CptS 464/564
Project 4
Causally Ordered Event Delivery
10% of final grade

16th November 2004

Assigned Thursday, Nov. 18, 2004

Due Thursday, Dec. 9, 2004

Introduction

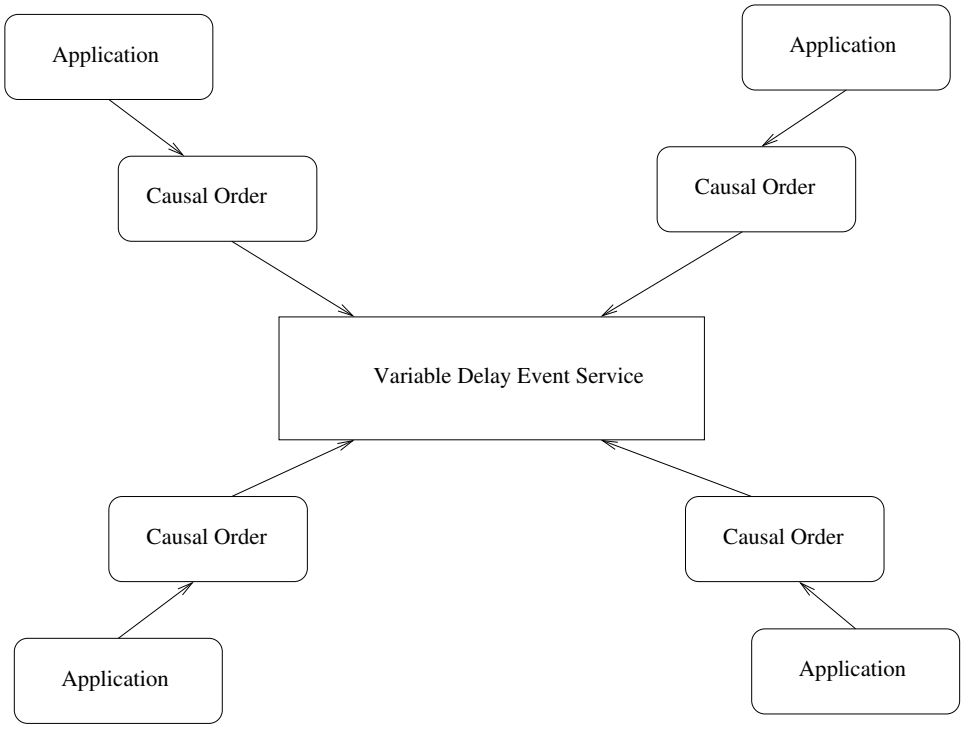
In this project you will implement machinery to deliver events to a collection of processes in causal order, using either the Lamport clock technique on page 58 (undergraduates) or the more precise vector clock technique on page 57 (graduates). The implementation exercise will give you experience with how causal ordering is achieved using a multicast communication channel with variable delays.

The project takes advantage of the CORBA EventService interface which allows event producers and consumers to be strung together in pipelines. It also uses the ability of CORBA any values to contain other CORBA any values.

A CORBA EventService is essentially a multicast-channel: messages sent to the EventService are multicast to all connected consumers. I will supply an EventService implementation that imposes variable delays on messages, but which will deliver messages from any particular sender to any particular receiver in order.

Your job is to implement a causal ordering (CO) layer to sit between the variable delay EventService and applications. Your code will conform to a subset of the EventService interfaces, so that the applications can use the layer in pretty much the same way that they would use a real EventService. Each application instance will have its own instance of this CO layer, which will process all incoming and outgoing messages for the application, making sure that messages are delivered in the correct order.

In order to demonstrate your causal ordering layer you will also create a sample application program and run several instances of it simultaneously.



The Reordering Event Service

The reordering event service that I will supply provides the for_consumers and for_producers methods of the EventServiceAdmin interface. Use these methods in the usual way to obtain a consumer and producer proxy object for your Causal Ordering component. Applications will use your component by having their push method invoked to receive messages and using the CO's push method to send messages. The CO component will in turn invoke the push method of the reordering event channel and and have its push method invoked by the reordering event channel to send and receive messages. To summarize, your applications will be push consumers and push suppliers of your CO components. The CO components will be push suppliers and push consumers as far as your applications are concerned; they will be also be push suppliers and push consumers of the reordering event service.

Applications can, of course, send and receive messages of arbitrary type using *any.* You should design the objects that are sent by different applications so that messages from different applications are distinguishable and that the order of messages sent is discernable. (The Car program from project 3 meets these criteria.) The CO component will need to *wrap* the application messages in another structure containing additional information and insert *that* structure into another *any* which is sent through the reordering event service. When the CO receives a message from the reordering event service it

will remove the wrapper, using the information it contains to decide when to deliver the message (which is, of course, the *any* inside the wrapper).

Summary

You are building two things: the application and the CO component. The purpose of the application is to help you demonstrate that your CO component works correctly.

The application must

- send messages containing enough information to identify the sender as well as a sequence number (the Car program from Project 3 meets these criteria).
- print messages in the order sent and received, including the senders identity and the sender's sequence number (the sniffer programs from Project 3 meet these criteria).

Thus one approach to the application is to combine the Car and Sniffer programs from Project 3. You will need to decide with what frequency you wish to send messages from your application.

The CO component must

- deliver messages to the applications in causal order using Lamport clocks (vector clocks for graduate students), even though the messages it receives from the reordering event service will have been shuffled.

Implementation notes

I suggest the following steps:

- design the IDL interface implemented by your CO for use by your application. At a minimum it will contain push methods invoked by your application and by the reordering event channel (you need not implement the EventService interface in all its generality; remember, you don't need to support multiple applications with one CO instance, nor do you need to support all the combinations of push/pull consumer/supplier).
- implement your application and a CO that doesn't attempt to do causal ordering but which wraps the application messages when sent and unwraps them when received. To this point you can use the standard EventService if you wish as the multicast channel for your project.
- implement causal ordering using Lamport clocks and finally, if you are a grad student, vector clocks. (Grad students need not hand in Lamport clocks but they are a good step along the way to vector clocks.) At this stage you must use the reordering event service to test your program.

You may assume that the reordering event service will always deliver messages eventually. That is, you do not have to worry about recovery from lost messages. You may also assume that it will deliver messages from each sender to each receiver in the order that they were sent (i.e. it implements a FIFO channel between each sender and each receiver).

An instance of the reordering event service can be started with

```
/net/niflab/orbacus/cs564/RES/ReorderService
```

which will start the service and write the IOR on standard output.

As always, *simplify.* The main purpose of this project is to implement the CO component. Everything else is secondary, so make decisions that will let you concentrate on what is most important.

This project should not pose any "plumbing" problems that you haven't already solved, but it does require you to implement a fairly tricky distributed algorithm and to design your own IDL interfaces to support that algorithm. So start early and make sure all the plumbing problems are addressed so that you can concentrate on the difficult part of the problem.

The other problem that you may face is deadlock. We will discuss this in class over the next several lectures.

As usual, do not leave unattended processes running on the NIF-lab machines.

What to turn in and when

The project is due Thursday, December 9, at class time. Turn in printed listings of your IDL, your CO, and your application, along with annotated printouts of execution output. The annotations should point out how your execution output demonstrates correct function of your project. Store the source and executable code for your project in your EECS home directory and do not touch the files after the due date. (Files that are touched after the due date may be considered late). Interviews with Harald will be arranged after the due date.