Designing and Implementing Inter-Client Communication in the O_2 Object-Oriented Database Management System

Antonio Carzaniga, Gian Pietro Picco
and Giovanni Vigna
CEFRIEL,
via Emanueli 15, 20126 Milano (Italy),
Tel. +39-2-66100083, Fax +39-2-66100448,
E-mail:picco@mailer.cefriel.it

Abstract

One of the requirements for an object-oriented database to support advanced applications is a communication mechanism. The Inter-Client Communication Mechanism (ICCM) is a set of data structures and functions developed for the O_2 database, which provides this kind of service. Communication is achieved through shared persistent objects, implementing the basic idea of mailbox. One to one connections are established between different processes accessing the database; methods and data structure defined in the ICCM support connection setup, disconnection, and all the basic data transfer facilities. In this paper, we describe the concepts of ICCM and an overview of its implementation.
Keywords and phrases: object oriented database, client/server architecture, communication.

1 Introduction

Object-oriented databases are widely used in many engineering fields requiring a sophisticated data modeling system, like software engineering environments and CAD applications. In such environments complex data are shared by many persons; cooperation among developers and interaction with tools is a critical issue.

Nearly all OODBMS currently available are based on a client/server architecture. In particular, we are using O_2 [Deu91] which is based on a client-oriented concept that gives computational power to each client; others (like GemStone [BOS91]) prefer to have methods executed by the server according to a more centralized schema.

Our experience in building a process-centered software engineering environment (PSEE) has pointed out the importance of distribution of data and computations among clients [BBFL93]. The management of such a distributed model requires complex data sharing and communication, in order to address both architectural and tool integration issues.

An *ad hoc* communication system other than the standard UNIX ones is necessary if we want to exchange complex data among clients. Thus, we isolated the problem of communication from the context of our application, and developed a stand-alone, general-purpose O_2 module, called ICCM (Inter Client Communication Mechanism), intended to be an extension of the services provided by the OODBMS.

These services do not depend upon the particular application. They may be of any use, possibly after slight changes, whenever complex data have to be passed between user applications allocated to different clients.

In this paper we present the idea behind our ICCM, together with an overview of the provided services. Since it is based on the O_2 OODBMS, in Section 2 we provide a short description of this system. In Section 3 the underlying ideas and concepts are described, while Section 4 explains the data structures and services provided by the ICCM. Section 5 gives some insights on how ICCM can be used in a real user application and how such an application has to be structured in order to interact with ICCM. Finally, Section 6 draws some conclusion and highlights some issues about our future work involving the ICCM.

2 The O_2 OODMBS

O_2 [O293], [Deu91] is a distributed Object-Oriented Database Management System, based on a client-server architecture.

The logical structure of an O_2 data base is bound to a *schema*, i.e. a collection of names and definitions of classes, types, applications, objects and values. There may exist any number of logically separate schemas at one time.

A *base* is a collection of data whose structure conforms to the structural definition in a schema. Several different bases might be associated with each schema.

Data manipulation is achieved using the O_2C language, an extension of ANSI-standard C, as well as the O_2SQL, an *ad hoc* database object-oriented query language, whose syntax is styled on IBM SQL standard, and which is likely to become the SQL standard for OODBMS. O_2 also provides an interface towards standard programming languages, namely C and C++.

Data are represented by *values* and *objects*. A value is an instance of a given *type*. A type is a generic description of a data structure in terms of atomic types (integers, characters, and so on) and structured types (tuples, sets, and lists). An object is an instance of a given *class* and incapsulates a value and the behaviour of that value. The behaviour of an object is fully described by the set of *methods* attached to it.

An object or a value may be given an identifier, i.e. a *name*, by which O_2 commands, methods, and application programs may refer to it quickly and specifically. Such name is global within the schema.

Objects and values in the system can be either *persistent* or not. An object is persistent if it remains in the database after the successful termination of the transaction which created it. Persistence is granted as follows:

Every named object in the database is persistent, and every component of a persistent object is persistent. No other objects are persistent. The same rule applies to values.

Thus, named objects and named values are the roots of persistence. That is, they are used as handles from which every persistent object or value can be reached.

Every update to persistent data must be performed within a transaction. If two clients access the same object or value in transaction mode, the locks obtained by the first client force the second to wait. Thus, critical sections corresponding to updates should be limited in time, and should not involve several objects, in order to improve performance and avoid deadlocks[1].

Objects and values not bound to a persistency root are automatically garbage-collected at the end of a transaction.

3 The underlying idea

Common UNIX inter-process communication mechanisms like pipes and sockets cannot be used for establishing a communication among O_2 clients, since these mechanisms are suitable only to transfer non-structured data, like integers or strings.

Our goal, on the contrary, is to exchange true O_2 objects. O_2 objects may have any internal structure, consequently, in order to transfer an object, e.g. via a socket, it would be necessary to transform it in a sequence of atomic values and then reassemble it again upon receipt. This solution is not feasible

[1] In the current implementation of O_2, locks are associated with pages rather than with objects. This may lead to the odd situation in which clients working on completely different objects within the same base can experience deadlocks. O_2Technology is steering towards replacing page locking with object locking.

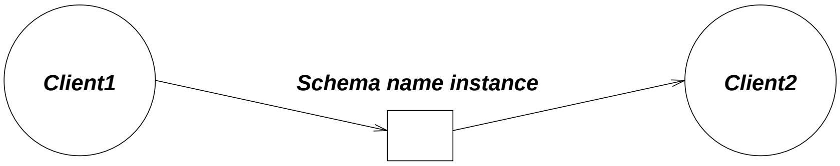

Figure 1: Using a persistent object to communicate two clients.

since it would be very inefficient and, in any case, it would produce *a copy* of the original object. In addition, it is not possible to simply pass to another client the pointer to a shared object, because this pointer (i.e., the object identifier) is unknown (it exists in the O_2 engine, but is not accessible by the user environment) and, moreover, the address spaces of the two clients are different.

Intuitively, since we can use neither UNIX mechanisms, nor dedicated O_2 primitives, the only feature currently available to establish a communication is the *database itself* or, in other words, *persistent objects* which can be accessed by both clients, as explained in Section 2.

The raw solution using this concept implements the mechanism illustrated in Figure 1: the message sender puts the O_2 object into a named (i.e. persistent) variable, whose name is known by both the sender and the recipient. The recipient can read the contents of the persistent named variable and operate on it.

This initial solution suffers of some drawbacks, if applied as it is within a concurrent multi-client perspective:

- The shared object name must be hard-wired within clients code.

- No parallelism is allowed, since after a communication session is started by the message sender by filling the named variable, no other client can update this variable (without causing data loss) until the recipient has actually read the variable.

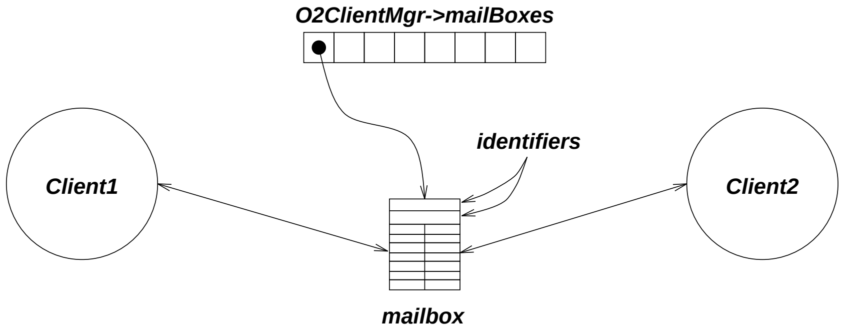

Figure 2: Two clients communicating through the ICCM.

We could improve parallelism increasing the number of named variables through which the information is exchanged. However, this solution implies that the number of statically defined names grows quadratically with the number of clients.

On the contrary, a highly parallel and dynamic mechanism is desirable, in order to limit the overhead imposed at run-time on the environment, and to provide a highly flexible communication mechanism. Figure 2 shows the Inter-Client Communication Mechanism (ICCM), an enhancement of the basic mechanism described above, that we designed to improve the communications among O_2 clients [PV93]. Once again, there is an object, called `O2ClientMgr`, that is given a name in the clients schema, and is therefore visible within both clients scopes, provided that they import the ICCM schema. `O2ClientMgr`, the ICCM manager, is not an information container itself, instead it owns a list of *mailboxes*.

The mailbox objects are the communication channels through which information flows from a client to another. Each mailbox contains two message queues, which at communication time contain the messages exchanged by the communicating clients in both directions.

`O2ClientMgr` provides services (i.e. methods) to correctly create and manage these mailboxes when establishing a connection between clients.

Communication within ICCM is split in three phases: i) connection, ii) message exchange, iii) disconnection. We will describe here only the first phase, since it is the most important. A detailed description can be found in [PV93].

The connection phase follows a defined sequence of steps, which are coded into `O2ClientMgr` services. For the caller, the sequence is:

1. Create a new mailbox object.
2. Post a connection message on the newly created mailbox.
3. The connection may involve a client already active in ICCM, or a client that has to be invoked, and then connected.
   (a) In the former case, the caller client searches the `O2ClientMgr` mailbox list for an existing mailbox already bound to the callee, and posts a message of connection request, containing its own identifier. In other words, it "hires" a mailbox currently used by the callee to communicate with another client.
   (b) If the connection involves a client that does not yet exist, the O_2 client is created by execution of a specific program, no matter if it is a C or C++ application interfaced with O_2 or a straight O_2 client. During the client start-up phase, the new client is given its own identifier, and is also informed about the identity of the client that invoked it.
4. Wait for the acknowledgement, which must be sent by the callee through the mailbox created in 1.

If the callee already exists, it will simply find in one of its mailboxes the connection request, containing the identifier of the caller.

Otherwise, the newly created client is informed by the creation mechanism about the identity of its creator. In both cases, the callee will:

1. Search the `O2ClientMgr` mailbox list for an existing mailbox whose identifiers match both caller and callee identifiers.

2. Read the message on the mailbox, checking if it is the right connection message.

3. Send an acknowledge back to the caller, through the same mailbox.

Since mailboxes are attached to `O2ClientMgr`, which is a persistent object, every mailbox is persistent too. Nevertheless, after the first scanning of `O2ClientMgr` mailbox list, performed at communication setup time, the callee client creates a local direct reference to the mailbox, thus allowing fast access.

Since every read/write operation on a persistent object has to be performed within a transaction, the situation described above has an important consequence: *once we have retrieved the correct mailbox M from the global data mailbox list, we can lock M exclusively without locking the whole persistent structure.*

In other words, we use the `O2ClientMgr` mailbox list only to ensure persistency and visibility of mailboxes to all the clients, but we can lock just one mailbox at a time (except for the relatively unfrequent communication setup phases), thus maximizing parallelism[2].

In the next section ICCM features and services are discussed in greater detail.

[2]This will have an even greater impact when O_2 will support object locks, as pointed out in Section 2.

4 ICCM features and services

ICCM is actually an O_2 schema (i.e. class definitions together with persistent objects name definitions) providing communication services. ICCM has to be imported into the user application schemas that intends to use inter-client communication facilities. In the following subsections we describe the basic components of our mechanism: mailboxes, messages, and the communication service manager.

4.1 Mailboxes

ICCM mailboxes are not just like the well-known mailboxes. Usually mailboxes are associated with recipients. Whenever a message has to be delivered, the message queue, representing the mailbox, is reached using a reference to the recipient, e.g. to send a message to `RcptObj` we would use `RcptObj->mailbox->putmsg(msg)`.

We could not afford this solution because write operations must be performed during a transaction, in order to make changes in the recipient message queue visible to the recipient itself; hence, accessing the recipient mailbox would lock the recipient object altogheter.

We decided to cluster the queues, acting as mailboxes between two objects, in a unique data structure, in order to avoid the recipient object locking. References to proper queues, during read/write operations, is accomplished using a couple of unique identifiers associated to the communicating objects. Adopting this solution, accessing the mailbox queues does not involve any of the communicating objects, allowing a greater level of parallelism and better performance.

By the way, ICCM mailboxes own methods to insert and retrieve messages, and to test emptiness, this way implementing an easy-to-use two-way communication link. ICCM mailboxes allow $1:1$ connections only, for performance reasons. In fact, since mailboxes are persistent, write operations have to be performed during a transaction; if $1:n$ connections were possible, any of the several clients, accessing a queue during a write operation, should synchronize with the others. Implementing $n:n$ connections could only worsen the problem.

ICCM mailboxes can be generally used to manage queues of data within an application. Nevertheless, as we mentioned in Section 3, they are used mainly within ICCM as the communication channels linking two clients.

4.2 Messages

Mailboxes can contain only O_2 messages, which can be logically divided into two classes:

- **User-defined messages.** Client communication is not established for its own sake, but to allow service invocation and data exchange. Consequently, user-defined messages can be used both to control clients behavior and to carry data.

- **Built-in messages.** They are used for "system" operations, like setting up or removing a connection, terminating a client execution and so on. They are used within `O2ClientMgr` methods, and they are also available to the user.

What follows is the ICCM class definition for an O_2 message:

```
class O2Message public type
tuple(msgName: string)
      data: list(Object))
end;
```

The `Object` type is the root of the O_2 object hierarchy; consequently, due to polymorphism and late binding, every O_2 object fits in the `data` field, no matter what its type is: it is the receiver job to correctly handle such objects.

Note that the current solution has both advantages and disadvantages: polymorphism allows to pass any kind of object (no matter what is its complexity and structure), but if you need to transmit bare *values*, you need to envelope them into *ad hoc* classes. We are investigating solutions to allow a "smart" inclusion of values into ICCM messages, in order to allow *generic* data to be exchanged.

4.3 ICCM manager

The ICCM manager provides all the data structures and services needed to establish and manage a communication between clients.

It is a named persistent object of type `O2ClientMgrObj`. The data structure is made of a list of mailbox objects representing all the communication channels connecting client pairs during ICCM execution. This structure is centralized, constituting the only bottleneck to communication management, because the access to data structure have to be serialized. Anyway, since such access is needed only at connection setup, the additional overhead is little, and generally negligible if compared with the longer time needed for client actual invocation.

The most important services provided by the ICCM manager are:

- *Create a new connection.* `O2ClientMgr` provides two distinct methods to get the job done: `setupO2Client`, which performs both the invocation of the callee and its connection to the caller, and `connectToO2Client`, which establishes a connection between the caller and an already running client.

- *Search for the mailbox to connect with.* As we mentioned in Section 3, as soon as the callee is either invoked or requested for a new connection, it has to scan the mailbox list in `O2ClientMgr` searching for the mailbox created by the caller and that ought to be used in order to communicate.

- *Remove a mailbox from the list.* Actually discards a communication channel between two clients. We suggest that the client that is in charge of performing the removal should be the caller, after the callee has communicated that all clean-up operations have been correctly performed.

5 An Example

In this section we describe a "toy" example, whose purpose is to show how mechanisms described in previous sections can be used within actual O_2C or C++ programs. We will refer to Figure 3. Three clients are involved:

- `Client1` and `Client2` play the role of "master" O_2 clients, e.g., they could be applications performing some kind of independent computation.

- `Tool` represent a "slave" service provider, receiving both connection and service requests from "master" clients and returning outputs to them.

Figure 3 represents the following situation. `Client1` invokes `O2ClientMgr->setupO2Client`, in order to actually invoke the `Tool`. Hence, the invocation of this method determines the creation of a new mailbox already con-

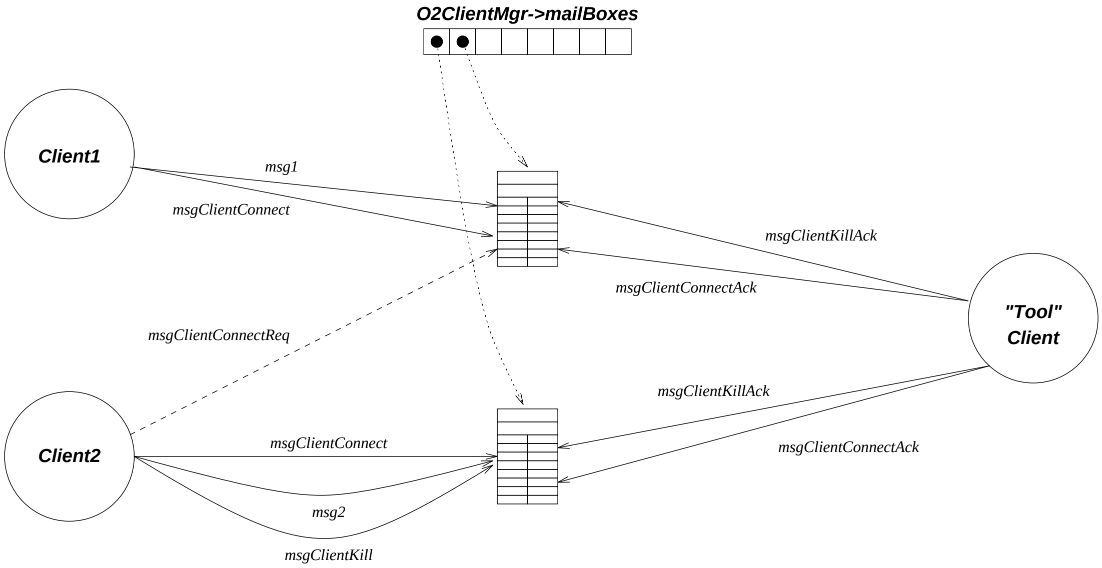

Figure 3: An example.

taining the message `msgClientConnect`. As soon as `Tool` is running, it takes its own identifier and the caller identifier from the command-line parameters, and executes method `O2ClientMgr->seekForMailbox(callerId,calleeId)`, which searches the centralized mailbox list for a mailbox carrying the corresponding identifier pair. When this is found, a connection acknowledge is sent back to the caller.

After the connection is succesfully setup, `Client1` posts a user message (`msg1` in the figure) to the `Tool`. Meanwhile, `Client2` requests a connection, too. Since the `Tool` is already up[3], `Client2` has to execute the method `connectToO2Client` instead of `setupO2Client`. This method implicitly (i.e., transparently to the user) searches the `O2ClientMgr` mailbox list for a mailbox already connected to `Tool`, and posts a `msgClientConnReq`

[3]We assume that `Client1` and `Client2` are client of the same application, thus they are aware of what clients have already been launched

in it (instead of the `msgClientConnection` that would be posted in a newly created mailbox). After receiving the message, `Tool` executes again the method `O2ClientMgr->seekForMailbox`, thus establishing a new connection.

Then, `Client2` decides that `Tool` is no longer needed, neither by `Client2` nor by the other "master" clients. Thus, `Client2` terminates the execution of `Tool`. This is accomplished by sending a kill message to the `Tool`, namely `msgClientKill`. The `Tool`, upon receipt of the signal, performs the correct clean-up operations, then it broadcasts a `msgClientKillAck` to all clients connected with it, to warn them that it is going to terminate. Among the clients receiving this message, there is also `Client2`, the client which started the termination process, that is now ready to remove the `Tool` mailbox from `O2ClientMgr` list.

Note that all message processing is made synchronously, by polling the mailboxes connected to clients. In this respect, ICCM resembles somehow programming with plain X, in an event-driven programming paradigm. In fact, the main polling loop has to contain conditional tests in order to recognize the received messages and execute the corresponding "callback".

6 Conclusions and future work

We presented here an Inter-Client Communication Mechanism for the O_2 OODBMS. The need for such a mechanism stemmed from some issues arosen in developing a process-centered software engineering environment. Nevertheless, the findings and the outcomes of our experience might be generalized to a wider range of applications which use an object-oriented repository, possibly even different from O_2. We described here both the idea underlying our mechanism, and an overview of the data structures and services composing it. We are currently using ICCM in the development of our PSEE, both testing and enhancing its features.

Future work will include:

- the developing of a sound technique to allow generic messages (including any value or object) to be passed through mailboxes;
- mailboxes allowing $1:n$ communication;
- the investigation of other methods for Inter-Client Communication in O_2, based on new O_2 features, provided by future releases. In particular, O_2Technology plans to introduce active database issues, as described in [OIC+93]. This feature will be based on the *trigger* concept, which should allow to achieve asynchronous message passing.

Acknowledgments

The authors wish to thank Ing. Luigi Lavazza, Dr. Sergio Bandinelli, and Prof. Alfonso Fuggetta for their comments and insights, which have been helpful both in developing ICCM, and in writing and refinining this paper.

References

[BBFL93] Sergio Bandinelli, Luciano Baresi, Alfonso Fuggetta, and Luigi Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository using Object-Oriented Technology. In *Proceedings of the International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, November 1993.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10), October 1991.

[Deu91] O. Deux. The O_2 System. *Communications of the ACM*, 34(10), October 1991.

[O293] O_2Technology. *The O_2 User Manual.* O_2Technology, 1993. Release 4.3.

[OIC+93] O_2Technology, Inria, Cefriel, University of Frankfurt, and University of Grenoble. Architecture and functionalities of the goodstep repository as implemented in the first prototype. Technical report, O_2Technology, 1993. Esprit Project 6115 (GOODSTEP) deliverable.

[PV93] Gian Pietro Picco and Giovanni Vigna. The SPADE Way to Inter-Client Communication in O_2. Technical report, CEFRIEL, 1993. Technical Report N.99401.