Title CORBA in the Undergraduate Computer Science Curriculum

Stuart Hansen
Computer Science Department
University of Wisconsin - Parkside
hansen@cs.uwp.edu

Timothy V. Fossum
Computer Science Department
University of Wisconsin - Parkside
fossum@cs.uwp.edu

Abstract

The Common Object Request Broker Architecture (CORBA) provides a middleware infrastructure that allows applications running on different hardware and operating systems, and developed in different languages to work together seamlessly. CORBA supplies high-level libraries and utilities that hide the details of message passing, making it relatively painless to develop distributed systems.

We have used CORBA in two different undergraduate courses, CS1 and an advanced elective entitled Event Driven Programming. In CS1 students are presented with a CORBA based email server and develop a simple email client to communicate with it. In Event Driven Programming students use CORBA to develop client-server and peer to peer applications. We have found that CORBA is an ideal infrastructure for introducing distributed computing at both levels. Students are relieved of the necessity to learn low level communication primitives and can concentrate almost entirely on application design and development issues.

This paper provides a brief high-level overview of CORBA and then discusses how we have used CORBA in our courses. We also include analysis of some preliminary data analysis on problems students encountered while developing CORBA based applications.

Introduction

One of the problems computer programmers face when developing distributed applications is that they must deal with many low-level technical details involved in establishing and maintaining communication between two processes. When the processes are running on different platforms or are written in different languages, the number of details grows significantly. In recent years there have been a variety of attempts to develop higher level abstractions for distributed processing, remote procedure calls, remote method invocation, DCOM, Microsoft .Net remoting and web services. CORBA is unique among these efforts in being truly platform and language independent..

CORBA is an architecture and infrastructure specified by the Object Management Group. CORBA is not an API or a software package. Instead, it is a specification. Different CORBA vendors implement the specification for a platform and language. These implementations are then sold (or given away) for developers to use. CORBA is vendor independent, hardware independent, operating system independent and programming language independent. It excels at tying together legacy systems running on different platforms. It also provides an excellent set of tools to develop client-server and peer to peer distributed applications.

CORBA Overview

Like many modern software specifications, CORBA has grown and evolved very rapidly. The discussion in this paper uses CORBA Version 2.3, which is the version currently delivered with Sun's Java SDK.

One goal of CORBA is to have the programmer write code in a way that is very similar to developing a standalone application. The programmer should be able to treat remote objects in the same way as local objects. To accomplish this, CORBA generates local stubs, used on the client side and object adapters (sometimes called skeletons), used on the server side, that hide the details of making the remote calls. The stubs and adapters communicate with an object request broker (ORB) that handles the communication.

Figure 1 shows the basic message path for client requests for services [5]. The Object Request Broker (ORB) is the standard CORBA component for data communication. ORBs "talk" with each other using the Internet Inter-ORB Protocol (IIOP).

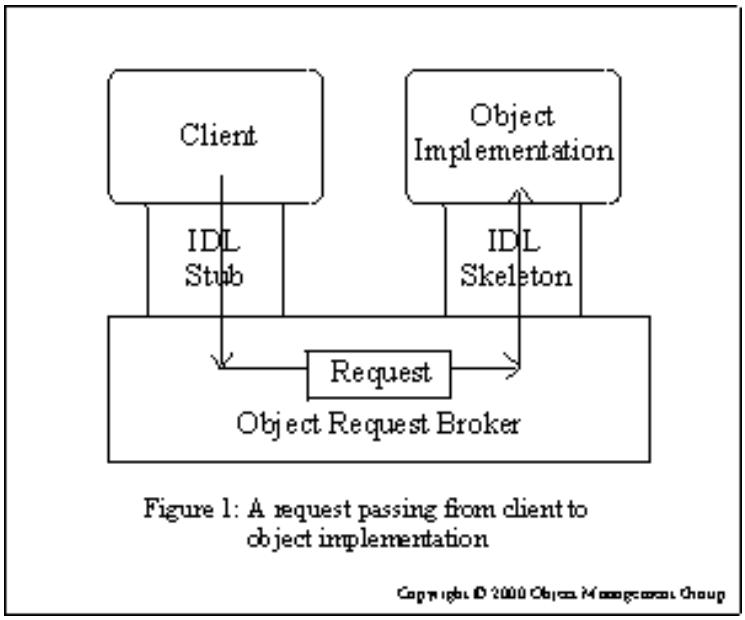

Figure 1: A request passing from client to object implementation

To develop an application using CORBA, we go through the following steps:

1. Design and Compile the Service Interface.
2. Implement the Server.
3. Implement the Client.
4. Start the Server and Start the Client.

We deal very briefly with each of these steps in turn.

Design and Compile the Service Interface

As stated earlier, CORBA is language independent. A client may be developed in Java and the server may be developed in C++. To get the different languages to work together, stubs and adapters must be generated for each. CORBA provides a metalanguage, the Interface Definition Language (IDL), used to generate the stubs and adapters. The programmer specifies the service's interface in IDL. Special compilers -- Java's compiler is named `idlj` -- are used to compile the IDL interface into both stubs and object adapters for each source language. When working in Java or C++ or Ada or any other language that has CORBA support, the same IDL file would be used. A different compiler would generate the stubs and object adapters for that language.

Below is a sample IDL file for a simple calculator.

```
1.   /* Calculator.idl
2.      This file contains the interface definitions for a
3.      simple calculator.
4.      Written by: Stuart Hansen
5.      Date: January 5, 2004
```

```
 6.   */
 7.   module calculator {
 8.      interface Calculator {

 9.         // A method to add two numbers
10.         long add (in long a, in long b);

11.         // A method to subtract two numbers
12.         long sub (in long a, in long b);

13.         // A method to multiply two numbers
14.         long mult (in long a, in long b);
15.      };
16.   };
```

Lines 1-6 are comments. Line 7 is a module declaration, which translates to a package in Java. Lines 8-15 define an interface, which is the equivalent of an interface in Java. The interface contains three operations, `add`, `sub` and `mult`. The calculator server must implement all three methods. Each method takes two `long` parameters and returns a `long`. A `long` in IDL translates to be an `int` in Java. The parameters are all declared as `in` parameters. This means that the argument values will be passed to the server, but, even if modified by the server, will not be returned to the client.

To compile this file in Java, we use the command:
`idlj -fall Calculator.idl`

This will produce a collection of Java files, key among them are:

- `CalculatorOperations.java` – which contains the equivalent Java interface to the IDL interface.
- `_CalculatorStub.java` – which implements the interface and is used by the client.
- `CalculatorPOA.java` – which implements the interface, but is abstract.

The concrete server class extends CalculatorPOA, implementing the Calculator methods.

Implement the Server

We can implement the server by extending CalculatorPOA and implementing the methods defined in the IDL. The server's constructor must contain some CORBA initialization code, but the majority of that code changes little from application to application. Students don't need to understand it in depth. They can cut and paste the CORBA initialization code with only minor modifications.

Implement the Client

In Java, the chief difference between a standalone application and a CORBA client is that a standalone application creates an object when it needs one, but a CORBA client must

find an interoperable object reference (IOR) to a pre-existing remote server. Once it has the reference, calls to the server look like standard Java. CORBA provides three solutions to this problem. The user may provide the location of the server via command line arguments. This works well in settings where the server's location is known, which is often the case when CORBA is being used to integrate legacy systems. An alternative is to share the server's IOR with the client by writing a string version of it to a shared file or by emailing it to each new client site. The final method is to use CORBA's name service. The name service is a CORBA service that maintains a table of CORBA object names and their IORs. A client can query the name service to look up the desired server.

Start the Server and Start the Client

Starting the server and client poses only a couple of minor issues. The first is that many operating systems lock users out of using lower numbered ports, but the default port for many orbs is in this range. The programmer can either hardcode a legal port number or supply one at startup via a command line argument. The other issue is sharing the server's IOR. The server must be started first and its IOR passed to the client in order to establish communications.

Peer to Peer Programming with CORBA

The above discussion assumes a client-server model. CORBA is also an excellent tool for developing peer to peer applications. Peer to peer programming with CORBA follows the same overall pattern as client server programming. The only major difference is that all the entities may be able to receive messages. As such, they are all treated as servers. That is, they all extend some POA class and their constructors do the server style of initialization.

CORBA in Undergraduate Courses

We have found that CORBA is an ideal infrastructure for introducing distributed systems programming. It has several advantages over comparable high level. It is object-oriented, so there is very little paradigm shift for most students. Students program to CORBA interfaces. They do not need to worry about the implementation details. Its object-oriented nature also promotes student collaboration. Students can easily develop client programs that collaborate with each other to accomplish a task. Finally, because CORBA is platform and language independent, the skills students gain can be readily transferred other situations.

Developing a CORBA Email Client in CS1

One of the big challenges of teaching object-oriented programming in CS1 is finding interesting problems that are approachable by the students. For instance, it is difficult to find examples where small collections of objects work together in a meaningful way. While client-server programming is generally considered too advanced for CS1, we developed a simple CORBA server object and we have students write client code that communicates with it. The CS1 students are not asked to program any CORBA. Instead, the students simply use the interface that CORBA produces.

Our CS1 course is currently taught in Java using BlueJ. Each week students have a two hour closed lab session where they use pair programming to carry out an exercise related to the current course material. For the CORBA lab exercise, the instructors provided a CORBA based email server. Each student created a simple email client that communicated with other clients via the server.

Our lab was inspired by a similar lab in Barnes and Kölling [1]. They introduce a simplified email system as one of their early examples of small collections of objects working together. All of their objects are local, however. We adapted and expanded their example for our lab exercise. Our server is treated as a black box. The students only see the public interface for the server, but are not introduced to any of the CORBA taking place behind the scenes. Our server contains the following public methods:

1. A constructor. The constructor initializes all the networking needed to make the server work.
2. `int getMessagesWaiting (String name)` returns the number of messages waiting for the named user.
3. `Message getNextMessage (String name)` returns the next message for the named user.
4. `void post (Message message)` takes the message directs it to the user specified in the message's to field.

The student's develop two classes, Message and MailClient. The Message class consisted of four String fields: `to`, `from`, `subject` and `text`. The only two methods are a constructor and `print` which displays the message to the screen.

Students also developed a MailClient to communicate with both the user and the server. The instructors specified the functionality of the MailClient, but since no other class depended on the MailClient, students were free to name their methods as they pleased. The developed client methods for each of the following:

1. Query the server for the number of messages waiting for this user. This required a call to the server's `getMessagesWaiting()`.
2. Get and print the next message. If the student had no messages waiting, the server returned a message saying there were no messages.
3. Send a message to another user on the system. The student's code queried the user for the `to`, `subject` and `text` fields, constructed a message and call the server's `post()` method.

Students tested their classes by first sending and receiving messages to themselves, then to their partners and finally to remote users.

This lab was used for the first time during the sixth week of the Fall 2003 semester. Most of our students had no problem completing the lab within the two hour period. Both students and the lab assistants had a lot of fun sending messages back and forth to friends sitting across the lab. One of our better students decided to build a spamming program. His program sent thousands of email messages to the user of his choice. While this annoyed his instructor and his friends, the CORBA email server had no problem handling all the messages and there was no noticeable performance degradation.

The lab assignment description is available online at:
http://ginger.cs.uwp.edu/Cs241/Labs/prelab6.html and
http://ginger.cs.uwp.edu/Cs241/Labs/lab6.html.
Source code for the completed lab is available from the authors.

CORBA Projects in Event Driven Programming

Our department offers a senior level elective entitled Event Driven Programming. The goal of the course is to have students gain a better understanding of the event driven paradigm. The course is not about building graphical users interfaces (GUIs). We build Java Swing GUIs during the course, but we treat Swing as only one representative event driven technology. Our goal is broader. We want to introduce students to the paradigm and show its applicability in many different computing domains. There are numerous different libraries and technologies that are event driven. We introduce students to only a handful of these and then explore the common themes that unify the event driven paradigm.

The event driven programming paradigm is generally characterized by very loose coupling between components, state based control and minimal assumptions about synchronization. CORBA is an excellent example of all of these. We spend approximately four weeks discussing CORBA in our course, and the students do a sizable CORBA programming project.

As discussed above, CORBA IDL defines the interfaces for the services being developed. We have found that developing the IDL as an in-class exercise strongly promotes student understanding and engagement in the project. The instructors have always approached this exercise with (what they consider to be) an ideal IDL already outlined, but the students have often come up with good ideas and innovations during the class discussion which are then incorporated into the model. Part of the students' insights come from their experiences with computing, which are often quite different from the instructors'. For example, many students have in depth experience using on-line chat servers and game servers. As such, they bring practical insights to the discussions that the instructors may have overlooked.

We have had students develop both client-server and peer to peer projects. Client server projects are conceptually simpler. From the student/client's point of view the server is there just to meet their needs. In peer to peer applications the clients may generate requests, but must also be ready to respond to requests arriving from others. The code becomes more complex, and careful thought must be given to synchronizing the clients in many applications.

The remainder of this section discusses several of our more successful CORBA projects in Event Driven Programming.

The Chat System

Chat systems are an excellent example of relatively simple distributed systems. In a chat system, clients send messages to each other which are displayed as soon as they arrive. The clients log on and off a chat server, which is responsible for delivering the messages to all other logged on clients. Students implemented both a chat client and chat server. The chat client provides the user interface and sends messages to and receives messages from server. The server receives messages from the clients and distributes them to other registered clients.

The server provides the following methods:

1. `long register (in string clientName)` Clients register with the chat server and with a name server. The chat server maintains a list of registered clients. The name server maintains the same list of clients, but also keeps each client's IOR. The chat server queries the name server when it needs a reference to a client.
2. `void unregister (in string clientName)` This method removes the client from the list of registered clients
3. `void post (in Message message)` This method sends the message to all registered clients.
4. `void whisper (in string clientName, in Message message)` This method sends the message only to the named client.

The Client Interface

The term chat client is a bit misleading. Clients originate messages, but the server also pushes messages out to the clients. Because clients receive messages from the server, they implement an IDL interface, much like a server. The chat client implemented the following methods:

1. `void sendMessageToClient (in Message message)` This method belongs to a client. The server uses this method to send a message to the client.

2. `void whisperMessageToClient (in Message message)` This method, too, belongs to a client. It is used by the server to whisper a message just to a particular client.

There is no technical reason to have two methods implemented by the client, as they both exist for the client to receive messages from the server. At our students' suggestion, we included two because they wanted to have their clients treat a whispered message differently than a broadcast message.

The chat project was a big success. In a class of twelve students all completed both the client and the server. Because each client and each server implemented the same interface, we were able to mix and match clients with servers. All twelve clients worked successfully with all twelve servers.

The Connect 4 Client and Server

Connect 4® is a game produced by Milton Bradley. The objective of the game is to get four checkers in a row vertically, horizontally or diagonally. Players alternate turns, trying to line up four of their checkers, while blocking their opponent from getting four in a row. A turn consists of dropping a checker into a column from the top of the board. The checker falls to the lowest unfilled location in that column. A GUI for one Java implementation of the game is shown in Figure 2.

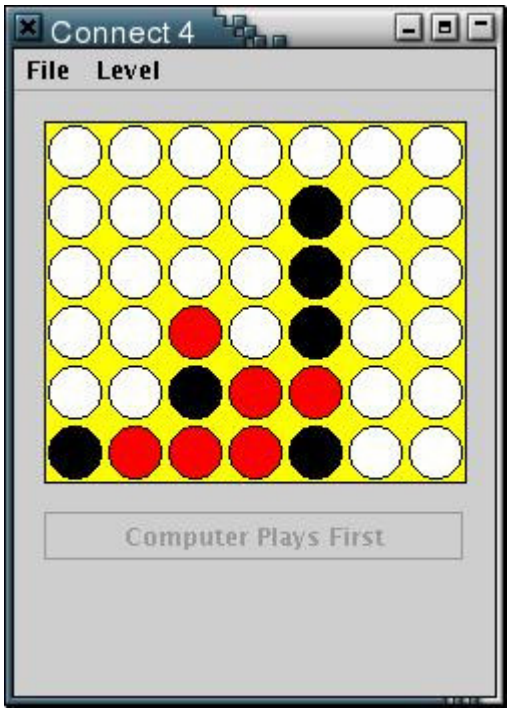

Figure 2. This figure shows a partially completed Connect 4 game. The red player must play on top of the three black checkers to prevent black from winning on its next move.

We have used Connect 4 for several different projects in Event Driven Programming. Early in the semester students developed a solitaire Connect 4 program. The computer served the opponent. The students were given a basic game engine, including minimax code and a dumb static evaluation method. Their primary task was to develop a GUI front end to the system. They were also allowed to try to improve the computer's static evaluation method, but no credit was associated with this task.

Later in the semester, they were assigned to modify their program into a CORBA based distributed game. The goal was to use CORBA to develop an on-line human client, an on-line computer client and a Connect 4 server. All of the logic for their clients had already been developed, but they had to pull apart their original code into two separate clients. Both clients implemented the same IDL interface, so it was possible to have two humans play each other, or have a human play a computer. It was also theoretically possible to have a computer play a computer, but this was not part of the assignment.

Initially, the instructors gave the students a Connect 4 server. The role of the server in this application is strictly to help clients find each other. Once a game is started, the clients played without intervention from the server. By giving the students a working server, they could develop and test their clients first. Once their client code was working, they could develop their own server.

This project was significantly larger than the Chat project, but the students had a large code base from which to work. The students clearly struggled with the project. Several deadline extensions were given for the CORBA implementations, and in the end, several students still failed to finish the assignment completely. Student evaluations stated that this was the first time they had ever been expected to understand and expand a relatively large existing code base.

Advanced CORBA Topics

The CORBA specification is large. A few weeks of lecture is only enough to give students a basic introduction. We have introduced students to a few of the more advanced features of CORBA without requiring them to use those features in their projects.

We have demonstrated how CORBA programs written in different languages can work together seamlessly. For example, we have developed CORBA servers in C++ and the corresponding clients in Java. Students have seen how a single IDL file translates to source code in each language.

We have also discussed the CORBA event demon. The CORBA specification includes an event demon that sits between event sources and handlers. The demon queues events for later delivery. Currently, Java's CORBA implementation does not come with an event demon, but there is one included with many other CORBA implementations. The advantage of using the event demon is that it further decouples event sources and handlers, potentially making the application even more robust.

Bug Patterns

We have begun collecting data on the most frequent problems students encounter when developing event driven systems. Students are asked two simple questions after they turn in each assignment, "What were the toughest one or two bugs/problems you faced when developing this project?" and "How did you finally resolve this problem?" The goal of the survey is to improve our course by determining where and how students struggle. In a recent section of Event Driven Programming, eight students completed the survey about their CORBA projects.

The most frequent problem encountered was initially registering the client with the server. Four students suggested that they had trouble with this initial step. This material was presented in class as something that could be cut and pasted between applications with only minor modifications. Connecting to a server is done in many applications and the code in each case is almost identical. While this is the approach the instructors have taken repeatedly, it did not seem to work for the students. On the other hand, students consistently claimed that they resolved this problem by looking at our code examples.

The next most frequent problem, recorded by three students, was coordinating the moves between the two clients when playing a game. This is a nontrivial problem. For example, in the non-distributed version of Connect 4, there is a while loop in which both the human and the computer make a move. In the distributed version, a client sends a move and then signals the other client that it is waiting for a response. Several students had problems with moves and signals being dropped. This left both clients in wait states where no progress could be made. Unfortunately, more than one student did not get this part completely debugged.

The two other problems that students reported were that CORBA error messages were much different than typical Java error messages and that testing and debugging were much more difficult than in traditional programming. Both of these are true statements. In future offerings of our course we hope to improve our discussion of testing and debugging CORBA systems.

Conclusion

CORBA proved fairly easy for the instructors to learn. There are many good resources, including texts and web sites. Students have struggled more with it. For many of them it

is their first experience developing distributed systems. We believe that it is still significantly easier for them to learn this using CORBA than using lower level communications primitives. CORBA provides a mature, stable infrastructure on which to teach event driven and distributed programming. Our event driven programming course has been a major success, with students regularly lobbying for more frequent offerings. Part of this success is due to the fun students have in developing using systems like CORBA.

References

1. Barnes, D. and Kölling, M. (2003), *Objects First with Java: A Practical Introduction using BlueJ*, Prentice Hall.

2. Brose, G., Vogel, A. and Duddy, K. (2001), *Java Programming with CORBA*, Wiley.

3. Deitel, H.M., Deitel, P.J. and Santry, S.E. (2002), *Advanced Java 2 Platform: How to Program*, Prentice Hall.

4. Henning, M. and Vinoski, S. (1999), *Advanced CORBA Programming with C++*, Addision-Wesley.

5. jGuru.com (1999), Introduction to CORBA, available on-line at: http://java.sun.com/developer/onlineTraining/corba/

6. Object Management Group (2003), CORBA-FAQ, available on-line at: http://www.omg.org/gettingstarted/corbafaq.htm

7. Puder, A., and Römer, K. (2000), *MICO: An open Source CORBA Implementation*, Academic Press.

8. Tari, Z. and Bukares, O. (2001), *Fundamentals of Distributed Object Systems: The CORBA Perspective,* Wiley.