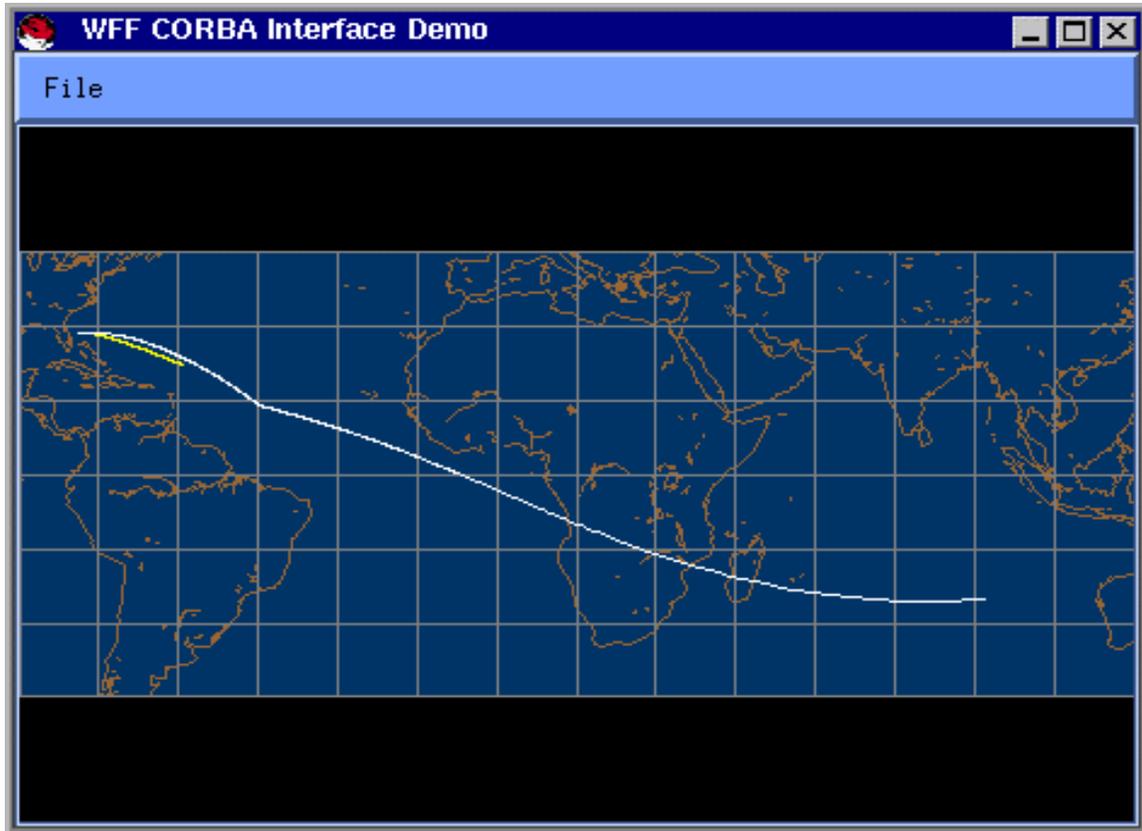


Figure 2. Map display from CORBA demo

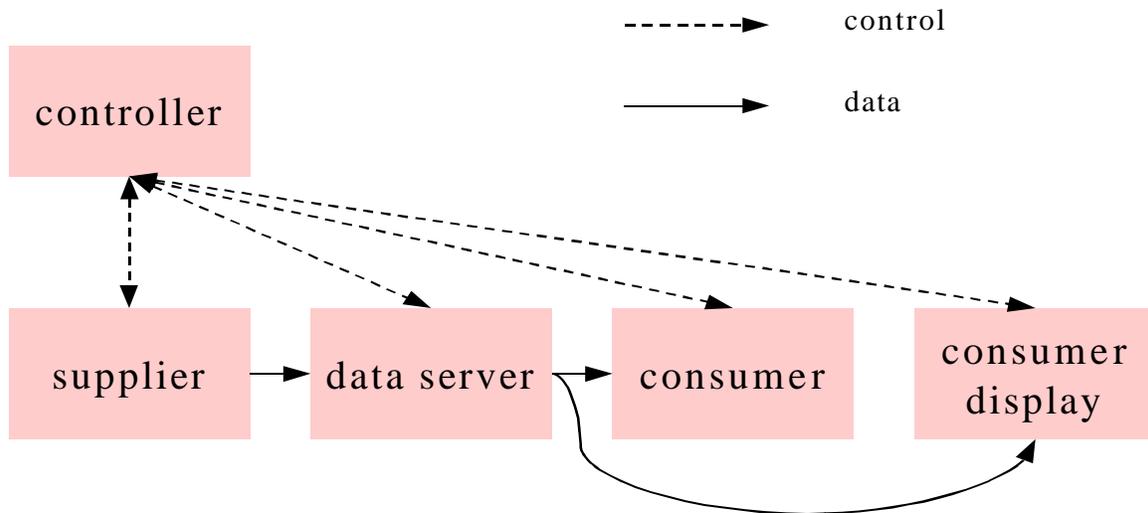
### 3 Demo system description

As part of our investigation into the CORBA standard, we put together two software demonstrations that attempt to show how CORBA could be used in the RADAC system. In the first demo, we used CORBA interfaces to pass nominal flight data at 10 points per second. In the second demo, we optimized the data passing and added control interfaces. Under this new scheme, a control program starts the other programs. It then sends commands to these programs via CORBA interfaces, under user control. This illustrates how CORBA can be used to configure and control an entire RADAC system from a single console.

The CORBA demos serve a number of purposes:

- ◆ To show the viability of using CORBA in a RADAC-like system. In the demo, vehicle present positions and instantaneous impact points are transferred at 10 points per second, which is representative of the types of data and the sampling interval in the real system. In addition, the demo shows how CORBA can be used to implement a system with central start-up and control.
- ◆ To resolve technical issues which might be problematic to the implementation of the RADAC using CORBA:
  - ◆ integrate control and data flows
  - ◆ combine CORBA and interactive displays using X Windows
  - ◆ enable code re-use
  - ◆ measure timing performance

Since the second demo is an extension of the first, we will only describe the second one here. The demo consists of five processes communicating through CORBA interfaces. Figure 3 shows the system diagram, including communication paths.



**Figure 3. Control and data flow**

The processes are as follows

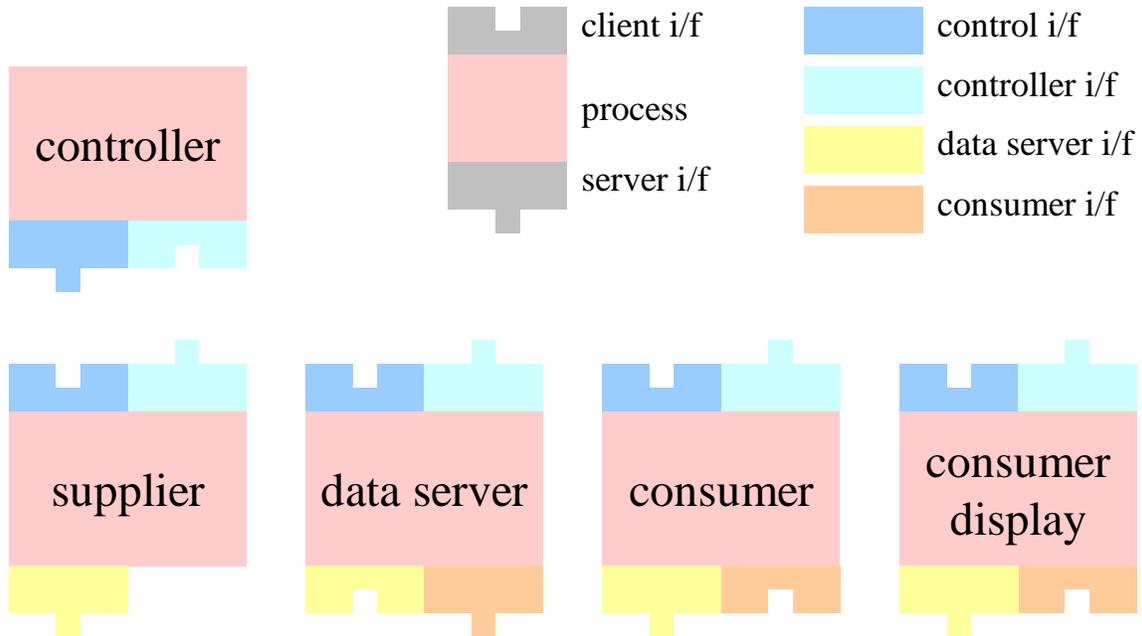
- ◆ The Controller starts the other processes, and sends commands under user control.
- ◆ The Supplier reads vehicle present position (PP) and Instantaneous Impact Point (IIP) data from a file, and sends the data to the Data Server 10 times per second
- ◆ The Data Server accepts new data, and sends it to registered consumers
- ◆ The Consumer accepts the PP and IIP data from the data server, and displays it in a text window
- ◆ The Consumer Display accepts the PP and IIP data from the data server, and displays their traces on a map display

The architecture of the real RADAC, while more complex, will be quite similar. There will generally be multiple suppliers and consumers of data, as well as modules that will accept CORBA inputs and produce CORBA outputs in turn, for example Kalman filters. Consequently, there will be many more types of messages. However, the control logic and control mechanisms will remain essentially the same.

All of the control and data flows in the demo system are implemented as CORBA interfaces. Figure 4 shows how the interfaces are arranged. There are four types of interfaces:

- ◆ Control interfaces provide the commands by which processes can be remotely controlled
- ◆ The Controller interface allows other processes to register with the controller process, and to report changes in their internal state
- ◆ The Data Server interface accepts data from suppliers. It also accepts registration requests from consumer processes.
- ◆ Consumer interfaces accept data from the data server

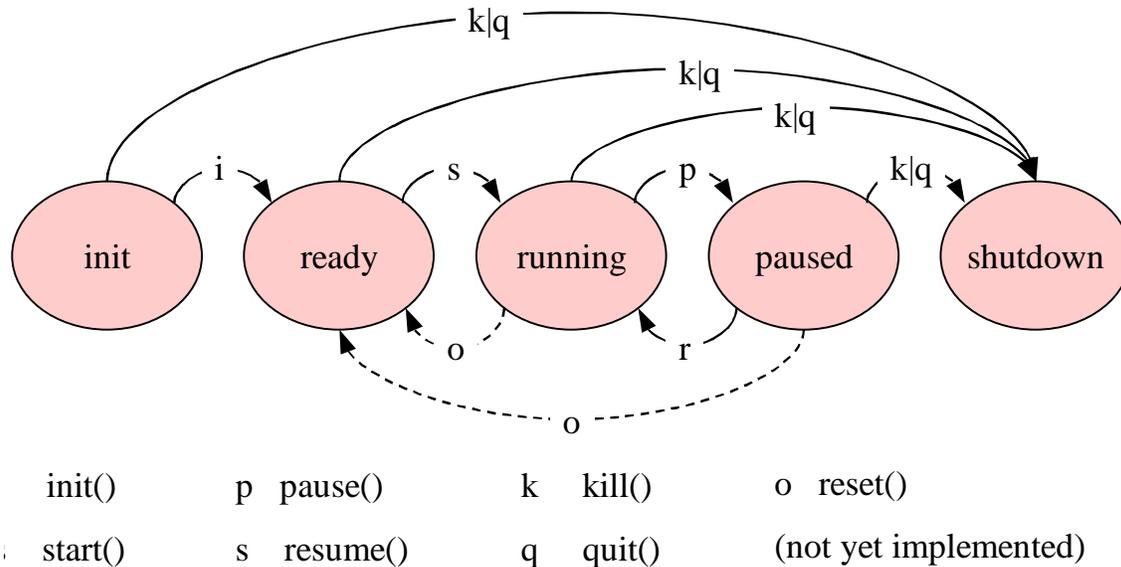
This type of system is driven by the data inputs. In the demo sequence, the supplier is in a timed loop executing 10 times per second. With each iteration, the supplier reads data from a file, and sends it to the data server. In turn, the data server sends a copy of the data to each consumer process that has previously registered to receive it. In the RADAC system, the driving input for this process will be the program time clock.



**Figure 4. CORBA interfaces**

Both the controller and control interfaces implement finite state machines. Figure 5 shows a state transition diagram for the control interface. The Control interface is always in one of the following states:

- ◆ Init
- ◆ Ready
- ◆ Running
- ◆ Paused
- ◆ Shutdown



**Figure 5. Control states**

The state transitions occur when the control interface receives a command from the controller. The commands are indicated on the diagram.

The controller sends commands when the operator requests them. When the operator chooses the start command, for example, the controller sends the start command to each process in the list.

The controller's state diagram is similar to that of the controlled process, but the conditions causing the state transitions are more complex. The controller will transition to a new state when all of the controlled processes have moved to that state. Since the controller doesn't wait for a transaction to complete, the other processes notify the controller whenever they change their state. In addition, the controller recognizes certain processes as critical, in which case it will not send commands to any further processes in the list until the critical process has confirmed its state change. These factors complicate the implementation of the controller logic.

We can now describe the entire demo sequence. The following points are worth keeping in mind:

- ◆ Information about the demo processes is read from a text file at init time. There is no prior knowledge about any of these processes, other than what appears in the file.
- ◆ The demo processes find each other at init time, using the CORBA naming service. There is no prior knowledge about the location of any process
- ◆ The demo processes can run on one or more processors
- ◆ Each data and control path is a CORBA interface
- ◆ There is no waiting for transactions to complete. A slow or stalled process cannot hang up the whole system, except by failing to produce the expected outputs
- ◆ CORBA translates between diverse platforms automatically.

In the course of the demo, the following sequence of events takes place:

The operator starts the controller process from the command line.

The controller process reads the file \$RADAC\_ROOT/demo\_2/data/processes.dat to find out which programs to run. Each line of the file has the

following format:

name wait\_for\_confirm command

There should only be a single space between each field.

name is a unique identifier which is used to look up the process in a process table. It should not contain any white space.

wait\_for\_confirm is a flag, defined as either 0 or 1. If 1, the controller suspends processing until the process confirms that the command has been executed. Processes always send confirmation upon completion of any command.

command is the shell command which starts the process.

The controller presents a "menu" of keyboard options. These are

- i initialize. Start the processes in the file processes.dat. Each process looks up the controller's object reference using the naming service. It then registers its own object reference with the controller. Ordinary processes (all except the data server) also look up the data server's object reference. Consumer processes register with the data server to receive data.
- s start. Begin real-time execution. The supplier process enters a timed loop, reading the nominal data from a file, and sending a data record 10 times per second. The data server accepts the data, then passes it on to all registered consumers.
- p pause. The supplier stops reading and sending data. All other processes wait for new data as usual.
- r resume. The supplier resumes sending data.
- k kill. The controller sends the shutdown command to each process. The processes shut down gracefully, then exit. The normal shutdown condition is to receive confirmation of the shutdown, but to receive an exception on the shutdown command itself. That's because the process shuts down without exiting the shutdown() method.
- q quit. The controller shuts down all processes, then exits.

## **4 Results and recommendations**

### **4.1 Results**

This study and proof of concept have demonstrated the viability of CORBA for the new RADAC system. The demo system use CORBA to implement both data and control interfaces. The demo features a control process which is able to start, initialize, pause, resume, and shut down all the other processes.

The demo system provides a simplified, but functionally complete method for implementing data and control interfaces, including a high-level method of data description which takes care of differences in machine architecture, operating system, and computing language between different processors in the system. Should the team decide to implement the RADAC in-house, the demo system would form a suitable baseline.

In the course of building the demo system, several technical hurdles have been resolved, including the integration of CORBA interfaces with X window displays, and the modularization of the code so that it is straightforward to create new interfaces and processing modules.

In addition, we were able to obtain some timing measurements to assess the performance of the CORBA interfaces. Running on a 300 MHz PC with a 10 Mb/s Ethernet adapter, we took the following averages:

Overhead (time to send data)	150 microseconds
Latency (end-to-end time delay)	1 millisecond / hop

By comparison the clock interval at 10 points per second is 100 milliseconds.

#### 4.2 Recommendation

We recommend that the RADAC team strongly consider the use of CORBA in the RADAC system, whether or not the system is developed in-house. Until a decision is reached as to how to proceed with the RADAC implementation, we further recommend that Wallops continue to extend the capabilities of the current demo system. If the demo system forms the basis of the proposed filter testbed, then adding features such as a data logging system will increase the utility of the filter testbed in addition to providing a viable option for the RADAC implementation.

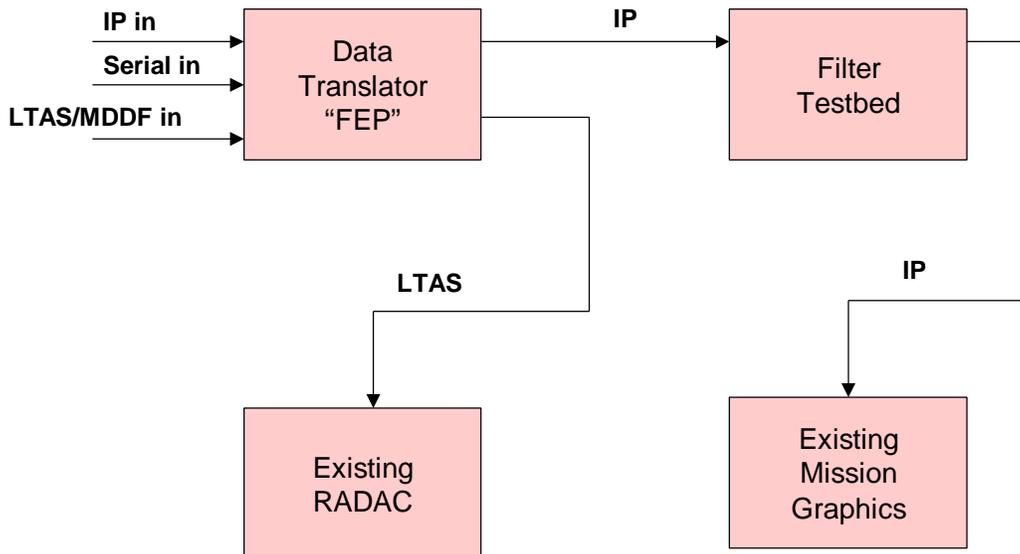
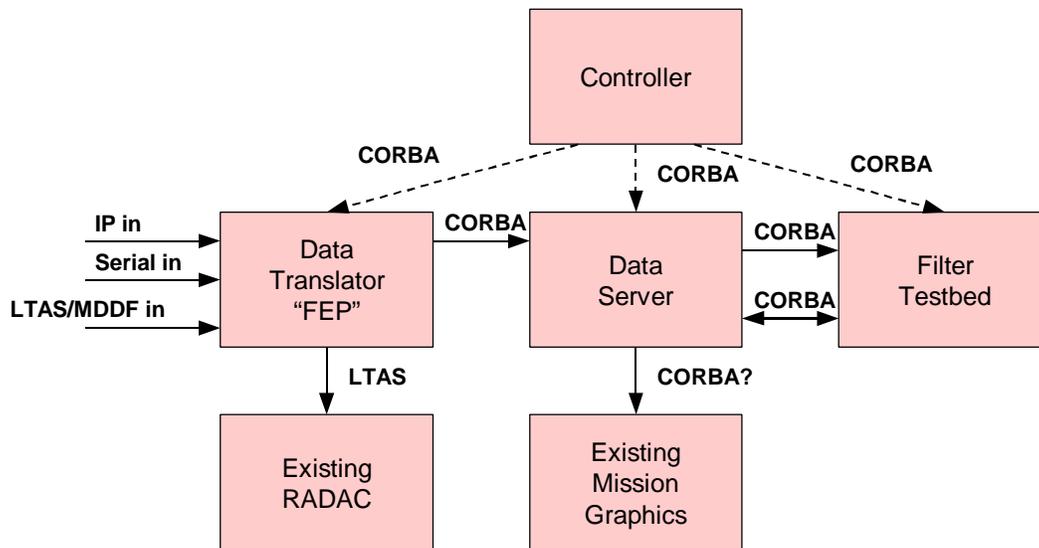


Figure 6. Filter testbed concept



**Figure 7. Proposed implementation of filter testbed**

Figure 6 shows the proposed filter testbed system. Figure 7 shows a possible implementation of the system building upon the current demo system. Such a course of action will involve the following steps:

- ◆ Extend the data server to handle multiple IDL data types. All data types should inherit from a common parent type. The data server should maintain a table of known data types. Each table entry should contain a list of registered clients
- ◆ Extend the consumer interface to handle multiple IDL data types. Each consumer interface should inherit from a common parent interface definition. The data server will see each consumer interface as its common parent type.
- ◆ Implement a platform-independent data logging capability. A logical place to put this would be in the data server. As the data server receives each piece of data, it uses CORBA services to convert the data automatically to a platform-independent, binary form.
- ◆ Implement a platform-independent playback capability using CORBA services to reconstruct the log data on any CORBA platform.
- ◆ Implement the new data types needed for the timing and tracking sources. Add code to the front end processor to pass the data to the data server.
- ◆ Implement the filter process, with data interfaces to obtain the timing and tracking data from the data server. The filter outputs should also be sent back to the data server. Use real-time plotting software for display, interpretation, and evaluation of filter outputs.

In addition, there are a number of optional steps which would further demonstrate the utility of the CORBA concept:

- ◆ Develop a graphical user interface for the controller process, consisting of a health and status display plus a set of menus, forms and dialog boxes. Provide for interactive editing of the list of component processes.
- ◆ Develop a configuration setup capability, including
  - A source file format, reader and writer for configuration parameters
  - A central or distributed repository of configuration parameters
  - CORBA interfaces to pass interface parameters
- ◆ Explore the export of data to remote computers for viewing with internet browser technology

- ◆ Modify existing displays to operate on CORBA/IDL data.
- ◆ Port display programs which use Silicon Graphics' proprietary GL library, to OpenGL, an open-systems standard.
- ◆ Investigate the advantages of other ORB's, including development tools and support for multiple languages.

## **Appendix A: Experiences with TAO (the ACE ORB)**

TAO is a free ORB from Washington University in Saint Louis. TAO is built upon ACE (Adaptive Communication Environment), which provides portable C++ wrappers for many operating system services, as well as a rich class library for inter-process communications.

We found that TAO and ACE work quite well, with very good real-time performance. ACE and TAO both run on many Unix and Windows platforms. There are many commercial applications based on ACE, and support is available from an outside company.

The main disadvantage to TAO is that it only supports C++. An ORB which at least supported C as well would be attractive to a larger group of programmers. We found that the documentation, while substantial was a bit lacking in some key areas, especially in the area of integrating CORBA and X Windows. On the other hand, TAO comes with a large body of sample programs, one of which was the starting point for our first demo program. We also encountered a conflict when we tried to use STL (Standard Template Library), a class library which is part of the emerging C++ standard. It is possible in principle to use STL with TAO by explicitly instantiating the STL classes, but we never got that to work.

In sum, though we found TAO to work quite well, it is worth a look at other ORB's for their support of multiple languages, as well as their development tools.

## References

CORBA documents and specifications  
TAO documentation and downloads  
RADAC requirements

<http://www.omg.org/>

<http://www.cs.wustl.edu/~schmidt/TAO.html>

<http://www.wff.nasa.gov/~RADAC/index.html>

