Structure and Encapsulation in Distributed Systems: the Proxy Principle

Marc Shapiro
INRIA, * B.P. 105, 78153 Le Chesnay Cedex – France, tel.: +33 (1) 39-63-55-11

Exact transcription of article from *Proc. 6th Int. Conf. on Distributed Computing Systems* (ICDCS), Cambridge MA (USA), May 1986, pages 198–204.

Abstract

We present a novel view of the structuring of distributed systems, and a few examples of its utilization in an object-oriented context.

In a distributed system, the structure of a service or subsystem may be complex, being implemented as a set of communicating server objects; however, this complexity of structure should not be apparent to the client. In our proposal, a client must first acquire a local object, called a *proxy,* in order to use such a service. The proxy represents the whole set of servers. The client directs all its communication to the proxy. The proxy, and all the objects it represents, collectively form one distributed object, which is not decomposable by the client. Any higher-level communication protocols are internal to this distributed object.

Such a view provides a powerful structuring framework for distributed systems; it can be implemented cheaply without sacrificing much flexibility. It subsumes may previous proposals, but encourages better information-hiding and encapsulation.

1 INTRODUCTION

Structuring programs into their logical components, and encapsulation (hiding a structure into a single module with a clear, restricted communication interface), are universally considered good programming practice. We consider its application to distributed systems.

First, let us consider the situation in a conventional OS like Unix [Ritchie74]. A simple program is a process; within it, structuring and encapsulation may be achieved by using suitable programming techniques and/or languages. A complex program may be written as a collection of communicating processes. The use of distinct processes, in addition to allowing parallel processing, gives structure to a program because each process is totally independent of the other. However, the communication interface between them is clumsy and ambiguous; the semantics of the data in a pipe or a socket is purely a matter of convention between processes, implying that its knowledge is spread throughout the system. This is acceptable in Unix because of the typical life cycle of a program: if some part of it needs to be changed, the program stopped, re-compiled, and run again; if the kernel itself needs to be changed, the machine must be halted and re-booted.

In a modern distributed system, the situation should be quite different. Such a system is a collection of distributed services, and runs continuously, even if one of

*This research was supported in part by the ESPRIT program of the European Community, under contract no. 367, "Secure Open Multimedia Integrated Workstation" (SOMIW).

its machines is halted. New services are continuously being erected, and running ones modified. Communication between remote objects may involve a complex protocol. Connections between objects are subject to continuous change. The conventional approach is no longer satisfactory: for instance, a service might be upgraded; this should not force its clients across the network to be re-compiled or re-linked. There is a need for run-time support for encapsulation, for the definition of clear communication interfaces. The system should help check agreement on the interface and communication protocols, even when these may change over time.

Some proposed architectures for distributed systems are extremely flexible but, being low-level designs, provide inadequate structuring tools. Other architectures instead are very high-level, but appear inflexible and expensive to use. It is desirable to strike a middle ground: to provide structure without sacrificing too much flexibility, at a low implementation cost.

The outline of this paper is as follows. First, we present the object-oriented model of computation, and a previously proposed incremental design method, and introduce our proxy concept. The following section gives three significant examples of the utility of the proxy principle. An other section compares our approach to related work, and discusses the various properties of proxies. Finally, we conclude by an assessment of our current proposal, with respect to our goals, and by indicating future directions of research.

2 OUR PROPOSAL

2.1 The Object-Oriented Approach

Our model of computation is based on the object-oriented approach [Jones79]. A program is a collection of intelligent entities, or objects. An object contains both data (its internal representation) and code. The representation is accessible only by calling the procedures defined in the code, and is not accessible outside of the object. This allows to ensure that some representation invariant holds between calls. The interface of an object is well-defined, being restricted to its procedures.[1]

The object model is well adapted to distributed systems, as it allows to abstract processors, processes, services, servers, resources, etc., into one concept. One popular implementation of distributed system, is one where each object is a server process (or a closed group of processes), communicating with other servers by exchanging messages; every message carries an **opcode** field, containing a conventional constant which identifies a request type [Zimmermann84]. The meaning of the constant is not manifest; its knowledge is spread across all the servers. We advocate, instead, an implementation where objects are passive and communicate through a procedural interface, allowing the use of standard program production tools and the checking of procedure names and types.

2.2 An Incremental Design Method

Our structuring concepts derive from the design method described in [Hoare79]. Briefly, the first step in a design consists of the specification of a set of naive communicating objects (in this case, CSP processes [Hoare78]). At each subsequent

[1] Note that in the literature on object-oriented programming, invoking a procedure of an object is often called "sending a message to the object" [Goldberg83]. This does not imply any asynchronous processing; it simply means that a particular procedure name, when applied to different objects, may have a different meaning, as decoded by the object's manager itself. For instance, the code **a+b** (also noted **a.plus(b)**), may be read "send message + to **a** with argument **b**." If **a** is an integer, this will be interpreted (by the code of **a**) to mean "add **a** to **b**"; or, if **a** is a set, as "compute the union of **a** and **b**."

refinement step, one such object gets hidden behind an "enhancer" object, transforming into something less naive (and so on recursively). The code of the old object is preserved; the enhancer is supposed to add functionality (e.g. buffering) while keeping the original interface. For instance, in Figure 1a, a computer system consists of a single **job** writing on a line-printer **lpr.** The line-printer interface consists of two procedures: **write_line()** and **top_of_page().** The job blocks during output. In Figure 1b, the job's calls are re-directed to a **spool** object, which buffers lines so that **job** may proceed more quickly. It is important that this redirection is done transparently; the **job** still makes the same calls on something mimicking **lpr;** and **lpr** receives commands as if directly from **job.**

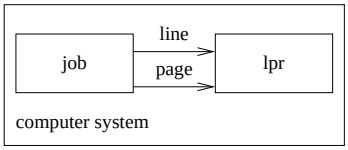

Fig. 1a

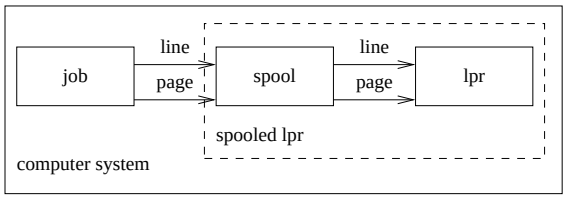

Fig. 1b

Figure 1: Enhancing a naïve specification

Spool is indistinguishable from **lpr** for the **job;** we will call it the *proxy* representing the **lpr** service from the viewpoint of **job.** In [Shapiro82], we show that this methodology for designing structured, distributed programs, although powerful, is awkward to use in the CSP framework of [Hoare79]. The specification becomes especially unmanageable when protocol layers are involved.

What we have just called re-direction, may be dually viewed as *substitution.* Going from Figure 1a to 1b, we have substituted **lpr** with the object composed of **spool** and **lpr.** In an object-oriented system like Smalltalk-80 [Goldberg83], substitution is performed by the **become** primitive, by which an object asks to be replaced by some other object, while keeping the same identity as the old one. **Become** is more flexible and more generally useful than the CSP substitution rules by name of [Hoare79]; we will see an example of its usefulness hereafter (Section 3.3).

2.3 Using Proxies for Structure and Encapsulation

We will now generalize the above model into a structuring principle for distributed systems. Our proposal aims at reconciling flexibility with the capability of encapsulating a structured object behind a black-box boundary. Clients wishing to use the services of this group are restricted to the visibility of the black box; any complex communication protocols will be hidden behide this boundary.

Our "proxy principle" is the following: *In order to use some service, a potential client must first acquire a proxy for this service; the proxy is the only visible interface to the service.*

We will call the object(s) represented by a proxy its *principal(s).* The proxy and its principal(s) collectively form a single distributed object, which we will call a group. (Please refer to Figure 2.)

The underlying system must implement the following properties. A proxy is always local to its client.[2] All interaction with the service goes through the proxy. The proxy is, from the client's point of view, indistinguishable from the group. A proxy has the property of being visible from its caller, while its internals are not; it, in turn, has full visibility of the group. All communication other than a local

[2]If the service is local too, then the proxy might actually be the same object as its principal.

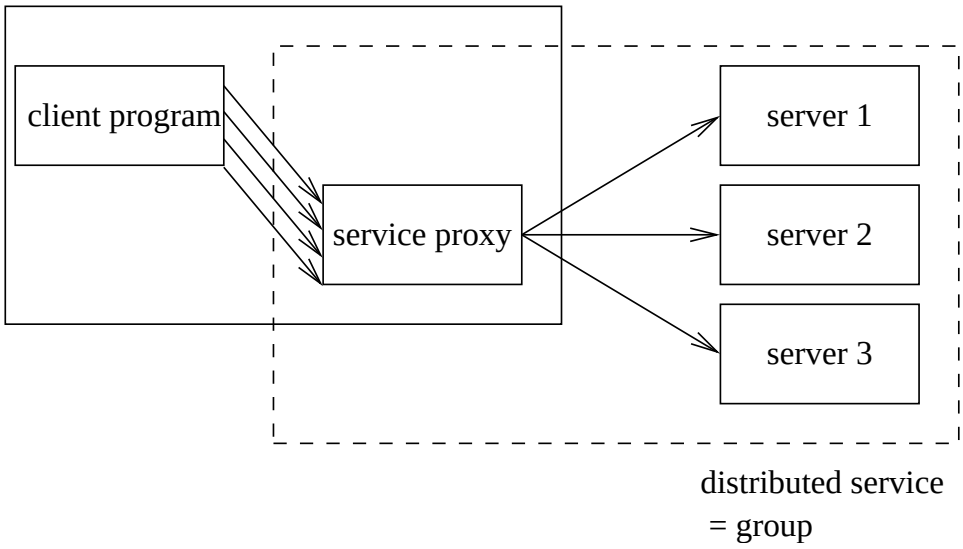

Figure 2: A proxy for a distributed service

procedure call, viz. network connections or shared data, is considered unsafe, and is allowed only within a single group. (However, one object may be part of more than one group.) This ensures that any communication with a service is filtered by a valid representative of that service.

This turns out to be a very powerful structure. We will illustrate some of its properties by examples in the following chapter, then discuss them more at length in the next one.

3 EXAMPLES

We will now present a few examples, to illustrate the use of proxies. These are: Unix-style conventional file access and pipes, and a distributed game. Each example involves the use of application-specific objects, as well as system-defined ones (name service, storage service, communication).

We postulate the existence of an object-oriented kernel, supporting object creation and deletion, procedure invocation, and controlled migration of objects between sites. The kernel shall also enforce the properties of proxies, listed in the previous section, namely that a proxy is always local, that unsafe communication may take place only within a group, and that a group is indistinguishible from the proxy which represents it. These properties may be implemented cheaply, e.g. by restricting the visibility of each object to its group(s). Note that, for the kernel, there need not be any difference in nature between a "principal" and a "proxy" (this is just a useful terminology).

In the following examples, we use a notation similar to the C language. Invocation of operation **y** of object **x** with parameter **z** will be noted **x.y(z).**

3.1 File Access

Our first example will show the structure of conventional (i.e. Unix-style) file access. Although it may be argued that files, viewed as uninterpreted strings of bits, are rendered obsolete by storage of objects, we will use this simple and familiar example. We make no attempt at optimal algorithms, as this is solely an example.

A file is an object, which normally supports operations **read, write, dup** and **close,** with the same semantics as the similarly-named Unix primitives. Some files may have additional operations, such as **seek** for disk and tape files, **ring_bell** for interactive terminals, or **connect** for sockets.[3]

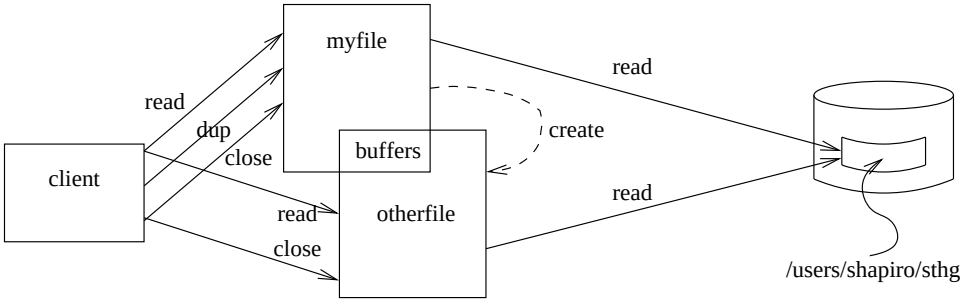

Figure 3: Disk access

Please refer to Figure 3. Suppose we want to access the data named "/users/shapiro/sthg". One must locate the hardware location corresponding to this name, and start the correct driver for that location. These two steps are accomplished by the following sequence:

```
file myfile;
myfile = nameservice.open("/users/shapiro/sthg", "r");
```

This causes the **nameservice** object to invoke its **open** operation, with the two string parameters indicated. The file location is sought and found. What is returned is an object in charge of that location, with sufficient knowledge to access and use it; *this object is migrated and installed in the calling context* under the name **myfile.** This object has the data needed to access the file (location, seek pointer, buffers). If the file is on a local disk, the location information will include inode or block numbers; **myfile** is actually a driver for the local disk. If the file is remote, the information includes a network address for the remote disk driver; **myfile** is a network driver. If the name "/users/shapiro/sthg" happens to correspond to a terminal, then the imported **myfile** is a terminal driver.

Suppose now that the file is remote. Execution of

```
int i;
char str[100];
i = myfile.read(&str, sizeof(str));
```

will read up to 100 bytes of file data into the string **str.** One way of doing this might be to copy buffers from **myfile** into **str.** (Thus one use of proxies is better locality; some requests can be answered locally without an expensive access to the remote resource.) If these buffers are empty, a request for data will be sent over the network. (An other use of proxies is as a "stub", to copy data in and out of network packets [Birrell84].)

Any attempt to execute the instruction

```
i = myfile.write(&str, sizeof(str));
```

will fail; since **myfile** was opened as a read-only file, it lacks a **write** entry point. (Thus as stub can act as a "capability" protecting a resource. Here the file data is totally protected against writing by the lack of a **write** procedure.)

The following sequence:

[3]Note that the operation **open(pathname)** is not an operation on files; it is an operation on the name service which returns the object stored under the specified pathname. This may or may not happen to be a file of one type or an other.

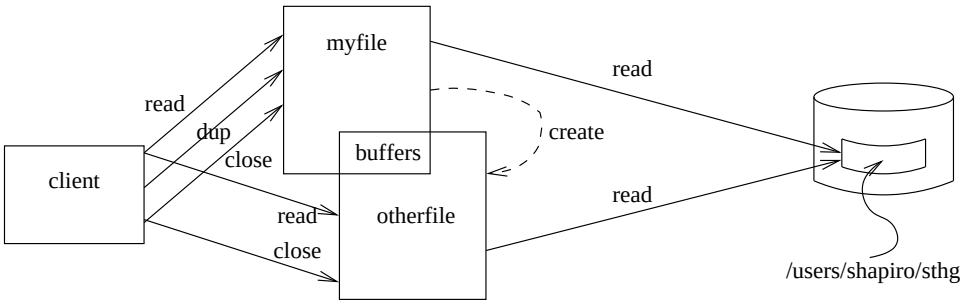

Figure 4: The **dup** operation

file otherfile;
otherfile = myfile.dup();

creates a new object which is a duplicate entry to **myfile,** sharing the same buffers, seek pointer, and location information (see Figure 4).

Whenever an object, such as **myfile,** is released, either voluntarily by calling **myfile.close()**, or for external causes, like the destruction of its owner, a special procedure **omega** will be called by the system, to allow the object to clean up first. Similarly, a special procedure **alpha** is invoked before any other entry point, to initialize.

In the previous example, we have used the object **nameservice.** This is a proxy for the distributed name service. Its job is to map a name into an actual object. This involves searching the local directory; **nameservice** is the same as the local name server object. If the name is not found locally, then it must also query the remote name servers. The local name server encapsulates the distributed name service group.

Since all access to a previously unknown object involves using the name service, **nameservice** must be pre-installed by the booting procedure, and visible to all clients.

3.2 Pipes

A pipe is similar to its Unix counterpart. It is created by calling **pipeservice.pipe(),** which returns two objects, one for each end of the pipe. Both ends implement **dup, close, alpha,** and **omega.** Only one end implements **read,** and the other **write;** they share the same buffer space (Figure 5a). The **omega** procedure of the **write** side sets an end marker on the pipe data; on the **read** side, **omega** sets a flag in the shared space, signalling the "broken pipe" condition to the **write** side.

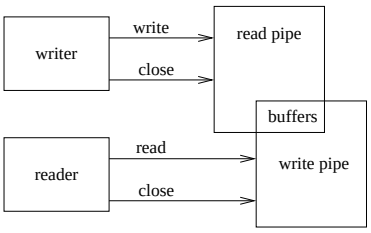

Fig. 5a. A pipe

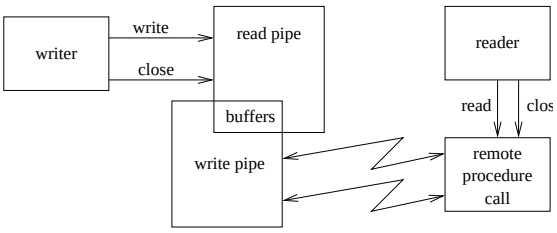

Fig. 5b. A pipe across the network

An interesting issue is how to implement a pipe between remote machines. There must be a proxy for the pipe on one of its ends; let us say on the reading side. The

simplest construction is shown in Figure 5b. The read proxy reproduces the same procedural interface **(read, dup, close, alpha, omega)** as its principal, and simply relays every call to the other site. Setting up the network pipe involves the following steps: (1) create a local pipe with **pipeserver.pipe()** which returns two file-like objects, **readpipe** and **writepipe;** (2) give away a handle to the read end by executing **readpipe.export(X);** this in turn will look up an object to do the remote procedure calls, and instantiate it on the site of X.

A proxy which simply reproduces its principal's interface, a "remote procedure call object" (similar to Birrell and Nelson's stubs [Birrell84]) is generally useful; it will be one of a library of generic communication objects to be supplied with such a system. A remote procedure call proxy is parameterized by the names and types of the procedures it represents; these must be checked either when the proxy is bound to the client or when it is called. When such an object is called, it must send and re-create the argument stack on the called machine; invoke the correct procedure; and return the results and modified stack to the caller. Appropriate time-outs must be set, in case of lost packets, which must be retransmitted [Birrell84].

3.3 Distributed Game

In this section, we will present the structure of a hypothetical distributed game, along the line of Amaze [Berglund84]. This example uses a different proxy structure than the others, and shows the usefulness of the **become** primitive.

The game is a starship pursuit; each player has her own starship. A player may join or leave a game at any time. Once the game is joined, her commands are restricted to **launch**ing only one starship or quitting; and once launched, to changing its direction or speed **(faster, slower, left, right),** firing shells straight ahead **(fire),** and leaving the game **(close).** A shell, once fired, continues in a straight line until it encounters a starship or another shell. The same is true for a starship, unless its owner changes its direction or speed. When a starship encounters a shell (or another starship), they are both destroyed, but the starship owner is given a chance to launch a new ship from her base.

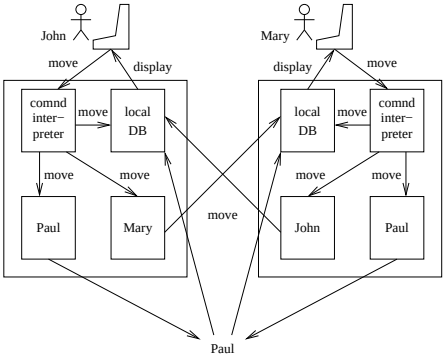

Fig. 6a Distributed Game with Proxies

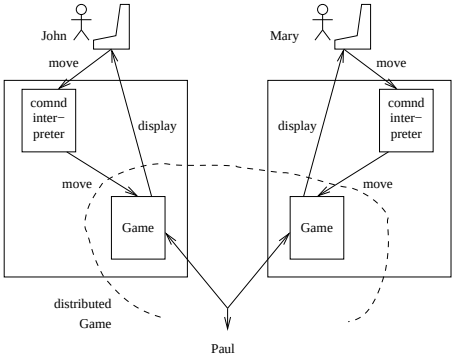

Fig. 6a Distributed Game Object

Figure 6 illustrates the structure of the game with three players, Paul, John and Mary. Each player program maintains a database of the locations, directions and velocities of all pieces (spaceships and shells); the contents of the database are displayed graphically on the screen. At every clock tick, the position of every piece is re-computed from its previous position and momentum. As long as no player takes a move, no network traffic is generated; each site computes the new position of all the pieces using only local information. When a player does move, the new

absolute coordinates and velocity of its piece is sent to all the other players, whose database is updated accordingly.

One possible structure is proposed in Fig. 6a; this is a straightforward extension of the previous examples. Each player program carries a proxy for every other player. This structure is clumsy. Moves are sent to (the proxy of) each opponent in turn, precluding broadcasting. The player can communicate only its move, i.e. acceleration, to the others, and not absolute co-ordinates and momentum. In these conditions, the different databases might rapidly become inconsistent with each other, due to variable network delays and lost messages.

A better solution is proposed in Fig. 6b. Here it is recognized that the **Game** is in fact one distributed object, represented locally by the database. All moves (both of the owner, John, and of his opponents, Mary and Paul) are sent to **Game.** It is now possible to broadcast interesting information to all the players. Exchanging absolute information is possible. Since all network communication takes place within the Game group, an efficient ad-hoc communication protocol may be used.

It is interesting to consider starting a new game, and joining and leaving an ongoing game. To start a game, the first player, John, instantiates a **Game** object (from the program library). As part of initialization, this will register a proxy for it with the name service. Mary, who wishes to play too, will look it up, and instantiate it as her own proxy for the **Game.** The "joining" proxy has a very restricted interface, just enough to get the information to decide to accept her or not (possibly asking John for advice). When Mary is accepted, John's site enters her into the database, and makes a copy of the database, which is sent to Mary's machine. The joining proxy installs the database copy, and does a **become;** it is replaced and the variable **Game** now denotes (for Mary) the up-to-date database with its full interface for doing all the moves. In order to leave the game, Mary simply **close**s her **Game.** The **omega** procedure will broadcast this information to the other sites, before closing up shop. Care must be taken to remove the entry for the joining proxy in the name service when the last player leaves the game, and the game group is terminated. This will not work if the group is terminated by a failure.

This example illustrates a few more uses of proxies. Access control ("capability") is performed by the "joining" proxy. No access is allowed until the "joining" proxy has verified Mary's rights (possibly by asking John); when this is done, the "joining" proxy replaces itself with a full-fledged **Game.** At his point, no more access control is necessary.

Since network messages can only come from partners in the group, it is possible to use an ad-hoc, efficient encoding of messages (it is not necessary to use a general-purpose data encoding). It is not necessary either to check the rights of the sender, since they have already been tested by the sending proxy. The communication protocol is entirely encapsulated within the distributed **Game** group.

Finally, there is the question of protocol agreement. The approach taken in this example is that it is not necessary to check that the partners in a distributed communication protocol agree on a common protocol: since all partners in the game are proxies of the same generation (all created by copying the same initial object) there is no need to check protocol. However, this approach is insufficient to cope with failures.

4 DISCUSSION AND COMPARISON WITH RELATED WORK

4.1 Properties of Proxies

The previous examples illustrated a few uses and properties of the proxy. Its most striking property, which makes it attractive in building a distributed system, is that it is an unforgeable *capability* [Fabry74] protecting the resource it represents. This follows from the properties of the underlying object-support kernel: proxies are unforgeable because an object or a group is permitted to create only proxies for itself; they protect the group because all communication other than local procedure calls are filtered by a valid proxy. Moreover, a proxy is totally programmable, making it more flexible than the traditional capability.

Let us list some more interesting properties of proxies, most of which are illustrated by the previous examples:

1. *Encapsulation.* The service is a black box, accessible only through the proxy. Its structure is not exposed.

2. *Locality.* A form of network transparency is achieved, in that all accesses are local. Some requests might be answered locally by the proxy. Buffers may be kept locally. The current state of the connection may be recorded in the client machine, allowing services to be memory-less.

3. *Access Protocol.* The proxy enforces per-client ordering constraints on calls (e.g. enforce a request - acknowledgment - access - release ordering on the use of a resource).

4. *Capability.* The proxy can implement access controls, test the validity of arguments, or the right to perform certain operations; it is totally programmable.

5. *Stub.* The proxy is a stub [Nelson81]; it does marshalling (packetizing data) and performs the network access.

6. *Trusted Communication.* A proxy is created by the service itself. Therefore all communication received by the service comes from a trusted partner; the complexity of the server may be reduced.

7. *Protocol Encapsulation.* The protocol between the client and the service is totally encapsulated within the distributed object formed by the proxy and its principal(s). Therefore, agreement on a protocol is not an issue (disregarding faults and malicious programming).

Looking at the list above, one might criticize the proxy idea as trying to be too many things for too many purposes. Indeed, a single proxy which would attempt to do all of the above at once should itself be structured. The above functions may be broadly split into two categories: local functions (concerned with access control, ordering, buffering; points 2–4); and communication (marshalling, protocol agreement; points 5–7). The former functions are application-dependent, and must be programmed specially when needed. The latter, which are often generic, are performed by "communication objects" supplied with the system.

The "remote procedure call object" of the pipe example is one instance of a system-supplied communication proxy. The communication protocol (checking parameter names and types, encoding the request, performing the remote call, setting and responding to time-outs) is entirely encoded within the communication object and not in the clients and services; thus it is not necessary to check for protocol

agreement. Other authors propose the separation of communication from the partners in the communication, e.g. the activity messages of [Banino85] or circulating tasks of [Betourne85], where the knowledge of a protocol is centralized in an active object, circulating between the partners; or the Scripts of [Francez83], i.e. the abstraction of a distributed communication described as a series of roles, to which actual processes enroll. We plan to develop research along these lines, interpreting a protocol as an object.

4.2 Comparison with Related Work

Many existing systems may be classified as low-level, promoting flexibility by offering primitive, unstructured communication; for instance Chorus [Zimmermann84], the V-System [Cheriton84] and Accent [Rashid81]. Although in Chorus and the V-System it is possible to broadcast a message to a group of entities [Cheriton85], the address space is flat and there is no simple way to hide the functionality of a subsystem behind a black-box boundary. Communication is unstructured and there is no way to verify that communicating entities are actually using the same protocol.

In the high-level system Argus [Liskov83], on the contrary, protocol agreement is not an issue: the only communication primitive, the "remote procedure call within an atomic action" ensures that a transaction will either terminate correctly without failures and without any interference by concurrent transactions, or will not take place at all. There is a price to pay for this generality of mechanism, which is loss of potential concurrency, and high complexity of the support system. Every application will pay the price, even if it didn't need the full generality of atomic transactions.

We wish to strike a desirable middle ground: enough flexibility to allow many applications and satisfy the operating system builder; and sufficient control to allow the use of atomic transactions (with only its users paying the price). Hence the provision of subsystems represented by a local proxy with a clear, restricted interface.

The idea of using a local representative for a remote service is nothing new. For instance, even though Chorus is a message-based system, a library of (hand-built) procedures, to be bound with the client at link-edit time, gives a procedural interface to services. Accent's Matchmaker [Wri82] builds a similar procedural library automatically from the service specification.

Closer to the preoccupations of this article, LeBlanc [LeBlanc85] advocates the hierarchical composition of processes. In his proposal, a program is a set of processes connected by anonymous communication channels. The output of one process is connected to the input of another by an overseeing process. Any collection of connected processes may be encapsulated by a boundary, forming a new process. The interface to the new process is given by the unconnected channels of its components. He gives the example of a distributed file system, for which there is a local representative on each processor.

The proposal most similar to ours is Nelson's [Nelson81]. He proposes a client-service structure within a flat address space. The link between clients and services is accomplished by so-called "stubs" which give a local procedural interface to the (possibly remote) service. A stub does "marshalling", i.e. stuffs data into packets, and performs the remote call, setting up time-outs and retransmitting packets when necessary. Stubs are brought in at link-edit time or at run time; importing of stubs is controlled by a special configuration language. The system checks that a stub imported at run-time conforms to its compile-time declaration, by checking a timestamp which is incremented every time the service is re-compiled. The normal way of creating a stub is by invoking the "stub translator" at compile time; this

program automatically translates the interface specifications of the service into a conforming stub program. Nelson notes that arbitrary code might be included in a stub by writing it by hand (instead of invoking the translator), allowing access controls for instance; however he gives no examples of this.

What sets our proposal apart from previous ones is our insistence that *all* access to *any* subsystem must be mediated by procedure calls to a proxy. To ensure this, the underlying kernel simply checks that any other form of communication takes place only within a single distributed object group. (We have devised a very simple and efficient implementation of this property.) A proxy is created dynamically by the subsystem it represents, and is bound dynamically to the client. This implies some form of dynamic link-editing, as well as checking that the proxy, imported at run time, conforms to its interface declared at compile time.

5 CONCLUSION

We have proposed to structure a distributed system as a set of services or subsystems, each of which may be made of a number of communicating objects across the network. The interface of any client to any such subsystem is a local proxy. The system ensures that all access must go through a proxy; this allows the subsystem components to be confident that the messages it receives are correct. The proxy subsumes and generalizes many previous proposals: ports, stubs, capabilities. It is more complex but provides more structure than a simple broadcasting facility like process groups. Since proxies are imported at the time of use, and may be (with some care) replaced while running, a service may be changed without having to stop the whole systsem.

The proxy principle gives better facilities for structuring and encapsulation than previous low-level proposals, while allowing more flexibility than the high-level ones. The proxy-based structure also takes one step towards allowing agreement on protocol by ensuring that only agreeing partners may communicate. However, the view taken in this paper is too simple-minded to cope with failures.

Future work planned includes: the implementation of an object kernel supporting the properties listed here, and building an object-oriented operating system based on the proxy principle (currently in progress); work on tools for writing and importing proxies; and developping protocol agreement, including reliable protocols, through the use of a library of communication objects, along the lines of activity messages [Banino85] and Scripts [Francez83].

References

[Banino85] J.-S. Banino, G. Morisset, M. Rozier, "Controlling Distributed Processing with Chorus Activity Messages", Proc. 18th Hawaii Conf. on Syst. Science (Jan. 85).

[Berglund84] E.J. Berglund, D.R. Cheriton, "Amaze: a Distributed Multi-Player Game Program Using the Distributed V Kernel", Proc. 4th. Int. Conf. on Dist. Computing Syst., San Francisco CA (May 84).

[Betourne85] C. Betourne, M. Filali, G. Padiou, A. Sayah, "Distributed Control Through Task Migration via Abstract Networks", Proc. 5th Int. Conf. on Dist. Computing Syst., Denver CO (May 85).

[Birrell84] A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Calls", ACM Trans. on Comp. Syst., vol. 2 no. 1 (Feb. 84).

[Cheriton84] D. R. Cheriton, "The V-Kernel, a Software Base for Distributed Systems", IEEE Software (April 84)

[Cheriton85] D.R. Cheriton, W. Zwaenepoel, "Distributed Process Groups in the V Kernel", ACM Trans. on Comp. Syst., vol. 3 no. 2 (May 85).

[Fabry74] R. S. Fabry, "Capability-based addressing", Comm. ACM. vol. 17 no. 7 (July 1974).

[Francez83] N. Francez, B. Hailpern, "Script: a Communication Abstraction Mechanism", Proc. 2nd. ACM Symp. on Principles of Dist. Comp. (1983); reprinted in the Operating System Review vol. 19 no. 2 (April 85).

[Goldberg83] A. Goldberg, D. Robson, "Smalltalk-80, the Language and its Implementation", Addison-Wesley, 1983.

[Hoare78] "Communicating Sequential Processes", Comm. A. C. M., vol. 21 no. 8 (Aug. 78).

[Hoare79] C.A.R. Hoare, R.M. McKeag, "The Structure of an Operating System", cole de l'INRIA "Mthodologie de la programmation" (March 79).

[Jones79] A. K. Jones, "The Object Model: a Conceptual Tool for Structuring Software", in: "Operating Systems, an Advanced Course", R. Bayer, R. M. Graham, G. Seegmuller (ed.), Springer-Verlag, New York (1979).

[LeBlanc85] T. J. LeBlanc, S. A. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems", Proc. 5th Int. Conf. on Dist. Computing Syst., Denver CO (May 85).

[Liskov83] B. Liskov, R. Scheiffler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs", ACM Trans. on Prog. Lan. and Syst., vol. 5 no. 3 (June 83).

[Wri82] K. Wright, "Matchmaker: a Remote Procedure Call Generator", Technical Report, Dept. of C.S., Carnegie-Mellon University (April 82).

[Nelson81] B.J. Nelson, "Remote Procedure Call", Carnegie-Mellon University report CMU-CS-81-119 (May 81).

[Rashid81] R. Rashid, G. Robertson, "Accent: a Communication-Oriented Network Operating System Kernel", Proc. 8th Symp. on Operating Syst. Principles, Asilomar Conference Grounds, Pacific Grove CA (Dec. 81).

[Ritchie74] D.M. Ritchie, K. Thomson, "The Unix Time-Sharing System", Comm. A. C. M., vol. 17, no. 7 (July 74).

[Shapiro82] M. Shapiro, "An Experiment in Distributed Program Design, Using Control Enrichment", Proc. 3d. Int. Conf. on Dist. Computing Syst., Miami/Ft. Lauderdale (Oct. 82)

[Zimmermann84] H. Zimmermann, M. Guillemont, G. Morisset, J.-S. Banino, "Chorus: a Communication and Processing Architecture for Distributed Systems," Rapport de Recherche no. 328, INRIA, Rocquencourt (Sept. 84).