The Design of a System for Distributing Shared Objects*

J. DOLLIMORE,† E. MIRANDA AND WANG XU
Department of Computer Science, Queen Mary and Westfield College, Mile End Road, London E1 4NS

The primary motivation for the work described in this paper is the design of a platform for building applications that are intended for use by a group of people. The paper introduces the requirements for applications in which people use workstations on a local area network to share information and to communicate with one another. The paper discusses how we addressed these requirements in the context of three sample applications – a departmental database, a room-booking program and a shared spreadsheet.

Received May 1991, revised July 1991

1. INTRODUCTION

This paper discusses the design and implementation of a distributed object-oriented programming environment.

Our motivation for designing this environment is to provide a platform for building applications that are intended to be used by a group of people who would like to use computers for communication and sharing information. We assume that users have access to workstations connected to a local area network.

The type of application we have in mind is illustrated by the following examples:

- simple shared databases, e.g. address lists, information about people in a department;
- shared diaries and booking systems;
- shared spreadsheets.

Such applications are characterised primarily by supporting information that is shared by the members of a group. Their secondary characteristic is that each application allows users to manipulate quite complex information structures. We use the term object* to refer to a unit of information within an application, such as a week in a diary or a cell in a spreadsheet. We note that objects belonging to a particular application generally contain connections to other objects; for example, a spreadsheet cell can contain a reference to the objects containing its value and its formula.

We make no assumption as to whether users access objects at the same time as one another or at different times. In the case of shared databases, users will access objects whenever they require them and will add new information whenever it becomes available. The fact that more than one user may access the same object simultaneously is generally of no importance to the users involved except in so far as it may cause conflict.

When users choose to access a set of objects simultaneously, each user can have an independent view of the information. Although the views of different users may overlap in that both contain some of the same objects, there is no assumption that they would share a view. However, this does not prevent users from having the same views as one another, as for example in multi-user editing systems such as ShrEdit[14] and the multi-user drawing program described by Hagsund.[9]

Our main design goals relating to the support of shared applications are to be able to:

- share data that is inherently distributed;
- support user interfaces that are appropriate for shared information and have acceptable performance.

2. REQUIREMENTS

We have identified the following requirements for a programming system that provides a platform for building distributed applications for workstation users on a local area network:

(i) it must be possible to place shared objects in any computer that runs our software – this may be in a user's workstation or in a server; in addition it should be easy to make objects available for sharing;
(ii) location and access transparency – the application builder should be able to design application software without concern as to the location of the objects accessed;
(iii) information should be presented to users through an interactive user interface allowing them to view and manipulate the information as easily as in today's single-user applications; a side effect of this requirement is the need to place replicas of shared objects in users' workstations;
(iv) several people must be able to view and edit the same objects simultaneously and observe one another's effects;
(v) when objects are accessed by more than one person at a time, the overall effects should be consistent;
(vi) privacy and protection of information – unauthorised users must be prevented from seeing or altering objects that they are not intended to use in that way;
(vii) long-term reliable storage of objects is required – this requires a mechanism for transparent persistency in which certain objects are automatically preserved for as long as they are needed.

Our requirements and the experience described by other researchers has led us to choose an object-oriented programming environment. The object model has been

* This work is supported by the Esprit SPIRIT High Performance Workstation project.
† To whom correspondence should be addressed.
* It will become apparent in a later section that the notion of object also includes operations.

shown to be effective for constructing distributed programs in which objects that reside in different computers in a local area network communicate by means of messages (e.g. Arjuna,[15] Argus,[11] ANSA[8]). Object-oriented programming environments have been proved particularly effective in the design of interactive software (e.g. Analyst, Hypercard). In addition, object-oriented programming and databases have been shown to have potential for sharing data in group work.[7] The Emerald distributed programming language[10] and Distributed Smalltalk[1] are examples of distributed object-oriented programming languages.

Neither Emerald nor Distributed Smalltalk makes any attempt to deal with privacy, protection or long-term storage of data. Distributed Smalltalk provides the ability to construct interactive user interfaces, but has not addressed the issue of allowing changes made by one user to be provided immediately to other users of the same objects.

We have considerable experience building applications with Smalltalk-80 and have our own local implementation of Smalltalk-80.[13] Our implementation has performance comparable with ParcPlace Smalltalk-80 and, being written in C, is portable and easy to modify. We have therefore decided to follow on from the Distributed Smalltalk project by adding remote invocation to Smalltalk.

In the next two sections of this paper we outline our scheme for transparent remote invocation and discuss an architecture for distributed applications. We illustrate the use of this architecture in some simple applications that we have built. These examples will be used as illustrations later in the paper.

The remainder of the paper contains a discussion of how we addressed each of our requirements.

3. REMOTE INVOCATION OF THE METHODS IN OBJECTS

In Smalltalk, all entities are represented by objects that encapsulate a set of methods and private states. All computation proceeds by sending a message to some object which invokes a method which may in turn send further messages. The only direct access to an object's state is through manipulations by one of that object's methods. We can distinguish between mutable objects, which are objects that provide methods that alter their state, and immutable objects which do not. Once an immutable object has been initialised it will never change its state. Immutable objects include integers, characters and boolean values.

Transparent remote method invocation allows an object to send a message to any other object and to receive a reply without being aware of whether the receiver is local or remote. Transparency is achieved by automatically providing a local proxy for each remote object that can be invoked by a local object.[4] The function of a proxy is to behave like a local object towards the message sender, but instead of executing the message, it forwards it to the process (in another computer) where the remote object is located. The remote object performs the message and replies without being aware that its reply is sent back to a sender on a remote computer.

Every object in Smalltalk has a local identifier (or object reference)* that is valid within a single Smalltalk process. In our system any object that is to be referenced by an object in another computer is given a globally unique object identifier. This global identifier is generated at the object's local computer the first time its local identifier is to be sent to another computer in a reply to a message, in which case the global identifier is sent in place of the local identifier. The forwarding mechanism and the marshalling of messages is described in detail in Ref. 16, in which performance figures are given.

All arguments and return values in Smalltalk are local object identifiers – that is, parameters and results are effectively passed by reference. The logical extension in distributed Smalltalk is to pass global object identifiers whenever an argument or reply is passed to a remote process. For example, in Fig. 1 objects A, B and C each contain three slots – for the identifiers of objects containing their names and of other objects on the left and right. Object A is connected to objects B and C. Messages *name*, *left* and *right* cause the receiver to return identifiers of its name and the left and right objects respectively. Suppose also that X is an object in another computer and X sends the message *left* to A – the result is that a global identifier for B is returned to X and a proxy is created.

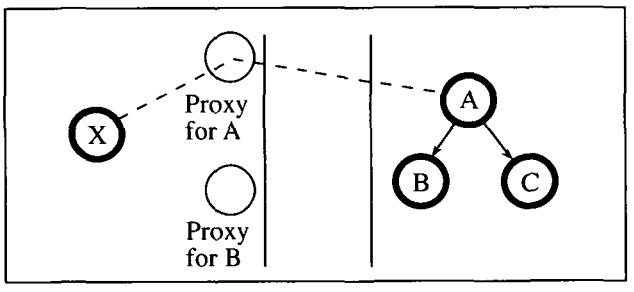

Figure 1. A proxy is created when the global identifier for B is returned.

Now consider the user of object X, who would like to see a view of the three objects showing the names and connections. When X sends the message *name*, A returns a global object identifier for the object containing the name and another proxy is created. To access the characters in the string, further global object identifiers and proxies are created. This mechanism has poor performance and causes an explosion in object identifiers and proxies. A pass-by-value mechanism can provide better performance. In general, we would like whenever possible to copy or move objects between machines rather than access them via proxies and relatively expensive remote message sends.

One of our sample applications is a simple departmental database containing information about lecturers, students and courses. When we first distributed the objects in this program, we decided that a process in one computer would contain the potentially shared database objects and that users could access them via user interface objects on their workstations. In Fig. 1, imagine that X is a user interface object and that the database is represented by objects A, B and C. In our first attempt we passed all arguments and results as global identifiers,

* This identifier is commonly called OOP – for object-oriented pointer.

apart from immutable objects which were passed by value. The performance was of course extraordinarily slow when strings were passed one character at a time to the user interface. However, we experienced another far more serious problem in that we discovered we had placed proxies in the database. To explain this, assume that the user creates a new object, Y, and supplies its name and then asks that it should be placed on the right of B (Fig. 2). If all arguments are passed as object identifiers (or global object identifiers in the remote case), the new object remains in the user interface instead of being passed to the database. It will be unavailable whenever the user decides to quit and may even be lost.

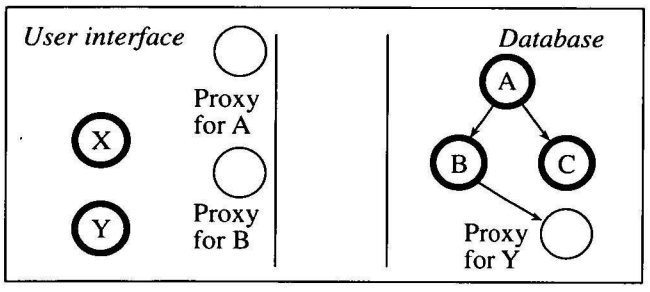

Figure 2. Proxies in database.

Thus we have identified a requirement to pass arguments and results by value in two cases:

- to place copies of shared information in the user's workstation for viewing and local manipulation;
- to move new objects to a place where they will be available for sharing and if necessary to be made persistent.

We have designed a scheme which exploits the Smalltalk memory manager's ability to detect the referents of an object. When a result of a remote invocation has been obtained by the remote process, the latter can discover whether that result is referenced by other parts of its system. If the result is not so referenced, because, for example, it has been newly created, it can be migrated to the invoking process, because as far as the rest of its system is concerned, the object does not exist. Arguments may be migrated in similar circumstances.

The two approaches to the treatment of parameters and results:

- copying the immutable objects freely between machines
- migrating objects that are unreferenced by the originating system

significantly reduce the number of proxies that are created and hence significantly reduce the number of remote-method invocations our system performs.

As an example, the list of courses shown in the user interface in Fig. 3 is obtained by the user interface process as the result of a message to the database process. The latter constructs the list as a sorted collection from the keys in a dictionary of courses in the shared database objects. This list has been newly created by sorting the keys and would be migrated. The strings in the list, being immutable, would be copied.

4. AN APPLICATION ARCHITECTURE

The facility for remote invocation allows us to construct applications in which objects may be placed at any computer in the network. We illustrate the requirements for an application architecture by reference to the departmental database example. We mentioned in the previous section that when we distributed the objects in our departmental database we considered two kinds of objects – (i) the shared objects that represent the information in the database, for example objects representing lecturers, students and courses; (ii) user interface objects.

The shared objects must be available before anyone can access the database. Ideally they would be in a process (e.g. a server process) that is always available. In applications that require long-lived stable information, a server that provides persistent storage will be chosen. Smalltalk provides a degree of persistency that may be sufficient for some requirements. We discuss our design for persistent objects in Section 7.

The shared objects form a connected set with roots from which all other objects may be accessed directly or indirectly. Before a set of shared objects can be accessed by remote objects, the root objects must be given global identifiers that will be made known to any objects that access them.

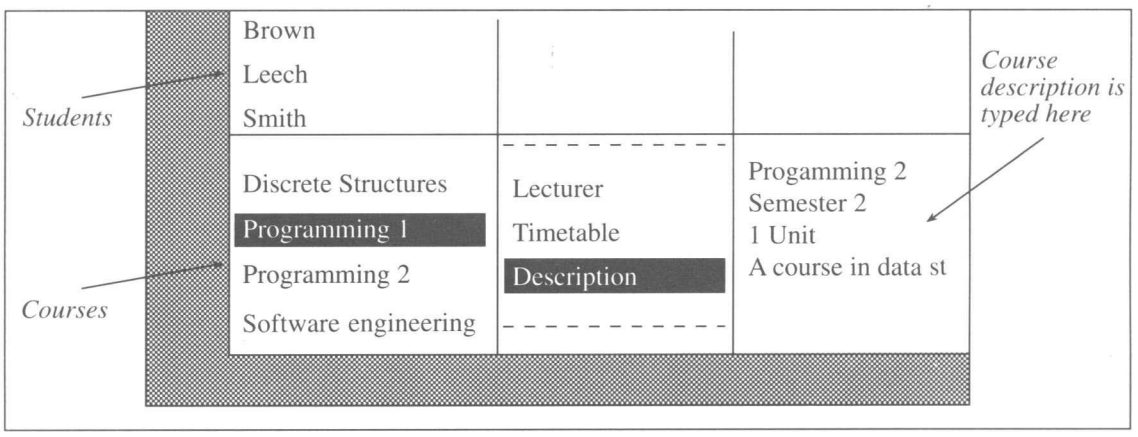

Figure 3. Part of the user interface of the departmental database.

A set of user interface objects participates in each user's interaction with the database. During this interaction, each user will at any instant be interacting with only a chosen subset of the objects in the database. For example, the user might be a lecturer who decides to supply a new description for a course in the database. In order to carry out this task, the user must select the course, indicate that they want to update its description and then type the description; finally they must add the description to the course entry in the database. During this task the user will have been presented with a view of a list of courses from which to select one. They will then select the description attribute of the course and will edit the description in a pane (see Fig. 3).

Here we have examples of user interface objects – the list of courses is an object created by accessing the names from the database and is held as a local replica. The original version of the course description is obtained from the database and held as a local replica while the user edits it. The list of the attributes of courses (lecturer, timetable, description) is an immutable object belonging to the user interface.

It is quite common to implement interactive user interfaces according to an architecture with three types of component: (i) a model – a collection of objects that are to be viewed and or changed; (ii) one or more views – objects that provide visual representations of the model; (iii) controllers – objects that deal with keyboard and mouse interactions. We are currently working within the limitations of this architecture, as discussed for example by Coutaz.[3] We return to these limitations at the end of this section.

As far as the distributed nature of the application is concerned, we assume that the user interface objects consist of models, together with associated views and controllers. The model can in general consist of any objects, but in our distributed architecture it will contain objects from two separate sources – (i) objects that are always in the user interface; (ii) local replicas of shared objects. The set of local replicas changes each time the user alters his or her point of interest. For example, if the user editing the course description decides to edit another course description, the replica of the first will be removed from the set. The set holds sufficient replicas of shared objects to represent a local model of the user's current point of interest, but no more. When the user changes a replica, the change is reported immediately to the corresponding remote object.

To summarise, the application architecture that we propose consists of two kinds of sets of objects as follows.

(i) The shared objects: these form a rooted connected set, in which the roots have global identifiers. The set of objects may be located in one or more computers, but without replication. All that need be done to make the set of shared objects available for sharing is to publicise the global identifiers of the roots.
(ii) The user interface objects – with one set in each active user's process. This includes replicas of shared objects comprising a model of that user's current point of interest.

The global identifier of the root of the shared objects must be known to the user interface objects, which communicate with it via a proxy. The remoteness of the shared objects from the user-interface objects is entirely transparent in all subsequent communication.

4.1 Room-booking program

Our second example is a room-booking program. It provides a user interface to a database of buildings, rooms and weekly booking sheets. The shared objects comprise a rooted tree of objects with the buildings at the top. Under each building there is a list of rooms and under each room is a list of weekly bookings. These objects could be arranged with different buildings in different servers, in which case each building would have an independent root with its own well-known global identifier.

The model in the user interface will include replicas of the names of all the buildings of interest, then for a selected building the names of all the rooms in that building. When the user selects a room and a week, a replica of the booking sheet for that week will be added to the model. When a particular time is selected a replica of the object containing the details of the booking will be added.

4.2 Chess program

The chess program was originally implemented as a user interface to the UNIX chess program. The communication between the two programs provided for turn-taking and reporting moves. There are two approaches to making this program available for use by a pair of competing players on separate workstations – the remote player and the shared board.

In the first approach, there is a single shared object representing a remote player and two user interface processes – one for each player. The shared object knows whose turn it is and the most recent move. A user interface process can send messages to the shared object: (i) at the end of a turn – to report a move made by their user; (ii) at the beginning of a turn – to ask about the move made by the opponent. On receipt of the first message the shared object informs the appropriate user interface about its user's turn. The two user interfaces contain the positions of the pieces on the board and the ability to check moves, as well as controllers and views, including objects containing bit maps representing the pieces.

The chess application can also be structured by sharing the board. In this model the shared objects include the board, the positions of each piece and the rules for legal moves, as well as knowing the player whose turn it is and the most recent move. The shared objects might also contain the bit maps representing the pieces – to be copied to the user interfaces when the game commences. The user interfaces are concerned with displaying the current state of the board and with allowing the local user to make a move when it is their turn.

4.3 Shared spreadsheet

The shared spreadsheet program is illustrated in Fig. 4. The shared objects consist of a two-dimensional array of cells, each of which has a name, a value and an optimal formula. The formula defines an expression that may

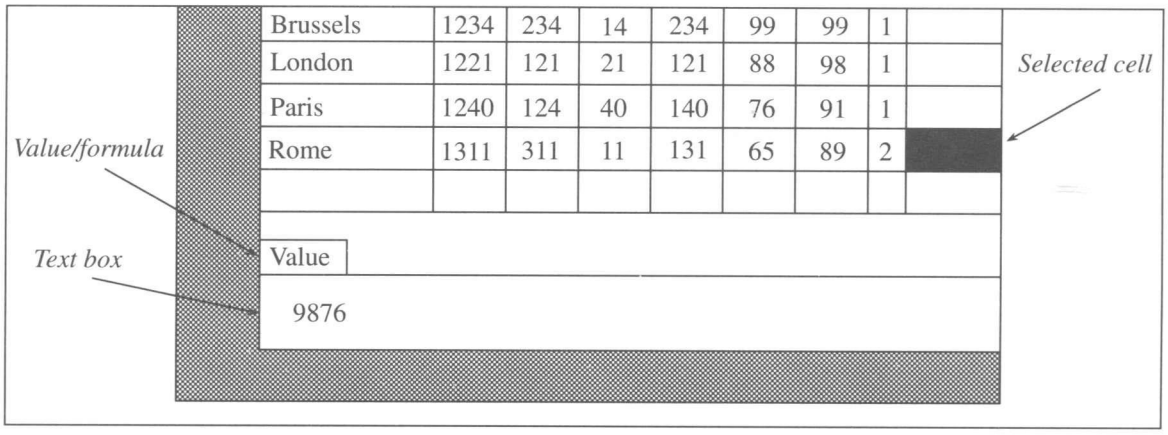

Figure 4. User of shared spreadsheet provides new value in text box.

involve the values in any of the other cells and is used to calculate the value of the cell. Thus the value of any cell with a formula depends on the values of the cells mentioned in its formula.

The user interface objects include a model containing replicas of the values of all the cells that are currently visible and of the currently selected formula (if any). The value or formula of the currently selected cell may be edited in the test box, and when the user is ready the new value will be assigned to the corresponding cell in the shared objects.

4.4 Keeping the views up to date

The model-view-controller architecture makes use of a dependency mechanism that allows any object to be made a dependant of another object. Any object can broadcast messages to all its dependants. In particular, sending a *changed* message to an object causes all its dependants to be sent an *update* message. The changed and update messages can be parameterised in order to localise the effects by saying more precisely what has changed.

In a user interface it is assumed that there can be several different views of a model, and all the views are made dependants of the model. Each time something changes in the model a notification of the change is broadcast to all the views. The views then send messages to the model to get the information they require to keep their display up to date.

In our distributed architecture, the model in the user interface objects contains replicas of some of the shared objects. Therefore if the user changes their point of attention, the set of replicas in the model must be refreshed from the shared objects. Similarly if the user alters something, the replica should be replaced by the new value proposed by the user and the new value passed on to the shared objects.

As an example of how this works when the model in a user interface contains replicas of the shared objects, we return to the departmental database shown in Fig. 3. Suppose now that the user selects *Timetable* instead of *Description* in the middle pane. The model component of the user-interface objects requests a replica of the information about the timetable from the shared objects and then notifies its views that this aspect has changed. The effect is that the view object displayed in the right pane requires updating and it gets the information from its local model.

As a second example, we return to the spreadsheet and consider what happens when a user changes the value of a cell. The model component of the user-interface objects will inform the shared objects about the new value of the cell involved. The change in this cell will cause further changes in the values of any cells whose formulas depend on its value. The user interface will need to refresh its set of replicas of the cell values from the shared objects.

The spreadsheet example illustrates a limitation of the model-view-controller architecture. The problem is that the shared objects have many inter-dependencies due to the way formulas are used to calculate the values of cells. The model in the user interface has no knowledge of these dependencies and therefore resorts to requesting new replicas of the values of all the cells currently of interest to the user. In most cases, only a few of the cell values will have changed. A similar problem will occur in any application in which there are non-hierarchic dependencies between the data objects.

5. SHARED VIEWS

In this section we discuss the issue of allowing several people to view the same information at the same time and to observe one another's actions on the information.

Suppose that several users are working together on a shared application and that one of the users makes a change affecting one of the objects. We described in the previous section the mechanism by which the user that makes a change may be shown a view of the effects of the change. We now discuss the issue of showing the effects to the other users involved.

In the normal course of events, the other users will not be aware of any change until they do something to change the set of replicas in their user-interface objects. For example, if a user changes their focus of attention, the set of replicas is altered and the affected views are redisplayed.

To illustrate the point about showing changes to other users, we return to our example where a user has just altered a value in a spreadsheet and suppose that this user belongs to a group of people who have agreed to work together on the spreadsheet. The other users in the group will not see the change unless they have done something such as scrolling the spreadsheet view that

causes the set of replicas in their user-interface model to be replenished.

We now describe our solution to the problem of passing the changes on to all the users currently involved in an application. This mechanism has been tried out with our departmental database application.

We have made additional use of the object-dependency mechanism to enable the broadcast of changes to all users who currently observe a set of shared objects. This is achieved by providing a shared model as a component of the shared objects. Initially the shared model has no dependants, but each time a user interface first accesses a set of shared objects it makes its local model a dependant of the shared model (by the normal procedure for becoming a dependant, but using remote invocation). Whenever the user changes something, their model sends a changed message to the shared model. The latter automatically broadcasts update messages to its dependants – the models in each of the currently active user interfaces. These 'broadcast' messages are sent transparently one by one to their remote receivers.

The receivers of the update messages from the shared model are the models of the currently active users. Each of these models requests new replicas of the shared objects and then informs its local views about the change. The latter behave in their normal way by requesting the latest information from the model.

When users are sharing a view they will all see one another's changes through this mechanism. In situations where users have not chosen to share a view, it is unlikely that two users will have identical views. In the departmental database example, all users see the lists of the names of lecturers, students and courses, but the likelihood of two users viewing the same course description at the same time is quite low.

To take this factor into account, the model in the user interface is selective about its subsequent actions when it receives an update message. The local model contains only the objects in the user's current focus of interest – if the update message refers to other objects it is ignored.

Hagsund's multi-user drawing editor that we mentioned earlier maintains the same view at all sites by means of the ordered multicast protocol provided by the Isis system.[2] Isis provides virtual synchrony – when order is relevant, events are observed in the order they were initiated. This semantics is not appropriate for the sort of applications we plan to support – in our case, events are observed if and when they are retrieved by the user's local user interface, because they are within the current user's area of interest.

A subsidiary requirement arises from the possibility of users observing the effects of other users' actions. This is that when users observe such effects they may want to know who produced them. Users of databases are essentially anonymous, but users of shared applications need not be. When other users are anonymous, they tend to be regarded as a form of interference (perhaps because they cause data to be locked), but when they can be recognized, co-operation is more likely to be possible.

6. PRIVACY AND PROTECTION – CAPABILITIES

In this section we discuss how we provide privacy and protection of objects. Privacy allows users to restrict who is allowed to view the information and protection prevents unauthorised alteration of information.

We provide both protection and privacy by means of capabilities. A capability is in effect a message filter; it forwards the permitted messages to the object it protects and rejects all other messages. The capability mechanism is supplied by a class that provides operations for creating capabilities and for restricting them by removing the rights to use particular operations. When a user requests access to a shared object that is protected by a capability, a new capability with the appropriate message filter is constructed in the same process as the shared object and its global identifier is returned. The capability mechanism is described in detail in Ref. 5.

As an example we discuss the use of capabilities in the departmental database. The shared objects are protected by four alternative capabilities allowing gradually more restrictive access via the root. For example, the first allows full access to all operations and the last allows limited viewing of some of the information. These capabilities are intended for use by administrators, lecturers, students and others respectively. We have found it necessary to supply each user with an additional personal capability allowing access to their own records.

This particular application contains lists of administrators, lecturers and students and can identify the user category. Most applications would require an additional naming service for this purpose.

The capability mechanism can be used only if the process containing the shared objects is absolutely certain of the identity of the user requesting capabilities. This form of authentication can be provided by means of an authentication system such as Kerberos,[17] which provides the user with a ticket to be used when requesting capabilities from a particular application process. Once the capabilities have been obtained the ticket is no longer needed.

The use of access control affects the operations available to a particular user. If the user interface appears to provide all operations irrespective of the rights of a particular user, the latter may attempt operations that are not permitted. The result of attempting an operation that is not permitted is that it is filtered out by the capability and the user's effort has been wasted.

A similar remark can be made about objects protected by some form of concurrency control. The permitted operations for a particular user are restricted during the time that the object is protected because another user is currently updating it.

To be effective in the presence of access control, a user interface must be configured according to the rights possessed by a particular user. And in the presence of concurrency control it must change its configuration according to the current protection of the object and who is currently allowed to update it.

7. LONG-TERM STORAGE – PERSISTENT OBJECTS

In this section we discuss how we provide for long-term storage of shared objects. We expect that large numbers of objects will be generated and that users will expect to be able to have future access to such objects from time to time. This suggests that some objects should be made

persistent. A persistent object is one that is saved in its most recent consistent state and can be restored whenever it is needed. When they are no longer needed, persistent objects are garbage-collected.

Smalltalk itself provides a limited form of persistence in that a snapshot of the current state of all objects in the Smalltalk system may be saved on user request. The system can be restored from such a snapshot.

The Smalltalk mechanism does not provide controlled persistency for selected objects whenever they are in a consistent state. We have designed and implemented a prototype of an environment that we call a **persistent object store**. The idea was suggested originally by Low[12] and is described in detail in Ref. 6. Our persistent object store has persistent roots, and objects are made persistent by attaching them directly or indirectly to one of the persistent roots. In the persistent environment all objects respond to a checkpoint operation that causes their current state to be recorded in a log file. Persistent objects are retrieved from the log on demand whenever they need to respond to invocations.

The persistent object environment is essentially an extension of Smalltalk that allows selected objects to be persistent. Such an environment is suitable for the shared objects in our example applications.

8. CONCLUDING REMARKS

Our current prototype distributed Smalltalk includes the facility we have described for remote method invocation by means of proxies and global object identifiers. Redesign of the language is making improvements to the efficiency of our system by reducing the number of proxies required to communicate between two processes. Both capabilities and the shared model are available by means of classes that can be used to enhance the functionality of the shared objects. To provide a capability layer in a particular application, the method for generating capabilities must be extended (for example by subclassing) to generate capabilities with the appropriate message filters.

Each of the example applications was implemented as a single Smalltalk program and then divided into the two sets of objects – the shared objects and the user-interface objects. Once the objects were divided into the two sets, all we had to do was to get a global object identifier for the root of the shared objects and to make it known to the user-interface objects. This procedure has proved to be effective and seems a productive way of making distributed applications. Its use is, however, limited to self-contained applications.

Some of our current work is concerned with the design of an extension to our remote invocation scheme to make use of interface definitions, with a view to allowing servers to be developed separately or implemented in another language (C^{++}). Interface definitions will be used to generate tailored proxies for use in Smalltalk client environments.

We have not yet addressed the issue of concurrency control, but we expect to be able to make use of a shared token to protect any set of shared objects that is being accessed concurrently by several users.

An aspect of the work that seems particularly interesting at present is the development of the shared model idea. We have designed the basic mechanism in which the shared model records its dependent local models and is able to inform the latter when shared objects have been updated. The granularity of this mechanism can be controlled to the extent that the argument of the messages about changes can indicate what aspect of the shared model has changed. In some applications the network of shared objects may become quite large and we will require a more subtle means of control of the information about changes.

Lastly, we still have to explore the design of a user interface in the presence of capabilities and concurrency control. As different users are given different capabilities, they should be given user-interface controllers to correspond to their capabilities. For example, their menus should contain only the operations they are permitted to perform. In the case of concurrency control, the operations available to a user vary from time to time. For example, the rule might be that only one user at a time can update a particular value. The user that can make the changes should have a different controller than the ones who are forbidden.

Acknowledgements

We would like to thank George Coulouris for his comments on this paper, which have led to improvements in the design. We would also like to thank Douglas Steel for designing and implementing our initial remote invocation scheme and Claudio Nascimento and Sunggi Min for their contributions.

REFERENCES

1. J. K. Bennett, Experience with Distributed Smalltalk. *Software – Practice and Experience* **20** (2), pp. 157–180 (1990).
2. K. Birman and T. Joseph, Exploiting virtual synchrony in distributed systems. *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pp. 123–138 (1987).
3. J. Coutaz, The construction of User Interface and the Object-Oriented Paradigm. *The European Conference on Object-Oriented Programming, Paris*, pp. 135–144 (1987).
4. D. Decouchant, Design of a distributed object manager for the Smalltalk-80 system. *Proceedings*, OOPSLA-86.
5. J. Dollimore and Wang Xu, *Using capabilities to ensure correct access to shared objects in a distributed object-oriented system*. QMW CS Report No. 537.
6. J. Dollimore, Sunggi Min, E. Miranda, C. Nascimento, D. Steel and Wang Xu, *Communicating Distributed Persistent Objects in Smalltalk*. QMW CS Report No. 504.
7. I. Greif and S. Sarin, Data sharing in group work. *Proceedings of the Conference on Computer-Supported Cooperative Work, Austin, Texas*, pp. 175–183 (1986).
8. A. Herbert, The Computational Projection of ANSA. In *Distributed Systems*, ed. Mullender, S. Addison-Wesley New York, pp. 401–414 (1989).
9. O. Hagsund, *A Multi User Draw Editor*, presented at the 1st MultiG Workshop, Stockholm (1990).
10. E. Jul, H. Levy, N. Hutchinson and A. Black, Fine-grained mobility in the Emerald System. *ACM Transactions on Computer Systems* **6** (1), pp. 109–133 (1988).

11. B. Liskov, Distributed programming in Argus. *Comm. ACM* **31** (3), pp. 300–312 (1988).
12. C. Low, A shared, persistent object store. *Proceedings of the Second European Conference on Object-Oriented Programming, Oslo*, pp. 390–408 (1988).
13. E. Miranda, BrouHaHa – a portable Smalltalk Interpreter. *Proceedings OOPSLA '87.*
14. J. R. Olson, G. M. Olson, L. A. Mack and P. Wellner, Concurrent editing: the group's interface. *Proceedings of a Conference on Human–Computer Interaction, INTERACT '90, Cambridge, England*, pp. 835–840 (1990).
15. S. K. Shrivastava, G. Dixon, G. D. Parrington, F. Hedayati, S. Wheater and M. Little, The design and implementation of Arjuna. *Proceedings of the Third Conference on Object-Oriented Programming, Nottingham* (1989).
16. D. Steel, Distributed object-oriented programming: mechanism and experience. *Proceedings, Tools USA 91, Santa Barbara.*
17. J. G. Steiner, C. Neuman and J. I. Scheller, *Kerberos: An Authentication Service for Open Network Systems.* Project Athena Report, MIT (1988).
18. D. Thomas and K. Johnson, Group Object-Oriented Programming: Orwell – A Configuration Management System for Team Programming, Carleton University, Canada. *Proceedings, OOPSLA '88.*

Book Review

VALERIE ILLINGWORTH (Editor), *Dictionary of Computing* (third edition), Oxford University Press. £6.99. ISBN 0-19-286131-X.

Not even a reviewer could be expected to read a dictionary straight through, so a review has to be derived from browsing. In my browsing I concentrated on two areas: (i) terms that are commonly misused; (ii) terms that I feel unsure of and would like to understand better. While looking up such items my eye, inevitably, kept finding other entries of interest, so I would look at them on the way (and sometimes forget what I was originally searching for). The result of my browsing is remarkably favourable and I congratulate the general editor, two consultant editors, 45 contributors and the publishers on a splendid product at a reasonable price.

So far I have found only two errors that a proof-reader should have spotted – a remarkable degree of accuracy. These are (i) a missing decimal point in the example of BNF production rules; (ii) 11 words missing from the definition of 'F distribution'.

Many definitions are timeless, but some entries are necessarily out of date when such a publication appears. I doubt, for instance, whether it is still reasonable to say that ICL is 'a wholly British company'.

A few other points are worth mentioning in hoping for an even better next edition. (Incidentally, why do the publishers put the ridiculous words 'New edition' on the front? Of course it is new when it is new, and of course it will soon cease to be so.) The definitions of some of the searching techniques would not have told me to use a binary search when wanting to know where a function crosses zero, but a golden section search (or a *Fibonacci* search) when looking for a maximum or minimum. The impression is given that these are merely alternatives. The definition of BNF correctly uses angle-brackets, not to be confused with less-than and greater-than signs, in its example, but unfortunately less-than and greater-than are used where BNF is employed in some other definitions. In spite of being a British publication, there are some Americanisms, the oddest being where 'colored' is used in the definition of 'coloured book', and I strongly dislike 'named for' instead of 'named after'. The word 'contemporary' is misused in the entry for 'IBM system 360', where 'contemporary IBM mainframes' is presumably meant to mean 'present-day' rather than contemporary with the 360.

I was surprised at finding no entry for 'dangling else', and even 'else' on its own is treated as arising only in connection with decision tables. To find a description of the problem you have to look under 'ambiguous grammar'.

I was disappointed in the definition of *K*. Perhaps I am out of date, but I was certainly taught 30 years ago that *K* had been introduced to computing as a quick simple way of indicating 1024. Except mnemonically it had nothing to do with *k* meaning kilo- which, of course, remained as 1000. Unfortunately *K* has been stolen from us, and misused to mean 1000, by administrators and accountants, but I regret finding a dictionary of computing that merely says

k (or **K**) *Symbols for* kilo-

and then defines kilo- as indicating 1024 when the binary system is used. Their advice to avoid the capital *K* is not followed elsewhere by themselves.

I have enjoyed comparing this dictionary with the glossary at the back of Lord Bowden's classic *Faster than Thought* (1953). That glossary contained only 55 terms, compared with over 4000 in this dictionary, but 40% of them have survived as basically the same term with the same meaning. One term, microprogramming, appears in both lists but with disjoint meanings. I am sorry that 'Hartree constant' has gone ('The time which is expected to elapse before a particular electronic computing machine is finished and working'); perhaps it does not now apply so strongly to hardware, but it is surely still a relevant concept in terms of when computers will all understand natural language and accept voice input. This has been 'about 5 years' as long as I can remember. One rather sad note comes from Lord Bowden's borrowing from Dr Johnson's definition of 'lexicographer' in defining 'programmer' as 'a harmless drudge' – the present dictionary has to have an entry for 'hacker' that was absent then.

My complaints are few indeed. I strongly recommend this excellent dictionary. There are, after all, not many dictionaries where one could find such intriguing entries as:

mother *Another name for* parent, rarely used.
worm *See* virus.

I. D. HILL