A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures

François Taïani[1], Jean-Charles Fabre, Marc-Olivier Killijian
LAAS-CNRS, 7 avenue du Colonel Roche
F-31077 Toulouse Cedex 4, France
f.taiani@computer.org, {jean-charles.fabre, marco.killijian}@laas.fr

This article was presented at the International Conference on Dependable Systems and Networks (DSN'2005), held in Yokohama, Japan, June 28 - July 1, 2005. It has been published in the proceedings of the aforementioned conference under the following reference:

François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian, *A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures*, The International Conference on Dependable Systems and Networks (DSN'2005), Yokohama, Japan, June 28 - July 1, 2005, pp.270-279

A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures

François Taïani[1], Jean-Charles Fabre, Marc-Olivier Killijian
LAAS-CNRS, 7 avenue du Colonel Roche
F-31077 Toulouse Cedex 4, France
f.taiani@computer.org, {jean-charles.fabre, marco.killijian}@laas.fr

Abstract

The past decade has seen an increasing use of complex computer systems made of third party components to develop mission critical applications. To insure the dependability of those systems in a sound and maintainable manner, technologies are needed to add fault-tolerance mechanisms transparently, while maintaining efficiency, high coverage, and evolvability. In this paper, we present a generic framework that addresses this problem and can be used within current industrial software. Our proposal is based on a limited set of core concepts inspired from plant biology and meta-object protocols. It provides separation of concerns for the implementation of adaptive fault tolerance strategies, while maintaining a global inter-level perception of the system runtime behavior. We demonstrate its practicality by using it to control the non-determinism of a CORBA/UNIX system.

1 Introduction

In computer-based systems that are built upon numerous software layers (OS, libraries, virtual machines, middleware, *etc.*), system complexity demands *the separation of fault-tolerance concerns* from the main functional design. Dependability being a *global* property, any solution to this problem must at the same time guaranty a *high coverage* of the system components by the chosen fault-tolerant mechanisms.

Computational Reflection has been proposed in the last decade as a powerful means to address this issue. Reflection is the ability of a computing system to act upon itself as part of its own computation [12]. In the context of fault-tolerance, the basic idea of reflection is to provide a system with additional observation and control abilities to permit (1) deviations from the specified behavior to be detected and then (2) recovery strategies to be put in place with limited intrusiveness to the system functions.

Architecturally, reflection introduces two distinct parts in a computing system (Figure 1): a base level where functional computation occurs, and a meta-level where fault-tolerance mechanisms are implemented using a self-representation of the system (which is called its meta-model). This meta-model can be regarded as a kind of "glue" that transparently connects the non-functional fault-tolerance mechanisms to the functional computation. In object-oriented systems, this transparent connection is usually structured in terms of objects (at the base level) and meta-objects (at the meta-level). Objects and meta-objects interact through what is known as a *Meta-Object Protocol* - MOP for short [8].

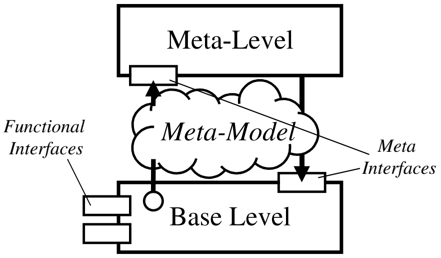


Figure 1. Architecture of a Reflective System

Although several works have shown that dependability (as well as other non-functional aspects) can benefit from reflection, major limitations prevent this technology from scaling up with system complexity, layering and information hiding. In a previous work [21] we have proposed a new framework termed *multi-level reflection* to overcome those limitations by extending reflection to complex multi-layer architectures (we will come back to it in more details in Section 2.2). We have shown that it was an appealing framework to address the development of fault tolerant strategies on top of complex system platforms. In

[1] François Taïani is now working as a lecturer at the Computing Department of Lancaster University (UK).

developing this new approach we *assumed* the existence of a Meta-Object Protocol (MOP) able to observe and control a complex system at multiple abstraction levels. *The design and implementation of such a "multi-level" MOP is a conceptual and technical challenge in its own right, which we address in this paper.*

The contribution of this paper is to show that, with a limited number of novel reflective concepts, and conventional object-oriented techniques, a multi-level Meta-Object Protocol can be developed which supports multi-level reflection on real-life industry grade platforms in an efficient, lightweight and minimally intrusive way.

This article is organized as follows: in Section 2, we briefly present some fault-tolerant systems based on reflective computing and discuss their limitations. In Section 3, we introduce the notion of semantic context to better understand the nature of complex software systems. Section 4 is then devoted to the actual presentation of our new meta-object protocol, which isbased on the novel notion of meta-markers. Sections 5 and 6 present an application of the proposed MOP to a CORBA-POSIX platform. Section 7 concludes the paper.

2 Problem statement and related work

2.1 Reflective fault-tolerant systems

Reflection has been used to develop fault-tolerant architectures based on object-oriented principles, and distributed computing. Platforms such as GARF [4], MAUD [1], and FRIENDS V2 [11] are representative of this trend. These systems use the reflective features of their implementation language (Smalltalk, HAL and OPENC++ respectively) to provide reflective capabilities dedicated to fault-tolerance and security. Using these reflective capabilities, a rich set of fault-tolerance and security mechanisms can be implemented within these architectures as reusable meta-level components.

Reflection has also been used to harden runtime executives, such as real-time micro-kernels . Arlat *et al.* [2] for instance have proposed a framework in which the internal behavior of a real-time micro-kernel is partially exposed as a set of temporal logic predicates. The predicates provide an abstract representation (meta-model) of the micro-kernel computation. Once combined into temporal logic formulae, they can be compiled into meta-level wrappers for error confinement and error recovery in order to enhance the original kernel's dependability.

Interestingly, these examples all use the reflective capabilities of a single abstraction level: the SmallTalk and C++ languages respectively for GARF and FRIENDS, the actor and communication concepts of the HAL actor language for MAUD, and kernel abstractions (events, signals, queues, locks, timers) for reflective real-time micro-kernels. Being single-level, these approaches are blind to any information or behavior that is not contained within the target abstraction level. This limitation strongly hinders the range of fault-tolerant mechanisms that can be developed using single-level reflective approaches, in particular in complex multi-layered systems, as we will see in the next section.

2.2 Problem statement

To better understand the limitations mentioned above and to illustrate the problem we want to tackle, let's take a core example that captures the multi-level nature of the information required to master fault tolerant computing.

Consider for instance the active replication of a multi-threaded server. Because active replication requires replicated nodes to behave deterministically, the control of the non-determinism induced by multi-threading has been an active research area in recent years [14, 7, 3, 13]. One popular approach (under minimally constraining assumptions) is to insure that all replicas follow the same mutex acquisition pattern. This can be done either by (i) serializing incoming requests [14], or (ii) using a deterministic scheduler [7], or (iii) imposing a consensus on mutex acquisition decisions [3, 13].

These solutions are very hard to realize with a single-level reflective approach limited to a higher system level (programming languages, application libraries, *etc.*). Language-based reflection for instance is limited to the programming model of its underlying language. In most programming languages neither context switches, nor critical sections, nor lock allocations (semaphores, mutexes) are explicitly visible. A reflective approach based on those languages cannot provide enough information on the underlying runtime to control non-determinism [10]. More generally, higher system-level abstractions lack too much information about low-level layers (hidden states, non deterministic events, detailed implementation choices) to permit the implementation of efficient and powerful fault-tolerance mechanisms.

A single-level reflective approach limited to lower system levels (OS, system libraries) does not solve the problem either, as shown on Figure 2. On this Figure, the above-mentioned solution (iii) is implemented using OS-level reflection. Replication is realized as a reflective component that intercepts mutex activity and insures that all replicas follow the same consistent behavior. (Note that we assume here that all mutex lock and unlock operations translate into OS-level operations that can be intercepted.) Unfortunately, in a complex multi-layer system, this approach rapidly becomes intractable in practice because of the high number of mutex operations involved. For example, we recorded that the commercial CORBA implementation ORBACUS [6] could make up to 203 mutex calls (i.e. calls to functions of the form *pthread_mutex_xxx*) for only one CORBA request. Other popular CORBA middleware like TAO [18] and OMNIORB [5] don't make as many mutex operations, but their numbers remain fairly high (respectively 52 and 64 calls). Modern locking technology (fast user locks, "thin" locks) insures that those numerous mutex operations don't have a high performance cost *locally*. However, *replicating* these operations introduces a communication overhead that in some cases may be intractable. Napper *et al.* report a 375% overhead in execution time when this approach is used on a Java Virtual Machine running a database benchmark, a performance price too high to pay in most cases [13].

This scaling problem is not due to a major flaw in the approach itself, but to the lack of any global system vision. The key observation is that most lock operations inside the middleware have no influence on the observable determinism of the platform, and don't need to be replicated. The main challenge is thus to tell relevant locks from the remaining ones. This distinction, however, goes far beyond the OS programming model, and is not available using only OS-level reflection.

Low-level reflection limits the efficiency of fault-tolerance mechanisms because it can only provide a low-context view of a system, and lacks global semantics about the current system activity.

2.3 Multi-level reflection

To overcome the limitations of single-level reflection, we have proposed *Multi-Level Reflection* in prior works [21, 20]. The key idea is to combine OS-level reflection with information obtained from the middleware and application layers in order to provide a powerful programming model for fault-tolerance computing. In the case of the previous example, multi-level reflection allows the identification of the mutexes that are really relevant for the platform determinism (Figure 3). By selecting those mutexes only, it tremendously reduces the amount of observation that needs to be distributed among the replicated nodes, and allows approaches for the control of non-determinism to scale up to realistic multi-layer platforms.

More generally, multi-level reflection is based on the observation that a multi-layer system contains several overlapping programming logics. These logics offer different viewpoints on the system that correspond to different abstraction levels. At the OS communication API, for instance, the network activity of a CORBA [15] process is viewed as a succession of socket creations, deletions, sent and received packets. Inside the middleware, the system is seen in terms of received requests, distributed events, and marshaled parameters.

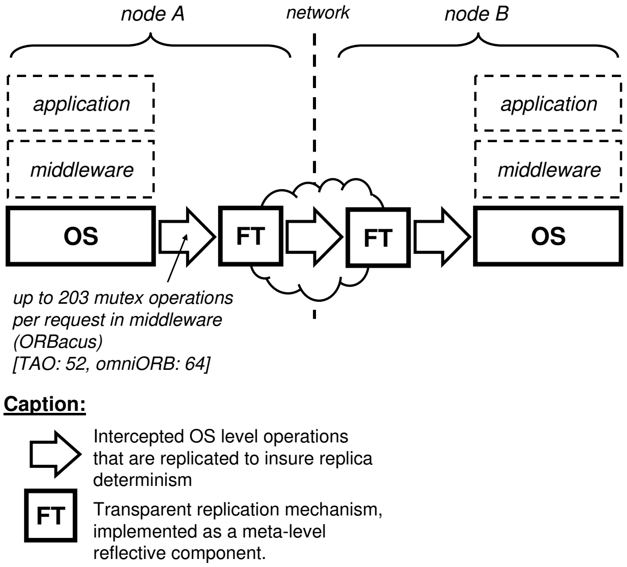

node A
network
node B
application
middleware
OS
FT
FT
application
middleware
OS
up to 203 mutex operations per request in middleware (ORBacus) [TAO: 52, omniORB: 64]
Caption:
Intercepted OS level operations that are replicated to insure replica determinism
FT
Transparent replication mechanism, implemented as a meta-level reflective component.

Figure 2. Controlling non-determinism due to multi-threading using single-level reflection

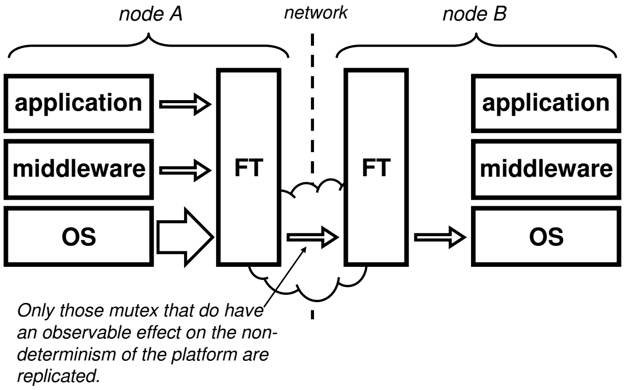

node A
network
node B
application
middleware
OS
FT
FT
application
middleware
OS
Only those mutex that do have an observable effect on the non-determinism of the platform are replicated.

Figure 3. The example of Figure 2 addressed using multi-level reflection

Each level of abstraction (OS kernel, system libraries, middleware, application) can be reified into a corresponding meta-model that exports the information that is available at this level. However, as illustrated in the previous section, each of these single-level meta-models is limited to the information that is available at its corresponding abstraction level: a meta-model that uses an open compiler to reify the language constructs of the application code cannot give any information about the thread management if threading is not a first class entity of the considered language. In the same vein, reifying the mutex-activity of the multi-threading library gives no indication of the request currently being processed by the middleware, because the notion of remote method invocation is not included in the programming model of the operating system.

Multi-level reflection builds on the complementary natures of the high and low levels found in a complex system to extend their individual reflective capabilities. In our previous work we have presented the principles and the general architecture of multi-level reflection. We did *not* however propose any concrete implementation mechanism to support our vision. The contribution of this paper is to present a concrete meta-object protocol that directly supports multi-level reflection on realistic industry-grade platforms. Our design is based on the novel notion of semantic context, which we present in the next section.

3 A Context aware approach

In this section, we introduce the *semantic context* of a low-level operation. This new concept aims at capturing the nature of interactions between higher and lower abstraction levels in complex multi-layer systems. It will be the starting point of our new MOP design.

Consider the OS API of a complex multi-layer system. Any invocation of the OS API is the result of some computation process within the higher layers of the system. Backtracking this computation process makes it possible to understand the original purpose of a low-level OS operation *in the context* of the higher-level activities. One straightforward way of backtracking is for instance to introspect the invocation stack of the thread that is invoking an OS operation. In a multi-layer software such a stack typically spans several layers and in many cases provides an accurate understanding of the system current behavior.

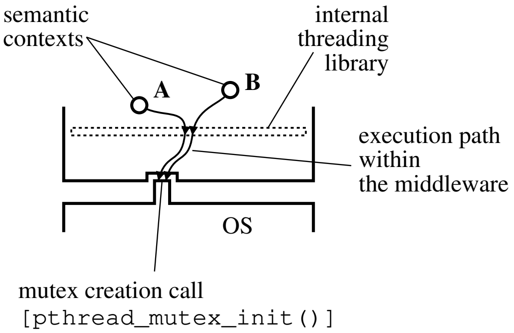


Figure 4. Semantic context of a low-level platform call (here *pthread_mutex_init()*)

Based on the core example of Section 2.2, Figure 4 shows how this approach can be applied on a POSIX platform to control the non-determinism induced by multithreading. In Figure 4, **A** is some location within the middleware code where a C++ object ***a*** is instantiated. ***a*** supports automatic garbage collection. During its construction, a mutex protecting its reference counter is created (*pthread_mutex_init()* on the figure). Because the middleware is internally layered, *pthread_mutex_init()* is not directly invoked by ***a***'s constructor, but instead is the result of a sequence of nested calls.

On the same figure, **B** is some other location within the middleware code where a queue structure ***b*** is initialized. This queue is used to buffer incoming remote requests. Also for thread safety, access to this queue (to add and remove requests) is protected by a mutex, so that the instantiation of ***b*** also causes *pthread_mutex_init()* to be invoked.

The mutexes created for ***a*** and ***b*** don't serve the same purpose, and don't have the same effect on the determinism of the middleware. The mutex created in Context **A** insures serialization of reference counter operations. It does not need to be controlled to insure determinism as it has no observable effect at the application level. The mutex created in Context **B** on the other hand determines the ordering in which requests are dispatched to the application level. It must be controlled to insure determinism [21]. Those two situations cannot be distinguished at the OS level, since by default no information is available regarding the context in which those two mutexes are created and will be used. They cannot be distinguished statically inside the middleware code either, since the same code containing *pthread_mutex_init()* is used to create both mutexes. This is due to the internal layering of middleware platforms, which typically contain their own internal threading libraries (ACE for TAO, JTC for ORBACUS, *etc.*). Observing the stack state of the current thread, however, allows tracking back the sequence of nested invocations that resulted in *pthread_mutex_init()* being invoked, and thus enables pinpointing the location in the middleware code (**A** or **B**) where the purpose of the mutex invocation becomes explicit. Using the terminology of aspect-oriented programming [9], we will call this kind of location (**A** or **B**) *the semantic joint point* of *pthread_mutex_init()*.

As any causality related notion, the concept of semantic context is recursive. The semantic context of a low-level operation must thus be defined with respect to what we will term a *semantic level*. A semantic level is a collection of possible semantic joint points (i.e. source code locations) within a platform, so that any low-level operation can be tracked back to at least one of the semantic joint points. On Figure 4, the semantic level used to determine **A** and **B** would contain most of the ORB main classes (request queues, thread pools, requests) along with code related to high-level non-functional mechanisms such as garbage collection.

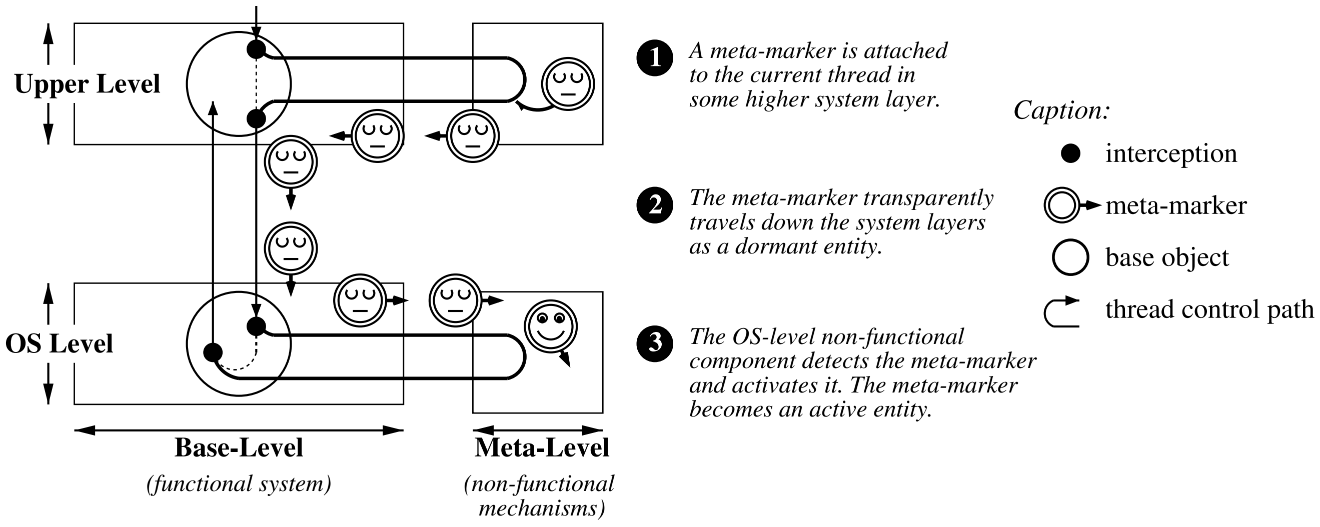

Upper Level
OS Level
Base-Level
(functional system)
Meta-Level
(non-functional mechanisms)
1 A meta-marker is attached to the current thread in some higher system layer.
2 The meta-marker transparently travels down the system layers as a dormant entity.
3 The OS-level non-functional component detects the meta-marker and activates it. The meta-marker becomes an active entity.
Caption:
● interception
meta-marker
base object
thread control path

Figure 5. Inter-level coordination of non-functional activity using meta-markers

4 A multi-level MOP for fault tolerance

In this section, we present the new multi-level meta-object protocol (MOP) we have developed to support multi-level reflection. This new MOP is based on the semantic context notion we have just presented, and on a new biology-inspired construct termed *meta-marker*. Meta-markers allow the transparent transport of meta-information between different abstraction levels. As such they build the core of our new MOP.

4.1 Drought, phytohormones and meta-markers

In order to support multi-level reflection, our MOP should reify semantic contexts at all abstraction levels of the targeted system. To achieve this objective in a lightweight, efficient and non-invasive manner, we derive the notion of meta-marker from plant biology. Most plants, and in particular trees, implement regulation loops between their root system and their foliage using chemical markers (phytohormones) that travel through their sap. Abscisic Acid (ABA) for instance is a phytohormone that is used to regulate water consumption in the leaves during drought periods [17]. Acting as a signal of reduced water availability in the soil, it is synthesized at the root level before it is transported by the sap to trigger appropriate leave responses (stomatal closure, leaf decay, *etc.*). In our MOP, meta-markers play a similar role. In the same way abscisic acid is transported "transparently" by a plant's sap, meta-markers rely on threads to convey meta-information transparently between the higher- and lower-levels of a complex software system.

More concretely, a meta-marker is an object that can be attached and detached from a thread[1] to realize multi-level non-functional mechanisms.

```
class MetaMarker {
  void attachToThread() ;
  void detachFromThread() ;
};
```

Figure 5 illustrates how meta-markers can be used to coordinate non-functional activities between the heterogeneous levels of a complex multi-layer system. First a meta-marker is attached to the current active thread in some higher layer of the system (application logic, business-oriented middleware). As an object, this meta-marker encapsulates information regarding the current behavior of the system as observed at this high abstraction level. As the thread travels down the layers of the system, the meta-marker travels along. It remains transparent to any piece of code that is unaware of it ("it is dormant"), thus guarantying a strong separation of concerns between functional and non-functional concerns.

When the thread reaches the OS layer and invokes an OS operation, this invocation is intercepted by the OS reflective facility. The interception mechanics of the OS detects the presence of the meta-marker and activates it. Once activated, the meta-marker can trigger appropriate actions at the OS level to contribute to some non-functional mechanism, on the basis of the high-level contextual information it transports. By doing so, it creates a transparent coordination channel between non-functional components located at different abstraction levels.

[1] The idea relies on piggybacking facilities that enable information to be attached to threads at runtime, a feature that can be found in most multi-threading environments. In POSIX, for instance, this corresponds to the functions *pthread_setspecific / pthread_getspecific*.

Because they establish a communication channel between different abstraction levels within a complex system, meta-markers are ideal to transport semantic contexts. In the core example of Section 2.2 for instance, mutexes that are relevant for the determinism of the platform can be identified based on the semantic context of their creation. They can thus be selected using meta-markers. Once selected, appropriate measures can be taken to insure that any future operation on these selected mutexes will be intercepted. To implement this approach, two problems must be solved:

(P1) the semantic context of each mutex creation must be captured and attached to threads as a meta-marker;

(P2) once attached to a thread, this meta-marker should select the mutexes that are relevant for determinism, and insure that future invocations on these mutexes will be reified to fault-tolerance components.

4.2 Meta-markers and semantic contexts

In order to address Problem **P1**, we first need to identify an appropriate set of semantic joint points (cf. Section 3) within the source code of the middleware. Consider for instance the following code extract:

```
init_and_run_middleware(..) {
  init_request_queue(..) ;
  init_some_refcount_object(..) ;
  ...
  run_ORB();
}
```

Assume this represents the initialization code of some middleware written in C/C++. This code example does not contain any object for simplicity reasons, but a true object-oriented example would work the same. *init_request_queue(..)* initializes the queue that buffers incoming requests. The mutexes that this function creates are relevant for the control of determinism. The remaining lines (*init_some_refcount_object(..)*, ..., *run_ORB()*) also initialize mutexes, but those are not relevant for determinism. These lines are the semantic joint points that capture the mutex semantics of the middleware. The line *init_request_queue(..)* represents the semantic context of mutexes relevant for the platform determinism. The remaining lines represent the semantic context of mutexes not relevant for the platform determinism.

Because we don't know exactly where and when mutexes will be initialized in *init_request_queue(..)*, we annotate the middleware source code around *init_request_queue(..)* with meta-marker activation. This is shown on Figure 6.

```
MetaMarkerForRelevantMutex myMMarker() ;

myMMarker.attachToThread() ;
init_request_queue(..) ;
myMMarker.detachFromThread() ;
```

Figure 6. Semantic contexts as meta-markers

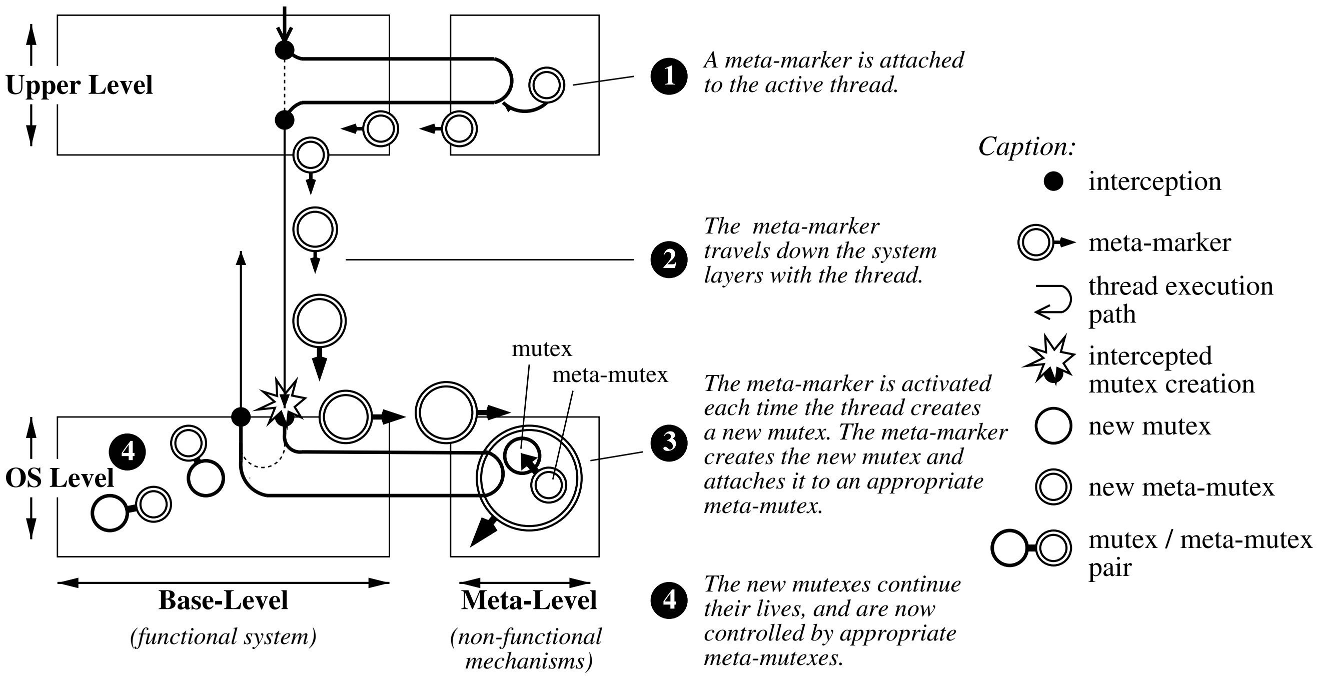
MetaMarkerForRelevantMutex is a specialized subclass of *MetaMarker*. In the code extract, an instance of *MetaMarkerForRelevantMutex* is attached to the thread executing *init_request_queue(..)* and will be present until this function returns. Assuming an appropriate set of semantic joint points can be found in the middleware source code, this approach solves Problem **P1**.

4.3 Meta-Markers in Action

To address Problem **P2**, we assume our OS has basic reflective capabilities, similar to those found in the reflective OS MUSE [22] (we explain in Section 5 how this can be done on top of POSIX). In particular, *MetaMutex* meta-objects can be attached dynamically to one or more mutexes. A *MetaMutex* transparently intercepts any operation invoked on the mutex it is associated with. It can thus modify a mutex behavior, e.g. replicate mutex operations across replicated nodes.

Figure 7 describes how this can be used to solve Problem **P2**. On this figure the meta-marker used to solve Problem **P1** (*MetaMarkerForRelevantMutex*) is attached to the active thread, and travels down the system layers (Steps 1 and 2). When the thread reaches the OS and attempts to create a new mutex, an interception library diverts the mutex creation call, and tests for the presence of this meta-marker on the thread. If no meta-marker is present, control is returned to the base-level OS, and the normal mutex creation occurs. If a meta-maker is detected, the mutex creation is delegated to the meta-marker, which decides what to do. In this case the meta-marker creates the new mutex and attaches it to an appropriate meta-mutex (Step 3). It then gives control back to the system, returning the new mutex that has just been created, as the normal mutex creation does. The mutexes that are instrumented in this manner continue their life at the OS level, but are now controlled by appropriate meta-mutexes. These meta-mutexes will intercept any future operations on the mutexes they control and divert these operations to the fault-tolerance level (meta-level).

The mechanism shown on Figure 7 is not limited to mutexes, and can be used each time the interception of some low-level entities (mutexes, but also sockets) requires to be conditioned by the activity of the system as observed at some higher abstraction level. This defines a new flavor of meta-object-protocols that can be used to provide



1 *A meta-marker is attached to the active thread.*

2 *The meta-marker travels down the system layers with the thread.*

3 *The meta-marker is activated each time the thread creates a new mutex. The meta-marker creates the new mutex and attaches it to an appropriate meta-mutex.*

4 *The new mutexes continue their lives, and are now controlled by appropriate meta-mutexes.*

Figure 7. Meta-markers in action

a powerful yet disciplined programming model for fault-tolerance on complex multi-layer architectures.

5 Implementation issues

To validate the concrete value of meta-markers on real-life platforms, we implemented a solution to the example of Section 2.2 using the mechanisms we have just presented. Our solution is a generic C++ library that can be used to control the determinism of any C++ CORBA implementation running on a POSIX compliant OS. In this section we report on this library, and in the next section we will explain how we used it to instrument a commercial CORBA product (ORBACUS) running on a LINUX OS.

Our C++ library implements the interface presented on Figure 8. We showed in a previous work that this kind of meta-interface allows to control the determinism of a request-oriented middleware in a multi-level reflective manner [21]. Basically this meta-interface reifies the events of a request life cycle that are relevant to the determinism of the middleware, and makes these events available to fault-tolerant mechanisms. It covers three main facets of a request lifecycle: *communication* (reception, reply), *control path* (reaching and leaving the application code), and *synchronization* (“request contention points”). *Request contention points* correspond to the mutexes that are relevant for determinism. When the middleware attempts to lock a relevant mutex during the processing of a request, the lock operation is intercepted and *requestBeforeContentionPoint* is called. Symmetrically, when the middleware tries to release the lock, the release is intercepted, and *requestAfterContentionPoint* is called.

```
class MetaRequestLifecycle {

  /** Communication **/
  requestHasBeenReceived (RequestID);
  replyHasBeenSent       (RequestID);

  /** Control Path **/
  requestBeforeApplication (RequestID);
  requestAfterApplication  (RequestID);

  /** Synchronisation **/
  requestBeforeContentionPoint
    (RequestID, RequestContentionPoint);
  requestAfterContentionPoint
    (RequestID, RequestContentionPoint);
};
```

Figure 8. The multi-level meta-interface to control non-determinism

We implemented the communication and synchronization facets of the meta-interface using active meta-markers. Global request IDs similar to those found in the FT-CORBA standard [16] were also implemented with meta-markers. The control path facet was not directly implemented in the library. We will address it in the next section.

The main effort in building the library went into provid-

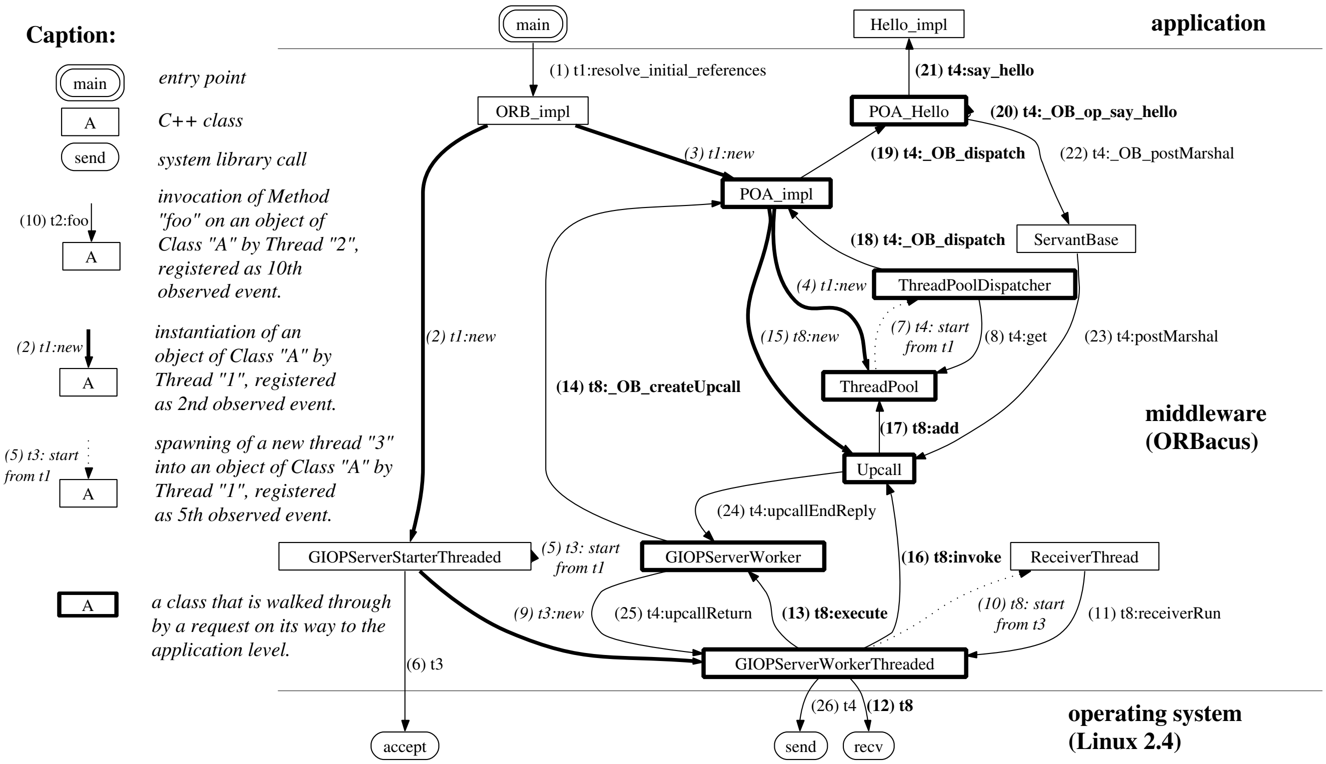


Figure 9. High-level representation of the request processing in ORBACUS

ing basic OS-level meta-objects on top of a POSIX interface using library interception. Using the LD_PRELOAD mechanism and the *dlopen* interface, our library provides the classes *MetaSocket* and *MetaMutex* to intercept and redefine the behavior of sockets and mutexes at the OS level. It provides an abstract *MetaMarker* class to realize the attach / detach operations of the meta-marker mechanism. Two concrete meta-marker subclasses are based on this abstract *MetaMarker* class: *MetaMarkerForMutex* intercepts mutex creations, and *MetaMarkerForSocket* intercepts socket creations. *MetaMarkerForMutex* corresponds to the mechanism showed on Figure 7, and *MetaMarkerForSocket* does the same for sockets. *MetaMarkerForSocket* was needed to implement the communication facet, as we will see.

These basic OS-level meta-objects and these different meta-marker classes are the basis of the implementation of the communication and synchronization facets. Our library realizes the synchronization facet with two new classes: *RequestContentionPoint* and *ContentionPointFactory*. *RequestContentionPoint* is a subclass of *MetaMutex* that diverts all the operations made on a mutex to the methods *requestBeforeContentionPoint(..)* and *requestAfterContentionPoint(..)* of the interface *MetaRequestLifecycle* (Figure 8). *ContentionPointFactory* is a particular *MetaMarkerForMutex* that attaches a *RequestContentionPoint* to each new mutex created by the thread it is attached to (Step 4 of Figure 7).

The communication facet is realized in a similar way with the classes *MetaMarkerForSocket* and *MetaSocket*. In a CORBA process, sockets can be used to transport information that is not related to CORBA (HTTP traffic, the X11 windowing protocol, *etc.*). As for mutexes, CORBA relevant sockets must be intercepted. The same meta-marker mechanism shown on Figure 7 for mutexes can thus be applied to sockets to realize the *requestHasBeenReceived(..)* and *replyHasBeenSent(..)* methods of the meta-interface of Figure 8.

We also used meta-markers in the communication facet to transport global request IDs across the different layers of the system in a bottom up fashion: global request IDs are piggybacked on requests, unmarshaled when the request reception is intercepted, and encoded on the transporting thread as a meta-marker.

6 Instrumenting an industrial platform

We have used the library we have just described to instrument a commercial CORBA product with the meta-interface of Figure 8. The key element in instrumenting

```
(meta-corba) New meta-socket for socket 8 (meta-corba) In
LAAS::CORBARequestMetaSocket::accept (meta-corba) (6439) Received
Request with ID: 1:6436 (meta-corba) (6439) Before Contention Point 1
with RequestID: 1:6436 (meta-corba) (6439) After Contention Point 1
with RequestID: 1:6436 (meta-corba) (6439) Before Contention Point 2
with RequestID: 1:6436 (meta-corba) (6439) After Contention Point 2
with RequestID: 1:6436 (meta-corba) (6427) Before Contention Point 1
with RequestID: 1:6436 (meta-corba) (6427) After Contention Point 1
with RequestID: 1:6436 (meta-corba) (6427) Before application with
RequestID: 1:6436 Hello World!: 1 (meta-corba) (6427) After
application with RequestID: 1:6436 (meta-corba) (6427) Sending Reply
for Request with ID: 1:6436 (meta-corba) Destroying a meta-socket
CORBARequestMetaSocket
```

Figure 10. A trace of ORBacus' request processing as intercepted by our library

a concrete platform is to identify a semantic level where the higher-level semantic of mutex and socket use is explicitly apparent (Section 3). This cannot be done on black box software, as it requires at least some high-level understanding of the platform's internals and of inter-component interactions.

Using a dedicated reverse engineering tool [19] to guide us, we were able to extract high-level behavioral models of several popular CORBA implementations: TAO, omniORB and ORBacus. Figure 9 shows the high-level model we obtained for ORBacus. Many intermediate classes and several internal libraries were abstracted away to outline the core of the request processing. This kind of model focuses on the path followed by CORBA requests as they travel up and down the middleware. On Figure 9, Invocation 6 (*accept)* by Thread *t3* is where new CORBA related sockets are created, and Invocations 17 (*add)* by Thread *t8* and 8 (*get)* by Thread *t4* are where requests encounter a contention point. The mutexes related to this contention point are created along with the object *ThreadPool* when Invocation 15 is executed by Thread *t1*.

Using this high-level model, we were able to identify the semantic joint points where to introduce the necessary meta-markers. We then used the approach illustrated on Figure 6 to annotate those semantic joint points and "hook" our generic multi-level library into ORBacus. The following code extract shows how we introduced contention point reification into the middleware implementation:

```
// Extra code
contentionPointFactory.attachToThread();

// Original ORBacus code
pools_[i] = new ThreadPool(i, nthreads);

// Extra code
contentionPointFactory.detachFromThread();
```

Using this approach we were able to "weave" the communication and synchronization facets of the meta-interface *MetaRequestLifecycle* into ORBacus with only 20 extra lines added to its runtime library (i.e. less than 0.02% of the original library code!): 4 lines for the synchronization facet, 6 for the communication facet, 3 for the global request IDs, the remaining 7 being "*#include*" statements. Because this weaving process relies on a high-level representation of the middleware, it is highly robust to change, and can be easily ported to new platforms, or adapted as the middleware evolves into new versions.

The "control path" facet of the meta-interface *MetaRequestLifecycle* is easier to realize than the other facets. It does not require any inter-level coordination, so we decided to simply modify the IDL compiler of ORBacus to insert the appropriate code into the generated skeletons that connect the middleware to the application layer. The modification was quite easy and straightforward, adding only 13 lines to the source code of ORBacus's IDL compiler.

Figure 10 shows the result of this instrumentation when the meta-interface *MetaRequestLifecycle* is simply used to trace the request processing. On this trace, we see that only two contention points (numbered 1 and 2) are hit only 3 times by a request being processed. Those 3 hits correspond to the 3 underlying mutex operations that, according to our analysis, must be intercepted to insure the determinism of the middleware layer. This is a 67-fold improvement over the 203 mutex operations per request which we reported on in the introduction of this article (Section 2.2).

7 Conclusion

As complex software systems integrating many third party components are increasingly used for mission critical applications, flexible engineering approaches are needed to address the dependability of those platforms in a maintainable, adaptable, and principled manner.

In this paper we have presented a new meta-object protocol (MOP) that addresses this technical challenge. Our MOP is based on a core set of new concepts (*semantic* *contexts*, *semantic joint points*, and *meta-markers*) inspired from traditional meta-object protocols and from our prior work about multi-level reflection. The key idea is to permit the transparent coordination of meta-level components that are located at different abstraction levels of a complex multi-layer system. The result is a generic framework for fault-tolerance computing that provides a high degree of separation of concerns and uplifts the limitations of former proposals.

More generally this work can be seen as a transposition to multi-layer architectures of Aspect-Oriented Programming notions. In the near future, we plan to investigate more deeply this promising connection.

Acknowledgments

This work was partially supported by the European IST Project "Dependable Systems of Systems" n.1999-11585. The authors would like to thank the anonymous referees for their detailed reviews. François Taïani would also like to thank Matti Hiltunen from AT&T for his kind feedback on early versions of this article.

References

[1] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Transactions of the Third IFIP Conference on Dependable Computing for Critical Applications (DCCA-3)*, pages 197–207, Palermo (Sicily), Italy, 1992. Elsevier.

[2] J. Arlat, J.-C. Fabre, M. Rodriguez, and F. Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, Feb. 2002.

[3] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *The International Conference on Dependable Systems and Networks (DSN-2003)*, pages 149–158, San Francisco, CA, 2003. IEEE Computer Society.

[4] B. Garbinato, R. Guerraoui, and K. R. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal*, 2(1):14–27, 1995.

[5] D. Grisby, S.-L. Lo, and D. Riddoch. *The omniORB version 4.0 User's Guide*. Apasphere Ltd. & AT&T Laboratories Cambridge, July 2004.

[6] *Orbacus User's Guide V.4.2.0*. IONA Technologies PLC, June 2004.

[7] R. Jiménez-Peris, M. P. Mart'inez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 164–173, Nürmberg, Germany, 2000.

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

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[10] M.-O. Killijian and J.-C. Fabre. Implementing a reflective fault-tolerance CORBA system. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 154–163, Nürnberg, Germany, 2000.

[11] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-García, and S. Shiba. A metaobject protocol for fault-tolerant CORBA applications. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 127–134, West Lafayette (USA), 1998.

[12] P. Maes. Concepts and experiments in computational reflection. In N. K. Meyrowitz, editor, *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 147–155, Orlando, Florida, Oct. 1987. ACM.

[13] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, June 2003.

[14] P. Narasimhan, L. E. Moser, and P. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *18th Symposium on Reliable Distributed Systems (SRDS)*, pages 263–273, Lausanne, Switzerland, 1999. IEEE.

[15] *Common Object Request Broker Architecture: Core Specification (Version 3.0.1 - formal/02-11-01)*. Needham, MA, U.S.A., 2002.

[16] *Common Object Request Broker Architecture: Core Specification (Version 3.0.2 - formal/02-12-02)*, chapter 23. Fault Tolerant CORBA. Needham, MA, U.S.A., Dec. 2002.

[17] A. Sauter, W. Davies, and W. Hartung. The long-distance abscisic acid signal in the droughted plant: the fate of the hormone on its way from root to shoot. *Journal of Experimental Botany*, 52(363):1991–1997, Oct. 2001.

[18] D. Schmidt and C. Cleeland. Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine*, 16(4), Apr. 1999. Special Issue on Design Patterns.

[19] F. Taïani. COSMOPEN: A reverse-engineering tool for complex open-source architectures. In *DSN-03 supplemental volume, Student Forum of DSN'03*, pages A49–A51, San Francisco, CA, June 2003.

[20] F. Taïani, J.-C. Fabre, and M.-O. Killijian. Principles of multi-level reflection for fault-tolerant architectures. In *2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, pages 59–66, Tsukuba (Japan), 2002.

[21] F. Taïani, J.-C. Fabre, and M.-O. Killijian. Towards implementing multi-layer reflection for fault-tolerance. In *The International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA, June 2003.

[22] Y. Yokote, F. Teraoka, and M. Tokoro. A reflective architecture for an object-oriented distributed operating system. In S. Cook, editor, *Third European Conference on Object-Oriented Programming (ECOOP'89)*, pages 89–106, Nottingham, UK, 1989. Cambridge University Press.