[6] Antonio Carzaniga and Giovanni Vigna. The Design and Implementation of SPADE-1 2.0. Technical report, CEFRIEL, June 1994.

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

[8] 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.

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

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

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 the idea underlying the mechanism, an overview of the data structures and a brief description of the services exported. We are currently using ICCM in the development of our PSEE, both testing and enhancing its features.

Future work will include:

- the development of a sound technique to allow generic messages (including any value or object) to be passed between clients;
- the integration of new O_2 features, provided by future releases. In particular, O_2Technology plans to introduce active database issues, as described in [8]. This feature will be based on the *trigger* concept, which should achieve asynchronous message passing.

Acknowledgments

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

References

[1] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The Architecture of the SPADE-1 Process-Centered SEE. In *3rd European Workshop on Software Process Technology*, Grenoble (France), February 1994.

[2] 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.

[3] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sergio Grigolli. Process Enactment in SPADE. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992. Springer-Verlag.

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

[5] Antonio Carzaniga. O_2 sockets. Tecnical report, CEFRIEL, Milano (Italy), March 1994.

7 Evolution of the ICCM

We designed the ICCM for the purpose of supporting the distributed architecture of SPADE-1 2.0 [3, 1, 6]. The evolution of this architectural schema revealed some issues related to the ICCM. In particular, the ICCM serves the enactment environment of SPADE-1 2.0, supporting computation distribution, and the data integration mechanism, passing O_2 parameters to integrated tools. Performance and small critical sections are requested by the enactment environment. Flexibility and easy connection set-up are the requirements posed by the data integration mechanism.

7.1 A new ICCM

In order to fulfill the needs of performances and flexibility, we re-designed the ICCM. The new ICCM resembles much the concepts of UNIX sockets [5].

It features *port* objects that are the well-known access points for communicating clients. Two straightforward primitives set up the connection, namely, `connect` and `accept`. When an O_2 client issues a `connect` on a port, two connection end-points are created: one is returned to the calling client, the other one is enqueued in a list associated to the port. When another client calls the `accept` method on the same port, it retrieves the enqueued connection end-point, so that the communication link is completely established. Once the connection is available, methods `read`, `write` and `isEmpty` can be used to exchange data.

This mechanism passes object references, one at a time. There is no particular message format. It does not need any identifier. No handshake protocol is defined: it is up to the application programmer to coordinate the communication session. No explicit shutdown procedures are defined: the channels are persistent as long as the two communicating parties keep their references. Furthermore, the schema of the new ICCM comes with no pre-defined persistent objects; again, the application programmer is responsible for defining his/her own ports.

Summing up, the new ICCM is faithful to these principles:

- It is simple and, thus, very efficient.
- It provides low-level primitives. Thus, it is flexible.

Almost any object-oriented communication protocol can be built on top of the new ICCM layer.

8 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 concerning the architecture of a process-centered software engineering environment.

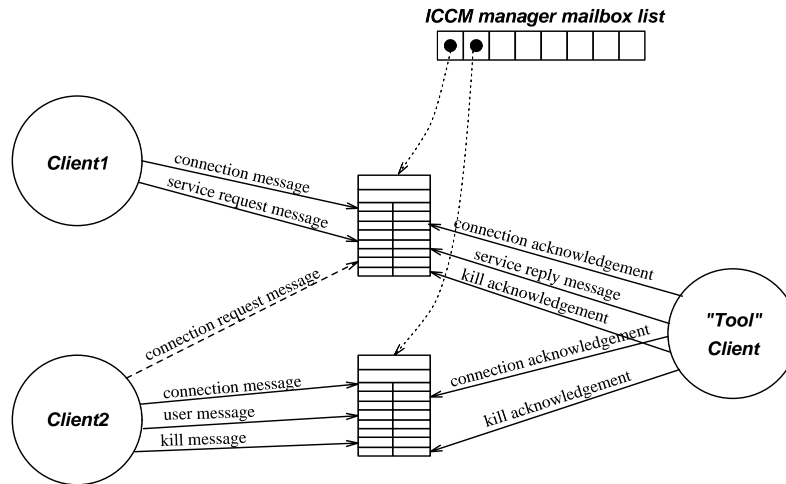

Figure 3: An example.

Since `Tool` is already up[3], `Client2` has to call the method `connectToO2Client` instead of `setupO2Client`. This method performs the following actions:

1. It creates a new mailbox object, tagged with `Client2` and `Tool` identifiers, and filled with a connection message.

2. It searches the ICCM manager mailbox list for a mailbox already connected to `Tool`, and posts a connection request message in it.

After receiving the connection request message, `Tool` calls the method `seekForMailbox` on the ICCM manager. The method retrieves the newly created mailbox and posts an acknowledge message in it, thus establishing a new connection.

At a given time, `Client2` decides that `Tool` is no longer needed, neither by `Client2` nor by other "master" clients. Thus, `Client2` terminates the execution of `Tool` by sending a kill message. `Tool`, upon receiving the message, performs the correct clean-up operations. Then, it broadcasts a message of acknowledgement 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`, that is now ready to remove the `Tool` mailbox from the ICCM manager 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, following 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 "callbacks".

[3] We assume that `Client1` and `Client2` are clients of the same application. Thus, they are aware of which clients have already been invoked.

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

Since mailboxes are attached to the ICCM manager which is a persistent object, every mailbox is persistent too. Nevertheless, after the first scanning of ICCM manager mailbox list, performed at communication set-up 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 mailbox list, we can lock M exclusively without locking the whole persistent structure.*

In other words, we use the ICCM manager 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 set-up phases), thus maximizing parallelism[2].

6 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 applications.

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` represents a "slave" service provider, receiving both connection and service requests from "master" clients and returning outputs to them. Obviously, it is an O_2 client.

Figure 3 represents the following situation. `Client1` calls the `setupO2-Client` method on the ICCM manager, in order to actually invoke the `Tool`. Hence, a new mailbox is created and filled with a connection message. As soon as `Tool` is running, it calls the method `seekForMailbox` on the ICCM manager. The method retrieves the callee and the caller identifier from the `Tool` process environment, and it searches the centralized mailbox list for a mailbox carrying the corresponding identifiers pair. When the mailbox is found, a connection acknowledge is sent back to the caller.

After the connection is successfully set up, `Client1` posts a service request message to `Tool`. `Tool` retrieves the message, performs the service, and sends a reply message. Meanwhile, `Client2` requests a connection to client `Tool`.

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

- *Communication shutdown.* The ICCM manager provide means to discard a communication channel between two clients. ICCM clients can remove mailboxes calling the **removeMailbox** method on the ICCM manager. Since *in itinere* data would be lost, the application programmer is responsible for the management of a data-loss free shutdown protocol.

5 Communication

An ICCM session has 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 [9].

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

1. Create a new mailbox object.
2. Post a connection message in the newly created mailbox.
3. The connection may involve a client already using the ICCM, or a client that has to be invoked, and then connected.
   (a) In the former case, the caller client searches the ICCM manager 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 invoked. 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 client already exists, it will simply find in one of its mailboxes the connection request message, containing the identifier of the caller.

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

1. Search the ICCM manager mailbox list for an existing mailbox whose identifiers match both caller and callee identifiers.
2. Read the message from the mailbox, checking if it is the right connection message.

4.2 Messages

Mailboxes can contain only ICCM messages, which are O_2 objects. They can be logically divided into two classes:

- *User-defined messages.* They are general-purpose object containers.
- *Built-in messages.* They are used for "system" operations, like setting up or removing a connection.

ICCM messages contain a `name`, which is a string, and a `data` field, which is a list of objects of class `Object`. The `Object` class is the root of the O_2 class 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 up to the receiver to correctly handle (cast) 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 communication between clients.

It contains, in its data structure, a list of mailbox objects representing all the communication channels connecting client pairs during ICCM execution. This structure is centralized. It constitutes the only bottleneck to communication management, because access has 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:

- *Connection set-up.* The ICCM manager object provides two distinct methods to set up a communication link: `setupO2Client`, which performs both the invocation of the callee O_2 client and its connection to the caller, and `connectToO2Client`, which establishes a connection between the caller and an already running client.
- *Mailbox retrieval.* As soon as the callee is either invoked or requested for a new connection, it has to scan the mailbox list owned by the ICCM manager, in order to retrieve the mailbox created by the caller, and that ought to be used for further communication. The ICCM manager provides the method `seekForMailbox`, which searches the mailbox list for a mailbox matching the caller and callee identifiers. Identifiers management is transparent to the user.

called ICCM manager, that is given a name (i.e., `O2ClientMgr`) in the clients schema, and it is therefore visible within clients scopes. 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. The ICCM manager provides services (i.e., methods) to correctly create and manage these mailboxes when establishing a connection between clients.

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 schema whenever it is necessary 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. For example, to send a message to `A` one would use `A->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 altogether.

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. Each communicating client owns a unique identifier, assigned by the ICCM manager at setup time. The mailbox queues are tagged with the identifiers corresponding to the connected clients, so that read/write operations access the proper queue. 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.

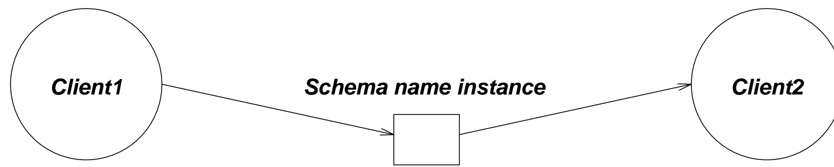

Figure 1: Using a persistent object to communicate two clients.

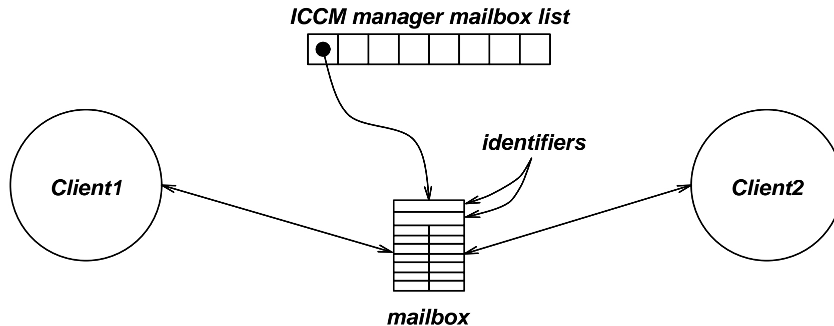

Figure 2: Two clients communicating through the ICCM.

The raw solution using this concept implements the mechanism illustrated in figure 1: the message sender puts an 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 from 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. After the message sender begins a communication session by filling the named variable, no other client can update this variable (without causing data loss) until the recipient has actually read the variable.

We could improve parallelism by increasing the number of named variables that support data exchange. Still, the number of common variables, would be statically limited. Furthermore, it would be proportional to the square of the number of communicating 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 communication among O_2 clients [9]. Once again, there is an object,

the behavior of that value. The behavior 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 referenced.

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 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 character stream and then reassemble it again upon receipt. This solution is not feasible 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 identifier of an object, because object identifiers exist only in the O_2 engine, and they are not accessible from the user environment.

Intuitively, since we can use neither UNIX mechanisms, nor dedicated O_2 primitives, the only mean currently available to support communication is the *database itself* or, in other words, *persistent objects* which can be accessed by every client, as explained in section 2.

[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.

computations among clients [2]. The management of such a distributed model requires complex data sharing and communication, in order to address both architectural and tool integration issues.

A specific communication system other than the standard UNIX ones is necessary if we want to exchange complex structured 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 whenever complex data have to be passed between user applications allocated to different clients.

In this paper we present the idea behind 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 describes the ICCM communication protocol. Section 6 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. Section 7 outlines ICCM evolution. Finally, Section 8 draws some conclusion and highlights some issues about our future work involving the ICCM.

2 The O_2 OODMBS

O_2 [10, 7] 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, functions, 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* object-oriented query language, whose syntax is styled on IBM SQL standard, and which is likely to become the SQL standard for OODBMSs. O_2 also provides interfaces 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 encapsulates a value and

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)

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 set-up, disconnection, and all the basic data transfer facilities. In this paper, we describe the concepts of the 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 OODBMSs currently available are based on a client/server architecture. In particular, we are using O_2 [7] which is based on a client-oriented concept that gives computational power to each client. Others (like GemStone [4]) 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