Towards Implementing Multi-Layer Reflection for Fault-Tolerance

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

This article was presented at the *International Conference on Dependable Systems and Networks* (DSN'2003), held in San Francisco (CA) on June 22nd - 25th, 2003. It has been published in the proceedings of the aforementioned conference under the following reference:

François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian, Towards Implementing Multi-Layer Reflection for Fault-Tolerance, proceedings of the the *International Conference on Dependable Systems and Networks* (DSN'2003), San Francisco (CA), June 22nd - 25th, 2003, pp. 435-444

Towards Implementing Multi-Layer Reflection for Fault-Tolerance

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

Abstract

Third party software is now increasingly used in systems with high dependability requirements. This evolution of system development raises new challenges, in particular regarding the implementation of fault-tolerance. As systems are often built of black-box components, some crucial aspects of their behavior regarding replication cannot be handled. This is also true to some extent for open-source components as mastering their internal behavior is sometimes very tricky (e.g. OS and ORBs). During the last decade reflection has emerged as a very fruitful paradigm for the disciplined management of non-functional aspects, among which fault-tolerance. In this paper we discuss how to apply reflection to multi-layer systems for implementing fault-tolerance in an independent and principled manner. We analyze the connections between the underlying assumptions of fault-tolerance strategies and the different layers of a system. Based on this multi-layer analysis we show how the requirements of a family of replication algorithms can be addressed on a concrete architecture, resulting in what we name Multi-Layer Reflection.

1. Introduction

Flexibility, reuse, and adaptation are becoming key aspects of today's large computer systems (satellite systems, transport, automotive), and explain the increasing use of component-based approaches (including COTS). This trend raises two challenges when considering the dependability of the resulting systems: How can we build dependable systems from components that don't specifically target dependability concerns? What are the dependability figures of the resulting systems? We focus in this paper on the first question, and more particularly on the implementation of fault-tolerance into systems made of third party software components. Fault-tolerance is very difficult to achieve without a minimal understanding and control of the internal structure and behavior of the considered systems. This implies intrusion within system components, which is very problematic. For this reason, integrators are looking for sound and principled approaches that help them separate functional development from fault-tolerance concerns, within large projects, over long life cycles.

Computational Reflection [11], an architectural paradigm that appeared in the late eighties, and related technologies such as aspect oriented programming, appear as very promising approaches to tackle this issue. Using reflection to implement fault-tolerance into multi-component systems induces, however, several sub-problems. Reflective architectures are centered on a key element, their meta-model, that ensures the separation of concerns between the "base" system (here the system resulting from component integration) and the mechanisms (here fault-tolerance) that are added to the base system. To be effective, this meta-model must take into account both the multi-component nature of the system and the requirements of fault-tolerance that it should help implement. In this paper, we address this dual issue and propose a methodology to help designing meta-models that specifically target the implementation of fault-tolerance into systems made of third party components.

The paper is organized as follows. Section **2** briefly recalls essential notions regarding computational reflection and introduces the steps of our approach. Section **3** proposes a requirement analysis of a set of well-known replication strategies from a reflective perspective. Based on a small example, Section **4** shows how this analysis can be applied to a concrete system architecture made of several components. This discussion leads us to the notion of Multi-Layer Reflection (MLR). Section **5** further develops the practical use of this notion by presenting, on a concrete architecture (CORBA and POSIX based), how the requirements obtained in Section **3** lead to the precise specification of a meta-model that is both optimized for fault-tolerance and the considered system structure.

2. Computational Reflection

A reflective system is basically structured around a representation of itself —or *meta-model* — that is causally connected to the real system [11]. This approach divides the system into two parts: a *base-level* where normal computation takes place, and a *meta-level* where the system computes about itself (*meta-computation* or *meta-level software*). (See Figure 1)

The meta-model is structured around notions that are major (runtime) elements of the base level, and common to all applications sharing the same programming model. The systems we are interested in are made of third-party components that are most often organized in a layered architecture: OS kernel, system libraries, compilers, virtual machines, middleware, etc. These layers introduce different abstraction levels that each provide different sets of elements from which applications can be built to run on top of these levels. As a consequence, different meta-models corresponding to different abstraction levels of the same system can be defined. For instance, the meta-model of an object-oriented application considered at the language level would typically contain entities and events such as "`Class`", "`Method`", "`Instanciation`", "`Invocation`", or "`Attribute`", but would probably not contain anything about OS-level issues such as memory paging, or task scheduling.

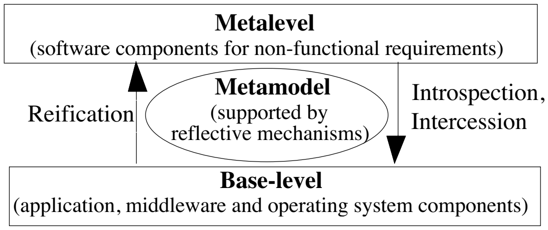

Metalevel
(software components for non-functional requirements)
Metamodel
(supported by reflective mechanisms)
Reification
Introspection, Intercession
Base-level
(application, middleware and operating system components)

Figure 1. Organization of a Reflective System

Meta-models provide an abstract view of the base-level system that enables the implementation of non-functional mechanisms at the meta-level (notion of separation of concerns). The information contained in the meta-model determines the range of non-functional mechanisms that can be implement at the meta-level. In our case, an ideal meta-model should provide all the reflective features that are required to implement correctly and efficiently fault-tolerance. To this aim, we propose the following steps when designing a fault-tolerance oriented meta-model:

1. *Establish the set of reflective features required by fault-tolerance*
2. *Map the requirements of Step 1 onto the different layers of the considered system*

In the next section, we investigate from a reflective perspective the requirements of a set of fault-tolerance mechanisms (Step 1). In section 4, we address Step 2.

3. A reflective View of Replication Strategies

Defining the complete meta-model that allows the implementation of all known fault-tolerance strategies is very ambitious. For illustration purposes, we limit our analysis to well known replication mechanisms namely passive replication (e.g. primary-backup strategy), semi-active replication (or leader-follower strategy) and active replication (e.g. TMR strategy)[1]. This section discusses the set of reflective features that are required to implement these strategies. We only address the *requirements* (i.e. the "*What is needed*?") of the different replication strategies from a *logical* viewpoint and express them in reflective terms. From a conceptual viewpoint, this exercise is very interesting as it collects the assumptions (e.g. interception of client requests, identification of non-deterministic decisions, state access, etc.) fault-tolerance designers have in mind when they propose a given algorithm. These assumptions usually become implicit, as the designer dives into the details of the algorithm, fault assumptions or performance aspects. Those *conceptual and implementation assumptions* are, however, key aspects to decide on the practicality of the proposed algorithms, and have often a crucial impact on the implementation. If the information is easy to obtain, then fine, if not, then the proposed algorithm cannot be implemented or can only be implemented with some restrictions, which often make the resulting implementation questionable. In this section, we try to collect all (most of) the *conceptual and implementation assumptions* made by the designers of the three replication strategies we investigate, and we factorize them into a meta-model.

3.1. System Model

We assume a conventional client / server model where servers process client requests and return the results of this processing. Servers encapsulate data (their state) and code (describing the services they offer to clients). When a service request is received, an "execution point" appears within the server. This execution point travels through the code, processes the received request, possibly modifies the server's state, and possibly produces a reply that is returned to the client. In this section, we don't make any assumption about the nature of servers, but we assume that server replicas are "distributed" so that they fail independently. Our notion of server is very similar to those of "replication entities, whatever they are" or "distributed processes" commonly found in works on distributed algorithms.

3.2. Considered Replication Strategies

We consider three replication mechanisms (passive, semi-active, and active replication techniques) according to the criteria identified above. We focus here on implementation requirements rather than on fault assumptions. To simplify the analysis, we also assume that requests are delivered to server replicas using an atomic multicast protocol. Table 1 summarizes the key well-known characteristics of the three replication strategies.

[1] Please see [14] for more details about these strategies.

Passive replication: the checkpointed information must ensure that the backup execution after a primary crash remains consistent with the previous execution as perceived by the rest of the system before the crash.

Semi-active replication requires that all non-deterministic decisions made by the leader are intercepted and forwarded to the followers.

Active replication (i.e. TMR) can only be considered for deterministic servers. If this holds, non-deterministic decisions do not need to transit between the replicas.

3.3. Control and Observability Requirements

For each considered replication strategy, we distinguish three control and observation facets: *communication*, *execution*, and *state*. At each level, we consider the *entities* that are concerned by the replication strategy, the *actions* of these entities that must be observed and controlled, the *motivation* for this, and finally the available *means* to satisfy these requirements. We do not consider the cloning of new replicas, as cloning involves operations (request synchronization, state transfer) that are very similar to those found in passive replication. The result of our analysis is presented in Table 2.

Our analysis is limited here to the requirements of the three considered replication strategies *provided* the assumptions of Table 1 are guaranteed. For instance, semi-active replication requires a mechanism that ensures "determinism" across replicas (e.g. notification messages). However, mechanisms have been proposed to enforce replica determinism with no communication between the replicas, and could be used for active replication [2, 7, 12].

3.4. The resulting Meta-Model

The essential reflective features given in Table 3 result from the aggregation of the requirements presented in Table 2. The corresponding meta-model results from the interactions between the base-level (application), and the meta-level (fault-tolerance). These interactions (see Figure 1) are classified as follows:

1. **Reification**: initiated by the base level to provide information to the meta-level.
2. **Introspection**: initiated by the meta-level to obtain information from the base-level.
3. **Behavioral intercession**: initiated by the meta-level to modify the behavior of the base-level.

Strategy	*Fault assumptions*	*Tolerated faults*	*Replica Determinism*	*Resource overhead*	*Communication overhead*	*Recovery overhead*
Passive	*Fail-silent servers*	*Crash faults*	*Not required*	*1 active server*	*High (checkpoints)*	*Medium (re-execute)*
Semi-active	*Fail-silent servers*	*Crash faults*	*Not required*	*2 active servers*	*Low (no checkpoints)*	*Low (switch)*
Active	*Fail-uncontrolled*	*Value faults*	*Required*	*3 active servers*	*Low (no checkpoints)*	*Low (null)*

Table 1. Assumptions and Key Characteristics of Well-Known Replication Strategies

Passive replication	*Entities*	*Action*	*Motivation*	*Means*
Communication	*requests / replies*	*send / receive*	*Synchronization between replicas.*	*Interception*
Execution	*execution points*	*activation / progress / termination*	*Capture/ restore on-going requests in concurrent servers.*	*Interception* *Platform instrumentation*
State	*internal data, platform data*	*change on internal data, interactions with the local platform*	*Consistent state restoration, i.e. transparent recovery from the client point of view.*	*Memory dump* *Serialization* *Interactions journals*

Semi-active replication	*Entities*	*Actions*	*Motivation*	*Means*
Communication	**idem as passive**	**idem as passive**	*control over leader / follower notifications*	**idem as passive**
Execution	**idem as passive** + *non-deterministic decision points*	**idem as passive** + *non-deterministic operations*	*control over non-deterministic decisions*	**idem as passive**
State	**idem as passive**	**idem as passive**	*control over platform interactions with non-deterministic results*	**idem as passive**

Active replication	*Entities*	*Actions*	*Motivation*	*Means*
Communication	**idem as passive**		*reply validation & propagation*	**idem as passive**
Execution	Not needed			
State	Not needed (cloning not considered)			

Table 2. Control and Observability Requirements for the considered Replications Strategies

Reflective features	Communication	Execution	State
Reification	*RequestReception* *RequestSending* *ReplySending* *ReplyReception*	*ExecutionPointStart* *ExecutionPointEnd* *ExecutionPointReach* *NonDeterministicFlowChange*	*NonDeterministicPlatformCall*
Introspection	*getRequestContent* *getReplyContent*	*getExecutionPoint*	*getServerState* *getPlatformState*
Behavioral Intercession	*doSend* *doReceive*	*createExecutionPoint* *setExecutionPoint* *forceResultOfFlowChange*	*ForceResultOfPlatformCall*
Structural Intercession	*piggyBackDataOnMsg*		*setServerState* *setPlatformSate*

Table 3. Towards an Aggregate Meta-Model for Replication Strategies

4. **Structural intercession**: initiated by the meta-level to modify the state of the base-level.

Table 3 does not contain all the possible features one may encounter in generic reflective systems [5, 10, 18]. However, although limited, these reflective features supports a meta-model for the replication strategies discussed in § **3.3**. Interestingly, according to the motivations of each reflective feature found in Table 3, the proposed facets (*Communication, Execution,* and *State*) can be related to two different concerns of replication strategies. The communication facet enables the coordination of the different replicas and is the least intrusive. It can be implemented using wrapping techniques for instance. The execution and state facets relate to the control of consistency across replicas. Those facets are the most intrusive, as they deal with internal non-determinism and state information. The key question is now: *How can this meta-model be implemented on a real platform?*

4. Introducing Multi-Layer Reflection

As previously mentioned in Section **2**, a real platform encompasses several abstraction levels that correspond to the different components of the concrete system. Several reflective architectures have been proposed for fault-tolerant systems [1, 6, 13], and have proved the interest of reflection in this context. All these reflective architectures, however, use reflective capabilities from a single abstraction level. Using a small example, we show in this section that the meta-model obtained in Section **3** cannot be implemented at a single level without threatening the interest of reflection itself. This example illustrates the motivation for MLR (Multi-Layer Reflection concepts previously introduced in [17]).

Consider the semi-active replication of a concurrent server implemented on top of an Object Request Broker (ORB), used in thread pool mode. Most ORB implementations offer such a concurrency model by spawning a fixed number of threads at initialization, and putting them in a "waiting state". When a request arrives, the ORB forwards the request to one of the threads of the pool, and this thread starts processing the request. The initial size of the thread pool (say p) determines the highest number of active concurrent requests. How a particular thread is assigned a particular request is ORB-implementation dependent,and remains totally hidden (i.e. non-deterministic) to the application[2]. In the same way, the ORB doesn't necessarily follow the order in which it receives requests from lower communication layers; requests may be delivered to the application objects in any order, even if they are received at lower layers through an atomic multicast protocol. In summary, the role of the ORB is two-fold: (i) dispatching of at most p requests among received requests to the application, and (ii) allocation of the selected requests to available pool-threads (at-most p). In our example, we further assume that the application itself is deterministic, i.e. that the results returned by the different requests only depend on the order in which requests are processed by application objects.

Consider now three fault-tolerance programmers who must add a replication mechanism to this kind of server.

- The *first* one has no access to any meta-information regarding the executive layers (black-box case).
- The *second* can control all OS-level thread related operations (scheduling, synchronization, etc.) through a dedicated OS-level meta-model (mono-level reflection).
- *The third one* can both inspect and control the OS and the ORB through a multi-layer meta-model.

4.1. The Black-Box Case

This first programmer has only access to the application level, all underlying executive layers being black-boxes. This approach gives him no control whatsoever on the order in which requests are delivered to the application objects. If a thread-pool-ORB with a pool size of two

[2] The CORBA standard does not recommend any specific multi-threaded object implementation.

threads simultaneously receives three requests, only two out of the three requests will be non-deterministically delivered to the application (the third one being queued). Consider three requests R, S, and T delivered in this order to the replicated ORBs. Assume the ORB of the leader dispatches R and S to the two threads of its pool, and the leader chooses to serve R first and then S. If the ORB of the follower dispatches S and T to the application instead, the follower will be unable to follow the leader's choice. Although visible at the application level, this decision is internal to the ORB and so cannot be controlled in this case. This problem (called PB1) relates to ORB internal messages shuffling, which destroys any total order provided by an underlying atomic multicast protocol.

A possible solution to this problem is to serialize incoming requests at the communication level before they are delivered to the ORB (but it also eliminates the benefits of the thread-pool mode [12]).

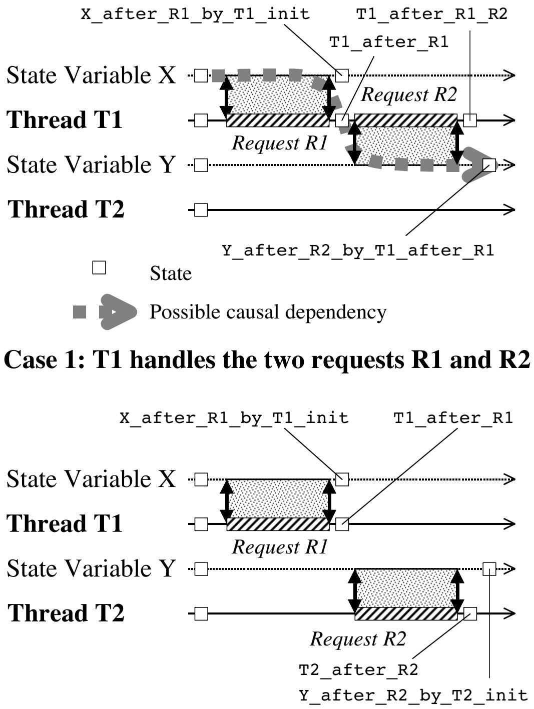
4.2. The Mono-Layer Reflection Case

Our second fault-tolerance programmer has access to the OS level only, the ORB remaining a back-box. In this case, low level communication primitives, thread scheduling and synchronization can be controlled. Forcing all replicated OS to schedule threads and to allocate mutexes in exactly the same way, ensures that requests are processed in the same order by all object replicas. This approach inhibits ORB message shuffling and solves problem PB1 under our assumptions (deterministic application). This is however quite complex in a multi-layer architecture and not optimal. We call this "non-optimality" problem PB2.

The reason is that, forcing threads to process requests in exactly the same order at the OS level enables object replicas to reach identical states, but introduces useless constraints. Indeed, different activation profiles can reach the same state, even when threads are not run exactly in the same order. An example is given in Figure 2.

Consider two different dispatchings of two successive requests (R1 and R2) on a pool containing two threads (T1 and T2). Request R1 interacts with the application state variable X (potentially shared), whereas request R2 interacts with the state variable Y. Request processing is represented by dashed bars and interactions by dotted areas limited by double arrows.

In Case 1, both requests are handled by T1. In Case 2, request R1 is processed by T1, request R2 is processed by T2. Clearly, having full visibility of the thread behavior leads to understand that the final states after both computation profiles are identical. However, as OS level instrumentation restricts the visibility (and thus semantic understanding) to threads and mutex actions alone, the fault-tolerance programmer cannot easily reach this conclusion. From his point of view, as only one thread T1 is used, the result of processing R1 may impact the processing of R2. In other words, as a potential causal dependency exists between the state of Y after R2 (`Y_after_R2_by_T1_after_R1`), and the state of X before R1 (`X_init`), the application may reach two different states (X,Y) depending on the dispatching decision. Figure 3-a traces the FT-programmer view of the computation as perceived at the OS level, and shows that two possible major states are perceived after processing R2 (as shown in Figure 3-b).

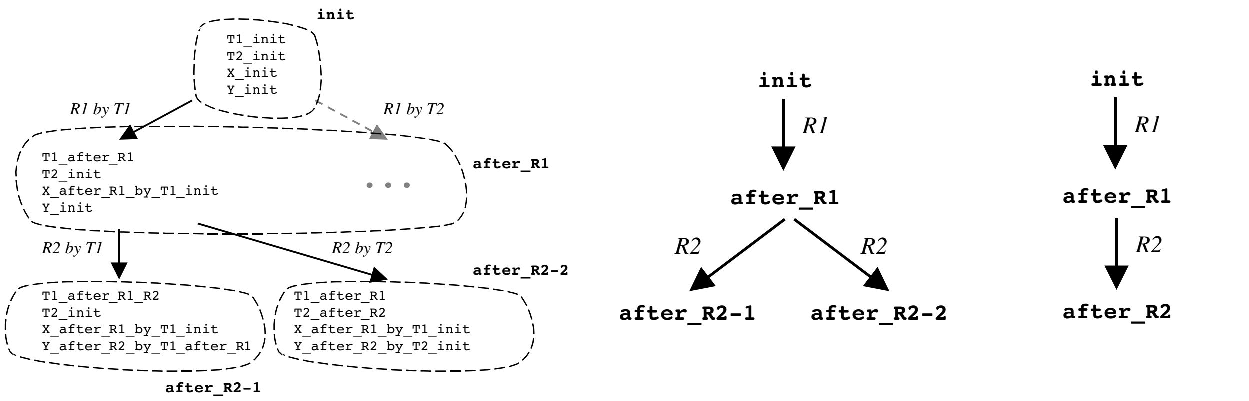


Case 1: T1 handles the two requests R1 and R2

Case 2: T1 handles R1, T2 handles R2

Figure 2: Request vs Thread using a Thread Pool

In practice, one of these two computation profiles will be imposed to both leader and follower. So, the non-determinism problem PB1 is solved at the expense of blind forcing of thread scheduling at both replicas (as in [7]). However, in a complex multi-layer architecture controlling all individual OS actions induces unacceptable overheads as middleware layers intensively use threading and mutex locks. The non-optimality of this solution is due to the lack of visibility and semantics of the computation in the ORB.



a- Detailed states (OS view) **b- Major of the states (OS view)** **c- States (MLR view)**

Figure 3: Different Views of the Computation

4.3. The Multi-Layer Reflection Case

Consider now the third fault-tolerance programmer, who can control both the OS and the middleware, running in thread pool mode in our example. From the particular semantics of a thread-pool, he knows that the thread states `T2_init`, and `T1_after_R1` are equivalent, as pool threads do not keep memory of previously processed requests. In other words, threads in the pool always process requests from a pre-defined initial state. No information regarding the processing of R1 can propagate to R2 through thread T1. The potential causal dependency shown on Figure 2 does not exist[3]. So, taking into account the semantics of concurrency models at the middleware level allows him to discard the "request-to-thread allocation" as a source of non-determinism. There is no need for the ORB running the leader replica to force its follower to allocate requests to exactly the same threads. The distinction made in Figure 3-b between the states `after_R2-1` and `after_R2-2` is useless. `after_R2-1` and `after_R2-2` are grouped into the single state `after_R2`, as shown in Figure 3-c.

Our third programmer can thus avoid problem PB2. In addition, having access to the internal decision of the ORB, i.e. delivery and retrieval to/from the pool, solves problem PB1 in a more elegant and efficient manner. In §5, we describe how this can be done in practice.

This small example illustrates how combining information obtained from several levels can help discarding sources of non-determinism as non-relevant for handling replication of multi-threaded objects.

The complementary nature of high and low level reflection and lessons learnt from reflective systems development [15], prompted us to introduce the notion of multi-layer reflection and its attached terminology [17]. In brief, this notion focuses on the interdependencies between individual system layers to provide an end-to-end meta-model that is explicitly tailored for fault-tolerance. Notions of mapping and projection support the analysis of interlevel coupling from a reflective perspective. A mapping describes the various possible representations of a given entity at a given abstraction level i by entities available at a (lower) abstraction level i-1. A projection is the transitive closure of mapping relations that maps a top-level entity to lower level entities (useful for state handling). Reverse projections map low-level entities to higher level ones (useful for error confinement).

5. A Multi-Layer Reflection: Case Study

In this section, we present on a concrete architecture, how the MLR solution (cf. §4.3) can be implemented in practice, and propose for the chosen case study an explicit meta-model that corresponds to the requirements of Table 3. From the reverse engineering of a simple application running on an ORB, we discuss step-by-step the two facets of the consistency problem of replication strategies: the control of non-determinism and the state transfer.

5.1. Case-Study Description

We consider a system composed of a POSIX-compliant OS, a CORBA-compliant middleware, and a simple application that implements the following IDL interface:

```
interface Hello {
  unsigned long say_hello();
};
```

[3] In our example the R1 and R2 do no share state variables, and the resulting state does not depend on their interleaving. However, the causal dependency through shared variables could be handled by enforcing access to shared variables X and Y in the same order, using mutex-control approaches as in [2].

On receiving a request "`say_hello()`", the application increments an internal counter (originally set to 0), and returns the new value to the client. A possible C++ implementation of this application can be as follows.

```
CORBA::ULong Hello_impl::say_hello() {
  CORBA::ULong result ;
  pthread_mutex_lock(&_object_lock);
    _count++ ;
    result = _count ;
    cout << "Hello World!: "
         << _count << endl;
  pthread_mutex_unlock(&_object_lock);
  return result ;
}
```

The counter `_count` represents the application's internal state. As this counter is returned to the client, the order in which requests are scheduled (indirectly through the mutex `Hello_impl::_object_lock`) determines which client sees which result.

In order to replicate this very simple application, we need to identify the reflective features of Table 3: control over execution points and determinism for active and semi-active replication, and the state transfer for passive replication and cloning. To reach this goal, Figure 4 shows the reverse engineering of a concrete CORBA implementation running our example. This figure shows simplified traces of the different active threads within the Orbacus CORBA implementation when processing the `say_hello()` request in pool mode. Orbacus (version 4.1.1) was used in `thread_pool` mode with four threads in its pool (*p* = 4), on Linux (version 2.4.18), leading in this case to 8 active threads (4 additional service threads!).

On the figure, four threads are shown, with numbers 1, 3, 4 and 8. 1 is the main thread, 3 the thread that accepts socket connections (i.e. it executes the `accept` system call). Thread 4 is one of the pool threads (the other pool threads correspond to the numbers 5, 6, 7– not shown). Thread 8 is the receiver thread associated to the invoking client. The thread number 2, not shown, corresponds to the manager thread of the current Linux pthread implementation. This manager thread is totally hidden to the user of the pthread library, and is used internally to carry out all thread management actions (blocking, signaling, suspension, creation, and destruction). This manager thread is an example of implementation choices that remain totally invisible to higher system levels implemented on top of it.

In this figure, we can distinguish four main phases:

1. First, the ORB is initialized (calls numbered from (0) to (5)). The thread pool is created (calls number (2) and (3)) and the accepting thread 3 is spawn.
2. In the second phase, a connection request is received from a remote client, and a receiver thread is launched (call number (6)): several connections, several receiver threads. Connection management realized within the ORB is transparent to the application level.
3. In a third phase, the request is received by the receiver thread (call (8)), and travels up to the application code (call (14) to `say_hello()`). The transfer of the request from Thread 8 (receiver) to Thread 4 (thread pool member) occurs through the a shared request queue (Thread 8 invokes `ThreadPool::add(..)`, which awakes Thread 4 and have it return from `ThreadPool::get(..)`.).
4. Thread 4 returns from the application and calls a sequence of object methods (15 to 18) to return the result of the request execution (call (19)) to the client.

5.2. Request Execution Related Meta-Model

Definition of the Meta-Model

In order to handle the non-determinism and control the execution, we focus on the part of the meta-model of Table 3 related to request execution. We model here the lifecycle of a request as follows: (i) a request is received by the ORB, (ii) delivered to the application and finally (iii) results are sent back to the client. This lifecycle could be refined, but other aspects have not been identified as relevant to the fault-tolerance mechanisms discussed in Section 3. Based on this lifecycle, we must be able to observe the following classes of reified events (see Figure 4) for a detailed control of request execution through the ORB:

```
BeginOfRequestReception
EndOfRequestReception
RequestBeforeApplication
RequestAfterApplication
BeginOfRequestResultSend
EndOfRequestResultSend
RequestContentionPoint
```

The processing of a request reifies exactly one instance of each of these event classes, except for the last one: `RequestContentionPoints` correspond to the several decision points, in the ORB and the application, that determine the ordering of request processing.

The Meta-Model applied to the Example

From the reverse engineering[4] analysis in Figure 4, one can easily identify the first six "Request related events" mentioned previously. `BeginOfRequestReception` is mapped to the call to `recv` (number (8) in the figure); `EndOfRequestReception` to the return of the same call. `RequestBeforeApplication` is mapped to the call to `say_hello()`; `RequestAfterApplication` to the return of the same call; `BeginOfRequestResultSend`

[4] A special reverse engineering tool was developed on purpose to obtain this graph by analyzing the runtime execution of an open-source ORB, here Orbacus.

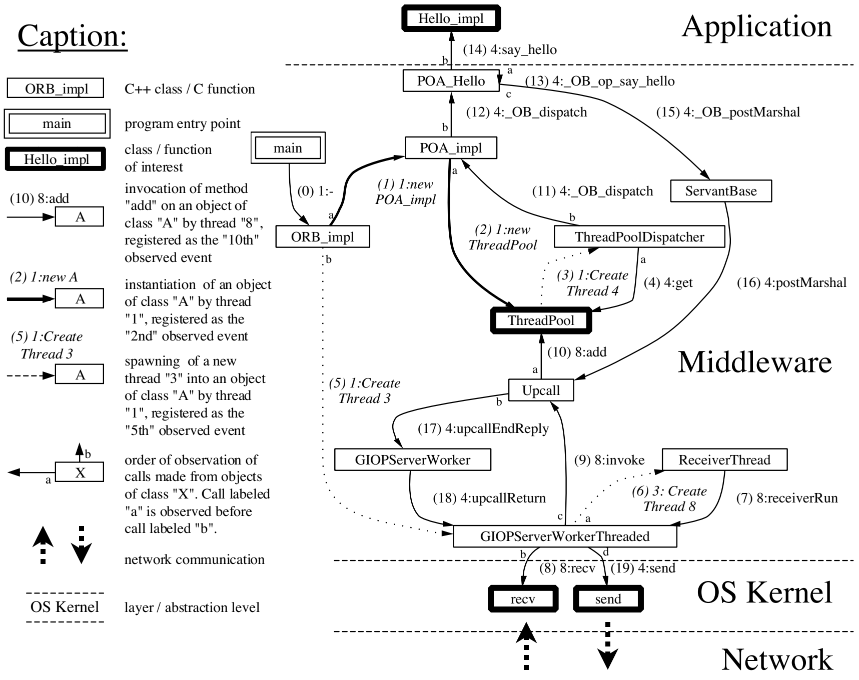


Figure 4: Request Dispatching using a Thread Pool in Orbacus

and `EndOfRequestResultSend` are mapped to the call and the return of `send` respectively (number (19).

From the analysis of the control and data path followed by a request, we identify two places corresponding to `RequestContentionPoints`, namely RCP1 and RPC2:

- **RCP1**: A lock protects the object `ThreadPool` and serializes accesses to the request queue by both receiver-threads and pool-threads in `ThreadPool::add()` and `ThreadPool::get()`.
- **RCP2**: A second contention point in the application code, which uses the `this->_object_lock` mutex.

Quite interestingly, controlling those two contention points appears to be necessary and sufficient to control the non-determinism introduced by the request dispatching within the ORB. According to the above meta-information, non-deterministic decisions can be identified and propagated to a replica.

- RCP1 controls the order in which the pool-threads extract requests from the request queue within the ORB,
- RCP2 enables controlling in which order requests are processed within the application objects.

This is a nice example of MLR as RCP1 belongs to the ORB and RCP2 to the application. Both contention points are needed to control non-determinism and solve PB1 and PB2. RCP1 only cannot ensure the order in which requests are processed by the application object. RCP2 only cannot ensure that the same set of requests is extracted from the queue. Both have an impact on the results returned the clients (i.e. the value of the counter `_count`).

5.3. State Related Meta-Model

Passive replication requires a state-related meta-model to be able to capture and restore a consistent server state (platform and application). This can be done in two steps: first restoring the ORB/OS state, then the application state in a consistent way. The request life cycle presented previously is used to restore the ORB state.

Indeed, once we control how requests and their associated threads interleave within the ORB, we can restore an ORB state by simply "re-injecting" requests at the communication interface. This can be seen as an adaptation of log-based checkpointing techniques [16]. Actually, the aim is not to restore the entire ORB state but the relevant state driving the execution of our application. This is the key benefit of our approach. This implies subtle re-execution of the ORB parts related to the processing of on-going requests thanks to meta-models information. The objective of this re-execution is to reach a state that is equivalent (not necessarily identical) for the application.

From the point of view of the ORB, requests can be pending in the ORB for execution, in progress at the application level (limited by the pool size) or pending in the ORB for termination. The following ORB level intercession actions are thus required:

- (i) Insertion of pending requests for execution,
- (ii) Insertion of requests in progress in the pool,
- (iii) Insertion of pending requests for termination.

In order to perform these operations in a portable and disciplined manner, intercession facilities are required (see ORB meta-interface in Figure 5). All requests are inserted using the `InjectRequestAtCommLevel` ORB meta-interface intercession function. As we control `RequestContentionPoints` reified events, we can block the execution of pending requests for execution (i). For requests pending for termination (iii), we bypass the execution of the request by forcing the continuation of the execution within the ORB by means of the ORB meta-interface function `SkipCallToApplication`. To trigger this bypass operation, we intercept the execution flow before it reaches the application, thanks to the reification of event `RequestBeforeApplication` (cf. Section 5.2).

The management of requests in progress at the application level (ii) implies complementary state recovery actions at both the OS and the application layers:

- Update thread related data structures (OS);
- Update application state variables (Application);
- Resume execution of application objects (OS)

Similar meta-interface and reification facilities are thus required at the OS level. Numerous techniques can be used to restore thread execution stacks, for instance [8] that is portable, or [3, 4] for platform specific solutions (respectively on Java and Linux/i386).

The update of application state variables (here `_count`) can be done in many ways, including reflective approaches (already proposed as in [9]) that are very portable.

5.4. Towards Implementation of the Meta-Models

Figure 5 summarizes in a Java-like format the meta-interface we have obtained. This meta-interface is quite generic, and can be applied on any ORB that follows the assumptions we made on this example. Its design synthesizes the requirement analysis of the fault-tolerance mechanisms, and the Multi-layer analysis we proposed. Implementation using Orbacus and Linux is in progress today.

6. Conclusion and Future Work

The rapid evolution of system platforms and the variability of their configurations and surrounding environments induces an increasing need for adaptive systems at both functional and non functional levels. In the particular case of fault-tolerance, many attractive algorithms have been proposed to generically provide fault-tolerance to a wide range of system classes. These algorithms are based, however, on the availability of specific observation and control features in the underlying system platform that are often difficult to obtain in practice. The difficulty arises from the multi-layer structure of the considered platforms, and the increasing use of off-the-shelf software components. These control and observability features are, however, essential to the correctness of these algorithms and cannot be ignored.

```
class Request ;
class Thread ;
class StackChunk ;
class ReifiedEvent ;
class RequestLifeCycleEvent extends ReifiedEvent {
       public Request reifiedRequest ;
       public Thread  reifyingThread ;
}
class BeginOfRequestReception    extends RequestLifeCycleEvent ;
class EndOfRequestReception      extends RequestLifeCycleEvent ;
class RequestBeforeApplication   extends RequestLifeCycleEvent ;
class RequestAfterApplication    extends RequestLifeCycleEvent ;
class BeginOfRequestResultSend   extends RequestLifeCycleEvent ;
class EndOfRequestResultSend     extends RequestLifeCycleEvent ;
class RequestContentionPoints    extends RequestLifeCycleEvent ;

class IntercessionCommand ;
class ContinueExecution          extends IntercessionCommand ;
class SkipCallToApplication      extends IntercessionCommand ;

interface MetaLevel {
       IntercessionCommand reifyEventToMetaSynchronous(ReifiedEvent e);
}
interface BaseLevel {
       State captureApplicationState ();
       void  restoreApplicationState (State s);
       StackChunk captureApplicationStack (Thread t);
       void       restoreApplicationStack (Thread t, StackChunk stack) ;
       void  InjectRequestAtCommuncationLevel(Request r);
}
```

Figure 5: The resulting meta-interface

In this paper we proposed an approach that provides early answers to the above problems by using Reflection to make platform assumptions easier to address. After analyzing several conventional replication techniques, we factorized the requirements of these techniques into a high-level meta-model. We illustrated on a case study how this meta-model can be mapped to a given system platform by analyzing the structural and behavioral elements across several abstraction levels. Multi-Layer Reflection aggregates the meta-model of several abstraction levels in order to select the most appropriate location to obtain relevant meta-information.

In many works, people often tend to ignore implementation details raising the argument of standard interfaces. As far as fault-tolerant computing is concerned, this kind of argument does not hold and we showed how Multi-Layer Reflection could overcome this problem. However, implementing MLR, although possible on open-source executive layers, is very complex. The reason is that, even when complying with standard interfaces, the implementation of middleware and operating systems is difficult to master for the development of fault-tolerance strategies. The implementation of cross-cutting concerns definitely calls for appropriate implementation frameworks, and to some extent, a new generation of components, namely reflective components. This is one of our long term objectives for a better mastering and adaptation of fault-tolerance in complex component-based systems.

Acknowledgements: This work was partially supported by the European IST Project DSoS “Dependable Systems of Systems” n°1999-11585. We also would like to thank our colleagues Gordon Blair at Lancaster University (UK) and Jean-Bernard Stefani at INRIA (F) that helped us in bootstrapping these ideas. Anonymous referees are also deeply acknowledged.

7. References

[1] G. Agha, S. Frølund, R. Panwar, and D. Sturman, “A Linguistic Framework for Dynamic Composition of Dependability Protocols,” presented at *the IFIP Conference on Dependable Computing for Critical Applications (DCCA-3)*, Palermo (Sicily), Italy, 1992.

[2] C. Basile, Z. Kalbarczyk, and R. Iyer, “A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas,” presented at *The International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA, 2003.

[3] S. Bouchenak, “Making Java Applications Mobile or Persistent,” presented at *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, Texas, USA, 2001.

[4] W. R. Dieter and J. E. Lumpp Jr., “User-level Checkpointing for LinuxThreads Programs,” presented at *2001 USENIX Technical Conference*, Boston, Massachusetts, USA, 2001.

[5] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, “THINK: A Software Framework for Component-based Operating System Kernels,” presented at *Usenix Annual Technical Conference*, Monterey (USA), 2002.

[6] B. Garbinato, R. Guerraoui, and K. R. Mazouni, “Implementation of the GARF Replicated Objects Platform”, *Distributed Systems Engineering Journal*, vol. 2, pp. 14-27, 1995.

[7] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo, “Deterministic Scheduling for Transactional Multithreaded Replicas,” presented at *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Nürmberg, Germany, 2000.

[8] F. Karablieh and R. A. Bazzi, “Heterogeneous Checkpointing for Multithreaded Applications,” presented at *21st Symposium on Reliable Distributed Systems (SRDS'02)*, Osaka, Japan, 2002.

[9] M.-O. Killijian, Ruiz-Garcia Juan-Carlos, and J.-C. Fabre, “Portable serialization of CORBA objects: a reflective approach,” presented at *18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*, Seattle, Washington, USA, 2002.

[10] F. Kon, F. Costa, G. Blair, and R. H. Campbell, “The case for reflective middleware”, *Communications of the ACM*, vol. 45, pp. 33-38, 2002.

[11] P. Maes, “Concepts and Experiments in Computational Reflection,” presented at *Conference on Object-Oriented* Programming *Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, 1987.

[12] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, “Enforcing Determinism for the consistent replication of Multithreaded CORBA Applications,” presented at *18th Symposium on Reliable Distributed Systems (SRDS)*, Lausanne, Switzerland, 1999.

[13] T. Pérennou and J.-C. Fabre, “A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach”, *IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems*, vol. 47, pp. 78-95, 1998.

[14] D. Powell, “Delta-4: A Generic Architecture for Dependable Distributed Computing,” : Springer-Verlag, 1991, pp. 484.

[15] J.-C. Ruiz-García, M.-O. Killijian, J.-C. Fabre, and P. Thévenod-Fosse, “Reflective Fault-Tolerant Systems: From Experience to Challenges”, *IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems*, 2003.

[16] R. E. Strom and S. Yemini, “Optimistic Recovery in Distributed Systems”, *ACM Transactions on Computer Systems*, vol. 3, pp. 204-226, 1985.

[17] F. Taïani, J.-C. Fabre, and M.-O. Killijian, “Principles of Multi-Level Reflection for Fault-Tolerant Architectures,” presented at *2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, Tsukuba (Japan), 2002.

[18] Y. Yokote, F. Teraoka, and M. Tokoro, “A Reflective Architecture for an Object-Oriented Distributed Operating System.,” presented at *Third European Conference on Object-Oriented Programming (ECOOP'89)*, Nottingham, UK, 1989.