

Using CORBA Asynchronous Messaging, Pluggable Protocols and the Real-Time Event Service in a Real-Time Embedded System

Bruce Trask
Contact Systems
50 Miry Brook Rd
Danbury, CT 06810
btrask@contactsystems.com

This paper has been submitted for inclusion in The Object Managements Group's First Workshop on Real-Time and Embedded Distributed Object Computing, Falls Church, VA USA. July 24-27, 2000

Abstract

This paper will share our experiences using commercial-off-the-shelf CORBA[1] middleware in an actual embedded real-time system to both serve the real-time needs of our problem domain and to drastically reduce the time and money spent to develop world class commercial real-time embedded software. Furthermore, this paper will outline how a CORBA-compliant ORB can be easily and seamlessly extended with additional QoS capabilities for the purposes of meeting the real-time requirements of our domain.

In addition to discussing the non-technical concerns surrounding the use of CORBA middleware in real-time embedded systems, this paper will share our technical experiences using the following CORBA middleware capabilities to help achieve our design objectives:

1. *Remote Method Invocation (two way synchronous)*
2. *Event Service and Real-Time Event Service*
3. *Pluggable Protocols*
4. *CORBA Messaging – Asynchronous Method Invocations.*
5. *Naming Service.*

1. Introduction

The industry of automated Pick and Place Surface Mount Technology (SMT) assembly equipment is a

mature one that is currently extremely competitive. There are some big players in this industry: Siemens, Phillips Electronics, Panasonic, Fuji to name just a few. While just a few years ago there were only a handful of SMT machine designers and manufacturers, now there are more than fifty. The current dynamics of the industry require that a company's engineering team be fast and effective with both design and implementation. If one hopes to survive and prosper in this particular industry, all available state-of-the-art technologies need to be brought to bear upon the design and implementation of current and next generation machines. This increase in global competition, with its attendant tightening of budget constraints, has forced us to reconsider the methodologies and technologies we use to develop state-of-the-art equipment so that our efforts result in reduced time-to-market development times and higher quality software products

Commercial-off-the-shelf real-time CORBA middleware is one such technology that we are employing in the development of our next generation machines. While hardware technology has been continuing to make its predictable advances, the recent advances in software technology have leveled the playing field for companies taking advantage of them. Application of CORBA technology has been key in aiding our company to quickly, reliably and cheaply realize our larger-scale, next generation SMT assembly equipment, thus entering us into a new league of competition.

The rest of the paper is organized as follows: Section 2 gives a brief overview of SMT pick and place machines; Section 3 shares our experiences in choosing real-time CORBA middleware as our communications infrastructure; Section 4 will discuss some of the design challenges and how these were addressed using real-time CORBA middleware.

2. Understanding the Context: A Crash Course in SMT Placement Machines

Definition of SMT Pick and Placement equipment: SMT pick and place machines combine high speed and precise positioning and placement of devices onto printed circuit boards. Figure 1 shows a state-of-the-art SMT placement machine.



Figure 1: An SMT Pick and Place Machine

Under the hood: Figure 2 shows a view of the mechanical internals of the pick and place machine. Central to the operation of machine are two Y-axis positionable gantry beams, each having an X-axis positionable head. Throughput is maximized by having one head picking while the other is placing. On each head are three spindles used as the "fingers" that do the actual picking, placing and theta alignment of parts. To accomplish the task of picking and placing parts, these spindles move up and down in the Z-axis. On each side of the machine is a pluggable cart that contains up to 30 feeders that contain reels of SMT parts. These are the parts that ultimately end up being placed on the printed circuit board(s) in the center of the machine.

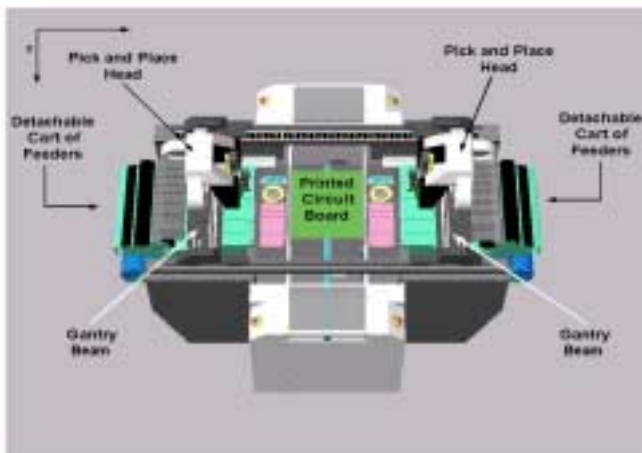


Figure 2: The parts of the machine

The basic use case: Figure 3 shows a closer birds-eye view of the machine. The simplistic operating sequence of the machine is as follows: Each head picks as many as three parts from its bank of feeders. After picking the parts, the head moves to the first placement location and in the process of this move it flies over an upward looking camera much the same way that a plane might fly over an airport. The camera will snap a picture as the head flies by so that analysis can be done to make sure that the parts are not damaged and are properly aligned for accurate placement. Note that the head does not stop over the camera but rather continues at high speed on its way to the first placement destination. The vision system provides the head with alignment correction information so that the head can make final positioning adjustments while enroute to its XY placement location. Any delay in getting this correction information to the head will cause unacceptable performance degradation. One of the critical criteria that customers look for in a pick and place machine is how many components per hour (cph) a given machine can successfully place. This specification is a vital statistic of the machine and alone can be the determining factor between gaining and losing a prospective customer.

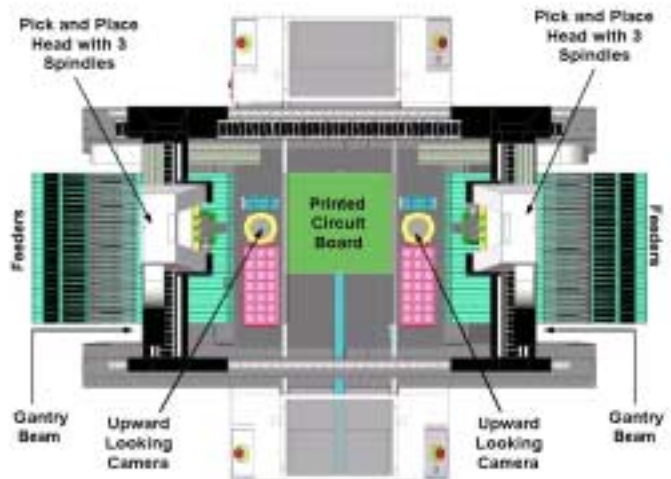


Figure 3: A top view

How is this distributed computing?: The SMT machine is composed of three basic subsystems: User Interface, the motion and vision and the two carts.

In this machine the two carts are in fact detachable and contain a stand-alone embedded system. Each cart has its own high speed CPU. This CPU manages and communicates with up to 30 feeders, each containing their own microcontroller. Each machine has two of these carts, one plugged into each side. See Figure 4.

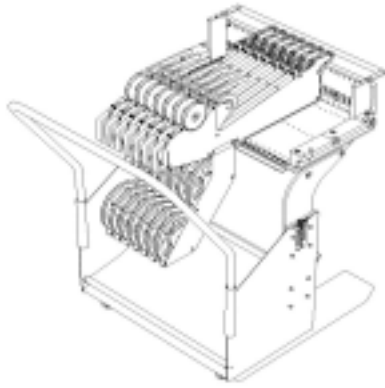


Figure 4: The Cart and Feeder Subsystem

Similarly, the other subsystems also have their own dedicated processor for handling their respective responsibilities. The subsystems executing the real-time tasks use a Real-Time Operating System while those operating on less critical tasks (such as GUI) use a General Purpose Operating System. During the course of the operation of this machine all these systems communicate and cooperate with each other to achieve the result of a correctly populated printed circuit board. The subsystems are connected by both a Fast Ethernet 100BaseT connection as well as a proprietary high-speed deterministic transport for handling critical passage of data throughout the system. See Figure 5.

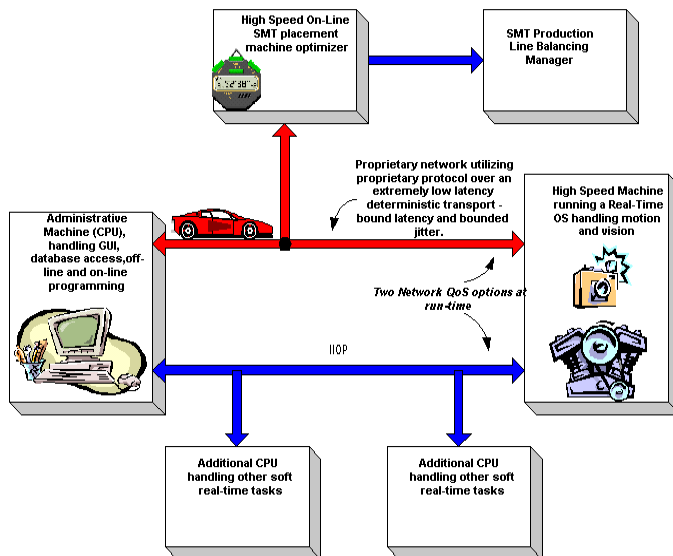


Figure 5: Basic distributed architecture

The system is inherently concurrent and event driven. There are any number of tasks that need to execute simultaneously, such as: motion, vision capture and analysis, correction, feeder actuation and handling, GUI update, logging, optimization, and job processing etc.

The above is a very brief and simple description of the problem domain and the basic architecture of the solu-

tion. Our job has been to write the software that will allow all this to happen.

3. Our Experience with Increased Software Architecture Complexity and Code Reuse

Getting existing personnel in the game of reuse: Real-time embedded software developers can be stubborn. Many suffer from the "not-invented-here" syndrome. As embedded systems by nature imply immediate control of hardware, many of these developers have their hands in the design and use of hardware as well as the software and so may not be able to track all the advances in software technology. Additionally, embedded systems software developers are used to writing very specialized software that is tightly coupled to the context. Our real-time embedded systems in the past were islands unto themselves and as a result little attention was placed on making the system necessarily extensible and modular. Part of handling the mindset of the engineering staff is an education into the costs of writing embedded software and the new dynamics of the industry itself.

A Parallel to Real-Time Middleware: Real-Time Operating Systems. In a recent Embedded Muse Newsletter, Jack Ganssle made the following observation: "Every book on software engineering talks about creating (and using!) reusable software. Yet in the embedded world, there is just not a lot of commercial software that we can actually buy and reuse. Real-time operating systems are one of the very few packages commonly available and instantly reusable. Currently, there are some 80 or more different commercial operating systems available and more and more are used in serious real-time embedded applications today. Yet surveys continue to show that some 70% of all real-time OSes are custom made! It seems that most of us don't practice even this minimal amount of reuse." [2] The cost of embedded code has been estimated at \$15-\$30 per line. A custom solution for an RTOS requiring 10,000 lines of code will cost between \$150,000 and \$300,000.

It was an easy decision for us to use a real-time operating system on our new high-speed SMT pick and place machine. Additionally, with the low cost and high power of hardware these days, competitive next generation SMT machines require multiple CPUs. Our next generation machine graduated from a single CPU machine to one with up to five high speed Pentiums IIIs interfacing to more than 8 custom processor boards (handling motion and machine vision) and over 60 microcontrollers in a fully stocked machine. Having mul-

multiple CPUs handling the tasks at hand gives us a clear separation of concerns and simplifies the software design enabling us to achieve our performance goals.

Choosing Real-Time CORBA Middleware: After having made the obvious decision to use RTOSes on the CPUs performing the real-time tasks, we mistakenly settled back into our old and stubborn ways, crafting our own real-time "middleware" that would be the communications infrastructure of the machine. Our expertise is in writing software for SMT assembly equipment, not in writing network software or distributed middleware infrastructures. So in starting to write the homemade network software infrastructure, we realized that we had taken on a large task outside our problem domain. Using custom RTOS code costs as a guide, we calculated that a homegrown communications infrastructure of 20,000 lines of code would cost over \$300,000, a large investment into an area of software outside our expertise. Given the cost of writing embedded code combined with the dynamics of our industry (short market windows with many competitors) writing our own middleware was not an option.

4. Technical Design Challenges

The broad design challenges (both technical and otherwise) facing us in our development of this new machine, included the following:

1. Concurrency.
2. Speed
3. Determinism
4. Distributed System
5. Platform independence.
6. Low Cost
7. Fast development
8. Reliability
9. Easy to learn tools and libraries – i.e. pattern based.
10. Resource Management

4.1 The Solution: Enter CORBA and TAO[3], The ACE ORB[4].

Initially our main concern, was handling the distributed commands between the various subsystem objects. We were originally attracted to the standard CORBA operation invocation model between clients and servers, particularly the two-way model with its intuitive mapping onto the object->operation() paradigm supported by our chosen development language, C++[5]. The transpar-

ency of remote operation invocations was the first part of CORBA that caught our interest. As we pursued this further we found that many of the above listed concerns could be directly addressed by CORBA middleware.

We used TAO, the ACE ORB, as our distributed communications infrastructure. TAO is an open-source, standards-based, high performance, real-time ORB endsystem DOC middleware that is capable of supporting applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements [3]. Out of the box, it offered us the necessary CORBA-compliant implementation for us to achieve the distributed nature of our machine's software.

However, we also had additional requirements above and beyond the capabilities set forth by existing CORBA specs. TAO has certain additional capabilities that enabled us to achieve our technical goals. This section will briefly describe how these technical challenges were met.

4.1 Concurrency and Asynchrony

The Problem: Inherently, real-time embedded systems are closely tied to hardware that is, in turn, closely tied to the real world where multiple events and requests for service occur concurrently. Choosing a concurrency model for the system is one of the major architectural decisions of the design. In the case of a pick and place machine, more than one SMT part is usually being picked, analyzed and placed simultaneously. In addition to the concurrent actions being taken to pick and place the parts that are on the spindles, subsystems are also working on both future pick and place tasks (in order to ensure maximum throughput) and past pick and place tasks (to ensure appropriate subsystems are updated as to the results of the placements). To a large degree, the real-time part of the machine is I/O bound. I.e. commands are sent to external special purpose processors (vision, motion, feeders) and it takes a relatively long time for these actions to complete. In order to achieve the needed specs of a machine that could viably compete in the industry, an efficient concurrency and messaging model was needed.

Thread-Pool Reactor at the ORB Core: Fortunately, TAO provides an ORB configuration option that allows the use of a thread-pool reactor [6] at its core. This allowed us to easily run various subsystems as multithreaded CORBA servers. Prior to using the thread-pool reactor, much of the thread specific programming was littered throughout the application itself. Taking advantage of this thread-pool capability removed concurrency concerns from the application code and placed

it directly into the ORB infrastructure. Servants in the application layer borrow the thread of execution from the ORB infrastructure to carry out their responsibilities. Prior to this approach, use of Active Objects [7] and other concurrency related strategies and implementations in the application layer were required. Using the Thread-Pool approach leaves the applications with only synchronization concerns for server-level objects [8]. We were able to use the existing Service Configurator pattern provided by the ACE Framework to configure the ORB as we needed. [9]. In our domain, this was efficacious in implementing the Cart software as many commands for feeders arrive simultaneously and need to be handled concurrently. There is, however, an upper bound in the number of commands that can arrive during a given time and so we are able to scale our thread pool to the appropriate size to handle the incoming commands.

Asynchronous Message Invocation - CORBA for Impatient Clients: Due to the inherent concurrency of the problem domain and the number of tasks to be performed, an efficient messaging scheme was also needed. In the pick and place universe, the time it takes to actuate a feeder to expose a new part is relatively long (on the order of 500 ms) and this task happens repeatedly during the course of a pick and place job. Systems requesting that feeders actuate themselves have many tasks to do in a pick and place machine and so cannot spend time waiting for a response as in SMI. Additionally, the various subsystems do not have the resources/CPU cycles to spawn a thread or create a new connection for each request. For the same reasons, remote polling of the servant for results is also not an option.

Prior to the release of the CORBA Messaging Spec [10] and its subsequent implementation into TAO by the Distributed Object Computing group at Washington University and the University of California at Irvine, we achieved the needed asynchrony through a combination of Synchronous Message Invocation (to achieve the remote command invocation), Active Objects at the application layer on the server side (to achieve decoupling of method invocation from method execution) and the CORBA Event Service [11] (to achieve the needed asynchronous response). While ultimately efficacious, this scheme resulted in a complex messaging and concurrency model at the application level. This scheme has been significantly simplified by the use of the asynchronous messaging capabilities outlined in the CORBA Messaging Specification. In our implementation, we took advantage of the Callback model for Asynchronous Messaging Invocations as implemented in the TAO ORB. This model allows applications to re-

ceive results in an application-defined callback object. All interfaces are type-safe and generated by the IDL compiler, simplifying the work for the application developers.

Ultimately, ORB instances configured with thread-pool reactors on both the client and server side, in combination with the use of Asynchronous Messaging Invocations on the client side, created an almost complete solution to our asynchronous messaging and concurrency needs. System components communicating with feeder carts are able to "fire and forget" and quickly return to other tasks currently in line for CPU resources.

Matching request with reply: In using AMI, however, we were not totally free from adding some complexity to the program. It is an inherent consequence in using AMI that the application code is a bit more complex with regard to the message handling proper. One facet of AMI facing the application program writer is the matching of the reply with the request. We evaluated three schemes for matching the reply with the request [12]:

1. Servant-per-AMI-call strategy (i.e. use a different reply handler for each request)
2. Activation-per-AMI strategy (i.e. use the POA dynamic activation mechanisms to distinguish between all requests)
3. Server differentiated-reply strategy (e.g. return some kind of request ID from the server)

We employed choice #3 as it was the easiest and most efficient. It paralleled the problem domain. I.e. each servant (in our case, a Feeder) already had a unique ID, its slot number in the cart. Further, since this ID is just an integer, the additional bandwidth usage and network latency as a result of sending this slot number as part of the reply, is minimal.

Resource economy: TAO's AMI implementation has the ability to use different transport multiplexing strategies [13]. With a multiplexed transport strategy, many requests can be sent to the feeder cart over the same connection, even when replies are pending from earlier requests (which is frequently the case in our domain due to the relatively slow mechanical actuation time of feeders). In the case of preactuation (that is, actuating a feeder well before its exposed part is going to be picked - a basic optimization scheme in placement machines) we want to "fire and forget" and only check when we finally pick the part. As with the thread-pool reactor, we were able to configure the TAO ORB to use the muxed transport strategy simply through using the existing Service Configurator Pattern in the ACE framework [9].

Further, using the request-reply scheme as outlined above, only one servant needs be instantiated to distinguish all the AMI callbacks. The servant only needs to be activated once in the clients POA.[13]. This simplifies the client side programming as well as saving on server resources.

In our pick and place system, no subsystems are solely pure clients and so the added complexity of becoming a server through the use of AMI is not a problem.

4.2 High Speed Deterministic Transport

The latency requirements of particular parts of the system required that a custom transport be used between some parts of the subsystem to handle time-critical data. Ideally, we wanted to keep the location transparency that the CORBA model offered us but also be able implement a faster and more deterministic transport than IIOP (GIOP over TCP/IP).

Using TAO's Pluggable Protocols Framework: We already had the proprietary hardware needed for this transport but the design challenge was to make it usable with CORBA is some way. The TAO ORB has a framework in place that addressed this very need: the Pluggable Protocols Framework [14]. See Figure 6. for its architecture.

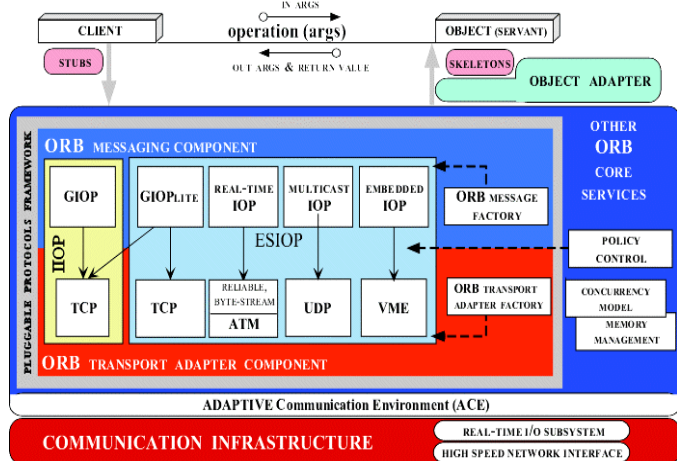


Figure 6. TAO's Pluggable Protocols Framework Architecture

This framework allows for straightforward and seamless integration of new messaging and transport facilities into the ORB infrastructure without changing the ORB infrastructure itself. Having its implementation based on well-known patterns, including general purpose patterns as well as concurrent, distributed and communications patterns, made understanding and extending this framework straightforward.

Implementing the New Protocol: Simply, one needs only to implement new protocol classes parallel to existing messaging and transports (for example IIOP or UIOP) classes. Figure 7a and 7b show the basic class diagrams of the TAO pluggable protocols framework and the necessary classes that exist for IIOP[15]. Also shown are some of the base classes from which new transport-related classes can inherit as part of a new protocol implementation. In our case it was sufficient to use GIOP over our custom transport and as such there was no need to use a different messaging scheme, just a different transport. That being the case, all that was necessary was to create classes similar to the IIOP_* classes in figure 7 (such as Custom_*) and "plug" these into the pluggable protocols framework. These new classes inherit from the same base classes as the concrete IIOP_* classes. The same type of procedure was required for new components that needed to be added to the underlying ACE Framework. Once we created the necessary classes, filled in their implementation, and connected them to the existing hierarchy, we used the Service Configurator facility to install the necessary concrete factory[16] for producing the object instances that effected our custom transport. For a full discussion about how to implement a pluggable protocol into the TAO ORB see [15].

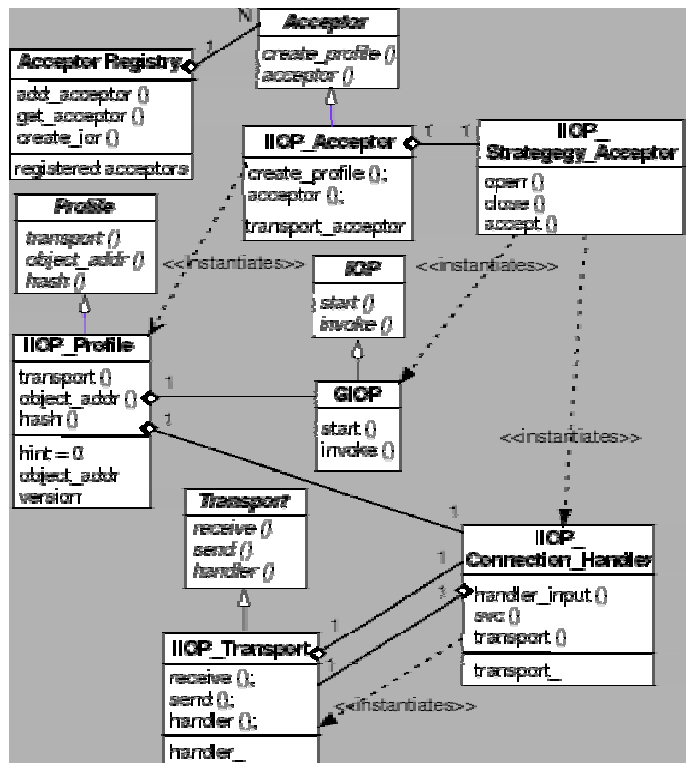


Figure 7a: Class Diagram of TAO's Pluggable Protocols Framework - server side

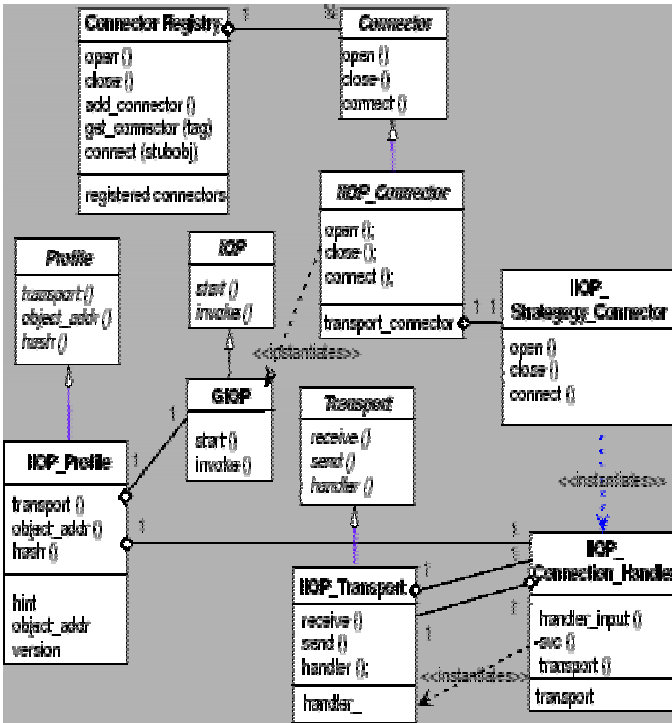


Figure 7b: Class Diagram of TAO's Pluggable Protocols Framework - client side[15]

Our new transport relies on interrupts as a signaling mechanism. What was needed was an I/O demultiplexing mechanism that would work with Win32 Events since our RTOS includes a subset of the Win32 Threading and Synchronization API. This was easily accomplished as the TAO ORB core can be configured with a Reactor [6] that uses the Win32 API call WaitForMultipleObjects.

Certain subsystems of our machine use both IOP and the custom transport. Having the ability to configure the ORB core with different types of reactors enabled us to have multiple instances of the ORB in the same address space, one configured with a Select Reactor (using the select() call at its core) and one with the WaitForMultipleObjects Reactor (with the WaitForMultipleObjects() call at its core).

For our purposes, this custom transport allowed us to pass time-critical data (vision and optimization data as shown in Figure 5) in a timely fashion such that the performance of the machine was at a predictable optimum. Data of a non-critical nature (e.g. GUI updates) could pass on its own network (IOP), not impeding the time-critical data.

Additionally, our RTOS has a rather CPU-intensive TCP/IP stack implementation, a fact we discovered late in the development cycle. The custom transport allowed us to not only remove this CPU-cycle consuming stack, allowing for more free CPU time, but also provided a more deterministic transport for our time critical data.

4.3 Implementing the Observer Pattern [16] in a Distributed Real-Time System

Due to the performance constraints of our problem domain, certain tasks are performed by different CPUs in different address spaces. Time critical tasks that directly determine the performance of the machine run on a RTOS while non-time-critical tasks such as updating the GUI and database access are relegated to a General Purpose Operating System. Nevertheless, these various subsystems share important run-time data that needs to be synchronized throughout the system. Referring to Figure 5, for example, the results of the motion and vision tasks need to be rapidly communicated to the optimizer and vice-versa. Similarly, feeder carts need to notify any malfunction to the GUI subsystem so the machine operator can handle them quickly to achieve minimal loss of throughput. Addition and removal of feeders is another example of data that needs to propagate through the system. Position changes of feeders can affect the results of the on-line optimizer and can create the need for the entire job to be reoptimized. What is needed is a component that functions as the mediator [16] between the various distributed components of the system, a distributed implementation of the Observer Pattern[16]. As with our original decision to use Real-Time CORBA middleware as our remote procedure call mechanism, the choice to as to whether to invent our own distributed update mechanism or reuse code was clear. The mechanism and implementation already existed and served our purposes. This was the TAO's Real-Time Event Service[5].

Use of TAO's Real-Time Event Service [5]: TAO's Real-Time Event Service extends the COS Event Service specification in order to provide functionality to satisfy increased quality of service needs of real-time applications.

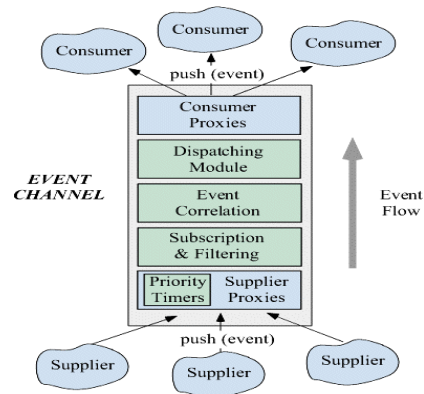


Figure 8: TAO's RT Event Service Architecture

In our application, this RT Event Service provided us the additional capabilities we needed to synchronize all data on the distributed system. Configuration of the RT Event Service in the Event Forwarder Discriminator configuration has proved sufficient at this time for our update needs. This configuration allows the Event Service infrastructure to filter event notifications to subscribed consumers. In our case many of the events sent from the cart are of no interest to the motion and vision system but are of interest to the GUI and optimizer. Similarly, there are events generated by the cart that are of interest to the motion system and conditionally of interest to some other subsystems. Having the filtering capabilities of the Real-Time Event Service allowed us to reduce and control the overhead of our update mechanism.

4.4 Some final notes on the design challenges.

The final concerns from the list mentioned in the beginning of section 4, were all addressed by our use of TAO.

Platform Independence: Originally, TAO did not support our Real-Time Operating System. Since TAO does support many OSes, both Real-Time and General Purpose, we were able to port TAO over to our platform very easily.

Fast Development: As TAO is CORBA compliant, the distributed development went forward without a snag. Concepts learned from books and papers based on the CORBA standard mapped directly over into writing distributed applications with TAO. Additionally, being that the TAO code and ACE framework are heavily based on design patterns, understanding the internals (as was required for our implementation of the pluggable protocol) was straightforward.

Low Cost: With TAO being an Open-Source project, the cost of the code certainly fits into our business plan.

Reliability: We have experienced no reliability problems using TAO's CORBA-compliant middleware in our real-time embedded application.

4.5 Bootstrapping the distributed system: As is the case with most distributed systems, real-time or otherwise, the CORBA COS Naming Service provided the main bootstrapping mechanism for our distributed system.

5. Conclusion

A number of factors, including increased global competition and increased system complexity, have forced us to reconsider our current design and operating methodologies. One result of this change has been the use of CORBA-compliant middleware in our real-time systems as the means for handling the machine-wide communications infrastructure. Not only have we benefited from the transparency that the standard CORBA operation invocation model between clients and servers give us (particularly the two-way model with its intuitive mapping onto the object->operation() paradigm), but also we have benefited from the CORBA standard services. In cases where additional capabilities were needed we have had success using the extensions provided by the TAO ORB that allowed us to implement needed concurrency models and proprietary transports. The use of CORBA compliant middleware has been key in developing our new line of SMT pick and place machines. Moreover, our experience has been that use of CORBA middleware is not only desirable but vital for our commercial success.

6. Acknowledgments

I would like to thank Louis Franco of Droplet Inc, Nanbor Wang of Washington University at St. Louis, Carlos O'Ryan of the University of California at Irvine, and Bruce Lokeinsky, for their comments and suggestions on this paper. Additionally, I would like to thank the members of the Distributed Object Computing group at Washington University at St. Louis and the University of California at Irvine for their work in the development of ACE and TAO and for their assistance to us in using it.

References

- [1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999
- [2] J.Ganssle, "The Embedded Muse 46", March 3, 2000
- [3] D. C. Schmidt, D. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications*, Elsevier Science, Volume 21, No 4, April, 1998.
- [4] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.
- [5] T. Harrison, C. O'Ryan, D. L. Levine and D. C. Schmidt,

"The Design and Performance of a Real-Time CORBA Event Service", IEEE Journal of Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems.

[6] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529-545, Reading, MA: Addison-Wesley, 1995.

[7] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[8] D. C. Schmidt, S. Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool", *C++ Report*, April 1996.

[9] P. Jain and D. C. Schmidt, "Dynamically Configuring Communication Services with the Service Configurator Pattern," *C++ Report*, vol. 9, Jun. 1997

[10] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[11] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995

[12] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt and M. Kircher, "Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks," *C++ Report*, vol. 12, Mar. 2000

[13] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," *Proceedings, Middleware 2000 Conference*, April 3-7, 2000

[14] F. Kuhns, C. O'Ryan, D. C. Schmidt and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware"

[15] F. Kuhns, D. C. Schmidt, C. O'Ryan, O. Othman and B.Trask, "Implementing Pluggable Protocols for TAO", TAO distribution, 1.1.

[16] E.Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.