Towards Proof-Based Real-Time Distribution Middleware

Jérôme Hugues    Laurent Pautet    Fabrice Kordon

Main author: Jérôme Hugues (PhD Student)
jerome.hugues@enst.fr, laurent.pautet@enst.fr
GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
fabrice.kordon@lip6.fr
Université Pierre & Marie Curie, Laboratoire d’Informatique de Paris 6/SRC
4, place Jussieu, F-75252 Paris CEDEX 05, France

Abstract

Distribution middleware is now the back-bone of many critical, real-time embedded applications. New developments strive for ensuring and preserving system properties as part of the engineering process. Thus, middleware must demonstrate its properties, such as dependability, reliability or real-time constraints. This paper discusses middleware architecture geared towards verification and validation of its properties. Our contribution presents the “schizophrenic” middleware architecture, and how we adapt and deploy this architecture to support the many requirements of critical systems, and to enable the verification of its properties.

Contents

1. Introduction: Issues in Middleware Engineering
2. Objectives
3. The Schizophrenic Middleware Architecture
4. Formal Verification of the µBroker
5. Validating middleware real-time properties
6. Conclusion

Keywords: Middleware, Verification, Validation, Petri Nets, Real-Time
1 Introduction: Issues in Middleware Engineering

Distribution middleware is now widely integrated as a building block in Distributed Real-Time Embedded (DRE) systems. In this context, the properties of each building block must be known to ensure its correct integration to the application [4] and to ensure the correctness of the system as a whole.

The IST ASSERT project\(^1\) defines scenarios and requirements for middleware, to be integrated in a full engineering process that ensures and preserves the properties of the system to be built. For instance, the European Space Agency identified several use cases for middleware: they encompass ground stations interacting with satellites as well as fleets of collaborating satellites and drones; they require multiple distribution mechanisms to handle variations in communication channels, flexible resource management, and to ensure autonomy for long missions.

In addition to distribution needs, these systems come with non-functional requirements, inherited from real-time engineering such as reliability, availability, dependability. Hence, properties (like determinism, safety, liveliness, timeliness) must be verified during the design process and in particular at the middleware level.

Middleware solutions now support the requirements of most distributed applications, including support for real-time or fault tolerance.

Yet, the verification of the behavioral properties of distributed application (e.g. request fairness, absence of deadlock or correct resource dimensioning) is usually the domain of verification-domain experts, using sophisticated techniques. This verification is usually limited to the semantics of the application.

On the other hand, the verification of middleware properties is usually done on a limited scale, restricted to very specific scenarios, or to the semantics of the distribution constructs used (e.g. RT CORBA).

Thus, middleware can barely act as a “Commercial Off-The-Shelf” (COTS) component. This calls for a next-generation of middleware that addresses many challenges [26]: middleware should be versatile to meet many application needs; it should also follow an extensive proof-based engineering approach to provide strong evidence it is correct with respect to application requirements; finally it should provide hints on how to validate its properties.

This paper details our current work on middleware architecture, and middleware support for verification and validation. We first review existing middleware architectures, and list requirements for the construction of a versatile middleware that supports both functional requirements, and a verification pro-

\(^1\)ASSERT is part of the Sixth Framework Programme of the IST
cess; then, we introduce the schizophrenic middleware architecture as a solution. We present how we verify key behavioral properties for some configurations of our middleware, refining our middleware architecture and modeling it using Petri nets. We summarize some of the algorithms we used to address state space explosion and apply them to check middleware behavioral properties. Finally we discuss the real-time capabilities of our architecture. This provides a first step towards the definition and implementation of proof-based middleware.

2 Objectives

In this section, we present current trends in middleware architectures, and their limits with respect to validation and verification. We then introduce requirements for proof-based middleware.

2.1 Middleware for DRE systems

Distribution encompasses both distribution models that provide the entities to build distributed applications, such as Message Oriented Middleware (MOM), Distributed Shared Memory (DSM), RPC or Distributed Objects (DOC); and also specific architectures for supporting them e.g. group communications, dedicated buses, scheduling techniques.

In this context, Rajkumar et al. [25] advertised Message Passing as a solution for DRE systems, and proposed the Real-Time Publisher/Subscriber service. The Ada Distributed Systems Annex [17, 23] supports the DOC, RPC and SM mechanisms. RT-CORBA extends CORBA’s DOC mechanisms and integrates support for many QoS policies [27]. OSEK/VDX [6] defines a target-specific middleware for automotive applications. Each of these approaches provide an acceptable solution for some classes of applications.

This leads to the definition of many tailorable middleware architectures:

- **Configurable middleware**, such as TAO [28], let applications select specific run-time policies to support the DOC distribution model, implementing the CORBA specifications. It relies on architectural and design patterns [13] to support a large number of policies.

- **Adaptive and reflective middleware** [26, 3] extends middleware configurability mechanisms to enable adaptability to specific changes in application context. This architecture provides promising properties to meet QoS applications requirements.
Generic middleware, such as Jonathan [31], defines abstract canonical components and architecture, their instantiation provides a specific distribution model. Jonathan provides a CORBA personality (David), a Java RMI personality (Jeremie) and specialized personalities for multimedia systems.

2.2 Issues in the verification and validation of distributed applications

Middleware platforms have shown in various project they can meet stringent requirements. They are now used in many mission-critical, including space applications, for instance using the microORB middleware developed by SciSys².

Building distribution platform for such systems is a complex task. One has to cope with the restrictions enforced to achieve high integrity standards, or to meet certification requirements, such as DO-178B. Thus, one has to be able to assert middleware properties, e.g. functional behavioral properties such as absence of deadlocks, request fairness, or correct resource dimensioning; but also temporal properties, to validate real-time properties.

Thus, middleware engineering should also provides provisions for some verification mechanisms as defined by the ISO committee [16] as “[the] confirmation by examination and provision of objective evidence that specified requirements have been fulfilled. Objective evidence is information which can be proved true, based on facts obtained through observation, measurement, test or other means.”

However, we note there is a double combinatorial explosion when considering middleware as a whole: the number of possible execution scenarios for one middleware configuration increases with the interleaving of threads and requests; the number of possible configurations increases with middleware adaptability and versatility. Finally, the behavior of a middleware highly depends on the configuration parameters selected by the user. Thus, verifying a middleware is a complex task.

Some projects consider testing some scenarios, on multiple target platforms. The Skoll Distributed Continuous Q&A project [22] relies on the concepts of SETI@Home to test TAO many configurations and scenarios on volunteers computers around the world. This provides some hints on the behavior of the middleware, but cannot serve as a definite proof of its properties.

The formal-based verification of distributed application behavioral properties is usually the domain of verification-domain experts, using specific verifi-

cation techniques, e.g. calculi, formal methods. However, such a verification process is usually used only to verify the semantics of the application (e.g. set of correct message sequences) [18]. This provides no information on the underlying distribution framework or middleware integrated to the system; and thus reduces the scope of the properties proved for the application under study.

One may instead contemplate the verification of middleware properties. Yet this is usually done on a limited scale, restricted to the very specific scenarios of the application to be delivered and the semantics of the distribution model used (e.g. RT CORBA), for instance using the Bogor model checker [7]. However, middleware implementations of the same specifications may behave differently [2]. Some properties may be withdrawn by implementation issues, such as the use of COTS, that are hidden by this modeling process, or by different interpretation of the same specifications. Besides, such verification process usually do not take into account implementation-defined configuration options, and target capabilities. Finally, such methods may be limited by combinatorial explosion that arise when building the system state-space.

Then, we claim that the verification process of a distributed application should also focus on the middleware as a building block, and thus middleware architecture should be made verification-ready so as to ease this process.

2.3 Towards proof-based real-time middleware

Modeling for verification or validation purposes, and middleware engineering are usually considered as two different expert domains. They are usually considered either by separate teams of the project or at separate steps in the design process. We propose to reconcile system modeling and middleware engineering. We list several requirements for proof-based real-time middleware:

1. clear separation of concerns to provide a clear description of the middleware architecture, and to enable the separate analysis of middleware components using formal techniques,

2. extreme tailorability to support application requirements for specific middleware services, protocols or hardware,

3. high code reuse, thanks to generic services, to increase the knowledge and confidence in code reused, and to enforce the value of middleware as a COTS.

Besides, one need to select specific validation and verification methods to extract middleware properties; these algorithms should be adapted so as to fight
combinatorial explosion and enable the analysis of complex configurations, consistent with the deployed system.

In the remainder of the paper, we present our current work on middleware architecture to support the requirements of proof-based real-time middleware. We first introduce the schizophrenic middleware architecture, and show how it enables verification. Then, we show how we model our design to test its properties, and finally select the most adequate middleware components to meet the requirements of distributed real-time applications.

3 The Schizophrenic Middleware Architecture

In this section, we introduce the key elements of the schizophrenic middleware architecture, and its role in the definition of proof-based real-time middleware.

3.1 Decoupling middleware components

Middleware combines two complementary facets: (1) a framework to implement distributed systems, using the host and operating system resources; and (2) a set of services to build portable distributed applications. In [14], we introduced the “schizophrenic” middleware architecture: a unique architecture that advertises these two aspects, and enforces separation of concerns.

In [32], we present PolyORB, our implementation of such a schizophrenic middleware. We assess its suitability as a COTS middleware for industry projects, that supports multiple specifications (CORBA, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun’s JMS).

From our experiments, we note that a reduced set of services is sufficient to describe various distribution models. We identify seven steps in the processing of a request, each of which is defined as one fundamental service. Services are generic components for which a general implementation is provided. Alternate implementation may be used to match more precise semantics. Each middleware instance is one coherent assembling of these entities. The µBroker component coordinates the services: it is responsible for the correct propagation of the request in the middleware instance.

Figure 1 illustrates how PolyORB services cooperate to transmit one request between two application entities, located on two separate nodes.

First, the client looks up server’s reference using the addressing service (1), a dictionary. Then, it uses the binding factory (2) to establish a connection with the server, using one communication channels (e.g. sockets, protocol stack).
Request parameters are mapped onto a representation suitable for transmission over network, using the *representation* service (3), this is a mathematical mapping that convert a data into a byte stream (e.g. CORBA CDR).

A *protocol* (4) is implemented for transmissions between the client and the server nodes, through the *transport* (5) service; it establishes a communication channel between the two nodes. Both can be reduced to *finite-state automata*. Then the request is sent through the network and unmarshalled by the server.

Upon the reception of a request, the middleware instance ensures that a concrete entity is available to execute the request, using the *activation* service (6). Finally, the *execution* service (7) assigns execution resources to process the request. These two services rely on the *factory* and *resource management* design patterns.

Hence, services in our middleware architecture are **pipes and filters**: they compute a value and pass it to another component. Our experiments with PolyORB showed all implementations follow the same semantics, they are only adapted to match precise specifications. They can be reduced to well-known abstractions.

The µBroker handles the coordination of these services: it allocates resources and ensures the propagation of data through middleware. Besides, it is the only component that controls the whole middleware: it manipulates critical resources such as tasks and I/Os or global locks. It holds middleware behavioral properties.

Hence, the schizophrenic middleware architecture provides a comprehensive description of middleware. This architecture separates a set of generic services dedicated to request processing from the µBroker. The latter is directly responsible for middleware behavior. Thus, we isolate the control loop of our system, present in all middleware instances; and its functional elements.
3.2 \( \mu \)Broker: core of the middleware architecture

We identified the \( \mu \)Broker as the control loop of our architecture. Several “strategies” have been defined to create and use middleware resources: [24] detail different request processing policies implemented in TAO; the CARISM project [19] allows for the dynamic reconfiguration of communication stacks. Hence, the \( \mu \)Broker must be adaptable enough to support most of them, and still provides a clear design to enable modeling and then verification.

We propose the following architecture for the \( \mu \)Broker (figure 2):

- the \( \mu \)Broker Core API handles the functional parts of the interactions with other middleware services; it provides an interface to configure the middleware instance and helper routines to execute specific functions such as managing I/O. This component interacts directly with the Binding, Transport, Execution and Addressing services;

- the \( \mu \)Broker Controller manages the state automaton associated to the \( \mu \)Broker. It grants access to middleware internals (tasks, I/O and queues) and schedules tasks to process requests. It is responsible for the behavioral part of the \( \mu \)Broker. Several policies refine its behavior: the Asynchronous Event Checking policy sets up the polling and read strategies to check events on I/O sources; the Request Scheduler sorts request to be processed (e.g. FIFO, EDF orders), the Dispatcher selects threads that execute requests.

![Figure 2: The two sides of the \( \mu \)Broker](image)

\( \mu \)Broker entities are defined by their interface and a common high-level behavioral contract, instances of these entities may refine this behavior to support different policies. This architecture has been implemented in PolyORB. It uses well-defined entities, and demonstrates its adaptability to support classical middleware mechanisms, or specifications such as RT-CORBA.

The \( \mu \)Broker offers a comprehensive description of the middleware control loop, and a step towards verification and validation of middleware properties.
4 Formal Verification of the \(\mu\)Broker

In this section, we discuss the formal techniques used to model the \(\mu\)Broker, and then verify some of its expected properties using model-checking.

4.1 Modeling one middleware configuration

We propose to use formal methods to model and then verify our system. We selected Well-formed coloured Petri nets [5] as an input language for model checking. They are high-level Petri nets, in which tokens are typed data holders. This allows for a concise and parametric definition of a system, while preserving its semantics. Using these methods, we can now model our architecture using Petri nets as a language for system modeling and verification (figure 3).

**Figure 3: Steps of the \(\mu\)Broker modeling**

*Step 1:* we build one Petri net for each middleware components variation. Petri net transitions represent atomic actions; Petri net places are either middleware states or resources. Common places between different modules define interactions between Petri nets modules, they act as channel places [29].

*Step 2:* for one configuration of the \(\mu\)Broker, some Petri net modules are selected to produce the complete model. Communications places (outlined in black) represent links to other \(\mu\)Broker functions or to middleware services.

*Step 3:* the selected modules are merged to produce a global model, it represents one middleware configuration. This model and one initial marking enable the verification of the middleware properties.

Then, middleware functions can be separately verified and then combined to form the complete Petri net model. Many models can be assembled from a common library of models. Thus, we can test for specific conditions (policies and settings).

The initial marking of the Petri Net defines available resources (e.g. threads, I/Os); or sets up internal counters. Its state space covers all possible interleaving of atomic actions; thus all possible execution orders are tested.
4.2 \(\mu\)Broker configurations and models

In this section, we review the key parameters that characterize the \(\mu\)Broker, and some of the properties one might expect from such a component.

The \(\mu\)Broker is defined by the set of policies and the resources it uses. These settings are common to a large class of applications. We consider one middleware instance, in server mode, that processes all incoming requests. We study two configurations of the \(\mu\)Broker: Mono-Tasking (one main environment task) and Multi-Tasking (multiple tasks, using the Leader/Followers policy described in [24]). The latter allows for parallel request processing.

We assume that middleware resources are pre-allocated: we consider a static pool of threads; a bounded number of I/O sources and one pre-allocated memory pool to store requests. This hypothesis is acceptable: it corresponds to typical engineering practices in the context of critical systems. Our implementations and the corresponding models are controlled by three parameters:

- \(S_{\text{max}}\): is the upper bound of I/O Sources listening for incoming data;
- \(T_{\text{max}}\): is the number of Threads available within the middleware;
- \(B_{\text{size}}\): is the size of the Buffer allocated to read data from I/O sources.

\(S_{\text{max}}\) and \(T_{\text{max}}\) define a workload profile for the middleware node, \(B_{\text{size}}\) defines constraints on the memory allocated by the \(\mu\)Broker to process requests. These parameters control middleware throughput and execution correctness.

We list three essential properties of our component. They represent basic key properties our component must verify to fulfill its role.

- \(P_1\), no deadlock: the system process all incoming requests;
- \(P_2\), consistency: there is no buffer overflow;
- \(P_3\), fairness: every event on a source is detected and processed.

\(P_1\), \(P_3\) are difficult to verify only through the execution of some test cases: one has to examine all possible execution orders. This may not be affordable or even possible due to threads and requests interleaving. Besides, the adequate dimensioning of static resources to ensure consistency (\(P_2\)) is a strong requirement for DRE systems, yet it is a hard problem for open systems such as middleware. Thus, we propose to verify them for some configuration of the \(\mu\)Broker: each property is expressed as a LTL formula, then verified by model-checker tools.

4.3 Achieving formal analysis

One known limit to the use of Petri Nets as model checker is the combinatorial explosion when exploring the system’s state space.
We tackle this issue using recent works carried out at the LIP6. By detecting the symmetries of a system [30], and exploiting the symmetries allowed by a property [1], we enable full LTL model-checking while efficiently fighting the well-known combinatorial state-space explosion problem. These methods allow the construction of symbolic state-space, a quotient state-space, where nodes are equivalence classes of states, and arcs equivalence classes of events. In most favorable cases, it is exponentially smaller than the concrete state-space, and thus more amenable to computations within reasonable delays.

Thus, these symmetry-based reductions enable the verification of our system, while plain state-space generation is unfeasible with classic techniques. We took advantage of these to verify our models in [15], and sum up the results we obtained.

Figure 4 summarizes the size of the state space for $B_{size} = 5$ and $S_{max} = 4$. Petri Net models have an average size, yet the state space is large, it denotes the complexity of the semantics of the system. The size of the state space increases exponentially with $T_{max}$ and $S_{max}$. Yet, we note that the symbolic reachability graph is smaller by several powers of ten, and that this ratio increases with $T_{max}$. Thus, even for big configuration, the quotient state space remain within affordable bounds that allow for complete verification of LTL properties.

![Figure 4: $\mu$Broker’s state space (typical approach), quotient reachability graph (LIP6’s approach) and gain for $S_{max} = 4$ and $B_{size} = 5$.](image)

From these different figures, we claim that the analysis of PolyORB could not have been performed without the use of these advanced model checking techniques.
techniques because of intrinsic memory limitations of typical workstations. This is depicted in figure 4. It clearly shows an exponential ratio between the concrete state space and the quotient reachability graph. As an illustration, even for common middleware configurations (7 threads and 4 I/O sources) the system presents over $10^{11}$ states, but we could compute and evaluate its properties on the quotient reachability graph.

Computations were completed within acceptable delays (less than 10 hours for the biggest models), on a 2.6 GHz Pentium-4 computer with 512MB of memory, without swap, running GNU/Linux. Performance will improve with the maturity of the tools used, allowing for an in-depth analysis.

We verified two configurations: we tested LTL formulae representing expected properties, we extracted scenarios that help dimensioning resources. Our model-checking tools overcome the complexity of the µBroker model. The verified properties provide strong evidence that our architecture is correct.

The tools we used have a significant role to achieve verification. Our models were created using CPN-AMI [20], a Petri net CASE environment that integrates structural analysis tools and model checkers. We then analyse data type symmetries and compute the reduced state space using the GreatSPN [8] tool-suite. LTL model-checking was done using the model-checking library SPOT [21], in conjunction with GreatSPN.

This study allowed us to formally verify our middleware design for some configurations that may now be deployed. PolyORB’s tailorability allows us to select the most adequate component that correspond to this design and enforce specific functional properties. This is discussed in the next section.

5 Validating middleware real-time properties

This section discusses PolyORB’s real-time properties. We first discuss PolyORB’s engineering guidelines, we then present one configuration geared towards real-time settings, and then present benchmarks.

5.1 Enforcing real-time engineering practices

At the implementation level, we enforce the use of selected algorithms, patterns and run-time ensure properties of each building block of the middleware, and thus makes PolyORB deterministic.

The Ravenscar profile [11] is a subset of the tasking and concurrency constructs of Ada 95. It has been designed so that the restricted form of tasking it allows can still be used in high-integrity programs and be certified. Ba-
sically, it explicitly forbids any dynamic behavior (dynamic priority change, task creation/destruction), and enforces scheduling techniques that allows for full schedulability analysis based on the Rate Monotonic Analysis (RMA).

As part of its configurability options, PolyORB implements one concurrency library compliant with the Ravenscar profile. This allows us to ensure the determinism of the tasking and concurrency elements we use.

In section 3, we classified middleware functions either as being involved in the control of middleware, such as the configurable µBroker component and concurrency patterns; or as middleware functional-only components (protocol, transport, activation, etc). We also reduced functional components to classical abstractions: pipes, filters, dictionaries, factories, finite-state automata.

PolyORB genericity enables the user to select the most adequate implementation of each services, using the previous assumptions as configuration and/or implementation guidelines. In this respect, middleware services can be described using well-known design patterns, that are completely defined and studied, for instance in [12]; they can be made real-time:

1. maximum size of data to be manipulated (e.g. number and size of request parameters, number of application objects); and bounds on the number of objects (e.g. threads, requests, I/O sources) to be used during middleware lifecycle. This helps to efficiently dimension resources and preallocate them;

2. memory allocation policies, using specific Storage Pools to handle any transient overload in resource usage;

3. a priori knowledge on application entities (operation name, servant and POA hierarchy, etc), to efficiently use static and dynamic perfect hash tables [9] and then enable $O(1)$ look up time in dictionaries.

Besides, the configurability capabilities of PolyORB, and more specifically of the µBroker enables the implementation of real-time thread scheduling disciplines, event checking or concurrency strategies, e.g. deriving from existing concurrency policies as presented in [24]. Thus, PolyORB tailorability enables a complete adaptation of the middleware to very specific needs, e.g. real-time requirements.

5.2 Preliminary measures

In this section we detail benchmarks on PolyORB performance and support for real-time distributed applications. We defined two tests to measure:
1. **request processing jitter**: we measure the time required to process one CORBA request. Each request “echoes” an unsigned long parameter. We execute 1,000 times the test and analyze the statistical sample.

2. **compliance to RT-CORBA semantics**: we test the correct propagation of request with respect to their priorities, and measure priority inversion. Three requests, representing respectively low, medium, high level alarms are sent, in random order. We check that request are propagated in correct order, high-level alarms being processed first.

All tests were done on Ultra-5 workstations, running Solaris 9, with 128MB of RAM, operating one Ultra-Sparc III CPU at 333Mhz. This platform is a good compromise between all-purpose development platform that is easy to operate, and real-time systems. Solaris is known to have limited latency and good time properties with respect to its driver and protocol stack. Thus, it provides a first information on PolyORB capabilities to handle real-time application requirements.

We configured PolyORB to use the RT-CORBA application personality, along with the IIOP protocol personality. We chose to measure three different configurations for each test: 1) one local test, client and servers are in the same process, multi-tasking is enabled; 2) distributed tests, using either multi-tasking, or not, for each the nodes. The results are summarized in figure 5.

![Figure 5: Dispersion of the processing time of a RPC around the mean value](image)

An exhaustive analysis of the execution trace using a system logger and display utility shows there is no priority inversion at the level of requests prop-
agation and processing on the server side, ensuring compliance with the semantics of RT-CORBA. Besides, dispersion analysis shows that most of samples are in an interval that is 100µs wide, representing less than 10% of the total duration. A few artifacts (less than 10% of measured values) denote non-determinism of the OS, linked to memory allocation, jitter of the TCP/IP stack. They are negligible at that stage of the analysis.

We can see that PolyORB holds interesting properties as real-time middleware. Even though we must undertake a more in-depth analysis to ensure there is no discrepancy in our implementation, these tests demonstrate how the schizophrenic middleware architecture can be adapted to build a real-time middleware.

5.3 **Reducing middleware jitter**

The previous analysis demonstrates that the dispersion of RPC duration fits into a narrow temporal window, leading to computable bounds of the WCET. We can go forward and reduce this interval using

1. *Real-time Kernel*, to reduce jitter in task activation, and to enforce locking policies to limit priority inversion.

   PolyORB is written in solely in Ada 95, this reduces porting issues on new targets to the availability of an Ada runtime, and bindings to a transport interface such as TCP/IP.

   In this respect, we contemplate porting PolyORB to ORK (Open Ravenscar Kernel) [10], a real-time kernel that supports the Ravenscar subset.

2. *Real-time protocol*, to reduce latency in request propagation and demultiplexing. Many options are available, ranging from protocol operating on top of dedicated hardware such as VME, CAN or 1553 bus, but also on COTS hardware. We consider using a real-time protocol over Ethernet, such as RT-EP, provided by ORK.

   These different experiments are part of our future work on real-time middleware architecture, and will take place in the context of the ASSERT project, first to devise a prototype of a real-time middleware, and then a full-featured middleware that will fit requirements from the space and avionics industry.
Conclusion

In this paper, we focused on middleware architectures and the specification and implementation of proof-based middleware.

First, we outlined trends in middleware engineering: architectures exist to support the functional and non-functional requirements of DRE systems, meeting the requirements of many mission-critical, real-time applications.

We discussed verification practices in distributed applications engineering. We noted that the verification and the determination of the middleware properties is seldom contemplated. However, middleware holds properties that may greatly impact the system, they must be fully determined.

Thus, we proposed to define a middleware architecture that strives for supporting both distributed functional needs; and enable verification. This would complete the analysis of one core building block of distributed applications.

We listed requirements for such a middleware architecture: 1) clear separation of concerns to reduce architecture obfuscation, 2) extreme tailorability to support functional requirements, 3) high code reuse to gain knowledge and confidence in middleware becoming a COTS per se.

We presented the schizophrenic middleware architecture, and showed it fits the previous requirements. The schizophrenic architecture emphasizes on the separation of concerns in middleware: a set of fundamental services covers middleware functional components, a middleware main loop - the $\mu$Broker-coordinates them. Services can be adapted to support specific requirements.

We detailed the steps to model and then verify some configurations derived from our middleware design using Petri nets. These configurations correspond to potential deployment scenarios for our middleware architecture.

Then, we discussed the verification of three properties of our system. The state-space explosion is a typical problems when modeling system with Petri nets. We present advanced techniques used to overcome this problem, so that we could assess our model is fair for incoming requests, without deadlock; we detailed how we can check that resources are correctly dimensioned.

Finally, we assessed the capabilities of our architecture to be meet the constraints of real-time application. The analysis of the statistical samples for various configurations demonstrated most samples were in a window that is less than 100$\mu$s wide. This provides first results that our architecture may be linked to real-time kernels, and be deployed in real-time distributed applications.

Thus, we achieved the implementation of a middleware whose design is formally assessed, and is tailorable to real-time constraints. This is a first step towards the construction of a proof-based middleware for critical systems.

Later work will consider two main directions. The first one is to adapt our
middleware so that it can serve as a prototype middleware for the ASSERT project, demonstrating its compliance to a proof-based engineering process, and its versatility to application needs. The second direction is to increase the link between models and implementations of our middleware. We will investigate the used of the AADL -Architecture Analysis & Design Language - to ease the tailoring of middleware with respect to application requirements.

Note: PolyORB is free software, supported by AdaCore\(^3\), PolyORB's research activities are hosted by the ObjectWeb consortium\(^4\); for instance the IST European project ASSERT will use PolyORB as a foundation to build a middleware, following advanced engineering methods and process.

References


\(^3\)http://www.adacore.com
\(^4\)http://polyorb.objectweb.org


[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.


