An Efficient Component Model for the Construction of Adaptive Middleware

Michael Clarke[1], Gordon S. Blair[2], Geoff Coulson[1], and Nikos Parlavantzas[1]

[1]Distributed Multimedia Research Group, Computing Department, Lancaster University, Lancaster LA1 4YR, UK.
{mwc,geoff,parlavan}@comp.lancs.ac.uk

[2]Dept of Computer Science, University of Tromsø, N-9037 Tromsø, Norway.
(On leave from Lancaster University)
gordon@cs.uit.no

Abstract. Middleware has emerged as an important architectural component in modern distributed systems. Most recently, industry has witnessed the emergence of component-based middleware platforms, such as Enterprise JavaBeans and the CORBA Component Model, aimed at supporting third party development, configuration and subsequent deployment of software. The goal of our research is to extend this work in order to exploit the benefits of component-based approaches within the middleware platform as well as on top of the platform, the result being more configurable and reconfigurable middleware technologies. This is achieved through a marriage of components with reflection, the latter providing the necessary levels of openness to access the underlying component infrastructure. More specifically, the paper describes in detail the OpenCOM component model, a lightweight and efficient component model based on COM. The paper also describes how OpenCOM can be used to construct a full middleware platform, and also investigates the performance of both OpenCOM and this resultant platform. The main overall contribution of the paper is to demonstrate that flexible middleware technologies can be developed without an adverse effect on the performance of resultant systems.

1 Introduction

Middleware has emerged as an important architectural component in modern distributed systems. The role of middleware is to offer a high-level, platform-independent programming model to users, and to mask out problems of distribution. Examples of key middleware platforms include CORBA, DCOM and the Java-based series of technologies (RMI, JINI, etc). These platforms generally provide an *object-oriented* programming model for the development of distributed applications and services. More recently, however, the industry has witnessed the emergence of *component-based approaches* such as JavaBeans, Enterprise JavaBeans, .NET and the CORBA Component Model (contained in CORBA v3). Such platforms aim to provide underlying support for the third party development, composition and subsequent deployment of components, and also typically ease the task of the management of non-functional properties of applications (e.g. security).

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 160-178, 2001.

With the approaches described above, component-based models are offered on top of the middleware platform. We however believe that there are considerable advantages to also exploiting component-based techniques *within* the middleware platform. In other words, a middleware platform would then be one particular configuration of components, thus encouraging both configurability and reconfigurability of the platform. For example, this approach would enable the selection of a minimal middleware configuration for an embedded device, or indeed a richer configuration with additional quality of service management facilities to offer guaranteed multimedia services. More specifically, we advocate the use of component-based techniques together with *reflection* [7] for developing next generation middleware platforms. Middleware platforms traditionally have a black-box architecture; in our approach, we exploit reflection to open up this black box and to encourage introspection and indeed adaptation of the underlying structure and behaviour of the platform [2]. The resultant platform exploits a (minimal) component model to construct the middleware platform, with the middleware platform then supporting an enhanced component model for the subsequent development of distributed applications.

Previous papers have reported on the motivation and design of OpenORB, our component-based reflective middleware architecture [1] [2]. Prototypes of this architecture have also been developed using the Python language [4]. This paper reports on OpenCOM; an efficient, lightweight and reflective component model that we have used to efficiently re-engineer OpenORB.

The specific goals of this paper are:

- to provide a detailed introduction to OpenCOM in terms of both design and implementation,
- to illustrate how OpenCOM can be used to construct a configurable and reconfigurable middleware platform,
- to investigate the performance of the underlying OpenCOM component model, and also (briefly) the resultant middleware platform.

The rest of the paper is structured as follows. Section 2 reports on the design of OpenCOM, highlighting the programming model offered by OpenCOM, and also the associated meta-interfaces. Section 3 then reports on the associated implementation of this component model. Following this, section 4 outlines the re-engineering of our OpenORB architecture using OpenCOM, while section 5 presents a performance evaluation of both the underlying component model and the resultant middleware platform. Section 6 contains some discussion of related work, and section 7 contains some concluding remarks.

2 The Design of OpenCOM

2.1 Background

OpenCOM is a lightweight and efficient *in-process* component model[1], built atop a subset of Microsoft's COM. We chose COM as the basis of our component model for

1 Meaning that all components in an OpenCOM based system exist in a single address space.

the following reasons: *i*) COM is standardised [10], 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).

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 binary-level interoperability standard (i.e. the *vtable* data structure), *ii*) Microsoft's Interface Definition Language (IDL), *iii*) COM's globally unique identifiers (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:

- 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 [9]);
- it adds mechanism-level functionality for reconfiguration, such as mutual exclusion locks to serialise modifications of inter-component connections;
- 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 [1].

2.2 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 a service *S*, it would declare a receptacle of type *S* which would be *connected* 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 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* which can be connected to more than one interface instance (see section 3.1 for more detail).

OpenCOM deploys a standard run-time substrate that is available in every OpenCOM address space (it is implemented as a singleton component called

1 The term 'receptacle' is also employed by the CORBA Components Model [14]. The concept itself appears in various other models under various names.

"OpenCOM" and exports an interface called *IOpenCOM*). The primary role of the run-time is to manage a repository of available 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/ disconnect receptacles and interfaces in its address space. Furthermore, to facilitate reconfiguration, the run-time records every creation/ deletion of each component/ connection in a per-address space meta-structure called the *system graph*. This enables it 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.

Each OpenCOM enabled component must implement the following pair of *component management* interfaces. These are called by the runtime and assist it in, respectively, creating/ deleting connections and in creating/ deleting components:

- *IReceptacles* offers operations to alter the interface(s) currently associated with (i.e., connected to) each of the host components' receptacles. These operations are only ever called by the run-time's connection management operations.
- *ILifeCycle* offers operations to be called by the run-time when an instance of the host component is created or 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 enabled component must inherit the implementation (through *containment* [17]) of three standard sub-components (called *MetaInterception*, *MetaArchitecture* and *MetaInterface*). These implement the reflective facilities identified in our previous work [1] and (respectively) export the following meta-interfaces from the host component:

- *IMetaInterception* enables the programmer to associate (dissociate) *interceptor* components with (from) some particular interface. Interceptors implement interfaces that contain *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/ removed at run-time and reordered as desired.
- *IMetaArchitecture* enables the programmer to obtain the identifiers of all current connections between the host components' receptacles and external interfaces. These identifiers can then be submitted to the above-mentioned IOpenCOM interface which returns information on the receptacle/ interface/ components involved in the connection.
- *IMetaInterface* supports inspection of the types of all interfaces and receptacles declared by the host component.

Figure 1 visualises the component model. It shows the OpenCOM run-time component (below) and an OpenCOM enabled component (above). The components' 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. 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. The OpenCOM runtime component is shown encapsulating the system graph and type libraries, and exporting the *IOpenCOM* interface.

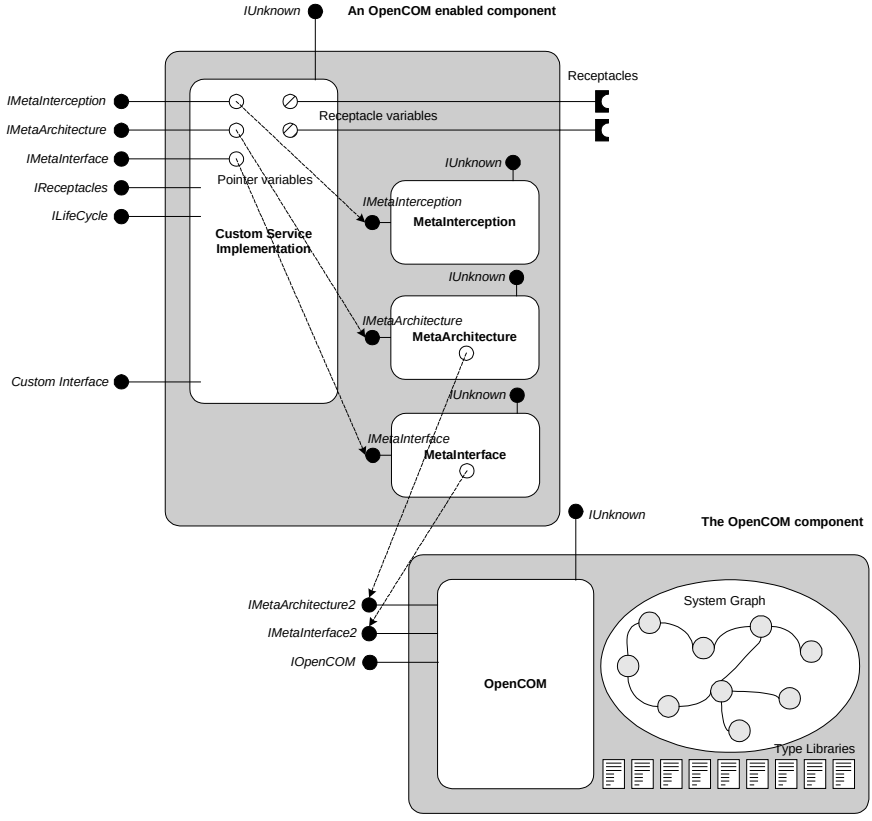

Fig. 1. The Architecture of OpenCOM.

2.3 Reconfiguration Support

The ability to dynamically reconfigure a system is most useful when these reconfigurations can occur arbitrarily and from any point within the system (not just from within the participating members of the reconfiguration). This is known as third party dynamic reconfiguration. However, any system that proposes to allow this type of reconfiguration must consider the implications for its own stability and integrity in the face of such operations. For instance, deleting a component while it is still being used will cause the results to be at best unpredictable and at worst lead to a crash.

COM supports the dynamic configuration of components and can also guarantee to delete components when they are no longer in use (assuming that strict reference

counting guidelines have been followed). In addition, components can also be dynamically reconfigured (but with no support for ensuring the resultant system's integrity) in the sense that they may have their raw interface pointers resolved against new interfaces. However, this can only be achieved from a first party point of view, i.e. only the component hosting the interface pointer can attach to a new interface. This is because interface pointers are simple variables that are not known to the runtime and therefore cannot be arbitrarily reconfigured by a third party.

In contrast, OpenCOM supports arbitrary, third party dynamic reconfiguration of both components and their connections. This is achieved through a combination of the first class status we place on receptacles (thus allowing component interdependencies to remain explicit and be managed at runtime) and the system graph (to allow access to the current runtime status of these interdependencies). OpenCOM makes dynamic reconfiguration safe by associating a lock with each receptacle. This lock is asserted whenever the first of an arbitrary number of simultaneous invocations takes place on the receptacle and is reset when the last invocation completes. During this time, any reconfiguration operations[1] involving the associated connection are blocked. However, before such a reconfiguration operation actually blocks, it is able to ensure that any new invocations are aborted (guaranteeing that it will acquire the lock when the current batch of invocations complete). Once the lock is acquired, all future invocations continue to be aborted until the lock is reset by higher level software (presumably after the receptacle has been successfully connected elsewhere).

3 The Implementation of OpenCOM

In this section we present the implementation of OpenCOM focusing mainly on the concepts introduced in the previous design section. Although our current 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.

3.1 Receptacle Implementation

Developers declare *receptacles* as a templated class (templated by its interface type) within the body of the implementation of their OpenCOM enabled components. A receptacle contains (among other things as discussed in section 3.1.2 below) 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 and is used in the implementation of the *IReceptacles* interface (see section 2.2). The developer must ensure that the correct receptacle is used to store an interface pointer passed in by the

1 By reconfiguration operation we mean the disconnection of a connection either directly or as a consequence of a component deletion (which causes all of the components' connections to be disconnected). This is followed by a reconnection to a new interface implementation to complete the reconfiguration.

run-time at connection time and, conversely, that the correct receptacle has its interface pointer set to NULL at disconnection time.

In general, we have found three styles of receptacle useful in our implementation:

- the *single pointer* receptacle 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 contains multiple pointers to implementations of the same type of interface. The pointers are discriminated by passing in contextual information when invoking a method on the receptacle. This style is used heavily when there is a need to select one of a number of plug-ins in a Component Framework (CF), see section 4.1,
- the *multi-pointer* receptacle 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.

3.1.1 Locking and Non-locking Receptacles. OpenCOM offers mechanism-level support for the maintenance of system integrity in the presence of dynamic reconfiguration through the provision of per-receptacle locks. However, OpenCOM can be built with or without these locks and this does not affect the way in which receptacles are invoked or manipulated from their users point of view.

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 software is constructed in such a way as to make reconfiguration safe at its own level (see section 4.1). In contrast, when locking is used, higher level software can rely on OpenCOM to make reconfiguration safe but must incur an invocation overhead (see section 5).

Invoking a receptacle is achieved by calling its overridden de-reference operator (i.e. `->()` in C++) along with the desired method, c.f. smart pointer classes. In the non-locking case, this simply returns the stored interface pointer and the compiler then generates code to invoke the supplied method on the pointed-to interface. In contrast, the sequence of events that occur after an invocation of a method on a locking receptacle are more complex and are examined in detail below.

3.1.2 Invocation of Locking Receptacles. To understand the implementation of locking receptacles, one must first be aware of the layout of a COM component in memory. 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 (see 'the component' in figure 2). Given a pointer to an interface, i.e. a pointer to an lptvbl, 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 de-reference operator returns a pointer to the receptacle's fake lpvtbl thus ensuring that subsequent invocations pass through the locking code (see figure 2 – in the 'before' interception state). Each slot in the fake vtable points to hand-crafted 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 code[1]), 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-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.

3.2 Meta-space Implementation

This section details the realisation of the standard components that implement each of OpenCOMs' meta-spaces and which must be inherited by every OpenCOM enabled component.

3.2.1 MetaArchitecture. The meta-architecture meta-space component leverages support from the OpenCOM runtime (in terms of accessing a private interface) in order to access the system graph. The graph stores numerous pieces of information about each component when they are created and updates this information as and when the components are involved in reconfigurations. Most importantly, the graph maintains two lists for each component representing the identities of the components connected to their interfaces and connected from their receptacles respectively. This information allows the meta-architecture component to isolate all the connections that the current component is involved in.

3.2.2 MetaInterface. COM supports interfaces as first class entities and provides a convenient way to query a component for them, namely the Type Library facility. These are binary files generated at the same time that the component's IDL files are compiled and contain many type details about a component's implementation that

---

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 (see section 3.2.2) 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 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.

would otherwise be lost when its source files are compiled. The meta-interface meta-space component uses the COM system *ITypeLibrary* interface to query a component's type library file and return the IID's of the interfaces it implements.

Ideally, we would like to extend Microsoft's IDL to allow receptacles (i.e. required interfaces) to have the same status as interfaces, i.e. to be emitted as part of a type library and made accessible through the *ITypeLibrary* interface. Currently, however, we tie the publication of a components' receptacles into its implementation (i.e. part of the declaration of its receptacles) and have our runtime extract them from the component's host Dynamic Link Library (DLL) using a pattern.

3.2.3 MetaInterception. The implementation of our per-interface interception architecture (embodied by the meta-interface meta-space component) is based on a marshal-by-value delegation architecture proposed by Brown [3], 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* lpvtbl 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 lpvtbl 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 2 shows receptacle based invocation and interface interception working together according to the descriptions above. In this diagram, the receptacle (left) is of interface type IY and contains a pointer to the IY interface of the component (middle). This interface is intercepted by the interceptor (right) in order to add pre and post processing routines over all the implementations of the methods in interface IY.

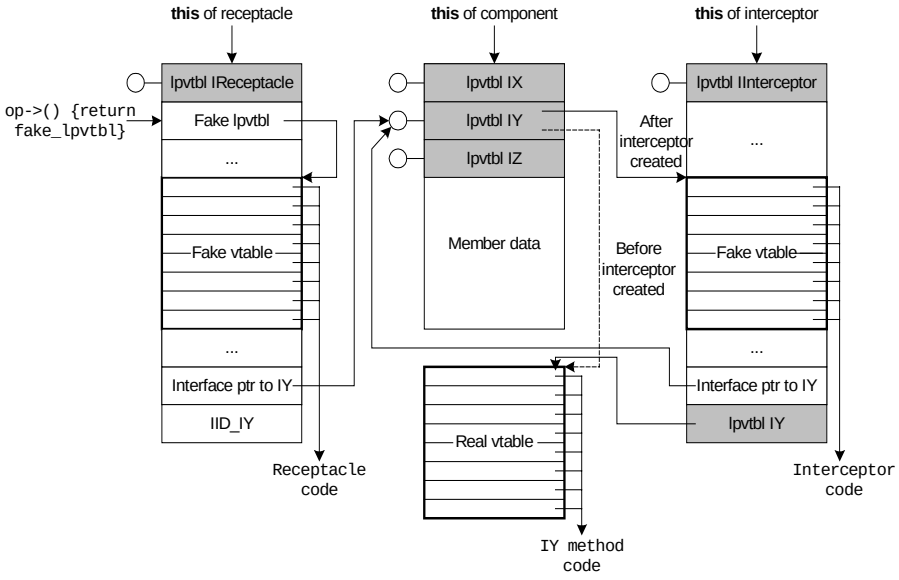

Fig. 2. Receptacles and Interceptors in OpenCOM.

4 OpenORB v2: An OpenCOM Case Study

Our ultimate purpose in designing and implementing the OpenCOM component model is to experiment with the construction of adaptive middleware platforms. This section details our experiences in implementing such a platform which we have named OpenORB v2 in reference to our previous implementation of reflective middleware [4].

4.1 Design

Although OpenCOM allows the construction of lightweight, efficient and reconfigurable components it does not directly support their *sub-composition*, c.f. nested components that can be treated as an individual unit through a unified API. This was a deliberate design decision; we do not wish to enforce any particular system-wide nesting model upon the platforms built from OpenCOM. We envisage that different domains within these platforms would have different nesting requirements and it is therefore the responsibility of the platform builder to specify appropriate support for nesting within the components that populate each of these domains.

In particular in the design of OpenORB v2, we have instantiated the notion of *Component Frameworks* (CFs) [18]) to support the nesting of components. CFs refer to “collections of rules and interfaces (contracts) that govern the interaction of a set of components plugged into them”. OpenORB v2’s CFs each define an abstract interface

and manage different implementations of this interface embodied by and plugged in as separate components[1] (see figure 3). CFs are targeted at a specific domain and embody rules and interfaces that make sense in that domain. The idea is that users of CFs interact with them for services through well defined APIs that encompass the services of the CF's constituent components. Additionally, these APIs include operations for the constrained (re)configuration of the CF. This implies that in OpenORB v2, it is only the CFs themselves that use OpenCOM's runtime support for reconfiguration (IOpenCOM); external entities use the CF's own API. For instance, the communications domain of a middleware platform may mandate that it will only accept reconfiguration operations on sub-components that support a specific interface, e.g. IProtocol, so that it can constrain its own reconfiguration to units of communication protocols (rather than allowing the replacement of the whole domain).

Note that the design of our CFs does not mandate whether they must be supported by a locking or non-locking OpenCOM substrate. Although using locking receptacles makes reconfiguration trivially safe it does incur overhead (see section 5). Alternatively, a CF may be able to avoid the need to use locking receptacles through other techniques, e.g. the checkpointing of safe reconfiguration points. This becomes plausible if the CF restricts the reconfiguration options for its sub-components.

4.2 Structure

OpenORB v2 is structured as a top-level CF that is composed of three layers of further CFs (see figure 3). The top level CF enforces the three layer structure by ensuring that each component/CF only has access to interfaces offered by components/CFs in the same or lower layers. 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.

The *resources layer* currently 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 *filter* components. Finally, the *binding layer* contains the *binding CF* that accepts *binding type implementations*. This is a crucial part of the platform's architecture because it determines the programming model offered to its users.

4.3 Implementation

OpenORB v2 consists of approximately 50,000 lines of C++ (including 10,000 lines for the OpenCOM runtime and support components) divided into 30 components and six CFs[2]. The bulk of the OpenORB v2 code is derived from GOPI, a CORBA

1 The CFs employ a *multi-pointer-with-context* receptacle to select between multiple managed components at run-time.

2 Note that not all of the components belong to a second level CF. Some exist purely as independent services and are therefore not exposed for semantically managed reconfiguration (though they could still be reconfigured through direct access to the IOpenCOM runtime interface if desired).

compliant, multimedia capable, middleware platform that we have developed previously [5]. We chose to reuse an existing middleware platform's code base in order to reduce the development effort needed to produce an OpenCOM enabled ORB. It has allowed us to rapidly experiment with aspects of dynamic reconfiguration within the ORB rather than be side-tracked by the development of its services.

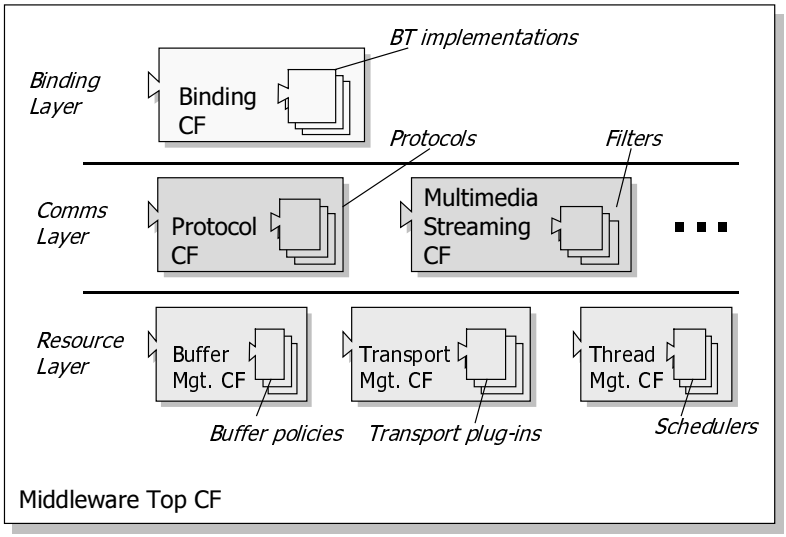

Fig. 3. Top level architecture of OpenORB v2.

GOPI was originally written in C for Unix platforms and consists of a single library statically linked to its applications. In reusing GOPI's code base within an OpenCOM environment we had to undertake a number of tasks, including i) the porting of GOPI to Win32 (as COM and its tools are only faithfully implemented on this platform), ii) the conversion of GOPI to C++ (as COM components are most conveniently implemented in C++), and iii) the conversion from C++ to OpenCOM (i.e. the breaking down of the static GOPI library into dynamically loadable components).

Particularly problematic experiences during this process were:

- the re-implementation of a number of Unix style services under Win32 including; the *signal* abstraction used heavily in GOPI timing code (re-implemented using events and some undocumented Win32 Structured Exception Handling code) and *pipes* (re-implemented using shared memory through memory mapped files),
- the identification of discrete services and their publicly available methods from the C code in order to guide their C++ re-implementation in terms of classes (the basis of COM components) and pure virtual classes (the basis of COM interfaces),
- the use of C++ class and pure virtual class definitions in the reverse engineering of IDL component and interface specifications respectively when migrating the C++ code to the OpenCOM environment,
- the identification of the interdependencies between OpenORB v2 components to facilitate the declaration of their receptacles, and

- the isolation of dependency interactions (i.e. invocations on dependent interfaces) within each OpenORB v2 component, which were then replaced by receptacle based invocations.

Our experience with this sizeable implementation effort has alleviated concerns we had about the explicit identification of component inter-dependencies; we feared that this may lead to a combinatorial explosion in the higher layers such that every component would begin to directly depend upon every other. This would make it difficult to code such components and make the system graph extremely complicated. However, we found that the maximum number of direct dependencies was seven (on the IIOP component) while the average was just three.

5 Performance Evaluation

In this section, we investigate the performance of OpenCOM and the overhead of its deployment within OpenORB v2. To provide meaning for the figures, we also compare against relevant baseline and equivalent technologies. All tests in the subsequent sections 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.

5.1 Performance of OpenCOM

In evaluating the performance of OpenCOM we are primarily interested in the additional overhead of its augmentations over COM *on the functional control path*, i.e. the overhead that OpenCOM introduces into a platforms' services. Specifically, we do not try to measure the overhead of *non-functional* OpenCOM characteristics (i.e. reconfiguration and architectural and interface reflection) because such operations are relatively rare and are executed off the functional control path, e.g. by third parties monitoring the functional aspects of the system..

The primary mechanisms of OpenCOM that affect the performance of a systems functional control path are receptacle based invocations and intercepted invocations[1] (i.e. intercepting reflection). Figure 4 presents the raw performance of these mechanisms in terms of maximum calls/sec throughput on a method with a NULL body. We compare against C based invocations (the basis for method calls in GOPI) and COM invocations (the baseline). In addition, we provide figures for the various combinations of locking / non-locking receptacles and interception in OpenCOM.

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

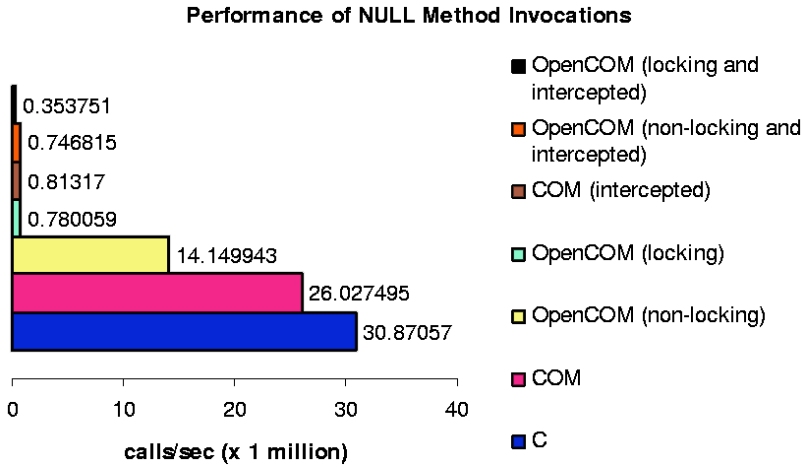

Fig. 4. Performance comparison of NULL method invocations.

The figures demonstrate a marked difference in the processing needed to execute simple method calls compared to the complex interactions embodied by OpenCOM based invocations. The C based invocation simply loads an immediate register and executes a machine level `CALL` through that register. A COM based invocation (i.e. a C++ virtual method call) must traverse lpvtbl and vtable pointers (through memory accesses) before performing the method `CALL`. 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. Section's 3.1.2 and 3.2.3 discuss the considerable processing involved in locking receptacle based and intercepted invocations respectively. 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 requires synchronisation to protect its lock variable. We use a Win32 `CRITICAL_SECTION` object to minimise this overhead as it spins at user level for a pre-determined time before blocking in the kernel.

Although the difference in raw invocation throughput between COM and OpenCOM is considerable (especially when using locking and intercepted invocations) it does not represent the actual effect of OpenCOM on a real system. This is because it is expected that the time taken to invoke a method is far smaller than the time taken to actually execute the method's body. Figure 5 demonstrates the effect of replacing the empty method body used in figure 4 with a relatively busy method body (implementing a 1000 iteration empty loop). It clearly shows 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.

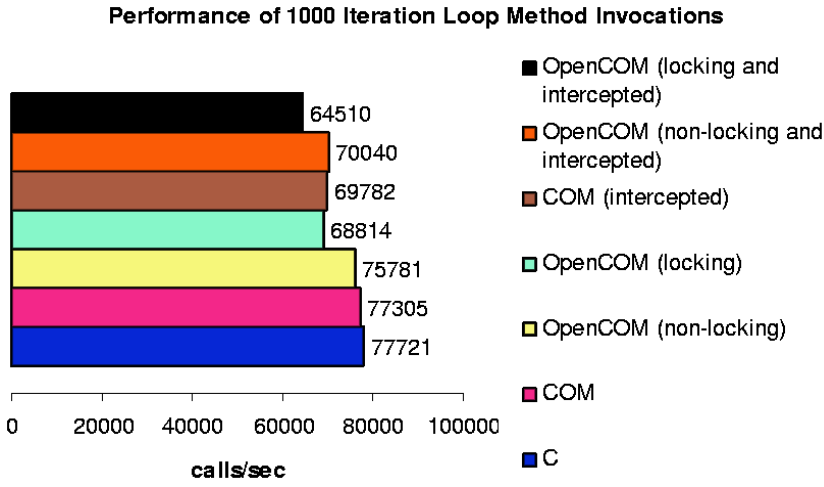

Fig. 5. Performance comparison of complex method invocations.

5.2 Performance of OpenORB v2

Given that receptacle and interceptor based invocations should not overly affect the performance of a realistic system, i.e. one that performs significant amounts of processing within its methods, then we would expect that OpenORB v2 will perform on a par with an equivalently coded system without OpenCOM. To test this theory, we compared the performance of OpenORB v2 with that of two other ORBs: GOPI v1.2 and Orbacus 3.3.4. As stated, GOPI provided much of the source code for OpenORB v2 (i.e. is an equivalent system) but is written in C and implemented in a single library. Orbacus is well known as one of the fastest and most mature CORBA-compliant (i.e. equivalent to OpenORB v2 in this sense) commercial ORBs available.

Our tests compared raw RPC method invocations per second *over the loopback interface* on the reference machine for each ORB. An IDL interface was employed that supported a single operation that took as its argument an array of octets of varying size and returned a different array of the same size. The implementation of this method at the server side was null. OpenORB v2 was tested with both a locking and non-locking OpenCOM substrate and each of it's RPCs involved 67 receptacle based invocations on the control path (32 on the client side and 35 on the server side). The results are shown below in figure 6. It can be seen that for packets of less that 1024 octets, non-locking OpenORB v2 performs about the same as Orbacus, with GOPI running around 10% faster. Locking OpenORB v2 runs around 15% slower than GOPI, i.e. only 5% slower than non-locking OpenORB v2. Both GOPI and OpenORB v2 fair slightly better than Orbacus as packet size goes beyond 1024 octets; we believe this is due to the buddy memory allocation scheme [8] that they use (which performs better for larger buffer allocations). As might be expected, there is a diminishing difference between all three systems as packet size increases further; this is presumably because the overhead of data copying begins to outweigh the cost of call processing.

Despite the additional work involved in carrying out receptacle based invocations, it can be seen that the performance of OpenORB v2 is entirely comparable to the non-componentised ORBs in both non-locking and locking configurations.

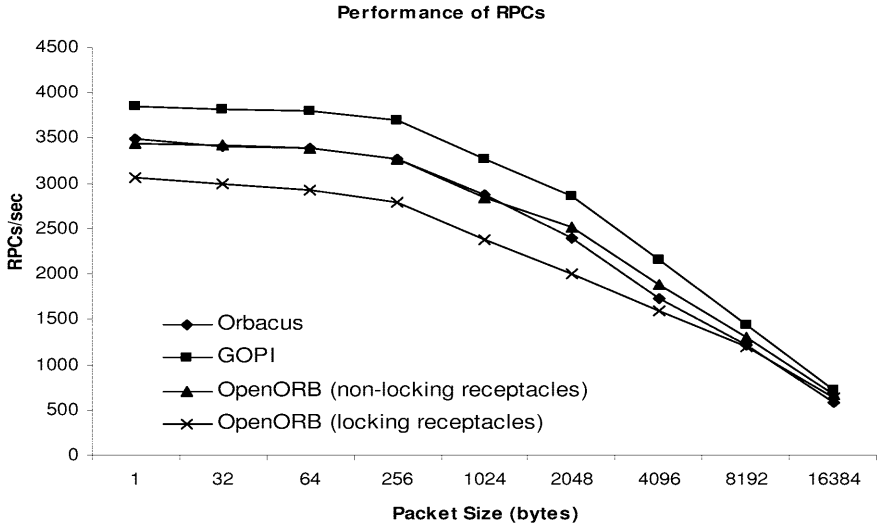

Fig. 6. Performance of OpenORB v2 versus GOPI and Orbacus.

6 Related Work

COM+ [11], Enterprise JavaBeans [19] and CORBA Components [14] are heavy-weight component models for building transactional distributed applications. They all employ a similar architecture providing a separation between the functional aspects of the application, which are captured by the components, and the non-functional, technical concerns, which are captured by the *container*. In contrast, OpenCOM is a lightweight, minimal component model which can be used uniformly throughout the system. Container-based component models can be built on top of OpenCOM if required. .Net [12], the new component model from Microsoft, is a major improvement over COM/COM+ in terms of introspection and dynamic type generation facilities. However, it still follows the same heavy-weight, container-based philosophy, whereby infrastructure services such as remoting (remote method invocation) are inseparable parts of the .Net runtime.

XPCOM [13] is a lightweight component model that, similarly to OpenCOM, is built on top of the core subset of COM. However, it does not provide any special support for dynamic reconfiguration. Knit [15] is a component model for building systems software. However, the model is specifically designed for statically composing systems; the components and their interconnections do not change after the system is configured and initialised. The component interfaces are not object-based and the model mainly targets low-level, C code. MMLite [6] is an operating system built using COM components. It provides limited support for dynamic reconfiguration

through the “mutation” mechanism, which enables the replacement of a component implementation at run-time.

DynamicTAO [9] and LegORB [16] are flexible ORBs that employ a dependency management architecture. This relies on a set of configurators that maintain dependencies among components and provide a set of hooks at which components can be attached or detached dynamically. OpenCOM supports a similar capability but it is an integrated part of the component model.

7 Conclusions

This paper has considered the design and implementation of OpenCOM, a lightweight and efficient reflective component model designed specifically for the development of middleware platforms. In other words, we exploit a component model for the construction of the middleware platform itself, which in turn provides an enhanced component model (for example, with intrinsic support for distribution) to application developers. The resultant middleware is more open and flexible, in terms of both configurability and reconfigurability.

Key features of OpenCOM include:

- the use of various styles of receptacles (single pointer, multi-pointer-with-context, multi-pointer) to make explicit the dependencies of components on their environment,
- the use of reflection to enable introspection of interfaces and receptacles and the associated component graph, as well as the dynamic insertion or deletion of interceptors, and
- backwards compatibility with the COM standard.

We have also demonstrated how OpenCOM can be used to construct a middleware platform, based on the related OpenORB architecture. In addition, it has been shown that the performance of the resultant system is on a par with established monolithic middleware platforms while simultaneously offering the benefits of componentisation and reflection introduced through the use of OpenCOM.

Acknowledgements. The research described in this paper is partly funded by the EPSRC together with BT Labs (through grant GR/M04242). The research is also partly financed by France Telecom R&D (CNET grant 96-1B-239). Finally, we would like to acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK, and the Universities of Oslo and Tromsø (all in Norway).

References

[1] 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.

[2] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., "The Design and Implementation of OpenORB v2", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.
[3] Brown, K., "Building a Lightweight COM Interception Framework Part 1: The Universal Delegator", Microsoft Systems Journal, January 1999.
[4] 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
[5] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.
[6] J. Helander and A. Forin. "MMLite: A Highly Componentized System Architecture". In Proc. of the Eighth ACM SIGOPS European Workshop, pp 96-103, Sintra, Portugal, September 1998.
[7] Kiczales, G., des Rivières, J., and Bobrow, D.G., "The Art of the Metaobject Protocol", MIT Press, 1991.
[8] Knuth, D.E., "The Art of Computer Programming, Volume 1: Fundamental Algorithms", Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.
[9] 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.
[10] Microsoft, "The Component Object Model Specification", http://www.microsoft.com/com/resources/comdocs.asp. Last updated: 15/04/1999.
[11] Microsoft, COM Home Page, http://www.microsoft.com/com/default.asp. Last updated: 01/06/2000.
[12] Microsoft, .Net Home Page, http://www.microsoft.com/net. Last updated: 01/02/2001.
[13] Mozilla Organization, XPCOM project, 2001, http://www.mozilla.org/projects/xpcom
[14] Object Management Group, "CORBA Components" Final Submission, OMG Document orbos/99-02-05.
[15] A. Reid, M. Flatt, L. Stoller, J. Lepreau, E. Eide "Knit: Component Composition for Systems Software". In proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), Usenix Association, pp. 347-360, October 2000.
[16] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., IFIP/ACM Middleware'2000 Workshop on Reflective Middleware. IBM Palisades Executive Conference Center, NY, April 2000.
[17] Rogerson, D., "Inside COM", Microsoft Press, Redmond, WA, 1997.
[18] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
[19] Sun Microsystems, "Enterprise JavaBeans Specification Version 1.1", http://java.sun.com/products/ejb/index.html.

Appendix A: Essentials of Microsoft's Component Object Model

This appendix offers a brief overview of Microsoft's Component Object Model (COM) [Microsoft,99] that relates *to its use in OpenCOM*, i.e. using it's *in-process server* model (where components reside in the same address space). It is not intended to provide an exhaustive overview of the technology. In particular, we do not discuss aspects of COM's distribution mechanisms, i.e. its *local server* model (where

components reside in different address spaces but on the same machine) and its latterly introduced *remote server* model extension (where components reside on different machines) known as DCOM.

COM is underpinned by three fundamental concepts: i) uniquely identified and immutable interface specifications, ii) uniquely identified components that can implement multiple interfaces, and iii) a dynamic interface discovery mechanism. COM supports uniqueness through the use of 128 bit globally unique identifiers known as GUIDs, these are generated through the use of platform specific tools. The interface discovery mechanism is implemented through the notion of a special interface called *IUnknown* that must be implemented by every COM component. The purpose of IUnknown is actually twofold: *i*) it allows the 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 implements *reference counting* in terms of the number of clients using a components' interfaces. Reference counting is used to garbage collect components when they no longer have any clients.

Component and interface definitions are specified in Microsoft's language independent Interface Definition Language (MIDL) and then a tool (*midl*) is used to automatically generate language specific templates of these specifications for programmers to complete. *Midl* also generates files known as *type libraries* that efficiently embody all manner of type information related to components and their interfaces. Though initially intended to support dynamic method dispatch through late binding, these files include meta-information describing components and their interfaces that would otherwise not be available to a compiled language at runtime.

Importantly, the COM standard also defines the way in which components interoperate at the binary level. Primarily, this means in terms of the *vtable*; a C++ native data structure that mandates the way in which access to a components' interfaces is achieved. The vtable is effectively a per-component table of function pointers and any language that can support function pointers may natively interoperate with components written in C++. In addition, languages that do not support function pointers can still implement COM components if their support environments 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 hosted Java classes available through a vtable. In addition to the use of the vtable to support binary compatibility, COM also mandates that all components must be compiled using the _stdcall calling convention (which essentially defines that each component method should clean the stack of its parameters before returning). This has important implications for our receptacle locking and interface interception architectures (see sections 3.1.2 and 3.2.3 respectively).

Finally, COM employs a system-wide repository known as the registry for locating component object files, type libraries, interface definitions etc. based on their GUIDs.