Principles of Multi-Level Reflection for Fault Tolerant Architectures

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

Principles of Multi-Level Reflection for Fault Tolerant Architectures

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[1]

This paper presents the principles of multi-level reflection as an enabling technology for the design and implementation of adaptive fault tolerant systems. By exhibiting the structural and behavioral aspects of a software component, the reflection paradigm enables the design and implementation of appropriate non-functional mechanisms at a meta-level. The separation of concerns provided by reflective architectures makes reflection a perfect match for fault tolerance mechanisms. However, in order to provide the necessary and sufficient information for error detection and recovery, reflection must be applied to all system layers in an orthogonal manner. This is the main motivation behind the notion of multi-level reflection that is introduced in this paper. We describe the basic concepts of this new architectural paradigm, and illustrate them with concrete examples. We also discuss some practical work that has recently been carried out to start implementing the proposed framework.

1. Introduction

Reflection has established itself as a very powerful concept to realize separation of concerns in computing systems. First introduced in functional languages [1], reflection was successfully applied to computing architectures as a structuring paradigm [2]. Since then, this paradigm has given rise to a very attractive research field for the handling of non-functional requirements of computer systems. This encompasses distribution, mobility, tracing, debugging, security, fault tolerance, etc. Reflection was shown relevant to a wide range of objects: processes, middleware, kernels, protocols but also compilers, and virtual machines, and is now considered a major step towards the disciplined management of system evolution.

Today, the increased deployment of reusable software components even in systems with high fault-tolerance requirements[2] raises new challenges, and reflection appears as one of the most promising technologies to tackle them. The design and the validation of modern systems must now take into account the possible use of *off-the-shelf* software components, while still ensuring the overall system's dependability, and respecting industry's traditional constraints in terms of *time-to-market*, *adaptation* and *evolution*. Because of the separation of concerns it provides, and its genericity, reflection seems particularly suited to help solve these problems.

So far however, most of the work related to reflection in dependable computing — including security — has addressed a single type of component: application entities (a process or a task, an actor, an object, etc.), an executive layer (a kernel, a middleware, a virtual machine) or a tool (e.g. a compiler). Our prior work has convinced us that the use of reflection on only one component is insufficient to insure the dependability of a complex system. Indeed, fault tolerance spans all the layers of a system, and requires the ability to observe and control objects in the large (network localization, transaction state, system configuration, etc.) and in the small (state information, context switches, memory mappings, etc.).

We advocate in this paper the notion of multi-level reflection as a consistent concept for the implementation of fault tolerance in multi-layered systems, in particular based on COTS. The core idea of multi-level reflection is to provide a meta-model based on a consistent view of individual meta-models that can be exhibited at each abstraction level. Fault tolerance meta-level software can then be defined and implemented according to a holistic understanding of the recursive representation of the system entities and their various interaction levels.

The paper describes in detail the principles of this approach and illustrates the notion with some concrete examples. We also address implementation issues. The paper is organized as follows. Section 2 briefly recalls the state of the art regarding reflection applied to fault tolerant systems, and discusses the limits identified by previous work. Section 3 introduces the key concepts of multi-level reflection such as meta-models, meta-interfaces, mapping, projection, and meta-filters. Section 4 describes a

[1] This work has been partially supported by the European IST Project "Dependable Systems of Systems" n°1999-11585.

[2] See for instance [Stolper 1999] for a report on the use of COTS in the Mars Pathfinder NASA spacecraft.

conceptual framework of multi-level reflection based systems. In Section 5 we present some implementation issues, in particular the establishment of meta-models from reverse engineering open source middleware. Section 6 concludes the paper.

2. Reflective Fault Tolerant Systems

2.1. What is Computational Reflection?

Reflection is the ability of a computing system to observe and modify its own structure and behavior as part of its own computation [2]. A reflective system is basically structured around a representation of itself — its *self-representation* or *meta-model* — that is causally connected to the real system. Any change in the system meta-model is reflected on the system's behavior, and conversely any evolution of the real system is reflected in its meta-model. Reflection introduces two distinct parts in a computing system: a base level where normal computation of the system takes place, and a meta-level where the system does the computation using its meta-model (*meta-computation* or *meta-level software*), this computation being related to some non-functional mechanisms. The meta-model can thus be regarded as a kind of "glue" that insures the connection between the base- (normal computation) and the meta- ("self" computation) levels. From an implementation viewpoint, the meta-model is usually established through some specific interfaces, called *meta-interfaces* of the system, that provide practical means to observe and control the interaction between the base and the meta-level (Fig. 1).

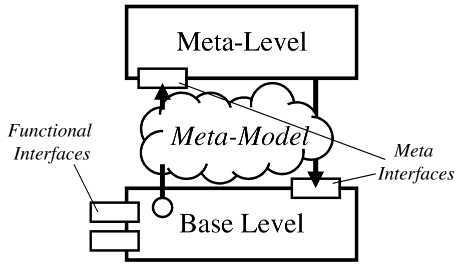

Fig. 1: Architecture of a Reflective System

Classically, meta-models and their corresponding meta-interfaces are defined in terms of computational entities that are specific to the base level, i.e. a particular language or a programming model, these entities being common to all systems that use the same programming framework. For instance, the meta-model of a component-based system developed using an object-oriented language could rely on notions such as "`Attribute`", "`Method`", "`Class`", "`MethodCall`", "`InheritanceRelationship`", etc. Indeed, as entities appearing in a meta-model are rather generic, the meta-computation that takes place at the meta-level can be specified in very generic terms, independently of the actual function performed by the base level. This feature is a corner stone of the separation of concerns provided by reflective systems and is of very high interest for various non-functional mechanisms, e.g. fault-tolerance strategies, as illustrated in the next section.

In practice, the meta-model is established first through *reification* mechanisms that expose structural and behavioral aspects (data structures and events) of the base level. The meta-model can be updated on the fly by the meta-level using *introspection* facilities that enable base-level information to be obtained on-demand by the meta-level. The meta-level software interprets this (meta-) information and triggers actions, including some affecting the base-level entities through *intercession* facilities.

2.2. Reflective Fault-Tolerant Systems

Several dependable system architectures based on reflection have been proposed during the last decade. Notable efforts have particularly been made to integrate reflection with object-oriented principles and distributed infrastructures. Platforms such as GARF [3], MAUD [4], and FRIENDS [5] are representative of this trend. A similar approach is used in all these systems. They are all based on reflective capabilities of some specific languages. Interacting application objects populate the base level. The meta-level software is responsible for the handling of fault tolerance strategies (or security strategies or both). These three projects differ mainly in the language-based reflective capabilities they use, and consequently in their meta-model, i.e. the detailed view of the base level provided to the meta-level. GARF is based on the exception handling of the Smalltalk language, and exhibits object interactions to the meta-level. MAUD is based on the HAL language and renames message destinations to redirect actor interactions to other objects. FRIENDS uses an open C++ compiler to intercept object interactions and to access the attributes of base-level objects. In each of these examples, the meta-computation is transparent to the application programmer, and implements some replication strategies using additional services, such as group communication.

Reflection has also been applied to the hardening of lower executive layers, such as real-time micro-kernels [6]. The base-level computation is in this case a real-time kernel. In [6], a very detailed meta-model is defined through temporal logic specifications of generic operating system primitives (scheduling, synchronization, memory management, timers, etc.). The formal expressions are then compiled (translated to C) to generate, among others, error confinement wrappers. The meta-computation occurs within several runtime wrappers controlling the correct execution of an executive layer. To perform the verification at runtime, the internal and external behavior of the kernel are partially exposed to the meta-level (called here *meta-kernel*). In practice, any deviation of the real behavior (as it is perceived by the meta-level) from the specification is detected. Accordingly, the meta-level software can halt the base-level computation and trigger error detection signals (error confinement) and possibly start some corrective actions (error recovery) to enhance the original kernel's dependability.

2.3. Lessons Learnt and Limitations

The reflective architectures we have just mentioned share a common feature. They all use reflective capabilities at 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. Of course, these approaches are not tied to any languages or OS, and may be ported to any other environment that provides the same abstractions. Nevertheless, the fact that they are mono-level make these approaches blind to any information or behavior that is not contained within the target abstraction level. Because the range of dependability mechanisms that can be implemented using a reflective approach depends greatly on the meta-model available to the meta-level, this limitation strongly hinders the deployment of extensive fault-tolerance strategies.

For instance, the fault tolerance of multi-threaded servers is very difficult to achieve using only high level abstractions, such as language-level reflection. Indeed, neither context switches, nor critical sections, or lock allocation (semaphores, mutexes) are visible within usual languages [7]. These are needed to replicate multi-threaded processes. More generally, high level reflective platforms miss information about low level hidden states, events and implementation policies.

Implementing replication at a lower layer with low level abstractions do not solve the problem either. Indeed, if we consider for instance the checkpointing problem, approaches that rely on binary core dumps of processes cannot work without some minimal knowledge of the semantics of the captured state. Raw core dumps contain data that only have a meaning on the computer system running the application (e.g. file descriptors). When a new incarnation of a checkpointed process is launched, corrective measures must be taken on the captured process-state to update those platform-dependent data. This includes for example process and file identifiers [8] but also kernel- and library-tables that contain the executive entities belonging to the checkpointed process. The performance of fault-tolerance mechanisms may further profit greatly from application semantics, which is not available at the executive layer.

To remove the limitations inherent to any mono-level approach, we propose a multi-level reflective architecture, which we describe in the following section. The basic idea is to use reflection at each layer of the system architecture to expose the meta-information which is necessary and sufficient to implement a given fault tolerance strategy, and to expose this in a consistent and unified way.

3. Multi-Level Reflection: Basic Concepts

As explained in the previous sections, although reflection is a very attractive technology for fault-tolerance mechanisms due to the separation of concerns it provides, this concept should ideally be applied to all layers of a system architecture. In this section, we first present a simple example that introduces the basic concepts of multi-level reflection (§3.1). These concepts are then further discussed and detailed in paragraph 3.2.

3.1. Example: Synchronizing Threads

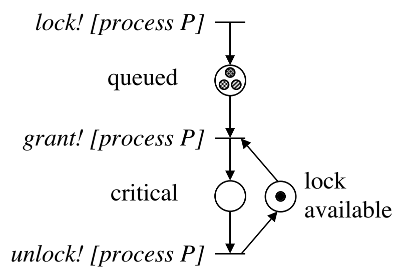

Fig. 2: A Simple Mutex Model

Figure 2 presents a simple model of synchronization by mutex in a multi tasking operating system. We used colored Petri-Nets, a well-known state / transition formalism. Let's consider a reflective operating system. We assume that, at runtime, kernel-calls and internal kernel events corresponding to the transitions of the model presented on Figure 2 are intercepted and dispatched to the meta-level. Using this reified information the meta-level animates a meta-model, and may modify the default behavior of the operating system through intercession facilities.

```
mutex m ;
task T1 is          task T2 is
   lock(m) ;           lock(m) ;
   doWork1() ;         doWork2() ;
   unlock(m) ;         unlock(m) ;
end T1 ;            end T2 ;
```

Fig. 3: Two Competing Threads

Consider two tasks t_1 and t_2 that respectively execute the code T1 and T2 shown on Figure 3, on top of such a reflective operating system.

At runtime, the kernel-calls *lock! [T1]* and *lock! [T2]* are reified to the meta-level. At this point the meta-level can decide to call the original lock code of the kernel, depending on the current state of the computation, reflected in the meta-model. If it does so for each thread, the internal kernel event *grant!* $[T_i]$ is intercepted some time later and redirected to the meta-level, with i=1 or 2 depending of the kernel scheduler. Here again, the meta-level can call the real grant call to trigger the lock allocation to the selected thread.

Because the meta-level controls when and how the actual mutex code is activated, it can modify the scheduling policy of the kernel. For instance, it may delay the call to the original lock function for some set of threads until some given condition is met.

The model on Figure 2 exhibits low-level activities such as context switches, and lock allocation. Combined with reflective capabilities obtained from higher levels, for instance a state-capture algorithm based on language-level reflection [9], this meta-model allows the implementation of fault-tolerance mechanisms such as the active replication of multi-threaded servers [10, 11], in a very efficient, scalable, and flexible way. Synchronizing replicas requires information both from the application level (object-attributes, high-level information about the application configuration and remote connections), and from the kernel level (thread synchronization, site-dependent variables, tables entries, etc.).

More generally, the explicit identification of the reflective capabilities that are required by a given family of fault-tolerance mechanisms happens to be crucial to the adaptation of a software component. Indeed, adding fault-tolerance to a component almost always requires some form of intrusion, and that at different levels of the system. The meta-information required by a family of fault-tolerance mechanisms explicitly formalizes the nature and the degree of that intrusion. It can be used as a specification for the instrumentation of the considered component. It allows the definition of the appropriate meta-model, and the selection the most appropriate levels for its implementation. One crucial issue in finding this meta-model regards the conceptual means required to encompass the multi-level nature of such systems, as discussed in the following sections.

3.2. Basic Concepts

We've seen in paragraphs §2.3, and §3.1 why fault-tolerant systems can benefit from the coordinated use of meta-models implemented at different abstraction levels. This suggests the construction of a single multi-level model that aggregates those heterogeneous meta-models. However, such a construction requires a precise understanding of the interactions between a system's layers. In this section we try to get more insight in this question.

In component-based systems, a component often recursively relies on the availability of other components to work correctly. This recursive "reuse chain" generally materializes itself in the final system in a layered architecture delimited by standardized interfaces such as CORBA [12] for the middleware, POSIX [13] for the operating system, or the Intel IA32 architecture for the CPU. Each standard offers a programming model that provides concepts, and primitive operations, constrained by a set of rules.

For instance, on Figure 4, layer L_{n+1} implements the Interface I_{n+1} while using the programming model of interface I_n, which is provided by layer L_n.

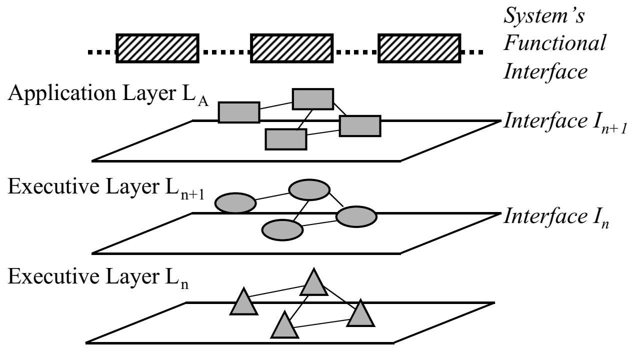

Fig. 4: Programming Models, Layers, and Interfaces

Each layer L_i contains a set of implementation entities (the rectangle, ellipses and triangles on Figure 4). Those entities are built out of the programming primitives offered by the interface(s) I_{i-1} underneath L_i, and give life to the new programming model PM_i that is implemented by L_i. However, they are not directly visible from outside the layer L_i. The interface I_i of L_i acts as a "paradigm firewall" that shields the user from the actual layer implementation, and protects the system from uncontrolled dependencies. (This is a well-known application of the general *Information Hiding* principle [14].)

Based on those considerations, we have identified the following five basic concepts to construct a multi-level meta-model of a layered system:

Meta-models: model the abstractions governing the structure and the behavior of a part of the system. The respective meta-models of individual layers should be combined into an integrated meta-model of the whole system, orthogonal to all levels.

Meta-filters: specify how to obtain a more specialized partial view of a meta-model. This partial view contains the necessary and sufficient meta-information required by a given fault-tolerance mechanism.

Mappings: describe the various possible representations of a given entity at abstraction level *i* by entities available at abstraction level *i-1*. Reverse mappings link entities at abstraction level *i-1* with the entities they implement at level *i*.

Projections: transitive closure of mapping relations that map a top-level entity to lower level entities (useful for state handling). Conversely reverse projections help trace top level entities related to a given low level entity (useful for error confinement).

Meta-interfaces: provide meta-information on a given subsystem. A meta-interface may be associated with a specific individual component, with a layer, or with the whole system.

These concepts are further discussed in the next section.

4. Multi-level Reflective Framework

In the multi-layer reflective architecture introduced in the previous section, there are in fact several meta-models, which can be classified along two orthogonal axis. The first axis relates to the scope of the considered meta-model (Does it encompass the whole system or only one level?), and the second one to its specialization or genericity (Is that a general-purpose meta-model, or an optimized, narrowly targeted one?).

This 2-dimensional conceptual space is represented in Figure 5. Traditional reflective approaches can be located on the X-axis, since they are mono-level. The multi-level meta-models we advocate are located in the upper region of the space.

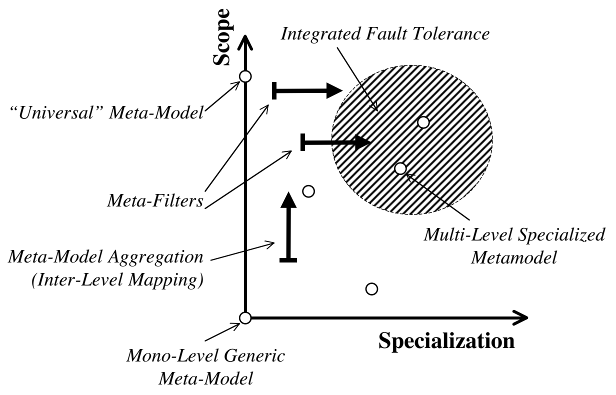
Scope
Integrated Fault Tolerance
"Universal" Meta-Model
Meta-Filters
Meta-Model Aggregation (Inter-Level Mapping)
Multi-Level Specialized Metamodel
Specialization
Mono-Level Generic Meta-Model

Fig. 5: A Multi-Level Meta-Model Space

Our motivation for introducing a "Specialization" dimension is that in most concrete cases the implementation of an all-encompassing meta-model is not needed for fault-tolerance. Too much genericity may even be counter-productive for the systems we're interested in. The implementation cost of meta-interfaces into non-instrumented COTS, and Open Source Software, depends directly on the needed degree of intrusion. Moreover, a too fine-grained runtime control can unnecessarily burden the overall system performance and render an ill-designed reflective architecture untractable.

As discussed previously, individual mono-level meta-models can be mapped onto one another according to the different level implementations. These inter-level *mappings* cement together the individual meta-models and permit their incremental aggregation (represented as vertical arrows) into a generic, all encompassing, *universal* meta-model.

The two goals we set in the introduction — integration of all system levels, and specialization to fault-tolerance — define a region on Figure 5, which is denoted with a shaded area. The corresponding specific meta-models can be obtained by filtering unneeded reflective capabilities from the universal meta-model we've just mentioned. This filtering occurs by means of *meta-filters* (represented as horizontal arrows) specific to the targeted crosscutting mechanisms.

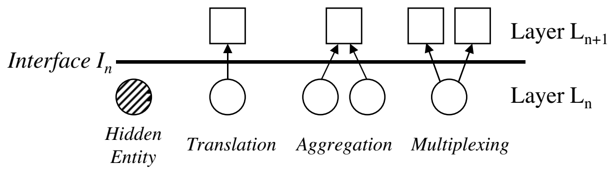

Fig. 6: A Basic Mapping Taxonomy

Now, what are the mappings between levels actually made of? A mapping taxonomy should at least be able to identify relationships such as translation (1:1), aggregation (n:1), and multiplexing (1:n) between the relevant entities of the considered system (Figure 6). These relationships may be refined, depending on the needs, into relationships with richer semantics such as creation, referencing, state dependency, etc. For instance, a semaphore is usually implemented as the aggregation of a counter and a process queue. A CORBA request in a multi-threaded server is often mapped to a composite entity made of a thread associated with a connection-oriented socket. Several CORBA objects can be mapped onto either a single or several UNIX processes, which makes a big difference regarding error propagation and confinement areas.

Another important part of the mapping description regards all implementation entities that do *not* explicitly appear in higher levels. These entities account for the hidden state (§2.3) and the non-deterministic behavior of a layer. For example, in the Linux implementation of the POSIX Thread Standard, a task known as the *thread manager* is responsible for the thread bookkeeping and inter-thread communication. This task is totally transparent to the application programmer.

The aggregation of several mono-level meta-models through mappings, and the subsequent filtering according to the targeted reflection capabilities yields a specialized multi-level meta-model. This meta-model is made accessible at runtime to the meta-level through a set of meta-interfaces, one for each level (Figure 7). These meta-interfaces are extended with aggregation and navigation capabilities, to help navigate through the different levels of the system, and offer a homogenous meta-programming model to the meta-level programmer.

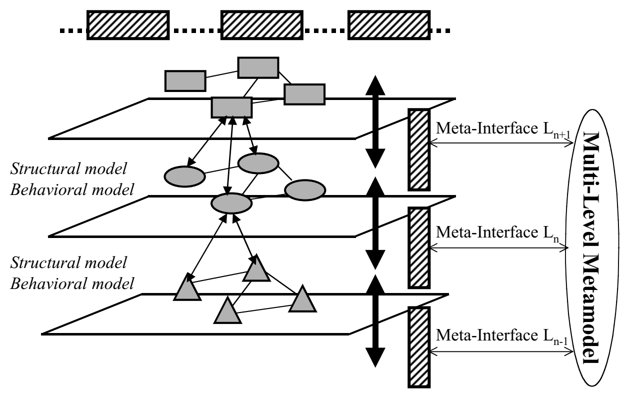

Fig. 7: A Multi-Level Reflective Architecture

At runtime, the meta-interfaces reify the structure and behavior of each layer to the meta-level (see Figure 1 on page 3) and enable both introspection and intercession facilities to observe and control the entities and interactions in a given layer. For each entity *E* of any layer, it's then possible recursively to track through several layers the set S_E of lower entities that are "related" to this particular higher instance (Figure 8). We call this set S_E the *top-down projection* of *E* with respect to the considered type of relationship/mapping.

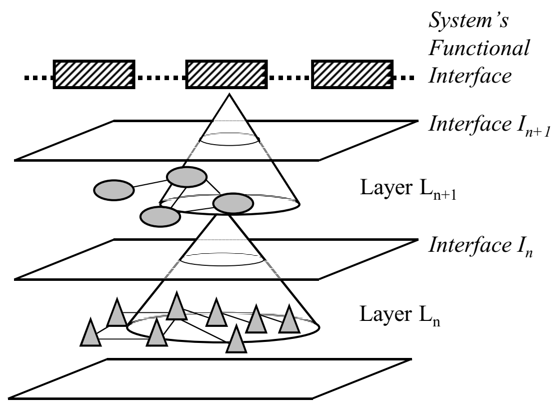

Fig. 8: Top-Down Projection

What "related" means depends on the kind of mapping relationships we want to consider. For instance, in order to partially checkpoint the system as proposed in [15], we'll track inter-level state dependencies. Such state-dependencies projection can also be very interesting to finely track causality between the nodes of a middleware based distributed system and implement some major families of checkpointing protocols [16].

Bottom-up or *reverse projections* are possible as well. Figure 9 illustrates how the impact of a fault in a lower layer could be tracked up the system level hierarchy.

This precise tracking would allow, for example, the generalization of the reflective wrapping technology proposed for micro-kernels in [6], and provide fined grained error containment possibly within the same machine.

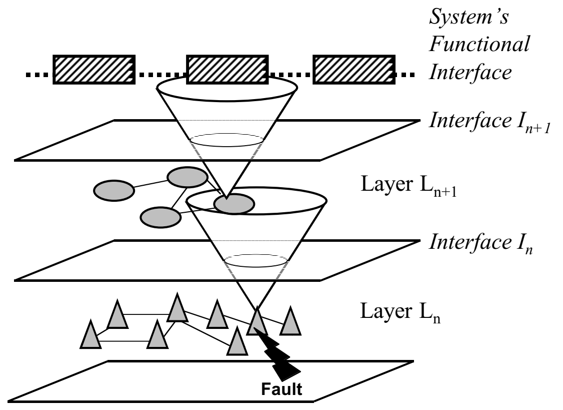

Fig. 9: Error Detection across the Layers of a System

5. Implementation Issues

The development of a multi-level reflective architecture founded on the principles described previously first requires a thorough understanding of existing system architectures. In particular, we're interested in the underlying meta-models of common system layers. A simple approach we considered in practice consisted in reverse engineering broadly available components (in particular open source components).

For instance, we obtained Figure 10 by intercepting syscalls on the kernel network API, during the processing of a remote request by a multi-threaded CORBA server. The graph represents the stack traces of the server threads for each syscall invocation, observed within a debugger. It was obtained under Linux (version 2.4.18), with the commercial *ORBacus*[3] CORBA implementation (version 4.1.1, in `thread_per_client` mode), with gdb 5.1-1. The graph only gives a partial view of the complex actions a remote invocation triggers within the middleware, and reached this form after various intermediate classes were abstracted away. The figure shows the different system layers (Network, OS, Middleware, Application), and their interactions.

Based on further similar observations, it is possible to build behavioral and structural models for each layer. Figure 11 for instance synthesize the observations of Figure 10 in a Petri-Net model that stresses the thread management in *ORBacus*. This figure identifies an accepting thread (thread ID 3 on the figure) that spawns a new thread on CORBA request reception (on the figure a new thread with ID 4 is spawned).

[3] *ORBacus*® is a registered trademark of IONA Technologies PLC.

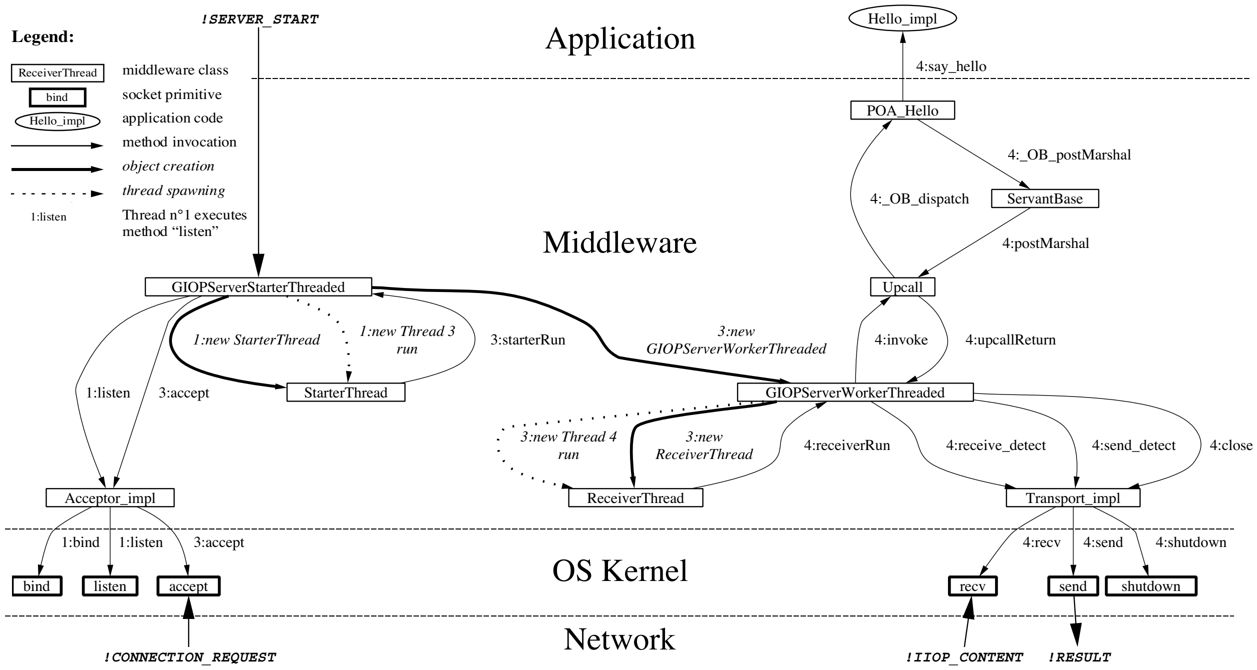

Fig. 10: Activation of the socket syscalls during the processing of a CORBA request in *ORBacus*

Interestingly, the model of Figure 11, which focuses on the thread management in the middleware layer, can be related to the programming model of POSIX sockets (executive layer). Figure 12 presents such a model, expressed as a State-Chart, in which transitions refer to the invocation of socket primitives. The "listening" socket (state-chart on the left) directly maps onto thread 3 on Figures 11 and 10. The communication sockets (state-chart on the right) map onto the threads that are launched each time a CORBA request is processed (thread 4 on Figures 11 and 10).

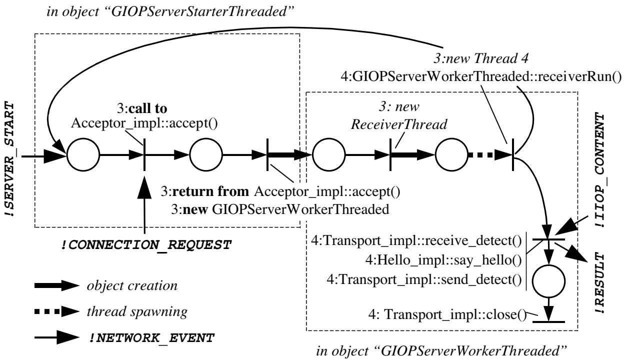
in object "GIOPServerStarterThreaded"
3:new Thread 4
4:GIOPServerWorkerThreaded::receiverRun()
3:call to Acceptor_impl::accept()
3: new ReceiverThread
!SERVER_START
!IIOP_CONTENT
3:return from Acceptor_impl::accept()
3:new GIOPServerWorkerThreaded
!CONNECTION_REQUEST
4:Transport_impl::receive_detect()
4:Hello_impl::say_hello()
4:Transport_impl::send_detect()
4: Transport_impl::close()
!RESULT
object creation
thread spawning
!NETWORK_EVENT
in object "GIOPServerWorkerThreaded"

Fig. 11: CORBA Request Processing and Thread Spawning in *ORBacus*

More generally, once a behavioral or structural model has been obtained for a given component as in Figure 11, it is then possible to relate this information to lower layer models, thus identifying the inter-level mappings we introduced previously.

The mappings that result from this analysis may not be the same for all possible implementations of the component interface. Several implementations must be analyzed in order to extract a few generic architectural patterns for the considered abstraction level, and help identify, at least partially, the generic universal meta-model introduced earlier.

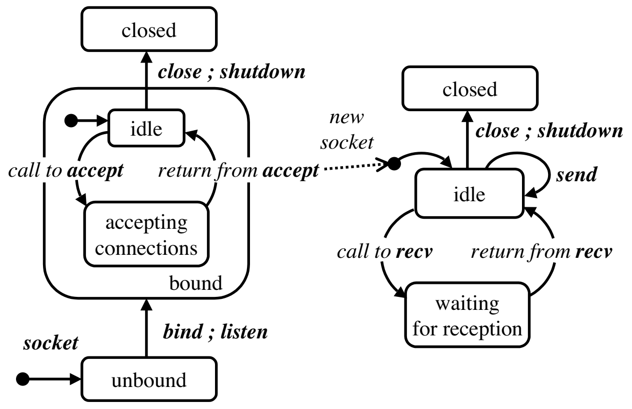

Fig. 12: State Machine Model of Berkeley Sockets

As far as ORB implementation are concerned, we looked at *omniORB* and *ORBacus*, and explored interesting differences in the design choices of both ORBs. For example, when run in equivalent thread-per-client modes, both ORBs don't realize the same garbage collection of idle connections. In *ORBacus*, the thread that processed the request is directly responsible for the closing of that connection after a given time-out. *omniORB*, on the other hand, delegates the time-out closing to some background thread, which remains invisible to the middleware programmer. Our approach allowed us to precisely identify the different points at which those policies are implemented, and to examine in both cases which internal ORB objects are involved.

Those analyses, and the models we constructed from them, only build the first steps towards the universal meta-model we advocated previously. More work is required in that direction. Once such a generic meta-model is built, a concrete meta-model, targeting specific fault-tolerance capabilities, can be obtained through meta-filtering. We have not yet reached that objective, but we expect some formalism for fault-tolerance requirements to be needed for this task.

6. Conclusion

Reflection is today a well-known paradigm that has been successfully used to address non-functional concepts in system architectures. In particular, security and fault tolerance have benefited from this concept as demonstrated by several projects and prototypes worldwide. Our previous research in the field was a contribution to the development of fault- and intrusion-tolerant systems using reflective languages. The main problem we identified was the limited meta-information available at a given level regarding the above or underlying layers of the system. This was the main motivation for the introduction of multi-level reflection.

The basic concepts identified at this stage enable meta-level software to be based on a clear understanding of the entities-relations at all levels in a computer system and of their links through several software layers. We also advocate in this approach specialized meta-models that can be defined for targeting a given non-functional requirement. Several notions such as mapping and projections provide means to draw error confinement areas, and identify state information through all system layers for error recovery.

From a practical viewpoint, we have focused on establishing meta-models of existing components. In a first step we have analyzed off-the-shelf CORBA open-source middleware with reverse engineering techniques. It is worth noting that this is ongoing work, but we believe that these concepts and framework are of high interest to master the adaptation and evolution of future fault tolerant architectures. The material provided in this paper opens up a very large field of investigations and is the basis for our future work.

7. References

[1] Smith, B.C. *Reflection and Semantics in Lisp*. in *Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL)*. 1984. Salt Lake City, Utah: ACM. p. 23-35.

[2] Maes, P. *Concepts and Experiments in Computational Reflection*. in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1987. Orlando, Florida. p. 147-155.

[3] Garbinato, B., R. Guerraoui, and K.R. Mazouni, *Implementation of the GARF Replicated Objects Platform*. Distributed Systems Engineering Journal, 1995. **2**(1): p. 14-27.

[4] Agha, G., *et al*. *A Linguistic Framework for Dynamic Composition of Dependability Protocols*. in *the IFIP Conference on Dependable Computing for Critical Applications (DCCA-3)*. 1992. Palermo (Sicily), Italy: Elsevier. p. 197-207.

[5] Pérennou, T. 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, 1998. **47**: p. 78-95.

[6] Rodriguez, M., J.-C. Fabre, and J. Arlat. *Formal Specification for Building Robust Real-time Microkernels*. in *21st IEEE Real-Time Systems Symposium*. 2000. Orlando, Florida, USA.

[7] Killijian, M.-O. and J.-C. Fabre. *Implementing a Reflective Fault-Tolerance CORBA System*. in *19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*. 2000. Nürnberg, Germany. p. 154-163.

[8] Dieter, W.R. and J.E. Lumpp Jr. *User-level Checkpointing for LinuxThreads Programs*. in *2001 USENIX Technical Conference*. 2001. Boston, Massachusetts, USA.

[9] Killijian, M.O., *et al*. *A metaobject protocol for fault-tolerant CORBA applications*. in *17th IEEE Symposium on Reliable Distributed Systems (SRDS-17)*. 1998. West Lafayette (USA). p. 127-134.

[10] Jiménez-Peris, R., M. Patiño-Martínez, and S. Arévalo. *Deterministic Scheduling for Transactional Multithreaded Replicas*. in *19th IEEE Symposium on Reliable Distributed Systems (SRDS)*. 2000. Nürmberg, Germany. p. 164-173.

[11] Narasimhan, P., L.E. Moser, and P.M. Melliar-Smith. *Enforcing Determinism for the consistent replication of Multithreaded CORBA Applications*. in *18th Symposium on Reliable Distributed Systems*. 1999. Lausanne, Switzerland: IEEE. p. 263-273.

[12] OMG, *Common Object Request Broker Architecture (CORBA/IIOP) (2.6)*. 2001. [http://www.omg.org/cgi-bin/doc?formal/01-12-35].

[13] ISO-IEC, *[IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. 1996. 784.

[14] Parnas, D.L., *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 1972. **15**: p. 1053-1058.

[15] Kasbekar, M., C. Narayanan, and C. Das, *Selective Checkpointing and Rollbacks in Multithreaded Object-Oriented Environment*. IEEE Transactions on Reliability, 1999. **48**(4): p. 325-337.

[16] Baldoni, R., J.-M. Hélary, and M. Raynal, *Rollback-Dependency Trackability: A Minimal Characterization and its Protocol*, . 1998, IRISA: Rennes (France).