Journal of the Brazilian Computer Society

versão impressa ISSN 0104-6500

J. Braz. Comp. Soc. v. 4 n. 2 Campinas Nov. 1997

http://dx.doi.org/10.1590/S0104-65001997000300003

A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software

Luiz E. Buzato
Institute of Computing Science
University of Campinas, Brazil
Caixa Postal 6176 - 13083-970
Campinas, SP
buzato@dcc.unicamp.br

Cecília M. F. Rubira
Institute of Computing Science
University of Campinas, Brazil
Caixa Postal 6176 - 13083-970
Campinas, SP
cmrubira@dcc.unicamp.br

Maria Lúcia B. Lisboa
Institute of Informatics
Federal University of Rio Grande do Sul, Brazil
Caixa postal: 15.064 - 91501-570
Porto Alegre, RS
llisboa@inf.ufrgs.br

Abstract *This paper proposes a reflective object-oriented architecture for developing fault-tolerant software. Reflective object-oriented programming promotes a modular structuring of systems by means of a new dimension of modularization—the separation between base-level objects and meta-level objects. This property allows the creation of metaobjects responsible for managing tasks of application objects located at the base level. In the context of this work, computational reflection is applied to implement various strategies of fault tolerance at the meta-level in a transparent manner for the application programmer, that is, without interfering with the original structure of application objects that require fault tolerance facilities. The use of the proposed architecture has the following advantages: (i) separation of concerns, that is, separate the concerns related to the application domain from those related to the implementation of fault-tolerant mechanisms; (ii) it promotes code reuse of fault-tolerance mechanisms; (iii) it allows application programmers to use the most adequate fault-tolerance strategy for his implementation, and (iv) it provides a design that is more adaptable, flexible and easier to extend than traditional designs for developing fault-tolerant software. Our reflective architecture is composed of three levels, and is based on the abstraction of object groups.*

Keywords: *software fault tolerance, hardware fault tolerance, object-oriented programming, computational reflection, meta-level architecture.*

1 Introduction

Fault tolerance represents a major challenge to designers of modern computing systems—in particular, in the development of critical applications, such as stock market, air traffic control, and the world wide web. The construction of fault-tolerant systems is not a simple task [8, 14]; it requires the use of appropriate techniques during the whole software development cycle. In general, these techniques are based on the provision of redundancy, both for error detection and error recovery. However, the provision of software redundancy implies (i) a cost increase of the software development, and (ii) a complexity increase of the system, caused by the addition of redundant components. Ideally, the additional software redundancy should be incorporated to the original system in a structured and non-intrusive manner in order to facilitate the task of constructing fault-tolerant systems by application programmers.

The object model has been considered a promising approach for the development of fault-tolerant software [23], because it integrates well-established software engineering principles like data abstraction, encapsulation, modularity, hierarchy, and strong typing. The adoption of object-oriented techniques for building fault-tolerant software allows easier control of software complexity since the object paradigm promotes a better system structuring than the traditional functional decomposition approach. Moreover, it encourages reuse of fault-tolerance mechanisms, contributing to reduce software development costs.

The incorporation of the computational reflection technique into the object model in the form of metaobject protocols can also facilitate the construction of fault-tolerant systems. Reflective fault-tolerant programming promotes system modularity by means of a clear separation between application objects (base level) and management objects (meta-level). That is, the reflection mechanism allows the creation of metaobjects responsible for the management of application objects at the base level. In the scope of this work, computational reflection is applied to implement various fault-tolerance strategies at the meta-level, in a transparent and non-intrusive way, that is, without interfering with the original structure of application objects that require fault-tolerance facilities. The adoption of a reflective architecture provides the following advantages for application programmers: (i) separation of concerns, that is, separate the concerns related to the application domain from those related to the implementation of fault-tolerant mechanisms [12, 18]; (ii) it promotes code reuse of fault-tolerance mechanisms, (iii) it allows application programmers to use the most adequate fault-tolerance strategy for his implementation, and (iv) it provides a design that is more adaptable, flexible and easier to extend than traditional designs for developing fault-tolerant software.

This paper proposes a new object-oriented reflective software architecture for developing fault-tolerant applications using meta-level programming. Our reflective architecture is composed of three levels, and is based on the abstraction of object groups. In the literature, various practical experiments show the feasibility of using meta-level programming for obtaining transparency of distribution [25], replication [10], persistence [3, 12], object location control [18], etc. Our architecture uses reflection to obtain transparency of hardware and software fault tolerance. To the best of our knowledge, very few concrete experiments have been reported on the use of meta-level programming for the provision of software fault tolerance.

The remainder of this paper is organised as follows. Section 2 discusses fault-tolerance strategies for both hardware and software design faults. Section 3 reviews the basic ideas related to computational reflection and meta-level software architectures. Section 4 describes our object-oriented reflective software architecture. Section 5 analyses examples of the use of the reflective architecture proposed. Section 6 discusses related work. Finally, we make some concluding remarks and suggest directions for future research.

2 Fault Tolerance Concepts

In the design of fault-tolerant systems, a major point of discussion and contention is how much fault-tolerance is needed and what kinds of faults a system should tolerate. That is, whether the system tolerates only hardware faults, such as network and computer failures, and/or software design faults related to residual "bugs" in the software. Following the terminology defined by Lee and Anderson [15], hardware fault tolerance is obtained either through simple replication of hardware/software components or through techniques of saving and recovering computation states. In the first case, one can allocate replicated objects to different parts of the system in order to provide availability in the presence of partial hardware faults. In the second case, critical services can be executed as an atomic action. In the literature, various distributed object-oriented systems adopt atomic actions for the provision of hardware fault tolerance [5, 24].

The provision of software fault tolerance, on the other hand, cannot be achieved by simple replication of software components or use of atomic actions since a design fault is a logical error in the state of the design. In what follows, we assume that a complex software system will inevitably contain residual faults despite all of the fault prevention techniques which may have been used. So design fault tolerance is required if it is necessary to tolerate such faults, and its provision can only be achieved if design diversity has been anticipatedly built into the system. There is a number of methods that have been proposed for tolerating faults in non object-oriented software based on design diversity. Classical examples of such methods are: the recovery block [21] and N-version programming [4]. A simple generalization of these two schemes is the following [15]:

(i) several variants are implemented for each software component,
(ii) each variant is designed independently, but should conform to the same specification,
(iii) a control framework (recovery block or N-version) executes the variants, and
(iv) the results are evaluated by an adjudicator.

This separation of variants, adjudicator and control frameworks support a well-structured and flexible approach to the construction of fault-tolerant software. From a designer's point of view, the production of fault-tolerant software implies using some amount of code diversity as we mentioned earlier.

2.1 Object-Oriented Fault Tolerance

For several reasons, the object model offers good support for the development of fault-tolerant applications:

- one can easily replicate objects;
- object diversity can be achieved by changing the implementation of objects without changing the object's interface or behaviour;
- objects can be considered the smallest grain of failure and
- distribution allowing the deployment of fault tolerance to only critical objects, and
- the message passing mechanism facilitates object monitoring. Object instrumentation using sensors and actuators becomes easier because messages can be intercepted.

2.2 Redundancy Granularity in the Object Model

Considering an object-oriented approach for implementing components with diverse design, as discussed above, one could choose different granularity for including software redundancy within the object model. We identify at least four categories according to the place where redundancy may be included, which we

will refer to as *operation or method diversity, object or data diversity, class diversity and meta-level diversity* [9, 23, 26]. We discuss each of these categories below.

Operation Diversity: In this case, variants are methods declared in the class, which are independently developed from the same functional specification.

Object Diversity: Redundancy is achieved by providing diversity in the data spaces of the classes and/or method implementation. Fault tolerance would be achieved by different (concrete) implementations of the same parent (abstract) class.

Class Diversity: Variants are objects and, therefore, both the state representation and the set of operations can be independently designed from the same specification of a type.

Meta-level Diversity: Redundancy could also be included at the meta-level depending on the application requirements. At the meta-level, it would be also possible to describe an implementation of redundant classes that could be used to build fault-tolerant objects at the application level.

In distributed systems, administrative requirements, such as recoverability and persistence, can be implemented by replication of components. In this case, the object is the smallest grain of distribution and fault tolerance.

Object Replicas: Redundancy is implemented by multiple instantiations of the same class and a replication technique shall be applied to guarantee the same status in all replicas during the system lifetime.

In our reflective software architecture it is possible to combine object replication with several fault tolerance strategies using design diversity.

3 Computational Reflection

Reflection is defined as being the ability of observing and manipulating the computacional behaviour of a system through a process called reification (or materialization). This idea has been originated in the field of logic, and more recently it has been recognised as an important notion in the construction of object-oriented systems. Computational reflection is based on the notion of defining a programming language interpreter in terms of the language itself. In the object paradigm, this means to represent all abstractions of the object model in terms of the object model itself. As a consequence, the representation of classes, methods, attributes and objects is reified by means of metaclasses and metaobjects. Thus, the object model semantics is represented by metaobjects and meta-level programming can be used to modify it. Computacional reflection defines an architecture composed of meta-levels. Actions to be realised upon the base-level objects are implemented in the meta-level. Reflection can be used to intercept and modify the effect of various basic operations of the object model, such as "to create an object", "to invoke a method", etc. This notion is a convenient and interesting manner of implementing transparently administrative tasks of systems, such as the provision of fault tolerance. For the purpose of illustration, suppose that for each base-level object **O** exists a corresponding metaobject **MO** that represents the behavioural and structural aspects of **O**. In a reflective system, if object **P** sends a message m to object **O**, the meta-object **MO** intercepts the message m, reifies the base level computation and takes over execution; later **MO** returns the thread of control to **O**. From the point of view of object **P**, computational reflection is transparent: **P** sends a message requesting a service to **O**, and receives the results with no knowledge that the message was intercepted and redirected to the meta-level.

In the construction of object-oriented systems, usually only a subset of all objects implement critical services. As a consequence, these objects should have a fault-tolerant behavior. Our idea is to add this

fault-tolerant behavior through metaobjects. The adoption of a reflective architecture addresses many of the requirements mentioned earlier: separation of functional activities related to the application domain from the administrative activities related to the provision of fault tolerance, transparency of fault-tolerance control mechanisms and selective provision of fault tolerance to critical objects.

Apertos [29] is an example of a reflective operating system developed by Sony in which reflection was apllied with success. The combination of reflection and object-oriented programming in the form of a metaobject protocol is provided by object-oriented languages such as CLOS (Common Lisp Object System) [3, 19] and ABCL/R. Metaobjects protocols (MOPs) are interfaces to the language of a system that give users the ability to modify the language's behaviour [12].

4 An Object-Oriented Reflective Software Architecture

The computational model chosen for the definition of the software architecture is based on object groups. A *group of objects* is a set of objects that implement the same specification. Members of a group may have different implementations for the same object interface, i.e. code diversity, when their implementation is identical they are called *replicas*. In our model, groups are the basic building-blocks of fault-tolerant software, their internal structures transparent to their clients. Group transparency means that a client only works with the abstraction of groups. Internally, a group coordinator manages the communication among individual group members. Thus, the notion of object groups is ideal for the implementation of fault-tolerance mechanisms. It is possible to build ideal components out of groups with the following advantage: *failure masking*, when a member of the group suffers a failure, its peers can take on the function of the faulty object and mask the failure so that clients do not become aware of the partial failure.

In an object-oriented application, fault-tolerant objects are those that have mechanisms that guarantee reliability and availability of their services. In our software architecture, group of objects with code diversity are used to implement reusable and transparent software fault-tolerance mechanisms. When an object group is formed by replicas then only hardware fault tolerance can be implemented. Finally, a group formed by one object does not offer any type of fault-tolerance. These three types of object groups can be combined to build applications with tailor-made fault tolerance.

Group transpareny is obtained through the used of computational reflection and object-orientation. Group management is implemented at metalevel. Thus, in the proposed architecture, the software fault-tolerant behaviour of an application object is determined by the behaviour of the metaobject (group) associated to it. Metaobjects implement recovery blocks [22], N-versions [4], and object groups.

The proposed architecture organizes objects and metaobjects in the following way: a metaobject is associated to several objects that are instances of the same class. This arrangement has been adopted because it maps easier to the constructs for computation reflection implemented by OpenC++ version 1.0. In order to allow computational reflection to take place, OpenC++ [11, 16, 17] pre-processes the source code of the reflective application and statically associates metaobjects to objects. At application level, programmers can choose what classes and attributes are to be controlled by its associated metaclass. At metalevel, programmers specify the methods of the metaobject that are to be invocated when a message sent to an object is intercepted. In the near future, we plan to experiment with alternative meta-level architectures [11, 16, 17].

4.1 Levels of the Reflective Software Architecture

Our reflective software architecture has three levels, described below:

- D_0: base or application level. Contain objects that use the mechanisms for software fault-tolerance implemented at metalevel and objects that do not use the metalevel. In our architecture, fault tolerance mechanisms can be applied selectively.
- D_1: management or meta-level. This level is composed by the objects that implement mechanisms for software fault tolerance, including N-version programming and recovery blocks (Section 2).
- D_2: the implementation of level D_1 requires the support of basic mechanisms for concurrency control, distribution and state recovery.

Figure 1 shows a diagram of the proposed software-architecture Suppose that a programmer wants to make objects of class X tolerant to software faults; N-version programming can be used to obtain such fault tolerance [4]. This technique consists of creating n versions of a program, denominated *variants*. For each input, all variants are executed and have their results compared by an adjudicator. Variants are developed by different teams, based on the same software specification. The voting mechanism implemented by the adjudicator evaluates the results of a computation and selects those upon which the majority of voters (group members) agree as the final computation result.

Initially, a programmer must provide the variants for class X, say classes VariantX1 and VariantX2 (Figure 1). Next, the programmer should define the metaclass that activates the N-version and group management mechanisms. Classes X and MetaX have an instantiation relationship that allows X to have an indirect access to the protocols implemented at management level. The class MetaX is a subclass of N-Version, that is associated to classes Adjudicator and Group. The class Adjudicator implements the voting mechanisms necessary to the implementation of N-versions. Different voting strategies can be implemented through the specialization of class Adjudicator. For example, subclass A1 can implement voting by simple majority and subclass A2 can implement it by unanimity of results. The class Group implements the mechanisms of group management, that is, creation, destruction, and group membership changes. Similarly, the class RecoveryBlock is associated to the classes Adjudicator and Group. Changes to classes of the management level are transparent to the application level. Support functions are implemented at level D_2, i.e., the operating system. For example, the class MetaGroup is responsible for the implementation of the replication protocol. MetaGroup has two subclasses, Active Replicator and PassiveReplicator (Figure 1). The instantiation relationship between Group and Passive Replicator implies that an application object is materia-lized at meta-level using passive replication techniques.

5 Use of the Reflective Architecture

5.1 Example 1: Coordinated Checkpointing

Consider the problem of obtaining and recording in stable memory, the global state of objects belonging to a distributed application. Consistent global states are necessary to implement backward error recovery techniques for hardware fault-tolerant systems. Each object of the distributed application should be capable of checkpoint its state to stable memory. Additionally, each checkpoint should be taken in a consistent manner to guarantee the consistency of the global state and avoid the domino effect in the case of failure. Several algorithms have been proposed to solve the problem of enforcing coordinated state recovery in distributed applications. One of the well-known algorithms was proposed by Koo and Toueg [13]. This algorithm can be implemented by metaobjects of the D_1 level of our reflective architecture, making global checkpointing of states transparent to applications.

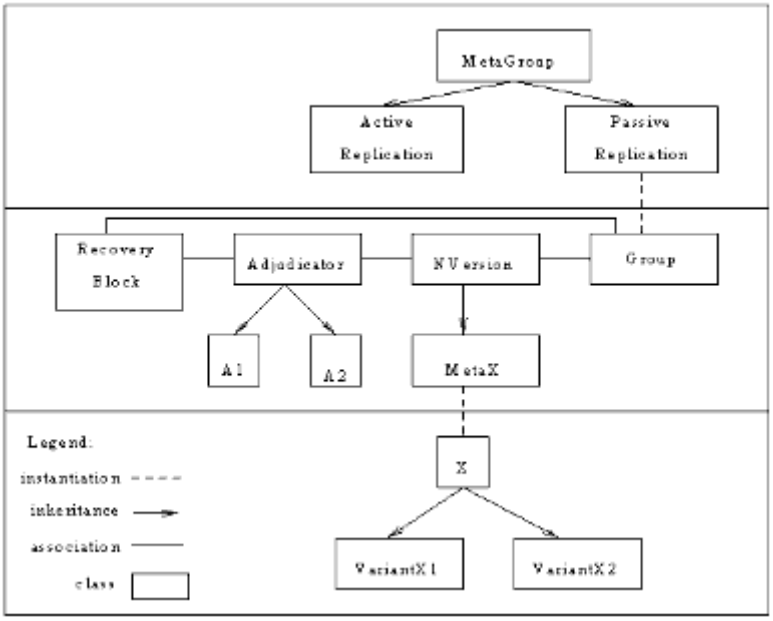

Figure 1: Reflective Software Architecture

5.2 Example 2: N-version Programming

Consider the implementation of a dependable stack. Two object-oriented solutions are discussed. The first one does not make use of our reflective object-oriented software archictecture. In contrast, the second one benefits from the metaobject protocols implemented in our architecture.

5.2.1 Object-oriented Solution

The abstract class Stack (Figure 2) has two pure virtual methods push and pop (Figure 2).

```
class Stack{

public:

virtual int pop( ) = 0; // pure virtual

virtual void push(int item) = 0; // pure virtual

);
```

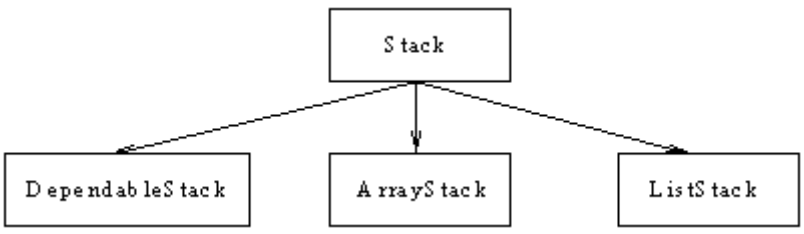

Figure 2: Stack Class Hierarchy

ArrayStack and ListStack are concrete classes derived from the abstract class Stack. They implement variants of the service specified by Stack. Pure virtual methods of abstract classes do not have implementation, they should be overloaded by derived classes. Thus, a stack can be instantiated as an array-based stack (ArrayStack variant) or as a list-based stack (ListStack variant).

DependableStack is also derived from Stack, and defines the public interface used by clients of Stack. An object of the type DependableStack forwards all messages it receives to the variant objects. DependableStack is equivalent to a group with two members, an instance of ArrayStack and an instance of ListStack. In this case, the instantiation of a DependableStack implies the instantiation of the two variants. A possible implementation of DependableStack is described below.

```
class DependableStack: public Stack {
private:

Stack *anArrayStack;
Stack *aListStack;

public:

DependableStack( )
{anArrayStack = new ArrayStack;

aListStack = new ListStack}

~DependableStack( ){ delete anArrayStack; delete aListStack;}

int pop( );

void push(int item); }
```

A partial implementation of the method pop is shown below:

```
int DependableStack::pop( )

{ ...

NVersion({anAraryStack, aListStack}, Stack::pop, adjudicator);

return item; }
```

The main disadvantage of this solution is that the fault-tolerance mechanism is not transparent to application programmers, that is, in order to use it they have to define the class DependableStack, program the instantiation of the variants (group members), and then send a message to the instance of class NVersion. Additionally, programs that used the original Stack class will have to be changed to use the newly defined class DependableStack. Next, we discuss an alternative solution to the creation of dependable stacks, based on our reflective framework.

5.2.2 Reflective Object-Oriented Solution

Using computational reflection, it is possible to obtain a transparent and easy to use dependable stack. First, a metaclass is defined to extend the functionality of the original class Stack. The variants are defined as in the previous Section. In OpenC++ version 2.0 [7] the definition of a dependable stack can be written as:

```
metaclass Stack : public NVersion;

class Stack{

public:

virtual int pop( );

virtual void push(int item); };
```

A client of Stack, located at level D_0 of the reflective architecture can use the newly defined dependable stack simply by instantiating a stack and sending messages to it.

```
Stack *p = new Stack;

integer i = p->pop( );
```

In the example above, the construction of p is intercepted by the metaobject protocol implemented at level D_1. After interception, the group of variants is created. The group-based implementation of Stack takes over and guarantees the activation of the N-version protocol for each method invocation of Stack. When the message pop( ) arrives, the following actions occur: (i) pop( ) is sent to each of the variants encapsulated by the dependable stack, (ii) the results of the execution of each pop( ) are collected and sent to the adjudicator, and (iii) the final result is passed to the application object (client). According to our reflective software architecture, the N-version and group management mechanisms are implemented at level D_1. Note that the programmer of a dependable application does not have to explicitly send a message to NVersion as in the previous solution (Section 5.2.1.).

6 Related Work

Computational reflection has been used by Fabre et al [10] to implement monitoring of replicated objects. Fabre et al. suggest an architecture for distributed applications where metaobjects implement different monitoring and replication strategies. In Fabre's model, based on a client-server architecture, each client and server has a metaobject associated to it. When a message from a client reaches a server, it is intercepted and reified by the associated metaobject. Their work allow the transparent implementation of multiple replication protocols. Their solution, however, is not based on object groups. Our definition of object groups is diverse from the definition adopted by the reflective system ABCL/1 [28] that allows definition of groups as a hierarchical group of objects whose funcionality is combined to execute protocols. The Object Communities framework [6] uses a metaobject protocol to manage distribution. In this framework, a centralized server has the role of collecting and distributing messages coming from and going to all members of the object community. Similarly to our framework, Object Communities was implemented in OpenC++, but addresses only distribution of objects. Ancona et al. [1] propose a software architecture similar to ours, but their focus is on code reuse not on fault tolerance. In contrast, our work extends the use of computational reflection, including software and hardware fault tolerance mechanisms, group management, and distribution. Most programming environments built to date for developing fault-tolerant software [2, 20] provide programmers with explicit fault-tolerance mechanisms. As a result, both application and fault tolerance code are braided together in a monolithic software component, making it very difficult to be maintained and reused. Finally, experimental and analytic results [27, 30] show that metaobject protocols do not represent a significant overhead to the performance of fault-tolerant applications.

7 Conclusions and future work

This work has presented an object-oriented reflective software architecture that can serve as a framework for the developmet of fault-tolerant applications. The architecture separates objects of the fault-tolerant application into well-defined levels. At the *base level* are the objects of the application, as his developer has implemented them. At the metalevel are the objects that implement fault tolerance mechanisms and group management. Finally, at the meta-metalevel are the very basic object management mechanisms, usually implemented by the operating system. This software architecture allows transparent reuse of application and metalevel objects. Experimental results show that our approach is particularly promising for the development of highly depandable systems.

The prototype of the architecture was implemented using OpenC++ version 1.0 that has some limitations, it allows reflection to take place only at method level. We are currently re-implementing our software architecture using two distinct environments: OpenC++ version 2.0 and Java.

Acknowledgements

Cecília and Luiz would like to thank his colleagues B. Randell, J. Xu, R. Stroud, A. Romanovsky, and Z. Wu, from the University of Newcastle upon Tyne, for interesting discussions on fault-tolerance and computational reflection. Maria Lúcia had the opportunity to discuss some of the ideas presented with Massimo Ancona and Gabriella Dodero from University of Genova, Italy. Cecília's work is partially funded by CNPq-Brazil under grant no. 301360/94-4. This work has also been funded by FAPESP grant 96/1532-9.

Source Code Fragments for DependableStack

This appendix brings fragments of the source code of DependableStack implemented using our reflective architecture; the application was programmed using OpenC++ version 1.0. The software fault-tolerance mechanis used was N-version.

```
class Stack : public Fault {

int item;

char op;

public:

Stack( ) {};

void init(int i, char a_op);

//MOP reflect:

virtual void push(Fault*);

virtual int pop(Fault*);

};
```

```
// code of init, push and pop has been sup//pressed

//MOP reflect class Stack : MetaFault;

main( )

{

int port[MAXVER]; char* machine[MAXVER];

for(int i=0; i<10; i++) machine[i]= new char[100];

printf("ArrayStack host port number\n"); scanf("%d",&port[0]);

printf("ListStack host port number\n"); scanf("%d",&port[1]);

strcpy(machine[0],"marumbi"); strcpy(machine[1],"atibaia");

refl_Stack stack(machine, port);

for(int i=0; i<2; i++)

{

stack.init(i,'1');

stack.push(&pilha);

}

}

class Fault {

public:

Fault( ) { };

virtual bool Adjudicator(Fault*
resp[MAXVER]) { };

virtual char* marshalling( ){ };

virtual Fault* unmarshalling(char* buff){ };

virtual void print( ){ };

};

class Socket {

int sock;

char buff_read[MAXBUF];
```

```
char buff_write[MAXBUF];
struct sockaddr_in server;
struct hostent *hp;
public:
Socket( );
void S_write(int port,char* machine,Fault* Obj);
Fault* S_read(Fault* Obj);
void S_close( ){ close(sock);};
};
class MetaFault : public MetaObj {
int count;
char* machine[MAXVER];
int port[MAXVER];
Fault* obj;
int n_version( ){return count;};
public:
MetaFault(char* mach[MAXVER], int port1[MAXVER]);
void Meta_MethodCall(Id mid,Id cat,ArgPac& args,ArgPac& reply);
void NVersion(Fault* obj);
};
void MetaFault::Meta_MethodCall(Id mid,Id cat,ArgPac& args,ArgPac& reply)
{
printf("interceptei a mensagem \n");
Fault* obj;
obj= (Fault*) args.PopPtr( );
this->NVersion(obj);
args.PushPtr(obj) ;
```

```
Meta_HandleMethodCall(mid, args, reply);

}

void MetaFault::NVersion(Fault* obj)

{

Socket s[MAXVER]; Fault* resp[MAXVER];

int n=n_version( );

for (int i=0;i<2;i++) s[i].S_write(port[i],machine[i],obj);

for (int i=0;i<2;i++) resp[i]=s[i].S_read(obj);

if(!obj->Adjudicator(resp)) return(FALSE);

return (TRUE);

}

MetaFault::MetaFault(char* mach[MAXVER],int port1[MAXVER])

{

for (int i=0;i<2;i++){

machine[i]=new char[100];

port[i]=port1[i];

strcpy(machine[i], mach[i]);

}

}
```

References

[1] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis and M. L. Lisboa. Reflective Architectures for Reusable Fault-Tolerant Software. *XXI Latin American Conference on Informatics*, Canela, RS, Brazil, 1995.

[2] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis and E. B. Fernandez. *A System Architecture for Fault Tolerance in Concurrent Software*, IEEE Computer, (23)10:23-32, , 1990.

[3] G. Attardi et al.. Metalevel Programming in CLOS, *Proceedings of ECOOP'89*, 1989, pp. 243-256.

[4] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12): 1491-1501, 1985.

[5] L.E. Buzato & A. Calsavara. *Stabilis: A Case Study in Writing Fault-Tolerant Distributed*

Applications using Persistent Objects, Proceedings of Fifth International Workshop on Persistent Object Systems, pp. 354-375, A. Albano & R. Morrison (Eds), Italy, 1992.

[6] S. Chiba, & T. Masuda. Designing an Extensible Distributed Language with Meta-Level Architecture. *Proc. ECOOP'93*, Lecture Notes in Computer Science, 707:~482-501, 1993.

[7] S. Chiba. A Metaobject Protocol for C++. *Proc. OOPSLA'95*, ACM Sigplan Notices, 30(10):~482-501, 1995.

[8] Chillarege. *Challenges Facing Software Fault-Tolerance*, IBM Research Report RC20281 (89648), 1995.

[9] A. Clematis, M. Ancona, G. Dodero, V. Gianuzzi and M. L. Lisbôa. An Object-oriented Approach To Fault-Tolerant Software. *3rd Euromicro Workshop on Parallel and Distributed Processing*, IEEE Computer Society, Sanremo, 1995.

[10] J-C. Fabre, V. Nicomette, T. Pérennou and Z. Wu. Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming. In Proceedings of *the 25th IEEE International Symposium on Fault-Tolerant Computing* (FTCS-25), Pasadena, CA, 1995.

[11] J. Ferber. Computational Reflection in Class-Based Object-Oriented Languages. *Proc. OOPSLA'89*, ACM Sigplan Notices, 24(10): 317-327, 1989.

[12] G. Kiczales et al. *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[13] R. Koo & S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1): 23-31, 1987.

[14] P. Lee. *Software-Faults: The Remaining Problem in Fault Tolerant Systems?*, Lecture Notes on Computer Science 774, M. Banâtre & P. Lee (Eds), 1995.

[15] Lee, P. & T. Anderson. *Fault Tolerance: Principles and Practice.* Second Revised Edition, Springer-Verlag, 1990. [Links]

[16] P. Maes. Concepts and Experiments in Computational Reflection. *Proc. OOPSLA'87*, ACM SIGPLAN Notices, 22(12): 147-155, N. Meyrowitz (Ed.), 1987.

[17] Matsuoka, T. Watanabe & A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. *Proc. ECOOP'91*, Lecture Notes in Computer Science, 512, 1991.

[18] H. Okamura & Y. Ishikawa. Object Location Control Using Meta-level Programming. *Proc. ECOOP'94*, Lecture Notes in Computer Science, 821: 299-319, 1994.

[19] A. Paepcke. PCLOS: Stress testing CLOS, *Proc. OOPSLA'90*, ACM Sigplan Notices, 25(10): 194-211, 1990.

[20] Purtilo & P. Jalote. *An Environment for Developing Fault-Tolerant Software*, IEEE Transactions on Software Engineering, (17)2:153-159, 1991.

[21] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2): 220-232, 1975.

[22] B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, J. Xu. From Recovery Blocks to Coordinated Atomic Actions. In: *Predictably Dependable Computer Systems*. Eds. Randell, Laprie, Kopetz, Littlewood, Springer-Verlag, pp. 87-101, 1995.

[23] C.M.F. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation*. Ph.D. Thesis, Department of Computing Science, University of Newcastle upon Tyne, October 1994.

[24] Shrivastava, G.N. Dixon & G. D. Parrington. *An Overview of the Arjuna Distributed Programming System*. IEEE Software, 8(1):66-73, 1991.

[25] R.J. Stroud. Transparency and Reflection in Distributed Systems. *Proc. of the 5th ACM SIGOPS European Workshop on Distributed Systems*, 1992.

[26] Xu, B. Randell, C.M.F. Rubira & R.J. Stroud. Towards an Object-Oriented Approach to Software Fault Tolerance. In: *Fault-Tolerant Parallel and Distributed Systems*, D.R. Avresky (Ed.), IEEE Computer Society Press 1994.

[27] Xu, B. Randell and A. Zorzo. Implementing Software Fault Tolerance in C++ and OpenC++: an object-oriented and reflective approach. Proceedings of the International Workshop on

Computer-Aided Design, Test and Evalution for Dependability, Beijing, China, pp. 224-229, 1996.

[28] A. Yonezawa et. al.. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[29] Y. Yokote. The Apertos Reflective Operating System: the Concept and its Implementation. *Proc. OOPSLA'92*, ACM Sigplan Notices, pp. 414-434, 1992.

[30] A. Zorzo, J. Xu and B. Randell. *Experimental Evaluation of Fault-Tolerant Mechanisms for Object-Oriented Software*, Proceedings of XXIII Integrated Seminars on Software and Hardware (SEMISH'96), pp. 457-468.

Sociedade Brasileira de Computação

Sociedade Brasileira de Computação - UFRGS
Av. Bento Gonçalves 9500, B. Agronomia
Caixa Postal 15064
91501-970 Porto Alegre, RS - Brazil
Tel. / Fax: (55 51) 316.6835

jbcs@icmc.sc.usp.br