COMPUTING SCIENCE

Designing Secure and Reliable Applications using FRS: an Object-Oriented Approach

J-C Fabre*, Y. Deswarte* and B. Randell**

TECHNICAL REPORT SERIES

No. 438 July, 1993
Designing Secure and Reliable Applications using FRS: an Object-Oriented Approach

J-C Fabre*, Y. Deswarte* and B. Randell**

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 network of a large number of workstations. This technique has been applied to security management, to file storage and more recently to the processing of confidential information, so as to achieve a high degree of security as well as reliability.

The main objective of this paper is an object-oriented approach to the design of FRS applications in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. 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). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. Replicas of non-confidential objects are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.
Bibliographical details

FABRE, Jean-Charles

Designing Secure and Reliable Applications using FRS: an Object-Oriented Approach
[By] J-C Fabre, Y. Deswarte and B. Randell


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 438)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

DESWARTE, Yves

RANDELL, Brian

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 network of a large number of workstations. This technique has been applied to security management, to file storage and more recently to the processing of confidential information, so as to achieve a high degree of security as well as reliability.

The main objective of this paper is an object-oriented approach to the design of FRS applications in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. 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). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. Replicas of non-confidential objects are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.

About the author

Jean-Charles Fabre is a Charge de Recherche INRIA, working on dependability in distributed systems, in particular on reliable processing of confidential information.

Yves Deswarte is currently "Directeur de Recherche INRIA". Formerly R&D engineer in a main French computer manufacturing company, he joined INRIA and LAAS in 1979, as a member of the Directorate team of the French National Action for Systems Dependability.

Professor Randell has been a professor of computing science in the Computing Science Department of the University of Newcastle upon Tyne since 1969.
Suggested keywords

FAULT-TOLERANT SYSTEMS
INTRUSION TOLERANCE
SECURITY
OBJECT-ORIENTED MODEL

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 658.47 001.6425
U.D.C. 519.687 519.718 681.322.06
Designing Secure and Reliable Applications using FRS: 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, UK

Keywords: fault-tolerant systems, intrusion tolerance, security, object-oriented model

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 network of a large number of workstations. This technique has been applied to security management, to file storage and more recently to the processing of confidential information, so as to achieve a high degree of security as well as reliability.

The main objective of this paper is to describe an object-oriented approach to the design of FRS applications in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. 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). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. Replicas of non-confidential objects are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.

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

Distributed systems are usually considered as a natural support for fault-tolerance: by means of distribution, most physical faults can be confined to the faulty units so as to prevent error propagation throughout the system. This requires either efficient error detection mechanisms and sufficient data redundancy enabling the global distributed processing to be recovered, or a higher data and processing redundancy to mask the errors: error detection-and-recovery and error masking are the two usual techniques of fault tolerance [Laprie 92].

Most fault-tolerant distributed systems are designed to cope with just a limited class of faults: namely 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 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\(^1\). In distributed systems composed of the individual's workstation 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 has been penetrated by an intruder. On the other hand, server administrators and other 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 on the other workstations, irrespective of the cause of the failure, be it an accidental physical fault or an intrusion.

---

\(^1\) 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).
Because they do not take intrusions into account classical fault tolerance techniques, such as data and processing replication, can help to tolerate accidental faults, but 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 is why a technique has been developed at LAAS to tolerate faults while preserving confidentiality, namely the fragmentation-redundancy-scattering (FRS) technique [Deswarte 91]. 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 [Deswarte 91]. The aim of this paper is to show how FRS can be used in the design and in the implementation of any application or system service so as to achieve reliable and secure storage and processing of confidential information. In this paper we discuss how the FRS technique can be extended and generalised and its use facilitated by associating it with the use of an object-oriented approach to system design.

2. Distributed system architecture and assumptions

The distributed system architecture (cf. figure 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 responsible for running fault-tolerant secure servers. A user of a trusted workstation is responsible for the security of his 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.)

Figure 1: Distributed system architecture

In this paper we assume the provision of two types of services already implemented using untrusted sites, namely the provision of storage and authentication/authorisation. The use of conventional FRS has been successfully demonstrated - see [Deswarte 91]. A user is first authenticated by a majority of security sites and receives some authentication information required to request access to storage and processing services.

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. Passive and active malicious actions are considered such as ones which involve attempting to obtain and/or modify static and/or dynamic information at any untrusted site (including memory segments, temporary files, system files, operating system kernel).
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 is 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 [Trouessin 91]. 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. This problem provides much of the first motivation for the use object-oriented techniques described in this paper.
3.2. Object view of FRS

The object model used here is not specific to any 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 are identified by references. The use of inheritance is not discussed very much in this paper (see [Fabre 92]). Nevertheless, solutions using inheritance for programming an FRS application are mentioned and other related aspects are discussed briefly at the end of this paper.

The main interest of the object model is that it allows the designer to decompose an application as a collection of related objects in such a way that most of the individual objects do not process confidential information. In fact, this model relates to both aspects of the use of FRS: the fragmentation of the confidential information being processed and the operations (the methods) performed on it. The design approach which is proposed in this paper 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 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 [Deswarte 91], 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 [shamir 79]. 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 [Rabin 89]. 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: 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

\(^2\) 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 greater than, or equal to T in order to tolerate faults and intrusions.

-7-
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 formulas. 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

\[(\text{meeting} \equiv \text{topic} \land \text{time/date} \land \text{venue} \land \text{person_list})\]

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: \[(\text{meeting} \equiv (\text{topic} \lor \text{time/date} \lor \text{venue}) \land \text{person_list});\] 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 \[(\text{person_list} \equiv \text{person} \land \text{person}^*)\] 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:

\[
\text{Unstructured confidential objects} \equiv \{<\text{object}> [, <\text{object}>] \}^*
\]

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 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 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. 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 (see figure 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
```

Figure 2: Fragmentation principle

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 [Powell 91]. 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 [Shrivastava 91] where for example any object can be declared as recoverable:

\[
\text{class } \textit{<reliable_object>} :: \textit{recoverable};
\]

("::" here stands for "inherits").

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. Provided the runtime system is able to offer such basic system services, this approach is a very elegant solution for programming reliable applications using object-oriented programming languages. The use of inheritance to declare \textit{secured} objects is therefore one of our current fields of investigation [Fabre 92].

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 7.

```
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
```

Figure 3: Scattering principle

The scattering phase is summarised in figure 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.

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, all (some) sub-objects must be created remotely. The create operation must be able to do this, i.e., by creating multiple copies of sub-objects and scattering them among untrusted sites. Access to such internal operations of the language runtime 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. In a reflective language, a meta-class can be associated to one object; this meta-class allows the designer to re-define operations such as creation and invocation. The re-definition of the default behaviour of such operations in the language itself is a promising approach to implementing non functional characteristics using object-oriented languages [Stroud 92].

5.4. Summary

The complete design process can be summarised in the several tasks that are represented in figure 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.
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 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 by means of an Interface Definition Language (IDL) used for remote method invocation.

---

\(^3\) DELTASE : DELTA-4 Application Support Environment
between object manager replicas. The set of servers provides an object management layer on top of the distributed runtime layer (figure 5).

<table>
<thead>
<tr>
<th>Object-based application design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object runtime layer</td>
</tr>
<tr>
<td>Distributed runtime system (DELTASE)</td>
</tr>
<tr>
<td>Local executive (Unix)</td>
</tr>
</tbody>
</table>

![Figure 5: Runtime system structure](image)

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 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.

The other approach, which involves creating a unique server for any object instance is not appropriate on such a system architecture due to the performance overheads related to the creation of a runtime server in the system. This approach requires a more appropriate object-oriented layer in which dynamic object instance creation can be done more efficiently. The Chorus Object Oriented Layer (COOL) developed on top of the Chorus micro-kernel is one good example [Lea 91]. This last solution is now investigated on Chorus.

6.2. User authentication and authorisation

The authentication scheme we envisage is one in which any user is authenticated with respect to a majority of security sites using a smart card system and appropriate agreement protocols [Deswarte 91]. Any time a user wants to access a service, he requests authorisation to the security server and obtains a key used to manage fragments. The key is reconstructed locally on the user site using a threshold scheme [Shamir 79] after having collected several shadows coming from several security sites.
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 authorisation protocol.

Looking more carefully at a fragmented application (cf. figure 6), 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.

![Diagram of a fragmented application with local and external references](image)

Figure 6: Structure of a fragmented application - Local and External references

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:

\[\text{reference} = E_K (\text{object\_name, application\_name, object\_permissions})\]

\(^4\) A reference is viewed here as a generalisation of the notion of pointer in a distributed environment.
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.

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 [NCSC 87]). 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.

6.4. Aggregation of sub-objects

The systematic fragmentation approach described in section 5.1. can lead to the creation of an unnecessarily large number of object-fragments. The objective of the aggregation process is to gather objects and sub-objects at different abstraction layer without breaking any confidentiality constraint; the number of final fragments is reduced and the resulting fragments have a coarse granularity. In the example used in previous sections, object meeting is confidential because $\text{topic} \land \text{time/date} \land \text{venue} \land \text{person_list}$ reveals confidential information to a potential intruder. Systematic fragmentation creates four fragments. If we just consider the constraint $\text{topic} \land \text{person_list}$, then just three fragments need to be created: meeting object-fragment (which contains time/date and venue information), topic object-fragment, and person_list object-fragment. If topic is implemented by a cipher string, then only two fragments are necessary: meeting object-fragment (which contains ciphered topic, time/date and venue information), and person_list object-fragment. (Of course, such aggregation of sub-objects can only be done between objects that have a common ancestor.)

6.5. Human-system interface

Finally, the human-system interface has to be taken into account in the design and in the implementation of an application by FRS. Any human-system interaction that manipulates confidential information must be performed by a set of objects located at the user site; input or output of any confidential information by the application is done in clear and (if necessary) such information is ciphered before being sent to object-fragments on untrusted sites. Outside the user
site such information must either be ciphered or in a fragmented form. In the experimentation described in section 8, the operator dialogue was implemented by a set of appropriate I/O objects, and made use of X Window System facilities.

7. Experimentation

We have investigated the above FRS design approach on a detailed example, a distributed Electronic-Diary5, which has been implemented on the DELTA-4 system. This diary system was first designed using Eiffel [Meyer 87] tools and then implemented using the distributed facilities of the DELTA-4 Support Environment (DELTASE). A first prototype is currently running on a set of Unix workstations. We describe here this application using a small series of classes, so leading to a hierarchical design of the E-Diary.

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. All these functions are available through a friendly operator dialogue on an X-window terminal. The italic words indicate most of the objects used in the design of E-Diary application.

5 The objective is not here to design a complete E-Diary with all the functionalities a real user would expect from such a tool. Only a subset is provided to illustrate the object-oriented approach of FRS.
7.2 Initial object-oriented design

The first design is based on the functional description given in the previous section. The following figure (figure 7) shows the composition hierarchy of objects and related classes which is the basis for fragmentation.

This object-oriented design presents the management of meeting information; the basic items forming a meeting descriptor and the person list are also taken into account but not shown in Figure 7. 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.

![Diagram](image)

Figure. 7 The E-Diary object composition hierarchy (first version)

The object hierarchy represented in Figure 7 for the E-Diary service is as follows: the E-Diary is composed of several month objects and is owned by a given user (owner). Each month is composed of a number of weeks and can be locked (lock_month) for a reason, which can be given. Each week is composed of a number of days and can also be locked (lock_week), with a given reason. Any day is composed of a list of meetings, a list of messages (note pad) and can be also locked, with a given reason (lock_day). Any lock set to true implies that no meeting can be allocated in the month, week or day, respectively. The E-Diary is considered as a persistent object and can thus be activated (from persistent storage) after being created. It offers several services to the owner: create, modify, move, delete a meeting, put, release a message in the note pad of a given day, and lock a month, a week, or a day with a given reason.
Note that the above makes no mention of inheritance, and hence of whether a public method's implementation is given in the class definition of the particular object, or is inherited from some other class definition.

The object which is of interest in the above hierarchy is the meeting object which contains confidential information. A meeting is composed of a persons list (P-list), a venue, time and topic. In the first design the P-list was implemented using the Eiffel pre-defined class list (of persons). Person is composed of three basic attributes (in Eiffel terminology) in our example (name, address and phone_number.) At this stage of the design all the information related to a given meeting is located in only one meeting object.

7.3 Confidentiality constraints

The description of the example given in section 7.1. can be augmented with an informal description of confidentiality constraints. These are 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:

<table>
<thead>
<tr>
<th>Confidentiality clauses</th>
<th>Unstructured confidential objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) person == {name \ (address v phone number)}</td>
<td>{topic, message, locking_reasons}</td>
</tr>
<tr>
<td>2) meeting == {venue \ topic \ time/date \ [person]^<em>, venue \ time/date \ [person]^</em>, time/date \ [person]^<em>, person \ [person]^</em>}</td>
<td></td>
</tr>
</tbody>
</table>

-19.
These constraints have to be taken into account in order to refine the first design and to identify fragments. They are also used for the aggregation phase and then for scattering.

7.4 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 figure 8.

![Diagram](image)

Figure 8: The E-Diary object composition hierarchy (final version)

The object hierarchies presented in figures 7 and 8 (in a form similar to Eiffel browser output) illustrate the various components in the design of the E-Diary object down to elementary objects, the latter being a combination of Eiffel elementary objects such as integers, booleans, strings... Some of the elementary objects represented by grey boxes are confidential leaves of the
tree that it is assumed for our purposes cannot be usefully decomposed into smaller objects; for instance owner, messages, 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. The same is true for lock objects which correspond to a boolean value and a string that indicates the locking reason.

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; a meeting object can then be also considered as a fragment. The analysis of the constraints and the aggregation phase thus leads to the final object-fragments definition shown in figure 9.

![Diagram of E_Diary and related objects](image)

**Figure 9: Object-fragments of the E-Diary**

This final design solves the pre-defined confidentiality constraints using the smallest number of fragments, and is that which has been implemented on the DELTA-4 system, except that messages and lock reason have been managed in separate object-fragments. The E_Diary object implements multiple instances of month, day and week strategically using standard types of the C programming language. (It has been convenient to do an initial design using Eiffel, even that Eiffel is not supported by DELTASE).
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, though some quite significant prior trials had been made at LAAS of the original FRS scheme, such as in [Trouessin 91]. As such the experiment has greatly assisted us in formulating a methodical approach to the use of the techniques (summarised in Figure 4), and helped to motivate the development of the scheme for expressing confidentiality constraints that we have described in Section 4.

The granularity of objects-fragments obtained in the example to solve the confidentiality problem might appear relatively small. Nevertheless, this technique can 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 authorisation.

The performance of FRS mainly depends on the granularity of the fragmentation. Nevertheless, FRS does not introduce any information and processing overhead (reassemble 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 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 [Lea 91] [Amaral 92], which runs on the Chorus micro-kernel operating system [Chorus 90], 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 [Stroud 92], which we believe will facilitate the structuring and implementation of applications using various dependability-related mechanisms in combination, including of course FRS.

9. References


