Meta-programming Middleware for Distributed Object Computing

Peter Breitling

Technische Universität München, Fakultät für Informatik,
Boltzmannstr. 3, D–85748 Garching, Germany
Peter.Breitling@in.tum.de
http://www11.in.tum.de

Abstract. A multitude of different middleware technologies exist for distributed object computing (doc). However well-known doc–middleware has practically not established itself in the context of wide–area distributed computing. Prominent examples like CORBA, COM+ or Java/RMI use quite similar distributed object models, that can be mapped pairwise with bridges. But they differ in their meta–programming functionality including methods for object generation, distribution, location, reflection and life–cycle management. A more high–level, abstract model that encapsulates this functionality and that allows the integration with existing doc–middleware and web systems can be a central prerequisite for emerging a web of objects. This paper introduces the design of a meta–programming middleware and its mapping to and application with existing doc– and web–technologies respectively.

1 Introduction

According to [1] meta–programming relates to "programming, where the data represent programs". Wang et al. [3] examined meta–programming mechanisms in the CORBA distributed object computing (doc) middleware. This paper states the term meta–programming in the context of doc as the "adaptability of distributed applications by allowing their behaviour to be modified without changing their existing software designs and implementations significantly". The adaptability of applications has a strong importance for the increasing number of open systems that are composed of complementary technologies.

In contrast to examining meta–programming mechanisms in existing doc–middleware we propose an independent middleware layer specifically for meta–programming. The layer shall provide all means for object– (as data) management and object reconfiguration and adaptation. Common meta–programming techniques can be defined in this layer and this layer can be built upon different doc–middleware and web–technologies.

The focus of the independent meta–programming middleware layer for doc is on the extensible and dynamic object management including:

– *Object life–cycle management:* Common methods for the dynamic creation (compiling), registration, deregistration and destruction of distributed objects.

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 41–48, 2003.

- *Object distribution management:* Object migration and replication between different middleware technologies or platforms.
- *Management events and handlers:* Programmable handlers for observing and reacting onto object life–cycle and distribution management events.
- *Object Reflection:* Reflection mechanisms for object properties including object–type, object–state (e.g. if registered), and –class information dependent of the type (e.g. the object interface) and further extensible meta–data.
- *Object Adaption:* Methods to dynamically create proxy objects for changing an object functionality or for replacing individual objects.
- *Object Reconfiguration:* Manage object references with the ability to reconfigure and dynamically map them to other target objects.

Different doc-middleware technologies in general provide a subset of the described methods and focus on characteristic object distribution and management methodologies[1]. And they apply different methods and semantics for using this functionality. We incorporate the complete life–cycle management into this middleware layer. The layer is capable of generalizing the whole object generation process from the object–source, over the –implementation, up to the distributed instance (service) of the object that can be mapped to different doc–middleware technologies. Thus one result of this work is to provide a common semantic for meta–programming, that can be used independent of the underlying doc–middleware.

Furthermore this middleware proposes a complete isolation of objects that are managed by this middleware. The context of an object includes proprietary access to the underlying doc–middleware, operating system and the network. A complete reconfiguration control over an object requires the ability to intercept every access out of the object to the context. A common example is a simple file access outside an object, which is a common source of problems that prevents the successful reconfiguration and adaptation of an application without changing its implementation. Instead of application specific configuration mechanisms, we propose to fundamentally isolate objects with this middleware layer and introduce the concept of an environment as a single point of access for every type of object. The environment performs the concrete mappings between abstract objects and concrete physical entities on a platform.

To enable these functionalities, especially the life–cycle management and object isolation, the object model itself is refined. We introduce resources that model basic types of objects that can represent programs as data plus the classic object model entities (interface, implementation, instances). The "isolating" environments manage those resources equally. The environments actually provide the meta–programming functionality.

[1] These include specific features like object persistency, transactions, caching, etc. but which are not in the part of the meta–programming layer.

2 The Meta-programming Middleware

The independent meta–programming layer provides functions for the reconfiguration and adaptation of objects. A common definition for an object in doc–middleware "is an identifiable, encapsulated entity that provides one or more services that can be requested by a client" [4]. The question for meta–programming that treats programs as data and converts data to objects and vice–versa arises: Is it all about (service) objects? To effectively reconfigure interobject and data dependencies we raise data and software objects onto the same management level.

2.1 An Abstract Resource-Layer for Meta-programming

We identified five fundamental resource–classes for the meta–programming layer: binaries, documents, interfaces, implementations and instances. For these classes we define common properties that are useful for the management of and meta–programming on these resources. Figure 1 gives an overview of the properties.

class :	binary	document	interface	implementation	instance
type :	mime-type	doc-type	doc-type	interface	interface
state :	a subset of [serializable | registered | persistent]				
name :	if registered , an identifying name that is unique in the registry for the same type				
attributes :	type specific [life-] attributes which are managed by value to the resource.				

Fig. 1. Resources and their properties

For every resource class we use a specific type system[2]. Every resource has a state, which expresses the capabilities with respect to management of an object. If a resource is registered it can have a name, which acts as an identifier local to the registry. Further attributes can be assigned to a resource as simple name / value pairs to store informating along with the object[3].

The binary resource encapsulates any unstructured block of data. In general it is mapped to files or binary streams. For the content–type and subtype information on binary resources we use the MIME media type system. Any other meta–information on the type of a binary resource could be used here, however this system is well–known and established in the context of wide–area internet computing. In this system binary resources can always be serialized and the

[2] We decided to use well–known type systems that are established in the wide–area internet computing.

[3] The attributes remain persistent during the objects life–time. Some attributes can be live–attributes, which are read–only and managed by the system (e.g. the content–length of a resource).

result of a resource serialization from any other resource class results in this (unregistered) binary resource class.

The document meta–object represents structured data. The eXtensible Markup Language (XML) as the state–of–the–art representation for structured documents is applied in the prototype. But any other syntax for structured data could be applied here. XML provides an analogous classification for documents like interfaces for objects. A document meta–object can have a reference to a document–type definition (in form of a DTD or XML–Schema). In this system a document can always be serialized

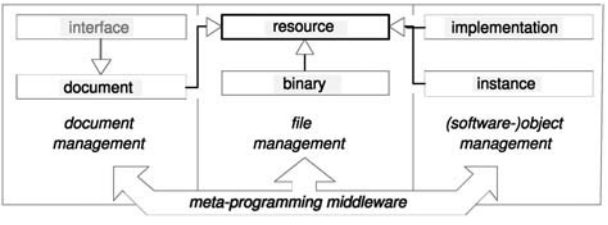

Fig. 2. Resource classes and relations

The interface of an object is a central entity in doc. In this meta–programming middleware it is important for the management and adaptation of objects and for providing reflection mechanisms.

Common RPC systems, such as the Distributed Computing Environment (DCE), and in distributed object environments, such as CORBA define an interface definition languages (IDL) as a standard set of primitive types, method signatures and object interfaces that are mapped to existing languages. We apply the same approach for interfaces in this middleware. However in the extended object model we model interfaces as document resources. In fact interfaces can be managed like documents with structured data containing meta–data for objects. Accordingly the interface resource is subclassed of the document resource. Given the importance of interfaces it is listed among the fundamental resource classes here and presented as a fundamental object. However interfaces are managed and can be accessed as documents in this middleware.

The extended model refines an object into two distinct resource classes, the object implementation and the object instance. The implementation resource is mapped to specific implementations in the underlying doc–middleware. In Java this is represented by the class object, which can be distributed seperately. CORBA in contrast does not allow the explicit distribution management of implementations[4]. The classification for implementations are references to an interface resource which defines the interface for the object–implementation. Dependent of the doc–middleware and platform implementation resources can be

[4] this matches the "object as service" view in CORBA.

serialized. This is required for implementation replication, which can be modelled with the serialization and distribution of implementation resources here.

The instance resource class represents an instantiated object. E.g. CORBA distributed objects are mapped to instance objects. The term object is commonly used for instances. We avoid the term "objects" to refer to resources of class instance here to avoid a confusion with that generic name. An Instance resource in general is represented directly by language objects. In this framework it the base class for deriving reconfigurable application objects. An instance object has a reference to the implementation object. Dependent of the doc–middleware and platform implementation resources can be serialized. This is required for object migration, which can be modelled with the serialization and distribution of instance resources here. Figure 2 gives an overview of all resource classes and their relations.

2.2 The Meta-programming Environment

While the resources are mapped to existing entities and provide a common concept across different technologies, the meta–programming environment is the functional part that has to be implemented for every doc–middleware. The environment is the middle–tier between the resources and physical entities. The connector can be viewed from three functional perspectives:

- *Provide a framework for meta–programming:* Map the abstract resources to the platform and language. Subclassing the resources can create new distributed objects. It implements the meta–programming interfaces.
- *Allow arbitrary configuration of the resource mapping:* The mapping of references to resources and physical entities is transparent to the objects. Dependent of the configuration the mapping could be performed locally or be delegated to other environments. The environment can provide a specific user interface for its configuration.
- *Be connected to other environments / services:* Environments can be interconnected or map the access for specific resource classes to other web–systems. Inter–environment connection architectures can range from hierarchical to peer–to–peer approaches. Existing web–services can be mapped into the environment for specific resources (like a web–server as a binary registry or a XML–database as a document registry and access backend).

The mapping of resources and the interconnection to other environments and services are dependent of the concrete implementation of the environment. These parts are transparent to an object, which requests other resources with the framework. The meta–programming framework itself is specified homogeneously across platforms: The abstract resources act as base classes for extending with application specific, adaptable and reconfigurable objects and the application programming interface (API) provides standards functions for meta–programming.

The meta–programming API consists of six interfaces that are be mapped and implemented on top of the underlying doc–middleware and the platform. In the following sections we describe the functionality and task of each interface.

resource–registry: An environment provides a registry where resources in the environment can be registered. Every doc–middleware has a kind of a registry where objects can be dynamically inserted. In COM+ components are registered in the Windows–Registry, while in CORBA objects are registered in the implementation repository. In this abstract middleware we register and deregister resources to and from the resource registry of the environment. Resources are registered with an arbitrary name and their class. The name must be unique in the registry for that resource class. The registry can be mapped to different technologies for each resource class. E.g. the interface or implementation registry can be mapped to the CORBA interface or implementation repository respectively. Or the binary and document registry can be simply mapped to a local file system or a Web–server. Document resources could be ideally mapped to XML databases. With this approach we can map different technologies into the same semantic concepts. The prototype description will show some concrete examples.

reference–resolve: any registered resources can be located with the resource–resolver. The resolver function takes a class, a name and optionally a type and returns a resolved reference if it can locate a corresponding resource. The reference itself is a placeholder for a resource. A reference can have one of three states: unresolved, resolved and mapped. The programmer in general only works with resolved references and maps it to resources. However a reference could be constructed manually. The namespace of a reference is not specified. An Uniform Resource Locator (URL) could be an example representation for a reference that is unresolved. This is dependent of the resolver implementation.

reference–map: The mapping of a reference results in an accessible resource. It is completely transparent to the program if the resource is actually on a remote platform or was migrated or replicated into the local environment.

resource–cast: The casting functionality is the foundation for a complete life–cycle control including the generation of objects starting with code compilation. A binary resource can then be casted to documents, implementations or instances. The serialization operation is the inverse–function to the cast–operation. It results into a binary resource.

registry–events: The registration and deregistration of resources in the registry is observed with an event handler. Every resource class can be individually monitored[5].

resource–factory: Beside of resolving existing resources, new resources can be created with this interface. A typical metaprogramming task would be to create a binary or document resource source code (e.g. a script), cast it to an implementation resource and register it in the environment.

[5] modern file systems offer the concept of file system–events to monitor a path for file creation or deletion. This can be well matched with the event–handler for a binary resource registry.

3 MetaDOC: The Java Prototype

An implementation of this middleware requires solutions for each of the three main functional parts of the environment: Provide a meta–programming framework API for the programming language, offer configuration mechanisms for the resource mapping and provide an interconnection layer to other environments or web–systems.

MetaDOC [2] is a first implementation of this meta–programming middleware. For the prototype we chose Java. The language provides reflection capabilities, it can dynamically create and load objects and the CORBA technology is an integral part of the Java 2 platform. It contains an ORB–implementation and API's for the RMI and IDL programming model respectively. Figure 3 gives an overview of the supported protocols (and technologies) in the prototype.

	binary	document	implementation	instance
IN (as server)	WebDAV	IIOP (Xindice)	WebDAV	IIOP / RMI
OUT (as client)	HTTP	HTTP	HTTP	IIOP / RMI
INTERNAL REG.	Filesystem	Xindice	Filesystem	JNDI

Fig. 3. MetaDOC–prototype: Supported resource access protocols

The MetaDOC prototype can be used with existing or new Java applications as:

- *a reconfigurable an integrated data and object access layer*: Binary and Documents can be accessed without committing to a protocol or technology (except this middleware) in an application object. This also applies to remotely accessing distributed objects. Abstract names can be chosen to identify resources. These names can be centrally configured with the environment to concrete resources that reside local or in the web or to dedicated ORB and RMI servers in case of remote object access.
- *a remote Java/CORBA object access layer*: As an alternative to directly programming RMI or CORBA IDL files you can use the framework and inherit dedicated objects from the instance base class. These objects can be registered in the environment. No commitment to one of these technologies is needed.
- *a high–level meta–programming*: In Java all meta–programming and reflection mechanisms that are described could be realized proprietarily by the application itself (in fact the prototype is built purely in Java). With this middleware however an application can access this functionality with high–level functions. Specific Java issues regarding the compiler, the classloader and reflection functions are hidden. Secondly we can meta–program remote objects (that exist on another platform) transparently with the same mechanisms.

4 Conclusions and Outlook

To our knowledge, the design of a flexible model that integrates the file–, document– and object–management in one layer for meta–programming is a new approach. The focus of the system is to provide a homogenous concept for meta–programming using the abstract resource–layer and common operations on that model. The middleware is related to and influenced by a multitude of technologies from different research areas including reflective middleware, system management, web–systems and common meta–programming mechanisms in existing doc–middleware.

This is an open system – it can be built on a number of languages and integrate different technologies. The system specifies only the environment framework API and the basic resource model. All other parts including environment implementation details, configuration methods, resolve strategies, supported protocols and environment interconnections are not fixed. This is in the sense of meta–programming. All those parts remain adaptable and reconfigurable.

Wang et al. [3] evaluated three meta–programming approaches in CORBA: pluggable protocols, smart proxies and interceptors. Here we can extend the list by pluggable technologies (with the open system approach), reconfigurable resource resolving (resource–resolve and –map) and methods for the object generation and life–cycle control (resource–casting).

The netscape cofounder Marc Andreesen stated in the year 1996, that the "HTTP protocol will be replaced by the IIOP protocol in just a few years". Behind that statement is the vision of a worldwide web of objects where applications can be dynamically composed of existing, reusable components.

No single doc–middleware will probably ever replace all existing and forthcoming (problem oriented) protocols with a new universal protocol. In this middleware we don't specify new protocols or doc–capabilities. We just provide one semantic umbrella where various web– and doc–technologies can be assembled. This can be a small step towards a more interoperable and visible web–of–objects.

References

1. J. Barklund. Metaprogramming in logic. In A. Kent and J. Williams, Encyclopedia of Computer Science and Technology., 1994.
2. P. Breitling. Metadoc homepage. http://www.metadoc.org/.
3. D. Schmidt N. Wang, K. Parameswaran and O. Othman. Evaluating meta-programming mechanisms for orb middleware. In *IEEE Communication Magazine, special issue on Evolving Communications Software*, volume 39, 2001.
4. OMG. The comon object request broker: Architecture and specification. Technical Report PTC/96-03-04, Object Management Group, 1996. Version 2.0.