Development of a Metaobject Protocol for Fault-Tolerance using Compile-Time Reflection

Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-Garcia

LAAS-CNRS
7 Avenue du Colonel Roche
31077 Toulouse Cedex France
(+33) 561 336 243
{killijian,fabre,ruiz}@laas.fr

Shigeru Chiba

Institute of Information Science and Electronics,
University of Tsukuba
Tennodai, Tsukuba, Ibaraki 305-8573 Japan
chiba@is.tsukuba.ac.jp

ABSTRACT

The use of metalevel architectures for the implementation of fault-tolerant systems is today very appealing. Nevertheless, all such fault-tolerant systems have used a general-purpose metaobject protocol (MOP) or are based on restricted reflective features of some object-oriented language. According to our past experience, we define in this paper a suitable metaobject protocol, called FT-MOP for building fault-tolerant systems. We briefly explain how to realize a specialized runtime MOP using compile-time reflection. This MOP is CORBA compliant: it enables the execution and the state evolution of CORBA objects to be controlled and enables the fault tolerance metalevel to be developed as CORBA software.

Keywords

Reflection, compile-time and runtime metaobject protocol, fault-tolerance, replication, object state, checkpointing.

INTRODUCTION

The implementation of fault tolerance in object-oriented systems has been investigated using various approaches, either based on the use of inheritance [1,2] and reflection [3-8] or investigating their implementation on/within CORBA runtime supports [9-11]. Reflective architectures have been developed in various application fields and research areas, in particular regarding dependability issues. The use of reflective ideas and metaobject protocols provides many interesting properties, such as transparency and separation of concerns. Reflection enables an object-oriented application to be controlled at a higher level of abstraction, the metalevel. Mechanisms previously developed as standard and object-oriented libraries or else integrated into an operating system and into an ORB are developed at the user level as metalevel software with this approach. Beyond transparency and separation of concerns, this approach also provides visibility of the mechanisms that can be easily customized for a given application or system configuration using object-oriented techniques. Nevertheless, to our knowledge, most of the examples are based on existing general-purpose metaobject protocols or restricted reflective features of some object-oriented languages. They have thus to cope with their limits.

The *corner stone* of a fault-tolerant reflective architecture is the MOP. We thus propose a special purpose MOP to address the problems of general-purpose ones. The definition and the implementation of an appropriate runtime metaobject protocol for implementing fault tolerance into CORBA applications is the main contribution of the work reported in this paper.

This work shows that compile-time reflection is a good approach for the development of a specialized runtime MOP[1]. This MOP, called FT-MOP (Fault Tolerance MetaObject Protocol), is sufficiently general to be used for other aims (mobility, adaptability, migration, security). FT-MOP provides a mean to attach dynamically fault tolerance strategies to CORBA objects as CORBA metaobjects, enabling thus these strategies to be implemented as CORBA software. It also solves the problem of user-defined inherited functions (e.g. save_state and restore_state) for handling the state of application objects; it is worth noting that a wrong definition of these functions (state information missing) prevents the fault-tolerant software to perform its recovery actions.

METAINFORMATION NEEDED FOR FAULT-TOLERANCE

Implementing fault tolerance using replication strategies in a distributed system imposes being able to control the following aspects of the runtime objects:

- *object creation/deletion*: depending on the fault tolerance strategy, the object has to be created into multiple copies (the replicas) and all copies have to

[1] An extended description of the work presented in this paper can be found in [12].

register in a communication group. Deleting an object implies removing all copies from this group.

- *object invocation*: any invocation to replicated objects involves broadcasting messages through a group communication system which ensures, at least, that all correct replicas will receive all messages in the same order (this is required for replica consistency). In case of replicated calling objects, all replicas must be able to control multiple copies of request messages (filtering). After execution of the method, state information transfer (checkpoints) or synchronization among the replicas must be performed.
- *object state evolution*: whatever the strategy is, the object state must be maintained consistent among all replicas. For active replication, the basic assumption required is determinism of objects. In this case, the same inputs lead to the same outputs and the same transformation of the internal state. For strategies either based on stable storage or passive replication, then the object state has to be sent periodically to the stable storage service or to the backup replica, respectively. Being able to handle the object state (variables and internal objects) is always required for cloning a new object replica during the reconfiguration process after a failure. This involves being able to handle the state of an object-oriented program.

Handling the object state requires fine grain information of the object program since it needs structuring information which are language dependent: identification of attributes and methods, type information for attributes and method parameters, internal classes used. The reified information needed here can only be obtained through compile-time reflection where all the structuring and type information is available. It is clear now that handling the object state (including for a CORBA object) is very language dependent. Given that and following our previous experience [5-8], we choose Open C++ v2 [13] to implement our own runtime metaobject protocol and for its abilities to provide fine grained, language dependent, metainformations.

REFLECTIVE ARCHITECTURE AND FT-MOP

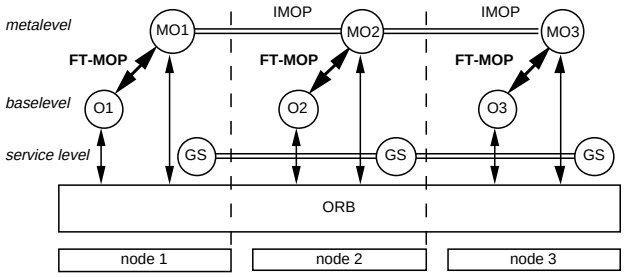

Fig. 1. Overall architecture

Our reference architecture, given in Fig. 1, is composed of a meta-level architecture for structuring fault-tolerant applications (CORBA), a collection of components implementing the fault-tolerant strategies (the metaobjects) and a metaobject protocol providing a nice interface between base and meta level(s), this is FT-MOP.

Mandatory services such as a group communication services (GS), replication domains management, error detection and object factories constitute the basic layer. Above this basic layer of CORBA services, metaobjects implement fault tolerance strategies. The FT-MOP controls the behavior and the state of base level CORBA objects (application level). Since metaobjects are CORBA objects they have also access to the ORB. The Inter MetaObject Protocol (IMOP) concerns the combination of several metaobjects at the metalevel; the composition of such metaobjects can be achieved either recursively (as in FRIENDS [5], or MAUD [3] since a metaobject is an object) or based on a meta-scheduler triggering several metaobjects within the same metalevel.

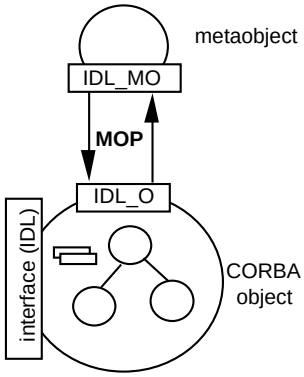

Fig. 2. Object-metaobject interaction with a MOP

In this architecture, FT-MOP is defined by two IDL interfaces (see Fig. 2): the object side and the metaobject side. The behavior observation consists in reifying the different aspects of our object model: instance creation and deletion, methods invocations and the object state. We describe in the following paragraphs how this information is reified.

Reifying methods invocation

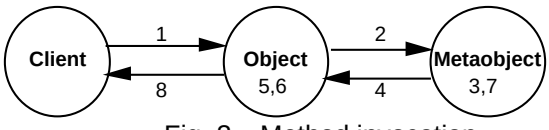

Fig. 3. Method invocation

During an invocation (see Fig. 3), the client invokes a method of the object, say `foo`, using the required parameters (1). The MOP traps this call, packs the parameters and call the `Meta_MethodCall` method of the metaobject (2) with the packed parameters. The metaobject may perform some actions before calling the base level method (3). Then, the metaobject calls the `Base_HandleCall` method of the object (4). This method, according to the method number, unpacks the parameters and call the original method (5), say

`real_foo`. The return value is packed and returned to the metaobject (6). The metaobject may execute some action before returning to the object (7) which in turns returns to the client (8).

Reifying creation-deletion

The instantiation process creates an object of a given class. Usually this creation process is initiated by another object or by a program (the main procedure in C++).

During the creation process, a new runtime metaobject has to be created for each new object. An instance of metaobject is created using a Metaobject Factory.

As illustrated in Fig. 4, the client requests the creation of an object, i.e. it calls the constructor (1). The constructor is executed but it has been translated in order to trap this call. It initializes the CORBA environment and calls the naming service for a reference to the Metaobject Factory (2). Then, the object constructor calls the Metaobject Factory to create a new metaobject of a given metaclass (3). The Metaobject Factory invokes the metaobject constructor (4). The metaobject constructors calls the original object constructor (5), i.e. `Base_Startup` which in turns activates `real_Startup` of the base level object. A reference to the metaobject is then returned to the object (6). Finally, a reference to the object is returned to the client.

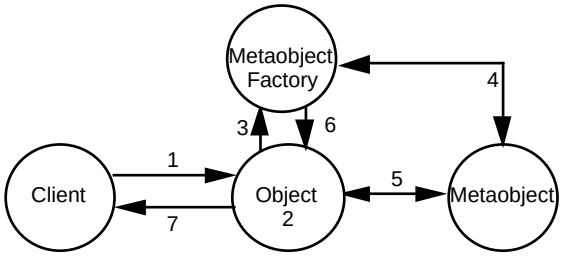

Fig. 4. Creation of participant

The metaobject protocol is able to control both before and after the real constructor (or destructor) of the object.

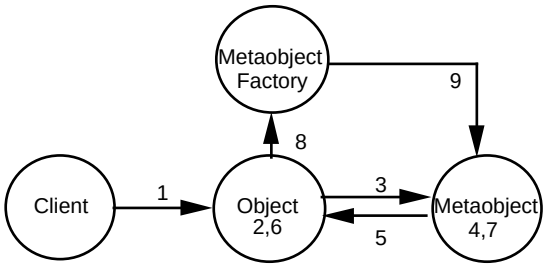

Fig. 5. Deletion of participant

Symmetrically to the creation process when the metaobject has to be created; during the destruction process (see Fig. 5), it must be deleted.

The client requests the deletion, i.e. it invokes the destructor (1) of the object. As previously explained, the destructor has been translated in order to trap this operation and thus behaves differently (2). The destructor calls the `Meta_Cleanup` method of the metaobject (3). The metaobject may execute some "pre-deletion" actions (4) and calls the original destructor, `real_Cleanup`, using the `Base_Cleanup` method of the base level object (5). The metaobject may now execute some "post-deletion" actions (7). Finally, the object invokes the Metaobject Factory to delete the metaobject just before to destroy itself (completion of the object destructor).

Reifying the object state

Our first objective is to handle the object state automatically and secondly to minimize, as far as possible, the state information to be transferred to object replicas [14]. This is language dependent and involves a deep analysis of the target source code using compile-time reflection. The runtime metaobject is responsible for handling a consistent copy of the base level object state. The *object state* corresponds to the whole set of attributes. From the initial state, the runtime metaobject can get the part of the state that has been updated after the execution of every method. This is called *delta_state*. The metaobject triggers method execution and gets the delta-state of the object.

The metaobject thus monitors the state evolution of the object it controls. The way the state information is handled by the metaobject is open and may vary according to the various ways of implementing a given fault tolerance strategy: the whole state can be sent to replicas after every method execution or only a sequence of delta-states can be sent from the initial state (notion of *incremental checkpointing*). The second approach may be far more efficient for some applications. This depends on several static and dynamic parameters (size of the object state, average percentage of the updated state, network bandwidth and load, processor load etc.).

A metaobject can build the object state by getting the initial state S_1 at object creation and then delta-states ΔS_i after each method invocation.

The main problem now is to obtain the delta-state ΔS_i after the i^{th} method invocation. To handle this problem, we use compile-time reflection and statically analyze the program. According to the complexity of the program, the results of this static analysis enable the appropriate technique to be selected for each method and each attribute of the object. Three techniques have been investigated :

- CTO - Compile-Time-Only: This technique aims at determining the delta-state at compile-time. Clearly, the delta-state cannot be determined statically, except for very simple method code. Mainly for didactic reasons, we use here compile-time reflection to detect a write access to an attribute. A compile-time metaobject insert some code in the method to produce the set of modified attributes, ΔS_i.

- CRT - Compile-Run-Time: In most case, the delta-state must be determined dynamically, i.e. which are the attributes that will change at runtime. Indeed, attribute modifications can be included in conditional or iterative blocks of code. Thus, these modifications depend on input parameters. The objective of the Compile-Run-Time technique is to compute in two phases (at compile-time and at runtime) the proper set of attributes ΔS_i for any method. The first phase, taking place at compile-time, consists in determining the set of attributes which are likely to change at runtime during a method invocation and in inserting just after each write access an instruction to flag the attribute. Instrumenting the code in this way enables during the second phase (at runtime) the exact set ΔS_i to be determined at the end of an invocation by checking the flags.
- CWS/CCS – Copy-Whole/Copy-Compare-State: Obviously, C++ is a permissive language: the programmer is able to use pointer arithmetic or to pass arguments by reference as function parameters (either using a pointer or a C++ reference). Such programming techniques generate "unpredictable modifications" of attributes and can lead to situations where the delta-state cannot be determined. For instance, when pointer arithmetic is used, any attribute can be accessed without compile-time reification of such a write access. For safety reasons, we assume here that all attributes change. Thanks to compile-time reflection such situation can be identified and computing the delta-state is done at runtime. All attributes before a method call are saved and compared to the new values when the method completes. Compile-time reflection is used here to insert code during code analysis for getting before-state, after-state and for computing the delta-state.

CONCLUSION

The reflective approach described in this paper enables strategies to be visible and easily customized at the application level, although their use is kept transparent for CORBA application programmers. The use of basic services, such as group communication, is devoted to metaobject programmers, specialized in a non-functional domain (here fault-tolerance). The implementation of fault tolerance using a metaobject protocol enables off-the-shelf ORBs to be used. Additionally, this approach provides a mean to develop fault tolerance software as any CORBA software with different object-oriented languages. This is a very good point for their reuse in many application domains. Being able to handle the object state automatically is the last benefit of the FT_MOP approach.

This work also illustrates the benefits of compile-time reflection for building a specialized runtime MOP. The same approach could be used to develop a specific runtime MOP in other application domains. Because, a compile-time MOP is a clever source-to-source translator, any recommended version of a compiler (as required in the industry) can be used to produce final binary code.

The use of the metaobject protocol described in this paper is in fact not limited to the implementation of fault-tolerant applications. Actually, this MOP may be used for handling other non-functional requirements, such as security (authentication) as done in the FRIENDS system. The automatic handling of the object state in combination with the runtime aspects of the MOP could also allow the migration of objects according to operational conditions. The dynamic linking between object and metaobjects enables as well mobile objects to gain fault tolerance and change strategies depending on the failure assumptions made on the underlying node.

FT-MOP has been partially implemented on COOL-ORB, a CORBA-compliant system available on UNIX. Handling the creation, deletion and invocation of CORBA object using FT-MOP has been implemented today. The full state capture has also been implemented; it supports simple data types, arrays and object composition whereas the delta-state capture supports today only attribute of simple types. Handling the object state of complex objects (arrays etc.) and performance measurements (benchmarking) is now carried out.

Two new Corba services have been implemented : a group communication service using xAMp [15] and an object factory that can create both application objects and metaobjects. We are working on the implementation of several fault-tolerant mechanisms as metaobjects, i.e. Corba objects.

ACKNOWLEDGEMENT

This work has been partially supported by the European Esprit Project n° 20072, DEVA, by a contract with FRANCE TELECOM (ref. ST.CNET/DTL/ASR/97049/DT) and by a grant from CNRS (National Center for Scientific Research in France) in the framework of international agreements between CNRS and JSPS (Japan Society for the Promotion of Science).

REFERENCES

1. Detlefs D., Herlihy M.P., Wing J.M., "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *Computer*, 21 (12), Dec. 1988, pp. 57-69.
2. Shrivastava S.K., Dixon G.N., Parrington G.D., "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, 8 (1), 1991, pp. 66-73.
3. Agha G., Frølund S., Panwar R., Sturman D., "A Linguistic Framework for Dynamic Composition of Dependability Protocols", in *Proc. of DCCA-3*, 1993, pp. 197-207.

4. Garbinato B., Guerraoui R., Mazouni K., "Implementation of the GARF Replicated Objects Plateform", *Distributed Systems Engineering Journal*, (2), March 1995, pp. 14-27.
5. Fabre J.C., Pérennou T., "A Metaobject Architecture for Fault Tolerant Distributed Systems: The *FRIENDS* Approach", *IEEE Transactions on Computers*, Special Issue on Dependability of Computing Systems, Jan. 1998, pp. 78-95.
6. Fabre J.C., Nicomette V., Pérennou T., Wu Z., Stroud R.J., "Implementing Fault-tolerant Applications using Reflective Object-Oriented Programming", in Proc. of FTCS-25, Pasadena, USA, June 1995, pp. 489-498.
7. Fabre J.C., Pérennou T., "FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications", in Proc. of EDCC-2, LNCS 1150, Taormina, Italy, Octobre 2-4, 1996, pp. 3-20.
8. Fabre J.C., "Desing and Implementation of the FRIENDS System", IEEE Workshop on Fault Tolerant Parallel Distributed Systems (FTPDS'98), Orlando, (USA), Avril 1998, pp. 604-622.
9. Landis S., Maffeis S., "Building Reliable Distributed Systems with Corba", Theory and Practice of Object Systems, (special issue on the future of Corba), vol. 3 (1), 1997, pp. 59-66.
10. Moser L.E., Melliar-Smith P.M., "The Interception Approach to Reliable Distributed CORBA Objects," P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Panel on Reliable Distributed Objects, *in 3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, (Or, USA), June 1997, pp 245-248.
11. Felber P., Garbinato B., Guerraoui. R., "Towards Reliable CORBA: Integration vs. Service Approach", *in Special Issues in Object-Oriented Programming*, Springer-Verlag, 1997.
12. Killijian M.-O., Fabre J.-C., Ruiz-García J.-C., Chiba S., "A Metaobject Protocol for Fault-Tolerant Corba Applications", to appear *in Proc. of Symposium on Reliable and Dependable Systems*, Vancouver, CA, Oct. 1998.
13. Chiba S., "A Metaobject Protocol for C++", in *Proc. of OOPSLA'95* (ACM Conference on Object-Oriented Programming, Systems, Languages and Applications), Austin (TX-USA), Oct. 1995, pp. 285-299.
14. Ruiz-Garcia J.C., Killijian M.O., Fabre J.C., Chiba S., "Optimized Object State Checkpointing using Compile-Time Reflection", EFTS'98 (IEEE Workshop on Embedded Fault Tolerant Systems), Mai 1998, Boston, USA, pp. 46-48.
15. Rodrigues L., Veríssimo P., "xAMp: A Protocol Suite for Group Communication", in Proc. Of the 11th IEEE Int. Symp. On Reliable Distributed Systems (SRDS-22), October 1992, pp. 112-121.