A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach

Jean-Charles Fabre and Tanguy Pérennou

Abstract—The FRIENDS system developed at LAAS-CNRS is a metalevel architecture providing libraries of metaobjects for fault tolerance, secure communication, and group-based distributed applications. The use of metaobjects provides a nice separation of concerns between mechanisms and applications. Metaobjects can be used transparently by applications and can be composed according to the needs of a given application, a given architecture, and its underlying properties. In FRIENDS, metaobjects are used recursively to add new properties to applications. They are designed using an object oriented design method and implemented on top of basic system services. This paper describes the FRIENDS software-based architecture, the object-oriented development of metaobjects, the experiments that we have done, and summarizes the advantages and drawbacks of a metaobject approach for building fault-tolerant systems.

Index Terms—Metalevel architecture, metaobject protocols, distributed fault tolerance, object-oriented methods and languages, reusability.

1 INTRODUCTION

THE use of a metalevel architecture to build dependable systems has emerged recently and conceptually appears to be promising. The advantages of this approach have been advocated in several other works [1], [2], [3] using a reflective system approach, rather than using a reflective language approach. The objective of FRIENDS was to investigate the use of object-oriented techniques and a reflective language approach (compile time reflection and metaobjects) [4], [5] for the development of fault- and intrusion-tolerant distributed systems.

The initial aim was to identify the advantages and drawbacks of this approach. Here is a nonexhaustive list of questions we were interested in: What is the role of metaobjects with respect to nonfunctional requirements? Can we always rely on metaobjects when implementing a given type of mechanism? What is the minimal system support for metaobjects? What is the performance of such a system? To what extent are the expected properties really satisfied? What are the advantages and drawbacks of such an architecture with respect to more conventional solutions? To what extent are metaobjects reusable components? In addition to these questions, the first one is the following: Why metaobjects? It is clear that now a number of fault tolerance mechanisms have reached full maturity. However, in practice, the integration of such mechanisms within applications still raises several problems, mainly related to flexibility. We understand flexibility in the following way: ease of use and transparency of the mechanisms for the programmer; independence of the mechanisms with respect to each other and composability on a case-by-case basis; reusability of existing mechanisms to derive new ones. These properties are detailed in Section 2. To our knowledge, none of the solutions traditionally used manages to ensure all these properties at the same time. The approach which is developed and illustrated in this paper aims at providing a good balance among these properties. It is based on object-orientation (languages and development methods), metaobject protocols and, also, to some extent, microkernel technology. The notions of reflection and metaobject protocols in object-oriented languages [6], [7] have already proven to be both efficient and elegant for the integration of application-orthogonal concerns in a highly flexible way. Among many other examples, PCLOS implements persistent objects [8], $\mathbf{R}^2$ allows inclusion of soft real-time constraints in object-oriented applications [9], Object Communities provides distribution transparency [10].

Section 2 describes related work on the integration of fault tolerance mechanisms within applications, describes what a metaobject protocol is, and delineates what is expected from its use. Section 3 describes the architecture of a system supporting dependable applications using a metaobject protocol. Section 4 describes our application model and stresses the separation of concerns obtained when programming fault tolerance or some security mechanisms[1] using multiple metalevels. The approach is then illustrated with examples of application objects and metaobjects. Section 5 describes the development of a hierarchy of metaobjects using an object-oriented design method. Section 6 is a general discussion of the approach. Section 7 briefly describes our experiments and provides performance measurements for the prototype running on a network of Unix machines.

- *The authors are with LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse cedex 4, France. E-mail: Jean-Charles.Fabre@laas.fr.*

Manuscript received 4 Feb. 1997; revised 1 Oct. 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105900.

[1] In this paper, the term “security” must mainly be understood as “secure communication.”

2 Definitions and Related Work

2.1 Definitions

The various properties described below that can be expected for the implementation of dependability-related mechanisms are referred as “flexibility” in the remainder of this paper. Nevertheless, we restrict ourselves here to the following properties in the case of hardware fault tolerance and communication security.

2.1.1 Ease of Use

Mechanisms implementing fault tolerance and secure communication should be easily used by application developers. This should also be true for the developers of the mechanisms themselves, for whom, e.g., group communication mechanisms should be easy to use. When different mechanisms are used in different applications, the programmer or the user of a given application should be able to select the adequate mechanism, even if its implementation remains hidden. So, we make a distinction between transparency from a usage viewpoint and visibility from a configuration viewpoint.

2.1.2 Reusability

From the viewpoint of the mechanism’s developer, reuse can take one of the following two forms, *generality* and *extensibility* [11]. Generality means that mechanisms can be reused without modification in new applications. Extensibility means that part of an existing mechanism must be updated and modified in order to derive a mechanism meeting different requirements. Object-oriented design methods and languages provide a good way to obtain both properties: Abstraction and encapsulation help ensure generality, while inheritance or delegation allow incremental development and extensibility.

2.1.3 Composability

This property refers to the ability to use several independent mechanisms within the same application depending on the application’s characteristics and requirements. For instance, several fault tolerance strategies can be used for different objects in the same application without any clash. Mechanisms can also be of different types: An application may need both fault tolerance and security mechanisms. The security mechanisms can be inserted or removed from the application quite easily without any modification of its source code. The definition of composability here is similar to the notion of composition of dependability protocols described in [1]. Nevertheless, our notion is more limited; the notion of composition of dependability protocols extends our notion of composability in the sense that a given fault tolerance protocol can be composed of several elementary protocols.

2.2 Conventional Approaches

The approach presented in this paper is an evolution of previous work on the integration of fault tolerance mechanisms within distributed systems, either at the system level, through system services, or at the user level, through the use of libraries.

The *system approach* provides fault tolerance mechanisms embedded in the underlying runtime system which are (almost) transparent to the application programmer. For example, the Delta-4 system [12] offers several replication protocols based on a multicast communication system supporting error detection and voting protocols. Mechanisms are transparent and easy to use in this case. However, because they are integrated into the operating system, mechanisms are not easy to access and customize. Moreover, composability of various mechanisms dealing with different fault classes is not always possible. Adding various types of mechanisms to an application on a case-by-case basis, e.g., client-server authentication, is not easy.

The *library approach* provides basic mechanisms allowing users to tailor their own fault tolerance mechanisms to suit their needs by using constructs and primitives. For instance, Isis [13] offers software constructs (e.g., *coordinator-cohort*), group management, and atomic broadcast primitives, on top of which primary/backup and active replication, for example, can be built. Although more flexible, the programmer is, however, responsible for correct use of library functions at appropriate places in the source code to implement a given fault tolerance mechanism. This ability may require a good knowledge of fault tolerance techniques and basically lacks ease of use.

The *object-oriented development* of such libraries provides the user with classes, rather than functions. Inheritance then makes it easier to adapt fault tolerance mechanisms incrementally to specific needs or to add new features. This approach should, therefore, achieve very good reusability. Examples of such a use of inheritance can be found in Avalon/C++ [14] and Arjuna [15]. So, on one hand, a system approach provides full transparency of the mechanisms, and, on the other hand, object-oriented libraries provide reusability, but none of these approaches manages to combine both properties nor provides a satisfying solution concerning composability. Actually, a careful observation of the code written by library users reveals that functions are used almost systematically at specific points of the computational model, such as object creation and deletion, beginning and ending of methods. This is the kind of problem a metaobject protocol can solve in an elegant way.

2.3 Metaobject Protocols and Language Issues

The essence of metaobject protocols (MOPs) is to give to the user the ability to adjust the language implementation to suit their particular needs. Metaobject protocols are based on reflection [7] and object-orientation. Reflection exposes the language implementation at a high level of abstraction, making it understandable for the user while preserving the efficiency and portability of the default language implementation. Object-orientation provides an interface to the language implementation in the form of classes and methods so that variants of the default language implementation can be produced, using specialization by inheritance. Instances of such classes are called *metaobjects*. The notion of *protocol* relates here to the interaction between object and metaobject. In class-based reflective languages, this interface generally comprises at least instance creation and deletion, attribute read or write access, method call. An a priori argument against reflection-based languages is that they are not efficient. A counter-example is ABCL/R2, an object-oriented concurrent reflective language that, in terms of performance, compares with C used with lightweight processes [16]. Compile time reflection leads to good performance with respect to partial evaluation, as illustrated in Section 7.

2.4 Using a Metaobject Protocol for Dependability

The ability to adapt some aspects of the language implementation can be delegated to a third party, rather than the user, e.g., a fault tolerance or security specialist. In this way, a clean separation of concerns between the application and mechanisms for fault tolerance or secure communication can be achieved. This approach enables the role of different programmers with a different basic knowledge to be clearly identified and their task made easier: the application programmer, the fault tolerance programmer, the security programmer, the distribution programmer. All of them share the same knowledge of object-oriented programming and some about metaobject protocols.

In a simple metaobject protocol, the metaobject is scheduled to perform some actions when the object is created (deleted) or when an object's method is invoked. Conversely, the metaobject can activate object's methods and access object's attributes (the object state). In practice, the interaction between a client object and a server object can be controlled by a couple of metaobjects, as illustrated in Section 3.2.1.

This approach can be applied recursively so that the addition of some dependability-related mechanism takes place in a simple and systematic way. Nevertheless, metaobject protocols are not a panacea and it is not claimed here that they can be used on their own to build dependable distributed systems. Several basic services must be implemented at the system level, like error detection (to ensure high coverage of the failure mode assumptions [17]) or a security kernel (that must be always invoked and tamper proof). Other system services, like group management and atomic multicast protocols, authentication, and authorization servers, are also necessary. The respective roles of metaobjects and system services in the presence of multiple mechanisms are presented in Section 3 and further discussed in Section 6.

2.5 Closely Related Work

The FRIENDS system shares the same philosophy of closely related projects like MAUD [1] and GARF [3]. The idea is to handle dependability mechanisms at a separate abstraction level and to bind them to application objects according to their needs. In the above mentioned examples, the approach relies mainly on the ability to intercept invocations and to redirect them to some behavioral object or metaobject. The interception relies more on some runtime system capabilities to redirect messages than on language facilities. Also, creation of objects is reified by the runtime system rather than by the language. Our approach relies more on language facilities to handle object creation, invocations, and, also, object state (attributes). Reification of both object behavior and state information, is of high interest with respect to fault tolerance. The more powerful the reflective language is, the more object state information can be optimized in checkpoints. The main advantage of a language approach is that no assumption on the underlying system is required. The main drawback is that a specific language is needed and that some programming conventions still have to be obeyed.

In the Deva Esprit project, several other colleagues take advantage of reflective capabilities of a language and metaobjects to handle real-time constraints [18], software fault tolerance [19], or transactional models [20]. The handling of real-time issues at the metalevel is more complex, because it involves access to features which are normally devoted to internal components of the system kernel.

All these works illustrate, in different ways, the interest of reflection in dependable computing.

2.6 Motivations and Previous Work

In our previous work [4], the use of metaobjects for implementing fault tolerance mechanisms was investigated using a single metalevel. Several metaobjects classes for various replication strategies were developed and experimented. This was very promising and showed that transparency and separation of concerns could be obtained for the application programmer.

However, a single-metalevel approach suffers from several drawbacks. First of all, the interaction between replicas handled at the metalevel was rather complex because of the use of system calls to group management services. Likewise, remote interaction between application objects was implemented at the base-level and, thus, very dependent on the communication mechanisms used. A second problem was the difficulty in adding some security aspects transparently (authentication, ciphering, and signature checking). All security-related statements were mixed with the source code for fault tolerance and group management. Because remote interaction and security were not handled as separated and independent abstractions, the flexibility and the reusability of the metaobjects that were initially developed was very limited.

This was the first motivation for separating fault tolerance, security, and group communication within several metaobjects. The idea was to hide distribution at the fault tolerance level using communication metaobjects leading, thus, to two metalevels instead of one. The separation of concerns between fault tolerance and distribution protocols as two distinct metalevels enables security mechanisms to be inserted as a new metalevel.

3 THE FRIENDS SYSTEM

The objective of the FRIENDS[2] system was to provide mechanisms for building fault-tolerant applications in a more flexible way. Flexibility is obtained through the provision of object-oriented libraries of metaobjects and also through the provision of subsystems on a microkernel platform. Subsystems can be reused as they are and ported to various platforms. Metaobject libraries can be rapidly ported to a new platform (compiler availability) and can be extended using object-oriented development techniques. The metalevel approach used here provides new means to develop functionalities that are traditionally in the operating system, as metalevel software.

[2] FRIENDS stands for Flexible and Reusable Implementation Environment for your Next Dependable System.

3.1 System Architecture and Assumptions

The architecture of the system (see Fig. 1) is composed of several layers:

1) The kernel layer, which can be either a Unix kernel or better a microkernel, such as Chorus [21],
2) The system layer, composed of several dedicated subsystems, one for each abstraction, and, finally,
3) The user layer, dedicated to the implementation of applications.

3.1.1 System Layer

The system layer is organized as a set of subsystems. In microkernel technology, a subsystem corresponds to a set of services implementing any software system, for instance, an operating system on top of the microkernel (e.g., Unix on Chorus). In FRIENDS, each subsystem provides services for fault tolerance, or secure communication, or distribution. Any subsystem may be hardware- and software-implemented. The three necessary subsystems are the following:

- *FTS* (Fault Tolerance Subsystem) provides basic services mandatory in fault-tolerant computing, in particular, error detection and failure suspectors, which must be implemented as low level entities. This subsystem also includes configuration and replication domains management facilities and a stable storage support.
- *SCS* (Secure Communication Subsystem) 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 authorization server, a directory server, an audit server.
- *GDS* (Group-based Distribution Subsystem) 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.

Subsystems provide basic services required by the mechanisms implemented with metaobjects. These services can be seen as *Software Replaceable Units* (SRUs). Using microkernel technology, the system layer can easily be composed of the required subsystems, each of them using the appropriate SRUs.

3.1.2 User Layer

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

- *libft_mo* provides metaobject classes for various fault tolerance strategies (based on stable storage or replication) with respect to physical faults considering fail-silent nodes.
- *libsc_mo* provides metaobject classes for various secure communication protocols, using ciphering techniques, signature computation, and verification based on secret or public key cryptosystems.
- *libgd_mo* 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.

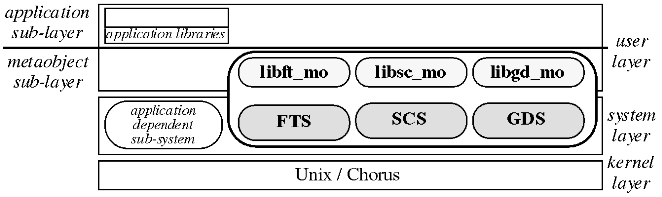


Fig. 1. Overall system architecture.

For active replication strategies, we assume that application objects have a "deterministic behavior." Deterministic behavior ensures that all recipients processing the same input messages obtain the same results. Since atomic multicast ensures that all correct recipients receive the same input messages in the same order, any method invocation will produce the same results on any of the object replicas. Concurrency and other sources of nondeterminism have not been considered yet.

3.1.3 Overall Architecture

The static view of the overall system architecture is illustrated in Fig. 1. FRIENDS provides a set of subsystems and several libraries of metaobject classes.

The implementation of any abstraction (fault tolerance, secure communication, distribution) is thus divided into a library of metaobject classes and the corresponding subsystem, thus spanning, at least partially, the user and system layers.

3.2 Multilevel Application Model

For every object in a FRIENDS application, a single or several metaobjects can be used. When just distribution is required, only communication metaobjects are used at the metalevel. When fault tolerance is required, then the application level declares fault tolerance metaobjects according to the preferred strategy. Distribution is now handled by fault tolerance metaobjects by meta_-metaobjects. From the viewpoint of the programmer of fault tolerance metaobjects, distribution is handled at the metalevel using communication metaobjects. When security is necessary, then, instead of using standard communication metaobjects, the fault tolerance programmer declares security metaobjects. These metaobjects ensure security at the communication level and take advantage of the metaobject approach for remote communications. Such a recursive use of metaobjects leads to several metalevels in the final application. In the next sections, we describe the use of metaobjects for handling distribution, communication security, and fault tolerance, including inter-replica protocols.

3.2.1 Distributed Application Model

An application is regarded as a collection of communicating objects developed using an object-oriented programming language (currently, C++). Every application object is mapped by GDS onto a runtime object, depending on enti-

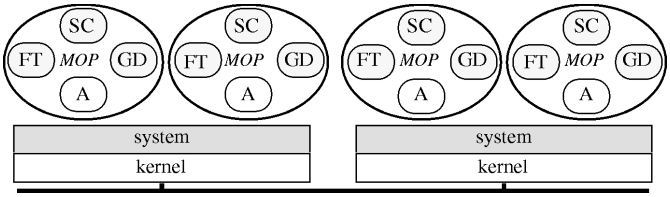


Fig. 2. A distributed application using FRIENDS.

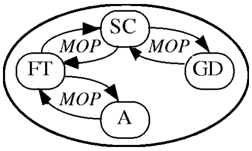


Fig. 3. Base and metalevel interaction.

ties handled by the underlying operating system (Unix processes or Chorus actors). Each runtime object is not only composed of an application object, but also contains one or several metaobjects within the same address space.

```
Runtime_object = {A, FT, SC, GD} with:
  A   : application object
  FT  : fault tolerance metaobject
  SC  : secure communication metaobject
  GD  : group-based distribution metaobject
```

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. Ideally, adding properties to an application involves adding other metaobjects to the set. The notion of *metaobjects set* is similar to the notion of *metaspace*, defined in Apertos [22] and also to the notion of *reflective object tower* in ABCL/R2 [16]. Fig. 2 depicts our application model on a distributed system configuration.

Within one runtime object, the interaction between the application object and the metaobjects is done through the MOP. 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 abstraction is handled by a protocol between two metaobjects. The application programmers just write the application objects and select the appropriate set of metaobjects.

3.2.2 The Multilevel Approach

In order to solve the problems mentioned in Section 2.6, a *three metalevel* application model was defined. Any (runtime) object is organized using several levels: the first level or the base-level (the application object), several intermediate metalevels (metaobjects for fault tolerance and secure communication), and, finally, the last metalevel responsible for handling objects interaction. This structure of the application implies a sequence of interactions through the MOP, as shown in Fig. 3. The set of metaobjects is organized as a *stack*.

The minimal specifications of the MOP (see Section 4.1 for a more complete description) that we need and that we have used are the following:

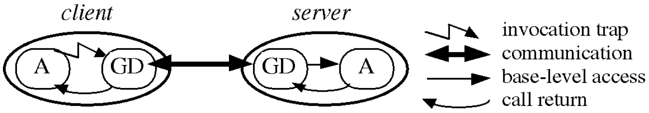


Fig. 4. Using metaobjects for distribution.

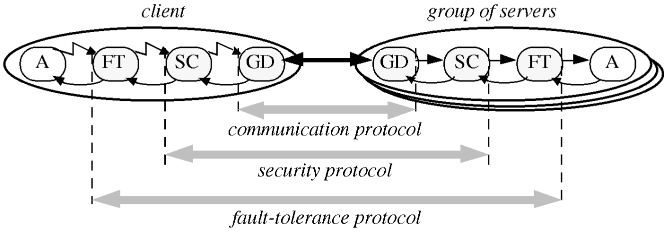


Fig. 5. Multilevel implementation of the client-server protocol.

- Object creation/deletion: The creation of object involves creating several replicas, registration in a communication group and authentication when needed;
- Invocation trapping: The method invocation semantics can be implemented in a different way, according to a given fault tolerance strategy, for instance;
- Base-level access: The base-level methods and attributes can be manipulated from the metalevel, respectively, to execute object's methods at active replicas and to get the object state.

Considering just one metalevel handling distribution, any server method invocation is trapped by the client metaobject on the client side, an invocation message is forwarded to the server side, this message is received by the server metaobject, and, finally, the method is executed at the base-level. This protocol is illustrated in Fig. 4. An example implemented using this model is given in Section 4.2.

Considering now several intermediate levels, each server method invocation is trapped by the next metalevel. This is recursively done until the last metalevel (GD), where an invocation message is forwarded to the server. The invocation message is received by the server metalevel (GD) and the invocation is propagated recursively through intermediate levels to the base-level where the method is executed. This is illustrated in Fig. 5.

This figure also shows the various underlying protocols between a client object and a replicated server object. With this application structure, any intermediate metalevel can be added or removed quite easily, thus adding or removing the corresponding underlying protocol.

This multilevel model is also used in the implementation of inter-replica protocols on the server side. For instance, the interaction between the primary and the group of backup replicas is such that the primary is a client of the group of backup replicas. The primary replica captures the state of the base-level object and transparently invokes an `update_state` method of the backup replicas (Fig. 6). This is possible because, in many replication strategies, the inter-replica protocol is a client-server protocol.

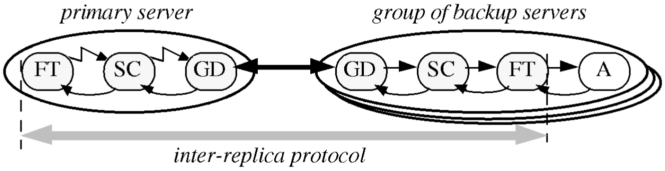


Fig. 6. Multilevel implementation of the primary-backup protocol.

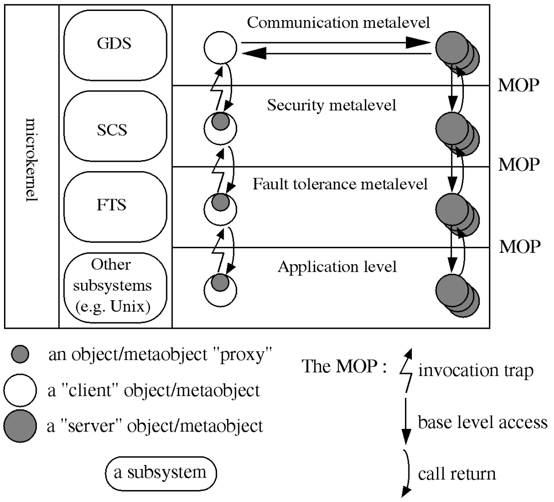


Fig. 7. The FRIENDS system approach.

3.2.3 A Global View of FRIENDS

The overall architecture and application model using metaobjects is depicted in Fig. 7. This figure illustrates the FRIENDS system from a different perspective. The application structuring is highlighted and shows that the same model is used for programming at any level. Objects and metaobjects, but also metaobjects themselves, cooperate through the MOP. This figure also shows that metaobjects rely on basic services running on top of the microkernel.

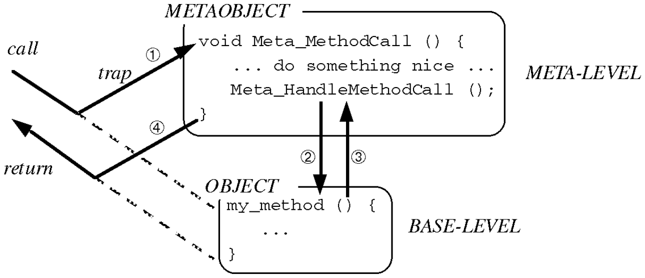
4 Implementing and Using Metaobjects

The objective of this section is to show how objects and metaobjects can be programmed and bound (at compile time) using a simple MOP. The transparency of the mechanisms for the application programmer is illustrated by a simple example. According to the models presented in Section 3, metaobjects for each abstraction are briefly presented in an Open C++-like syntax. Open C++ v1.2 [23] is the language used in our experiments. The role of the binding declarations is explained in each case. We also illustrate how to build the object-metaobject stack with no, one, or two intermediate metalevels.

4.1 A Simple MOP Definition

The following simple MOP is used throughout the rest of this section: Each object is controlled by a unique metaobject and the binding is done on a class-by-class basis. The binding between an application class A and a metaobject class M is realized by the statement: `reflect A: M`. All metaobject classes inherit from the predefined class `MetaObj` and have the following interface:

```
class MetaObj {
public:
  void Meta_StartUp ();
  void Meta_CleanUp ();
  void Meta_MethodCall (int m_id, ArgPac args,
    ArgPac reply);
  void Meta_Read (int var_id, ArgPac value);
  void Meta_Assign (int var_id, ArgPac value);
private:
  void Meta_HandleMethodCall (int m_id, ArgPac
    args, ArgPac reply);
  void Meta_HandleRead (int var_id, ArgPac
    value);
  void Meta_HandleAssign (int var_id, ArgPac
    value);
};
```



```
void Meta_MethodCall () {
   ... do something nice ...
   Meta_HandleMethodCall ();
}
```

```
my_method () {
   ...
}
```

Fig. 8. Invocation trapping.

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` packs the results when `Meta_MethodCall` returns. `Meta_Read` is called when an attribute identified by `var_id` is read and `value` contains 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 behavior 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. 8 illustrates how invocation is trapped and can be adjusted with this MOP.

Classes that are not bound to a metaobject class have the default class behavior. This MOP is a simplified version of the Open C++ MOP [10]. Nevertheless, any reflective object-oriented language providing this MOP could be used instead.

4.2 Handling Distribution

The idea here is to provide a set of metaobjects classes providing access to remote, replicated, and/or shared objects. These metaobjects use other classes, based on GDS, providing an object-oriented interface to group management services. A server is seen by the client through a local representative, a proxy. This proxy is bound to a client metaobject whereas the server itself is bound to a server metaobject.

Client source code	Server source code

class Customer {
protected:
  BankAccount account ("Bob");
public:
  Customer () {
       account.Credit (1000);
       account.Debit (500);
       printf (account.Balance());
  }
};
	class BankAccount {
	protected:
	  int val;
	public:
	  BankAccount()
	       { val = 0; }
	  void Credit(int x)
	       { val = val + x; }
	  void Debit(int x)
	       { val = val - x; }
	  int Balance()
	       { return val; }
	};

reflect BankAccount: CLIENT_MO;	reflect BankAccount: SERVER_MO;

int main () {
  Customer client;
}
	int main() {
	  BankAccount server("Bob");
	}

Fig. 9. A client-server application using metaobjects.

The CLIENT_MO and SERVER_MO metaobjects provide transparent access to remote groups of objects.

Fig. 9 shows a simplistic client-server application example. The extra argument "Bob" passed to the server constructor is a group global identifier automatically transmitted to the metalevel CLIENT_MO and SERVER_MO constructors. It is used to identify the underlying group dedicated to this service. Nodes where server replicas are created are declared in a configuration file (notion of replication domain). The server is created on the appropriate nodes and shared if different clients use the same identifier. The user only runs the client; then, the constructor of CLIENT_MO creates all remote server replicas when necessary. Likewise, the destructor of CLIENT_MO is responsible for the deletion of all server replicas. A service called factory was developed in order to deal with the creation of remote objects.

The extra task of the application programmer just corresponds to some declaration statements. To use an additional metalevel, say fault tolerance, the programmer only has to change reflect BankAccount: CLIENT_MO into reflect BankAccount: FT_CMO in the client source code, and reflect BankAccount: SERVER_MO into reflect BankAccount: FT_SMO in the server source code. The declaration of metaobjects handling distribution is then delegated to the next metalevel. These declarations may be inserted automatically, as discussed in Section 6.

4.3 Fault Tolerance

Several fault tolerance mechanisms have been designed (see Section 5) and implemented in the form of metaobject classes implemented on FTS: a mechanism based on stable storage, primary-backup, and leader-follower replication protocols.[3] In most cases, from the fault tolerance programmer viewpoint, the development of metaobject classes is done using the same class pattern. Fig. 10 shows a simplified view of the LFR_CMO and LFR_SMO classes implementing the leader-follower strategy [12].

The client class defines mainly how a method invocation is handled: Meta_MethodCall(). The method invocation is propagated to the server metaobject using a simple statement: FT_server.FT_method_call(). The server metaobject is transparently invoked from the client metaobject thanks to the use of the upper distribution metalevel. The server is declared as a local attribute FT_server of class LFR_SMO. This class is bound to CLIENT_MO handling the client behavior at the distribution metalevel.

The server class holds methods for handling the invocation (all replicas execute the method in this case) and the inter-replica protocol (IRp). The FT_method_call() method is responsible for handling the server method invocation at the base-level. This is done using Meta_HandleMethodCall(). The FT_notify method enables the leader to synchronize with the followers by telling them that a given method was executed.[4] This is done after any base-level method execution, by Followers.FT_notify(). The followers are invoked transparently from the leader, again thanks to the use of the distribution metalevel. In all replication mechanisms, a replica crash is detected by FTS, which activates the FT_recover method of one of the alive replicas. In the example given here, if the leader crashes, then FT_recover causes a follower becoming a leader and the creation of a new replica within the appropriate replication domain.

Two declarations are mandatory: reflect LFR_SMO: SERVER_MO and reflect IRP_LFR_SMO: CLIENT_MO indicate that the server and the followers are, respectively, bound to a server metaobject and a client metaobject at the distribution metalevel. The latter declaration enables the leader to transparently invoke the followers during the in-

[3] Only server failure is tolerated by these mechanisms.

[4] In the primary-backup metaobject, the method FT_Update is used instead of FT_Notify to update the base-level state after each base-level method invocation. This method writes the base-level state of backup replicas using Meta_HandleAssign.

LFR_CMO	LFR_SMO

class LFR_CMO {
protected:
  LFR_SMO FT_server;
public:
  void Meta_StartUp () {...}
  void Meta_MethodCall (...) {
      ...
      reply = FT_server.FT_method_call (...);
      ...
  }
};
	class LFR_SMO {
	protected:
	  IRP_LFR_SMO Followers;
	public:
	  void Meta_StartUp () {...}
	  Argpac FT_method_call (...) {
	      ...
	
	      Meta_HandleMethodCall(...);
	      ...
	      Followers.FT_notify (...);
	  }
	  void FT_notify (...) {...}
	  void FT_recover (...) {...}
	};

reflect LFR_SMO: CLIENT_MO;	reflect LFR_SMO: SERVER_MO;
	reflect IRP_LFR_SMO:CLIENT_MO;

Fig. 10. Fault tolerance metaobject classes.

ter-replica protocol, as already mentioned. This frees fault tolerance metaobjects from handling distribution problems, such as remote creation/deletion, group management, and atomic multicast message passing. When implementing metaobjects for replication protocols, the programmer only assumes that all server replicas receive the same invocation requests in the same order, even when the server is shared by multiple clients. In this example, no secure communication level is used and, thus, fault tolerance metaobjects are bound to distribution metaobjects. If such a level is used then binding declarations must be updated accordingly.

4.4 Secure Communication

The introduction of metaobjects for secure communications illustrates the flexibility provided by the metaobject approach described in this paper. The interface of metaobject classes at this level is similar to the class interface defined in the previous section, as shown in Fig. 11.

Based on this model, metaobject classes responsible for the authentication of the client user have been implemented. Currently, authentication is based on the Needham-Schroeder protocol [24] upon client creation and signatures are used upon every server invocations

On the client side, `Meta_StartUp` authenticates the client and receives a session key. This session key is transparently propagated (in a ticket) to the server by the invocation `SC_server.SC_GetSession-Key()`. Within `Meta_MethodCall`, every server method invocation is signed and propagated transparently to the server by `SC_server.SC_method_call()`. On the server side, `SC_method_call` verifies the signature and, if it is correct, the invocation is propagated to the base-level by `Meta_HandleMethodCall()`. The base-level can be the fault tolerance or the application level.

5 Object-Oriented Development of Metaobjects

In this section, we describe the design of fault tolerance mechanisms and underline how metaobjects provide a clean and convenient framework for the design process. This is mainly because application objects and fault tolerance metaobjects interact through a well-defined interface, i.e., the metaobject protocol. This interaction is automatically handled at compile-time. In addition, fault tolerance metaobjects are designed and implemented without direct communication statements because they are hidden in communication meta-metaobjects. So the design of fault tolerance metaobjects can be made without concern for either the application functionalities or communication details.

5.1 Object-Oriented Design Notation

We first briefly present a graphical notation used to describe some of our design steps. This notation is taken from BON (Business Object Notation [25]) and provides support for the description of both the structure and the behavior of a system. Static diagrams describe the structure of a system in terms of classes, represented by ellipses, and their relationship, inheritance, or composition, respectively, represented by a single arrow and a double arrow. Dynamic diagrams represent the behavior of a system in terms of the messages exchanged by objects. Objects are represented by rectangles and named by their class; when this is not accurate enough, an instance name is also mentioned. Messages are represented by dashed arrows with a sequence number and oriented from the sender to the receiver. A scenario is associated with each dynamic diagram; it is a table where a description of each message is given according to its sequence number. Arrows having no origin object represent external input events received by the system, and arrows having no destination object represent external output events.

5.2 General Structure

In backward error recovery, an error-free state substitutes for the erroneous state being detected as such; this state transformation consists of 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.

SC_CMO

```
class SC_CMO {
protected:
  SC_SMO SC_server;
  session_key SK;
public:
  void Meta_StartUp () {
       // get SK from the
       // authentication server
       SC_server.SC_GetSessionKey(SK);
       ...
  }
  void Meta_MethodCall (...) {
       ...
       sign (request, SK);
       reply=SC_server.SC_method_call(...);
       check (reply, SK);
       ...
  }
};
```

```
reflect SC_SMO: CLIENT_MO;
```

SC_SMO

```
class SC_SMO {
protected:
  session_key SK;

public:
  void Meta_StartUp () {...}
  void SC_GetSessionKey (...) {...}
  ArPac SC_method_call (...) {
       ...
       check (request, SK);
       Meta_HandleMethodCall(...);
       sign (reply, SK);
       ...
  }
};
```

```
reflect SC_SMO: SERVER_MO;
```

Fig. 11. Secure communication metaobject classes.

```
class FT_CMO: public MetaObj {
  FT_SMO FT_server;
public:
  void Meta_StartUp ();
  void Meta_MethodCall (...);
};
```

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

Fig. 12. Basic interface for backward recovery mechanisms.

The object model and the use of a metaobject protocols encourage 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 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 method requested and defines the recover point before sending back the method execution result to `FT_CMO`. Both classes are inherited and specialized for all the mechanisms described hereafter. The metaobject protocol also provides means to access object attributes from the metalevel. The state information is considered here as the set of object attributes.

The fault tolerance metaobject interfaces given in Fig. 12 are a generalization of the leader-follower replication metaobjects already described in Section 4.3. The client metaobject interface mainly consists of 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 that 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: SERVER_MO` makes the `FT_SMO` instance an RPC-like server, waiting for request messages from remote clients. Initialization of the stack and registration in the corresponding service group is done by redefining the `Meta_StartUp` method.

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,

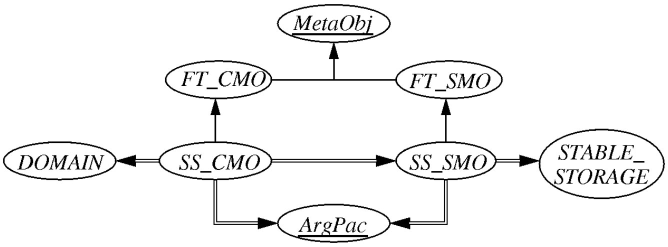


Fig. 13. Static diagram—recovery points on stable storage.

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.

5.3 Recovery Points on Stable Storage

In this mechanism, the current invocation (method id and args) and the server's state are saved to stable storage at the end of every method call. Errors are signaled to the client metaobject and recovery consists of choosing a valid site, creating a substitute server on this site, and, if necessary, resending the current invocation. The substitute's initialization consists of restoring the last invocation and state available on stable storage.

We introduce here two classes interfacing system services. STABLE_STORAGE provides methods to read and write data to stable storage (NFS in our experiments) in the form of ArgPac instances. DOMAIN keeps a consistent view of valid sites by interfacing to an error detection service in the fault tolerance subsystem (*x*AMp [26] in our experiments). It provides methods to check this view and to obtain the name of valid sites. Invocation and state are implemented with the Open C++ ArgPac predefined class which acts as a metalevel container for any type of argument. Classes SS_CMO and SS_SMO, respectively, implement the client and server parts of the mechanism and inherit, respectively, from FT_CMO and FT_SMO. These metaobjects are causally connected with group-based communication metaobjects, which enables SS_SMO to be seen by SS_CMO, from a modeling viewpoint, but also, in practice, through a simple composition link. This link hides the use of a proxy of SS_SMO dedicated to the remote interaction with SS_CMO and simplifies the design. Fig. 13 gives the static diagram of the stable-storage-based mechanism.

We now study two important execution scenarios: the definition of a recovery point at the end of a method execution and the recovery upon error detection. The former is implemented by FT_method_end in SS_SMO (on the server side) and invokes the Save method provided by STABLE_STORAGE. This scenario is illustrated by Fig. 14.

```
1. Meta_MethodCall of client metaobject
2. SS_CMO memorize the request
3. invoke FT_method_call
4. SS_SMO memorize the request
5. execute base-level method requested
6. SS_SMO memorize the reply
7. capture new base-level state
8. save request, reply and state on
    stable storage
```

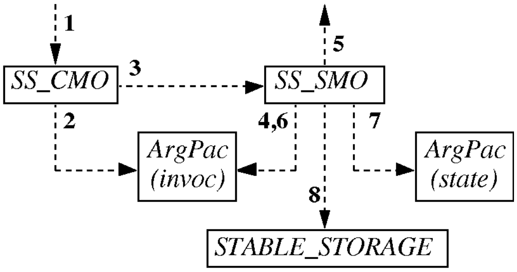


Fig. 14. Scenario—recovery point definition.

The recovery point definition mainly consists of capturing the base-level state and saving to stable storage. As this base-level state is made accessible by the metaobject protocol, the fault tolerance programmer does not need to call any application-specific routine.

The second scenario describes the recovery procedure. During method execution, an error can be detected by the error detection system service either before the new state is saved to stable storage or in between the time it is backed up and the time the server sends the reply to the client.[5] In both cases the error is signaled to the client metaobject, which starts the procedure described at the beginning of this section and illustrated by Fig. 15. As for the case of state capture, state restoration is handled through the metaobject protocol and, therefore, does not involve application-specific statements. State information is obtained by FT_SMO using Meta_HandleRead, a new replica is created, and its state is restored by Meta_HandleAssign.

```
1 error detection
2. get a valid site
3. create the substitute on this site
4. get last request, reply and state
   from stable storage
5. restore state
6. resend the memorized request
7. if the answer is not available,
   execute the request
```

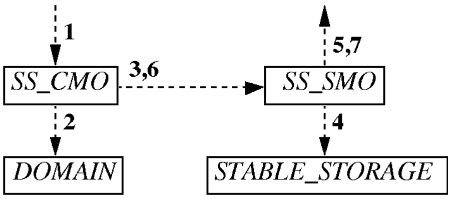


Fig. 15. Scenario—recovery procedure.

[5] The communication subsystem is assumed to be reliable.

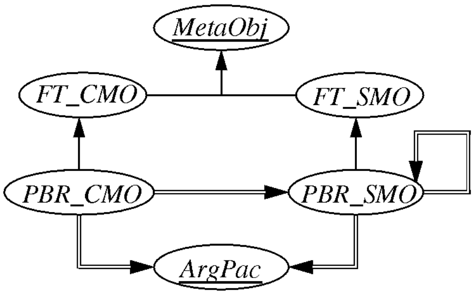


Fig. 16. Static diagram—primary-backup replication.

5.4 Primary-Backup Replication

In this replication strategy, all replicas of a server belong to the same atomic multicast group and, thus, receive the same input messages (e.g., invocation messages) in the same order. This assumption is not mandatory, but makes the design of the mechanism easier. It is enforced by the use of multicast communication protocols and groups in the communication metalevel. Among the replicas, only one (the primary) handles the client requests and checkpoints its new state at the end of every method executed to the other replicas (the backups). We assume that any primary error is detected by the backups which choose a new primary among themselves. This new primary restores the last checkpointed state and, if necessary, executes the current request before returning the reply to the client.

We introduce the PBR_CMO and PBR_SMO metaobject classes (derived from FT_CMO and FT_SMO, respectively) to implement the client and server parts of the primary-backup replication. Any backup can become primary, so the primary and backup behaviors are both implemented by PBR_SMO. The primary or backup status of the PBR_SMO metaobject is handled through a local variable. The primary can be seen as a client of the backups, and, therefore, PBR_SMO is a client of itself, as shown on Fig. 16. The replica interaction (i.e., the interaction among PBR_SMO instances) is made transparent through the use of communication metaobjects as for the client/server metaobjects interaction.

The recovery point definition scenario (Fig. 17) is similar to that of the stable storage mechanism. The main difference is that the captured state and invocation are transmitted to the backups, rather than saved to stable storage. This can also be handled using some of the metaobject protocol facilities and does not involve any application-specific programming for the fault tolerance programmer.

The recovery scenario (Fig. 18) is much more simple than in the case of the stable storage because, in the case of replication, reconfiguration (i.e., cloning a new replica on a valid site) is not mandatory.

Method execution at the base-level must be done only if the available reply does not correspond to the memorized *req*$_c$. It is handled by Meta_HandleMethodCall and, therefore, does not require any application-level programming. The PBR_SMO instance which handles recovery then becomes the new primary and is, therefore, in charge of executing subsequent client requests.

```
1. Meta_MethodCall of client metaobject
2. invoke FT_method_call
3. memorize the client request (reqc)
4. execute base-level method requested
5. memorize the reply
6. capture new base-level state
7. send request, reply and state to
   the backups
8. restore the base-level state
9. memorize request and reply
10. return the reply
11. eliminate reqc
```

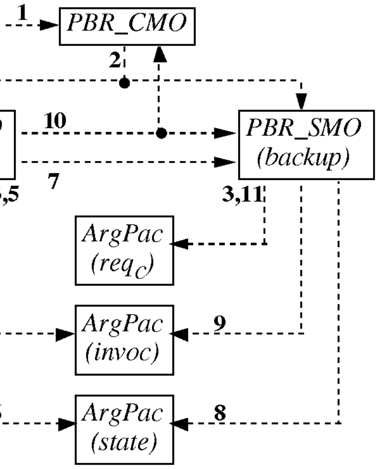


Fig. 17. Scenario—recovery point definition.

```
1. compare the memorized request and reqc
2. if necessary execute reqc at the base-
   level
3. return the reply
```

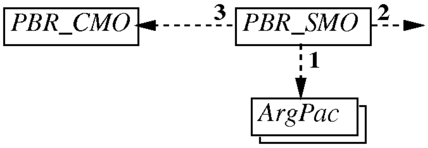


Fig. 18. Scenario—recovery procedure.

5.5 Leader-Follower Replication

This replication mechanism is inspired by the semiactive replication in Delta-4 [12]. In this protocol, all replicas process input messages, but only one (the leader) sends output messages. As in the case of primary-backup replication, all replicas of a server belong to the same atomic multicast group and receive the same messages in the same order. In our implementation, the leader first executes the request and then notifies it to the other replicas (the followers), which, in turn, execute the request. Only the leader returns the reply to the client. The metaobject classes obtained to implement this protocol are called LFR_CMO and LFR_SMO. The static diagram and the scenarios are quite similar to those obtained for primary-backup replication.

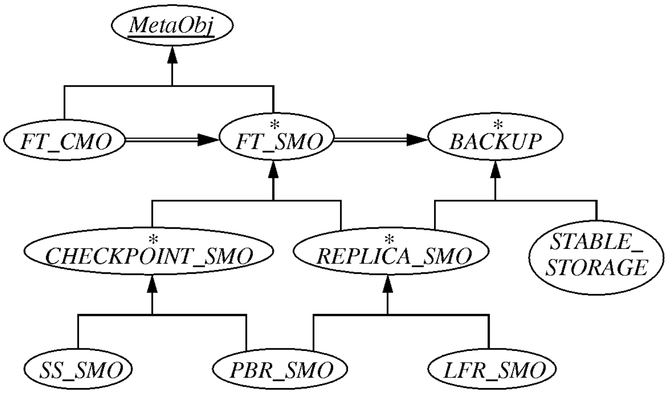


Fig. 19. Backward recovery metaobject class hierarchy.

5.6 A Complete Hierarchy of Metaobjects

The above mechanisms present several structural and behavioral common points. These can be factorized and reused in a hierarchy. In all these mechanisms, there exists a backup of the server's state, either on stable storage or as the state of a replica. We introduce the BACKUP class, which is used in any 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). All previously defined classes STABLE_STORAGE, PBR_SMO, and LFR_SMO inherit from BACKUP and define FT_update, respectively, as:

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

As for stable storage and primary-backup replication, a state capture is systematically performed upon method termination before updating the backup (stable storage file or backup replica). This can also be factorized within the definition of FT_method_end in a class CHECKPOINT_SMO, which inherits from FT_SMO. SS_SMO and PBR_SMO inherit from CHECKPOINT_SMO and extend FT_method_end, respectively, to save the captured state on stable storage or to send it to the group of backup replicas.

For both replication mechanisms, the recovery procedure and the handling of client requests by spare replicas (backups and followers) are similar. This common behavior can also be factorized 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 redefines FT_method_end so that the current invocation is notified to the followers. PBR_SMO multiply inherits from CHECKPOINT_SMO and REPLICA_SMO. These also handle reconfiguration by *cloning* a new replica. These structural and behavioral common points, factorized classes lead to the class hierarchy presented in Fig. 19.

6 General Discussion and Lessons Learned

6.1 Meeting the Initial Requirements

We discuss here the properties of the FRIENDS architecture and compare them with the initially required properties described in Section 2.1.

6.1.1 Ease of Use

Fault tolerance mechanisms can be added to an application simply by connecting the appropriate metaobjects to the application objects (using a few metaobjects declaration statements). In addition, as all fault tolerance metaobjects have the same interface (the MetaObj interface), the mechanism used can be changed simply by switching metaobjects. An application programmer can, thus, develop application objects locally as a single runtime unit and test the functional behavior. Once this first version works all right, it can be distributed by connecting remote objects to communication metaobjects, provided that each main application object is transformed into a single runtime unit. Then, stable storage metaobjects, for instance, can be inserted to make the application fault-tolerant. For performance reasons, e.g., the recovery time overhead, the application programmer can remove the stable storage metaobjects and replace them with leader-follower metaobjects. All these operations simply consist in connecting application objects and metaobjects, and do not involve significant changes in the application objects.

6.1.2 Generality

This aspect is, in fact, not tied to our application model, but rather, to the very nature of the mechanisms implemented, which can be used in a wide range of applications. In brief, the use of an object-oriented design method and of a metaobject protocol enables mechanisms to be plugged and adapted in application objects without any change to affiliation source code. Adaptation can take place by deriving new mechanisms from existing ones, but the interaction between application objects and metaobjects is fixed through the MOP.

6.1.3 Extensibility

New mechanisms can also be derived from existing ones by inheritance at the fault tolerance metalevel. For example, we can derive mechanisms based on delta-checkpointing and/or message logging, or mechanisms where the state is backed up only after writer method executions, or else various active replication strategies. These mechanisms can then be used as easily as the others by simple causal connection with application objects. More problematic is the extension of mechanisms when changing the underlying assumptions about the communication system. Suppose, on a given platform, no group management and multicast protocol is available and development of a primary-backup-like replication protocol is needed. Then, although possible, the reuse of the current primary-backup metaobjects may be limited, but this is not due to the metaobject approach.

6.1.4 Composability

Different types of fault tolerance mechanisms can be used in the same application by connecting different remote objects with different types of metaobjects. For example, a client object can interact with a remote object backed with a stable storage and with another remote object replicated according to the primary-backup model. The use of multiple metalevels also provides composability among different metafunctional properties. It is just a matter of declarations at compile time. No change in the fault tolerance metaobject source code has to be made when just distribution metaobjects are used or when security metaobjects are also used. Security metaobjects can be inserted in our architecture between fault tolerance and group-based communication metaobjects.

6.2 About Metaobject Development

A multilevel approach enables group communications to be disregarded a priori since they are provided by a (next) metalevel. This does not mean that the properties of group communication are ignored when developing metaobjects. It is a core aspect of the design of fault tolerance metaobjects that must be considered because properties of the communication system simplifies the fault tolerance solution.

However, the implementation of distributed fault tolerance mechanisms implies a strong interaction with group communication services. For instance, during recovery, communication metaobjects have to clear some message queues. This is why the design of communication metaobjects has to take into account some aspects of the fault tolerance strategies. This is also true for security metaobjects, e.g., propagating session keys to a new replica during recovery must be performed by security metaobjects. This means that precautions have to be taken when designing metaobjects; the context of usage has to be precisely defined and the role of companion metaobjects has to be considered. Nevertheless, from a practical viewpoint, the fault tolerance programmer is not responsible for handling calls to the group communication subsystem but is just involved in the declarations of object-metaobject links.

This architecture enables other group communication services to be used, provided they offer the same communication properties, say, atomic multicast of invocation messages. The interface to the underlying communication system can be different. If the properties provided by the communication system are weaker (say, global ordering of message delivery is not provided, such as for Chorus groups), then there is a strong impact on the design of the fault tolerance metaobjects because identical input message delivery cannot be assumed any more. Also, for security metaobjects the impact might relate to properties of the communication subsystem with respect to security; if messages are ciphered by hardware devices at a very low level, then ensuring confidentiality of message passing does not rely on the use of metaobjects anymore.

6.3 About the Programming Model

In this architecture, structuring is based on a stack of metaobjects and, thus, object invocations are trapped recursively by metaobjects in the stack on the client side and executed effectively by metaobjects in the stack on the server side. In fact, the same programming model is used at any level, either the application or any metalevel. Any remote object is accessed by a proxy in the client address space. The protocol between the proxy object and the remote object is handled by a pair of metaobjects (client and server). This simple programming model is very convenient because it is well understood by both application and system programmers. At the fault tolerance metalevel, for instance, the group of server replicas is seen as a single local object, “the backup,” by the client metaobject. The “group of backup replicas” is seen as a unique local object by the primary metaobject replica. This object is associated with a metaobject handling the communication within the “group of backups.” At the security metalevel, the authentication server is seen as a unique and local object also.

6.4 Intermediate Metalevels and Properties

The order of metalevels may vary according to the properties that must be guaranteed. Except for the last metalevel (responsible for communications) and the base-level (application object), a permutation of all intermediate levels might seem possible and sound. Although this is perfectly possible from a practical viewpoint (all metaobject classes share the same metaobject protocol interface), the ordering actually affects the properties that the final application has. For instance, if message integrity must also be guaranteed (e.g., by means of signature computation and verification) on information added during method invocations by the fault tolerance metalevel, then the security metalevel must be invoked after the fault tolerance metalevel. If message integrity must just be guaranteed on the application information during the method invocation (method and parameters), then one would expect the security metalevel to be put straight after the base-level. Suppose, then, that the intermediate levels are organized in the following order: The security metalevel is put straight after the application level, then the fault tolerance metalevel, leading, thus, to the following sequence (A; SC; FT; GD). When the primary takes a checkpoint of the application object, the access to the base-level state should go through the security metalevel. This implies that the security metalevel is able to propagate this access down to its base-level, which is not possible with the metaobject protocol that we use. Updating the backup state would also be a problem, because writing base-level attributes should again go through the security metalevel. This could be solved by slightly changing the MOP, making both the base-level *and* the bottom (application) level accessible, whatever the location of the metaobject in the stack. Another solution would be to implement the chaining of metaobjects using a flat linking, instead of the stack. With the current solution (A; FT; SC; GD) and the simple MOP used, reading or writing base-level attributes is easily achieved. Moreover, errors detected by the security metalevel, such as authentication error after multiple retries, can be delivered to the security base-level, i.e., the fault tolerance metalevel which handles recovery actions. Actually, according to the expected properties of the application, this is the only suitable sequence of intermediate levels if both fault tolerance and secure communication are required.

6.5 Role of Subsystems

Fault tolerance metaobjects implement part of the mechanisms, error processing mainly. They rely on subsystems implementing services shared by metaobjects. In FRIENDS, subsystems related to fault tolerance (FTS, GDS) are responsible for error detection, group communication, replication domains management, object cloning during reconfiguration, stable storage management. The reasons why these services are implemented as subsystems on a microkernel are the following:

- Some services often depend on specialized hardware components (e.g., watchdog timers for error detection, fail silent network attachment controllers for group communication, double memory boards with autonomous power supply for stable storage); the corresponding software needs to be run on dedicated components and needs access to the kernel address space (e.g., Chorus actors running in supervisor mode in the implementation of *x*AMp protocols within FRIENDS).
- Some other services need a global view of the distributed architecture (global view of available nodes, status and configuration for replication domains management, for instance) and remote facilities provided by the microkernel (remote creation microkernel system call for object cloning during reconfiguration, for instance).
- The partition between metaobjects and subsystems is also interesting for reusing off-the-shelf components; in FRIENDS, the *x*AMp software communication service was reused and implemented as a subsystem on Chorus; the Unix file system was reused as part of the Unix SCO subsystem available on Chorus Fusion 2.0.

All services provided by subsystems are often mandatory for the related metaobject libraries. They are not subject to change, although metaobjects implementing precise mechanisms can evolve quite easily.

6.6 Role of Metaobjects and Meta-Functional Properties

The role of metaobjects may vary according to the metafunctional property implemented at a metalevel.

6.6.1 Fault Tolerance

This was illustrated already in the previous sections for tolerating physical faults. Indeed, metaobjects for physical fault tolerance implement quite a lot of the mechanisms even if they rely on some mandatory basic services. This might be different for other types of faults such as software faults. In this case, most of the mechanisms can be implemented as metaobjects. Subsystems and low-level access to system software are useless in this case. The difference here is that metaobject behavior depends on information provided by the application objects, which was not really the case when handling physical faults. In the implementation of recovery blocks or N-version programming, for instance, the set of versions must be declared by the application programmer to the metalevel. The adjudicator routines must also be declared because they are application dependent. In summary, the application level must parameterise the metalevel. This may have an impact on the metaobject protocol.

6.6.2 Security

Providing security in a distributed system involves many different mechanisms, system structuring, and underlying properties. In FRIENDS, we have only considered authentication, confidentiality, and integrity of message passing, which have proven to be quite easily handled by metaobjects. The SCS subsystem is responsible here for user authentication and key management. The existence of such a subsystem enables some COTS authentication software to be integrated within FRIENDS , such as Kerberos. Another need for this SCS subsystem is that it must be responsible for permanent keys secure storage. Authorization aspects have not been considered yet. Nevertheless, it is clear that these aspects are totally dependent on global information (user rights) and system structuring (TCB, security kernel). Ideally, the verification of access rights according to a given security policy (either for confidentiality or for integrity) must be implemented by a security kernel that is part of the microkernel. Moreover, this security kernel must be tamper-proof, always invoked, and proven correct. The latter properties impose design and implementation approaches that have, to our viewpoint, nothing to do with metaobjects. The role of the metaobjects in this case is just to trap object invocations and deliver this invocation to the security kernel for authorization.

6.6.3 Summary

The main interest of the approach is to maximize mechanisms handled within metaobjects. As a consequence, mechanisms can be more easily updated since they are partially implemented at the user level. It is also a prime objective to efficiently implement metafunctional properties. This is why metaobjects and companion subsystems on a microkernel seems a promising solution. The role of metaobjects in the implementation of metafunctional properties may vary a lot; it is worth noting that, for some of them, the role of metaobjects is not significant (merely an additional indirection). From an abstract viewpoint, metaobjects and the corresponding subsystem (including the microkernel) could be understood as an implementation of a given metalevel. Although, from a conceptual viewpoint, subsystems are part of the metalevel, the motivation for this implementation was to benefit from the advantages of both a metalevel architecture (easy configuration of applications with respect to nonfunctional requirements) and microkernel technology (easy configuration of the OS).

6.6.4 Limits and Drawbacks.

Those identified so far essentially relate to the MOP that was used. To overcome a problem due to the static binding of application classes to metaobject classes, several names must be given to application classes defining the same behavior. In addition, Open C++ V1 provides limited metainformation and, thus, application objects have not been implemented using inheritance. However, the latter limit can be relaxed with a more powerful MOP providing control over the inheritance tree. Another point concerns the abstractions dealt with. The organization of the metaobject

type	Sparc Station IPC	Sparc Station IPX	Ultra Sparc 1
OS	SunOS 4.1.4	SunOS 4.1.4	Solaris 5.5
RAM	24 MB	32 MB	64 MB
clock	25 MHz	40 MHz	167 MHz
SpecInt92	13.8	21.8	252

Fig. 20. Workstations characteristics.

stack is possible because neither the security level nor the communication level need to access the application level. This is not typical of these abstractions and others, like soft real-time aspects, could be handled easily in this stack. Nevertheless, if two metalevels need to access the application level, then their respective behavior must be handled at a single metalevel. Here again, this limit could be relaxed with a richer MOP allowing systematic access to both the base-level and the bottom base-level. From a design viewpoint of fault tolerance metaobjects, it is also important to take into account the properties of the underlying group communication system.

Another lesson we have learned from this experience is that changing the order of levels in the stack whenever possible leads to different properties. This must be carefully analyzed when using several metalevels because such changes could lead to unexpected side effects, as shown above. Several other problems are also not easy to solve, but they are not, to our viewpoint, due to the use of metaobjects (handling multiple replies, are view changes to be considered as failures, etc.). Finally, the FRIENDS system today is very dependent on the language used (Open C++) and on its homemade object-oriented distributed support. An alternative approach, using an operational implementation of the MOP within an object-oriented run-time layer, would be language independent. CORBA-compliant layers and COTS (including microkernels) will be considered in the near future.

Dynamic binding of metaobjects was not implemented in FRIENDS today. Even if dynamic binding may make validation more difficult, this is possible with a reflective language approach. In this case, object and metaobjects are separated runtime entities. Any application object has a local simple metaobject (based on the MetaObj class in Open C++) which redirects actions to be done to the effective metaobject using local communication mechanisms. The programming interface for the application programmer is still the same. The implementation of metaobjects (recovery in particular) must be revised in this case because of the need to

1) access base level state information,
2) to share information between metaobjects.

6.7 Programming FRIENDS Applications

As shown by the example given in Section 4.2. (Fig. 9), writing applications on FRIENDS essentially leads to writing traditional C++ code. The only additional code needed is the MOP reflect declaration, which connects an object to a metaobject. In practice, several metaobjects can be used, as explained in the paper. So, when a client-server application requires only distribution, then the first and unique declaration corresponds to a distribution metaobject declaration in both the client and the server. When fault tolerance is required, then this declaration is replaced by the declaration of fault tolerance metaobjects. Distribution metaobject are, in this case, declared within fault tolerance metaobjects (see Fig. 10). From a practical viewpoint, those declarations can be included automatically in the program by an off-line configuration tool and can be updated without any user intervention at compile time.

Because of the language used, a few programming conventions have to be obeyed. For instance, for any class of objects C, the Open C++ preprocessor generates a reflective class (refl_C) when C is associated with a metaobject class. The reflective class must be used instead of the standard one when reflective objects are created. Because the Open C++ V1 MOP does not support inheritance, derived classes must be avoided. Finally, any FRIENDS reflective server object inherits an init method for initialization and a start method. The latter method activates the message reception loop at the communication metalevel.

7 Implementation and Performance Issues

7.1 A Prototype Implementation

Two prototypes of FRIENDS are now available, one on Unix and one on Chorus. Our measurements have been done with the first prototype on an Ethernet network of Sun IPC and IPX workstations with SunOS 4, using Open C++ version 1.2. We also used Sun Sparc Ultras running Solaris for the measurement of the Open C++ metaobject protocol overhead. Fig. 20 gives the characteristics of all these machines.

Several fault tolerance mechanisms (stable storage, primary-backup, and leader-follower replication in lock-step mode) have been implemented. GDS includes an object-oriented extension of the *x*AMp package (version 3.1) and the metaobject library provides a support for distributed applications. The library is comprised of several basic classes (handling creation / deletion of objects, group registration and naming, message management, etc.) used for implementing group-based distribution metaobjects. The size of the source code for all metaobjects libraries is about 25,000 lines of C++ (*x*AMp and libraries of cryptographic functions not included).

Fault tolerance mechanisms have been designed and implemented using inheritance, as described in Section 5. Metaobjects using a stable storage approach have been implemented first, then a primary-backup strategy, and, finally, a leader-follower strategy was implemented. Metaobjects implementing the leader-follower strategy have been derived from the previous one quite easily. In both replication examples, the inter-replica protocol was also implemented with a multilevel approach. In FTS, the Unix file system is used as a stable storage service and failure detection is performed by means of station failure signals provided by *x*AMp. Finally, a service manages system configuration and replication domains in a very simple way.

A first version of the metaobjects handling secure communications has been implemented, based on the Needham-Schroeder authentication protocol with secret keys. Thus, SCS contains today a simple implementation of a Needham-Schroeder authentication server used in our examples; this first implementation can easily be upgraded to be in accordance with the Kerberos authentication protocol.

The distributed application managing bank accounts, described in Section 4.2, has been developed. All kinds of metaobjects have been tested with this application in different experiments: First, only distribution metaobjects have been used, then secure communication and distribution metaobjects or fault tolerance and distribution metaobjects, and, finally, fault tolerance, secure communication, and distribution metaobjects. These various configurations were obtained by simply changing `reflect` associations between objects and metaobjects at compile-time; for easy use, the corresponding declaration statements are made including files and not directly visible in the user source code. In these experiments, we have also simulated physical faults (crash failure) and authentication faults (authentication error, session key expiration). All possible configurations have been tested and the whole FRIENDS prototype has been ported to our experimental platform of Chorus-based i486 PCs, requiring the port of the *x*AMp package and the Open C++ compiler.

7.2 Performance Issues

These platforms proved that the properties we expected could be obtained within the FRIENDS architecture and application model. The experimental platform was also used to establish some quantitative performance measures. The most important result is that the runtime execution overhead due to the use of a metaobject protocol is negligible with respect to the runtime execution cost of the mechanisms implementing metafunctional properties within the metaobjects. This result was foreseeable; our experiments confirm it in a quantitative way. We also give additional information concerning how much execution time is spent in each metaobject.

7.2.1 Overhead of the Metaobject Protocol

In this section, we compare the execution time of reflective Open C++ objects and normal C++ objects with respect to the following operations:

1) object creation / deletion and
2) method invocation with variable number and size of arguments.

Object creation, invocation, and deletion are empty operations. The method invocation was performed using either *m* arguments of 512 bytes or a unique argument of $n \times 512$ bytes, $0 \leq m, n \leq 8$. Experiments were done on Sun IPC, IPX, and Ultra workstations, and each result is the average obtained on 10^6 executions. The case of an *m*-argument method invocation is given in Fig. 21.

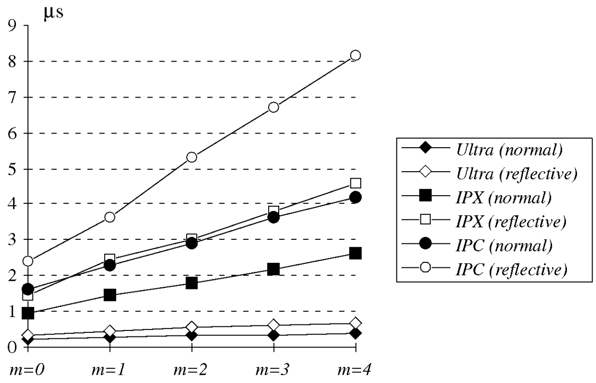


Fig. 21. Method call with m arguments.

Remote creation with fork/exec	7 ms
Joining and leaving an *x*AMp group	10 ms
Atomic multicast within an *x*AMp group	4–8 ms
Signature computation or verification	5–13 ms
Read operation on stable storage	6 ms
Write operation on stable storage	10–40 ms

Fig. 22. Average execution time of basic operations.

The overhead of a reflective object creation / deletion is approximately 200 percent; it includes metaobject creation and initialization, plus object / metaobject causal association. The overhead of a reflective method call does not depend on the size of the arguments but only on the number of arguments, with an irreducible overhead of 40 percent. The total overhead thus increases from approximately 40 percent (no argument) to 100 percent (four arguments). In terms of absolute execution time, this overhead corresponds to only few microseconds per call.

7.2.2 Individual Cost of the Mechanisms

We give here the individual costs of various basic mechanisms used in the different metaobjects. The measures given here have been made on Sun IPC and IPX workstations. Two sorts of costs can be distinguished: the cost of mechanisms used at object creation time and the cost of mechanisms used upon method invocation. When creating an object (replica) on a remote site, a new server process is launched with a fork / exec UNIX system call; the client and all the server replicas must join the same service group in order to communicate. When invoking a method on a remote object, a request and a reply message are multicast in the service group; additional messages may be sent for the inter-replica protocol, or write operations may be executed on the stable storage (NFS in our experiments). Moreover, the costs of signature computation and verification can be added to these costs for each message multicast. Measurements of these basic operations on a Sun IPC are reported in Fig. 22; these values are the average or range obtained on a thousand executions. Some results depend on the size of the input data (atomic multicast, writing to stable storage, signature operations).

The costs reported in Fig. 22 are almost all of a few milliseconds. When building a fault-tolerant application, these

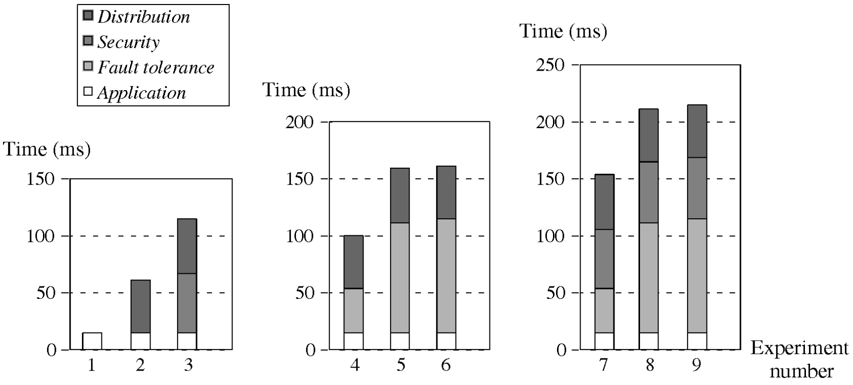


Fig. 23. Distribution of execution time.

basic operations are unavoidable and their costs are several orders of magnitude higher than the cost of the Open C++ metaobject protocol (a few microseconds) used for activating them. This claim is substantiated in the next section.

7.2.3 Execution Time Decomposition

Here, we study how execution time is distributed among the various types of metaobjects used within an application. Extensive experiments were made on the Sun IPC and IPX workstations using all possible combinations of different mechanisms within a simple client-server application. The total execution times were measured in each case, which allowed their comparison for different mechanisms.

Fig. 23 shows the measures obtained for a set of invocations within the banking application; each value reported is the average obtained on 1,000 runs. In this figure, the overhead resulting from the use of a given type of metaobject (fault tolerance, security, or distribution) is indicated by a specific gray level. Nine cases were tested: the application (1), the application + distribution metaobjects (2), the application + security + distribution metaobjects (3), the application + fault tolerance + distribution metaobjects (4-6), the application + fault tolerance + security + distribution metaobjects (7-9). All types of mechanisms for fault tolerance were used: stable storage (4 and 7), primary-backup (5 and 8), and leader-follower (6 and 9).

In all these cases, the execution of the application is short with respect to the execution of the mechanisms; however, the execution time of the mechanisms is fixed and does not depend on the execution time of the application, so it may become relatively small for higher application method execution times. The overhead resulting from the use of distribution and security metaobjects is the same in all cases. The overhead induced by replication protocols (cases 5, 6, 8, and 9) is significant, but this is probably due to a rather poor implementation of distribution metaobjects. Finally, leader-follower replication (cases 6 and 9) has been implemented in lockstep mode, so its execution time is not better than the execution of the primary-backup protocol.

8 Conclusion

This paper summarizes several years of research and the development of a metaobject architecture for building dependable systems. It describes the FRIENDS system, an experimental platform that enables object-oriented and metalevel programming to be used for implementing meta-functional properties. This architecture also takes advantage of microkernel technology as a way of structuring the underlying operating system to improve flexibility and efficiency. According to the questions asked in the beginning of this paper, the lessons we have learned from this work are rather positive. Although the properties we advocate are still limited, either by the language used or by its metaobject protocol, we believe that the new trend opened by metalevel architectures in our field is very promising. Clearly, our experiments substantiate and illustrate ideas that are now being recognized by the dependability research community.

The FRIENDS system is a very suitable platform for experimenting with object-orientation and metalevel programming in various directions. Among them, we have identified the following: The first one is reusing metaobjects for implementing new mechanisms with respect to various fault assumptions and for evaluating how much can be reused and what the impact on the existing set of classes is. The development of a more powerful metaobject protocol will be done for a more efficient implementation of existing mechanisms and, also, for the implementation of others meta-functional properties, mainly related to real-time. We are currently improving the underlying distributed object-oriented support using CORBA layers and implementing a metaobject protocol in this support. Some experiments have already been done using Open C++ V2. Another interesting aspect is to analyze alternative ways of designing the metalevel by different chaining of metaobjects and to allow dynamic connection between objects and metaobjects. Engineering the metalevel is a long term activity. Considering more advanced computational models and evaluating to what extent this architecture enables more easy validation are two future directions of this work within the DeVa project.

Finally, the approach presented here should be a good step towards the idea of *business objects* in and for dependable computing. Various programmers specialized in various technical fields can cooperate in an attractive fashion using the approach presented here; this was observed by the people who contributed to the implementation of the various prototypes.

ACKNOWLEDGMENTS

The authors wish to thank our FRIENDS, Brian Randell, Robert Stroud, and Zhixue Wu in the PDCS and, now, DeVa projects, who participated in the elaboration of these ideas, Paulo Veríssimo and Henrique Fonseca for xAMp, Shigeru Chiba for Open C++, and Marc Rozier for his advice on porting the xAMp package on top of Chorus. Vincent Nicomette, Françoise Cabrolié, Frédéric Salles, Cyril Delamare, Frédéric Melle, Arnaud Ladrech, Tanja van Achteren, Gregory Lajon, and Marc-Olivier Killijian have contributed to the development of FRIENDS and must be acknowledged. The authors wish to thank Jean-Claude Laprie and Robert Stroud for their pertinent comments on an early version of this paper and the referees for their suggestions that helped very much in the preparation of this final version. This work has been partially supported by the DeVa Esprit project no. 20070.

REFERENCES

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

[2] R.J. Stroud, “Transparency and Reflection in Distributed Systems,” *ACM Operating Systems Review*, vol. 22, no. 2, pp. 99-103, Apr. 1993.

[3] B. Garbinato, R. Guerraoui, and K. Mazouni, “Implementation of the GARF Replicated Objects Plateform,” *Distributed Systems Eng. J.*, vol. 2, pp. 14-27, Mar. 1995.

[4] J.C. Fabre, V. Nicomette, T. Pérennou, Z. Wu, and R.J. Stroud, “Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming,” *Proc. FTCS-25*, pp. 489-498, Pasadena, Calif., June 1995.

[5] J.C. Fabre and T. Pérennou, “FRIENDS: A Flexible Architecture for Implementing Fault-Tolerant and Secure Distributed Applications”, *Proc. EDCC-2, Lecture Notes in Computer Science*, vol. 1,150, pp. 3-20, Taormina, Italy, Oct. 2-4, 1996.

[6] P. Maes, “Concepts and Experiments in Computational Reflection,“ *Proc. OOPSLA ’87* , pp. 147-155, Orlando, Fla., 1987.

[7] G. Kiczales, J. des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[8] A. Paepcke, “PCLOS: Stress Testing CLOS—Experiencing the Metaobject Protocol,” *Proc. OOPSLA ’90*, pp. 194-211, 1990.

[9] Y. Honda and M. Tokoro, “Soft Real-Time Programming Through Reflection,” *Proc. Int’l Workshop Reflection and Metalevel Architecture*, pp. 12-23, Nov. 1992.

[10] S. Chiba and T. Masuda, “Designing an Extensible Distributed Language with Metalevel Architecture,” *Proc. ECOOP ’93, Lecture Notes in Computer Science*, vol. 707, pp. 482-50.1, Kaiserslautern, Germany: Springer-Verlag, 1993.

[11] G. Kiczales and J. Lamping, “Issues in the Design and Specification of Class Libraries,” *Proc. OOPSLA ’92*, pp. 435-451, 1992.

[12] D. Powell, “Distributed Fault Tolerance—Lessons Learnt from Delta-4,” *Hardware and Software Architecture for Fault Tolerance: Experiences and Perspectives*, M. Banâtre and P.A. Lee, eds., *Lecture Notes in Computer Science*, vol. 774, pp. 199-217. Springer Verlag, 1994.

[13] K.J. Birman, “Replication and Fault Tolerance in the Isis System,” *ACM Operating Systems Review*, vol. 19, no. 5, pp. 79-86, 1985.

[14] D. Detlefs, M.P. Herlihy, and J.M. Wing, “Inheritance of Synchronization and Recovery Properties in Avalon/C++,” *Computer*, vol. 21, no. 12, pp. 57-69, Dec. 1988.

[15] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington, “An Overview of the Arjuna Distributed Programming System,” *IEEE Software*, vol. 8, no. 1, pp. 66-73, 1991.

[16] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa, “Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently,” *Proc. OOPSLA ’92*, pp. 127-144, 1992.

[17] D. Powell, “Failure Mode Assumptions and Assumption Coverage,” *Proc. FTCS-22*, pp. 386-395, Boston, 1992.

[18] S.E. Mitchell, A.J. Wellings, and A. Burns, “Developing a Real-Time Metaobject Protocol,” *Proc. IEEE Workshop Object-Oriented Real-Time Dependable Systems*, 1997.

[19] J. Xu, B. Randell, and A.F. Zorzo, “Implementing Software Fault Tolerance in C++ and Open C++: An Object Oriented and Reflective Approach,” *Proc. CADTED ’96* , pp. 224-229. Beijing, China: Int’l Academic Publishing, 1996.

[20] R.J. Stroud and Z. Wu, “Using Metaobject Protocols to Implement Atomic Data Types,” *Proc. ECOOP ’95, Lecture Notes in Computer Science*, vol. 952, pp. 165-189, 1995.

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

[22] Y. Yokote, “The Apertos Reflective Operating System: The Concept and Its Implementation,” *Proc. OOPSLA ’92*, pp. 414-434, 1992.

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

[24] R.M. Needham and M.D. Schroeder, “Using Encryption for Authentication in Large Networks of Computers,” *Comm. ACM*, vol. 21, no. 12, pp. 993-999, Dec. 1978

[25] K. Waldén and J.M. Nerson, “Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems,” *The Object-Oriented Series*. Prentice Hall, 1995.

[26] L. Rodrigues and P. Veríssimo, “xAMp: A Protocol Suite for Group Communication,” *Proc. SRDS-11*, pp. 112-121, 1992.

Jean-Charles Fabre received his Doctorate in computer science in 1982 from the University of Toulouse, France. He was first involved in the Chorus project at INRIA and was responsible for the design and implementation of fault tolerance strategies in the Chorus distributed architecture. Since 1984, he has been with the LAAS-CNRS in Toulouse, working in the Fault Tolerance and Dependable Computing Group. His past and current research interests concern distributed algorithms, implementation validation by fault-inject, and fault and intrusion-tolerance in distributed systems. Today, his activity is concerned with object-oriented development of fault and intrusion-tolerant systems and validation of COTS microkernels by fault-injection. The author or coauthor of more than 40 publications in conference proceedings, journals, and books, he is currently involved in two European Esprit projects, DeVa and Guards.

Tanguy Pérennou received his MS (1992) and Doctorate (1997 in computer science from the National Polytechnic Institute of Toulouse (INPT), France. His main research interests include object-oriented programming, distributed systems, fault tolerance, and metaobject protocols. He is currently working as an engineer at the French Air Navigation Research Center (CENA).