Processing of confidential information in distributed systems by fragmentation[1]

J.-C. Fabre*, T. Pérennou

LAAS-CNRS and INRIA, 7, avenue du Colonel Roche, 31077 Toulouse, France

Abstract

This paper discusses how object orientation in application design enables confidentiality aspects to be handled more easily than in conventional approaches. The approach is based on the Fragmentation-Redundancy-Scattering technique developed at LAAS–CNRS for several years. This technique and previous developments are briefly summarized. The idea developed in this paper is based on object fragmentation at design time for reducing data processing in confidential objects; the more non confidential objects can be produced at design-time, the more application objects can be processed on untrusted shared computers. Still confidential objects must be processed on non shared trusted workstations. Rules and limits of object fragmentation are discussed together with some criteria evaluating tradeoffs between fragmentation and performance. Finally, a distributed object-oriented support especially fitted for fragmented applications is briefly described.

Keywords: Confidential objects; Fragmentation-Redundancy-Scattering technique; Fragmentation

1. Introduction and problem statement

Processing confidential information in a hostile environment is a difficult issue for which a number of conventional solutions are now well understood. A first solution is to process ciphered information; this solution is mainly based on special ciphering techniques called *privacy homomorphisms* but, although attractive, it is limited in use and can be subject to simple intrusions as described in [1]. A much simpler solution, very costly and of course not realistic, is to process clear information on *trusted and physically protected non-shared computers*. However, in today's systems architecture and heterogeneous environments, one should benefit from powerful shared computers, some of which being specialized for a given part of the application, without making strong assumptions about their internal security and surrounding environment.

In fact, distribution has long been perceived as a conflicting paradigm as far as confidentiality is concerned. Actual solutions to distributed security rely on protection mechanisms (notion of trusted computing base [2]); this means that part of the system must be trusted.

Nevertheless, an efficient protection of confidential information in a distributed architecture may be difficult with standard computers and operating systems. Actually, applications *should* take advantage of the distributed architecture and its specialized components without endangering confidentiality although the information is processed in clear. The main challenge of the work reported in this paper is to propose a new approach for designing such applications.

The proposed approach is part of a general technique for handling both accidental and intentional faults in distributed systems, called Fragmentation-Redundancy-Scattering (FRS) [3,4] and described in Section 2.2. We concentrate here on its application to confidential information processing, in which case the core aspect is *fragmentation at design time*.

An application can be organized in such a way that only a minimal part of the application needs a trusted execution environment. According to well defined assumptions and few mechanisms (Section 2), the rest of the application can be executed on untrusted shared computers. This part of the application can thus be replicated for fault tolerance[2] and scattered without endangering confidentiality.

The approach consists first in identifying during the design phase the information which is confidential. Then, the application can be divided into two parts: confidential data processing and non confidential data processing (Section 3). Minimizing confidential data processing is the

* Corresponding author.

[1] This work has been partially supported by the ESPRIT Basic Research Action no.6362, PDCS2 (Predictably Dependable Computing Systems).

[2] Whatever the type of fault, physical fault or sensitive information destruction.

main objective of the *fragmentation process* which is described in this paper. Using an object-oriented design, the application is seen at each design iteration as a collection of objects. The result of the fragmentation process is a new collection of objects, in which only few objects are confidential and must be executed on trusted non-shared computers (Section 4). The amount of costly non-shared trusted resources is minimized, thus allowing more untrusted shared computers to be used. The method is very application-dependent and of course in some situations the result may be not satisfactory because of performance overheads. Quantitative and qualitative aspects of performance evaluation are discussed (Section 5). Finally, we describe FRIENDS (*Flexible and Reliable Implementation Environment for your Next Dependable System*), an object-oriented support for distributed applications, providing fault-tolerance and security mechanisms in a very flexible way. This type of runtime support provides appropriate facilities for the implementation of fragmented object-oriented applications.

2. System environment and related assumptions

2.1. System architecture

The global architecture of the system is composed of trusted user workstations and untrusted processing servers (see Fig. 1). User workstations are non shared devices[3] with weak computing and memory resources whereas processing servers are shared computing resources where different objects belonging to various applications can be executed simultaneously in an efficient way. The user workstations are trusted computing resources in such a way that, according to their protection mechanisms and surrounding physical environment, the probability of an intrusion during a user session is very low. On the contrary, shared processing servers can he subject to non restrictive intrusions (passive, active, malicious attacks on memory segments, system and temporary files, etc., even performed by privileged users, namely host administrators). These shared processing servers can be specialized off-the-shelf computers of various types with standard operating systems. Confidential information is limited and only processed on few non shared trusted workstations.

Any user application is only activated from the user's trusted workstation and may involve the use of untrusted processing servers for running parts of the application.

2.2. Background and previous work

The aim of the Fragmentation-Redundancy-Scattering technique is to tolerate simultaneously accidental and intentional faults (i.e. intrusions). Both types of faults have to be

[3] They also can be time shared devices: only one user session at a time and no remote access during the user session at any abstraction level.

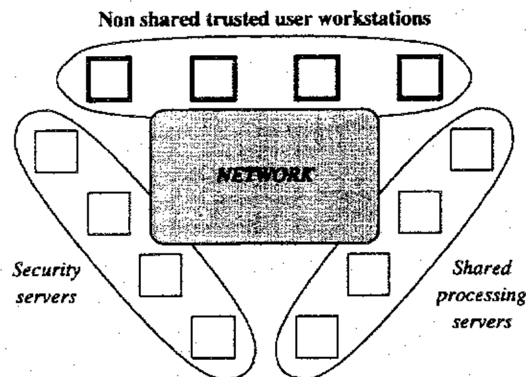

Fig. 1. System architecture and services.

considered in today's distributed systems. The idea consists in defining data fragments or data processing fragments in such a way that each individual fragment does not provide any significant information. Such fragments can thus be scattered in a distributed system in a redundant (even replicated) way to ensure availability and integrity. Our work using FRS lead to several investigations and prototypes development in the past decade, in particular the storage of confidential files and the design of security services, which are described in the rest of the following sections.

2.2.1. Storage of confidential files

This technique was first applied to the storage of confidential files (see Fig. 2) considered as unstructured information (seen as string of bytes, e.g. UNIX files). Files are divided into fixed size pages (16 kB), the last page being completed by padding information. Then each page is ciphered using a simple feedback mode ciphering technique, a checksum is added at the end of the page, and verified when the page is read (integrity control). Each ciphered page is fragmented into a fixed number of fragments (16 fragments of 1 kB): the first byte of the ciphered page goes into the first fragment, the second into the second fragment and so on. Lastly, all produced fragments are replicated and scattered randomly among a large number of computers. Scattering can be performed either in a centralized or in a distributed fashion. Fragmentation, including ciphering and naming of fragments, is based on a secret information, the fragmentation key that must be always available and securely stored. Each fragment is given a unique name (reference) produced by a one-way function with the following parameters (file name, page number, fragment number, fragmentation key).

The notion of page enables just one page to be rebuilt on demand. The size of pages/fragments is arbitrary and can be chosen to optimize the performance of the system. The efficiency of FRS does not rely on ciphering (the ciphering algorithm is simple and fast) but on the difficulty of gathering fragments belonging to a page and of putting them in the correct order before cryptanalysis (16! possible permutations i.e. $\approx 2\ 10^{13}$).

With respect to other similar solutions, such as IDA

(Information Dispersal Algorithm) [5], our fragmentation technique is much less CPU-time consuming. The IDA technique optimizes disk space usage at the expense of a very high computation time.

2.2.2. Security services

Then, the FRS technique was applied to security management such as user authentication and verification of access rights (authorization). Authentication is based on smartcards and performed by a set of security sites. An authentication request is broadcasted to all security sites and each security site locally authenticates the user. This local decision is then broadcasted to all security sites and the final decision is sent to the user workstation. A majority of them must authenticate the user before the user can log into the system. In the end, the user obtains session keys, one for each security site, used for later requests to the security sites (cf. authorization).

Control of permissions (authorization) is based on the same principle. Any open request to an object is sent to all security sites where the access is either accepted or refused according to local information. A voting decision is obtained and if the decision is "accept" the user obtains information to access the object. In particular the user rebuilds the fragmentation key from a set of images. Actually, the storage of very confidential fine grain information (the fragmentation key, for instance) relies on a similar technique to FRS, i.e. Shamir's threshold schemes [6].

By comparison with systems where there is only one security server, such as Kerberos, the security sites are managed by different system administrators. Thus an intrusion performed on a minority of security sites, even by system administrators, do not break the security of the system.

2.2.3. Status and objective

Several prototypes have been developed and various algorithms experimented. The last two prototypes of the complete system axe operational on a network of Sun IPX/IPC machine with SunOS and on a network of PC486 machine running the CHORUS micro-kernel. The services developed on top of UNIX represent more than 30 000 lines of C code, not including cryptographic libraries.

The objective was then to investigate the use of such ideas in a more general context, i.e. for applications and services handling confidential data in general. This can also be understood as a consequence of the synthesis of our experience in implementing specific services with FRS. In doing so, an object-oriented approach emerged quite rapidly: objects gather data and treatments and thus the confidentiality attribute can be easily determined for individual objects, a priori independently of the operations performed on them. This was the first motivation for the use of object-oriented principles and techniques in our developments.

2.3. Software architecture and system services

Applications are seen at runtime as a collection of distributed runtime units (active objects) interacting by messages and owned by a unique user. The underlying runtime system should enable active objects to be executed on any processing unit and to communicate with each other. The ideal runtime support can be an object-oriented runtime layer, such as COOL [7], but better a fault-tolerant execution support for object-oriented applications like FRIENDS (see Section 6).

We also suppose that user authentication, authorization mechanisms and key management are performed by security services running on security servers as in [3,8] (see Fig. 3). When an application is activated by an authenticated user, some active objects are loaded on the trusted user workstation (confidential objects) and the others are loaded on untrusted processing servers (non confidential objects). Each active object is uniquely identified by a reference. The set of objects (references) belonging to a single application is only known at the trusted location. Active objects located on untrusted servers do not know each other even if they belong to the same application. We then suppose that intruders on processing servers are not able to identify objects belonging to a given application. Any object invocation is protected by authentication mechanisms (using

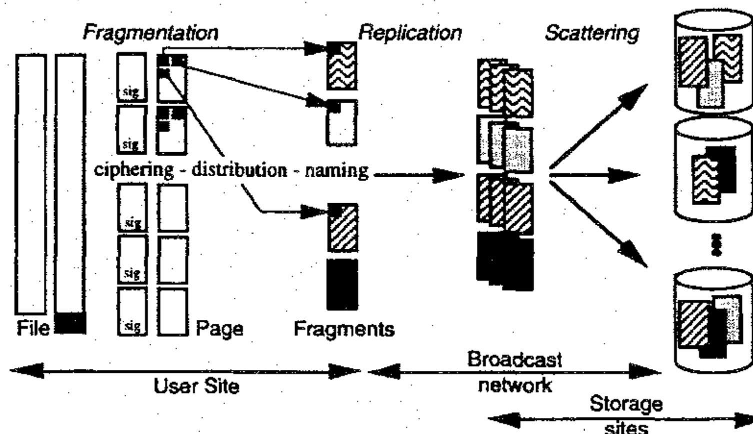

Fig. 2. FRS applied to the storage of confidential files.

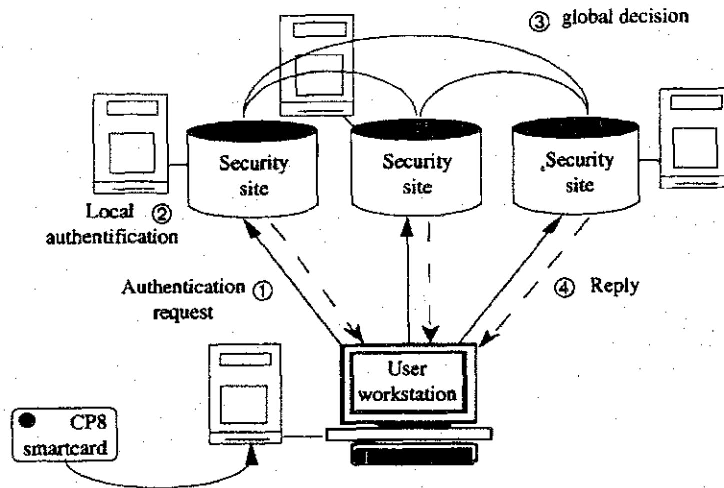

Fig. 3. Authentication.

public key authentication schemes for instance, e.g. [9]) thus preventing illegal invocation by intruders.

3. Confidentiality and object-orientation

3.1. Confidentiality in applications

The notion of confidential information relates to the interpretation an intruder can make about its semantics in a given operational context. Information semantics may be confidential depending on its value: for instance, a string of characters might be sufficiently meaningful in isolation to be easily interpreted as a confidential information independently of any usage in any program. But this is not always the case; a numerical value (in the form of 32 contiguous bits) is most unlikely to be interpreted as a confidential information without any knowledge of its internal representation or of its usage in a given application context. For example, a real variable is a confidential salary information if and only if it is associated to a given person, period and currency[4]. Moreover, this viewpoint is also true using a much coarser granularity; for instance, let us consider a medical record system where the information is classified into two parts, administrative and properly medical. In this quite simple example, confidentiality is preserved as soon as the link between these two large fragments (a pair of references) is retained on the trusted site.

In these simple examples, we can see that structuring a confidential information enables such information to be perceived as a set of non confidential items. The classical approaches do not take into account any structuring of the confidential information, often considered as a string of bits. Our approach relies on a different viewpoint: at a given abstraction level in the design of an application, most of the confidential data processed can be perceived as a collection of insignificant data items: only the links between such items reveal sensitive information to a potential intruder. When the information is not structured (e.g. strings, UNIX files) confidentiality has to be maintained through other solutions such as ciphering techniques, threshold schemes [6], IDA [5] and FRS [3].

3.2. Objet model and confidentiality

Designing the application as a set of objects enables confidentiality to be precisely identified in the application, since, from a software engineering viewpoint, objects abstract real entities whose semantics is well known (a person, a medical record, a bank account, a key). The confidentiality of an object is considered as a boolean function; thus at any level of abstraction the set of application objects can be easily divided into confidential objects (set *C*) and non confidential objects (set *NC*), as shown in Fig. 4.

The architecture of the application is then composed of confidential objects located on trusted workstations with references to non confidential objects located on shared processing servers; the non confidential objects do not communicate with each other, but only with one or several confidential objects.

In our approach, the identification of confidential objects starts from early stages in the design. Such confidential

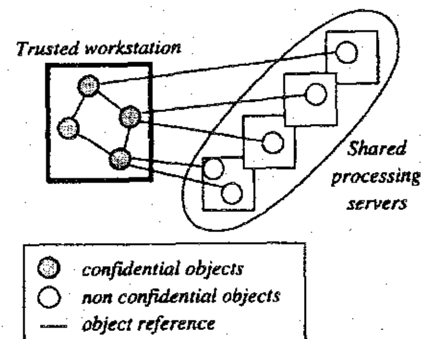

Fig. 4. Confidential and non confidential sets of objects.

[4] Nevertheless, some fine-grain objects are inherently confidential, like a session key for instance.

Fig. 5. Confidential object transformation.

objects can be designed as a set of new objeets, some of which being non confidential. This idea leads to extract as much as possible information and processing from the confidential objects. In Fig. 5 the object O has been redesigned as a set of new objects (O_1, O_2, O_3, O_4, O_5). The objective of this transformation is to have the amount of data and/or processing in the still confidential object O_1 and O_2 much lower than in O. Only objects for which no useful substitution can be found process confidential information.

This approach can be generalized to multi-level security: an application can be divided into secret objects (set S), confidential objects (set C) and non confidential objects (set NC), objects with a high level of classification being re-designed as sets of objects with a lesser classification level.

4. Fragmentation

Fragmentation is here an iterative design process and provides the designer of a confidential application with a general framework which is in fact a variant of a classical object-oriented design (in our case similar to a hierarchical object-oriented design like HOOD [10]). For the sake of simplicity, we consider here as a starting point of the fragmentation process that the application is composed of a unique confidential object which satisfies the functional specifications. At each step of the design process, the designer obtains a set of objects which satisfies the functional specifications of the application at some abstraction level. Confidential objects are identified within the current set. As stated in Section 3, a confidential information can consist of related items including non-confidential ones. Such a structuring is used to design confidential objects as a set of (new) objects including non confidential ones. Among the objects thus produced, the confidential ones are examined at the next step. The process stops either when no more object is confidential or when the granularity of confidential objects is minimal or when no substitution of a confidential object is interesting with respect to confidentiality (e.g. when all objects produced are confidential).

Those points are developed throughout the rest of this section: accurate definitions are given, then guidelines for the substitution of confidential objects are discussed, and finally the whole design is summarized using a more formal expression.

4.1. Definitions

The following definitions will be used in the rest of this paper.

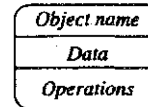

Fig. 6. Extensive representation of an object.

4.1.1. Objects

We consider here a simple object model, where an object encapsulates data and provides a set of operations to manipulate these data, its interface[5]. The interface of an object x is denoted intf(x). The object's data cannot be accessed directly by other objects. Fig. 6 provides an accurate graphical representation of an object which will be used later in this paper. Confidential objects will be presented in grey. Let O be the set of objects thus defined.

4.1.2. Confidentiality

As stated previously, objects abstract real entities, thus allowing the designer to decide whether an object is confidential or not according to the specifications. Let $C{:}O \rightarrow \{\text{TRUE, FALSE}\}$ be the predicate characterizing confidential objects.

4.1.3. Substitution mechanism

Substitution is a designer action that consists in replacing an object x by a non empty set of objects S_x such that S_x provides the same functionalities as x, which we denote by $S_x|x$ (this is read as S_x substitutes for x).

If a set of objects S_x substitutes for an object x, the services provided by x to the rest of the application must be distributed among the objects in S_x. The interface of x is then either located in a unique object in S_x or distributed among several objects in S_x. Throughout the rest of this paper we adopt the first solution which corresponds to a conventional object decomposition:

$$\forall x \in O, S_x|x \Leftrightarrow$$

$$\begin{cases} S_x \subset O & \text{cooperatively provides the same services as x} \\ \exists x' \in S_x & \text{intf(x)} = (\text{intf(x')} \end{cases}$$

4.1.4. Substitutable object

The substitution of a collection of objects for a confidential object is *useful* when the following conditions axe satisfied:

1. non confidential new objects can be produced;
2. gathering such non confidential objects does not enable the confidential object to be easily obtained;
3. processing in non confidential objects is heavy enough to justify the substitution, so that it is interesting to handle part of this processing on untrusted sites and decrease the load of the trusted site.

[5] This definition does not preclude considering an object as an instance of a class, this class being part of an inheritance hierarchy from a software engineering viewpoint.

All these coalitions must be satisfied to consider a confidential object as *substitutable*. For instance when dealing with the substitution of fine-grain objects such as integers or strings the two last conditions are often unsatisfied. Such objects must then be ciphered using conventional techniques (e.g. strings) or be kept in the trusted area (e.g. integers). Let $S{:}O \rightarrow \{\text{TRUE, FALSE}\}$ be the predicate characterizing substitutable objects. In summary:

$$\forall x \in O, S(x) \Leftrightarrow \begin{cases} C(x) \\ x \text{ satisfies each of the three preceding conditions} \end{cases}$$

4.2. Substitution guidelines

This section discusses how the designer should perform the substitution of a confidential object. From the confidentiality viewpoint, the interest of substitution is to produce non confidential objects. When identifying a confidential object, the designer should answer the following questions:

1. Why is this object confidential?
2. How can it be structured and how does it perform the provided operations?
3. Is this structuring suitable with respect to confidentiality and/or performance?

The first question should suggest part of the appropriate structuring requested by the second question, while the third one evaluates the usefulness of the prospective substitution, particularly with respect to other solutions (including keeping the object as a whole, i.e. a still confidential object). In the following, we examine object substitution at early and late stages of the design.

4.2.1. Early stages

At early stages in the design, objects represent complex abstractions. In the example illustrated by Fig. 7, the designer identifies x as confidential and providing an operation called *F*. Then lie determines that x is confidential because it includes a relation between two objects y and z, both providing operations *G* and *H*. So he decides to design x as an object x′ holding two references to y and z and describes the algorithm of *F* in terms of y and z, and their operations *G* and *H* (see Fig. 7). This substitution is suitable because the conditions defined in the previous section are satisfied:

1. (c1) the confidential association has been broken into separate items y an z which are not confidential;
2. (c2) gathering y and z does not provide any confidential information because the link between them is a complex algorithm *F* hidden in x′;
3. (c3) operations provided by y and z are considered CPU-time consuming.

After the substitution, the relation that makes x confidential is distributed among x′, y and z, x′ playing the role of a key; x′ is therefore confidential because it holds the links allowing to rebuild the confidential information.

4.2.2. Late stages

Later in the design, newly defined objects become closer to the implementation level of the corresponding entity. The structure of a confidential object should then appear to the designer as an aggregation of data of simple types within the confidential objects, the operations being series of simple instructions thus making the link rather weak with respect to confidentiality. Fig. 8 illustrates this in terms of objects.

The structuring might seem appropriate since the '+' relating y and z remains hidden in x′. But if we pay more attention to y and z, we can see that y. *GET* = y and z. *GET* = z because they have simple data types: y and z remain in fact local to x′ (see operation *F*). So this substitution does not manage to extract y and z from x and in fact x′ ≃ x: therefore this substitution is useless with respect to the reduction of the amount of data in x.

Going back to the conditions of the previous section, the '+' operation relating y and z is a simple relation which might be extrapolated by an intruder, which can invalidate c2. Moreover, operations provided by y and z are simple, not CPU-time consuming SET and GET operations, which invalidates c3.

So why not considering such x (and more generally objects dealing with fine-grain attributes) as always non substitutable? Actually, the granularity of the objects produced is not sufficient to answer the question. It depends on the complexity of all the operations provided by y and z: if they are CPU-time consuming operations then it can be interesting to run them on untrusted shared resources.

Going through the fragmentation process obviously leads in most cases to fine-grain objects, so performance evaluation is needed to state on the usefulness of a substitution. This issue will be discussed in Section 5.

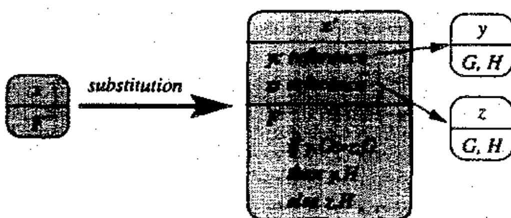

Fig. 7. Early substitution.

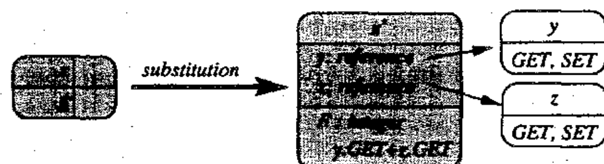

Fig. 8. Late substitution.

4.3. *Formal description of fragmentation*

The process consists in producing recursively a set of objects covering the functional and confidentiality aspects of the specifications. It will be presented here in an algorithmic form. At each step $i \geq 0$, A_i is the set of objects not treated yet. Each step of the algorithm can be described as follows.

4.3.1. *Fragmentation algorithm*

Let $A_0 \subset O$ be the set of objects deducted from the functional specifications. At each step $i \geq 0$ of the recursion, A_i is partitioned in confidential objects, constituting a set C_i, and non-confidential objects, constituting the set NC_i.

$$C_i = x \in A_i | C(x)$$

$$NC_i = x \in A_i | \neg C(x)$$

Among the objects of C_i, some are substitutable and others not, according to the criteria defined in Section 4.1. The C_i set is therefore partitioned in S_i and NS_i, defined as follows.

$$S_i = x \in C_i | S(x)$$

$$NS_i = x \in C_i | \neg S(x)$$

We now consider only the following sets, which are a partition of A_i. S_i is the set of elements of A_i that are confidential and substitutable. NS_i is the set of elements of A_i that are confidential but for which no useful substitution can be found.

NC_i is the set of non-confidential elements of A_i. Therefore $A_i = S_i \cup NS_i \cup NC_i$.

We substitute a set $S_x \subset O$ for each element $x \in S_i$, i.e. $S_x | x$, S_x being functionally equivalent to x, i.e. the interface and services provided by x are also provided by S_x. The algorithm then continues with A_{i+1} defined as:

$$A_{i+1} = \bigcup_{z \in S_i} S_x Z$$

A_{i+1} holds all the objects substituted for all the confidential objects at step i: step i + 1 will therefore only study new objects produced at step i.

4.3.2. *Resulting sets and algorithm properties*

A sufficient condition for termination is that no more object is confidential or substitutable, i.e. $\exists I \geq 0$ $S_i = \phi$. Let then $A = \bigcup_{i=0}^{i=I} A_i$, $C = \bigcup_{i=0}^{i=I} NS_i$ and $NC = \bigcup_{i=0}^{i=I} NC_i$. A is the whole set of objects whose cooperation meets the application specifications, C is the subset of still confidential objects, and NC the subset of non-confidential objects. $A = C \cup NC$, C and NC being a partition of A.

This condition is in fact always satisfied since the granularity of confidential objects eventually becomes very small, and then by definition objects become no more substitutable since their interface comes down only to *SET*/*GET* operations, which invalidates the condition c3. At one extreme, going down to a single bit, the criteria c2 and c3 are obviously not satisfied! More seriously, conventional solutions must be used when confidential objects are simple types of unstructured data. This means that finally the application can include a large number of medium-grain or fine-grain objects[6] although large non confidential objects can be produced during the early stages of the design.

5. Performance evaluation

We have shown in the previous sections that a confidential application can be organized as a collection of objects, some of them being non confidential. For the sake of clarity, we consider here that the sets C and NC have been defined in a first step without considering performance aspects. Security is ensured as long as objects belonging to C are executed on the trusted user workstation (called PC i.e. some personal computer in the rest of this section) in a physically secure environment. The non confidential objects can then be executed on untrusted shared processing servers (called servers in the rest of this section) without threatening the confidentiality of the whole application. We discuss in the following whether it is interesting or not from a performance viewpoint to remotely execute objects of NC on an untrusted server rather than on the PC.

The performance evaluation of a fragmented application depends on several parameters related to both the application (size of invocation/reply messages, method execution time, objects implementation) and the system architecture (relative CPU power of PCs and processing servers, communication throughout the network). The organization of the confidential application in terms of distributed objects must respect some trade-offs between security (fragmentation), performance, cost and usage of the overall architecture. Remote execution of non confidential objects is interesting in several cases and for the non exclusive following reasons:

- When the communication overhead due to the cooperation of objects produced by the fragmentation process is balanced by the better performance obtained by executing non confidential objects on powerful remote computers (including computers with specialized architectures);
- when good parallelism is achieved among the non confidential objects executed remotely;
- When flexibility and extensibility of the network configuration and fair use of existing shared resources are also important goals;
- When fault tolerance is a prime objective that must be achieved through software-based fault tolerance techniques (e.g. non confidential objects can be replicated).

[6] It can also be noticed that the more objects become fine-grain, the more their semantics becomes difficult to obtain.

Only the first of these features can be evaluated quantitatively. However we believe that a qualitative evaluation of other remaining features is of great interest from a pure pragmatic viewpoint. These points are now discussed in detail.

5.1. Quantitative aspects

We suppose here in a first step that the architecture where the implication is to execute is composed of a secure PC and powerful servers, e.g. 10 times as powerful as the PC in a first step. We discuss criteria for determining whether non confidential objects should be run on the PC or on the server and for evaluating the respective costs.

The execution of an object on the PC is almost immediate because it is not shared by several users; just few user applications are simultaneously active in a time-sharing system[7]. On the contrary the remote execution of an object must include time spent on message passing and scheduling on a multi-user processing server.

Just for easy understanding, the balance can be illustrated by the simple following example; suppose that a method M of an object O is invoked by message *im* (invocation message) and results are returned by message *rm* (reply message). Table 1 summarizes the results of an experiment comparing a local vs. a remote execution of M. We denote t_{im}, t_e, t_s, t_{rm} the average times spent on transmission of *im*, execution of M, system scheduling of this execution and transmission of rm respectively. The letters t and τ are used for the PC and for the server respectively. Times are given in milliseconds.

Such values can be easily obtained by simulation or by simply running the object locally on the user PC and also on processing servers. They can also be evaluated using similar techniques as those used for the evaluation of maximum time execution of real time application [11]. In the experiment the method example was run on a PC 486 DX2/33 with UNIX SVR4 and on a Sun Sparc server S690 with SunOS 4.1. The transmission time values reported in the table correspond to the average message time delivery between applications on the (heavily loaded) Ethernet network of workstations of our team for a message size of 1 kB (most object invocation messages are short and can he sent within such message).

In the situation depicted in this table, fragmentation is of interest because performance of the fragmented application is identical to the local execution of the non-fragmented application. The invocation is local on the PC, therefore $t_{im} = t_{rm} = 0$ ms; moreover the PC is not shared and only the current application is active, therefore $t_s = 0$ ms. We consider also that the mean transmission time for the invocation and reply messages is the same: $\tau_{im} = \tau_{rm} = \tau_m$. Therefore the total execution times for t and τ of the local object invocation on a user PC and the remote object invocation on a processing server are:

$$\begin{cases} t = t_e \\ \tau = 2\tau_m + \tau_s + \tau_e \end{cases}$$

The difference between these response times is given by:

$$\tau - t = 2\tau_m + \tau_e + \tau_s - t_e$$

Running $O.M$ on the server is interesting if $\tau - t \leq 0$, i.e. if the remote execution is faster than the local one. Suppose now that the execution time on the server (including scheduling time) is proportional to the execution time on the PC: $\exists \lambda, \tau_e + \tau_s = \lambda t_e$. As the server is supposed to be more powerful than the PC, we have $\lambda < 1$ and therefore:

$$\tau - t \leq 0 \Leftrightarrow t_e \geq \frac{2\tau_m}{1-\lambda}$$

For instance, going back to the simple example given $\lambda = 0.75$; therefore it is interesting to run $O.M$ remotely only if t_e is greater than 80 ms. For simple methods with short execution time, and if we consider more powerful computers (several orders of magnitude as powerful as the user PC), the relative execution time obtained on the processing server (including the average scheduling time) becomes negligible, i.e. $1 - \lambda \simeq 1$ and therefore $\tau - t \leq 0 \Leftrightarrow t_e \geq 2\tau_m$: in this case, a non confidential object can be executed remotely provided that the execution time of the method measured on the PC is greater or equal to the round trip time of invocation/reply messages. Using high speed networks, this round trip time should be much lower than some ms. Thus, objects whose average method execution time is about a few hundreds μs can be remotely executed without any degradation in performance.

When considering processing servers with a specific architecture (such as massively parallel computers) the implementation of an object (matrix computation and other complex numeric computations, image processing) can also be very efficient. This can be true not only because of the hardware architecture of the node but also according to available software tools and libraries. This strengthen the assumption $1 - \lambda \simeq 1$. On the other hand, the designed objects can be implemented to be run in parallel on several nodes. For the above reasons, the scattered execution of the fragmented application is in some cases more efficient than the local execution.

Table 1
Local vs. remote method execution of $O.M$

Time in Milliseconds	Local execution	Remote execution
Transmission of *im*	$t_{im} = 0$	$\tau_{im} = 10$
Exectition	$t_e = 80$	$\tau_e = 10$
Scheduling	$t_s = 0$	$\tau_s = 50$
Transmission of *rm*	$t_{rm} = 0$	$\tau_{rm} = 10$
Total execution time	$t = 80$	$\tau = 80$

[7] A UNIX system with a single user for example.

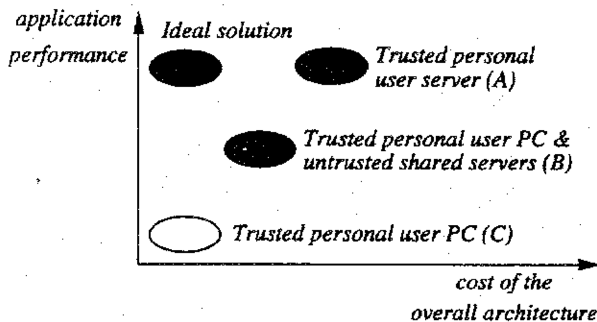

Fig. 9. Trade-offs between performance and cost.

5.2. Trade-offs between cost and performance

We examine in this section several solutions that could be investigated for running confidential applications and we concentrate on qualitative evaluation aspects. The best solution in terms of performance (solution A) should be to only use secured non shared (personal) powerful processing servers. This solution is very costly and the CPU usage very low but it prevents using shared processing servers on which intrusions can be performed. The worst solution in terms of performance (solution C) is to only use personal low cost user PCs in a secure environment for running sensitive applications. The performance is very bad and, as in the previous solution, the execution of the application does not take advantage of existing processing servers. The fragmented solution (solution B) lies in between. This solution involves trusted personal user workstation at low cost and several (existing) untrusted powerful processing servers. This situation is illustrated in Fig. 9.

From a performance viewpoint, solution B can provide better results than solution C as soon as parallelism among objects belonging to NC is considered and the implementation of specific objects is optimized. Solution B maximizes the use of shared (existing) computing power and reduces the cost of the overall architecture (see Fig. 10).

Finally, solution B is of course more flexible since adding either trusted users PCs or processing servers can be done independently according to the needs in terms of user accesses to the system and in terms of computing power. It is also possible to run replicated copies of remote objects for fault tolerance purpose (using software-based techniques) without endangering the security of an application. Fault tolerance at the user station must be based on other solutions, e.g. based on stable storage. Fault tolerance is thus provided with standard computers instead of more costly specific fault tolerant computers. Finally, the flexibility of our solution may allow some degradation in terms of performance to be acceptable.

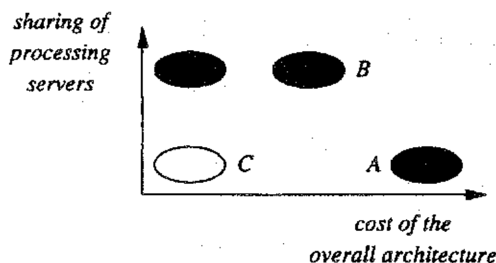

Fig. 10. Trade-offs between performance and sharing.

6. Implementation issues

The execution of such a fragmented application designed as sets of distributed objects implies the use of an appropriate runtime support for distributed objects. This support must provide flexible means for running remote objects, for replicating objects according to various replication strategies, and must also provide means to ensure secure communications between application objects. The first property of such a runtime support must be transparency of any mechanism for the user, either for distribution, fault tolerance or security. All mechanisms required for a given implementation of an application must be *independent* from each other and composed easily on a case-by-case basis. Thanks to object-oriented methods and techniques, some reusability can be expected leading thus new mechanisms to be derived from existing ones. None of the existing solutions in dependable system design and implementation provide all these properties at the same time. The runtime system we are developing today is based on a *metalevel architecture* which enables a good balance of these properties to be obtained. This work lead to the development of the FRIENDS system described in this section. More details can be found in [12].

6.1. Metaobject protocols

The essence of metaobject protocols (MOP) [13] is to give the user the ability to adjust the language implementation to suit its particular needs. Metaobject protocols are based on reflection and object-orientation [14]. Reflection exposes the language implementation at a high level of abstraction, making it understandable for the user while preserving the efficiency and portability of the default language implementation. Object-orientation provides an interface to the language implementation in the form of classes and methods so that variants of the default language implementation can be produced in a simple way, using specialization by inheritance. Instances of such classes are called *metaobjects*. The *protocol* rules the interaction between objects and metaobjects. In class-based reflective languages, the interface of the language implantation generally comprises at least instance creation and deletion, attribute read or write access, method call.

6.2. Structuring the operating system

The architecture of the operating system is composed of several layers (see Fig. 11): (i) the kernel layer which can be

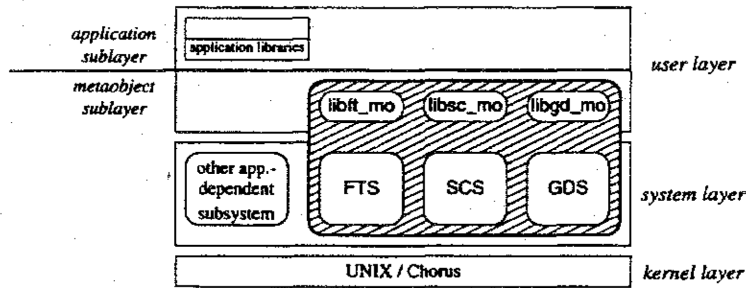

Fig. 11. Overall system architecture.

either a UNIX kernel or better a micro-kernel, such as CHORUS [15], (ii) the system layer composed of several dedicated sub-systems, and finally (iii) the user layer dedicated to the implementation of applications.

The *system layer* is organized as a set of sub-systems: FTS (Fault Tolerance Sub-system), SCS (Secure Communication Sub-system), and GDS (Group-based Distribution Sub-system). FTS provides basic services mandatory in fault-tolerant computing, in particular error detection, configuration management and a stable storage support. SCS provides basic services that must obviously be implemented as trusted entities (Trusted Computing Base) and include an authentication server and also an authorization server. GDS provides basic services for implementing a runtime distribution support for object-oriented applications where objects can be replicated using group membership and atomic multicast protocols.

The *user layer* is divided into two sub-layers, the application layer and the metaobject layer controlling the behavior of application objects. Some libraries of metaobject classes for the implementation of fault-tolerant and secure distributed applications are implemented on top of the corresponding sub-system and provide the user with mechanisms that can be adjusted, using object-oriented techniques. Metaobject classes for various fault tolerance strategies (based on stable storage or replication), for various secure communication protocols, for handling remote object interaction, are now available and used in prototype applications.

6.3. Metaobjects and current status

In our application model, any runtime object is organized using several levels: the first level or base-level (the application object), several intermediate *optional* meta-levels (metaobjects for fault tolerance and secure communication) and finally the last meta-level responsible for handling objects interaction. This structure of the application implies a sequence of interactions through the MOP as shown in Figs 11 and 12.

The metaobject classes have been developed using Open C++, a preprocessor of C++ providing an adequate metaobject protocol [16]. Fault tolerance mechanisms rely on conventional techniques such as stable storage and replication (passive, semi-active, active). Secure communication can be obtained using metaobject classes for ciphering messages (confidentiality) but also for the verification of cryptographic signature (authentication and message integrity. Metaobject classes for fault tolerance mechanisms have been defined with an inheritance hierarchy: a set of mechanisms (based on stable memory and passive replication) have been developed and then factorized as basic classes in a first step. Then, new mechanisms have been obtained using inheritance from these basic classes (semi-active replication strategies, for instance).

Today our experiments axe running on a network of UNIX machines (Sun IPX/IPC under SunOS 4) using the xAMp atomic multicast protocols [17], previously developed in the Delta-4 project [18]. All the metaobjects have been used and combined in various ways in a prototype application (distributed management of bank accounts). We observed the flexibility provided by this approach, in particular the possibility of changing mechanisms without any impact on the application source code. Properties, limits and performance of the FRIENDS system are still under evaluation. Nevertheless, these first experiments are very satisfactory and the performance overheads are reasonable; the extra cost due to multiple metalevel indirections is negligible with respect to the cost of multicast communications.

The implementation of a second version of the FRIENDS system on a micro-kernel platform based on CHORUS is under way. The flexibility provided by micro-kernel technology at the operating system level is complementary to the flexibility provided by metaobject protocols at the language level. Services described in Section 2.2 can easily be integrated in this architecture as a new sub-system, the FRS sub-system. FRIENDS is an attractive runtime support for distributed object-oriented applications in general for

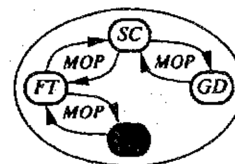

Fig. 12. Base and meta-level interaction.

which dependability is of high interest, and in particular for running fragmented applications designed using the approach developed in this paper.

7. Conclusion

We have shown in this paper that the object-oriented approach to application design enables confidentiality to be easily taken into account since objects abstract real entities having clear semantics; it is thus easy to decide whether an object is confidential or not. Looking more carefully to the question *why* is an object confidential often leads to perceive a confidential information (object) as a collection of non confidential items (sub-objects). Confidential objects can thus be substituted for a collection of sub-objects, some of which being non confidential. Links between sub-objects are kept within still confidential objects for which a trusted computing environment is required. Non-confidential objects can be executed on shared untrusted processing servers. Since the notion of object gathers both data and processing, our solution provides security of data processing in distributed systems.

The performance aspect is one key aspect of object fragmentation. A priori, object fragmentation can be done independently of performance aspects, since the aim is first to encapsulate confidential processing within few confidential objects. The recursion ends as soon as object substitution is not useful from a confidentiality viewpoint (Fragmentation). Performance aspects come into place when the placement of objects has to be decided (Scattering). Remote execution of non confidential object replicas (Redundancy) is sound when it does not lead to high degradation of performances; communication overheads (on high speed LANs) are then balanced by the high computing power of processing servers. The recent and future advances in network technology will make this assumption more and more realistic. As a side effect, simple objects should be executed remotely without any performance degradation. Moreover, considering parallelism between non confidential objects in the implementation of the application, but also for some particular application objects implemented on specific architectures, some gain in performance can be expected.

Finally, from a system architecture viewpoint, the proposed approach to the design of sensitive application provides more flexibility than other solutions. The system architecture can be organized as a set of low cost secured non shared workstations, one for each user, and a set of high performance shared processing servers. The latter computers can be off-the-shelf computers without any specific features with respect to security or to fault-tolerance (just software-based mechanisms). The two types of computing units can be added independently according to the needs.

We concentrate today on the design and implementation of flexible runtime supports for dependable object-oriented applications in general, and fragmented applications in particular. Our recent work based on the use of metaobject protocols lead us to develop the FRIENDS system which provides, as far as we know from our first experiments, a good balance of the expected properties for building fault and intrusion-tolerant distributed systems.

Acknowledgements

The authors wish to thank very much Brian Randell from the University of Newcastle-upon-Tyne (UK) who participated in the elaboration of these ideas during the numerous discussions on the subject.

References

[1] Ronald L. Rivest, Len Adleman, Michael L. Dertozizos, On Data Banks and Privacy Homomorphisms. In Richard A. Demillo, David D. Dobkin, Anita K. Jones, Richard J. Lipton (Editors), Foundations of Secure Computation, Academic Press, 1978, pp. 169–179.

[2] NCSC, Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria. Technical Report NCSC-TG-005, National Computer Security Center, July 1987.

[3] Yves Deswarte, Laurent Blain, Jean-Charles Fabre, Intrusion Tolerance in Distributed Computing Systems. In: Proc. 1991 IEEE Computer Soc. Symp. on Research in Security and Privacy, Oakland, CA, May 1991, IEEE Computer Society Press, pp. 110–121.

[4] Jean-Charles Fabre, Yves Deswarte, Brian Randell, Designing Secure and Reliable Applications using FRS: An Object-Oriented Approach. In: Proc. 1st European Dependable Computing Conference (EDCC-1), Lecture Notes in Computer Science 852, Berlin, Germany, 1994, Springer-Verlag, pp. 21–38.

[5] Michael O. Rabin, Efficient Information Dispersal for Security, Load Balancing and Fault Tolerance. JACM 36(2) (1989) 335–348.

[6] Adi Shamir, How to Share a Secret, Communications of the ACM 22(11) (1979) 612–613.

[7] Rodger Lea, James Weightman, Supporting Object-Oriented Languages in a Distributed Environment: The COOL Approach. In: Proc. 5th Technology of Object-Oriented Languages and Systems Conference, Santa Barbara, CA, USA, 1991, Prentice Hall, pp. 37–47.

[8] J. G. Steiner, C. Neuman, J. L. Schiller. Kerberos: an Authentication Service for Open Network Systems. In: Proc. USENIX Winter Conf., Dallas, TX, February 1988.

[9] B. Taylor, D. Goldberg. Secure Networking in the Sun Environment. In: Proc. USENIX Summer Conference, Atlanta, GA, 1986, pp. 28–37.

[10] Peter Robinson, Hierarchical Object-Oriented Design, Prentice Hall, 1992.

[11] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. IEEE Micro 9(1) (1989) 25–40.

[12] Jean-Charles Fabre, Tanguy Pérennou, FRIENDS – A Flexible Architecture for Implementing Fault Tolerant and Flexible Distributed Applications. In: Proc. 2nd European Dependable Computing Conference (EDCC-2), Taormina, Italy, September 1996.

[13] Gregor Kiczales, Jim. des Rivières, Daniel G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[14] Patti Maes, Concepts and Experiments in Computational Reflection, In: Proc. 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, FL, October 1987, ACM Press, pp. 147–155.

[15] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, S. Langlois, P. Léonard, W. Neuhauser,

Overview of the CHORUS Distributed Operating Systems, Technical Report CS-TR-90-25, Chorus Systèmes, 1990.
[16] Shigeru Chiba, Takashi Masuda, Designing an Extensible Distributed Language with Meta-level Architecture, In: Proc. 7th European Conference on Object-Oriented Programming, Kaiserslautern, Germany, July 1993, Springer-Verlag, Lecture Notes in Computer Science 707, pp. 482–501.
[17] Luís Rodrigues, Paulo Veríssimo, xAMp: A Protocol Suite for Group Communication, In: Proc. 11th IEEE Symp. on Reliable Distributed Systems (SRDS-11), Houston, TX, October 1992, pp. 112–121.
[18] David Powell. Distributed Fault Tolerance — Lessons Learnt from Delta-4 IEEE Micro 14(1) (1994) 36–47.