

AN APPROACH TO BUILDING REFLECTIVE COMPONENT-BASED MIDDLEWARE PLATFORMS

Nikos Parlavantzas, Gordon Blair, Geoff Coulson

Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster LA1 4YR,
UK

contact: [parlavan, gordon, geoff]@comp.lancs.ac.uk

ABSTRACT

It is now well established that middleware platforms must accommodate an increasingly diverse range of requirements which arise from the needs of both applications and underlying systems. Moreover, it is clear that to achieve this accommodation, platforms must be capable of both deployment-time configurability and run-time reconfigurability. This paper describes our approach to the design and implementation of a middleware platform that addresses these needs. The approach suggests the use of a lightweight component model, employs the notion of component frameworks, and uses reflective techniques to facilitate reconfiguration. We also discuss our plans for applying our approach in the .Net environment.

1. Introduction

It is now well established ([Blair,98], [Schmidt,99], [Kon,00], [Hayton,98]) that middleware platforms must accommodate an increasingly diverse range of requirements imposed both by *applications* (e.g. real-time, multimedia, 7x24, collaborative) and by *underlying systems* (e.g. workstations, PDAs, embedded systems, wireless networks and high speed networks). To achieve this accommodation, platforms must be capable of both deployment-time *configurability*, and run-time *reconfigurability*. As an example of configurability, the same middleware technology should be deployable in environments ranging from embedded processors in refrigerators to scientific supercomputers. As an example of reconfigurability, middleware should be capable of dynamically loading and unloading protocols, stream media processors etc., and be able to intelligently adapt to varying qualities of connectivity in a mobile computing environment involving various types of network and limited RAM [Blair,98].

Unfortunately, the current generation of mainstream middleware is, to a large extent, heavyweight, monolithic and inflexible and, thus, fails to properly address such needs. There have been some efforts to introduce (re)configurability (e.g. Iona's Orbix2000), but these are typically piecemeal, ad-hoc, and usually involve selection between a fixed number of options. In our opinion, a more systematic, principled and dynamic solution is needed. To this end, we are proposing an approach to building configurable and reconfigurable middleware that relies on two complementary technologies: *components* and *reflection*.

Component technology has recently gained widespread acceptance as a promising approach to the construction of flexible software systems [Szyperski,98]. Components retain their visibility in a deployed system, enabling the system to be easily adapted by adding, removing or replacing components. Currently, component technology is almost exclusively applied at the application level. Most component models are inseparably bound to an underlying support infrastructure that hides distribution and other extra-functional concerns from component developers. Our approach, in contrast, is to provide a lightweight component model that is independent of any such infrastructure, thereby enabling the *middleware itself* (and other systems software) to be built as an assembly of components. To contain the complexity of designing the required component architecture and managing its configurability, we are employing the notion of component frameworks, which provide domain-specific constraints for composing components.

Reflection is a useful technique for “opening up” black-box systems and supporting inspection and adaptation of their internal structure and behaviour [Maes,87], [Kiczales,91]. The technique suggests the use of *causally connected* data structures that represent (“reify”) aspects of the system, and the provision of *meta-interfaces* through which these “reified” aspects can be manipulated. In our approach, reflection is applied in conjunction with component frameworks to support dynamic reconfiguration of component assemblies in a controlled way.

The rest of this paper is structured as follows. First, section 2 details the baseline technologies that underpin our approach for flexible middleware. These are: *i*) a simple, general *component model*, which is based on the core of Microsoft COM, *ii*) *component frameworks*, which facilitate the structuring of the architecture, and *iii*) *reflection*, which together with component frameworks supports *dynamic reconfiguration management* and helps constrain the scope of reconfigurations and ease the task of integrity maintenance. Following this, section 3 presents the architecture of OpenORB v2, an experimental middleware platform that has been built in terms of these technologies. Subsequently, section 4 discusses our plans for an application of our technology to .Net, and section 5 discusses related work. Finally, we present our conclusions in section 6.

2. Baseline Technologies and Frameworks

2.1 Component model

The first baseline technology underpinning our approach is the use of a general-purpose, lightweight, non-distributed component model. The component model was designed to only include aspects that are essential in supporting the notion of a component. The motivation was to maximize the applicability of the model and thus enable it to be used for composing both applications, and middleware and other systems software.

The component model is inspired by Microsoft COM (and, indeed, our current implementation employs aspects of COM). Components are opaque implementations that can generate instances, which offer multiple interfaces. The model also provides basic, mechanism-level support for reflection, which is inadequately addressed in COM. Specifically, the following reflective features differentiate the component model from COM:

- (i) *Support for required interfaces.* Component instances explicitly specify their *required* interfaces and offer an interface (*IREceptacles*) through which they can be externally connected (by a third party) to other instances that *provide* matching interfaces. This is a generic mechanism that can be used to make dependencies between components visible and modifiable.
- (ii) *Interception.* Components offer an interface (*IMetaInterception*) that allows one to associate (dissociate) *interceptors* with (from) some particular interface. Interceptors insert wrapping behaviour before or after method invocations on the specified interface, and can be added, removed and reordered as desired. The mechanism can be used, for example, to inject monitoring code or other lightweight extra-functional behaviours (e.g., security checks).
- (iii) *Extensible component-level metadata.* Components are associated with an extensible set of *attributes* that are contained within the component's shipping format. The component model does not prescribe the meaning of the attributes; rather, this is a generic mechanism with many possible uses. For example, attributes can be used to provide specification and verification information needed for reuse. They can also be used to declaratively specify requirements for infrastructure services that are offered by container-based component frameworks (e.g., transactions) or to specify QoS requirements on underlying resources.
- (iv) *Introspection.* All the meta-information associated with a component can be exposed before deployment or at runtime. This comprises information about the types of all provided and required interfaces as well as the names and values of the component-level attributes.

As mentioned, the component model has been implemented atop a subset of COM [Clarke,01]. The implementation (called *OpenCOM*) ignores higher-level features of COM, such as distribution, security and transactions and relies only on the core of COM (mainly, the binary level standard and the infrastructure for dynamic deployment of components in an address space). We chose COM as the basis of our component model because it is widely-used, inherently language independent, and significantly more efficient than other component models such as JavaBeans. However, due to its lightweight nature, we believe that our component model can be easily supported on top of all other major component technologies (e.g., Java, .Net).

2.2 Component Frameworks

Although *necessary*, constructing middleware using a general-purpose component model is not in itself *sufficient* (we believe) for fulfilling our goals of modifiability and extensibility. The component model provides the basis for interoperation but does not specify how one can build meaningful assemblies that solve specific problems. To address this concern, we are exploiting the concept of *component frameworks (CFs)* [Szyperski,98].

A CF is defined by Szyperski as a “collection of rules and interfaces that govern the interaction of a set of components plugged into it”. Essentially CFs are reusable architectures that apply to specific domains and are designed to be instantiated in terms of components. For example, we employ a *protocol CF* that embodies reusable

rules and strategies for composing “plugged-in” protocol components in order to build protocol stacks.

The main motivation for using CFs is to facilitate composability; that is, to enable components originating from independent sources to work together effectively. Importantly, by constraining the interactions among their plug-ins in a domain-relevant manner, CFs provide a means of enforcing desired architectural properties and invariants. These properties can be both functional (e.g., how some functionality is decomposed among plug-ins) and extra-functional (e.g., modifiability or performance of plug-in assemblies). As additional benefits, CFs simplify component development through design reuse, enable lightweight components, and increase the understandability and maintainability of systems.

From a practical point of view, a CF is a set of design artifacts together with (optionally) software that supports or enforces the CF rules at runtime. A CF can be viewed as a set of *collaborations* that define specific *roles* that should be played by participating plug-ins. We model CFs using UML ‘parameterized collaborations’ along with additional modeling tools and techniques (e.g., OCL constraints, free text). Naturally, the necessary support software is itself designed and implemented as a set of components in our component model.

We apply the CF concept in two primary forms. An *encapsulated* CF describes the realization of a system (a behavioural unit) as an assembly of components. On the other hand, an *open* CF describes a scope of interactions among components that serve some common purpose (e.g., how to access a shared service). From this perspective, a component can participate simultaneously in multiple open CFs but at most one encapsulated CF¹. If one considers a CF-based assembly as a higher-level, composite “component”, then this can plug into another CF introducing a hierarchical structure into the component system. This property is useful in the context of our dynamic reconfiguration approach, as will be seen next.

2.3 CFs and Reflection

Reflection is a valuable technique for facilitating dynamic reconfiguration. In our approach, we apply reflection to “open up” CF-based component assemblies. Specifically, a CF-based assembly is explicitly represented (reified) by a component, termed CFR (CF representative), that maintains meta-information about the current configuration of plug-ins, monitors significant events emitted by the plug-ins and effects changes on them. These changes may involve the invocation of operations or setting properties on its plug-ins, or modification of the plug-in configuration (i.e., adding/removing/connecting/disconnecting plug-ins using component model primitives). Importantly, the CFR exposes a *CF-specific* meta-interface that allows clients to achieve reconfiguration in a principled and controlled way.

Using CFRs as the locus of reconfiguration has a number of important benefits: *i*) it *constrains the scope and effect* of reconfiguration operations due to the hierarchical structure of CFs, *ii*) it *separates concerns* between reconfiguration functionality and domain functionality by isolating the former within the CFRs, and *iii*) it contributes to *maintaining integrity* in the face of dynamic change by exploiting the domain-specific knowledge and built-in constraints that are embodied within the CF.

¹ This terminology is analogous to encapsulated/open collaborations in Catalysis [D’Souza,98]. From now on, the term CF will imply “encapsulated CF” unless specified otherwise.

To give a concrete example, consider our multimedia streaming CF (see section 3.1), which accepts media filter plug-ins. The CFR maintains a causally connected graph representation of its current configuration of media filters and offers a meta-interface for manipulation of this graph. Importantly, it exploits its implicit domain-specific knowledge to manage requested reconfiguration operations with minimum perceived disruption of the media stream. For example, it buffers data while reconfigurations are taking place.

A final point worth emphasising is that CFs can enforce a desired level of integrity across reconfiguration operations, which can be appropriately traded-off against the degree of afforded flexibility and efficiency. When dynamically inserting new components, integrity management can range from simple type conformance checks (e.g., checking that plug-ins implement a required set of interfaces) to more sophisticated operations. For example, when replacing a plug-in, a CF may use a simple notification protocol to notify registered clients of the pending replacement so that they refresh their references. Alternatively, it may employ a sophisticated mechanism to cause the instance to enter a state where no threads are currently executing within it, transfer its state to a replacement and update the necessary references to point to the replacement.

3. The OpenORB Architecture

3.1 Overview

OpenORB [Coulson, 02a] is structured as a top-level CF (termed the *service CF*) that contains further CFs and components organized in three layers (named the *binding layer*, *communications layer* and *resources layer* respectively). Each component/CF in a given layer is allowed to use functionality offered by components/CFs in the same or lower layers. The layers are logical and serve to clarify the architecture and offer guidance to CF developers for the design and placement of new CFs. The service CF manages the configuration of hosted CFs and imposes policies concerning dynamic changes in this configuration. The second level CFs then address more focused sub-domains of middleware functionality (e.g., binding establishment and thread management) and enforce appropriate sub-domain specific policies.

OpenORB's hierarchical structure opens up two distinct dimensions of flexibility. First, the service CF can be configured by selecting the set of CFs that will initially populate the layers plus appropriate reconfiguration policies. This configuration defines a so-called *middleware architecture* that is published to developers who want to use/configure/extend the platform. Our currently implemented middleware architecture consists of the CFs shown in the static view of Figure 1 and employs a simple reconfiguration policy that enforces integrity constraints by forbidding the removal of certain key CFs. However, we see this as only one possible middleware architecture that can be built using our approach. Different architectures can be defined for different platforms or application domains. Thus, when we change the initial CFs in the various layers, we are essentially defining a new platform architecture that potentially has a different programming model and a different set of meta-interfaces. For example, a more sophisticated architecture could include CFs that deal with transactions, security, or persistence.

Second, a particular instantiated middleware architecture is configurable in terms of dynamically introducing new CFs (as long as this is allowed by the policies of the service CF) and replacing/ customising/ extending the existing second-level CFs (both statically and dynamically). For example, in the current OpenORB middleware architecture, a signal processing CF could be added dynamically in the communications layer, or the thread management CF could be dynamically customised with a new scheduler component.

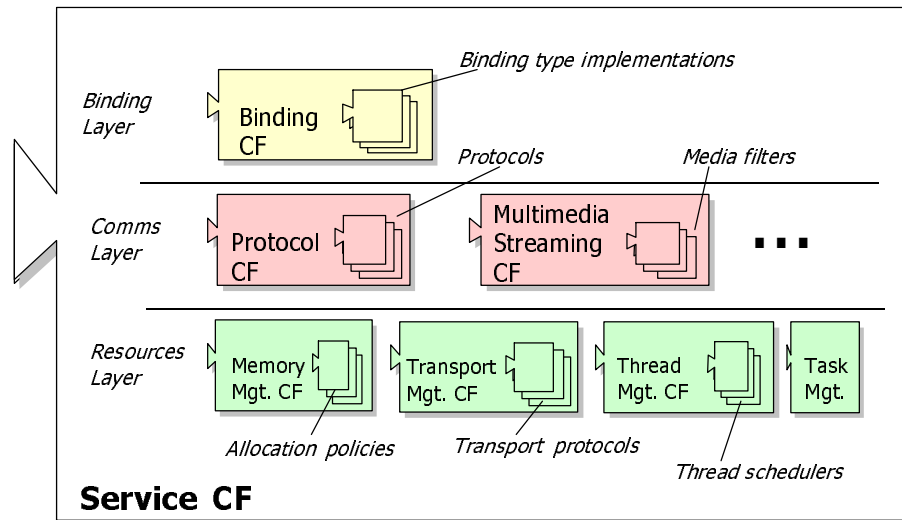


Figure 1: OpenORB Architecture

In more detail, our currently implemented middleware architecture is structured as follows: First, the bottom, *resources layer* contains the *memory*, *transport*, and *thread management* CFs which respectively accept memory allocation policies, transport protocols and thread schedulers as plug-ins. Moreover, it contains a *task management* component, which participates in the resource CF (see section 3.3). Next, the *communications layer* contains *protocol* and *multimedia streaming* CFs, which define an assembly environment for composing *protocol* and *media filter* plug-ins respectively. Finally, the *binding layer* contains a *binding CF* that accepts *binding type implementations* that provide communication and coordination services with various semantics (e.g., remote method invocation, publish-subscribe, message queuing, group communications, continuous media streaming, auction protocols, etc.). The binding layer is a crucial part of the architecture because it determines the programming model offered to middleware users. Further details of the binding CF can be found in [Coulson,02a], [Parlavantzas,01].

The following section outlines the design of (encapsulated) CFs in OpenORB in a general way. Following this, section 3.3 covers in more detail the resource CF, which is an example of an open CF. The resource CF is important because of its potential applicability in the .Net environment (see section 4).

3.2 CF Design

Encapsulated CFs in OpenORB are designed according to a common pattern. The CFR has the responsibility to manage the lifecycle of its plug-ins; a generic lifecycle interface is defined for this purpose. The CRF either instantiates plug-ins itself or obtains them from an outside source, and has the obligation to dispose of them.

Moreover, the CFR has the responsibility to resolve and manage the dependencies of its plug-ins. The CFR/plug-in relationship does *not* imply strict encapsulation in the sense that the internals are not directly accessible from the outside or that all communication with the outside must go through the CFR.

CFs can accept multiple types of plug-ins which can interact directly or indirectly with each other and the CFR in various configurations. Note also that there can be many instances of a CFR and the associated assembly at run-time. As seen in section 2.3, CFRs offer meta-interfaces that allow dynamic inspection and reconfiguration of the assembly. The meta-interfaces are CF-specific in order to exploit the domain-specific restrictions of the CF. However, for reasons of consistency and generality, we do provide conventions for the structure of meta-interfaces as well as a simple, uniform meta-interface with operations to retrieve, add and remove plug-ins.

Another recurring need in the design of the CF-specific reconfiguration functionality is the ability to customize aspects of this functionality itself. This is addressed by employing *reconfiguration policy* plug-ins that externalize decisions made by the CFR. For instance, CFs frequently employ reconfiguration policy plug-ins that customize the lifecycle management functionality of the CFR. Also, the service CF employs such plug-ins that either allow or veto proposed reconfigurations on the basis of the current plug-in configuration, and on meta-data packaged with the to-be-inserted plug-in. These policy plug-ins are usually themselves subject to static or dynamic replacement.

3.3 The resource CF

The role of the resource CF is to provide a simple and uniform interface for fine-grained resource management based on the abstraction of a *task*. Tasks represent units of computation to which resources are allocated and resource usage is charged. The framework enables accurate resource accounting and allows explicit control of the distribution of resources among different logical activities, which is highly beneficial for QoS-aware middleware systems [Blair,99], [Duran,02].

The CF is designed to be extensible in order to be able to capture diverse types of middleware-managed resources, including memory, processing, and communication resources. Moreover, it is designed to provide maximum control to its users by exposing low-level primitives that increase the available implementation options. It is not a goal of the framework to offer transparency over task and resource aspects. For instance, the framework does not attempt to hide task boundaries or impose a fixed component-to-task mapping (e.g., a component instance runs always in a single task). Such constraints can always be added on top of the CF together with appropriate support functionality (e.g., support for task switching).

The resource CF is an open CF and defines a collaboration between the following three roles played by participating components: *task manager*, *resource manager* and *resource user* (see Figure 2). The task manager creates and terminates tasks. Tasks are dynamic entities organized into a hierarchy whereby a task is created in the context of a parent task; the hierarchy serves to delineate computations/resources that contribute to the same goal. Resource managers create and manage resources of a certain type (e.g., threads, buffers). Resources can also be organized into hierarchies whereby a higher level resource builds on lower-level resources (e.g., a user-level thread builds on a scheduler context). A resource is associated with a single task but the association is dynamically controllable. For instance, a thread (resource) can be

transferred to a different task to reflect the fact that it performs work contributing to another goal. Finally, resource users access and manipulate resources and tasks.

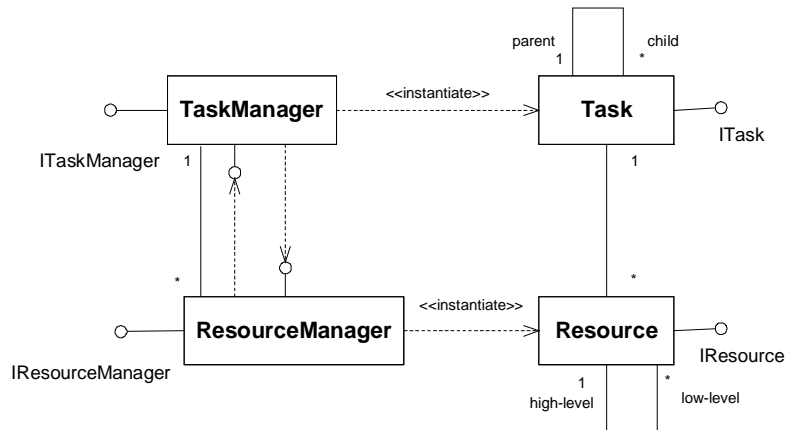


Figure 2: Structure diagram for the resource CF

The CF specifies rules that govern the cooperation between the roles. More specifically, resource managers have the responsibility to enforce resource control and task-based allocation and accounting. In addition, they must keep track of the dynamic associations between resources and tasks. The task manager acts as the access point for obtaining resource allocation information and also offers support functionality. By assigning most of the responsibilities to resource managers, this design encapsulates the large diversity in resource types, resource management mechanisms and policies, and fosters extensibility.

Generic COM interfaces are defined for all the basic CF roles (task manager, resource manager, task, resource). For example, the interface for resources (*IResource*) has operations to return the associated task, get/set the QoS of the resource (represented as a string), return the higher-level resource manager that is currently using the resource, and transfer this resource to a different task. The interface for resource managers (*IResourceManager*) includes operations to create resources and return the set of managed resources. Note that a resource manager implicitly associates created resources to the task within which the client executes unless requested otherwise.

The resource CF is instantiated within the OpenORB architecture as outlined in the following. The role of the task manager is played by the task management component in the resources layer. Any component can be a resource user including application-level components and CFs. The role of the resource manager is played by CFs in the resources layer. More specifically, a single CF exposes resource managers for multiple resource types. For example, the thread management CF exposes *threads* and *scheduler contexts* as resource types. A scheduler context represents an allocation of CPU processing time, which is distributed among a number of supported threads according to a scheduler context-specific scheduling policy (realised by a thread scheduler plug-in). Similarly, the memory management CF provides *buffers* and *memory contexts* and the transport management CF provides *transport access points* and *network contexts* as resource types.

With regards to tasks, the architecture employs a separation into *middleware tasks* and *application tasks*. The former are associated exclusively with the middleware implementation and represent shared activities, such as book-keeping, event logging, or dynamically linking a plug-in. The associated task hierarchy is aligned with the CF hierarchy. There is a task associated with the top-level CF (child of the root task), which has subtasks for each binding/communications layer CF. This design isolates CFs from each other and enables accurate tracking of their resource usage.

Application tasks are associated with middleware users. Importantly, when an application component invokes a middleware service, the middleware implementation spawns one or more subtasks of the application task to represent middleware processing performed for the benefit of the application. Those subtasks encompass the necessary resources to support the requested service (e.g., sockets and buffers needed for establishing a binding to a remote object).

The genericity of the resource CF interfaces allows dynamic inspection and adaptation of tasks and resources at any hierarchy level in the system and in potentially unanticipated ways. Finally, note that the creation of multiple tasks has only a small performance overhead due to the use of a lightweight task-switching mechanism.

4. Application to .Net

We are presently planning to implement and evaluate our approach to building flexible component architectures in the context of .Net [Microsoft,02]. First, we plan to transfer our component model to the new environment by imposing simple conventions and patterns on the native components (called “*assemblies*”). Custom attributes can be used to facilitate the development of our components using .Net programming languages. We expect that the extensive .Net facilities for introspection and dynamic type generation will be very useful in implementing the reflective features of our component model.

The .Net remoting system is a good target for further exploration. Interestingly, this is structured as a framework with numerous extensibility points, such as pluggable channels and formatters, message filters and custom properties. A preliminary investigation, however, has revealed that the remoting framework has a number of limitations from our perspective. First, it does not have adequate support for dynamic reconfiguration. While one can configure aspects of the framework at various times, the resulting configurations remain usually static. For example, the set of interception-based services associated with a context and implemented as chains of message sinks are fixed at context creation time and an object stays in a context for life. Second, it only realises a remote method invocation binding type and it has no support for other interaction mechanisms and programming models (e.g., streams, events and group communications). Finally, it does not offer any special support for resource management to application or middleware developers (e.g., channel developers).

To address those limitations, we plan to investigate how the remoting framework can be reengineered using our component-based approach. We will initially attempt to structure the framework design and implementation as a set of CFs in order to clarify the architecture and make it easier to extend. We will then develop CF-specific meta-interfaces for managing the dynamic reconfiguration of component assemblies (e.g., message sink chains) in a controlled way. Furthermore, we plan to

expand the scope of the framework to support multiple binding types similarly to OpenORB. At the same time, we plan to expose resource management functionality by extending the .Net infrastructure (CLR or system class libraries). We will then apply the resource CF in order to provide a uniform, task-based interface for resource management. This will enable the development of a QoS-aware remoting system, which will better support the timeliness and predictability of .Net applications.

5. Related Work

There is currently increasing interest in the application of component technology in systems software. Knit [Reid,00] is a component model which is being used for building custom operating systems. However, the model is specifically designed for statically composing systems; the components and their interconnections cannot change after the system is configured and initialised. Also, the component interfaces are not object-based and the model mainly targets low-level, C code. MMLite [Helander,98] is an operating system built using COM components. It provides limited support for dynamic reconfiguration through a “mutation” mechanism, which enables the replacement of a component implementation at run-time. However, it does not provide any framework to support and manage this reconfiguration. There are also a growing number of ‘component models’ appearing in literature, which are targeted at specific domains such as building modular routers [Kohler,00], protocol stacks [Matthijs,99] and field devices [Nierstrasz,02]. Our work is complementary to those, since we can accommodate such designs as CFs within our CF-based approach.

With regards to middleware platforms, COMERA [Wang,98] offers customisation of the remoting infrastructure of COM (i.e., DCOM), but the components are coarse-grained; we apply more aggressive componentisation guided by a set of CFs. DynamicTAO [Kon,00] and LegORB [Roman,00] are component-based reflective ORBs that rely on a set of ‘configurators’ that maintain dependencies among components and provide ‘hooks’ at which components can be attached or detached dynamically. Their approach to reconfigurability seems to favour replacing shared, platform-wide components (e.g., the scheduling strategy or the IIOP protocol); it is not clear how finer-grained reconfiguration can be achieved (e.g., changing the scheduling parameters of a thread of a specific binding), and the interface that triggers reconfiguration is generic and potentially unsafe. In [Joergensen,00], a component-based ORB architecture is presented that supports run-time reconfiguration in a Java environment on a per-remote method invocation basis. The configuration is driven by declarative, application-specific policies. The presented meta-interface is thus very useable, but its power is restricted to selecting between alternative implementations of a given component type. Our meta-interfaces potentially support a higher degree of flexibility (e.g., dynamically restructuring component graphs in the protocol and multimedia streaming CFs). Declarative meta-interfaces for specific CFs are, of course, also possible in our architecture. Jonathan [Dumant,98], FlexiNet [Hayton,98], FlexiBind [Hanssen,99], TAO [Schmidt,99] and Quarterware [Singhai,98] are other flexible ORBs but they are structured as language-specific, object-oriented frameworks and do not conform to any component model.

COM+ [Microsoft,00], Enterprise JavaBeans [Sun,00] and the CORBA Component Model [OMG,99] are all examples of container-based CFs, which aim at a separation between functional aspects (implemented by application components) and extra-functional aspects and services (implemented by the ‘container’). The limitation of these systems is that the container implementation remains a black box;

only predetermined services are provided and their configurability is static and limited. Research efforts to “open-up” the container are currently under way (e.g., [Andersen,01]). Our current middleware architecture does not constrain the selection of a CF for application components allowing such container-based CFs to be implemented *on top* of an extended version of our platform.

Finally, in the area of resource management, our task concept is similar to a number of other proposed abstractions for resource ownership, such as *resource containers* [Druschel,99] and *activities* in Rialto [Jones,95]. However, in those systems, the resource ownership abstraction is implemented by an operating system in order to expose kernel-level resources and there is no emphasis on extensibility with regards to resource types.

6. Conclusions

In this paper, we have presented an approach for the design of flexible middleware platforms (and other systems software) that relies upon component technology and reflection. In particular, the approach proposes the use of a general-purpose, lightweight, component model with low-level reflective features as the basic common infrastructure. The approach exploits multiple CFs to provide extensibility and guidance for building component assemblies targeted at specific domains. Finally, reflection is applied in conjunction with the CF structuring principle to provide runtime reconfiguration that is consistent with domain-specific constraints.

Following that, we described the architecture of OpenORB v2, an experimental middleware platform that has been implemented in terms of this approach. The OpenORB architecture is structured as a top-level CF that accommodates an extensible number of second-level CFs and supports the construction of a family of configurable and reconfigurable middleware platforms. Next, we discussed our plans with respect to .Net, which include implementing our minimal component model on top of this new environment, expanding the scope of the remoting framework to support multiple binding types and applying our resource CF to expose resource management aspects.

We believe that our approach is a highly promising basis for building configurable and reconfigurable systems software. To validate this claim, apart from .Net, we are also applying the approach to other non-middleware domains, most notably programmable networking systems [Coulson,02b].

References

- [Andersen,01] Andersen, A., Blair, G.S., Goebel, V., Karlsen, R., Stabell-Kulø, T., and Yu, W., “Arctic Beans: Configurable and Re-configurable Enterprise Component Architectures”, Work in Progress session at Middleware 2001, Heidelberg, Germany, November 2001. Published in IEEE Distributed Systems Online, Vol. 2, No. 7, 2001.
- [Blair,98] Blair G.S., Coulson G., Robin P. and Papathomas M., “An Architecture for Next Generation Middleware”, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Davies N.A.J., Raymond K. & Seitz J. (Eds.), The Lake District, UK, pp. 191-206, 15-18 September 1998.

- [Blair,99] Blair, G.S., Costa, F., Coulson, G., Duran, H., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., and Stefani, J.B., "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp115-134, 1999.
- [Clarke,01] Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N., "An Efficient Component Model for the Construction of Adaptive Middleware", Proceedings of the IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001), Heidelberg, Germany, November 2001.
- [Coulson,02a] Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", ACM Distributed Computing Journal, Vol 15, No 2, pp 109-126, April 2002.
- [Coulson,02b] Coulson, G., Blair, G.S., Hutchison, D., Ueyama, J., Ye, I., Lee, K., and Surajbali, B., "NETKIT: A Software Component-Based Approach to Programmable Networking", Computing Dept. Internal, Report, Lancaster University, 2002.
- [Druschel,99] Druschel, P., Banga, G., and Mogul, J.C., "Resource Containers: A New Facility for Resource Management in Server Systems", Proc. Third Symposium on Operating Systems Design and Implementation (OSDI '99), New Orleans, LA, February 1999.
- [D'Souza,98] D'Souza, D. and Wills, A., "Objects, Components, and Frameworks with UML - the Catalysis Approach", Addison-Wesley, 1998.
- [Dumant,98] Dumant, B., Dang Tran, F., Horn, F. and Stefani, J.B., "Jonathan: an open distributed processing environment in Java", Middleware'98, The Lake District, U.K., September 1998.
- [Duran,02] Duran-Limon, H., Blair, G.S., "Reconfiguration of Resources in Middleware" In the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002), San Diego, CA, January, 2002.
- [Hanssen,99] Hanssen, Ø., and Eliassen, F., "A Framework for Policy Bindings", Proc. DOA'99, Edinburgh September 1999, IEEE Press.
- [Hayton,98] Hayton, R., Herbert, A., and Donaldson, D., "Flexinet: a flexible, component oriented middleware system", Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications, Sintra, Portugal, 7-10 September 1998.
- [Helander,98] J. Helander and A. Forin. "MMLite: A Highly Componentized System Architecture". In Proc. of the Eighth ACM SIGOPS European Workshop, pages 96--103, Sintra, Portugal, Sept. 1998.
- [Joergensen,00] Joergensen, B.N., Truyen, E., Matthijs, F., and Joosen, W., "Customization of Object Request Brokers by Application Specific Policies". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [Jones,95] Jones, M.B., Leach, P.J., Draves, R.P., Barrera, J.S., "Modular Real-Time Resource Management in the Rialto Operating System," Proc. of the Fifth Workshop on Hot Topics in Operating Systems, May 1995.
- [Kiczales,91] Kiczales, G., des Rivières, J., and Bobrow, D.G., "The Art of the Metaobject Protocol", MIT Press, 1991.
- [Kohler,00] Kohler, E., Morris, R., Chen, B., Jannotfi, J., Kaashoek, M., "The Click Modular Router", ACM Transactions on Computer Systems 18(3), Aug 2000, pg 263-297
- [Kon,00] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [Maes,87] Maes, P., "Concepts and Experiments in Computational Reflection", In Proceedings of OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.
- [Matthijs,99] F. Matthijs, "Component Framework Technology for Protocol Stacks", Phd. Thesis, K.U.Leuven, ISBN 90-5682224 -1, December 1999

- [Microsoft,00] Microsoft, COM Home Page, <http://www.microsoft.com/com/default.asp>; Last updated: 01/06/2000.
- [Microsoft,02] Microsoft, .Net Home Page, <http://www.microsoft.com/net>
- [Nierstrasz,02] Oscar Nierstrasz, Gabriela Arévalo, Stephane Ducasse, Roel Wuyts, Andrew Black, Peter Müller, Christian Zeidler, Thomas Gensler, Reinier van den Born, "A Component Model for Field Devices" Proceedings First International IFIP/ACM Working Conference on Component Deployment, ACM, Berlin, Germany, June 2002
- [OMG,99] Object Management Group, CORBA Components Final Submission, OMG Document orbos/99-02-05.
- [Parlavantzas,01] Parlavantzas, N., "Design of a Binding Component Framework", Lancaster University Technical Report, MPG-01-03.
- [Reid,00] A. Reid, M. Flatt, L. Stoller, J. Lepreau, E. Eide "Knit: Component Composition for Systems Software". In proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), Usenix Association, pp. 347-360, October 2000.
- [Roman,00] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., "LegORB and Ubiquitous CORBA", in Workshop on Reflective Middleware, IFIP/ACM Middleware'2000, IBM Palisades Executive Conference Center, NY, April 2000.
- [Schmidt,99] Schmidt, D.C., and Cleeland, C., "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine Special Issue on Design Patterns, April, 1999.
- [Singhai,98] Singhai, A., Sane, A. and Campbell, R., "Quarterware for Middleware", 18th IEEE International Conference on Distributed Computing Systems (ICDCS 1998). Amsterdam, The Netherlands. May 1998.
- [Sun,00] Sun Microsystems, Enterprise JavaBeans Specification Version 1.1, <http://java.sun.com/products/ejb/index.html>.
- [Szyperski,98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
- [Wang,98] Wang, Y.M. and Lee, Woei-Jyh, "COMERA: COM Extensible Remoting Architecture," in Proceedings of COOTS, April 1998.