Design and Implementation of the FRIENDS System

Jean-Charles Fabre

LAAS-CNRS and INRIA
7, Avenue du Colonel Roche, 31077 Toulouse cedex - France
Tel. +33/ (0)5 61 33 62 36 – Fax. +33/ (0)5 61 33 64 11 – E-mail. fabre@laas.fr

Abstract. The paper describes a metaobject architecture for distributed fault tolerant systems. Basically metaobject protocols enables functional objects to be independent from meta-functional properties implemented by metaobjects. Metaobjects can thus be specialised for fault tolerance, security, distribution and used on a case-by-case basic within application. The runtime support for metaobjects must include basic common services required in distributed fault tolerant computing (i.e. atomic multicast protocols and group management facilities, detection mechanisms). Off-the-shelf microkernels correspond to the very basic layer of the system. Architectural issues, application issues, development issues, experimental and performance issues are presented. Some implementation details and properties (ease of use, reusability, configurability, etc., namely flexibility) of our system are also discussed. Two prototypes have been developed today, the last one being based on the Chorus microkernel.

1. Introduction

The objective of the *FRIENDS* system described in this paper was to provide to programmers a metalevel architecture for implementing distributed fault tolerant applications. This system architecture was early described in [1, 2]; more details on implementation issues are given here.

The main property of this system is flexibility which is provided at various levels. Flexibility is provided first at the language level thanks to the use of reflective object-oriented programming languages [3] and also at the operating system level thanks to the use of microkernel technology. More precisely, a metaobject protocol [4] is used for adding in a transparent way non-functional mechanisms to applications, including fault tolerance and distributed features. These mechanisms are called non-functional with respect to functional specifications implemented by application objects [5]. A metaobject is connected to the application object it controls: a metaobject is an object which able to adjust the structure and the behaviour of an application objects. Applications objects are located at the so-called base level, the set of metaobjects forming the so-called metalevel. The information about objects which is manipulated at the metalevel is called meta-information. Object features that can be controlled at the meta-level are called reified features. Any change in this meta-information is reflected at the base level. In class based languages, the metaobject protocol can be viewed as a class providing control means to object creation, object deletion, object invocation at least. The interaction between an application object and its related

This work was partially supported by the *DEVA* Esprit project n°20072.

metaobject is thus driven by this (root) class. The default implementation of this class consists in performing the standard behaviour (standard object creation, standard object deletion, standard object invocation). Variants of this default implementation can be defined (derived in object oriented languages) from this root class. For instance, new metaobject classes can implement non-functional mechanisms such as remote interaction, replicated invocation to a group of objects, authenticated interaction by means of electronic signatures, etc. From a practical viewpoint (at least with the MOP used in *FRIENDS*) a metaobject is strongly connected to the object it controls and shared the same address space at the user level. Nevertheless, depending on the non-functional mechanism which is considered and for efficiency, all needed mechanisms cannot be implemented at the user level. This is why companion subsystems interacting in a tightly manner with low level services and the hardware are mandatory. For, instance distributed fault tolerance involves detection of machine crashes and group communications which relies on a low level high priority protocol. In our system metaobjects rely on a companion sub-system offering generic and basic services. A system configuration is then composed of several sub-systems (or personalities) which can be tuned up according to the needs. The overall architecture of the *FRIENDS* system is described in section 2.

The advantage of this approach is that metaobjects can be developed using object-oriented techniques and development methods. In the present version of our system, a hierarchy of metaobject classes providing various local and distributed fault tolerance mechanisms has been defined and the corresponding mechanisms implemented. From these hierarchy, several others can be more easily developed (see section 3).

In order to evaluate the properties of our system a distributed application has been developed, first of all in a centralised manner. Then metaobjects handling remote interaction have been used. Then, fault tolerance features have been added to the distributed version by using fault tolerance metaobjects. This part of the work is reported in section 4. Two prototypes have been developed today. Implementation and performance issues are reported in section 5 . Nevertheless, our system suffers from several drawbacks which are not related to the approach itself but more to the metaobject protocol and the language used in the implementation. A summary of the limits and drawbacks together with some solutions is given in conclusion.

The use of a metalevel system architecture for dependable systems was early described in [6]. The contribution of this work is two-fold: first, the definition of a suitable architecture for a flexible implementation of dependable systems, and second a sizeable experiment in order to judge the advantages and drawbacks of this approach.

2. Architecture of the *FRIENDS* system

2.1. Overview

The architecture of the system is composed of several layers: (i) the kernel layer which can be either a Unix kernel or better a microkernel, such as Chorus [7], (ii) the system layer composed of several dedicated sub-systems, and finally (iii) the user layer dedicated to the implementation of applications and mechanisms as metaobjects. A simplified static view of the overall system architecture is given in Fig. 1. The

implementation environment provided by *FRIENDS* corresponds to the whole set of sub-systems and libraries of metaobject classes.

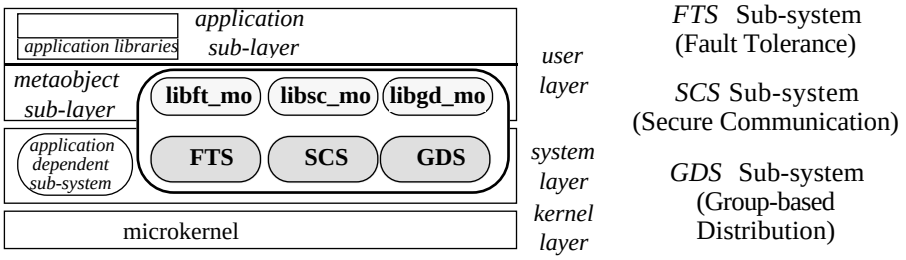

Fig. 1. Overall system architecture

The implementation of any type of mechanism (fault tolerance, secure communication, distribution) is thus spanning partially the user and system layers.

2.1.1 System Layer

The *system layer* is organised as a set of sub-systems. In *FRIENDS* three sub-system provide services for fault tolerance, secure communication and group-based distribution. Any sub-system may be hardware- and software-implemented.

FTS provides basic services mandatory for fault-tolerant metaobjects, in particular error detection and failure suspectors (as in [8]) which must be implemented as low level entities. Failure detection and configuration management services in *FTS* are shared with the communication sub-system. The information handled by these services reflect the current configuration of alive nodes in the distributed system. Another fundamental service, also located into *FTS*, is the replication domains management service: it provides information describing static and dynamic information related to the available nodes and provides facilities to allocate a node to a newly created object. The link between application services and the required number of object copies is handled by this service. Several facilities implemented here require the use of a file system provided by a Posix companion sub-system. The latter is also used to implement a stable storage support in our prototypes.

SCS provides basic services that must obviously be implemented as trusted entities within the system (notion of Trusted Computing Base). These services should include in particular an authentication server, but also an authorisation server, a directory server, an audit server. The simple authentication server implemented in *SCS* provides to authenticated users (Unix login) session keys for later authentication and ciphering of remote messages. A more sophisticated authentication server providing secure user authentication based on smartcards was also developed but it is not yet integrated into *SCS*.

GDS provides basic services implementing a distribution support for object-oriented applications where objects can be replicated. These basic services include group management facilities and atomic multicast protocols (xAMp [9]). Any new object is created by a factory service and is mapped onto a runtime unit, depending on entities handled by the underlying operating system (Unix processes or Chorus actors). Any

object is inserted into a group; a group name (as a string) is given to any application service. Any modification of the list of members (new view) is signalled to the user level and activates a user-defined handler which is responsible for taking appropriate recovery actions. The group communication service provides several group communication semantics. In our case, multiple messages sent to a group involve all correct receivers to get messages in the same order (`Atomic_Send` primitive). The very basic communication protocols also provide host failure detection (crash); this information is used in *FTS* to maintain a consistent view of the current configuration of the system and replication domains. Such event is also signalled to the user level and activates user-defined handlers.

Every basic services required by the mechanisms implemented as metaobjects can be seen as *Software Replaceable Units* (*SRU*). The system layer can easily be composed of the required sub-systems, each of them using the appropriate SRUs.

2.1.2 User Layer

The user layer is divided into two sub-layers, the application layer and the metaobject layer controlling the behaviour of application objects. Some libraries of metaobject classes for the implementation of fault-tolerant and secure distributed applications are implemented on top of the corresponding sub-system and provide the user with mechanisms that can be adjusted, using object-oriented techniques.

The *libft_mo* library provides metaobject classes for various fault tolerance strategies (based on stable storage or replication) with respect to physical faults considering fail-silent nodes. According to the strategy used, these metaobjects provide appropriate recovery and reconfiguration actions. In normal operation, their role is to save state information either to stable storage or to send it to a group of spare replicas. They also define failure/view handlers which activate recovery actions defined as object methods. When a server object crashes, the recovery action consists in creating a new object within the replication domain. The object creation is done by the *GDS* factory service according to the information handled by *FTS* about the present configuration of the system and the current status of replication domains.

The *libsc_mo* library provides metaobject classes for various secure communication protocols using ciphering techniques, signature computation and verification based on secret or public key cryptosystems. These metaobjects, when used, add security properties to basic group communications. Invocations are handled as raw data messages and cryptographic signatures are computed when a message is sent and verified when a message is received. They also interact with the authentication server to obtain a session key when an object is created and propagate the key to the group of server objects. All objects in the same group use the same session key. Session keys have a pre-defined lifetime and thus new session keys are generated during operation. The detection of corrupted signature several times lead also to key renewal.

The *libgd_mo* library provides metaobject classes for handling remote object interaction, which can be implemented with groups. The combination of these metaobjects and *GDS* provides a runtime support for distributed object-oriented applications. The metaobjects here fill the gap between application objects written in C++ and the operating system computational model by mapping user defined objects

as active OS objects. They are also responsible for handling the link between individual objects and group, forwarding messages to peers, handling message queues. The latter takes into account the object status according to the fault tolerance strategy used in fault tolerance metaobjects. The message delivery policy to fault tolerance metaobjects depends on the status "active or passive" of the replica. Messages received may remain undelivered depending on this status. We also assume that application objects have a "deterministic behaviour". Any method invocation with identical input parameters produces the same results on any of the object replicas. Concurrency and other sources of non determinism within objects have not been considered yet.

2.2. The Open C++ MOP

The reflective language used for programming *FRIENDS* is Open C++ [10]. This language is an extension of C++ which provides the user with two levels of programming, the base-level dedicated to the implementation of application objects and the meta-level dedicated to the implementation of metaobjects. Objects and metaobjects are written in C++, Open C++ V1 [11] providing special declaration statements to associate an object with a metaobject (one-to-one mapping). This declaration "`//MOP reflect Class_C : MetaObject_Class_M`" should be understood as "*any object of Class_C is controlled by a metaobject of class MetaObject_Class_M*".

```
class MetaObj {

// reified features
public:
   void Meta_StartUp ();        // instance creation
   void Meta_Cleanup ();        // instance deletion
   void Meta_MethodCall (...); // method invocation
   void Meta_Read (...);        // attribute read access
   void Meta_Assign (...);      // attribute write access

// default implementations
protected:
   void Meta_HandleMethodCall (...);
   void Meta_HandleRead (...);
   void Meta_HandleAssign (...);
};
```

Fig. 2. The Open C++ Metaobject Protocol

Open C++ V1 is a pre-processor of C++ and thus the capabilities of the MOP are very limited and the meta-information is very poor. Anyway, this metaobject protocol was sufficient to define a metalevel architecture for fault tolerant systems and to make extensive experiments. In Open C++ V1, the metaobject protocol is defined by the class `MetaObj` (Fig. 2).

Methods `Meta_StartUp` and `Meta_CleanUp` are called respectively after creation and before deletion of the base-level object; between creation and deletion, object and metaobject can refer to each other. `Meta_MethodCall` is called when a base-level method is invoked: `m_id` identifies the method, `args` packs its input arguments and `reply` is supposed to pack the results when `Meta_MethodCall` returns. `Meta_Read` is

called when an attribute identified by `var_id` is read and `value` is supposed to contain the result of the read access. `Meta_Assign` is called when an attribute identified by `var_id` is written and `value` is the value that should be assigned. Private methods implement the default behaviour of the language: `Meta_HandleMethodCall`, `Meta_HandleRead`, `Meta_HandleAssign` enable the metalevel to invoke a base-level method or access (read, write) a base-level attribute, respectively. Finally, `ArgPac` is a stack-like class that may contain all types of objects (including `ArgPac` objects). Fig. 3 illustrates invocation trapping and how it can be adjusted with this MOP and also how state information can be captured.

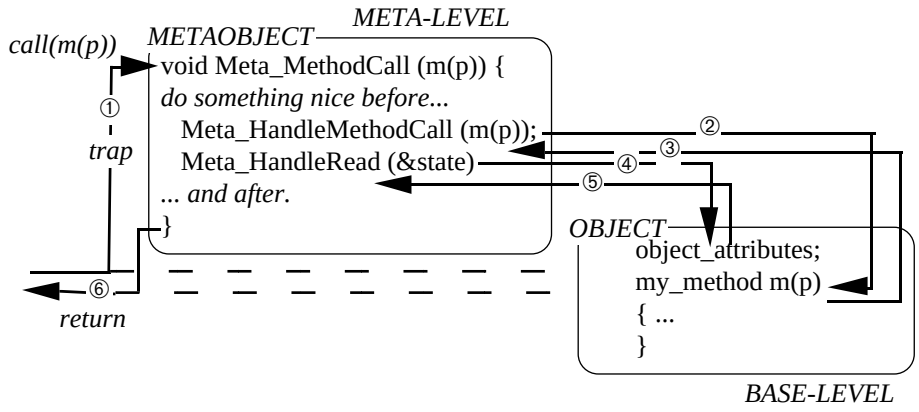

Fig. 3. Invocation and state access

Classes that are not bound to a metaobject class have the default class behaviour. This MOP is a simplified version of the Open C++ MOP. Nevertheless, any reflective object-oriented language providing this MOP could be used instead.

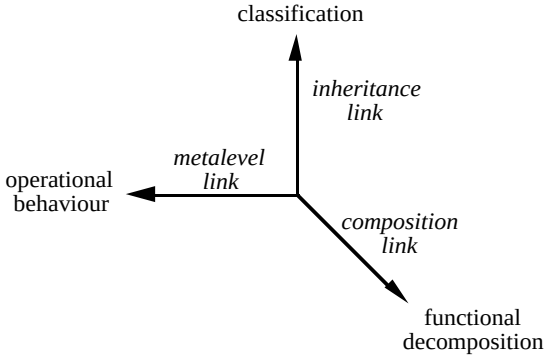

Fig. 4. Indirections in reflective OOPLs

In summary, a metaobject protocol provides a new kind of indirection to the programmer familiar with object oriented languages. It is in fact a new link that is able to connect the object to some other object controlling part of its internal behaviour. The three links are shown on Fig. 4. Whereas the inheritance link and the composition link relate to design the other link relate to operation. The metalevel link enables the object to delegate part of its internal behaviour to a third party, the

metaobject. In other words, the object is able to ask for additional actions which are automatically synchronised with the standard internal object operation. A simple example of actions performed by metaobjects can be to trace method calls transparently, for instance.

2.3. Application model

An application is regarded as a collection of communicating active single threaded objects developed using an object-oriented programming language (currently C++). Any application object is mapped by *GDS* on a runtime object. Each runtime object is not only composed of an application object, (denoted A) but also contains one or several metaobjects within the same address space. The set of metaobjects depends on the properties that must be provided to the application and includes at least one metaobject, `GD`, for distributed interaction using group-based communications. Ideally, adding properties to an application involves adding other metaobjects to the set. This set may include a fault tolerance metaobject (denoted `FT`) and a metaobject for secure communications (denoted `SC`). When all properties are required a runtime object is thus composed of the following sequence of metaobjects `{A, FT, SC, GD}` .

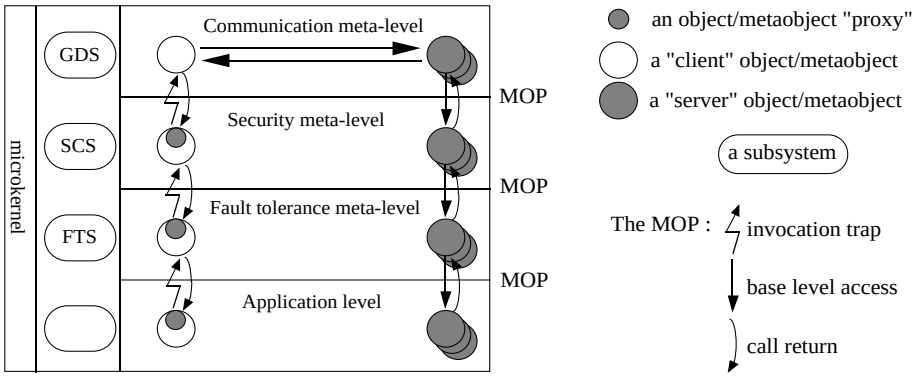

Fig. 5. Operational view of the *FRIENDS* system

The notion of *metaobjects set* is similar to the notion of *metaspace* defined in Apertos [12] and also to the notion of *reflective object tower* in ABCL/R2 [13]. Within one runtime object, the interaction between the application object and the metaobjects is done through the MOP. The invocation is thus processed in sequence by metaobjects, first FT, then SC and lastly GD. The interaction of runtime objects is based on the client-server model. More precisely it is based on the proxy model: a server object is perceived as a proxy within the client address space. The proxy server is attached to a metaobject handling the client side, the effective server is attached to a metaobject handling the server side. Any type of mechanism is thus performed by a protocol between two metaobjects. The application programmers just writes the application objects and selects the set of appropriate metaobjects. Fig. 5 shows the client-(replicated)server interaction with the MOP.

The use of metalevel programming is not restricted to application programmers. Also fault tolerance, security programmers can take advantage of this programming approach. This is why the repeated use of metaobjects at various programming levels

leads to a recursive use of metaobjects in applications as shown in Fig. 5. As far as fault tolerance mechanisms are concerned, the interaction between a primary object and a backup can also be transparently performed using metaobjects. All implemented distributed fault tolerance mechanisms have been implemented with this approach; the remote interaction between replicas (inter-replica protocol) is performed by means of communication metaobjects. More details about this approach can be found in [2].

3. Programming on *FRIENDS*

3.1. Application example

Several fault tolerance metaobjects have been developed today (see section 4) either based on stable storage (denoted STS), passive replication (primary-backup model) and active replication (leader-follower model) strategies (denoted PBR and LFR respectively). Metaobjects handling distribution are based on a group management and communication protocols package. This package was reused in *FRIENDS* and is a major component of the GDS sub-system. Metaobjects for secure communications provide authentication based on the Needam-Schroeder protocol. All these mechanisms have been used a simple banking application: customers can perform operations on remote bank accounts and may have several accounts in different banks. We report here the two main classes used in this application, namely `Customer` and `Bank`.

Client class	Server class

`class Customer {`
`public :`
 	
`Customer(String name);`
`~Customer();`
 	
`void New_account();`
`void Deposit();`
`void Withdraw();`
`void Balance();`
`void Operations();`
`void Display();`
 	
`private:`
`Proxy_Bank *A_bank;`
`}`
 	
<metaobject declaration>
	`class Bank {`
	`public:`
	
	`Bank();`
	`~Bank()`
	
	`//MOP reflect :`
	`int Create_account(String filename,`
	`Account_info info, operation op);`
	`double Read (String name,`
	`int account_num, double amount);`
	`double Store (String name,`
	`int account_num, double amount);`
	`Account_info How_much (String filename);`
	`Operation* Last_operations`
	`(String filename, int i);`
	
	`private :`
	`void Load_from_file(String filename);`
	`void Write_to_file(String filename);`
	`//MOP reflect:`
	`void Start();`
	`}`
	<metaobject declaration>

`Customer` provides interactive facilities to the user for managing an account. The class `Bank` provides all the operations to manage bank accounts and is used to create a new instance any time a new customer is requesting access to its account. All

information related to a bank account is stored in a file whose access is provided by non-reflective private methods `Load_from_file` and `Write_to_file`. Metaobjects in the stack are initialised in sequence by their respective `Meta_Startup` method. After the recursive initialisation of metaobjects, control is return to the upper level for message reception; this is performed by a recursive interception of the empty reflective method `Start`.

Metaobject declarations used in our simple example are explained in the next sections with the following naming conventions:

	Client metaobject	Server metaobject
Distribution	`GD_CMO`	`GD_SMO`
Security	`SC_CMO`	`SC_SMO`
Fault-Tolerance	`[STS,PBR,LFR]_CMO`	`[STS,PBR,LFR]_SMO`

3.2 Using metaobjects

The remote interaction between a customer and the bank can simply be done using communication metaobjects with the following declarations:

Client side metaobject declaration
`//MOP reflect class Proxy_Bank: GD_CMO;`

Server side metaobject declaration
`//MOP reflect class Bank: GD_SMO;`

The object of class `Proxy_Bank` declared by the customer is connected to a communication metaobject. This `Proxy_Bank` object is empty and all method invocations are trapped by the `GD_CMO` metaobject. This metaobject forwards invocations to the `GD_SMO` metaobject. The later issues the corresponding base-level calls using `Meta_HandleMethodCall` provided by the MOP.

Adding fault tolerance mechanisms involves changing the previous metaobject declarations as follows (in the example below the have used the stable storage strategy implemented by a couple of metaobjects, namely `STS_CMO` and `STS_SMO`) :

Client side metaobject declaration
`//MOP reflect class Proxy_Bank: STS_CMO;`

Server side metaobject declaration
`//MOP reflect class Bank: STS_SMO;`

Now all method invocations are trapped by the `STS_CMO` metaobject and forwarded to the `STS_SMO` metaobject which is responsible for handling the call at the base-level and for saving state information to stable storage, a file accessible by NFS in the prototype. When a failure is detected, the client metaobject `STS_CMO` requests a new

copy of the server to be created and its state initialised with the last state information saved. The client metaobject is also responsible for re-sending uncompleted requests. Creation of new objects is performed by the factory service provided by the *GDS* sub-system. In the client metaobject `STS_CMO`, remote invocation is performed using the same model presented earlier. The server metaobject `STS_SMO` is seen as a proxy object by the client metaobject `STS_CMO`. The recursive use of metaobject implies that the stack is now composed of two metaobjects.

Using a different fault-tolerance strategy would simply involve changing the above declaration as follows (using the primary-backup strategy, for instance):

Client side metaobject declaration
`//MOP reflect class Proxy_Bank: PBR_CMO;`

Server side metaobject declaration
`//MOP reflect class Bank: PBR_SMO;`

The creation of the server object leads to the creation of two server replicas which are inserted in a group. All remote invocations are then sent by `PBR_CMO` to the group of replicas and thus received through communication metaobjects by two copies of `PBR_SMO`. When all metaobjects (fault tolerance, security and distribution) are used, the full set of declarations is the following :

Metalevel link		
SC_SMO	<MOP>	GD_SMO
PBR_SMO	<MOP>	SC_SMO
Bank	<MOP>	PBR_SMO

Metaobject declarations (server side)
`//MOP reflect class SC_SMO: GD_SMO`
`//MOP reflect class PBR_SMO: SC_SMO`
`//MOP reflect class Bank: PBR_SMO`

The new intermediate metaobjects (`SC_CMO, SC_SMO`) add a cryptographic signature to invocation messages at the client side and verify the signature at the server side respectively. The ciphering of messages could also be done by these metaobjects. All possible combinations of metaobjects have been successfully tested and involves updating those declarations which are defined in an include file at compile time. All these test cases are described in section 5.

4. Designing mechanisms/metaobjects

The mechanisms implemented in *FRIENDS* have been design using an object oriented design method (Business Object Notation) [14]. This notation provides a support for the structural and behavioural description of an object system. Static diagrams describe the structure of a system in terms of classes, represented by ellipsis, and their relationship, inheritance or composition, respectively represented by a single arrow

and a double arrow. Dynamic diagrams represent the behaviour of a system in terms of the messages exchanged by objects. The approach was the following; for each of the backward recovery mechanisms that we have implemented both static and dynamic diagrams were developed. Because of the similar nature of the mechanisms, we have been able from these diagrams to identify common structural and behavioural aspects. All these common aspects have been factorised in an inheritance hierarchy. The root class of the hierarchy is `MetaObj`. Intermediate levels provide generic classes for handling client or server metaobjects, and also checkpointing-based or replication-based mechanisms. The final leaves of this tree implement real mechanisms.

4.1. General Structure and Basic Classes

In backward error recovery, an error-free state substitutes for the erroneous state being detected as such; this state transformation consists in bringing the system back to a previously correct state. This involves the definition of recovery points, which are points in time during the execution of the process for which the then current state may subsequently need to be restored.

The object model and the use of a metaobject protocols encourages the definition of recovery points at the end of a method execution. When an error is detected, the currently running method must therefore be re-executed from the beginning, which implies the restoration of initial conditions (in particular the base-level object state) and the identification of the currently running method (method number, arguments, etc.). This framework is enforced by classes `FT_CMO` and `FT_SMO`, the former implementing the client part of the fault tolerance protocol and the latter implementing the server part. Application level method calls are trapped into `FT_CMO` by the method `Meta_MethodCall` which transmits the base-level method invocation to `FT_SMO`. The execution of the client's method invocation is performed in the method `FT_method_call` of the `FT_SMO` metaobject. `FT_SMO` executes the requested method and defines the recover point before sending back the method execution result to the client metaobject `FT_CMO`. Both classes are inherited and specialised for all the mechanisms described hereafter. The metaobject protocol also provides means to access object attributes from the metalevel to get the base-level object state. The state information is considered here as the set of object attributes.

The client metaobject interface (see Fig. 6) mainly consists in redefining `Meta_MethodCall` (belonging to `MetaObj`) so that it calls `FT_method_call` on the server metaobject. This server metaobject is seen locally in `FT_CMO` through a proxy called `ft_server`. The fact that it is remote is made transparent by the use of reflect declarations on the client and server sides. The declaration `reflect FT_SMO: CLIENT_MO` associates `ft_server` (instance of `FT_SMO`) with a client-type communication metaobject which turns `ft_server` into a proxy. So, the remote method `FT_method_call` can be executed using a simple `ft_server.FT_method_call` statement. On the server side, the declaration `FT_SMO: GD_MO` makes the `FT_SMO` instance a RPC-like server, waiting for request messages from remote clients. Initialisation of the stack and registration in the corresponding service group is done by redefining the `Meta_StartUp` method.

Basic fault tolerance client metaobject class

```
class FT_CMO: public MetaObj {
  FT_SMO ft_server;
public:
  void Meta_MethodCall (...) {
       reply=ft_server.FT_method_call();
  }
};

//MOP reflect FT_SMO: GD_CMO;
```

Basic fault tolerance server metaobject class

```
class FT_SMO: public MetaObj {
public:
  ArgPac FT_method_call (...) {
       Meta_HandleMethodCall(...);
       FT_method_end();
       return reply;
  }
  void FT_method_end () = 0;
  void FT_recover () = 0;
};

//MOP reflect FT_SMO: GD_SMO;
```

Fig. 6. Basic interface for backward recovery mechanisms.

In `FT_method_call`, `Meta_HandleMethodCall` (also belonging to `MetaObj`) is called first in order to propagate the method invocation to the base level; then `FT_method_end` is called to define the recovery point. The fact that effective method execution is handled by `MetaObj's Meta_HandleMethodCall` is essential because this means that the fault tolerance programmer does not need to call the application-level method explicitly and therefore does not need to know about the application functionalities. `FT_recover` is automatically called by the underlying error detection system service when an error is detected. Both methods strongly depend on the mechanism and are therefore abstract methods, which makes `FT_SMO` an abstract class.

4.2. A Complete Hierarchy

The fault tolerance mechanisms (STS, PBR, LFR) present several structural and behavioural common points. These can be factorised and reused in a hierarchy for backward error recovery mechanisms. In all these mechanisms, there is a backup of the server's state, either as a file on stable storage or as the state of a replica. We therefore introduce the `BACKUP` class which is used in general backward recovery mechanisms. This class defines the `FT_update` method; it is an abstract method because it depends on the mechanism implemented. The `FT_CMO`/`FT_SMO` framework is complemented with `BACKUP`, `FT_SMO` being a client of `BACKUP` (composition link). The stable storage service is accesses through a class called `STABLE_STORAGE`. All previously defined classes `STABLE_STORAGE`, `PBR_SMO` and `LFR_SMO` inherit from `BACKUP` and define `FT_update` respectively as:

- writing state information to stable storage;
- transmission of the primary's new state to the backups;
- notification of the method execution to the followers, which in turn execute it.

As for stable storage and primary-backup replication, the object state is systematically captured upon method termination before updating the backup object. This can also be factorised as `FT_method_end` belonging to a class `CHECKPOINT_SMO` derived from `FT_SMO`. `STS_SMO` and `PBR_SMO` inherit from `CHECKPOINT_SMO` and extend `FT_method_end`, respectively to save the captured state to stable storage or to send it to the backup replicas.

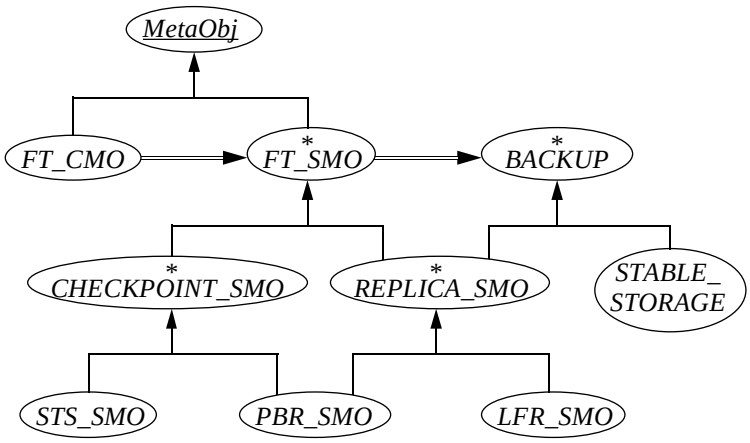

Fig. 7. Backward recovery metaobject class hierarchy.

For both replication mechanisms, the recovery procedure and the handling of client requests by spare replicas (backups and followers) are similar. This common behaviour can also be factorised in a class `REPLICA_SMO` inheriting from `FT_SMO`. `REPLICA_SMO` also inherits from `BACKUP` because its instances used in spare replicas hold successively saved states. `PBR_SMO` and `LFR_SMO` inherit from `REPLICA_SMO`, and `LFR_SMO` defines `FT_method_end` so that the current invocation is notified to the followers. `PBR_SMO` multiply inherits from `CHECKPOINT_SMO` and `REPLICA_SMO`. The later also handles reconfiguration by `cloning` a new replica. A simplified view of the class hierarchy is presented in Fig. 7.

5. Prototypes and performance issues

The first of our prototypes was developed on an Ethernet Network of Unix machines (Sun IPC, IPX, Ultra Sparc). Performance figures presented in this section have been obtained on this version. Then, a new prototype was developed on the Chorus microkernel. This involved porting the Open C++ compiler on Chorus and also porting the group communication system on this new platform. This software package was implemented as subsystem, the *GDS* sub-system. It is run as a set of actors, one of which is running in supervisor mode in order to have direct access to the Ethernet device driver into the microkernel. As soon as it was ready and that the

Open C++ compiler was also available on this new platform, the whole set of metaobjects libraries has been ported with limited effort. Then, we have tested various configurations of the application with a benchmark transaction composed of all possible operations provided by the banking application (5000 operations per transaction).

Operation type	Number
New account creation	1000
Credit	1000
Debit	1000
Balance	1000
Last operations	1000

In fact, the first of our experiment was to measure the cost of the metaobject protocol, i.e. a comparison between normal and reflective method invocation. As we can see on Fig. 8, the cost of trapping invocation by the metaobject protocol is about 2 times higher than the cost of a normal invocation. The variable *m* refers to the number of arguments, here from 1 to 5 times 512-bytes arguments. Nevertheless, it is less than 10µs whatever the machine used.

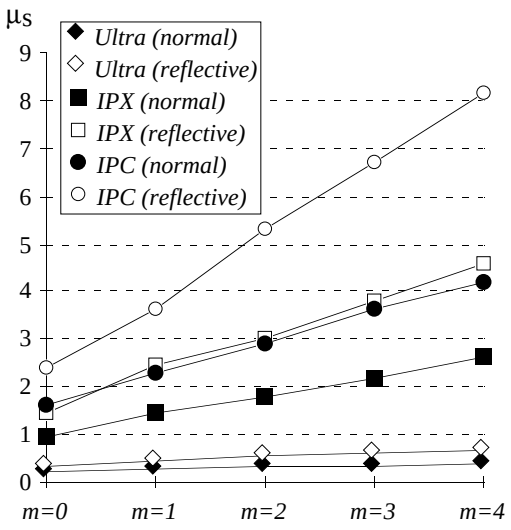

Fig. 8. Normal / Reflective invocation

Actually, the global performance of the system are very dependent on timing values of basic actions used by fault tolerance and security mechanisms, such as:

- cryptographic signatures computation and verification depending on the size of invocation messages (Fig. 9).
- group communication in replication protocols (Fig. 10), depending on the number of members in the group and the size of the message sent using the `Atomic_Send` primitive;

• read/write operations to stable storage depending on the machine used and the size of state information (Fig. 11);

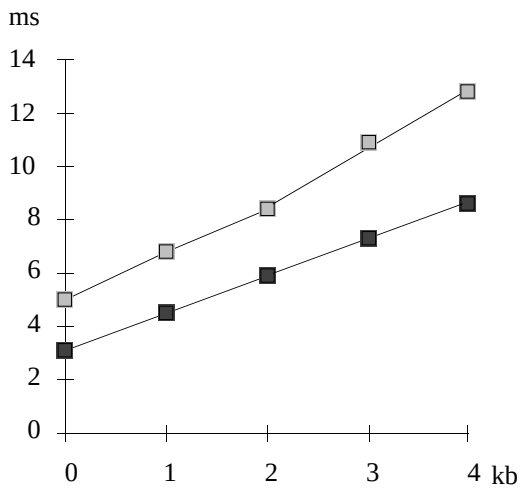

Fig. 9. Signature computation-verification

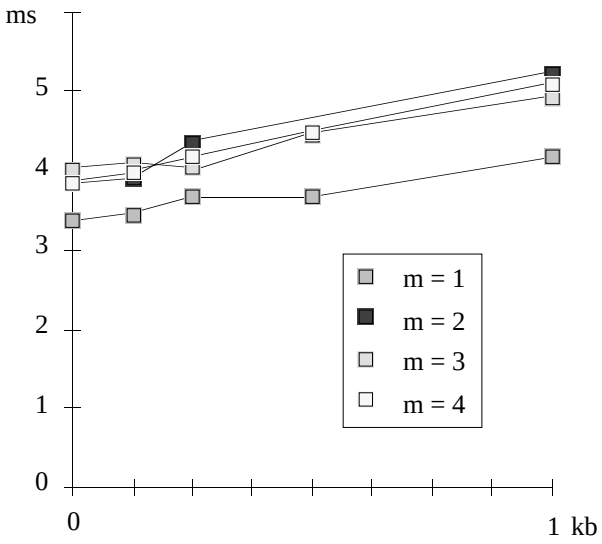

Fig. 10. Atomic multicast of messages

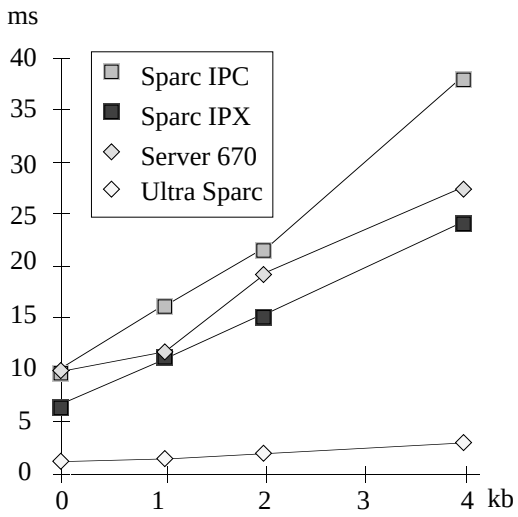

Fig. 11. Writing to stable storage

Clearly, the cost of a reflective invocation is negligible with respect to the time spent in security or fault tolerance metaobjects as shown in the figures above. This is summarised in Fig. 12.1 and 12.2. These figures show the execution time of each metaobject according to the configuration of the application, including the fault tolerance strategy used. The operation which was used here is the `Create_account` operation.

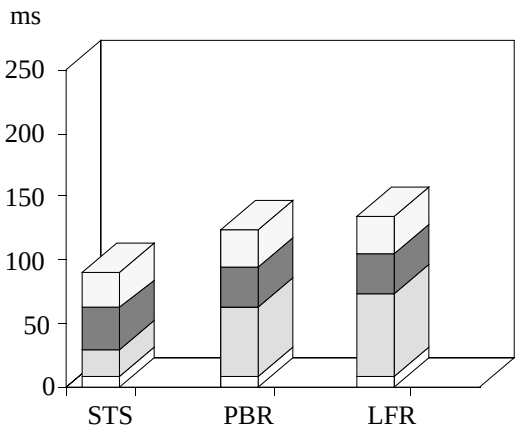

Fig. 12.1 Comparison of execution time for all fault-tolerance strategies

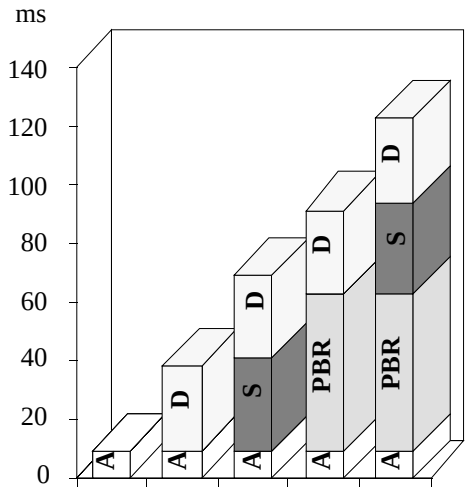

Fig. 12.2 Execution time for the test cases with the PBR strategy

The value given here have to be considered relatively to each other and the implementation on a different platform will give different results. Nevertheless the metaobjects execution (`GD: 29 ms; SC: 31,6 ms; FT-PBR: 61,4 ms` on a Sun IPX) is independent from the execution time of the application and thus should appear much lower in a different application context. The following table describes the 9 experiments performed on the Unix prototype and the execution time for each of the available operations according to various configurations of the application.

#	Configuration	create	credit	debit	balance	list last 10
1	{A}	8,9	53,7	55,1	19,3	19,3
2	{A,GD}	37,9	136,7	136,5	109,8	117,5
3	{A,SC,GD}	69,5	273,2	272,9	218,8	224,6
4	{A,STS,GD}	59,3	225,7	224,5	175,1	183,2
5	{A,PBR,GD}	91,5	360,9	382,8	287,8	300,2
6	{A,LFR,GD}	102,0	536,9	514,9	449,0	478,0
7	{A,STS,SC,GD}	86,0	362,0	359,0	282,9	287,0
8	{A,PBR,SC,GD}	123,1	392,5	698,8	319,4	303,8
9	{A,LFR,SC,GD}	133,6	568,5	546,5	480,6	509,6

6. Conclusion

The approach proposed and illustrated in this paper for the development of fault tolerant system is very positive and promising. The expected properties are transparency, independence, composability and reusability of mechanisms. The combined use of metalevel programming and object-oriented development makes mechanisms easier to develop and reuse in various operational environments. The limits we have observed mainly relate to the language used. The Open C++ compiler translates a normal class `C` into reflective class `refl_C` (with a `refl` prefix) and the latter has to be used in the programs instead of the normal one when necessary. This

makes programming more difficult on one hand but enables reflective classes to be easily identified on the other hand. Another limit was due to the pre-processing itself since Open C++ do not take into account inheritance of application classes. This problem should de solved easily using the new version of the Open C++ compiler [15]. Another limit is due to the Open C++ MOP because the meta-information does not contain any information regarding the structure of the object (composition and inheritance). This is an issue for further research that will lead to the definition of a more efficient metaobject protocol for implementing more complete fault tolerance mechanisms at the metalevel. Nevertheless, beyond these limiting factors, the architecture illustrated in this paper provided the expected properties.

The analysis of distributed fault tolerance mechanisms emphasis several similarities and common points. The identification of the structure and the behaviour of classes for each considered mechanism was well supported by an object oriented design method. Static and dynamic diagrams (i.e. *use cases* [16]) have been successfully used and lead to the definition of an inheritance hierarchy of generic fault tolerance mechanisms. Although physical faults (crashes) have been considered up to now, some classes could be reused to develop metaobjects for more subtle faults including software faults.

The experimental platform that was developed was used to validated this ideas. The number of mechanisms is obviously limited but sufficient to make extensive experiments and measurements. Client replication was recently implemented allowing nested invocations among objects replicated using various strategies, as in a more conventional fault tolerant distributed system such as Delta-4 [17]. The object model used in *FRIENDS* is very simple and more work has to be done in order to provide the user with more advanced computational models and OOPLs. Nevertheless, having a restricted programming model is not, to our viewpoint a real limit, since more complex model might be impossible to validate. Validation is thus another major research direction since testing object-oriented programs is, to our knowledge, still a challenge today. The object-oriented support provided by this system is not conventional and was designed and implemented on purpose. The use of CORBA-compliant layers on top of "componentized" microkernels such Chorus ClassiX r3 should be a promising issue. An implementation of the MOP within this layer should improve the openness of the system, making language issues less restricting.

References

1. Fabre J.C. and Pérennou T., "FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications", in *Proc. of EDCC-2*, LNCS 1150, Taormina, Italy, Oct. 1996, pp. 3-20.

2. Fabre J.C. and Pérennou T., "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", to appear in IEEE Transactions on Computers–Special Issue on Dependability of Computing Systems, Jan. 98.

3. Maes P., "Concepts and Experiments in Computational Reflection", in Proc. of OOPSLA'87, Orlando, USA, 1987, pp. 147-155.

4. Kiczales G., des Rivières J. and Bobrow D.G., The Art of the Metaobject Protocol, MIT Press, 1991.

5. Stroud R.J., “Transparency and Reflection in Distributed Systems”, ACM Operating Systems Review, 22 (2), April 1993, pp. 99-103.

6. Agha G., Frølund S., Panwar R. and Sturman D., “A Linguistic Framework for Dynamic Composition of Dependability Protocols”, in Proc. of DCCA-3, 1993, pp. 197-207.

7. Rozier M., Abrossimov V., Armand F., Boule I., Gien M., Guillemont M., Hermann F., Kaiser C., Langlois S., Leonard P. and Neuhauser W., “Overview of the Chorus Distributed Operating System”, Chorus Systems Technical Report CS-TR-90-25, 1990, 45 p.

8. Birman K. and Joseph T., Exploiting Virtual Synchrony in Distributed Systems”, on Proc. of the 11th ACM SOSP, special issue Operating Systems Review, vol. 21 (5), pp. 123-138.

9. Rodrigues L. and Veríssimo P., “xAMp: A Protocol Suite for Group Communication”, in Proc. of SRDS-11, 1992, pp. 112-121.

10. Chiba S. and Masuda T., “Designing an Extensible Distributed Language with Metalevel Architecture”, in Proc. of ECOOP'93, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993, pp. 482-501.

11. Chiba S.,”Open C++ Release 1.2 Programmer's Guide”, Technical Report No. 93-3, Dept. of Information Science, University of Tokyo, 1993.

12. Yokote Y., “The Apertos Reflective Operating System: The Concept and Its Implementation”, in Proc. of OOPSLA'92, 1992, pp. 414-434.

13. Masuhara H., Matsuoka S., Watanabe T. and Yonezawa A., “Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently”, in Proc. of OOPSLA'92, 1992, pp. 127-144.

14. Waldén K. and Nerson J.M., Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems, The Object-Oriented Series, Prentice Hall, 1995, 438 p.

15. Chiba S., “A Metaobject Protocol for C++”, in *Proc. of OOPSLA'95* (ACM Conference on Object-Oriented Programming, Systems, Languages and Applications), Austin (TX-USA), Oct. 1995, pp. 285-299.

16. Jacobson I., Christerson, Jonsson P. and Overgaard, “Object-Oriented Software Engineering: A Use Case Driven Approach”, Addison Wesley, Wokingham, UK, 1992.

17. Powell D., “Distributed Fault Tolerance — Lessons Learnt from Delta-4”, in Hardware and Software Architecture for Fault Tolerance: Experiences and Perspectives (M. Banâtre and P.A. Lee, Eds.), LNCS 774, Springer Verlag, 1994, pp.199-217.