Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming

Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou

LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse cedex, France

Robert J. Stroud, Zhixue Wu

Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

Abstract

This paper shows how reflection and object-oriented programming can be used to ease the implementation of classical fault tolerance mechanisms in distributed applications. When the underlying runtime system does not provide fault tolerance transparently, classical approaches to implementing fault tolerance mechanisms often imply mixing functional programming with non-functional programming (e.g. error processing mechanisms). The use of reflection improves the transparency of fault tolerance mechanisms to the programmer and more generally provides a clearer separation between functional and non-functional programming. The implementations of some classical replication techniques using a reflective approach are presented in detail and illustrated by several examples, which have been prototyped on a network of Unix workstations. Lessons learnt from our experiments are drawn and future work is discussed.

1 Introduction

The implementation of fault tolerant distributed applications largely depends on the computing environment which is available. The ideal case is when the underlying operating system provides fully transparent error processing protocols such as in Delta-4 [17, 19]. However, when the operating system does not provide such facilities, the application programmer is forced to integrate in the functional part of the application statements to initialise or invoke appropriate non-functional mechanisms for error-processing. This can be done using library calls to pre-defined mechanisms embedded in a specific environment such as in Isis [6]. Another approach, used by systems like Avalon/C++ [10] and Arjuna [22], consists of using properties of object-oriented languages, such as inheritance, to make objects recoverable. However, even if the object model seems appropriate for introducing fault tolerance into applications, there are significant problems with such an approach for implementing various replication techniques in distributed applications, and we show that the use of reflection is a more promising approach. Reflection [14] enables functional programming to be separated transparently from non-functional programming, i.e., in the present paper, programming of fault tolerance mechanisms. Reflection allows programmers to observe and manipulate the computational behaviour of a program. In particular, in object-oriented languages, this property enables some operations such as object creation, attribute access and method invocations to be intercepted at a meta-level and this ability will be used for implementing fault tolerance mechanisms. The idea of using a meta-level to hide the implementation of non-functional requirements such as dependability and distribution transparency from the application programmer is not new. For example, various authors have proposed using the CLOS meta-object protocol to add attributes such as persistence and concurrency control to application objects [3, 16], [25] has argued that reflection is an appropriate way to address distribution transparency and [1] has described the implementation of dependability protocols using reflection in an actor-based language. Dependability can thus be provided transparently from the programmer's point of view and dependability-related facilities can be reused in multiple applications.

The contributions of this paper are two-fold: (a) to provide a comparison of different approaches to implementing fault-tolerance with respect to the degree of transparency for the application programmer, and (b) to provide detailed case studies showing how meta-level programming can be used to implement various replication strategies transparently in the reflective object-oriented language Open-C++. The latter is illustrated by presenting the implementation of the following three replication techniques used in our examples: *passive replication, semi-active replication* and *active replication with majority voting*. Section 2 discusses various ways of using fault tolerance mechanisms in the development and implementation of distributed applications. Programming style is underlined in each case. Section 3 provides a brief overview of reflection in object-oriented languages and introduces the reflective capabilities of Open-C++, the language that was used in our experiments. Section 4 briefly presents the distributed processing model used and details the reflective implementation of the three replication

0731-3071/95 $4.00 © 1995 IEEE

techniques that are under investigation. Section 5 mainly describes implementation issues of meta-objects for further development.

2 Approaches to programming fault tolerance

The aim of this section is to describe several approaches and programming styles that have been used in practice to add redundancy to applications for fault tolerance. These approaches will be considered for programming fault tolerance in distributed applications. A distributed application will be seen here as a collection of distributed software components (objects, processes) communicating by messages. Various error processing techniques may be used either when the runtime units (corresponding to design objects) are created or at an early stage during the design of the application in terms of objects. They can be based either on software component replication or on other approaches such as checkpointing to stable storage.

Three approaches can be followed for implementing error processing: (i) in the underlying runtime systems through built-in error processing facilities, for instance to replicate software components, (ii) in the programming environment through predefined software constructions and libraries, and (iii) in the application design environment through properties of the programming language. These three approaches are discussed and will be used as a basis for comparing various programming styles and implementation approaches. We will also underline the limits of the role of the application programmer in each case.

2.1 System-based fault tolerance

In this approach, the underlying runtime system may offer a set of transparent error processing protocols, for instance based on replication as in Delta-4 [17]. Delta-4 provides several replication strategies: passive, semi-active and active replication. They rely on detection mechanisms and voting protocols implemented by the underlying multicast communication system. The error processing protocol is selected at configuration time according to the failure mode assumptions that can be made about the available nodes of the distributed computing architecture and the coverage of these assumptions [18].

Passive replication can be supported by the system, in particular for the management of replicas, but often involves the programmer in defining checkpoints [5]. Nevertheless, it has been shown, in particular in Delta-4, that checkpointing can be automatically issued by the underlying runtime system [24]. This approach enables non-deterministic behaviour of the software component. Semi-active replication enables several replicas to process input messages concurrently. Input messages are delivered by the underlying multicast communication system, thus providing input consistency. In this model, non-deterministic behaviour is possible but may require the programmer to define synchronisation checkpoints to enforce consistency of replicated processing. In some circumstances, synchronisation can be solved by the communication system [4]. Finally, when deterministic behaviour can be ensured, several active replication techniques can be defined which are transparent to the application programmer. In Delta-4, several inter-replica protocols (IRp) are available as part of the underlying multicast communication system [8]. When the deterministic assumption is valid, then the same component can be used with either a semi-active or any active replication technique without any change in the source code. The advantage of this approach is that it provides transparency in most cases for the application programmer; the main drawback is that it needs a specific runtime system and support environment.

2.2 Libraries of fault tolerance mechanisms

This approach is based on the use of pre-defined library functions and basic primitives (i.e. a *tool-kit*). A good example of this approach is Isis [6, 7]. The prime objective of this environment was not initially the implementation of fault tolerant applications, but rather the development of distributed applications based on the notion of process groups. With respect to fault tolerance issues, the underlying assumption is that nodes are fail-silent.

In Isis, a specific software construct called a *coordinator-cohort* can be used to implement fault tolerant applications, in particular based on passive replication. This generic software construct enables the computation to be organised using groups of processes (tasks) according to various objectives: parallel computations, partitioned computations, replicated computation for fault tolerance. In the implementation of passive replication, the updated states of the primary copy (coordinator) must be sent to the standby copies (cohorts). When the coordinator fails a new coordinator is elected and loaded with the current state of the computation [12]. A new member can be inserted in the group of replicas and its state initialised using a state transfer primitive. All this must be taken into account when programming the replicas. Different checkpointing strategies are left open to the application programmer.

Other fault tolerance mechanisms based on active replication can be defined using group management facilities and multicast communication protocols. A *token* mechanism may be used to determine a leader in the group

of replicas which is responsible for sending back the reply to the client (similar to semi-active replication in Delta-4). This is done by the application programmer, even if it can be hidden in library calls.

The main difference with respect to the previous approach is that in this case, error processing and application programming is done at the same programming level using specific programming constructs. This means that specific function calls (state transfer, synchronisation, sending results, voting) must be introduced into the application programs at appropriate points, for instance for sending updates (passive replication), token management (semi-active replication) or decision routines (active replication). In other words, such an approach provides *visible* error processing, whereas it was *invisible* at the programming level in the previous case. Another example of the use of a library for programming fault tolerant applications can be found in [11]. Nevertheless, the advantage of this approach is that the application programmer can tailor and optimise his own fault tolerance mechanisms. The main drawback is that functional and non-functional programming are mixed together, and may contradict reusability. The approach is not transparent to the application programmers and may impose a specific runtime environment.

2.3 Inheritance of fault tolerance mechanisms

The previous two approaches do not rely on any particular property of the programming language since they are based on appropriate mechanisms provided either by the underlying operating system or by a specific environment. The approach described in this section and also the reflective approach described in this paper take advantage of object-oriented properties for providing error processing features to applications.

The approach based on inheritance consists in defining the fault tolerance technique in pre-defined system classes that are responsible for the implementation of a given solution. The idea is to use the notion of inheritance to derive a fault tolerant implementation of objects. This solution consists in fact in making inheritable non-functional characteristics (persistence and recoverability), using appropriate system classes and programming conventions. This type of solution has been successfully used in particular in Avalon/C++ [10] and in the Arjuna project [22]. A class can be declared as "recoverable"; this declaration means that any instance of this class will perform some error processing, provided that some definitions are given by the class designer (virtual function definition, function overloading). In Arjuna, for instance, a recoverable class is derived from the pre-defined system class `StateManager` which is responsible for providing persistence (checkpointing to stable storage) and recovery mechanisms; the application programmer must define the virtual functions `save_state` and `restore_state` for a recoverable class [2]. As with a passive replication mechanism, the computation is done by a primary object, unless a failure occurs.

One might also consider inheritance for implementing different error processing techniques, based on active replication, for instance. It seems that other system classes, like `StateManager`, could be defined to provide replicated processing. However, there would be significant problems with such an approach. Error processing techniques based on active replication would require a mechanism for providing replicated method invocations and synchronising replicas on method invocation. Overriding the creation of objects can also be useful for creating several object replicas on different sites. These cannot be transparently achieved using inheritance. The essential difficulty with this approach is that inheritance does not allow access to the internal structure of objects and redefinition of the fundamental mechanisms of an object-oriented language (e.g. method invocation).

2.4 Summary and conclusions

The systems that we have described do not all use the same fault tolerance techniques. Nevertheless, they illustrate three different approaches for implementing fault tolerant applications. In each case, the role of the programmer is different, according to the degree of transparency and separation of concerns provided by the approach.

In the first case, the error processing mechanisms are provided by the underlying system and transparency and separation of concerns can be achieved. However, this approach lacks flexibility. In the second case, the environment provides library functions that enable the programmer to define his own error processing mechanisms. Transparency and separation of concerns are not achieved due to specific function calls that must be introduced in the program. With the last approach, as shown by the examples, inheritance can be used to add fault tolerance properties to object-oriented applications. Separation of concerns can be achieved but transparency is not totally achieved, because some programming conventions are required.

Our interest in this paper is to show how the object model and related properties can be used for programming various classical replication techniques to implement fault tolerant distributed applications transparently. Inheritance seems limited from this viewpoint: inheritance does not enable the underlying operations of the object model

(creation, invocation) to be redefined. Thus, inheritance cannot be used to take advantage of the object structuring for implementing replicated processing. The reflective approach which is described in this paper solves part of this problem since reflection provides at least access to internal object operations.

3 Reflection and object-oriented programming

In this section we introduce the concept of reflection in the environment of object-oriented programming, and give a brief description of Open-C++, the language that was used in our experiments.

3.1 Reflection in object-oriented languages

Reflection is the process by which a system can reason about and act upon itself. A reflective computational system is a computational system which exhibits reflective behaviour. In a conventional system, computation is performed on data that represents entities that are external to the computational system. However, a reflective computational system must contain data that represents the structural and computational aspects of the system itself. Moreover, it must be possible to access and manipulate such data from within the system itself, and more importantly, such data must be causally connected to the actual behaviour of the system: changes in the data must cause changes in the behaviour of the system and vice versa. Unlike a conventional system, a reflective system allows users to perform computation on the system itself in the same manner as in the application, thus providing users with the ability to adjust the behaviour of the system to suit their particular needs.

B. Smith invested the power of computational reflection in the environment of 3-Lisp [23]. P. Maes proposed a meta-object approach to implementing reflective systems in the framework of object-oriented computing [14]. Each object x is associated with a *meta-object* $\hat{}x$ that represents both the structural and computational aspects of x. $\hat{}x$ contains the meta-information of the object x: its structure and its way of handling operation invocations. By making an object x causally connected with its meta-object $\hat{}x$, a system can ensure that any change to $\hat{}x$ will automatically be reflected to x. Thus the structure and behaviour of x can be adjusted by modifying its meta-object $\hat{}x$. Since a meta-object is just another object, it can be manipulated in the same manner as a normal object. In class-based object-oriented languages, each meta-object is an instance of a meta-level class that defines its structure and behaviour, but in the rest of this paper, we will tend to talk about meta-objects rather than meta-level classes, thus emphasising the run-time aspects of the meta-object approach.

The meta-object approach has been used in many application areas: debugging, concurrent programming [15] and distributed systems [9]. A very successful example is the meta-object protocol in CLOS [13]. This provides a new approach to designing programming languages. By using the technology of reflection and object-oriented programming, CLOS gives programmers the ability to incrementally modify the language's behaviour and implementation to meet their particular requirements. The relation of reflection to object-oriented programming is crucial to the meta-object approach. Reflection makes it possible to open up a system implementation without revealing unnecessary implementation details, and the techniques and features of object-oriented programming make reflection practical to use. In particular, inheritance makes it easy to adjust the behaviour of objects incrementally.

The use of meta-level programming makes it possible to separate functional components from non-functional components in a system transparently [25]. If non-functional components can be implemented in an application-independent fashion, they are potentially usable across a wide range of possible problem domains. There are three tangible benefits in taking the meta-object approach to implementing fault tolerant mechanisms. Firstly, the separation of functional and non-functional components makes it possible for the realisation of non-functional requirements to be transparent rather than intrusive as far as the application programmer is concerned, thus solving the problems associated with traditional techniques for implementing fault tolerance mechanisms (assuming that system-based fault tolerance is not available). Secondly, relying on meta-objects to deal with a wide range of user requirements allows the basic implementation of a fault tolerant application to be simpler and thus easier to analyse with respect to its correctness. Thirdly, permitting each object to have its own meta-object makes it possible for an application to apply different strategies for different objects according to their characteristics. These features will be illustrated in the remainder of the paper.

3.2 The example of Open-C++

Reflection was described generally in the last sub-section. In this sub-section, we introduce a reflective object-oriented programming language based on the meta-object approach, Open-C++, used to describe the examples.

Open-C++ [9] is a C++ pre-processor that provides the programmer with two levels of abstraction: the base-

level, dedicated to traditional C++ object-oriented programming, and the meta-level which allows certain aspects of the C++ programming model to be redefined. For example, at the meta-level, one can redefine the general behaviour of a base-level class: how it handles method calls, how it reads or writes its member variables, what happens at instance creation and deletion time. Each instance of a reflective base-level class is controlled at run-time by its meta-object. The association of a base-level class and a meta-level class is made at compile-time by the Open-C++ pre-processor.

Programming the meta-level boils down to programming C++ classes since meta-objects are just instances of traditional C++ classes. Meta-level classes all inherit (directly or indirectly) from the predefined `MetaObj` class. They can redefine the methods describing creation, deletion of an object, method invocation, etc. In Open-C++, the control of base-level object operations is realised via traps towards the related meta-object. For example, the handler associated with a base-level method call is a virtual method belonging to the class `MetaObj` called `Meta_MethodCall`, as shown in Fig. 1. It is possible for the application programmer to choose which attributes and methods are reflective.

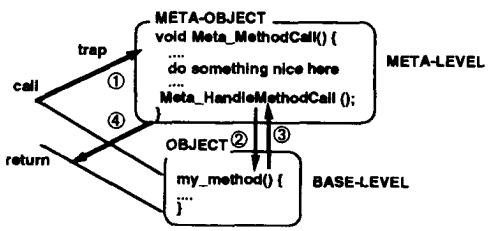

Fig. 1. Invocation trapping

When a reflective method is called at the base-level, the call is trapped and handled at the meta level by `Meta_MethodCall` (①). This meta method makes it possible to redefine the semantics of calling a method at the base-level. Usually, `Meta_MethodCall` invokes the application method from the meta level using another meta operation, `Meta_HandleMethodCall` (②③), but it may also perform some extra processing before or after calling the application method and perhaps not even call the application method directly at all. At the end of `Meta_MethodCall`, any results are returned to the caller as if for a normal method call (④).

The reflective attributes and methods of a base-level Open-C++ class are declared using a "`//MOP reflect:`" clause. For example in Fig. 2, the class `MyClass` has a reflective method `g()` and a reflective attribute `x`. These are the only attributes and methods that can be controlled by the meta-object associated with an instance of `MyClass`. The association of a class with a meta-level class is expressed using a "`//MOP reflect class`" clause. For example, in Fig. 2, the meta-level class for `MyClass` is declared to be `MyMetaObj`. Note that reflection is not completely transparent in Open-C++. Instead, the application programmer is required to use a special reflective version of the original application class which Open-C++ generates automatically. Thus, a reflective object of type `MyClass` is declared to be of type `refl_MyClass` and not `MyClass`.

```
/* Declaration of a class with reflective members */
   class MyClass {
   public:
      f();
   //MOP reflect:
      g();       /* public reflective method */
   protected:
      int i;
   //MOP reflect:
      float x;   /* protected reflective attribute */
   };
/* Association of a class with a meta-level class */
   //MOP reflect class MyClass : MyMetaObj;
      /* any object of class refl_MyClass (not MyClass!) */
      /* will be controlled by a meta-object of class */
      /* MyMetaObj */
/* Declaration of a reflective object */
   refl_MyClass MyObject;   /* reflective object */
```

Fig. 2. An Open-C++ class of a reflective object

Although Open-C++ supports meta-level programming, it only provides a limited model of reflection. First, it does not support structural reflection, i.e., the ability to make changes to the structure of an object by modifying its meta-object. Second, only limited computational reflection is supported in Open-C++: a meta-object in Open-C++ can control method calls and variable accesses. Third, the binding between objects and meta-objects in Open-C++ is made at compile time and cannot be changed subsequently. This means that the behaviour of an object in Open-C++ is determined statically and cannot be changed dynamically. Most of the limitations of Open-C++ arise from the fact that it is implemented by a pre-processor. To solve the above problems, a good cooperation between the pre-processor and the compiler must be established.

Although Open-C++ provides very limited reflection, its meta-level programming model provides the ability to separate applications into functional and non-functional levels that is most important to our investigations.

4 Meta-objects to support replication

In this section, we present a number of case studies that illustrate how a reflective approach can be used to implement a range of different replication techniques, namely passive, semi-active and active replication. Each replication technique will be implemented by a different meta-object. The runtime association of an object with a meta-object implementing a particular replication strategy enables the application programmer to arrange for

application objects to be replicated transparently. The details of how the replication mechanism is implemented are hidden at the meta-level and do not appear in the source code for the application.

The replication mechanisms we consider here follow the principles given in Section 2.1 but have been simplified for our experiments. We will present possible meta-objects for each technique, describing the implementation of passive replication in some detail. We will then discuss how this approach can also be used for semi-active replication, and how meta-objects can be used to support active replication with majority voting. Atomic multicast and failure detection are useful basic system services, but the implementation of the meta-objects described in this section does not rely on such services; we will return to this issue later in Section 5.

We consider a distributed application designed as a set of objects. From a distributed point of view we suppose that objects interact following the classic client-server model. For clarity, we will describe the inter-replica protocol implementation with one client and one replicated server. The details of possible inter-replica protocols for the proposed replication techniques are beyond the scope of this paper and can be found for instance in [4, 8, 24].

Distribution can be handled at either the meta-level or the base-level. For our first two replication examples, we chose to implement distribution at the base-level using client and server stubs. A server is composed of a stub that manages communications with the client and a "reflective object" that encapsulates the state of the server. The reflective object is managed by the server and is an instance of a base-level class associated with a meta-level class that implements a particular replication mechanism (e.g. passive replication). The server (a Unix process in the current implementation) encapsulates the object from a runtime viewpoint. When a server receives requests from its client via the stub, it calls the corresponding methods of the reflective object to meet the requests. These methods are intercepted at the meta-level as appropriate and dealt with according to the particular replication mechanism implemented by the meta-object associated with the application object representing the server's state.

For the last replication example distribution is handled at the meta-level. The structure of the client and the server is rather different in this case and communication stubs at the base-level are not used. This aspect of our design will be illustrated in Section 4.3.

4.1 Passive replication

The application is composed of a client, a primary server and one backup replica server. Client requests are processed by the primary server, and upon completion of an operation that updates its state, the server sends the new state to the backup replica. When the primary server crashes, the backup replica takes over the responsibility of providing continued service to the client and a new backup replica is started.

Base-level. In order to use passive replication for a particular application object, the application programmer must associate that object's class with the meta-level class `Passive_Repl_MetaObj` which is responsible for implementing the passive replication strategy. The application programmer must also decide which methods of the application class should be reflective - typically those methods which modify the state of the application object.

```
/* Class definition */
   class Medical_Info {
   public:
      void Read_Info(...);
   //MOP reflect:
      void Write_Info(...);        /* reflective method */
   protected:
   //MOP reflect:
      Medical_Record med_rec;    /* reflective attribute */
   };
/* Association with a meta-object */
   //MOP reflect class Medical_Info : Passive_Repl_MetaObj;
/* Definition of object methods */
   void Medical_Info::Read_Info(...) {
      <method statements>
   }
   void Medical_Info::Write_Info(...) {
      <method statements>
   }
/* Declaration of the object */
   refl_Medical_Info My_Info;
                              /*"refl_" is Open-C++ specific*/
/* Server stub */
   main() {
      server_main_loop();  /* handles client requests */
   }                      /* invokes methods of My_Info */
```

Fig. 3. Structure of a server (primary or backup)

In the example (see Fig. 3), the base-level class `Medical_Info` has been associated with the meta-level class `Passive_Repl_MetaObj`. Thus, instances of the class `refl_Medical_Info` such as the object `My_Info` will have a passive replica that is managed at the meta-level by `Passive_Repl_MetaObj`. The details of the passive replication mechanism are implemented by `Passive_Repl_MetaObj` and do not appear in the source code for `Medical_Info`. The communication protocols are managed by the server stub (`server_main_loop`). In the given example, the state of a `Medical_Info` object corresponds to the `med_rec` protected reflective attribute. Open-C++ requires this state to be reflective so that it can be accessed from the meta-level in order to generate checkpoints. The `Write_Info` method which updates the object state is also declared to be reflective using a `//MOP reflect` declaration. This enables an invocation of `Write_Info` to be trapped at the meta-level in order to checkpoint the updated state of a `Medical_Info` object to

the backup server after execution of `Write_Info`. In our example we consider that the `Read_Info` method does not update the state and thus does not need to be reflective. No checkpoint is sent in this case.

Meta-level. Reflection is used to control modifications to attributes of the reflective object. As previously mentioned, the methods that modify the data attributes of the primary state are made reflective. The meta-object which controls the primary's reflective object traps all the invocations of its reflective methods; we take advantage of this ability to checkpoint the server state to its backup replica. The inter-replica protocol is handled at the meta-level and includes the following actions:

Primary actions	Backup actions
• checkpointing the server state when a client request has been processed	• receiving and storing the primary checkpoints at the base-level
• recovery on backup crash	• recovery on primary crash
• reconfiguration	• reconfiguration

The base-level is identical for both replicas, but the actions performed at the meta-level by the primary and the backup replica are different: the primary sends checkpoints to the backup after each reflective method invocation, the backup replica processes these checkpoints. The meta-level also includes mechanisms for error detection and recovery. This protocol is summarised in Fig. 4.

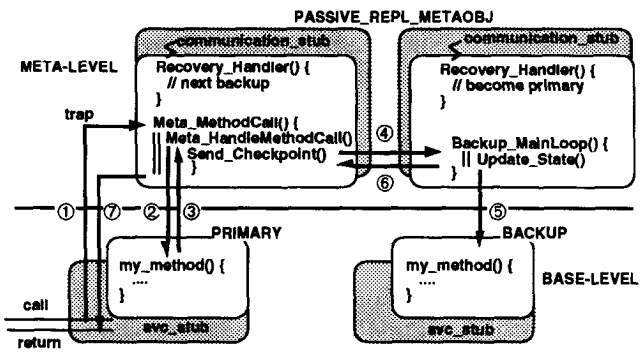

Fig. 4. Passive replication protocol

Both sides (primary and backup) presented in this figure are actually implemented by a single meta-level class as shown in Fig. 5. Every reflective method call is trapped (①) at the meta-level. Then the object method is called at the base-level from the meta-level (②). Control returns back to the meta-level (③) and the updated state of the primary replica is then sent in a checkpoint to the backup replica (④). The latter updates its base-level object state directly (⑤) and sends an acknowledgement (⑥) to the primary. The reflective method invocation completes and returns to the client (⑦).

As well as the communication stubs used at the base-level for communication between the client and the primary server, communication stubs are also used at the meta-level mainly for sending/receiving checkpoints and for detecting errors. The detection mechanism is simple but not efficient in the current implementation; this will be discussed later in Section 5. When the absence of any peer is detected, either when sending or receiving checkpoints, the `Recovery_Handler` is activated at the meta-level where a recovery procedure is performed. The `Recovery_Handler` can also be activated directly by the meta-level communication stubs. A simple periodic checking of the presence of peers can be implemented in these stubs as well as more sophisticated detection mechanisms with reduced latency which depend on the underlying communication protocols.

```
Passive_Repl_MetaObj :: Meta_MethodCall (Id my_method,...){
    /* execution of the method */
        Meta_HandleMethodCall(my_method);
    /* storage of all reflective data in a message */
        Init_Checkpoint(state);
    /* sending a checkpoint to the backup */
        if (Send_Checkpoint(backup,state) == ERROR)
            Recovery_Handler();
}
Passive_Repl_MetaObj :: Meta_StartUp()    {
/* status initialised by the meta-level class constructor */
        if (status == primary_status)   {
        /* selection and connection with a first backup */
                Replica_Select (backup);
                Replica_Connect(backup);
        }
        else   {   /* status == backup_status */
            /* waiting for server connection */
            Wait_For_Replica_Connect(primary);
            /* checkpoint receive and store loop */
                Backup_MainLoop();
        }
    /* begin execution at the base level */
}
Passive_Repl_MetaObj :: Backup_MainLoop()    {
        while (Receive_Checkpoint(primary,state) != ERROR)
            Update_State(state);
        Recovery_Handler();
}
Passive_Repl_MetaObj :: Recovery_Handler()    {
    /* primary crash: backup becomes primary */
        if (status == backup_status)    {
            status = primary_status;
        }
    /* selection and connection with a new backup */
        Replica_Select (backup);
        Replica_Connect(backup);
    /* storage of all reflective data in a message */
        Init_Checkpoint(state);
    /* send current state to the new backup */
        if (Send_Checkpoint(backup,state) == ERROR)
            Recovery_Handler();
}
```

Fig. 5. Passive_Repl_MetaObj simplified source code

For instance, when a primary crashes, the recovery procedure can be briefly described as follows: the backup leaves the main loop, its status is set to primary, a new backup replica is simply selected from a list of pre-created replicas, a connection is established with this new backup and finally the current state of the computation is sent to initialise the new backup. A connection is then established by the new primary with the client at the base level. The

recovery mechanism also involves numbering client calls, possibly including the current reply and a checkpoint number in every checkpoint, etc. Checkpoints are acknowledged in order to detect errors and failures. Not all the related details in the source code are presented here in order to keep it simple and clear. The source code for the `Passive_Repl_MetaObj` meta-level class is essentially composed of the methods mentioned in Fig. 5.

Finally, the meta-level is also responsible for reconfiguration after the failure of a peer has been detected. A new replica must be created and initialised with the current state of the computation. This is why the internal state of the reflective object must always be made accessible at the meta-level by declaring it reflective. The meta-level of an operational replica must be able to read the state, and the meta level of the new replica must be able to write this state down to the base-level. This is true for any replication protocol.

4.2 Semi-active replication

In this protocol, a client sends its requests to a leader replica which in turn forwards it to its follower replica. Both replicas process the request, but only the leader replies to the client (see Fig. 6).

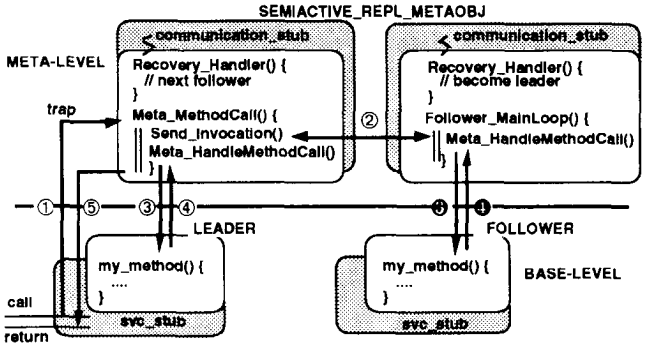

Fig. 6. Semi-active replication protocol

The simple example taken here considers deterministic behaviour of the execution only. One reason for using semi-active replication in this case instead of passive replication is determined by the size of the object state; when it is very large, passive replication would imply large overheads. Multicast protocols are not considered in this example, and therefore the request message received by the leader is forwarded to the follower at the meta-level.

The source code of the server is almost identical to the previous case (see Fig. 3), except that the base-level class is associated with a different meta-level class (`Semiactive_Repl_MetaObj` in this case). The object state in this protocol is updated by the concurrent execution of the leader and follower replica.

The implementation of `Semiactive_Repl_MetaObj` (see Fig. 6) is similar to `Passive_Repl_MetaObj`. The main difference is that a method invocation is transmitted instead of a checkpoint. A reflective method call is trapped (①) and transmitted by the leader to the follower (②) and acknowledged. Both the leader and the follower execute the method concurrently. On each side, the method at the base-level is called from the meta-level (③, ❸) and control is returned back to the meta-level when the method execution is completed (④, ❹). Finally, the initial method call returns to the client (⑤). Synchronisation between replicas could be added to this example in order to prevent the follower from getting too far behind the leader. Atomic multicast could also be used to simplify this protocol by broadcasting client requests to both replicas. This could be implemented by a meta-object on the client side; such a solution is mandatory for active replication and voting as described in the next section.

4.3 Active replication with majority voting

Several strategies can be defined for active replication. They all involve sophisticated inter-replica protocols on both the client and the server side. Our objective in this section is not to investigate these protocols, but briefly underline that they can be easily implemented using a reflective approach.

We consider here a simple example (see Fig. 7) with one client and a triplicated server: the client sends multiple requests to a group of servers and handles several reply messages (voting); all server replicas process client requests and send replies back to the client .

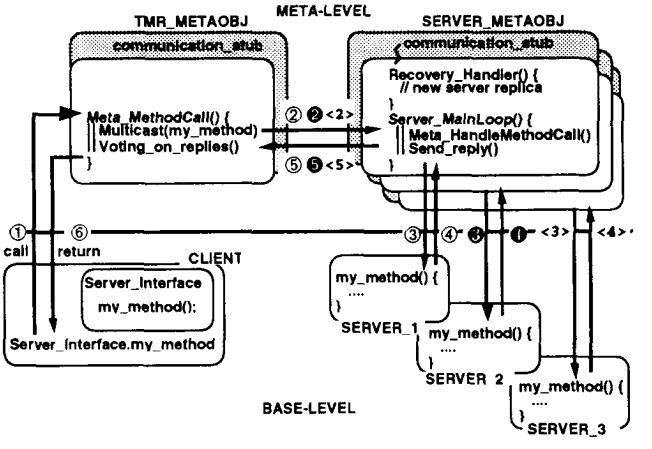

Fig. 7. Active replication with majority voting protocol

A possible implementation using meta-objects can be briefly summarised as follows. One can be defined for handling the client side, `TMR_MetaObj`, and one for the server side, `Server_MetaObj`. We consider in this section

that the client declares an object representing a remote object located in the server (just a remote object interface – `Server_Interface` in Fig. 7); the server encapsulates the remote object. The client invokes the remote object as if it was local and not replicated.

In the client, the interface object representing the remote object is associated with an instance of the meta-level class `TMR_MetaObj`; this meta-object traps method invocation on the remote object (①). The version of `Meta_MethodCall` defined by `TMR_MetaObj` is responsible for sending a request corresponding to the method invocation to each server replica (②, ❷, <2>), and then for voting on the replies (⑤, ❺, <5>) before returning the final result to the client base-level (⑥). Two comments can be made: (i) the protocol used for sending the request in this case should be an atomic multicast protocol; (ii) `Meta_MethodCall` for `TMR_MetaObj` does not call `Meta_HandleMethodCall` since the invoked methods will be executed remotely by the server replicas.

In each server replica, an instance of the meta-level class `Server_MetaObj` is bound to the "reflective object" that encapsulates its state. This meta-object is responsible for handling client remote requests issued by the `TMR_MetaObj` on the client side and for executing the corresponding methods on the server side via the `Meta_HandleMethodCall` (③-④, ❸-❹, <3>-<4>). The `Server_MetaObj` is also responsible for handling reconfiguration when one of the server replicas in the group fails. Creation of a new server replica is done by the `Server_MetaObj` of the operational replica(s): this operation updates the new replica with the current state of the server (accessible at the meta-level as shown in previous examples) and adds a new member to the group of replicas.

Just as in previous examples, the application programmer does not have to be aware of the details of how fault tolerance is implemented. Remote invocation details are also hidden.

5 Implementation issues

Most replication protocols need atomic multicast to deliver input messages to several replicas running concurrently or to send checkpoints to a set of standby replicas, for instance. This service can be implemented either at the environment level as in Isis or at the communication level as in Delta-4, the latter providing the better performance.

We discuss here the implementation of replication techniques based on the classic distinction between "application level" (user space) and "system level" (system space). The application level itself involves two programming levels, the base-level and the meta-level. The application meta-level implements the replication techniques (inter-replica protocols) based on the services provided by the system level:

- application meta-level: this level is dedicated to the implementation of the replication protocol; reconfiguration is also handled at this level;
- system level: programming meta-level protocols involves several services that should be provided by the underlying system. Failure detection, atomic multicast and group management protocols are some examples.

The frontier between these two levels may vary according to the underlying runtime system and the hardware architecture of the nodes. Micro-kernel technology provides a good basis for tuning the frontier between application level and system level for implementing meta-level functionalities.

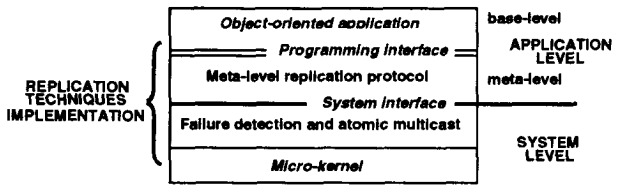

Fig. 8. Implementation layers of replication techniques

As shown in Fig. 8, this technology provides a good implementation framework for the system dependent services. This approach enables meta-programmers to define new meta-level classes for various replication protocols, according to several failure assumptions, but also with respect to various hardware architectures and node configurations. The development of meta-level classes can take advantage of inheritance as for the construction of base classes in Arjuna. Solutions proposed in Isis and Delta-4 for failure detection and atomic multicast protocols can be implemented directly on top of the micro-kernel. This approach will be experimented with in the near future using Chorus [21] and xAMp multicast protocols [20].

6 Conclusion

When the underlying system does not provide fully transparent fault tolerance mechanisms, programming fault tolerant applications is a difficult activity since functional and non-functional programming are often mixed at one level and the programmer needs to know details of the fault tolerant mechanisms that are used. The two (or more) levels of programming provided by reflective object-oriented languages enable these two rather different development activities to be done separately using the same language. This approach is obviously not restricted to fault tolerant mechanisms, but also encompasses distribution and other non-functional aspects such as security, transaction

management, configuration management, etc. As a side effect, this approach facilitates testing and debugging.

The simple examples presented in this paper have all been prototyped. Various replication techniques are now being implemented in order to obtain a library of meta-level classes for programming fault tolerant distributed applications. This activity will also involve the development of other meta-level classes for distribution and security purposes. Meanwhile, original meta-level classes will evolve according to improvements in error processing protocols, error detection mechanisms and communication protocols implemented in user space or at system level. Thanks to the reflective approach, such evolution can occur without any change in the source code of the user applications. Nevertheless, it is clear that a more sophisticated model of reflection would allow better results. For instance, a language with more reflective attributes than Open-C++ would ease the solution of several implementation problems (e.g. minimisation of the amount of information in a checkpoint) that we have encountered. It is also important to mention that the ability to bind application-level objects to meta-level objects dynamically could be used to allow the application to adapt dynamically to system evolution either with respect to the underlying operating system services or with respect to new hardware configurations and failure assumptions.

In conclusion, a reflective approach combines the advantages of the object model with the advantages of a system-based approach to fault tolerance (transparency and separation of concerns). Like a system-based approach, a reflective approach provides (i) well defined software component structuring in terms of objects and (ii) access to internal operations of the model. Fault tolerance mechanisms that were supported by the runtime layer can now be implemented as a set of meta-objects, thus making this approach more flexible.

Acknowledgements: the authors wish to thank our colleagues in the PDCS-2 project, especially Brian Randell, who participated in the elaboration of these ideas during the numerous discussions on the subject. We are very grateful to Felicita Di Giandomenico, Alexander Romanovsky and in particular David Powell for their valuable comments on previous versions of this paper.

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] Arjuna, *The Arjuna System Programmer's Guide*, Dept. of Comp. Science, Univ. of Newcastle-upon-Tyne, UK, July 1992.

[3] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella and M. Gaspari, "Metalevel Programming in CLOS", *Proc. ECOOP'89*, pp.243-56, 1989.

[4] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues and N. A. Speirs, "The Delta-4 Extra Performance Architecture (XPA)", *Proc. FTCS-20*, (Newcastle upon Tyne, UK), pp.481-8, 1990.

[5] J. F. Bartlett, "A Non-Stop (TM) Kernel", *Proc. SOSP-8*, (Pacific Grove, CA, USA), pp.22-9, 1981.

[6] K. P. Birman, "Replication and Fault-Tolerance in the ISIS System", *ACM Operating Systems Review*, 19 (5), pp.79-86, 1985.

[7] K. P. Birman and T. A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", *ACM Operating Systems Review*, 21 (5), pp.123-8, 1987.

[8] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier and J. Voiron, "Active Replication in Delta-4", *Proc. FTCS-22*, (Boston, MA, USA), pp.28-37, 1992.

[9] S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with Meta-Level Architecture", *Proc. ECOOP'93, LNCS 707*, (O. Nierstrasz, Ed.), (Kaiserslautern, Germany), pp.482-501, 1993.

[10] D. L. Detlefs, M. P. Herlihy and J. M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *Computer*, pp.57-69, December 1988.

[11] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience", *Proc. FTCS-23*, (Toulouse, France), pp.2-9, 1993.

[12] *The Isis Distributed Toolkit — User Reference Manual*, Isis Distributed Systems, Inc., 1992.

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

[14] P. Maes, "Concepts and Experiments in Computational Reflection", *Proc. OOPSLA'87*, pp.147-55, 1987.

[15] S. Matsuoka, T. Watanabe and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", *Proc. ECOOP'91*, pp.213-50, Springer-Verlag, 1991.

[16] A. Paepcke, "PCLOS: Stress Testing CLOS", *Proc. OOPSLA'90, ACM SIGPLAN Notices*, pp.194-211, 1990.

[17] D. Powell (Ed.), *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Research Reports ESPRIT, 484p., Springer-Verlag, Berlin, Germany, 1991.

[18] D. Powell, "Failure Mode Assumptions and Assumption Coverage", *Proc. FTCS-22* (Boston, MA, USA), pp.386-95, 1992.

[19] D. Powell, "Distributed Fault-Tolerance — Lessons Learnt from Delta-4", in *Hardware and Software Architecture for Fault Tolerance: Experiences and Perspectives* (M. Banâtre and P. A. Lee, Eds.), *LNCS 774*, pp.199-217, New York: Springer Verlag, 1994.

[20] L. Rodrigues and P. Veríssimo, "xAMp: A Protocol Suite for Group Communication", *Proc. SRDS-11*, pp.112-21, 1992.

[21] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser, *Overview of the CHORUS® Distributed Operating Systems*, Chorus Systèmes, Report, N°CS/TR-90-25, April 1990.

[22] S. K. Shrivastava, G. N. Dixon and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, 8 (1), pp.66-73, January 1991.

[23] B. C. Smith, "Reflection and Semantics in Lisp", in *11th Annual ACM Symposium on Principles of Programming Languages*, pp.23-35, 1984.

[24] N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to provide Dependable Distributed Computing", *Proc. FTCS-19*, (Chicago, IL, USA), pp.184-90, 1989.

[25] R. J. Stroud, "Transparency and Reflection in Distributed Systems", *ACM Operating Systems Review*, 22 (2), pp.99-103, April 1993.