

THE DESIGN OF A CONFIGURABLE AND RECONFIGURABLE MIDDLEWARE PLATFORM

Geoff Coulson¹, Gordon S. Blair², Michael Clarke¹ and Nikos Parlavantzas¹

¹Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster LA1 4YR,
UK

²Dept of Computer Science,
University of Tromsø,
N-9037 Tromsø,
Norway
(On leave from Lancaster University)

contact: [geoff,mwc,parlavan]@comp.lancs.ac.uk; gordon@cs.uit.no

ABSTRACT

It is now well established that middleware platforms must accommodate an increasingly diverse range of requirements arising 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 a middleware platform that addresses these requirements. The platform is built using a well-founded lightweight component model, uses reflective techniques to facilitate (re)configuration, and employs the notion of component frameworks to manage and constrain the scope of reconfiguration operations. Importantly, the platform also aims to achieve high performance and a level of standards conformance (e.g., with CORBA and COM). We demonstrate that, despite its high degree of configurability, the platform performs on a par with standard commercial CORBA ORBs.

1. Introduction

It is now well established ([Blair,98], [Schmidt,99], [Kon,00a], [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). Moreover, 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., 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 requirements. There have been some efforts to introduce (re)configurability (e.g. in 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.

At the same time, *component technology* [Szyperski,98] has recently emerged as a promising approach to the construction of configurable software systems. With component technology, one can configure and reconfigure systems by adding, removing or replacing their constituent components (importantly, components are packaged in a binary form and can be dynamically deployed within an address space). Additional benefits of component technology include increased reusability, dynamic extensibility, improved understandability, reduced development costs, and better support for long term system evolution. It should be noted, however, that current component architectures provide little or no support for *integrity management*; system integrity can easily be compromised if run-time reconfiguration operations are not carried out with great care.

Acknowledging the need for (re)configurable middleware and recognising the emergence of component technology, we are carrying out research on *configurable and reconfigurable component-based middleware*. The present generation of component models (e.g., Enterprise JavaBeans [Sun,00] or the CORBA Component Model [OMG,99]) already support distribution and other extra-functional concerns. However, they are inseparably bound to an underlying support infrastructure that hides extra-functional concerns from component developers. This means that component technology can only be applied at the *application level*. Our approach, in contrast, is to provide a lightweight, non-distributed, language independent component model that is independent of any such infrastructure, thereby enabling the *middleware itself* to be built using components. Furthermore, our design is highly *reflective* [Maes,87], [Kiczales,91]; in other words, the component configurations that comprise both the middleware and the applications are associated with *causally connected*¹ data structures (called *meta-structures*) that represent (or, in reflection terminology, ‘*reify*’) aspects of the component configurations, and offer *meta-interfaces* through which these reified aspects can be inspected, adapted and extended. The use of reflection facilitates the management of run-time reconfiguration, and also helps address the issue of integrity management referred to above.

Our previous research [Blair,98], [Blair,99], [Costa,00] has focused on the development of architectural abstractions for reflective, component based, middleware. The outcome of this work has been a prototype, called OpenORB v1, that was implemented in the Python language. This implementation focused primarily on achieving as great a degree of flexibility as possible. Our current focus, described in this paper, is to develop our prototype architecture (now referred to as OpenORB v2) and to implement it in a more pragmatic and ‘realistic’ environment [Parlavantzas,00]. This re-design and re-implementation is the subject of this paper.

In our OpenORB v2 implementation, while continuing to strive for flexibility, we additionally attempt to meet the following objectives:

- *efficiency*; in the worst case, performance should be on a par with that of conventional middleware platforms, and in the best case (e.g. in the case of cut-down configurations) it should be significantly *better*;

¹ Causal connection means that when the representative meta-structure is altered, so is the underlying middleware/ application, and vice versa. This causal connection is the essence of the notion of computational reflection.

- *standards conformance*; we attempt to maintain backward compatibility with key middleware standards; specifically, Microsoft's COM component model and the OMG's CORBA distributed programming environment;
- *integrity*; while permitting maximal reconfigurability, we want to be able to control and constrain the scope of reconfigurations so that nonsensical and damaging changes are discouraged and/ or disallowed.

The rest of this paper is structured as follows. First, section 2 introduces the baseline technologies and frameworks that underpin OpenORB v2. These are *i*) our *component model*, which is an enhancement of the core of Microsoft's COM and forms the basis of our approach to delivering performance and standards conformance, and *ii*) an instantiation of the notion of *component frameworks* which facilitates the structuring of component compositions and, with the aid of an associated *reconfiguration management* approach, helps to constrain the scope of reconfigurations and ease the task of integrity maintenance. Following this, section 3 presents the architecture of OpenORB v2 in detail. First, we discuss the structure of the platform in terms of its constituent component frameworks, and then focus on the *binding* component framework as a detailed case study. Subsequently, section 4 evaluates the performance of our implementation, and section 5 discusses related work. Finally, we present our conclusions in section 6.

2. Baseline Technologies and Frameworks

2.1 OpenCOM

2.1.1 Background and Requirements

The first baseline technology underpinning OpenORB v2 is a language independent, hierarchically composable, component model called *OpenCOM*. This is a lightweight, efficient, design that is built atop a subset of Microsoft's COM. We chose COM as the basis of our component model for the following reasons: *i*) COM is standardised, well understood and widely-used, *ii*) it is inherently language independent, and *iii*) it is significantly more efficient than other component models such as JavaBeans or the CORBA Component Model.

In implementing OpenCOM, we ignore higher-level features of COM, such as distribution, persistence, security and transactions, and rely only on certain low-level 'core' aspects. This is because our approach, as mentioned above, is to implement higher-level features such as these in a middleware environment that is itself constructed from components. The core on which we implement OpenCOM consists of the following: *i*) the basic binary-level interoperability standard (i.e. the *vtable* data structure), *ii*) Microsoft's Interface Definition Language (IDL), *iii*) COM's globally unique identifiers (referred to as GUIDs), and *iv*) the *IUnknown* interface (for interface discovery and reference counting). A brief overview of COM, which explains these features, is given in Appendix A.

OpenCOM builds on this core subset of COM as follows:

- i*) it makes explicit the *dependencies* of each component on its environment (i.e., on other components); this is an essential requirement for run-time reconfiguration as it is not otherwise possible to determine the implications of removing or replacing a component [Kon,00b];

- ii) it adds basic mechanism-level functionality for reconfiguration, such as mutual exclusion locks to serialise modifications of inter-component connections; this provides a substratum on which higher-level reconfiguration strategies can be layered;
- iii) it adds support for pre- and post- *method call interception*, enabling us to inject monitoring code (e.g. to drive reconfiguration policies), and offering a lightweight means of adding new behaviours that do not require a reconfiguration of existing components (e.g. security checks on method calls).

Essentially, OpenCOM reinterprets, in an efficient and standards based environment, the reflective introspection and adaptation capabilities we have identified as useful in our earlier work [Blair,98].

2.1.2 OpenCOM Functionality

The fundamental concepts in OpenCOM are *interfaces*, *receptacles*¹ and *connections*. Whereas an interface expresses a unit of service *provision*, a receptacle expresses a unit of service *requirement* and is used to make explicit the dependency of one interface on another (and hence one component on another). For example, if a component requires an interface *S*, it would declare a receptacle of type *S* which would be *connected*² at run-time to an external interface instance of type *S* (which would be provided by some other component). Thus, as well as declaring interfaces in the usual way, components that depend on services offered by other components must additionally declare a suitable set of receptacles. In our current design, each component can only support a single receptacle of any given type. However, we also support so-called *multi-pointer receptacles* that can be connected to more than one interface instance (see Appendix B for more detail).

OpenCOM deploys a standard *run-time* that is available in every OpenCOM address space (it is implemented as a singleton COM component called “OpenCOM” and exports an interface called *IOpenCOM*). The primary role of the run-time is to manage a repository of available OpenCOM component types and thus support the creation and deletion of components; this builds on underlying COM facilities. In addition, the *IOpenCOM* interface serves as a central point for the submission of all requests to connect or disconnect receptacles and interfaces in its address space. Furthermore, to facilitate reconfiguration, the run-time records each creation or deletion of each component or connection in a per-address space meta-structure called the *system graph*. This enables the run-time to support queries (again, on the *IOpenCOM* interface) which, given a connection identifier (see *IMetaArchitecture* below), yield details of the receptacle and interface(s) participating in the given connection, together with details of their hosting components.

To conform to OpenCOM, each component must implement the following pair of *component management* interfaces for use by the run-time. These assist the latter in, respectively, creating and deleting connections and in creating and deleting components:

¹ The term ‘receptacle’ is also employed by the CORBA Component Model [OMG,99]. The concept itself appears in various other models under various names.

² Connections in OpenCOM are always between receptacles and interfaces that reside in the same address space. Again, this is because we view distribution as a middleware service to be built on top of, or in terms of, our lightweight component model.

- *IReceptacles* offers operations to alter the interface(s) currently associated with (i.e., connected to) each of the target component's receptacles. These operations are only ever called by the run-time's connection management operations. Before associating a new interface with a receptacle, the run-time obtains a per-receptacle *lock* to avoid conflicting with any pending invocations on the receptacle (see Appendix B).
- *ILifeCycle* offers *startup()* and *shutdown()* operations which are called by the run-time (but see Appendix B) when an instance of the target component has been created or is to be destroyed. This interface essentially fulfils the role of constructors and destructors in an object-oriented language (we cannot rely on the availability of such facilities in our language independent environment).

Furthermore, each OpenCOM component has embedded within itself three standard sub-components (called *MetaInterception*, *MetaArchitecture* and *MetaInterface*). These implement the reflective facilities identified in our previous work [Blair,98] and (respectively) export the following meta-interfaces:

- *IMetaInterception* enables the programmer to associate (dissociate) *interceptors* with (from) some particular interface. Interceptors are components that implement interfaces containing *interceptor methods*; these are invoked before or after (or both before and after) every method invocation on the specified interface. Multiple interceptors can be added and removed at run-time and reordered as desired. Again, see Appendix B for more detail.
- *IMetaArchitecture* enables the programmer to obtain the identifiers of all current connections between the target component's receptacles and external interfaces. These identifiers can then be submitted to the above-mentioned *IOpenCOM* interface which returns information on the receptacle/ interface(s)/ components involved in the connection.
- *IMetaInterface* supports inspection of the types of all interfaces and receptacles declared by the target component. Access to interfaces leverages the standard COM type library facility. Access to receptacles is via a separate mechanism (see Appendix B).

Fig. 1 visualises the component model. It shows an OpenCOM component (above) and the OpenCOM run-time (below). The component's management and meta-interfaces are shown on its left hand side. The three meta-interfaces are linked to the embedded sub-components that implement OpenCOM's reflective capability. Of these, *MetaArchitecture* and *MetaInterface* are further linked to corresponding private interfaces in the run-time, which is shown encapsulating the system graph and type libraries, and exporting the *IOpenCOM* interface.

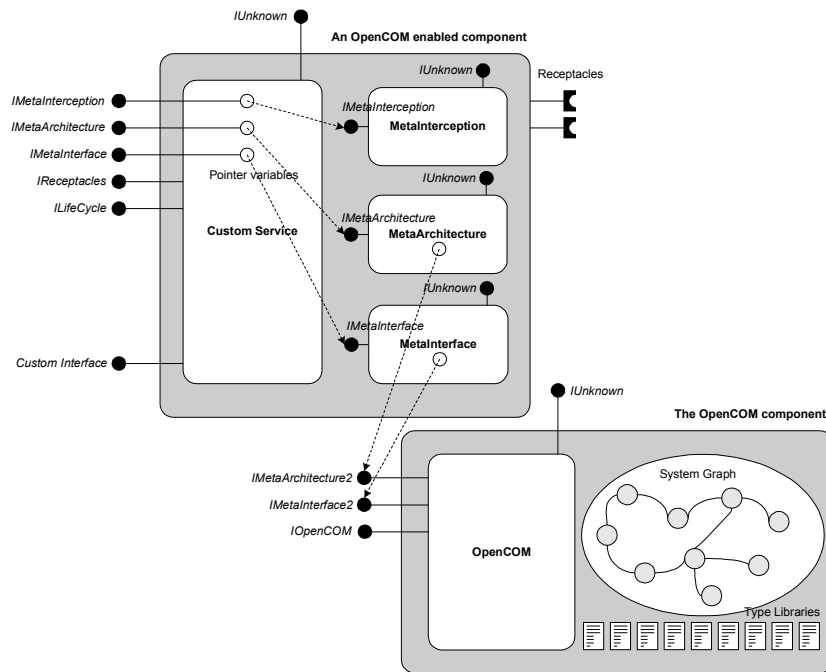


Figure 1: The Architecture of OpenCOM

Also associated with the illustrated component are a component specific interface (labeled ‘custom interface’) and two receptacles. Components can export any number of component specific interfaces and receptacles.

Further details on OpenCOM are given in Appendix B.

2.2 Frameworks and Patterns

2.2.1 Component Frameworks

The second key underpinning of OpenORB is a development of the concept of *component frameworks* (CFs) [Szyperski,98]. CFs were originally defined by Szyperski as “collections of rules and interfaces that govern the interaction of a set of components plugged into them”. CFs are targeted at a specific domain and embody “rules and interfaces” that make sense in that domain. For example, we employ a *protocol CF* that embodies knowledge, in the form of appropriate rules and interfaces, about the composition and reconfiguration of ‘plugged-in’ protocols. As discussed below, CFs are a main ingredient in our approach to integrity maintenance across reconfiguration operations.

In OpenORB, we develop the CF concept in *three* ways. First, CFs in OpenORB are not just a design concept. Rather, each CF is reified as a run-time software infrastructure (itself packaged as a component) that contains arbitrary CF-specific state, and supports and polices plug-ins to ensure that they conform to its rules. Second, for reasons of integrity management, and to reduce coupling and increase scaling, CFs disallow their plug-ins from directly depending on components outside the CF. Third, we *nest* CFs to gain the benefits of hierarchical composition. Thus, the top-level structure of OpenORB (see section 3.1) is itself a CF that constrains the interactions between, and reconfigurations of, its constituent CFs.

For us, the primary benefit of CFs is that they provide an environment of well defined architectural properties and invariants for their plug-ins. These properties

include both functional attributes and extra-functional quality attributes (e.g., modifiability or performance). Essentially, CFs provide a context that ensures control and predictability of such desired attributes. Moreover, CFs simplify component development and assembly through design reuse and guidance to developers. They also enable lightweight components (plug-ins), and increase the understandability and maintainability of systems.

2.2.2 Approach to Reconfiguration

The fundamental issues in effective reconfiguration management are: *i*) to *constrain the scope and effect* of reconfiguration, *ii*) to *separate concerns* between reconfiguration operations and core middleware functionality, and *iii*) to *maintain integrity* in the face of dynamic change. To address the first of these issues, we employ hierarchically-structured CFs as the natural scope for adaptation. We then address the second and third issues by applying a meta-level *manager/ managed* pattern within the resultant hierarchical scopes.

In this pattern, CFs adopt the *manager* role and their plug-ins adopt the *managed* role. At the same time, CFs are responsible for exposing themselves as *managed* entities with respect to higher-level CFs or other manager components (e.g., application components). In more detail, managers (e.g. CFs) maintain meta-information representing their current configuration of managed entities (e.g. plug-ins), monitor events emitted by their managed entities, and effect changes on the latter. These changes may involve the invocation of management operations on plug-ins, the setting of attributes, or modification of the configuration of plug-ins (i.e., adding/ deleting/ connecting/ disconnecting plug-ins using OpenCOM primitives).

As an example of the use of this pattern, consider our media streaming CF (see section 3.3). This CF accepts media filter plug-ins, maintains causally connected meta-level graph representations of its current configurations of media filters and offers a meta-interface for manipulation of the graphs. Internally, filter components provide a ‘managed’ interface to the CF that accepts management requests from the CF and emits management-oriented events to the CF. For example, a filter that cannot keep up with the data being sent to it generates a “flooding” event to which the CF may react by requesting an upstream filter to reduce its sample generation rate. If the CF does not understand an event, it forwards it through its own management meta-interface, thus allowing external components (e.g. in the application) to react appropriately (e.g., add a new filter to the graph). The CF also exploits its implicit domain-specific knowledge to manage requested reconfiguration operations with minimum perceived disruption of the media stream. For example, it can buffer data while reconfigurations are taking place.

In general, an important benefit of CF-mediated reconfiguration management is that CFs can exploit domain-specific knowledge and built-in constraints to enforce a desired level of integrity across reconfiguration operations. Furthermore, the desired level of integrity and consistency can be suitably traded-off against the degree of afforded flexibility. 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, CFs typically defer the removal of an instance until it contains no active threads. In addition, CFs often provide operations to facilitate the transfer of run-time state from a replacee plug-in to its replacement.

Finally, while we expect CF-based adaptation to serve most adaptation needs, it is also possible in OpenORB to bypass the CF structure and perform ad-hoc unanticipated adaptations directly at the OpenCOM level if and when the need arises. While this is rather dangerous in terms of integrity maintenance, we have successfully used this approach in the past to support the dynamic injection of QoS managers into a system at run-time [Blair00a], [Blair00b].

3. The OpenORB Architecture

3.1 Overview

OpenORB is structured as a top-level CF that is itself composed of three layers of further CFs. The three layers are called *binding*, *communication* and *resource*; see below for more detail. Each component/CF in a given layer is only allowed to access interfaces offered by components/CFs in the same or lower layers. The motivation for this restriction is to clarify the architecture and offer guidance to CF developers for the design and placement of new CFs. The top level CF manages the lifecycle of the hosted CFs and resolves their dependencies. Furthermore, the top-level CF imposes *layer-composition* policies concerning dynamic changes in layer composition, and supports a *resources meta-model* (again, see below). The second level CFs address more focused sub-domains of middleware functionality (e.g., binding establishment and thread management) and enforce appropriate sub-domain specific policies.

This hierarchical structure opens up two distinct dimensions of flexibility. First, the top level CF can be configured by mandating an appropriate layer-composition policy plus the set of CFs that will initially populate the layers. 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 (see Fig. 2) employs a simple layer-composition policy that enforces QoS properties by forbidding the removal of certain key CFs, and consists of the 5 CFs shown in Fig. 2. However, this is 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, etc.

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 top-level CF) and replacing/ customising/ extending the existing second-level CFs (both statically and dynamically). For example, in the current OpenORB architecture, a signal processing CF could be added in the communications layer, or the thread management CF could be dynamically customised with a new scheduler component.

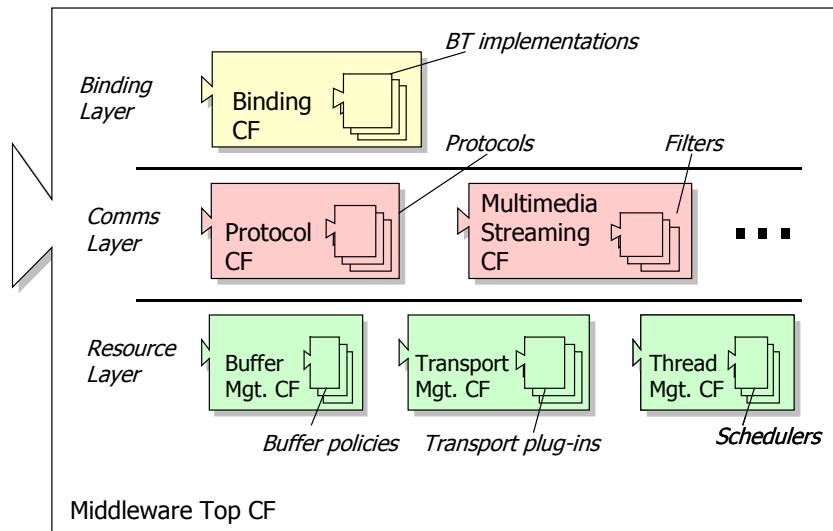


Figure 2: The Architecture of OpenORB

Our currently implemented middleware architecture is structured as follows: First, the *resource layer* contains *buffer*, *transport*, and *thread management* CFs which respectively manage buffer allocation policies, transport protocols and thread schedulers. Next, the *communication layer* contains *protocol* and *multimedia streaming* CFs. The former accepts plug-in *protocol* components and the latter accepts *media filter* components. Finally, the *binding layer* contains a *binding CF* that accepts *binding type implementations* (e.g., remote object invocation, streaming connections or publish/subscribe etc.). The binding layer is a crucial part of the architecture because it determines the programming model offered to middleware users.

3.2 The Role of the Top-Level CF

The top-level CF helps in decoupling the design of second-level CFs from the way in which these are dynamically combined in a middleware architecture. At design time, each second-level CF declares its static dependencies on abstract “services” that will presumably be offered by other CFs (a *service* in this context is a uniquely identified group of related interfaces—which may include factory interfaces). Then, at CF insertion time, instances of these required services are obtained via an operation of the form *GetService(serviceId, iid, ppUnknown)* that is supported by the top-level CF. If some dependencies cannot be resolved using the current layer composition, the top-level CF is able to dynamically load new CFs using a persistent repository service that maps *serviceIds* to CFs. Note that *GetService()* is only involved at insertion time and therefore incurs no overhead when CFs actually interact.

Similarly, second-level CFs resolve the dependencies of their own plug-ins either using functionality implemented directly within the host CF itself or using services that were obtained at CF-insertion time. In addition, to cover situations in which a dynamically loaded plug-in may have requirements unanticipated by its host CF, the host CF can delegate unknown service requests to the top-level CF.

Adaptation of the top-level CF is controlled by *layer composition policies*. These are realised as components which either allow or veto proposed layer composition changes (i.e. insertions or removals of CFs) on the basis of the current layer composition and on meta-data packaged with the to-be-inserted CF, which includes

the specification of provided/ required services and required resources (e.g., memory and thread requirements).

The following sections present our architecture in more detail. The resources and communications layers are covered in section 3.3, and the binding layer is discussed in detail in section 3.4.

3.3 The Resources and Communications Layers

Our implementation of these two layers consists of about 50,000 lines of C++ (including the OpenCOM run-time) divided into thirty components and five CFs. The bulk of the code is derived from GOPI, a CORBA compliant, multimedia capable, middleware platform that we have developed previously [Coulson,99a].

Each CF in the resource and communication layers follows a similar pattern: it defines an abstract interface and manages different implementations of this interface (which are plugged in as separate components). The CFs employ a *multi-pointer-with-context* receptacle (see Appendix B) to select alternative plug-ins at run-time. In addition, the CFs all offer meta-interfaces for domain specific dynamic reconfiguration. For example, there are operations to ensure that managed components can be dynamically loaded and unloaded without disruption to currently executing applications. All our currently implemented CFs are thread safe and reentrant.

The *thread management* CF multiplexes user-level threads over kernel threads (referred to as *virtual processors*), and supports the dynamic selection of plug-in *scheduler context* components, each of which manages its own user-level threads and dedicated virtual processors, and imposes its own scheduling policy and concurrency semantics [Coulson,01a]. The thread CF's dynamic reconfiguration interface enables the dynamic loading and unloading of these scheduler contexts. Currently there are three scheduler context implementations: one with a simple priority based policy, another with an earliest deadline first policy, and a third that simply maps every user level thread to its own dedicated kernel thread.

Also in the resource layer, the *buffer management* CF defines an abstract interface that enables developers to write tailored buffer allocation policies using a common buffer abstraction. Currently there are two plug-ins: a *malloc()* based implementation that maps directly to OS level memory management routines, and a more efficient 'buddy' allocation scheme [Knuth,73]. Similarly, the *transport* CF defines an abstract interface that enables developers to add transport protocols. Currently, we support TCP, UDP, multicast IP, and IPC (i.e. pipes on UNIX and memory mapped files on Win32). In addition, the transport CF supports another type of plug-in that supports alternative *message detection strategies* for incoming messages. The default strategy only looks for new messages when all threads in the address space are currently blocked. Other strategies rely on various combinations of server threads, thread pools and signal driven I/O notification. Further details are available in [Coulson,99a] and [Coulson,01b].

All the resource CFs support a common meta-level framework called the *resources meta-model* [Blair,99] that is implemented by the top-level CF. This supports a *task* abstraction that relates to some logical activity carried out by some set of cooperating objects. For example, the activity of incoming message handling can be designated a 'task'; this is performed by distinct *detector*, *transport* and *GIOP protocol* objects [Blair,99]. At the meta-programming level (accessible through

reflection) a task appears as a component that encapsulates a pool of resources and resource factories (e.g., buffer pools, threads, scheduler contexts, transport service access points), that are dedicated to the set of base-level components participating in the task. When a component requests resources, these are implicitly allocated from its associated task.

Using the resources meta-model, the resources in a task can be inspected via a hierarchical data structure, rooted at the pool of resources referred to above, of alternately layered *resources* and *resource managers*. This meta-level data structure links the alternating layers with *managed-by* and *builds-on* relationships: resources are *managed by* managers, and managers *build on* lower level resources. For example, threads are *managed by* scheduler contexts which *build on* virtual processors which are *managed by* the OS scheduler which *builds on* physical CPUs¹. This is a generic pattern and similarly supports all other resource types. Meta-interfaces are available which support both inspection and adaptation of both resources and their managers at any level of the hierarchy. For example, the scheduling policy of a scheduler context can be modified or replaced. In addition, resources can be moved from one resource manager to another. For example, one can change the way a thread is scheduled and/or resourced by moving it from one scheduler context to another. Similarly, one can switch an existing connection to use an alternative transport protocol simply by moving its transport service access point from one transport resource manager to another. More details of the resources meta-model are available in [Blair,99].

Finally, in the communications layer, the *protocol* and *multimedia streaming* CFs define a plug-in environment for stacked communication protocols and composable media processing filters respectively. Currently, four protocol implementations exist: a CORBA GIOP v1.2 object request protocol, a cut-down version of GIOP that can also be used for media streaming, a protocol that efficiently passes data between end-points in the same address space, and a protocol that employs shared memory for communication and the IPC transport for synchronisation. These CFs also maintain information about the current protocol/filter configurations (organised as a graph of instances) and offer meta-interfaces with specialised operations to reconfigure the graph in such a way that rules constraining permissible configurations are obeyed. The design of the protocol CF is guided by currently known best practice in avoiding multiplexing and other overheads as discussed in [Coulson,01b].

3.4 The Binding Framework

3.4.1 Overview

One of the most significant limitations of present-day middleware platforms, both industrial and research, is that they support only a small, pre-defined, set of fundamental *binding types* (e.g., CORBA only supports remote method invocation, media stream management and event handling). This limits the applicability of the platform because richer or more specialised forms of interaction (e.g. pipes, groups, SQL links, auction services, voting protocols, etc.) cannot easily be accommodated except as ad-hoc add-ons. In addition, the binding types that are provided tend to be isolated in their implementation and rely on ad-hoc and distinct infrastructures. For example, CORBA events and media stream management are completely distinct.

¹ Of course, the depth to which this particular resource hierarchy can be extended is highly dependent on the underlying OS environment.

The goal of the binding CF is to accommodate an *extensible* set of binding types (known as *BTs*), as plug-ins, and to provide a consistent, component-oriented programming model for their definition and use¹. By capturing diverse forms of interaction as *BTs*, the binding CF aims to significantly simplify application development and promote the reuse of interaction mechanisms over multiple applications.

Abstractly, *BTs* are defined in the following terms:

- i) a *binding establishment process*; that is, the pattern of collaboration that is required between the various parties involved (i.e. participants and *BT* implementation-provided objects) to create and initialise a binding;
- ii) a set of *interaction patterns*; that is, a set of *participant roles* together with a specification of the ways in which these roles are permitted to interact (a role is essentially an abstraction of the observable behaviour of an object and is specified as a set of interfaces and associated rules and constraints);
- iii) a *binding control process*; that is, how established bindings are managed (e.g., monitoring and adaptation interfaces, QoS control, adding and removing participants).

BT specifications are expressed mainly in UML. The parameterised collaboration construct in UML is particularly suited for capturing most of the structural and behavioural aspects of *BTs*. Other aspects are expressed using a variety of modeling tools including statechart diagrams, OCL constraints, tagged values and free text.

As illustrated in Fig. 3, the binding CF distinguishes two separate ‘contracts’ (i.e. sets of interfaces and rules).

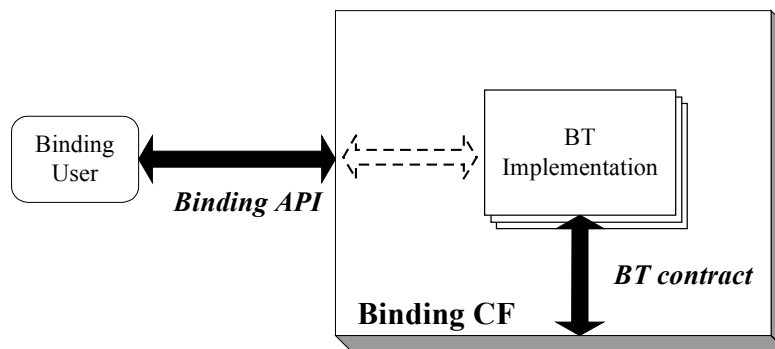


Figure 3: The Two Binding CF Contracts

These are *i*) the *binding API*, which defines the view of *BTs* seen by binding users, and *ii*) the *BT contract* which is the contract between *BT* plug-ins and the binding CF itself. We discuss these two contracts in more detail in the following two sub-sections.

3.4.2 The Binding API

The binding API defines the interactions that support *binding establishment*, *iref resolution/generation* and *binding control*. It is based on a generic ‘binding model’

¹ Note that *BT* implementations are typically deployed over more than one address space in order to support the creation of bindings, which are inherently distributed entities.

(see below) and a small number of generic interfaces that are designed to capture commonalities across BTs. The APIs of the various BTs conform to the binding API and extend it with their own specifics. BTs are themselves identified by 128-bit GUIDs which are mapped to BT implementations (deployable components) by the CF. This separation enables BT implementations to be replaced without affecting binding users.

In the generic binding model referred to above, bindings are established between *participants* and the responsibility for binding establishment is assigned to *binders*. Binders take as input a number of objects representing participants, together with related information such as QoS specifications. They then verify that the supplied participant objects conform to the roles specified in the associated BT specification (typically by using reflection), invoke appropriate operations on them, and establish the binding (with the aid of services offered by the communications and resources layers). If binding establishment succeeds, the binder returns a *bindingCtl* ('binding control') object, which represents the established binding.

Participants that are remote with respect to a binder's location are represented by *rep* objects ('remote participant representatives'). The process of creating a rep falls into two stages: First, a *generator* is used at the participant's (remote) site to generate both an *iref* and an associated communications infrastructure. An iref is an endpoint value that can be passed around the distributed system. Second, the iref is transferred to the binder's site (by whatever means) where it is passed to a *resolver* that is responsible for creating a corresponding rep. This whole process is referred to as *participant remoting*.

A very common special case, which is specially accommodated by the framework, involves first-party bindings that are initiated by an anonymous participant (e.g. standard CORBA bindings). In this case, the binding initiator is not explicitly represented by an object. Instead, the binder returns a so-called *apu* ('anonymous participant use') object on binding establishment that represents the participation of the anonymous binding initiator.

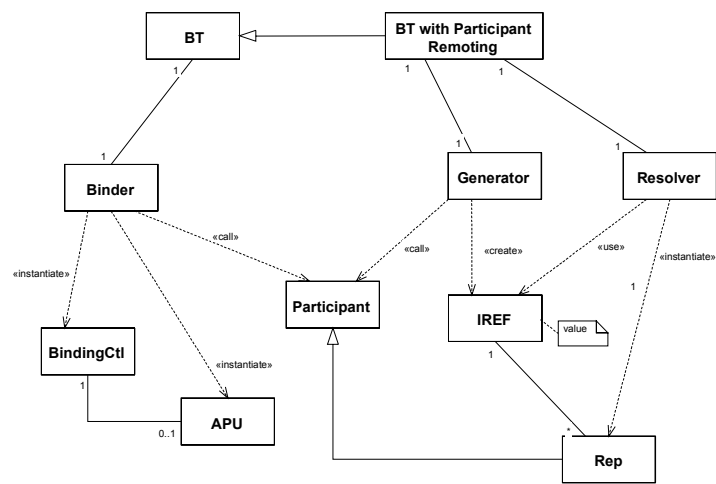


Figure 4: Binding API entities.

The basic entities comprehended by the generic binding model (i.e., *binder*, *generator*, *iref*, *resolver*, *rep*, *bindingCtl* and *apu*), together with their relationships, are shown in Fig. 4. Note that all these entities are essentially *roles* and that individual

objects can fulfill more than one role at a time (e.g., a resolver can also serve as a binder). An important implication of this is that a single object can take part in many different bindings of different types.

The binding API provides generic COM interfaces for the basic entities. For example, the generic interface for binders (*IBinderGeneric*) has an operation that takes as arguments a set of participant objects (represented as *IUnknown* pointers) and a generic QoS specification (represented as a string), and returns bindingCtl and apu objects (*IBindingCtl* and *IAPU* pointers). The verification that a participant conforms to the expected BT-defined role is accomplished dynamically using COM's interface negotiation mechanism (*IUnknown::QueryInterface()*) and/or the extended meta-information provided by OpenCOM.

3.4.3 The BT contract

The BT contract defines interactions that enable the binding CF to *i*) expose BT implementations to binding users, *ii*) manage BT implementations, and *iii*) offer services to BT implementations.

First, in terms of *exposing* BT implementations to users, the binding CF serves as an initial access point for BT entities (e.g. binders, resolvers and generators), and maps requested BT identifiers to BT implementation components by means of a configuration database. BT implementations are responsible for providing both the generic and BT-specific interfaces as described in the associated BT specification. The set of available BTs can either be fixed and configured statically (at platform build-time) or be dynamically extensible. A typical scenario is that a BT implementation is loaded and activated on demand when an iref associated with the specific BT arrives at an address space. However, the binding CF's policy with regard to automatic extension can be modified through a specific meta-interface.

Second, in terms of *managing* BT implementations, the binding CF deals with instantiation, initialisation and destruction of individual BT implementations. It also manages dependencies between BT implementations by maintaining information on all active BT implementations and their relationships. In general, BT implementations have two kinds of dependencies:

- dependencies on services offered by the lower platform layers (i.e., services in the communications and resource layers);
- dependencies on other BTs (e.g., a BT implementation that realises a distributed auction protocol may rely on a remote method invocation BT).

Note that the latter kind of dependency implies a compositional model of implementing BTs which helps simplify development through reuse.

Third, in terms of *offering services* to BT implementations, the binding CF provides access to services in the communications and resource layers (the contents of which are determined at middleware architecture definition time as explained in section 3.1). This is achieved by delegating service requests to the top-level CF, which incorporates knowledge about the current middleware architecture as seen in section 3.2. In addition, the binding CF provides two 'standard' (but replaceable) BT components. First, the *StdResolver* component is a resolver that maps arbitrary irefs onto an appropriate BT specific resolver. It simply extracts the BT identifier from the (universal) iref format and invokes the binding CF to retrieve the associated resolver.

Second, the *StdGenerator* component is a generator that marshals object references by retrieving an associated BT identifier and invoking its native generator. This indirection allows objects to nominate a preferred BT for their marshaling (similarly to DCOM custom marshaling [Microsoft,99]). Built-in support for marshaling object references is necessary because the component model presumes fully general object-oriented interfaces (i.e., object references may be passed as arguments over established bindings).

Note finally that providing dynamic reconfigurability of established bindings is *not* a concern of the binding CF. Rather, it is the responsibility of individual BT implementations. In general, reconfiguration of established bindings is achieved by building both on interfaces provided by other BTs (recursively) and on meta-interfaces offered by the lower platform layers. In terms of the manager/ managed pattern (section 2.2.2), a binding control object plays the manager role with respect to lower-level bindings and CFs in the communications and resource layers. At the same time, it exposes a “managed” interface (i.e., *bindingCtl*) to applications or higher-level bindings. Clearly, since bindings are distributed, managing their reconfiguration requires inter-process coordination. This is realised in an entirely BT implementation specific way. For example, one of our BTs realises distributed video bindings, which employ filters and protocols deployed over different processes. The associated binding control object coordinates binding reconfiguration using control messages propagated by an embedded remote method invocation binding.

3.4.4 Examples

To demonstrate its generality and extensibility, we briefly sketch how some example BT implementations can be accommodated by the binding CF.

First, a BT for simple remote method invocation is straightforwardly implemented as follows. The BT defines two participant roles: *server* (which accepts remote method invocations) and *client* (an anonymous participant). The BT’s resolver also functions as a binder: It establishes the binding on the basis of a given iref and returns an apu object (cf. a traditional proxy). In a more sophisticated variation, the resolver could create reps that need to be explicitly bound using special binder interfaces (e.g., in order to specify QoS or select particular protocol stacks).

Second, a BT that models a loosely-coupled publish/subscribe interaction style can be achieved by defining three participant roles: *event channel* (which is associated with an *event* interface type on which events are received), *subscriber* (which implements the event interface), and *publisher* (which emits events by invoking the event interface). The event channel is represented as an iref and rep(s), and there are separate binders supporting the binding of publishers and subscribers respectively to the channel.

Finally, a BT for client/ server based group communication might define three participant roles: *group* (a logical participant that provides a *group interface*), *group member* (which receives multicast invocations on the group interface and also implements a standard interface for receiving view change notification events and supporting state transfer), and *client* (an anonymous participant that sends invocations to the group interface). The BT’s generator receives as input the group interface type and creates an iref representing the group; when such an iref enters an address space, the resolver creates a corresponding rep object. The binding establishment process is separated into two parts: *i*) binding group members to the group, and *ii*) binding group

clients to the group. Each of these parts employs a separate binder as follows. The member-to-group binder takes as input an object conforming to the *group member* role whereas the client-to-group binder returns an apu implementing the group interface, which is then used to perform multicast invocations to the group. Assuming that a binding for group clients can be achieved implicitly, the rep can also implement the group interface and be used “as-is” for invoking the group (i.e., it can play an apu role too).

More detail on the Binding CF can be found in [Parlavantzas,01].

4. Performance Evaluation

4.1 Scope of Evaluation

In this section, we evaluate the performance of OpenCOM, and also investigate the overhead of its deployment within OpenORB v2 by comparing the overall performance of the latter with that of other standard ORBs.

In evaluating the performance of OpenCOM we are primarily interested in its overheads *on the call path*, i.e. the overhead that OpenCOM introduces in a platform’s normal operational mode. Specifically, we have not evaluated the overhead of non-functional OpenCOM characteristics (i.e. reflection and reconfiguration) because the use of these is relatively infrequent.

All tests in this section were performed on a Dell Precision 410MT workstation equipped with 256Mb RAM and an Intel Pentium III processor rated at 550Mhz. The operating system used was Microsoft’s Windows2000 and the compiler was Microsoft’s cl.exe version 12.00.8804 with flags /MD /W3 /GX /FD /O2.

4.2 Performance of OpenCOM

The main OpenCOM mechanisms that affect call path performance are receptacle based invocation and intercepted invocation¹ (i.e. the use of *IMetaInterception*). Fig. 5 presents the raw performance of these mechanisms in terms of calls/sec throughput on a method with a NULL body. In the figure, we show comparisons against C language invocations and COM invocations. In addition, we provide figures for the various possible combinations of locking/ non-locking receptacles and interception in OpenCOM.

The figures demonstrate a marked difference in the overhead of simple method calls compared to the complex interactions embodied by OpenCOM based invocations. This, of course, is to be expected due to the widely varying underlying mechanisms involved. For example, a C language invocation simply loads an immediate register and executes a machine level CALL through that register. On the other hand, a COM invocation (i.e. a C++ virtual method call) must traverse *lpVtbl* and *vtable* pointers (through memory accesses) before performing the CALL. Beyond this, a non-locking receptacle based invocation must execute the code for the overridden de-reference operator (to access the receptacle’s interface pointer) before performing a virtual method call on that pointer. As expected, there is slightly more overhead involved in locking than interception (see the intercepted COM and locking OpenCOM figures) despite both using similar techniques. This is because locking

¹ The interceptors used in these tests had a single pre and post method attached, each with a NULL body.

requires synchronisation to protect its lock variable. We use a Win32 CRITICAL_SECTION object to minimise this overhead as this spins at user level for a pre-determined time before blocking in the kernel.

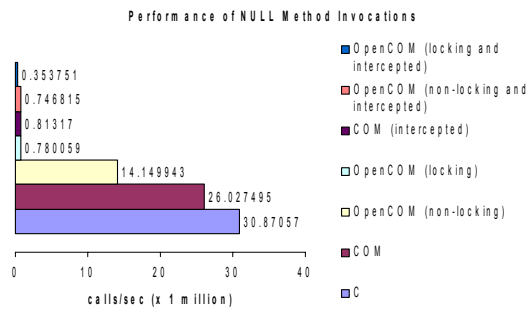


Figure 5: Performance Comparison of NULL Method Invocations

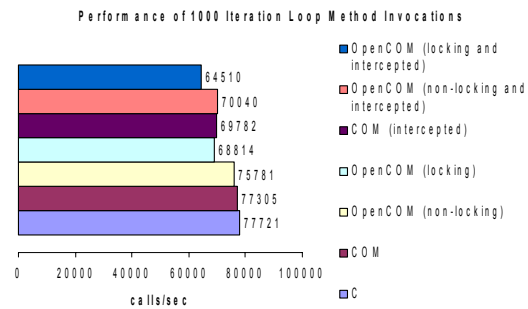


Figure 6: Performance Comparison of Complex Method Invocations

Although the difference in raw call throughput between COM and OpenCOM is not inconsiderable (especially when using locking and intercepted invocations) it is not necessarily significant in the context of a real system. This is because it is usual for the time taken to invoke a method to be far smaller than the time taken to actually execute the method's body. Fig. 6 demonstrates the effect of replacing the empty method body used in Fig. 5 with a slightly 'busier' method body (one that implements a 1000 iteration empty loop). The figures here clearly demonstrate that as the complexity of the method itself grows, the overhead of its invocation becomes less significant and the various invocation techniques begin to converge in terms of call throughput.

4.3 Performance of OpenORB v2

Our hope and expectation is that OpenORB should perform as well as existing ORBs while simultaneously providing dynamic reconfigurability through componentisation, reflection, and CFs. To evaluate this expectation we compared the performance of OpenORB¹ with that of two other ORBs: GOPI v1.2 and Orbacus 3.3.4. As stated, GOPI provides much of the source code for OpenORB v2 but is written in C and implemented in a single library. A direct performance comparison with GOPI should therefore directly yield insight into the overhead of our component model. Orbacus is well known as one of the fastest and most mature CORBA-compliant commercial ORBs available.

The OpenORB middleware architecture used in the tests was that shown in Fig. 2 in conjunction with a BT implementation that provides remote method invocation on top of CORBA GIOP v1.2. Our tests measured method invocations per second over the Dell's loopback interface. A minimal IDL interface was employed that supported a single operation that took as its argument an array of octets and returned a different array of the same size. The implementation of this method at the server side was null.

¹ Receptacle locking was not enabled in these tests as no equivalent overhead is present in the comparison ORBs.

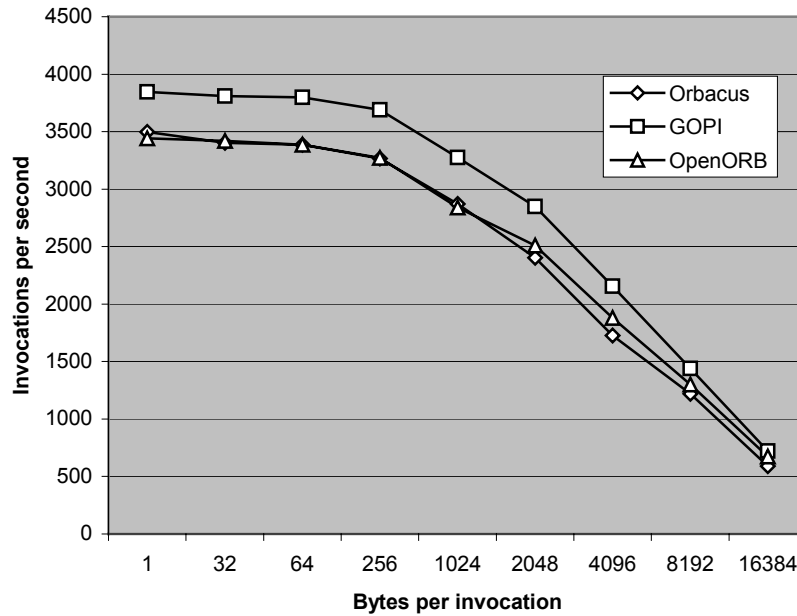


Figure 7: Performance of OpenORB versus GOPI and Orbacus

The results of timing a large number of round-trip invocations using this setup are shown in Fig. 7. It can be seen that for packets of less than 1024 octets, OpenORB performs about the same as Orbacus, with GOPI running around 10% faster. As might be expected, there is a diminishing difference between all three systems as packet size increases; this is presumably due to the fact that the overhead of data copying begins to outweigh the cost of call processing.

The relative overhead of OpenORB compared to GOPI can be attributed to the former's increased use of indirection (i.e. through receptacles and *vtables*). In our test configuration the data path for each GIOP invocation involved 67 of these receptacle crossings, 32 on the client side and 35 on the server side. Despite this additional work, it can be seen that the performance of OpenORB is entirely comparable to that of the non-componentised ORBs.

5. Related Work

COM+ [Microsoft,00], Enterprise JavaBeans [Sun,00] and the CORBA Component Model [OMG,99] all support a similar *declarative* programming model for building distributed applications. These architectures support a separation between functional aspects (implemented by applications) and non-functional aspects (managed by a 'container'). The limitation of such models is that the configurability of non-functional aspects is severely limited since the container implementation remains a black box. Our platform offers a *procedural* rather than declarative programming model in the sense that application components explicitly access the middleware through API calls. However, it would be a fairly straightforward task to implement a declarative, container-type model *on top* of our middleware platform as another CF accepting application components as plug-ins. This would entail limiting the degrees of freedom offered by the platform in order to increase the usability for developers. For example, a fixed number of binding types could be standardised and component-level meta-data could be used to declaratively select between them.

CORBA services such as the event service [OMG,00b] or the audio/video streams service [OMG,00c] represent an attempt to provide different binding types. However, this approach has major disadvantages with respect to our CF based approach. New “binding types” are implemented in an ad-hoc way by exploiting non-portable lower-level platform interfaces, and there is no coordination between the implementations of services (which is important for reasons of QoS management). Furthermore, services are not configurable or replaceable and their dependencies are not visible. Moreover, the programming models of the different services have little in common, which increases the cognitive load on middleware programmers.

Turning now to results from the research community, Jonathan [Dumant,98] is a Java-based ORB that, like OpenORB, employs an extensible binding framework that supports arbitrary binding types. Jonathan relies on a distinction between distributed interface references, managed by the ORB kernel, and the various ways to access and interact with these, which are managed by binding factories. FlexiNet [Hayton,98], FlexiBind [Hanssen,99], TAO [Schmidt,99] and Quarterware [Singhai,98] are other flexible ORBs, which focus only on remote method invocation bindings. Importantly, all these previous platforms are built in terms of object-oriented frameworks rather than CFs. Our component-based approach has important advantages over object-oriented frameworks. In particular, CFs are not bound to a specific programming language, and there is no implementation inheritance between components and the framework. As a result, components and CFs can be distributed in binary form, be independently developed, be combined at run-time, and evolve independently from each other.

OpenCorba is an open, reconfigurable ORB that depends on a reflective language (NeoClasstalk) [Ledoux,99]. 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,00a] and LegORB [Roman,00] are component based reflective ORBs that rely on a set of ‘configurators’ to manage dependencies among components and provide a set of ‘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.

.Net [Microsoft,01], the new component model from Microsoft, provides a remoting framework with numerous extensibility points, such as pluggable channels and formatters, message filters and custom properties. However, the remoting framework is an inseparable part of the component model (included in the .Net run-time) and cannot evolve independently from it. Furthermore, the framework relies

heavily on implementation inheritance from system-provided classes, which gives rise to the well-known fragile base class problem [Szyperki,98].

Finally, it should be noted that the issue of consistent dynamic reconfiguration is also being addressed within the software architecture and configurable distributed systems communities (see, e.g., [Oreizy,98] for the former and [Purtilo,98] for the latter). Much work in these areas has concentrated on devising reconfiguration models and algorithms that preserve a well-defined consistency constraint while minimizing system disruption [Kramer,90], [Bidan,98]. [Oreizy,98] proposes an architecture-based approach to reconfiguration which is related to our approach, but this is restricted to a specific architectural style which favours dynamic changes. Our work is complementary to this work, since we can accommodate such algorithms and styles within specific CFs depending on domain-specific needs.

6. Conclusions

We believe that the combination of a reflective component model and the CF-based structuring principle is a highly promising basis for the construction of configurable and reconfigurable ORBs. While the reflective component model inherently supports maximal flexibility and reconfigurability, on its own it is too powerful, and its unconstrained use can easily lead to chaos. The presence of CF-based structuring tempers this power by imposing domain specific constraints and semantics on the reconfiguration process.

We also believe that the three-layer structure of our top-level CF represents a generic and ‘future-proof’ basis for the future expansion, extension and evolution of OpenORB v2. We are encouraged in this belief by the fact that we have been able to easily accommodate several components and CFs that were not initially envisaged. For example, we have recently accommodated thread pool and OS abstraction components in the resources layer and plan the introduction of ‘signal processing’ and group communications CFs in the communications layer. As for the binding CF, we have not yet encountered a binding style that it could not accommodate, although we have less implementation experience in this area than in the other layers.

Our implementation efforts have also validated other aspects of the design. For example, we have discovered that the component model and our current set of CFs support the construction of ORB functionality that is at least as efficient as conventional object-based ORBs (see section 4). Furthermore, we have confirmed that the component model scales well in terms of its explicit enumeration of per-component dependencies. This is primarily due to the use of CFs which reduce dependencies by forbidding connections between plug-in components and components outside the CF. In our current implementation, the maximum number of dependencies in any single component is just seven and the average figure is just four. This leaves considerable scope for further reducing the granularity of componentisation which, if carried out with care, should correspondingly increase the ORB’s potential for reconfigurability.

Presently, our efforts are focusing on the further development of the OpenORB environment. For example, we are adding an additional plug-in component type to the protocol CF which will enable the configuration of a range of demultiplexing strategies (these strategies have previously been implemented in the GOPI platform [Coulson,01b] which, however, possesses limited scope for run-time reconfiguration).

We are also proposing, as mentioned above, to add communications layer CFs for group communications [Saikoski,00], and for software signal processing [Goel,98].

In the future, we plan to further exploit the reconfiguration management pattern outlined in section 3.2 and experiment with a wider range of reconfiguration scenarios. We also plan to investigate the integration of our work with other component models. For example, one possible direction would be to implement our component model on top of .Net or even Java instead of COM. This should be an easily achievable goal as the requirements of our component model are minimal. Such an implementation would increase the platform independence and/or applicability of our environment at the probable expense of performance. Another interesting direction would be to build a container-type CF similar to Enterprise JavaBeans or the CORBA Component Model on top of an extended version of our architecture.

References

- [**Bidan, 98**] Bidan, C., Issarny, V., Saridakis, T., and Zarras, A., “A Dynamic Reconfiguration Service for CORBA”. In Proceedings of the 4th International Conference on Configurable Distributed Systems, Annapolis, Maryland, USA, May 1998.
- [**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.
- [**Blair00a**] Blair, G.S., Andersen, A., Blair, L., Coulson, G., Sánchez, D., “Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform”, IEE Proceedings on Software Engineering, Vol. 147, No. 1, pp 13-21, February 2000.
- [**Blair00b**] Blair, G.S., Blair, L., Issarny, V., Tuma, P., Zarras, A., “The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms”, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), IBM Palisades, New York, April 2000.
- [**Brown,99**] Brown, K., “Building a Lightweight COM Interception Framework Part 1: The Universal Delegator”, Microsoft Systems Journal, January 1999.
- [**Costa,00**] Costa, F., Duran, H., Parlavantzas, N., Saikoski, K., Blair, G.S., and Coulson, G., “The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications”. In Walter Cazzola, Robert J. Stroud and Francesco Tisato, editors, Reflection and Software Engineering, Lecture Notes in Computer Science 1826. Springer-Verlag, 2000.

- [**Coulson,01a**] Coulson, G., and Moonian, O., “A Quality of Service Configurable Concurrency Framework for Object Based Middleware”, Concurrency Practice and Experience (to appear), 2001.
- [**Coulson,01b**] Coulson, G., and Baichoo, S., “Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment”, Distributed Computing Journal, Vol. 14, No. 2, April 2001.
- [**Coulson,98**] Coulson, G and Clarke, M.W., A Distributed Object Platform Infrastructure for Multimedia Applications, Computer Communications, Vol 21, No 9, pp 802-818, July 1998.
- [**Coulson,99a**] Coulson, G., “A Configurable Multimedia Middleware Platform”, IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.
- [**Coulson,99b**] Coulson, G., and Baichoo, S., “A Distributed Object Platform for Multimedia Applications”, Proc. IEEE Multimedia Systems, Florence, Italy, ISBN 0-7695-0253-9, pp 122-126, June 1999.
- [**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.
- [**Goel,98**] Goel, A., Steere, D., Pu, C., Walpole, J., “SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit”, OGI CSE Technical Report [98-009](#), poster presented at 2nd Usenix Windows NT Symposium September 1998.
- [**Gokhale,98**] Gokhale, A. and Schmidt, D.C., “Principles for Optimising CORBA Internet Inter-ORB Protocol Performance”, Proc. HICSS '98, Hawaii, Jan 9th 1998, <http://www.cs.wustl.edu/~schmidt/HICSS-97.ps.gz>.
- [**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.
- [**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.
- [**Kiczales,91**] Kiczales, G., des Rivières, J., and Bobrow, D.G., “The Art of the Metaobject Protocol”, MIT Press, 1991.
- [**Knuth,73**] Knuth, D.E., “The Art of Computer Programming, Volume 1: Fundamental Algorithms”, Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.
- [**Kon,00a**] 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.

- [**Kon,00b**] Kon, F and Campbell, R., "Dependence Management in component-Based Distributed Systems", IEEE Concurrency, pp 1-11, January 2000.
- [**Kramer,90**] Kramer, J., and Magee, J., "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Trans. Software Engineering, Vol. 16, Num. 11, pp. 1293-1306, November 1990.
- [**Ledoux,99**] Ledoux, T., "OpenCorba: a Reflective Open Broker," Reflection'99, Saint-Malo, France, Springer-Verlag, LNCS, Vol. 1616, 1999.
- [**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.
- [**Microsoft,00**] Microsoft, COM Home Page, <http://www.microsoft.com/com/default.asp>; Last updated: 01/06/2000.
- [**Microsoft,01**] Microsoft, .Net Home Page, <http://www.microsoft.com/net>; Last updated: 01/02/2001.
- [**Microsoft,99**] Microsoft DCOM web page: <http://www.microsoft.com/com/tech/DCOM.asp>; Last updated: 30/03/98.
- [**OMG,00a**] The Common Object Request Broker: Architecture and Specification, available at <http://www.omg.org/>.
- [**OMG,00b**] Object Management Group, Event Service v1.0, OMG Document formal/2000-06-15.
- [**OMG,00c**] Object Management Group, Audio/Video Streams, v1.0, OMG Document formal/2000-01-03.
- [**OMG,99**] Object Management Group, CORBA Components Final Submission, OMG Document orbos/99-02-05.
- [**Oreizy, 98**] Oreizy, P., Medvidovic, N., and Taylor, R., "Architecture-based runtime software evolution". In Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 1998.
- [**Parlavantzas,00**] Parlavantzas, N., Coulson, G., Clarke, M., and Blair, G.S., "Towards a Reflective Component Based Middleware Architecture", in Workshop on Reflection and Meta-level Architectures, June 13, 2000, Sophia Antipolis and Cannes, France.
- [**Parlavantzas,01**] Parlavantzas, N., "Design of a Binding Component Framework", Lancaster University Technical Report, MPG-01-03.
- [**Purtilo,98**] Purtilo, J., Cole, R., and Schlichting, R., editors, Proceedings of the Fourth International Conference on Configurable Distributed Systems (ICCDs '98), IEEE, Annapolis, Maryland, USA, May 1998.
- [**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.
- [**Rogerson,97**] Rogerson, D., "Inside COM", Microsoft Press, Redmond, WA, 1997.

- [Saikoski,00]** Saikoski, K. B. and Coulson G., “Configurable and Reconfigurable Group Services in a Component Based Middleware Environment”, Proc. International SRDS (Symposium on Reliable Distributed Systems) Workshop on Dependable and Group Communication (DSMGC 2000), October 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.
- [Shaw,96]** Shaw, M. and Garlan, D. “Software Architecture: Perspectives on an Emerging Discipline”, Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [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.

APPENDIX A: Essentials of Microsoft's Component Object Model (COM)

This appendix briefly overviews aspects of Microsoft's Component Object Model (COM) [COM,95] that relate to its use in OpenCOM. It is not intended to provide an exhaustive overview of the technology.

COM is underpinned by three fundamental concepts:

- uniquely identified and immutable interface specifications,
- uniquely identified components that can implement multiple interfaces, and
- a dynamic interface discovery mechanism.

COM supports uniqueness through 128 bit globally unique identifiers known as *GUIDs*. Further, the unique identifier of an interface is referred to as an *IID*, and that of a component type as a *CLSID*. The interface discovery mechanism builds on a special interface called *IUnknown* that must be implemented by every COM component. The purpose of *IUnknown* is actually twofold: *i*) it allows dynamic querying of a component (*QueryInterface()* operation) to find out if it supports a given interface (in which case a pointer to that interface is returned), and *ii*) it supports automatic *reference counting* of the number of clients using a component's interfaces. This allows the component to be garbage collected when it no longer has any clients.

Component and interface definitions are expressed in Microsoft's language independent Interface Definition Language (MIDL), and a tool called *midl* is used to compile these definitions and automatically generate language-specific class templates for programmers to complete. *Midl* also generates files called *type libraries* that store type information related to components and their interfaces. Though initially intended to support dynamic method dispatch through late binding, type libraries now also include meta-information describing components and their implemented interfaces that would otherwise not be available at run-time to a compiled language like C++.

Crucially for us, the COM standard defines the way in which components interoperate at the binary level; this is in terms of the *vtable*, a per-component table of function pointers based on C++ call conventions. *Vtables* natively support any language that supports function pointers. In addition, languages that do not support function pointers can still implement COM components if their support environment can be modified to export their functionality through function pointers. For instance, Java can implement COM components if the Java Virtual Machine (JVM) is modified to make its host Java classes available through a *vtable*. Related to binary level interoperability is the mandating of the *_stdcall* method calling convention (which essentially defines that each component method should clean the stack of its parameters before returning) on all COM components; this has important implications for our receptacle locking and interface interception architectures (see Appendix B).

Finally, COM employs a centralised, system-wide repository known as the *registry*. This stores component object files, type libraries, interface definitions etc. and supports GUID based retrieval of these entities.

APPENDIX B: OpenCOM

In the following, we expand on the description on OpenCOM given in section 2. Although our implementation is in C++ and the material below occasionally refers to C++ specific concepts, the design is sufficiently generic to be implemented in any language compatible with COM. Refer to section 2 for an overall, high-level, description of OpenCOM.

B1. Receptacle Implementation

Developers declare *receptacles* as a class template within the body of the implementation of their OpenCOM components. A receptacle contains an interface pointer and the supported interface type (expressed as a COM IID). When a receptacle is invoked, the interface pointer is used to invoke methods on the currently associated interface. The stored IID allows the component developer to differentiate between the various receptacles that their component implements. This is used in the implementation of the *IReceptacles* interface (as explained below). The developer must ensure that the correct receptacle is used to store an interface pointer passed in by the run-time at connection time and, conversely, that the correct receptacle has its interface pointer set to NULL at disconnection time.

B1.1 Receptacle Styles

We have found three styles of receptacle useful in our implementation:

- the *single pointer* receptacle

This contains a single pointer to an interface; it is the most common form and represents a simple requirement to utilise a given type of interface.

- the *multi-pointer-with-context* receptacle

This contains multiple pointers to implementations of the same type of interface. The multiple interfaces are discriminated by passing contextual information when invoking a method on the receptacle. This style is used in CF implementations where there is a need to select one of a number of plug-ins (see section 3.3).

- the *multi-pointer* receptacle

This contains multiple pointers to implementations of the same interface type but does not discriminate between them. It is useful for event notification where a callback is invoked on all the interfaces connected to the receptacle.

B1.2 Receptacle Invocation and Locking

OpenCOM offers mechanism-level support for integrity maintenance through the provision of per-receptacle *locks*. The system can be configured with or without these locks. With locking enabled, the higher layers can rely on OpenCOM to maintain integrity at the level of individual invocations. Each per-receptacle lock is competed for by: *i*) threads performing invocations involving the receptacle, and *ii*) the run-time performing a reconfiguration operation (e.g. a deletion) on the receptacle. When deleting a single connection, only the single receptacle need be locked, but when deleting a component, all receptacles involved in connections to the component's interfaces must first be locked. Receptacles remain locked until they are explicitly

unlocked by higher-level software; this is usually when they have been reconnected to appropriate interfaces after reconfiguration is complete.

When OpenCOM is configured without locking, invocations on receptacles do not incur locking overhead, but reconfiguration operations are potentially unsafe because they may disturb currently executing invocations. In this case, it is assumed that higher level CFs are constructed in such a way as to make reconfiguration safe at their own level, e.g. by employing checkpoints. Of course, custom integrity maintenance strategies can also be built using a combination of receptacle locking and CF based mechanisms.

B1.2.1 Non-Locking Receptacles

In non-locking receptacles, when a method is invoked, the dereference operator (`->` in C++) is overridden to return the stored interface pointer, and the compiler generates code to invoke the desired method on the pointed-to interface.

B1.2.2 Locking Receptacles

To understand the implementation of locking receptacles, one must first be aware of the layout of a COM component in memory (see figure B1). Essentially, each component instance contains a sequence of pointers to *vtables* (each known as an *lpvtbl* – long pointer to *vtable*) for each interface it implements. Given a pointer to an interface, i.e. a pointer to an *lpvtbl*, and a method to invoke on that interface, the compiler generates code to follow the pointers to arrive at the *vtable* and add an offset corresponding to the offset of the method in the interface's IDL specification. The slot at the calculated offset into the *vtable* points to the method's implementation, which is then called.

Our locking scheme requires the insertion of *reference counting* code to record the number of in-progress invocations on a receptacle (the run-time can only obtain a receptacle's lock if this count is zero) and *lock status checking* code that must be executed on each receptacle invocation. The latter code is implemented as follows: Each receptacle contains a 'fake' *lpvtbl* field pointing to a fake *vtable* also embedded within the receptacle. The overridden dereference operator returns a pointer to the receptacle's fake *lpvtbl* thus ensuring that subsequent invocations pass through the locking code. Each slot in the fake *vtable* points to hand-written assembly code that calculates the offset of the compiler's call into the fake *vtable*, checks to see if the receptacle has been locked by the run-time (if so, the invocation is aborted with an error code¹), increments the reference count, calls the intended method (by forming an address from the calculated offset and the stored interface pointers *lpvtbl*), decrements the reference count and returns the result to the invoker.

Note that it is not viable to simply set a receptacle's interface pointer to NULL and catch the ensuing exceptions that this would cause. This is partly because many COM compliant languages do not support exceptions. In addition, reference counting of in-

¹ As COM uses the *_stdcall* calling convention, aborting a method call presents the difficulty of having to clean the stack as part of the abort. This is achieved by maintaining a table inside each receptacle that indicates the number of parameter bytes for each method in the interface of the receptacle's type. This information is gleaned from the interface's type library and is filled in when the receptacle is first connected to an interface. We define macros for receptacle invocation that embody different behaviour to cope with aborted calls. The most widely is used is a macro that simply 'spins' on an invocation that is aborted until it succeeds, i.e. when the receptacle is (re)connected to an interface. Using macros avoids intrusion on the application code.

progress invocations would still required be make component instances deletion-safe. Finally, our invocation abort code is far more efficient than generating and handling an exception.

B2. Interception Architecture

Our interception architecture, embodied in the MetaInterception component and accessed via the IMetaInterception interface, is based on a marshal-by-value delegation architecture proposed by Brown [Brown,99], but extended with dynamic instantiation capabilities. In our architecture, we can dynamically attach and detach lists of pre- and post-processing methods over any interface. All clients of that interface transparently execute these methods before and/ or after any call to a method on that interface.

The method interception mechanism is very similar to the one used by locking receptacles. In fact, receptacles can be viewed as specialised interceptors, i.e. interceptors that have specific and fixed pre- and post-method processing routines (i.e., for reference counting, lock checking and call abortion). However, a fundamental difference lies in the way that the interception code is entered. A receptacle relates only to a single connection, whereas an interceptor needs to be present in every connection that the intercepted interface is participating in. For this reason it is not possible to simply integrate an interceptor with every receptacle instance because interception over the target interface would occur only on that connection to the interface. To resolve this issue, instantiation of an interceptor over an interface causes the *real lptvbl* in the component instance hosting the interface to be overwritten with a pointer to a fake *vtable* inside the interceptor. All invocations on the interface are now directed to the interception code. When deleting an interceptor, the component instance's *lptvbl* to the intercepted interface is restored. Note that this mechanism is completely separate to that used by the receptacles; when a receptacle's interception code invokes a real interface method, the invocation is transparently intercepted by any attached interceptor.

Figure B1 shows receptacle based invocation and interface interception working together.

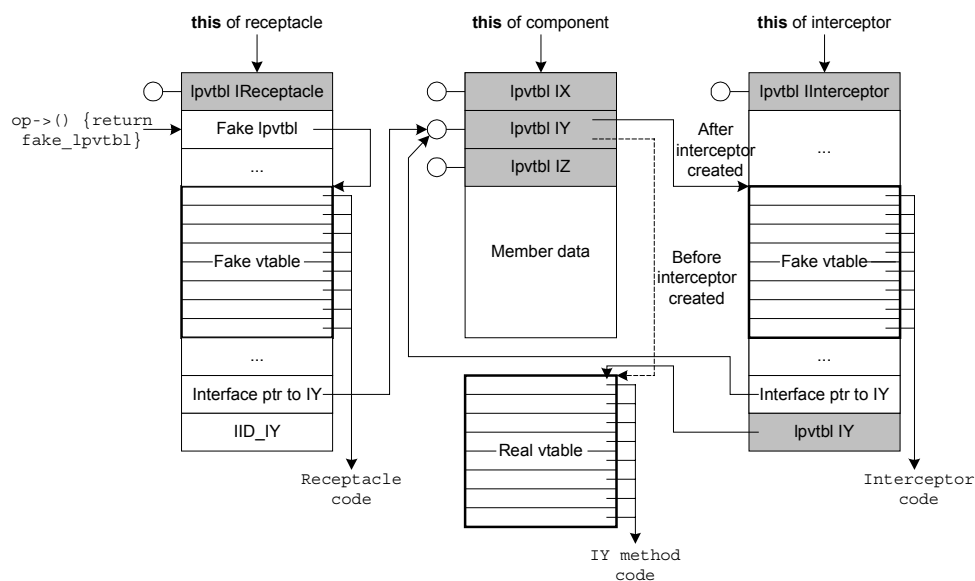


Figure B1: Receptacles and Interceptors in OpenCOM

B3. The OpenCOM Run-Time

The OpenCOM run-time services are implemented in a component called *OpenCOM*. This is a standard COM component that is created using COM's standard *CoCreateInstance()* method, thus allowing the rest of our component model to be bootstrapped. Only one OpenCOM instance exists per address space no matter how many times *CoCreateInstance()* is called, i.e., it is implemented as a *singleton*. The OpenCOM component implements the *IOpenCOM* interface and makes it available to OpenCOM compliant components:

```
interface OCM_IOCM : IUnknown
{
    HRESULT createInstance(
        [in] REFCLSID rclsid,
        [out] IUnknown **ppIUnknown,
        [in, string] const unsigned char *name);
    HRESULT deleteInstance(
        [in] IUnknown *pIUnknown,
        [out] OCM_IReceptacle *ppOCM_IRecps[],
        [out] int *pcElems);
    HRESULT connect(
        [in] IUnknown *pIUnkSource,
        [in] IUnknown *pIUnkSink,
        [in] REFIID iid,
        [in] OCM_RecpID_t recpID,
        [out] OCM_ConnID_t *pConnID);
    HRESULT disconnect(
        [in] OCM_ConnID_t connID,
        [out] OCM_IReceptacle **ppOCM_IRecp);
    HRESULT getConnectionInfo(
        [in] OCM_ConnID_t connID,
        [out] OCM_ConnInfo_t **ppConnInfo);
    HRESULT freeConnectionInfo(
        [in] OCM_ConnInfo_t *pConnInfo);
    HRESULT enumComponents(
        [out] IUnknown* *ppComps[],
        [out] int *pcElems);
    HRESULT getComponentName(
        [in] IUnknown *pIUnknown,
        [out] unsigned char **ppName);
    HRESULT getComponentPIUnknown(
        [in, string] const unsigned char *name,
        [out] IUnknown **ppIUnknown);
    HRESULT getComponentCLSID(
        [in] IUnknown *pIUnknown,
        [out] CLSID *pclsid);
};
```

CreateInstance() creates an OpenCOM component instance based on the specified component type, i.e. its CLSID. A pointer to the instance's IUnknown interface is returned which also acts as the instance's identity (given that it is unique within the hosting process). The instance can be named with a string, e.g. for compile time access to the instance, but can also be unnamed if NULL is passed as the name parameter. *DeleteInstance()* deletes a component instance based on its pointer to IUnknown. This method also invokes the instance's *ILifeCycle::shutdown()* method. Note that an array of pointers to receptacles is returned. All of these receptacles have been locked by the run-time during the instance deletion and it is the responsibility of some reconfiguring agent to unlock them (after they have been suitably reconnected) in order to allow invocations on them to proceed again (see section B1.2).

Connect() connects the source's receptacle to the sink's interface. A unique connection identifier is returned which can be used to manipulate the connection at a later stage. *Connect()* invokes the source instance's *IReceptacles::connect()* method so that connection specific actions can take place and the interface pointer can be stored in the appropriate receptacle. *AddRef()* is automatically called on the interface pointer. *Disconnect()* disconnects a previously established connection and returns a pointer to the now locked source receptacle in order to allow some reconfiguring agent to unlock it, when appropriate, later. This method also invokes the instance's *IReceptacles::disconnect()* method so that connection specific actions can take place and the interface pointer can be deleted from the appropriate receptacle. *Release()* is automatically called on the interface pointer.

GetConnectionInfo() returns a block of information describing a connection. *FreeConnectionInfo()* releases the memory internally allocated to hold the information block returned by *getConnectionInfo()*. *EnumComponents()* returns an array of pointers to IUnknown for all the component instances currently in the system. *GetComponentName()* returns an instance's name based on its pointer to IUnknown. An error is returned if the component has no name. *GetComponentPIUnknown()* returns a named instance's pointer to IUnknown. *GetComponentCLSID()* returns an instances CLSID based on its pointer to IUnknown.

B4. The Per-Component Management Interfaces

Each OpenCOM component must implement the following two interfaces (i.e. *ILifeCycle* and *IReceptacles*) in order to be manageable by the OpenCOM run-time.

B4.1 The ILifeCycle Interface

```
interface OCM_ILifeCycle : IUnknown
{
    HRESULT startup([in] OCM_IOCM *pOCM_IOCM);
    HRESULT shutdown(void);
};
```

ILifeCycle is a programmer implemented interface that allows a component to take lifecycle actions when an instance is created or deleted. We do not use the object-oriented class constructor and destructor for this purpose for two reasons, *i*) the implementation language may not support them, and *ii*) constructors are notoriously unstable for any but the most basic initialisation activities. A more specific reason for requiring lifecycle support is to allow instances to preserve and restore state over the system's lifetime (where instances may be created and deleted or replaced many times over).

The *startup()* method allows a component to take action whenever an instance is created. A pointer to the *IOpenCOM* interface is passed to allow the instance to store it as a member variable thus allowing convenient access from member methods. A component's *shutdown()* method allows an instance to take action while it is being deleted. Note that *startup()* should be called directly by the programmer after an instance is created. This separation between creation and initialisation was found necessary because *startup()* typically invokes receptacle based interfaces which must be connected after a component instance has been created by the run-time, i.e. before *startup()* can be invoked. On the other hand *shutdown()* is directly invoked by the run-time following a request to delete an instance.

B4.2 The IReceptacles Interface

```
interface OCM_IReceptacles : IUnknown
{
    HRESULT connect(
        [in] void *pSinkIntf,
        [in] REFIID riid,
        [in] OCM_ConnID_t provConnID,
        [in] OCM_RecpID_t recpID);
    HRESULT disconnect(
        [in] REFIID riid,
        [in] OCM_ConnID_t connID,
        [in] OCM_RecpID_t recpID);
};
```

IReceptacles is a programmer implemented interface that primarily allows a component to set and reset its receptacles' interface pointers at connection and disconnection time respectively. Its methods are not invoked directly by the developer. Instead, they are called from the OpenCOM run-time's *connect()* and *disconnect()* methods in order to allow the run-time to explicitly manage the connection process (to enable subsequent reconfiguration). Additionally, for each receptacle defined, this interface allows the programmer to take action whenever a connection is created/ deleted to/ from a specific receptacle. For instance, if a component has a thread sending data over a connection and the connection is to be torn down, then the developer is able to stop the thread from sending more data before physically disconnecting the receptacle.

The *connect()* operation connects the specified sink interface to a receptacle of the specified type in the target component instance. In the case of a multi-pointer-with-context receptacle, a context identifier must also be supplied. A connection identifier is returned. *Disconnect()* takes a previously established connection's connection identifier and disconnects the corresponding receptacle and interface by resetting the receptacle's internal interface pointer.

B5. The Per-Component Meta-Interfaces

The following three interfaces (i.e. *MetaInterface*, *MetaArchitecture* and *MetaInterception*) allow each component to access its meta-space. The interfaces must be explicitly inherited but their implementations are currently made available through *containment* of standard sub-components that we provide:

B5.1 The IMetaArchitecture Interface

```
interface IMetaArchitecture : IUnknown
{
    HRESULT enumConnsToIntf(
        [in] REFIID riid,
        [out] OCM_ConnID_t *ppConnsToIntf[]
        [out] int *pcElems);
    HRESULT enumConnsFromRecp(
        [in] REFIID riid,
        [out] OCM_ConnID_t *ppConnsFromRecp[]
        [out] int *pcElems);
};
```

EnumConnsToIntf() returns an array of connection identifiers detailing all the connections established on the specified interface of the target component instance. *EnumConnsFromRecp()* returns an array of connection identifiers detailing all the

connections established by the receptacle of the specified interface type on the target component instance.

B5.2 The IMetaInterface Interface

```
interface IMetaInterface : IUnknown
{
    HRESULT enumRecps(
        [out] OCM_RecpMetaInfo_t *ppRecpMetaInfo[],
        [out] int *pcElems);
    HRESULT enumIntfs(
        [out] IID *ppIntf[],
        [out] int *pcElems);
};
```

EnumIntfs() returns an array of interface IIDs defining the type of all the interfaces hosted by the target component instance. This information is extracted from the component's type library. *EnumRecps()* returns an array of information blocks defining the interface pointer types (as IIDs) and styles (single pointer, multi-pointer, etc.) of all the receptacles hosted by the target component instance. Ideally, we would like to extend Microsoft's IDL to make required interfaces have the same status as provided interfaces, i.e. to be emitted as part of a type library and made accessible through COM's *ITypeLibrary* interface. Currently, however, we tie the publication of a components' required interfaces into its implementation (i.e. as part of the declaration of its receptacles) and have our run-time extract them using a pattern.

B5.3 The IMetaInterception Interface

```
interface IMetaInterception : IUnknown
{
    HRESULT createInterceptor(
        [in] OCM_InterceptorType_t intcpType,
        [in] REFIID riid,
        [out] OCM_IInterceptor **ppOCM_IInterceptor);
    HRESULT deleteInterceptor([in] OCM_InterceptorType_t intcpType,
        [in] OCM_IInterceptor *pOCM_IInterceptor);
};
```

CreateInterceptor() creates an interceptor of the specified type on the specified interface of the target component instance, and returns a pointer to that interceptor's *control interface*. The interceptor intercepts on a per component interface basis, i.e. all methods in the interface are intercepted and once interception has been instantiated on a particular interface, all users of that interface are subject to intercepted calls; even those that connected to the interface before the interceptor was created (as discussed in section B2). *DeleteInterceptor()* deletes an interceptor of the specified type.

The *Interceptor*'s control interface is defined as follows:

```
interface IInterceptor
{
    HRESULT addPreMethod([in] const unsigned char *DLLName,
        [in] const unsigned char *methodName);
    HRESULT delPreMethod([in] const unsigned char *methodName);
    HRESULT addPostMethod([in] const unsigned char *DLLName,
        [in] const unsigned char *methodName);
    HRESULT delPostMethod([in] const unsigned char *methodName);
    HRESULT viewPreMethods([out] unsigned char *methodNames[]);
    HRESULT viewPostMethods([out] unsigned char *methodNames[]);
    HRESULT deleteInterceptor(void);
};
```

AddPreMethod() dynamically adds a named pre-processing method (from the named DLL) to the list of pre-processing routines attached to the specified interceptor component. The method is attached at the back of the list and will be invoked in order behind any other list members. *DelPreMethod()* deletes the specified pre-processing routine from the list of pre-processing routines. *AddPostMethod()* adds a post-processing routine to an interceptor component similarly to *addPreMethod()*. *DelPostMethod()* deletes the specified post-processing routine from the list of post-processing routines.

ViewPreMethods() returns a pointer to an array of strings representing the current list of pre-processing routines attached to the specified interceptor component. *ViewPostMethods()* returns a pointer to an array of strings representing the current list of post-processing routines attached to the specified interceptor component. *DeleteInterceptor()* deletes the target (this) interceptor. This method is called indirectly from *IMetaInterception::deleteInterceptor()*.