Designing Secure and Reliable Applications using Fragmentation-Redundancy-Scattering: an Object-Oriented Approach

Jean-Charles Fabre*, Yves Deswarte*, Brian Randell**

*LAAS-CNRS & INRIA
7, avenue du Colonel Roche
31077 Toulouse cedex
(France)

**Department of Computing Science
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU
(United Kingdom)

Abstract. Security and reliability issues in distributed systems have been investigated for several years at LAAS using a technique called Fragmentation-Redundancy-Scattering (FRS). The aim of FRS is to tolerate both accidental and intentional faults: the core idea consists in fragmenting confidential information in order to produce insignificant fragments and then in scattering the fragments so obtained in a redundant fashion across a distributed system, such as a large network of workstations and servers. Of these workstations, in principle just the user's own workstation needs to be regarded as trusted, whereas from this user's viewpoint the other workstations and servers, which in all probability are under someone else's control, can be untrusted devices.

This paper describes an object-oriented approach to the use of FRS, now under development at LAAS and Newcastle. This approach greatly eases the task of application programmers who seek to ensure reliable secure processing, as well as storage, of confidential information. The approach involves fragmenting a confidential object using its composition structure, i.e., in terms of a hierarchy of sub-objects (the "*is-part-of*" relation of the object model), each of course with its own subsidiary operations or "methods". The fragmentation process continues until the resulting sub-objects are as far as possible such as to be individually non-confidential. Replicas of non-confidential objects are then scattered among untrusted stations. By such means much of the processing of object methods, as well as the storing of much object state information, can be carried out safely on untrusted equipment.

1 Introduction

Mechanisms for fault tolerance in distributed systems are typically designed to cope with just a limited class of faults: usually just accidental, physical faults which occur during system operation (some designs take into account only an even more restricted subclass, such as crash failures). However, other classes of faults may also impede correct operation of distributed systems; nowadays a numerous such class is certainly that of intentional human interaction faults, i.e., intrusions. These are deliberate

This work has been partially supported by the ESPRIT Basic Research Action n°6362, PDCS2 (Predictably Dependable Computing Systems)

attempts at transgressing the security policy assigned to the system. They can originate from external intruders, registered users trying to exceed their privileges, or privileged users, such as administrators, operators, security officers, etc., who abuse their privileges to perform malicious actions.

Intrusions and accidental faults may have the same effects: that is the improper modification or destruction of sensitive information and the disclosure of confidential information. The user will perceive these effects as a system failure: the service delivered by the system to the user no longer complies with the system specifications[1] [1]. In distributed systems composed of users' individual workstations and shared servers, users can generally trust their own workstation providing that they control it completely, while an individual user usually distrusts the servers and the other workstations because he/she cannot know directly if these servers and workstations are failing or have been penetrated by an intruder. On the other hand, server administrators and users distrust other workstations, for the same reasons. However the trustworthiness of the distributed system can be improved if it is fault-tolerant, i.e., if the failure of a server or of a workstation is not perceived at the other workstations, irrespective of the cause of the failure, be it an accidental physical fault or an intrusion.

Because they do not take intrusions into account classical fault tolerance techniques, such as data and processing replication, although they can help to tolerate accidental faults, do not provide means of preserving confidentiality. Indeed, if intrusions are to be taken into account and if confidentiality of sensitive information has to be maintained, simple replication will decrease system trustworthiness, since several copies of confidential information can be targets for an intrusion. This was the motivation for a technique which has been developed at LAAS for tolerating faults while preserving confidentiality, namely the fragmentation-redundancy-scattering (FRS) technique [2]. Fragmentation consists of breaking down all sensitive information into fragments, so that any isolated fragment contains no significant information. Fragments are then scattered among different untrusted sites of the distributed system, so that any intrusion into part of the distributed system only gives access to unrelated fragments. Redundancy is added to the fragments (by replication or the use of an error correcting code) in order to tolerate accidental or deliberate destruction or alteration of fragments. A complete information item can only be re-assembled on trusted sites of the distributed system. The FRS technique has already been applied both to the storage of persistent files and to security management; this work has been described in several earlier papers, in particular in [2].

The aim of the present paper is to show how FRS, and in particular object-oriented FRS, can be used in the design and in the implementation of any application or system service so as to achieve not just reliable and secure storage but also secure processing of confidential information (e.g. protection from eavesdropping or interference). Secure processing of confidential information has been investigated

---

[1] System specifications describe what the system should do, according to performance and reliability requirements, as well as what it should not, according to safety or security requirements (e.g. the hazardous states from which a catastrophe may ensue, or the sensitive information that must not be disclosed to or modified by unauthorized users).

elsewhere using more conventional ciphering approaches, i.e. the scheme of processing ciphered data described in [3]. Such approaches need specific ciphers ("Privacy Homomorphisms" [4]) and are rather limited and relatively inefficient; simple attacks can manage to get clear information. The approach which is advocated in this paper is quite different since it relies on the fact that confidential information can very often be decomposed into a collection of non confidential items on which processing can be done in clear text. The original attempt to extend FRS to cover information processing [5] required significant manual redesign of the application programs whose execution was to be protected. In this paper we discuss how such requirements for application program redesign can be avoided by allying the FRS technique to an object-oriented approach to system design. In addition, we develop in this paper a scheme of "confidentiality constraints" expressed in terms of first-order logic formulae for defining the confidentiality requirements imposed on a given application, and provide a brief description of the first experiment on the use of FRS for information processing.

2 Distributed system architecture and assumptions

The distributed system architecture (cf. Fig. 1.) which we consider in this paper is composed of a set of trusted workstations (more exactly user workstations which are trusted by their respective users), and a set of untrusted machines which are the basis for providing a set of fault-tolerant secure servers. A user of a trusted workstation is responsible for the security of his/her workstation and also for taking all necessary physical security precautions for ensuring that such sensitive actions as logging in, and any required authentication are not being observed. During a session of usage of such a trusted workstation, that workstation resources are not sharable (e.g., remote access by others to the workstation is disallowed). Confidential information will be stored on such a workstation during a usage session. However, unless subsequent security precautions concerning access to that workstation are deemed adequate, such information will not be left on a workstation after completion of the session. (We do not consider network-related security and reliability issues in this paper, but would merely remark that analogous techniques to FRS, involving spread spectrum communications, already exist, as well of course as numerous conventional ones.)

In this paper we assume the provision of two types of services already implemented using untrusted sites, namely the provision of storage and authentication/authorization. The use of conventional FRS for such provisions has been successfully demonstrated - see [2]. The authentication and authorization are realized by a security server implemented as a set of security sites, administrated by different operators. As long as a majority of security sites is free of faults and intrusions (including intrusions by the security administrators), user authentication and access control are reliably achieved and no sensitive information is disclosed. This security server can implement various security policies, including multi-level security (MLS) policies.

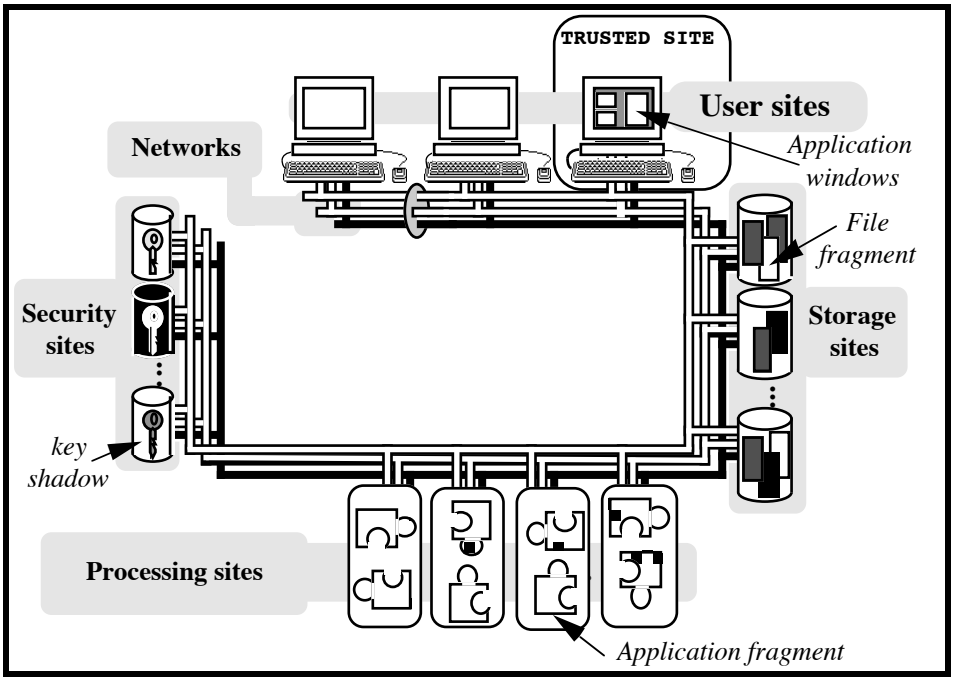

Fig. 1 : Distributed system architecture

With regard to these services, our fault assumptions encompass accidental faults, physical faults that would affect untrusted sites, but also any type of intrusion that would affect the untrusted sites or the networks.

Although we admit the possibility of intrusions of untrusted sites, we nevertheless assume that such intrusions are not particularly easy to carry out, and that the effort an intruder will have to provide to intrude separately several sites is proportional to the number of sites involved. (Clearly, the mechanisms described in this paper are intended to ensure that successful intrusions at one or a small number of untrusted sites does not provide means of accessing or modifying data or processing activities that are the responsibility of any other untrusted site.)

3 FRS data processing

3.1 Principles

The aim of the original FRS technique was to provide a general framework for the reliable processing of confidential information, assuming that what matters is the confidentiality of the information being processed (the data) rather than the confidentiality of the operations performed on it (the program). This was later extended to provide confidentiality of information processing [5]. For any application program or system service, such use of FRS results in the transformation of the software into a fragmented form according to several basic rules:

1. the application including code and data is divided into application fragments in such a way that the cooperation of the application fragments satisfies the specifications of the initial (complete) application;

2. any application fragment shall not provide any confidential information to a potential intruder on the site where the application fragment is located;

3. all the application fragments shall be scattered among the sites of a distributed architecture (separation) in such a way that groups of fragments stored at a given site provide no significant information to an intruder;

4. appropriate redundancy must be introduced either during fragmentation or scattering;

5. as far as possible, an intruder shall not be able to identify fragments belonging to the same application process or to the same object, since application fragments shall be identified from the user site by enciphered references.

A major problem with the use of this original FRS technique was that of how to deal with fragment code, and in particular how to deal with global variables, a problem whose solution frequently involved partial redesign of the application programs involved. This problem provides much of the first motivation for the use of object-oriented techniques described in this paper.

3.2 Object view of FRS

The object model used here is not specific to any particular object-oriented programming language: we simply assume that *objects* are derived from *classes* and encapsulate data structures that can be manipulated only by a set of functions (*methods*); *objects* can be decomposed into *sub-objects* that can be identified by *references*. The use of *inheritance* is not discussed very much in this paper. Nevertheless, inheritance can be used for programming FRS applications in conjunction with other properties of object-oriented programming languages, such as *reflection* (see Section 5.4).

The main interest of the object model in connection with FRS is that the fragmentation, being in terms of objects, naturally encompasses program code as well as data. It can normally be applied to an existing application program without requiring the designer to reprogram the application - all that has to be done is to identify which object classes are to be used as the basis on which data and code is to be fragmented. Such identification involves deciding at what level of the object structuring it will be the case that the individual objects, when examined in isolation, do not contain confidential information. Thus the programmer simply has to provide what are in effect some additional declarations, rather than invent new fragmented algorithms (which is what the original method of extending FRS to information processing required).

The design approach which is proposed in this paper thus relies on the fact that the fragmentation of the application can be based, at design time, on the semantics of the information being processed. The designer of the application has therefore to find an appropriate design structuring to obtain non-confidential objects and thus to define

application fragments. The object model offers a convenient design framework for several reasons: the object notion encapsulates information, objects can be decomposed into more elementary objects, and any object can readily be mapped onto an autonomous runtime unit on an appropriate fault-tolerant distributed system.

This approach can be used in different ways and for various applications. For example, in transaction-processing applications, large amounts of confidential information can be held in persistent objects but, in this case, the amount of processing may be relatively limited. The information and the operations performed can be organised (structured) in such a way that individual actions of a transaction are remotely executed by non-confidential objects. In other applications, such as numerical computations, processing is very intensive but objects are mainly temporary because there is little persistent state and thus all input parameters can be given for each activation. (In each case, the links i.e., the references, between objects belonging to the same application are kept secured at the trusted user workstation, where the application is started.)

The object-oriented approach to the use of FRS is thus attractive for implementing various types of applications that hold and process confidential information. A particular characteristic of the approach is that it provides application designers with a single unified design scheme for making their applications tolerant to both accidental *and* intentional faults.

4 Notion of confidential information

4.1 Principles

The notion of confidential information relates to the interpretation an intruder can have 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 a program. But this is not always the case; a numerical value 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, the bit string corresponding to a salary variable that holds the value 20000 in the data segment of a program must be mapped to a real representation in the machine before it could be interpreted as a real value. However this is not sufficient, as a confidential information item is in fact a combination of sets of items that bring together information to a potential intruder. Such an intruder can get meaningful salary information if and only if he is able to associate together several information items such as: person name, salary amount, salary period and currency. This simple example shows that very often, thanks to its structure, a confidential information item is in fact a set of non-confidential data items.

This notion of confidential information defined as a set of public items may not be appropriate in some applications or for the management of unstructured objects (strings, keys, files, etc.) where the semantics is unknown. For instance, in the file storage system described in [2], FRS was applied to unstructured files (Unix files) and was based on the use of ciphering techniques and a scheme of regular fragmentation to

produce fragments. Other techniques, such as threshold schemes[2], can also be used to deal with non-structured objects: a number of items higher than the threshold must be gathered to reconstruct the secret [6]. This technique has mainly been used for small information items such as cryptographic keys. A similar approach was also used at a coarse granularity in [7]. In the last two cases, fragmentation provides both redundancy and ciphering of the data.

The coexistence of both classes of fragmentation techniques can be illustrated by another example (in fact one which is used in our current major experiment): suppose a meeting of a group of people is a confidential information item. The information about the meeting is composed of a list of participants, a given topic, a venue and time/date items. A participant is defined by his/her personal identity which may be considered as public information; the same assumption can be made for other items such as the venue. However, the information about a meeting might be confidential because of the topic discussed and also because of the identities of the participants attending. Keeping the meeting information secret may involve ciphering the topic (given the lack of structural semantics of a character string) and scattering the list of participants ; only appropriate references to participants need then to be kept in the meeting object. An operation on the participant list itself is performed within the meeting object at a given site, while operations on the participant information are performed at other sites in the network where those participant objects are located.

4.2 Confidentiality constraints

The fragmentation principle relies on the notion of confidentiality constraints that define the confidential information used in the application. These confidentiality constraints are first expressed informally as part of the non-functional specifications of the application. These non-functional specifications are interpreted by the application designer so as to define an appropriate structuring so that each confidential information item is broken down into non-confidential items. In each object in the design, the information is structured in terms of a collection of sub-objects representing information items.

The interpretation of informal confidentiality constraints can be more formally described in terms of first order logic formulae. For instance, going back to the simple example given in Section 4, the confidential *meeting* information can be structured into more elementary objects such as *topic, time/date, venue, person_list* . The formula *{meeting == topic ∧ time/date ∧ venue ∧ person_list}* indicates first that meeting is decomposed into the aforementioned items and, second, that the conjunction of these items reveals sensitive information. Another example would be the following: *{meeting == (topic ∨ time/date ∨ venue) ∧ person_list}*; any combination of *person_list* and the topic discussed, or the location, or the date of the meeting is confidential. If the specifications indicate that the list of attendees is also a confidential information item for any meeting, then *{person_list == person*

---

[2] Threshold schemes consist in generating, from a secret information, several shadows so that a given number T of shadows (T being the threshold) is necessary to reconstruct the secret information, whereas T-1 shadows does not reveal any confidential information. The number of shadows is greather than, or equal to T in order to tolerate faults and intrusions.

[∧ person]}* indicates that any group of persons in the *person_list* is confidential information.

Such clauses specify in fact that the left hand side corresponding object is confidential because the right hand side logical formula composed of sub-objects may reveal confidential information to an intruder. Any sub-object in one formula may also be confidential and then be defined by another clause. Finally, a special clause is needed to specify the set of unstructured objects that are also confidential:

Unstructured confidential objects == *{<object> [, <object>] *}*

It is important to mention here that such a formal definition of confidentiality constraints by means of a set of clauses leads one to identify objects (in italic) used in further steps of the design process.

5 Object-oriented FRS

Based on the object model described in Section 3.2, the fragmentation design process operates on a strong structuring of the information in terms of a hierarchy (composition) of objects. In any object, confidential private information can be structured as a set of more elementary objects. The fragmentation is thus based on an appropriate structuring, as originally defined by the designer. The FRS design approach involves two main tasks:

i) definition of basic objects (classes) that do not contain confidential information or whose confidential information is ciphered, based on the object composition hierarchy (fragmentation);

ii) creation of autonomous instances of these basic objects in a large set of untrusted sites of a distributed computing system (scattering).

The main idea of the object oriented FRS is that it is a recursive design process that operates on the hierarchical representation of the application and yields application fragments; the recursion ends as soon as, on every branch of the design tree, an object that does not process any confidential information is encountered, or, no further decomposition exists already or can be applied (in which case the data in the object must be enciphered if its confidentiality is to be protected). The corresponding runtime fragments are then scattered among the distributed architecture and communicate via messages. If fragmentation by itself does not introduce adequate redundancy, then fragments are replicated before being scattered.

5.1 Fragmentation

The fragmentation design process can involve several design iterations, starting from a first version of the design of the application, i.e., a first object composition tree. At each iteration, the designer performs an analysis of the list of confidentiality constraints of the application in order to identify the objects containing confidential information. Then a new design step can be started if some confidential object can be decomposed into, or is already defined in terms of, more elementary objects. This new design step produces a refined version of the object composition tree. Then the

designer goes back to a new analysis of the confidentiality constraints that have not been solved by the previous design (see Fig. 2).

```
for any <object > in current design tree
do
        if object is confidential then
                decompose object further (fragmentation)
        or      apply ciphering technique
        or      leave it to a trusted site allocation
        end_if
end_for
```

Fig. 2: Fragmentation principle

This iterative design process with its analysis of the confidentiality constraints, continues until non-confidential objects are obtained or a confidential leaf is reached, and terminates when there are no more confidentiality constraints to solve in the list. Finally, should there remain any confidential objects that cannot be structured into more elementary objects, which might either be due to their granularity or their functionality, ciphering techniques are used.

5.2 Redundancy

Several approaches can be used for adding redundancy to fragments. Various error processing techniques may be used either when the runtime units corresponding to design objects are created or at a early stage during the design of the application in term of objects.

The underlying runtime system may offer a set of transparent error processing protocols that can be selected at configuration time to install runtime units in a redundant fashion, as in Delta-4 [8]. The latter relies on detection mechanisms and voting protocols implemented by the underlying multicast communication system. Several checkpointing strategies between passive replicas and synchronisation strategies between active replicas are available.

Another approach consists in defining the error processing technique at an early stage in the design using pre-defined system classes that are responsible for the implementation of a given solution. The idea is to use the notion of inheritance of the object model to derive a fault-tolerant implementation of any object. This solution consists in fact in making inheritable non functional characteristics, using appropriate system classes and programming conventions. This type of solution has been used in particular in the Arjuna project [9] where for example any object can be declared as recoverable.

This declaration means that any object from this class will be created in a redundant fashion, provided that some declarations are given by the object designer (virtual function definition, function overloading). System classes must provide by inheritance a large number of error processing protocols; the development of system classes can take advantage of basic system services such as error detection and

recovery, atomic broadcast, various voting protocols, stable memory management. The first work on the use of object-orientation in connection with FRS assumed that conventional object-oriented inheritance would similarly be used to declare *secured* objects [10]. However, there are significant problems with such an approach, and we now think that the use of reflection is more promising approach (see Section 5.4).

5.3 Scattering

The scattering phase consists then in allocating object-fragments replicas to the computing sites; any object instance must be created as an autonomous computing unit, i.e., mapped onto a basic runtime unit of the underlying operating system. This aspect is discussed in Section 6.1.

```
for any <fragment> in current fragment set
do
        if object-fragment is still confidential then
                        allocate to a trusted site
        else
                until a valid untrusted site is allocated
                        allocate to an untrusted site
                        if not creation of a confidential group of objects
                        then    this site is a valid site
                end_until
        end_if
end_for
```

Fig. 3: Scattering principle

The scattering phase is summarised in Fig. 3. The main problem in the scattering phase is to avoid creating sets of objects on the same site that correspond to a confidential information item. Confidentiality constraints between fragments must then be taken into account to identify such groups of fragments. The first simple rule is that object-fragments having the same parent object-fragment must be located on different sites. But this rule is not sufficient; scattering may group fragments which are not strictly brothers in the hierarchical design but that may reveal confidential information. A careful analysis of fragment groups must be done, especially if there are relatively few sites available to receive scattered fragments.

5.4 Use of inheritance and reflection

From an object-oriented programming language viewpoint, FRS leads to the scattering of sub-objects of a given object. This means that when the object is created, some or all of its sub-objects may need to be created remotely. Subsequently, the conventional scheme for invoking the methods of such remote sub-objects must be replaced by a scheme of remote method invocation.

The provision of means for so redefining what are normally basic internal operations (object creation and method invocation) of the language runtime system is not common. However some object-oriented languages do have the property that they

provide access to such operations and the ability to modify them in the language itself: this property is known as *reflection*.

Clearly, even if it were possible, it would be undesirable for the application programmer to have to program such a redefinition scheme explicitly in the definitions of each class of objects whose confidentiality is to be protected. What is needed is a means of indicating, for any given class, that such a scheme is to be used. In other words, if one considers that a class is itself an object belonging to some meta-class, the requirement is to have some means of providing in the definition of the meta-class, the methods of object creation and method invocation that are to be used by any of its class objects. Such a *reflective* facility in fact would seem to be of great promise for not just for object-oriented FRS, but also for implementing various non functional characteristics using object-oriented languages [11].

This type of facility was first provided in Common LISP, but has been recently implemented in a variant of C++ called OpenC++ [12], in which both method invocation and also access to local variables can be captured and re-defined at the meta-level. In OpenC++ the application programmer can readily redefine access behaviour at the meta-level. Inheritance of pre-defined meta-classes allows the definition of new meta-classes for any object class in the application. This scheme is now being investigated as a means of providing FRS facilities; the objective is to define meta-classes for every confidential class in the application, thus hiding object creation problems (including replication and scattering of sub-objects), but also remote access problems (including reference computation and access control mechanisms) from the application programmer.

5.5 Summary

The complete design process can be summarised in the several tasks that are represented in Fig. 4. This figure shows the major steps of the design and implementation of an FRS application. Several iterations on the design of the application taking into account confidentiality constraints on the information being manipulated, lead to the definition of non-confidential objects. These non-confidential objects are the application fragments. According to the runtime abstractions provided by the runtime system, application fragments are mapped onto autonomous runtime units. Adequate error processing protocols are then selected on an object-by-object basis leading to a set of autonomous runtime object replicas. This selection takes into account the functionality of the object and also the accidental fault assumptions that can be made regarding the available sites on the distributed configuration. The last phase of the design process consists in scattering these replicas. The scattering phase must take care to avoid gathering together groups of objects that can be perceived by an intruder as constituting a confidential information item. Confidentiality constraints between object replicas must thus be taken into account for the allocation of sites to runtime units. The set of replicas is in fact divided into two subsets: (i) object replicas that do not contain confidential information and that can be executed on untrusted stations, but also (ii) the set of some still confidential objects that must be executed on trusted sites of the distributed system.

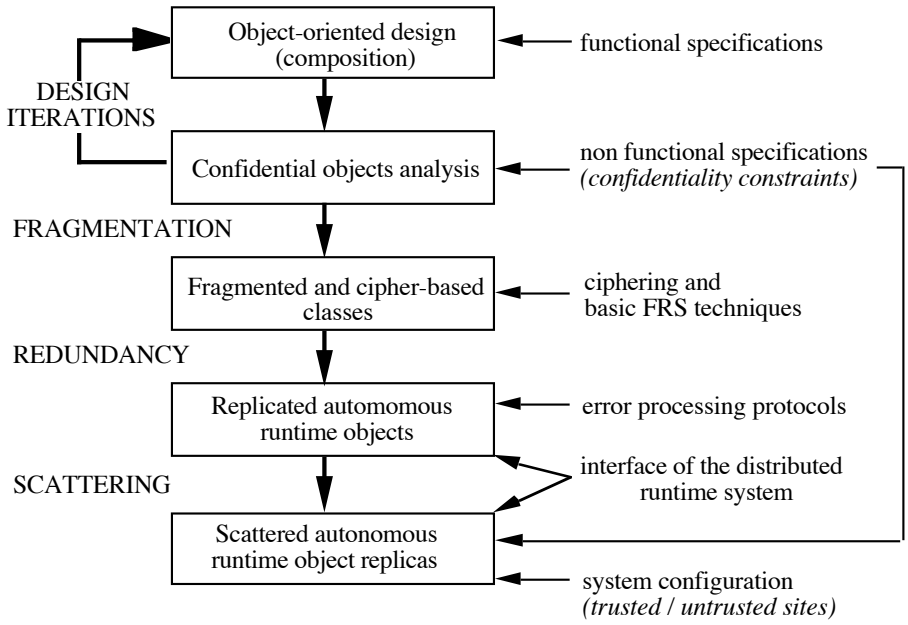

Fig. 4: FRS Application Design steps

6 Implementation issues

6.1 Distributed runtime environment

The degree of difficulty involved in implementing an object-based application largely depends on the abstractions provided by the distributed runtime system. Object fragments have to be mapped onto autonomous runtime units. The system we have used for our current major experiment, the Delta-4 system, does not provide the notion of object; instead it provides the notion of a server, though this is not far from the object notion as previously defined. It corresponds to a private address space and a set of operations with well-defined interfaces. Object mapping can be done in various ways: (i) any object instance corresponds at runtime to a server, or (ii) a server is responsible for any instance creation for a given class. The second of these approaches is the one we have used. The Delta-4 distributed runtime layer, namely Deltase[3], provides server mapping on top of Unix (the local executive) and a transparent multiple remote procedure call mechanism used for remote method invocation between object manager replicas. The set of servers provides an object management layer on top of the distributed runtime layer.

In the implementation of FRS, the object runtime layer may involve several instance managers (Deltase servers) per class. At one extreme, any site on the network may provide an instance manager for any class in the application. The scattering algorithm

[3] DELTASE : DELTA-4 Application Support Environment

may then allocate any object instance on any site. Objects can be created dynamically by invoking the appropriate create operation of the corresponding instance manager. The Delta-4 distributed runtime system layer includes a set of error processing protocols used to install replicated servers.

6.2 User authentication and authorization

As indicated in Section 2, user authentication and authorization are achieved by a distributed security server composed of several security sites. A user is authenticated when at least a majority of security sites agree to authenticate him [2]. One authenticated, the user can request access to services. This request is evaluated by each security site according to user privileges, service access control list and security policy. All the sites decisions to grant or deny the access are voted on each security site and if a majority is reached to grant the access, an access key is transmitted from the security sites to the user site by means of a threshold scheme [6].

The access control approach, briefly presented in this paragraph, is used for any application, system server or simply any object (files) implemented by FRS on untrusted computing resources. The key which is gathered at the user site, will be used latter on by the application for referencing fragments using cryptographic functions (see Section 6.3.).

6.3 Reference management

The scattering of objects in a distributed environment requires an identification mechanism to allow remote invocation. In fact, most of the security of FRS relies on the fact that an intruder is not able to gather fragments from outside the trusted user site or to invoke objects (fragments) directly. The reference[4] management system must first ensure that related fragments (belonging to the same application) cannot be identified just by looking at object references. References can then be dynamically computed at the trusted site using the secret key, provided for this application and for this user by the authorization protocol.

Looking more carefully at a fragmented application (cf. Fig. 5), one can see that the application is in most cases implemented finally as a "star structure" whose centre is located at the trusted user site. The centre of the star is at least the root of the object composition tree.

An ideal reference system must ensure: (i) unique identification of the remote object-fragment, (ii) authentication of the invoking application, and (iii) verification of permissions on the invoked object:

$$\text{reference} = E_k\,(\text{object\_name}, \text{application\_name}, \text{object\_permissions}).$$

A very simple way of using references can just be to consider them as capabilities: as soon as they are provided to an object manager (i.e., when the reference is known) then the corresponding object is activated. In this case, the ciphering algorithm **E** is a one-way function and **k** is the application secret key.

---

[4] A reference is viewed here as a generalisation of the notion of pointer in a distributed environment.

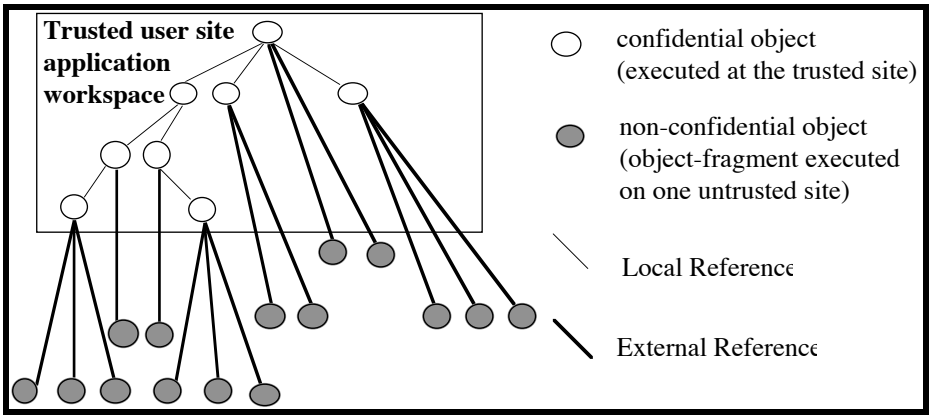

Fig. 5 : Structure of a fragmented application - Local and External references

A more sophisticated solution would be to decipher the reference at the object manager site to check authenticity and permissions. In that case a shared secret key must be used to implement this solution; the key must then be kept securely in any station in a local trusted sub-system (local TCB [13]). In this case, the ciphering algorithm **E** is based on a secret key cryptosystem and **k** is a secret key shared by the user application at the user site and one of the untrusted sites (where one copy of the invoked object is located).

Finally, shared objects between two or more different applications will have different references, thus preventing search by induction on shared objects.

7 Experimentation

We have investigated the above FRS design approach on a detailed example, a distributed Electronic Diary, which has been implemented on the Delta-4 system. A more detailed description can found in [10]. We describe here this application using a small series of classes, so leading to a hierarchical design of the E-Diary. In this simple example, a number of confidentiality constraints on the processed information have been defined and taken into account. The processing facilities provided (i.e. the operations that can be performed on defined objects) are in fact very limited in the current version of the E-Diary application and the defined objects are persistent. Another possible type of application would be to have no persistence and heavy computation such as in numerical computations on sensitive information (e.g. missile trajectory computation). However, the E-Diary example provides a convenient means of illustrating the object-oriented FRS design steps described in Section 5.

7.1 Functional specifications

The functional specifications only address the definition of management operations on meetings day-by-day; the information related to a meeting is composed of a given topic, a group of people attending, a venue and time/date information. Any person attending is defined by several identification items. The information used for the

management of meetings is stored in each of a set of *meeting* descriptors and can be summarised as follows:

- *topic:* topic to be discussed during the meeting;
- *venue/time/date:* place where the meeting is held and time/date information;
- *dynamic person list:* list of persons attending the meeting.

These descriptors are the main leaves of a tree (a sub-tree) of the E-Diary which is considered as being an object which is private to a given user (the E-Diary is not shared by multiple users). Each *person* in the list is defined by several information items such as *name/firstname*, full *address*, and *phone_number*. Some periods like *days*, *weeks* or *months* may be locked for a given *reason* (travel abroad or any personal reason for instance). The E-Diary also includes a *note-pad* where *messages* may be stored on a day-by-day basis. The E-Diary provides functions to insert, list or remove any of the above defined objects. The italic words indicate most of the objects used in the design of E-Diary application.

7.2 Confidentiality constraints

The description of the example given in Section 7.1. can be augmented with an informal description of confidentiality constraints. These were chosen to be the following:

1. Any two or more of items in a given *meeting* such as *topic, time/date, venue*, *person_list* considered as constituting confidential information.
2. Personal identification items such as *name, address* and *phone number* can be individually considered as being public information; but any pair of such information items including person *name* is confidential.
3. The group of *persons* attending the same *meeting* is considered as constituting a confidential information item.
4. Any unstructured information items such as *topic* of a meeting, *message* in the note pad, and *locking reason* for a *day*, *week* or *month* is confidential.

The interpretation we have made of this informal description of the confidentiality constraints leads to the following formal description:

Confidentiality clauses		**Unstructured confidential objects**
1) person ==	{name ∧ (address / phone number)}	{topic, message, locking_reasons}
2) meeting ==	{venue ∧ topic ∧ time/date ∧ [person]*, venue ∧ time/date ∧ [person]*, time/date ∧ [person]*, person ∧ [person]*}	

These constraints have to be taken into account in order to refine the first design and to identify fragments. They are also used for scattering.

7.3 Final object-oriented design

Several design steps were performed to obtain the final design of the *E-Diary* objects and to identify fragments in the design. In the first design the *meeting* object was not decomposed into sub-objects as candidate fragments. The list of *persons* attending a *meeting* also did not appear . Since *meetings* and *persons* are confidential objects (see clauses 1 and 2) some decomposition into more elementary objects was performed such as represented in Fig. 6. Some of the object classes (and their component objects) forming the E-Diary application object are shown, where an asterisk indicates the possibility of there being several components of a given object class.

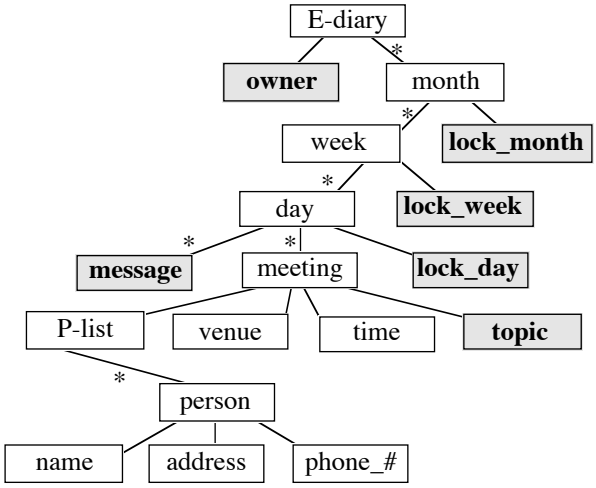

Fig. 6: The E-Diary object composition hierarchy (final version)

The object hierarchies presented in Fig. 6 illustrate the various components in the design of the E-Diary object down to elementary objects, the latter being a combination of elementary objects such as integers, booleans, strings, etc. Some of the elementary objects represented by grey boxes are confidential leaves of the tree that according to our assumptions cannot be usefully decomposed into smaller objects; for instance owner, messages, locking reason and meeting topic are strings that are assumed to be ciphered to ensure confidentiality as soon as they are entered by the user in the system.

Pre-defined confidentiality constraints lead to separating as fragments objects that will be managed by separate instance managers in the implementation. *Topic*, *venue* and *time*/*date* objects are assumed to be object-fragments. The *P-list* object may still be kept in the *meeting* objects since it contains only pointers (references) to *persons* managed by an instance manager of class *person* in the implementation. *Person* is thus another object-fragment. As a consequence, the *meeting* object is then relatively empty since *meeting* sub-objects are scattered in separate fragments.

8 Conclusions and future work

The electronic diary system is the first sizeable experiment we have undertaken in implementing an application using Object-Oriented Fragmentation-Redundancy-Scattering techniques. As such the experiment has greatly assisted us in formulating a methodical approach to the use of the techniques, and helped to motivate the development of the scheme for expressing confidentiality constraints that we have described in Section 4. More complex processing could be added to actual object-fragments even in this simple example without introducing any confidentiality problem.

The granularity of objects-fragments obtained in the example to solve the confidentiality problem might appear relatively small. However, this technique can also be used to solve some problems using a very coarse 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, there is no need to go further in the fragmentation process as soon as the link between these two large fragments (some references) is retained at the trusted site. Access to one or both parts of the information (if necessary) then needs appropriate user authentication (medical or administrative staff) to properly grant related authorization.

The performance of FRS mainly depends on the granularity of the fragmentation. Nevertheless, FRS need not introduce any significant information and processing overhead (reassembly is negligible); it obviously introduces communication overhead with respect to a pure processing replication, e.g., in an application that does not attempt to tolerate intentional faults. Although parallelism is not the aim of our fragmentation process, the additional opportunities it provides for the use of parallelism can be of significant benefit with regard to application performance in suitable circumstances. In particular they could reduce the impact of such communication overheads.

From a programming viewpoint, given the awkwardness of the manual translation involved in the final stages of implementation down onto the Delta-4 platform, more extensive trials of further applications will probably best await the provision of means for automatically installing applications onto a suitable object-oriented distributed runtime layer. We are at present just starting to investigate the suitability for this purpose of COOL [14], which runs on the Chorus micro-kernel operating system [15], in the hope that this will provide us with a good basis for using FRS in connection with C++. Other topics on which more work is needed include naming facilities for reference management, algorithms to compute references, and access control mechanisms for fine grain object invocation. By such work we hope to develop the object-oriented FRS scheme to the point where experiments can enable realistic cost/effectiveness assessment of the scheme on a variety of applications. However in parallel we also plan to continue recent closely-related work on object-oriented language concepts, not just inheritance but also in particular delegation and reflection [11], which we believe will facilitate the structuring and implementation of applications using various dependability-related mechanisms in combination, including of course FRS. The OpenC++ language [13] is currently our favoured candidate for experimenting reflection in the implementation of FRS application.

9 References

1. J.C. Laprie, Ed., Dependability: Basic Concepts and Terminology (in English, French, German, Italian and Japanese), series Dependable Computing and Fault-Tolerant Systems, (A. Avizienis, H. Kopetz, J.C. Laprie Eds.), Vol.5, Springer-Verlag, 1992, 265 p., ISBN 3-211-82296-8.

2. Y. Deswarte, L. Blain and J.-C. Fabre, "Intrusion Tolerance in Distributed Computing Systems", in *Proc. IEEE Symp. on Security and Privacy*, Oakland California (USA), 1991, pp. 110-121.

3. N. Ahituv, Y. Lapid, S. Neumann, "Processing Encrypted Data", in Comm. of the ACM, vol. 30, #9, Sept 1987, pp. 777-780.

4. R.L. Rivest, L. Adelman, M.L. Dertouzos, "On Data Bank and Privacy Homomorphisms", in *Foundations of Secure Computation*, Academic Press, ISBN o-12-210350-5, pp. 169-179.

5. G. Trouessin, J.C. Fabre and Y. Deswarte, "Reliable Processing of Confidential Information", *Proc. of the 7th IFIP/Sec'91*, Brighton (UK), 1991, pp. 210-221.

6. A. Shamir, "How to Share a Secret", *CACM*, vol. 22, #11, pp.612-613, 1979.

7. M.O. Rabin, "Efficient Dispersion of Information for Security, Load Balancing and Fault-Tolerance", *Journal of ACM*, vol. 36, #2, April 1986, pp. 335-348.

8. D. Powell, Ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing*, series Research Reports ESPRIT, Project 818/2252, Delta-4, Vol. 1 of 1, Springer-Verlag, 1991, 484 p., ISBN 3-540-54985-4.

9. S.K. Shrivastava, G.N. Dixon and G.D.Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, vol. 8, #1, 1991, pp.66-73.

10. J.C. Fabre and B. Randell, "An Object-Oriented View of Fragmented Data Processing for Fault and Intrusion Tolerance in Distributed Systems", in Proc. of *ESORICS 92, LNCS n° 648,* Springer-Verlag, Nov. 1992, pp. 193-208.

11. R. Stroud, "Transparency and Reflection in Distributed Systems", in *Proc. of the 5th. ACM SIGOPS European Workshop on Distributed Systems*, Le Mont Saint-Michel, France, Sep. 1992, 5 pages.

12. S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with Meta-Level Architecture", *Proceedings of the ECOOP '93*, *LNCS n°707*, Springer-Verlag, July 1993, pp. 483-502.

13. NCSC TNI, "Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria", *Tech. Rept. NCSC-TG-005*, NCSC, 31 July 1987.

14. R. Lea, P. Amaral, C. Jacquemot, "COOL-2 : an Object-Oriented support platform built above the CHORUS Micro-Kernel", in *Proc. of the IEEE I-WOOOS'91*, Palo Alto, CA (USA), October 1991, pp. 68-73.

15. M. Rozier et al., "Overview of the Chorus Distributed Operating System", *Chorus Systèmes Technical Report*, *CS-TR-90-25*, 1990, 45 pages.