A GENERIC COMPONENT MODEL FOR BUILDING SYSTEMS SOFTWARE

Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, Thirunavukkarasu Sivaharan

Computing Department, Lancaster University, UK
contact: geoff@comp.lancs.ac.uk

ABSTRACT

Component-based software structuring principles are now commonly and successfully applied at the application level; but componentisation is far less established when it comes to building low-level systems software. Although there have been pioneering efforts in applying componentisation to systems-building, these efforts have tended to be narrowly targeted at specific application domains (e.g. embedded systems, operating systems, communications systems, programmable networking environments, or middleware platforms). They also tend to be narrowly targeted at specific deployment environments (e.g. standard personal computer (PC) environments, network processors, or microcontrollers). The disadvantage of this narrow targeting is that it fails to maximise the genericity and abstraction potential of the component approach. In this paper we argue for the benefits and feasibility of a generic yet tailorable approach to component-based systems-building that offers a uniform programming model in a wide range of target domains and deployment environments. More specifically, we present our OpenCom component model which is explicitly tailorable to diverse domains and environments. The component model is supported by a reflective runtime architecture that is itself built from components. After describing OpenCom and evaluating its performance and overhead characteristics, we present and evaluate two case studies of systems we have built using OpenCom technology, thus illustrating its benefits and its general applicability.

1. Introduction

Application-level component-based software is now well established. Prominent examples include: browser plug-ins [Mozilla,05], JavaBeans and Enterprise JavaBeans [Sun,05], the CORBA Component Model [OMG,99], Microsoft's .NET [Microsoft,05], and ICENI Grid components [Furmento,02]. Overall, some key characteristics of the component approach are as follows: *i*) it promotes a high degree of genericity and abstraction in software design, implementation, and deployment, which leads to higher programmer productivity; *ii*) it facilitates flexible configuration of software (and, potentially, run-time *re*configuration), and *iii*) it fosters third-party software reuse [Emmerich,02].

A broadly-accepted definition of software components is that they are "*units of composition with contractually-specified interfaces and explicit context dependencies only ... that can be deployed independently and are subject to composition by third parties*" [Szyperski,98]. Some component models restrict composition to build time or load time, but many (including all of the above) additionally support the composition of components at *run time*. The "explicit context dependencies" aspect of the definition means that third parties can straightforwardly deploy independently-developed components into established runtime software environments. Deployment is straightforward because it is clear and explicit what a newly-deployed component expects from its environment.

Although component models are widely used at the application level, it is far less common to see the component approach being exploited in the construction of *low-level systems software* such as embedded systems, operating systems, communications systems, programmable networking environments, or middleware platforms. This is primarily because systems environments are typically far more demanding than application environments in terms of complexity, performance, and resource constraints. Nevertheless, the potential of the component approach appears in principle just as compelling in the systems area as it is in the applications area, and this view has been borne out by a number of pioneering efforts over the last few years. For example, proposals for component platforms for building *embedded systems* include Pebble [Magoutis,00], PECOS [Winter,02] and Koala

[Ommering,00]; proposals for componentised *operating systems* (OSs) include THINK [Fassino,02], OSKit [Ford,97] and MMLITE [Helander,98]; proposals for componentised *programmable networking environments* include VERA [Karlin,01], MicroACE [Johnson,03], and Netbind [Campbell,02]; and proposals for componentised *middleware platforms* include LegORB [Roman,00], k-Components [Dowling,01] and various JavaBeans-based approaches (e.g. [Bruneton,00] and [Joergensen,00]).

However, all of these efforts suffer from the key limitation that they are *narrowly targeted*. This applies in two senses: *i*) in terms of the *target domain* at which they are aimed (i.e. embedded systems, OSs, etc.), and *ii*) in terms of the intended *deployment environment* in which they will operate (e.g., most of the above-mentioned technologies have been deployed only on conventional desktop machines as opposed to more 'exotic' deployment environments like personal digital assistants (PDAs), embedded hardware, or network processors). The disadvantage of this narrow targeting is that it fails to maximise the genericity and abstraction potential of the component approach—for example it locks systems component programmers into narrow, non-transferable skill sets and areas of expertise. Also, it fails to support the construction of component-based systems that span target domains and/or deployment environments (e.g. embedded middleware, or OSs for network processors).

In this paper, we discuss a *general-purpose* component-based systems-building technology called 'OpenCom'. OpenCom tries to maximise the genericity and abstraction potential of the component based programming model *while at the same time* supporting a principled approach to supporting the unique requirements of a wide range of target domains and deployment environments. This is achieved by splitting the programming model into a simple, efficient, minimal kernel, and then providing on top of this a principled set of extension mechanisms that allow the necessary tailoring. We also recognise and support a *separation of roles* between programmers who use the extension mechanisms to realise an OpenCom-based platform in a given deployment environment, and programmers who then use this environment to develop a target system (e.g. an embedded system, OS, etc.).

In more detail, the design of OpenCom tries to address the following requirements:

- *Target domain independence*. A general purpose systems-building technology should provide only generic and fundamental functionality that is independent of the specialist needs of any particular target domain. For example, a generic technology should *not* inherently support characteristics such as real-time execution, sand-boxing, or 24x7 availability. This is because such characteristics carry an inevitable cost which should not be incurred where they are not required. Nevertheless, a general purpose technology should be *inherently tailorable* and *extensible* so that it can be specialised in a natural and explicitly-supported way to meet such needs where required. The same consideration applies to run-time reconfigurability: the technology should provide a basis for this but should not *dictate the policies* that control and manage it.

- *Deployment environment independence*. The technology should be straightforwardly deployable in a wide range of deployment environments from PCs, to supercomputers, to set-top boxes, to resource-poor PDAs, to bare-iron embedded systems with no OS support, to networks-on-a-chip. This implies *simplicity* (for ease of porting), *small memory footprint*, and *programming language independence*. More fundamentally, it again implies inherent support for *tailorability* and *extensibility*. A key aspect of extensibility at the deployment-environment level is that it should be straightforward to expose idiosyncratic hardware and software features of the underlying deployment environment (e.g. multiple processors, hardware hashing units, optimised inter-process communication (IPC) channels, memory hierarchies, etc.) in terms of the native abstractions of the generic component-based programming model.

- *Negligible overhead*. As well as incurring only a small memory overhead, the technology should impose as small a demand as possible on other resources—especially processing resources. This particularly implies that the 'in-band' execution path [Coulson,04] of target systems should be as independent as possible of any runtime support provided by the technology (e.g. inter-component communication should not be reliant on a kernel-mediated message passing service). In addition, exposing deployment-environment-specific features in terms of generic programming model abstractions (as discussed above), should incur as small a performance penalty as possible—ideally a penalty of zero.

The remainder of this paper is structured as follows: First, section 2 provides an overview and motivation of the general OpenCom approach. Then, section 3 presents in detail our minimal component-based programming model. Next, sections 4 and 5 discuss the key 'extension mechanisms' that are (optionally) layered on top of the

OpenCom kernel and are instrumental in providing the necessary tailorability and extensibility. Section 6 then analyses the inherent performance characteristics and overheads of the OpenCom approach; and section 7 presents case studies of our recent use of OpenCom, including its extension layers, in building non-trivial systems. Finally, related work is discussed in section 8, and our conclusions are offered in section 9.

2. Overall Approach

We approach the satisfaction of the requirements identified above—i.e. target domain independence, deployment environment independence, and negligible overhead—through the architecture illustrated in figure 1. At the heart of the architecture is a minimal *component runtime kernel* that supports the basic services of *loading* and *binding* components. This is discussed in detail in section 3. A runtime kernel is required to be able to support dynamic systems which have an inherent need for runtime reconfigurability (e.g. extensible OSs, active networking nodes, adaptive middleware etc. [Blair,04]). The kernel lies immediately above the (hardware and/or software) *deployment environment*. The kernel is policy free, and its application programmer's interface (API) is target-system and deployment-environment independent. For static systems, it is used only to initially configure the system—when configuration is complete, it can be unloaded so that it does not consume any resources. In dynamic systems, the kernel continues to exist at run time. However, even here its resource demands are minimal as shown in section 6.

Above the kernel is a layer of so-called *extensions* which enhance the basic loading and binding based programming model in accordance with the needs of various target domains and deployment environments. This layer thus plays a central role in providing the tailorability and extensibility that OpenCom aims to deliver. The extensions are independently and optionally deployable and configurable (via the kernel). Importantly, the extensions are themselves implemented as components, so there is no essential boundary between the extensions and target system 'layers' and thus no inherent layering overhead. The extensions that we currently employ fall into the two main classes: First, *platform extensions*, discussed in section 4, provide structured support for tailorability and extensibility at the deployment environment level—essentially, this layer addresses the above-mentioned requirement to efficiently expose unique features of deployment environments in terms of generic component-based abstractions. Second, *reflective extensions*, discussed in section 5, provide generic support for target system reconfiguration—i.e. inspecting, adapting and extending the structure and behaviour of dynamic systems at run time [Maes,87]. These reflective extensions build on and extend inherently reflective features of the kernel such as explicitly represented cross-component bindings and support for extensible meta-data (see section 3). We also provide a set of *security extensions*; these, however, are not as mature as the other extensions and are not discussed further in this paper.

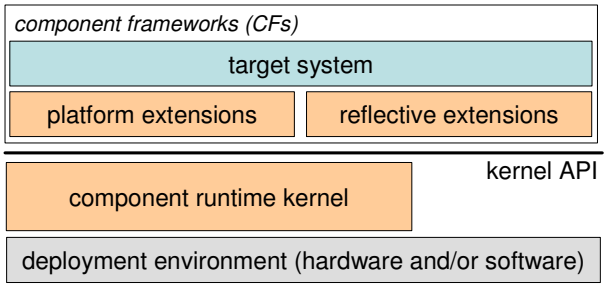

Fig. 1: Overall OpenCom architecture

A key architectural feature of the OpenCom approach is its extensive use of the notion of *component frameworks* [Szyperski,98]. Component frameworks work at a coarser granularity than components, and contribute a generic approach to the structuring and extensibility of software through component composition. In OpenCom, a component framework (hereafter CF) is a tightly-coupled set of components that *i*) cooperates to address some focused area of concern; *ii*) provides a well-defined extension protocol that accepts additional 'plug-in' components that modify or extend the CF's behaviour; and *iii*) constrains [Clarke,01] how these plug-ins may be organised. As an example, we have a protocol stacking CF that accepts protocol components as its plug-ins, and constrains its plug-ins to be composed into linear stacks [Coulson,02].

OpenCom CFs typically employ *run-time* pluggability as well as merely the design-time or build-time pluggability that is found in many component models. For example, a CF may be represented at runtime as a 'root'

component that exports an operation that accepts plug-in components as its arguments. Internally, this root component discovers the interfaces supported by plug-ins (using reflection as explained later), and configures and binds them in a manner appropriate to the CF (e.g. in the above-mentioned protocol stacking CF, the root component would discover and bind the interfaces of its plug-ins to generate a linear stack topology). We have also explored a more sophisticated approach in which CF constraints are expressed at design time in terms of an architecture description language (specifically, ACME [Garlan,00]) and are compiled to generate CF-specific constraint policing code [Joolia,05]. The key point, however, is that different CFs can adopt different approaches to pluggability and constraint. If fact, CFs do not inherently require anything beyond the facilities provided by the foundational component model; i.e. minimal CFs can be viewed simply as architectural patterns.

As will become clear in the remainder of this paper, CFs provide structure and extensibility at all levels of the architecture. At the level of the platform extensions, CFs are provided that, for example, support plug-in 'loader' and 'binder' components. Similarly, at the level of the reflective extensions, plug-ins take the form of, e.g., operation interceptors. At the target-system level, plug-ins are applied in such areas as protocol stacking (as discussed above), thread scheduling, packet forwarding, memory management, or user interaction; and they define plug-ins and constraints that make sense in those domains.

3. The Programming Model and the Kernel API

3.1 Programming Model

A high-level view of the elements of the OpenCom component-based programming model is given in figure 2 (the legends in brackets refer to the example given in section 3.3 below). *Capsules* are containing entities into which components are loaded, instantiated and composed. Each capsule defines a name space for its contained component instances (hereafter, we refer to component instances simply as 'components'), and offers the OpenCom kernel API which is discussed in section 3.2 below. Capsules do not recognise any nesting or hierarchical organisation of their contained components, although such organisations can be conceptually superimposed on this basic 'flat' organisation through the use of ADL-based CFs or other such formalisms.

Components are encapsulated units of functionality which interact with other components in their containing capsule exclusively through so-called *interaction points*, of which there are two types: 'interfaces' and 'receptacles'. *Component types* are templates from which components (instances) can be instantiated at run time. Each component type is defined by a name, the set of the interaction points it supports, and a set of statically-defined <*name, type, value*> properties (this so-called 'static property' facility is used to associate arbitrary meta-data with a component type which is available at run time using the *getprop()* kernel call; see below). *Interfaces* are units of service provision offered by components. Components types may support any number of interfaces including zero (in figure 2, each of the large components has one interface and one receptacle; and the small component has none). The use of multiple interfaces is useful in embodying separations of concern (e.g. between base functionality and component management). Interfaces are defined in terms of sets of operation signatures and associated datatypes. *Receptacles* are 'required interfaces' that make explicit the dependencies of a component on other components. Components may support any number of receptacles. Receptacles are key to supporting the *third-party* mode of deployment and composition inherent in a component-oriented environment: when third-party-deploying a component into a capsule, one knows by looking at its receptacles precisely which other components must be present to satisfy the component's dependencies.

For language independence, we use the OMG's IDL interface definition language [OMG,95] to define interfaces, receptacles and component types. Interfaces and receptacles are expressed using the standard interface definition syntax[1]. Component types, on the other hand, employ the following extended syntax (which is similar to a subset of the OMG 'component' syntax):

<comp_type_defn> ::= "componentType" <comp_name> "{" <static_props> <provides_and_uses> "}"
*<static_props> ::= (<prop_name> ":" <prop_type> "=" <prop_value> ";")**
*<provides_and_uses> ::= (<provides> | <uses>)**
<provides> ::= "provides" <interface_name> ";"
<uses> ::= "uses" <interface_name> ";"

[1] Actually, receptacles do not need to be defined explicitly as each receptacle is implicitly defined in terms of its associated interface.

The 'provides' clause refers to interfaces supported by the component type and the 'uses' clause refers to its receptacles. The elements <*comp_name*>, and <*interface_name*> are strings; the latter refers to the names of IDL interfaces defined elsewhere. Similarly, <*prop_name*> is a string, and <*prop_type*> refers to an IDL data type defined elsewhere; <*prop_value*> is a value of the appropriate <*prop_type*>.

Note that we do *not* provide any facilities at the component type definition level for 'nesting' component definitions, or to specify static bindings between receptacles and interfaces. The aim is to maximise the simplicity of the programming model. Nevertheless, such higher-level facilities can be straightforwardly built on top where required. For example, we have employed the ACME ADL to specify such concerns, and have also experimented with an XML-based formalism [Joolia,05]. However, when such formalisms are employed they always compile down to the basic OpenCom programming model—i.e. a 'flat' component structure in which all receptacle-interface bindings are created at run time.

A *binding* is a run-time association between a single interface and a single receptacle. Interfaces may participate in multiple bindings, whereas each receptacle may only participate in a single binding at a time. Like component deployment, the creation of bindings is inherently *third-party* in nature. That is, bindings can be created by any component within the capsule, not only by the 'first-party' components whose interface or receptacle is actually participating in the binding. Each binding is represented by a component (the small component in figure 2) that is implicitly created by the kernel. The semantics of these components (often called 'binding components') are identical to those of any other component—with the following exception: when such a component is destroyed, the interface/receptacle association that it represents is also destroyed. The OpenCom specification allows kernel implementations to employ binding components that themselves support any number of interfaces and receptacles. Binding components with zero interfaces/receptacles (such as the one shown in the figure) are sufficient for lightweight implementations in resource-poor deployment environment; but less constrained implementations are free to support binding components that have interfaces that, for example, support operations to obtain the identifiers of the bound interface/receptacle, or to insert interceptors. Such facilities, however, are more typically supported by extension binders as discussed in section 4.2.

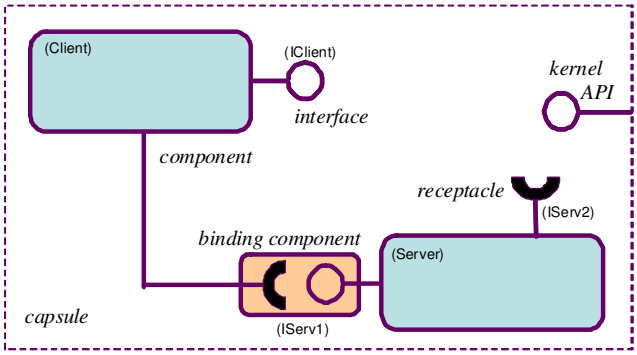

Fig. 2: Elements of the kernel-level programming model

3.2 The Run Time Kernel API

Each capsule embodies a single kernel instance which offers the following run time API[1]:

```
interface Kernel {
    typedef struct template {long id};
    typedef struct component {long id};
    typedef struct interface {long id};
    typedef struct receptacle {long id};

    status load(in string component_type_name, out template t);
    status instantiate(in template t, out component c);
    status unload(in template t);
    status destroy(in component c);
    status bind(in interface i, in receptacle r, out component binding);
    status putprop(in long entity_UID, in string key, in any value);
```

[1] The API, specified here in OMG IDL, has been slightly simplified for presentational purposes.

```
    status getprop(in long entity_UID, in string key, out any value);
    long   register(in long proposed_UID);
    status notify(in ICallback callback);
}
```

This API is deliberately minimal in nature and has been specified on the basis of considerable experience and experimentation with runtime component model APIs over the past few years [Clarke,01], [Coulson,03], [Grace,05]. The basic philosophy is to build the API in terms of two very primitive system-level facilities: *dynamic loading* (in the shape of *load()*, *instantiate()*, *unload()* and *destroy()*); and *dynamic linking* (in the shape of *bind()*)[1]. This API then supports the implementation of all other areas of systems-related functionality (e.g. concurrency, protection, distribution, etc.) as well-defined components that build on the basic kernel-level loading and linking services. The fact that the 'loading and binding layer' is offered as a well-defined component model which is consistently re-used in the higher layers lends great coherence, uniformity and flexibility to the approach, while at the same time allowing the kernel to be implemented in an extremely small, efficient and policy-free manner.

We now discuss the API in detail. The struct definitions are used to name the four types of entities comprehended by the kernel (i.e. templates, components, interfaces and receptacles). Wrapping the UIDs of these entities in structs facilitates language-independent type safety at a cost of only *sizeof(long)* bytes of memory overhead per UID. *Load()* loads a named component type into the capsule and returns a 'template'—i.e. an in-memory representation of a component type (including its executable code) which can subsequently be instantiated using *instantiate()*. Instantiation is separated from loading to assist the user in controlling the trade-off between memory economy and instantiation latency. For example, a programmer may choose to load and instantiate on demand (high instantiation latency, but with the benefit of only incurring memory overhead when a given component is actually required); or alternatively she may choose to pre-load templates so that instances can later be created quickly (low instantiation latency at the expense of having the templates occupy memory between the load and instantiate steps). *Unload()* unloads the specified template (to free up memory), and *destroy()* destroys a component instance. *Unload()* fails if there are extant instances of the target template; and *destroy()* fails if any of the target component's receptacles or interfaces are currently bound. As the kernel itself is modeled as a component which exports its API as an OpenCom interface, *destroy()* can be used *to remove the kernel itself*—this frees up the memory used by the kernel but leaves all components and bindings untouched; the effect is to forego the possibility of making subsequent run-time changes in the capsule. *Bind()* is used to create a binding between a specified receptacle and interface. As bindings are represented by components, *destroy()* is used to remove a binding. The arguments to *bind()* (i.e. an interface and a receptacle of the to-be-bound components) can be obtained from an internal kernel 'registry' which is accessed via the *putprop()* and *getprop()* calls. All components are required on instantiation to call *putprop()* to store in the registry their interfaces and receptacles, using built-in *key* arguments of "I" and "R" respectively. These interfaces and receptacles can then be retrieved, given the target component's identifier, using *getprop()*. Apart from these built-in keys, higher-level CFs and extensions can use the registry facility to attach arbitrary meta-data (using keys that they themselves define) to any component model entity (i.e. templates, components, interfaces or receptacles).

Finally, the purpose of *register()* and *notify()* is to provide specific support to the extensions, as discussed below in sections 4 and 5. *Register()* allocates a UID for any newly-created entity and stores it in the registry. It is used when entities are created by platform extensions rather than by the kernel itself. The role of *notify()* is to assist reflective extensions in obtaining and maintaining information about relevant activity in the capsule, and to serve as the basis of a *policy enforcement point* for security and consistency management purposes. When a callback is registered with *notify()*, every subsequent call on the kernel (i.e. of *load()*, *bind()* etc.) is reported to the callback. More specifically, the callback is invoked *twice* for each kernel call: The first invocation is made *before* the associated kernel call has been made; it reports the 'in' arguments of the kernel call (e.g. the *name* argument to *load()*). The second callback is made *after* the associated kernel call has been made; it reports the 'out' argument values that have come back from the kernel call. It is possible to use this callback facility to 'veto' a kernel call by returning a specific value from the first callback. This prevents the kernel call from being executed and as a consequence prevents the second callback from occurring; an error code is returned to the caller of the kernel call.

---

[1] Having said that, we do obviously provide some additional calls that have been found to be generally useful—i.e., basic facilities to (*i*) manage meta-data associated with component model elements (i.e. *putprop()* and *getprop()*), (ii) manage names (i.e. *register()*), and (iii) reflect on calls being made on the API (i.e. *notify()*).

Despite its minimality and relative ease of realisation (see section 6), the above-described API already provides a powerful and self-contained component-based programming model. It is nevertheless still quite limited in terms of its tailorability and extensibility. For example, it supports only a single (implicit) mechanism for loading and binding components, and it does not provide any specific support for principled reconfiguration beyond the basic capability to dynamically create and destroy components and bindings. The extensions layer described in sections 4 and 5 build on the basic kernel functionality to specifically address such concerns.

3.3 Programming Language Bindings

We have realised the above-described programming model and kernel in Java, C and C++, and in a range of deployment environments (see the case studies in section 7). All the language bindings employ an IDL compiler to generate glue code, and to define, according to standard OMG-defined programming language mappings, the language-specific representations of the four OpenCom entities (i.e. templates, components, interfaces and receptacles), and common data types like ints, strings etc.

Due to its simplicity and accessibility, it is most useful to use the Java programming language to exemplify language binding issues. In the Java binding, components and receptacles are represented as Java classes, and component interfaces are represented as Java interfaces that the component class 'implements'. The IDL compiler generates the receptacle classes and also a per-component class called _<*component_name*> from which the programmer's component implementation class should inherit. This generated class makes available the kernel API operations to its user-defined child class (this is done by transparently pre-binding a per-component receptacle called *kernel* to the kernel interface), encapsulates the declaration and initialisation of the receptacle classes, and uses *putprop()* to register the component type's static properties with the kernel. The child class itself may also, of course, register any further 'dynamic' properties with the kernel as it sees fit.

Given this preamble, it should be straightforward to understand the following simple example which refers to the component topology shown in figure 2. First, we define the interfaces used in the example:

```
interface IServ1 {int op1(int i, int j);}
interface IServ2 {int op2(int i);}
interface IClient {int setup(); int call(char op, int arg1, int arg2);}
```

Next, we define the two component types: the *Server* type provides an *IServ1* interface and has a receptacle for *IServ2*; and the *Client* type provides *IClient* and has a receptacle for *IServ1*:

```
componentType Server {version:int=1; provides IServ1; uses IServ2;}
componentType Client {provides IClient; uses IServ1;}
```

Next, we write Java application code corresponding to these specifications. This might appear as follows:

```
public class Server extends _Server implements IServ1 {
    public Server() { super(); }
    public int op1(int a, int b) {return ...;} /* defined in IServ1 */
}

public class Client extends _Client implements IClient {
    public Binding b;

    public Client() { super(); }
    public int setup() { /* defined in IClient */
        Template t_serv = kernel.load("Server");
        Component serv = kernel.instantiate(t_serv.id);
        OCM_IRefList ilist = (OCM_IRefList)kernel.getprop(serv.id, "I");
        IServ1 i_IServ1 = (IServ1)ilist.getIRef("IServ1");
        kernel.bind(r_IServ1.id, i_IServ1.id, b); /* binding component returned as b */
    }
    public int call(char op, int arg1, int arg2) { /* defined in IClient */
        ...
        int result = r_IServ1.op1(arg1, arg2); /* call the bound receptacle */
        return result;
    }
}
```

Client loads, instantiates and (first-party[1]) binds to *Server* when its *IClient.setup()* operation is called. It subsequently calls *Server.IServ1.op1()* when its *IClient.call()* operation is called.

Much of the machinery is defined behind the scenes in the automatically-generated *_Server* and *_Client* classes. This includes the definition of the receptacle classes (which are conventionally named as *r_<interface_name>*—i.e. *Client.r_IServ1* in our example) and the initialisation of these in the constructor. The constructor also includes generated code that stores the component type's static properties into the kernel, together with the component's interfaces and receptacles (using the 'standard' "I" and "R" keys mentioned above). The mirror-image of this latter mechanism, together with a helper class called *OCM_IRefList*, is employed to obtain the *IServ1* interface from the newly-instantiated *Server* component. Then *bind()* is called to bind the Client's *r_IServ1* receptacle to the just-obtained *i_IServ1* interface. The kernel type-checks the arguments passed to *bind()* using Java reflection.

The C and C++ language bindings follow a similar pattern, but suitably adapted to the target language's capabilities. For example, in the C binding, a component is represented by a .c file that contains implementations of all the component's interfaces' operations. The user must also provide *startup()* and *shutdown()* lifecycle-management functions, and take responsibility for allocating and deallocating within them the memory for receptacles and interface etc. The IDL compiler generates a per-component header file that contains the necessary definitions of receptacles and interfaces. These are represented as structs, with pointer-to-function members representing operations, and a per-type unique identifier (UID) member to help with type checking (the latter is used by the kernel to type-check the receptacle and interface arguments passed to *bind()*). The action of *bind()* is simply to assign the function pointers in the interface struct to those in the receptacle struct, resulting in an extremely minimal and efficient implementation of component binding. Interface operations are represented as user-provided C functions which conventionally take as their first argument the UID of the component instance being invoked. Operation invocations on bound receptacles are realised using per-operation IDL-compiler-generated *CALL_<opname>(<receptacle>, <arglist>)* macros which transparently dereference the given receptacle to determine and call the appropriate target C function with the appropriate first argument.

4. The Platform Extensions

4.1 Overview

The platform extensions augment the basic programming model elements discussed above with new, optional, abstractions and services that play a major role in delivering the tailorability and extensibility promised by OpenCom. The platform extensions are of three kinds, *caplets*, *loaders*, and *binders*, and they are collectively supported by a CF called the *Platform Extensions CF*. Figure 3 visualises caplets, loader and binders as being 'plugged into' the platform extensions CF using an *extend()* operation (see later).

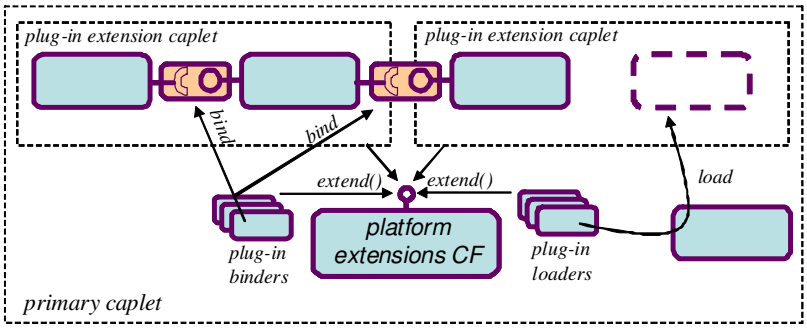

Fig. 3: The programming model extended with caplets, loaders and binders

In brief, *caplets* are specialised plug-in component-support environments that can be dynamically instantiated within a capsule; *loaders* are plug-ins that are responsible for loading components into caplets in various different ways; and *binders* are plug-ins that are responsible for creating bindings both within and across a capsule's caplets in various different ways. In the context of the Platform Extensions CF, the 'original' capsule environment (i.e.,

[1] We provide a complementary example that illustrates third-party binding in section 4.4.2.

before the Platform Extensions CF was loaded) is referred to as the *primary caplet*, and all subsequently-loaded caplets are referred to as *extension caplets*. Similarly, the 'original' loader and binder (which are implicit behind the kernel's *load()* and *bind()* calls) are referred to respectively as the *primary loader* and the *primary binder*, and all subsequently-loaded ones are known as *extension loaders* and *extension binders*. We motivate plug-in caplets, loaders and binders, and discuss them in detail, in section 4.3 below.

As well as hosting these plug-ins, the Platform Extensions CF recognises and embodies an implicit system development methodology that draws a clean distinction between two distinct 'programmer roles' as follows:

- the *deployment environment programmer* (hereafter *environment programmer*) creates suitable caplets, loaders and binders for a particular deployment environment using facilities native to that environment (this relates to the pink areas in figures 2, 3 and 4);
- the *target system programmer* (hereafter *system programmer*) then develops target systems using the APIs described in sections 3 and 4.2, *together with a specific palette of caplets, loaders and binders that has been provided by the environment programmer* (this relates to the blue areas in figures 2, 3 and 4).

Distinguishing these two roles is key to OpenCom's approach of offering a simple and generic programming model in a diverse and extensible environment of target domains and deployment environments. The use of the two roles is discussed in section 4.4; they are further discussed in the context of case studies in section 7.

4.2 The Platform Extensions CF's API

The API offered by the Platform Extensions CF is as follows:

```
interface Platform_Extensions {
    status extend(in extension_type ext_type,
                        in string ext_name, out component ext);
    status load(in string name, in component loader,
                        in component caplet, out template t);
    status instantiate(in template t, out component c);
    status unload(in template t);
    status destroy(in component c);
    status bind(in interface i, in receptacle r,
                        in component binder, out component binding);
    status setdefaultextension(in extension_type ext_type, in component ext);
    status notify(in ICallback callback);
}
```

It will first be observed that the API is similar to, but builds on, the basic kernel API given in section 3. *Extend()* calls on the primary loader to load and instantiate a component that will play the role of an extension. The *ext_type* argument is an enumerated type {CAPLET, LOADER, BINDER} which specifies which one of the three possible extension types is intended; the *ext_name* argument then refers to the name of a component type that will play the corresponding role (i.e. the role of a caplet, a loader or a binder). *Load()* is like the similarly-named operation in the kernel API; the difference is that this version allows a particular loader and caplet to be specified. Likewise, *bind()* allows the specification of a particular binder. Every time a component model element (i.e. a template, component, interface or receptacle) is created, the creating component must record appropriate registry entries pertaining to the element using *kernel.register()* and *kernel.putprop()* (e.g. a loader that creates a new component instance should register the latter and the latter's interaction points). This gives newly-created entities a UID and makes them visible to the rest of the system as if they were created as a result of calls on the kernel API.

Setdefaultextension() is used to designate an extension of a given *extension_type* (i.e. a caplet, loader or binder) as a default to be used in subsequent calls to *load()* and *bind()* if an argument of DEFAULT is passed as the (resp.) caplet, loader and binder arguments of these calls. This 'default' facility is useful in helping to manage the complexity of simultaneously supporting many caplets, loaders, and binders. The basic idea is to employ the 'strategy' pattern [Gamma,04]: *i*) the environment programmer writes a 'meta-loader' or 'meta-binder' that knows the set of loaders (binders) available in a given deployment environment, and the conditions under which each is used (e.g. to deal with specific component types; to load into specific caplets etc.); *ii*) the environment programmer uses *setdefaultextension()* to designate this meta-loader (meta-binder) as the default; and *iii*) when the system programmer calls *load()* (or *bind()*) the meta-loader (meta-binder) dispatches to a specific caplet, loader or binder according to its inbuilt knowledge.

Finally, *notify()* is identical in function to the similarly-named operation in the kernel API. Its use is discussed in section 5.

4.3 Caplets, Loaders and Binders

4.3.1 Caplets

Motivation As mentioned, caplets are specialised component-support environments that can be dynamically instantiated within a capsule. There are *three* main motivations for caplets. The *first* is for different caplets to represent different technology domains in the underlying deployment environment. For example, if it was desired to build a system that comprised both C++ and Java components, this could be achieved by employing a separate caplet for each of the two language environments. In such cases, the caplets might typically be realised as OS processes, with one executing a Java Virtual Machine (JVM). Alternatively, if a deployment environment consisted of multiple bus-connected microcontrollers, each with its own private memory, a caplet could be used to represent each microcontroller/memory pair. The essential difference between supporting multiple caplets in the same capsule and simply employing multiple separate capsules is that in the former case all the contained components, regardless of which caplet they are in, see a common name space and a single kernel API instance. This enables the 'third-party' loading and binding semantic of the kernel to operate transparently across caplets.

The *second* motivation for caplets is to provide privacy and isolation between components that are mutually distrustful, or which have different privileges. For example, when building an OS environment one might implement the OS kernel as one caplet and user space as another (or user space could be represented using multiple caplets, one per process; or caplets could be used to impose protection domains in a single address space). A similar strategy could be adopted in an active networking environment where it was necessary to isolate user-provided functionality from system functionality so that the latter could not crash the former (see, e.g., [Karlin,01]). To manage privacy and isolation, extension caplets can choose whether or not to allow their components access to the kernel: Where required, extension caplets arrange kernel access for their hosted components by providing a kernel proxy, and using a cross-caplet binder to bind this to the kernel interface in the primary caplet. But this arrangement can be selectively disallowed either by the primary caplet or by the extension caplet itself.

The *third* and final motivation for caplets is to support heterogeneous *component styles*—i.e. different implementations of the abstract component concept. The component style supported by the primary caplet is known as the *primary component style*; styles supported by extension caplets are known as *extension component styles*. While the semantics of all component styles must conform to the general characteristics given in section 3 (i.e. support for interfaces and receptacles etc.), each extension component style is free to take its own position on a range of issues such as the following:

- the layout of components on disc and in memory—this may be language/compiler/OS specific;
- whether components can be instantiated multiple times or are singletons (i.e. instantiable only once);
- whether components may support an arbitrary set of interaction points or if these are somehow constrained (e.g. in terms of numbers or types of interfaces, or numbers or types of operations in those interfaces);
- whether components support fixed sets of interaction points or if these can be dynamically created;
- whether components are represented as native executables or as interpreted code (e.g. Java).

There are two major reasons to support extension component styles. The first is to be able to accommodate components written using existing component models (e.g. for purposes of reuse, integration, or backward compatibility). For example, we could integrate Microsoft COM components and JavaBeans in a single system by providing a caplet for each of these component styles, together with a suitable cross-caplet binder. The second reason for supporting extension component styles is to support 'specialised' styles. For example, in a primitive resource-poor deployment environment such as a microcontroller or sensor network element, we could define a minimal component style that imposed severe restrictions on the numbers of interfaces components can support, or the types of arguments that can be passed to operations (e.g. integers only). Nevertheless, such specialised styles still look exactly the same to external third-party code that deploys and binds components in the standard manner supported by the enclosing capsule's kernel API. See section 7.2 for a detailed discussion of such a case.

Realisation From the point of view of the system programmer, a caplet is simply a (primary-style) component that is loaded into the primary caplet (where the Platform Extensions CF itself resides). This primary-style

component encapsulates all the deployment-environment-specific machinery needed to realise its particular instantiation of the caplet concept. The necessary machinery is created and/or initialised when the 'caplet component' is first instantiated (more detail is given below in section 4.4.1). At the environment programmer level, however, there is a basic requirement that caplets must provide a basic communicational facility to enable them to interact with the rest of the deployment environment (e.g. so that loader and binders can work with them). To meet this requirement the primary style component that represents a caplet must implement the following interface:

```
interface Caplet {
    int createchannel(void);
    status destroychannel(in long channel);
    status sendmessage(in long channel, in any message);
    any receivemessage(in long channel);
}
```

In other words, this interface is required for a component to be recognised as a valid caplet by the Platform Extensions CF. The fact that caplets provide an execution environment for components but are otherwise passive is reflected in the form of this interface which, as can be seen, provides only generic message passing services and does not support any caplet-specific functionality. The way in which these message passing services are used by loaders and binders that will be associated with the caplet is discussed below in section 4.4.1.

4.3.2 Loaders

Motivation Extension loaders are used to load components into a capsule (or, more specifically, a caplet) in some particular manner. In many cases, extension loaders are closely associated with particular caplet types. For example, a caplet type that supports a particular component style would typically have an associated loader that knows how to load and instantiate components of this style. However, the concept of pluggable loaders has a much wider applicability than this. In particular, separating the loader and caplet concepts allows one to associate several loaders with a particular caplet type, or to share common loaders across multiple caplet types. It also allows us to provide different loaders with specialised semantics and behaviours. For example, different loaders might get component templates from different places (e.g. from different repositories or over the network). Similarly, loaders might perform security checks on the templates they load and/or instantiate, or validate particular properties, or perform special behaviours on loading/instantiation. As an example of the latter, a loader might use reflection (see section 5) to transparently analyse a component's receptacles when loading it, and then recursively pre-load the full set of components on which it depends; or another loader could load balance across a set of processor/memory units managed within a single caplet.

Realisation As with caplets, a plug-in loader is, to the system programmer, simply a primary-style component that offers a facade that hides arbitrary deployment-environment-specific functionality. To be recognised as a loader by the Platform Extensions CF, loader components are required to implement the following interface, the operations of which are called by the Platform Extensions CF as a result of prior calls of the latter's corresponding calls:

```
interface Loader {
    status load(in string name, in component caplet, out template t);
    status instantiate(in template t, out component c);
    status unload(in template t);
    status destroy(in component c);
}
```

4.3.3 Binders

Motivation The motivation for plug-in binders is to represent different 'binding mechanisms' in the underlying deployment environment. For example, different binders can abstract over binding mechanisms such as interrupts, traps, special buses, shared RAM, optimised register transfers, nearest-neighbour registers in pipeline architectures, or OS-level IPC calls. The plug-in binder abstraction makes all such features uniformly available to the component programmer both within and across caplets, and between components of a common style or different styles. In addition, binders can support special behaviours such as operation interception, performing security checks on invocations, or supporting 'actor' like concurrency models in which thread context switches take place while a thread is executing inside a binding.

Binders can vary widely in their complexity. At the more complex end of the scale, bindings that operate across caplets or component styles may need to incorporate stubs and skeletons to mediate between components

that assume different calling conventions (perhaps because they were generated by different compilers), or which employ different representations of types or different language semantics. In some cases, these stubs and skeletons may be automatically generated by an IDL compiler in the classic 'middleware' style; in other cases they might be hand coded for performance reasons. In all cases, however, the necessary complexity is completely encapsulated within the specific binder and is thus hidden from the user of the generic component-based programming model.

Note that the possibility of cross-caplet binding raises questions concerning the degree of coupling and the distribution of the caplets that comprise a single capsule. The answers are deployment environment specific, but all bindings within an OpenCom capsule are at minimum assumed to be *reliable* in the sense that the semantic of making a call over a cross-caplet binding should be indistinguishable, apart from a slightly greater latency, from that of making a call over an intra-caplet binding. That is, the binding must not drop or reorder any messages. Given this, one would not usually expect the degree of coupling and distribution within a capsule to be so loose and widely-distributed that cross-caplet binders would need to implement complex middleware-like functionality in order to maintain the required degree of reliability. If a target system must operate in such a loose and widely-distributed environment, the preferred approach would be to design the system not as a set of caplets within a single capsule, but as a number of capsules containing middleware-like CFs (as described in the case study of section 7.3).

Realisation Like caplets and loaders, plug-in binders are simply primary-style components that offer a facade that hides arbitrary deployment-environment-specific functionality. To be recognised by the Platform Extensions CF they are required to implement the following interface:

```
interface Binder {
    status bind(in interface i, in receptacle r, out component binding);
    status destroy(in component binding);
}
```

4.4 Programmer Roles

4.4.1 The Environment Programmer Role: Creating Platform Extensions

As mentioned, the Platform Extensions CF sees plug-in caplets, loaders and binders simply as primary style components that support specified interfaces. Behind these interfaces, however, can lurk a great deal of deployment-environment-specific complexity that the environment programmer (but *not* the system programmer) must deal with. For example, consider a 'Java caplet' that encapsulates a JVM and supports a Java-based OpenCom component style. Having been instantiated with a call of *extend()*, such a caplet might proceed by forking a new process to run a JVM, and then establish contact with this new process using some OS-specific IPC mechanism—e.g. a UNIX pipe.

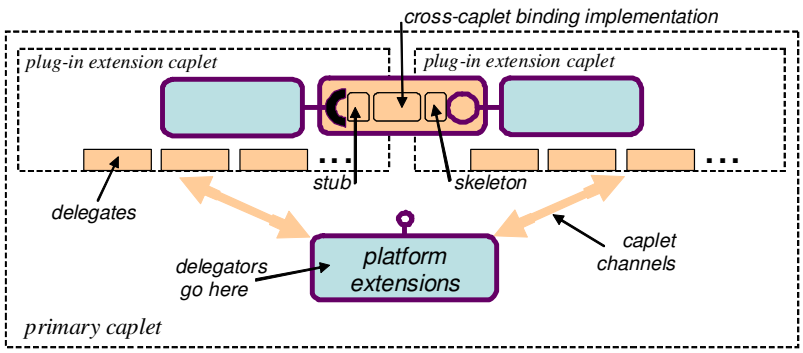

Fig. 4: Environment programmer concepts

In such cases, i.e. whenever there is a physical separation between the primary-style extension component and the software that implements the extension, we typically employ a 'delegator-delegate' pattern in which the extension component plays the role of a *delegator* and the 'separate software' plays the role of a *delegate*. The two players communicate, using some appropriate protocol, over the channels supported by caplets. The use of the delegator-delegate pattern is illustrated in figure 4. (Figure 4 also shows the binder-related environment

programmer concepts of stubs, skeletons and cross-caplet binding implementations—these are dynamically created by plug-in binders as required.)

As a more concrete example of the use of the delegator-delegate pattern, consider the following sequence of steps that might be taken in instantiating a new loader for the above-mentioned Java caplet example:

i) in the primary caplet, the loader component (delegator) is loaded using *extend()*;
ii) this delegator opens a channel to the Java caplet using *createchannel()* in the Java caplet's interface;
iii) the delegator uses a Java-caplet-specific protocol to send (using *sendmessage()*) a command to the Java caplet delegate to ask it to load the loader's delegate (loading the loader's delegate can be done using any appropriate means; as we are in the environment programmer domain here, this is outside the scope of the OpenCom programming model);
iv) the delegator opens a new channel and uses this to communicate, using a loader-specific protocol, with its newly-established delegate.

One would use similar steps to instantiate a binder plug-in for the Java caplet. Cross-caplet binders employ a single delegator and multiple delegates—one in each of the caplets it knows how to deal with. We consider more concrete examples of delegator-delegate based extensions in section 7.2.

4.4.2 The System Programmer Role: Using Platform Extensions

To further appreciate the distinction between the environment and system programmer roles, consider the simple program below in which an system programmer builds on a set of platform extensions that have been provided by an environment programmer. In this program, the system programmer instantiates and third-party binds a primary-style component in the primary caplet to a component in the 'Java caplet' discussed above. The system programmer also employs an extension loader that loads components into the Java caplet (as discussed above), and an environment programmer provided extension binder that binds components across the two caplet types.

```
template jtemp, ptemp;
component jcaplet, jloader, cbinder, jcomp, pcomp, binding;
interface ifaces[N];
receptacle recpts[N];

/* set up the Java caplet with a Java specific loader and binder */
extend(CAPLET, "MyJavaCaplet", jcaplet);
extend(LOADER, "MyJavaLoader", jloader);
extend(BINDER, "MyJavaToPrimaryBinder", cbinder);

/* load and instantiate the components */
load("PrimaryComp1", PRIMARY, PRIMARY, ptemp);
load("JavaComp1", jloader, jcaplet, jtemp);
instantiate(ptemp, pcomp);
instantiate(jtemp, jcomp);

/* obtain interaction points on the components and bind them */
getprop(pcomp.id, "R", recpts);
getprop(jcomp.id, "I", ifaces);
bind(ifaces[0], recpts[0], cbinder, binding);
```

The key point to notice is the transparency and generality of this system programmer code. First, the structure of the code is independent of the underlying caplet structure and caplet types involved: it would be essentially similar if we were dealing with only one caplet, or if the primary caplet ran on a PC and the extension caplet represented a primitive microcontroller attached to the PC by a PCI bus and supported a component style in which components were realised as small segments of machine code (see section 7.2). Second, the code could be executed unchanged with the same effect from within *any* component in the capsule (including one running in the microcontroller caplet cited above). This means that system programmers do not need to know or care in which caplet their components will execute, and that the choice of an optimal target caplet for a newly loaded component can be left to the platform (i.e. to an extension loader or meta-loader).

5. The Reflective Extensions

5.1 Overview

The purpose of the OpenCom reflective extensions is to support the construction of *dynamic* target systems that need to change or evolve during their execution in a controlled and principled manner. To achieve this, the reflective extensions provide generic support for inspecting, adapting and extending the structure and behaviour of systems at run time. They also help to maintain an architectural separation of concerns between *system building* (or 'base-level' programming) and *system configuration and adaptation* (or 'meta-programming'). This distinction is orthogonal to the distinction between the environment and system programmer roles discussed above.

Following Maes [Maes,87], we view the essence of reflection as enabling the inspection and manipulation of 'causally-connected meta-models' of a software system [Maes,87]. Causally-connected meta-models are representations of some aspect of the system under consideration, and they expose a so-called 'meta-interface' through which the representation can be inspected and manipulated. 'Causal connection' means that *i*) run-time changes made to a meta-model (effected via its meta-interface) cause corresponding changes to immediately be reflected in the represented system; and *ii*) changes in the represented system that occur due to some external cause are similarly reflected in the meta-model. A key principle of our approach to reflection support is to provide an *extensible set* of orthogonal meta-models, each of which is optional and can be dynamically loaded when required, and unloaded when no longer required (assuming no dependencies). We have found in our experimentation to date that the meta-models described in the following subsection are particularly useful.

5.2 Example Reflective Meta-models

The Interface Meta-model This provides two related capabilities: *i*) to dynamically discover (at run time) details of the interaction points of a component in terms of their operation signatures; and *ii*) to perform 'dynamic invocations' on dynamically-discovered interfaces. Together, these capabilities enable components to invoke interfaces which were not known to them at compile time (i.e. they need not support the requisite receptacles). Essentially, these capabilities are similar to Java core reflection except that they work at the OpenCom level and are therefore programming language independent. One possible drawback of the interface meta-model is that its dynamic invocation capability allows one to bypass the architectural structure of an OpenCom system in terms of its explicit bindings. But in some circumstances it is important to support such dynamic behaviour. For example, it is particularly useful in supporting generic functionality such as debugging, component database browsing, or generic bridging (cf. the CORBA dynamic invocation interface or DII [OMG,95]). It is also required to support the architecture and interception meta-models discussed below with the necessary typing information.

In implementation, the interface meta-model is realised as a singleton primary-style component that uses the OpenCom kernel's *getprop()* and *putprop()* APIs to store and obtain the interface types associated with a component, and pointers to IDL definitions of these types in an encapsulated IDL repository. It implements dynamic invocation by offering a generic *invoke()* API which is modeled on the CORBA DII interface. The arguments to *invoke()* comprise the target interface and operation, together with a stack of argument values that is built manually by the invoking component on the basis of the run-time IDL type information provided by the meta-model. Internally, the meta-model establishes a binding to the target interface in the normal manner (i.e. using the *bind()* call) so that the called component is unaware of the fact that it is being invoked in an unusual manner.

The Architecture Meta-model This represents the topology of the current set of components within a capsule. It is used primarily to achieve coarse-grained topological inspection, adaptation and extension of the structure of a dynamic target system. For example, in a media-streaming scenario, we have used it to dynamically manage the set of media codecs in use when a mobile PDA migrates between fixed and wireless networks [Blair,04]. This involves first inspecting the underlying component topology to locate the codec component, and then adapting/extending the topology to effect corresponding change (e.g. to replace the codec). The meta-model provides a 'graph-oriented' API in which components are represented as nodes and bindings as arcs. Inspection is carried out by traversing the graph, and adaptation/extension is achieved by adding or removing nodes or arcs (e.g. adding a node results in the deployment of a new component). An example of the use of the architecture meta-model is given in section 7.2.

In implementation, the architecture meta-model is realised as a singleton primary-style component that provides a topological view of the capsule contents based on *i*) the raw component/interface/receptacle data in the kernel's registry; *ii*) the typing information maintained by the interface meta-model; and *iii*) current binding

information as provided by a *notify()* callback (see section 3). When the architecture meta-model is used to adapt/extend the system topology (e.g. by adding a node as above), it effects the necessary changes by using appropriate kernel calls to load components, create bindings etc. as required.

The Interception Meta-model This meta-model, which is a version of probably the most widely explored reflective mechanism in general use [Kon,02], exposes the process of invoking an operation in a component's interface. More specifically, a meta-interface is provided that allows the meta-programmer to insert arbitrary code-elements called *interceptors* within bindings, such that an interceptor is executed whenever an operation is invoked across the binding (more specifically, either before, or after, or both before and after, the invocation). Such an interceptor might, for example, audit the pattern of invocations and their arguments for debugging purposes, or dispatch invocations to an alternative object instance ('hooking'), or insert a security or concurrency control check on an invocation. Interception is especially useful in adaptation scenarios; for example, in the above-mentioned media streaming/mobile computing scenario, an interceptor on a low-level protocol component could be used to monitor and detect the conditions under which a codec should be replaced by means of the architecture meta-model. In addition, interception can be used as a basis for dynamic aspect-oriented programming [Bencomo,05].

A commonly-cited disadvantage of interception is that it incurs an inherent performance overhead whether or not an interceptor is actually installed [Coulson,04]. This is because interception typically requires bindings to support a level of indirection. Our realisation of the interception meta-model uses the plug-in binder concept to sidestep this disadvantage. That is, we provide both interception-capable and non-interception-capable binders and select from these according to requirement—i.e., we choose an interception-capable binder only where we are likely to need interception. If we choose wrongly we can straightforwardly recover by destroying the current binding and rebinding using a different binder. An additional advantage of per-binder interception is that we can provide alternative models of interception that use different underlying implementations offering different trade-offs.

5.3 Controlling Access to Reflective Meta-Models

Reflection is a powerful and general technique, and its use should always be constrained to minimise programmer errors. Our approach to providing such constraint is to limit the set of components that can access the reflective meta-models: in particular, access is typically given to CFs but *not* to their plug-ins. As well as preventing spurious access, this helps ensure that meta-models are accessed only when conditions are 'safe'; for example, a CF might restrict component replacement via the architecture meta-model to situations in which no invocations are currently being made on interaction points owned by the 'old' component. Further, a CF could define a suitable state-transfer protocol to carry-over essential state from the old component to the new one.

6. Performance and Overheads

We now discuss the inherent performance properties and overheads of OpenCom. This section offers a generic treatment; more specific performance evaluations involving measurements of particular systems constructed using the technology are given in section 7.

In assessing inherent overheads, we recognise a key distinction between *in-band* and *out-of-band* execution [Coulson,04]. In-band execution refers to segments of code that are repeatedly executed in the normal course of events and are therefore particularly performance sensitive; out-of-band execution, on the other hand, refers to code segments that are executed only 'occasionally' to the extent that their impact on system performance is negligible. On the basis of this distinction it can be seen that the kernel inherently incurs zero in-band execution overhead—this is because it is only involved in out-of-band operations, *viz*. loading, instantiating and destroying components; and creating and destroying bindings. These operations are typically only invoked when a system is being (re)configured; that is, an established component topology need not make *any* calls on the kernel. Note that this is quite unlike the situation in operating system microkernels, which are unavoidably involved in critical in-band operations (such as interrupt handling or thread scheduling). Note also that the 'out-of-band' characterisation, with the exception of the overhead introduced by interception-capable binders (see below), also applies to the reflective meta-models.

Of course, these observations do not imply that the performance of target systems constructed using OpenCom is entirely unaffected by the use of the technology! In particular, the following overhead-contributing factors are important:

- overheads inherent in the primary and extension component styles used (e.g. per-component memory overhead)
- performance overheads inherent in bindings (whether created by the primary binder or extension binders)
- the granularity of the componentisation of the target system (which, in turn, affects the relative impact of the above two factors).

To gain insight into these overheads we now provide a brief overview of our 'reference' kernel implementation[1] and present some basic measures of its performance and overheads[2]. The reference implementation realises its primary caplet in terms of a standard Linux process (it also runs under Windows). The primary component style is based on the standard ELF executable format emitted by the GNU C++ compiler. The primary loader is based on Linux Shared Objects (or Windows DLLs), and the primary binder uses standard C++ vtables to realise bindings. As well as the primary binder we have two extension binders that work with primary-style components in the primary caplet: the first of these offers an implementation of the interception meta-model; the second optimises away the vtable apparatus by replacing the usual vtable-based indirected call in the caller's code segment with a simple CALL instruction. While this CALL-based extension binder saves some overhead, it can only be used where the component on the receptacle side is a singleton. Note that the CALL-based binding works by modifying the code associated with the receptacle in the caller's code segment. Such code rewriting approaches are sometime dangerous; our modification, however, is small, well-defined and constrained in scope.

Using our reference implementation, we carried out the following measurements of inherent overhead, the results of which are summarised in Table 1:

- *Memory footprint of kernel.* We measured this as 32 Kbytes. This is a modest overhead which should make the kernel deployable in many resource-scarce environments.
- *Memory footprint of a null component.* The memory requirement of a null primary-style component (i.e. one with no interfaces and receptacles and null initialisation/finalisation routines) is 36 bytes. This compares to an overhead of 20 bytes for a single null C++ object. In addition, the per-interface and per-receptacle overhead is 28 bytes (with an additional 5 bytes per operation).
- *Component loading and instantiation time.* The time taken to load a single null primary-style component (averaged over a few million loads) was measured as 9.8μs (compared to 7μs for a Linux Shared Object containing a null C++ object); and the time to instantiate an already-loaded null component was measured as 0.47μs (compared to 0.28μs for a null C++ object). The small overheads here are attributable to the larger file size of the OpenCom component template (due to meta-data etc.), and the slightly more complex instantiation process, including interaction with the kernel registry.
- *Time to create a primary binding.* This was measured as 2.4μs—which is identical to the time required to create a vtable in C++.

Table 1: Summary of Measures of Inherent Overhead

	OpenCom	*Native C++*
Memory footprint of kernel	32 Kbytes	N/A
Memory footprint of null component	36 bytes	20 bytes (null C++ object)
Mem. footprint of receptacle/interface	28 bytes	N/A
Component loading time	9.8μs	7μs (null C++ object)
Component instantiation time	0.47μs	0.28μs (null C++ object)
Time to creating a primary binding	2.4μs	2.4μs (C++ vtable creation)

Next, we measured the time taken to perform invocations across different types of bindings. The details, which are summarised in Table 2, are as follows:

---

1 In addition to our C++-based reference implementation, we have Java and C implementations as discussed in section 3.3.

2 In all experiments, we used a Dell Precision 340 series workstation equipped with 4 x 1.6GHz Pentium processors, 512 MB of RAM, and running Linux Redhat 8.0.

- *Overhead of calls made across a primary binding.* To measure this, we performed a series of invocations across a primary binding involving an interface with a single operation with no arguments and a void return value. We measured 101 x 10^6 calls per second. We then confirmed that, as expected, this is identical to the invocation rate achieved when calling a method in a simple C++ object (which employs vtables in an identical manner). Thus there is *zero* in-band overhead arising from the use of the primary binder.
- *Overhead of calls made across an intercepted binding.* This test used the same interface as above but employed a binding created by our interception-capable extension binder. We used a null 'before' interceptor and measured 5.8 x 10^6 calls per second. Comparing this overhead with that of the primary binder (see above), this represents a slowdown factor of 17.4—thus it is clear that there is a substantial cost incurred for interception. However, to put this into perspective, we repeated the comparison using a target operation with a more typical and representative non-null body—viz. an empty loop of 1000 iterations. This time we measured 0.54 x 10^6 calls per second for the primary binding and 0.5 x 10^6 for the intercepted binding—a slowdown factor of only 1.08. This illustrates that the absolute overhead of calls in still very small. Bear in mind also that, as explained in section 5, an system programmer only needs to incur this cost when interception is actually required.
- *Overhead of calls made across a CALL-based binding.* This test again used the above-described interface but this time employed our CALL-based singleton binder. We measured 198 x 10^6 calls per second, thus revealing that singleton bindings yield a two-fold speed-up over primary (vtable-mediated) bindings. We also confirmed that, as expected, this is identical to the performance achieved when calling a null C function in a simple C program. Again, this indicates *zero* in-band overhead arising from the use of the binder abstraction.

Table 2: Overhead of OpenCom Operation Invocation

	OpenCom calls/sec	*C/C++ calls/sec*
Primary binding	101 x 10^6	101 x 10^6 (C++ method calls)
Intercepted binding	5.8 x 10^6	N/A
Primary binding (non-null op)	0.54 x 10^6	N/A
Intercepted binding (non-null op)	0.5 x 10^6	N/A
CALL-based binding	198 x 10^6	198 x 10^6 (C function calls)

Overall, it can be fairly concluded from the above figures that the basic approach to building component-based systems adds little overhead to that of a traditional programming language based systems development environment.

7. Case Studies

7.1 Overview

Having described and evaluated the basic OpenCom technology, we now report on our experiences with the technology and present two representative and contrasting case studies of the use of the technology to build non-trivial target systems.

In terms of experiences OpenCom has been used by over 25 programmers at several universities, with different skill levels ranging from 3rd year undergraduates to experienced systems researchers. These programmers have applied OpenCom in a diverse range of systems domains including middleware (e.g. OpenORB [Parlavantzas,05] or ReMMoC [Grace,03]), sensor networks (e.g. Gridkit [Hughes,06]), embedded systems (e.g. RUNES [Costa,07]), programmable networking (e.g. NETKIT [Coulson,03]) and overlay networks (Open Overlays [Grace,05]). Some of these systems have been quite large—e.g. OpenORB employs over 40 components and comprises 63,000 lines of code, while ReMMoC employs over 25 components and comprises some 30,000 lines of code. Furthermore, over 10 OpenCom developers have worked on multiple projects in different domains; e.g. in both mobile computing and sensor networks. On the basis of this experience we can report anecdotally that

it is indeed the case that, thanks to OpenCom's uniformity of approach, a developer's prior experience in one domain considerably reduces their time to develop in a second domain.

In terms of case studies, we focus in the following sub-sections on the above-mentioned OpenORB and NETKIT projects. For each case study we discuss in context the application of the two programmer roles (system and environment programmers), and also provide an overall evaluation, including a performance evaluation.

7.2 Case Study 1: Programmable Networking Environments on Network Processors

7.2.1 Introduction

Network processors [NPF,05] are specialised multi-processor devices that process and forward network packets at gigabit speeds. Unlike traditional high-speed routers, network processors (hereafter, NPs) process packets *in software*—primarily for reasons of flexibility and modifiability. They offer an ideal case study for a systems-level software component model because *i*) they are widely acknowledged to be very difficult to program (e.g. they feature specialised hardware and often have no standard OS environment); and *ii*) the software must be extremely efficient to meet the demands for gigabit forwarding speeds. In this section we briefly describe work carried out in our recently-completed NETKIT project [Coulson,03] which explored the use of NPs (specifically, the Intel IXP range of NPs [Intel,04]) in building flexible programmable networking environments.

An Intel IXP-based router, as illustrated in figure 5, consists of an IXP card connected to a host PC (running Linux) via a PCI bus. IXP cards comprise the following: an XScale processor running Linux and serving as a general purpose control processor; an array of so-called *microengine* 'reduced instruction set' (RISC) processors that are attached to each other and to the control processor through a series of buses and a hierarchical DDR/SRAM shared memory architecture; and a set of dedicated hardware elements (not shown) such as network interfaces and a hardware hash unit. The microengines are responsible for 'fast-path' packet forwarding. They execute small code modules, written in C or assembler, that run in a 'bare-iron' environment (i.e. no OS). They have general purpose, although primitive, instruction sets, and also support specialised packet-forwarding functionality such as checksumming, hardware timing and pseudo-random number generation. As well as having access to shared memory, each microengine has a private instruction/data memory called a *microstore*.

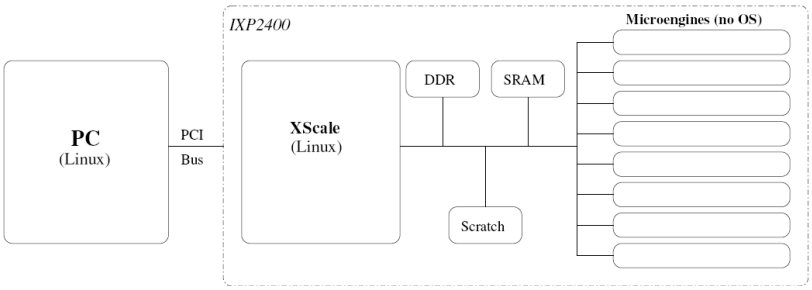

Fig. 5: Outline of an Intel IXP-based router

7.2.2 Programmer Role Considerations

System Programmer Role The system programmer role works entirely in terms of the abstract OpenCom concepts described in sections 3 and 4. Using these concepts, we have developed a 'Network Element' CF, and also a nested Routing CF that takes various routing protocols such as RIP, BGP or OSPF as its plug-ins, and a nested Translator CF that is capable of translating between different versions of IP. These are illustrated in figure 6 which also indicates which of the three environments (i.e. the PC, the XScale or the microengines) that each CF and component resides in. Essentially, the Network Element CF [Coulson,03] defines 'hot spots' in a software router architecture that accept different component-based implementations of network elements such as classifiers, forwarders, schedulers. It is thus closely related to programmable networking systems such as VERA [Karlin,01]. However, the key point is that by building on the set of plug-in caplets, loader and binders provided by the environment programmer (see below), the CF developer is able to largely disregard the underlying complexity and heterogeneity of the IXP architecture. Thus, when the Network Element CF accepts a new forwarder provided by its user, and must bind this to an existing classifier and a scheduler, it does not need to act differently depending on which microengines these components reside on, or what communication mechanism is being used to bind their interfaces and receptacles.

The power and generality of the OpenCom approach is also evident in dynamic reconfiguration scenarios involving the reflective meta-models. Consider, for example, a scenario in which a Network Element CF user dynamically installs an IPv6-to-IPv4 protocol translator. The initial CF configuration, illustrated in figure 6, comprises several components on the router's fast-path, namely a classifier and a forwarder, scheduling components, and a component for processing IP options on the slow-path. These various components are assigned by the Network Element CF to caplets in a way that best exploits the hardware capabilities of the IXP-based router; for example, the CF would typically deploy the fast-path components in the microengine caplets, the IP options component in a XScale caplet, and the Routing CF in an XScale caplet or even a PC caplet.

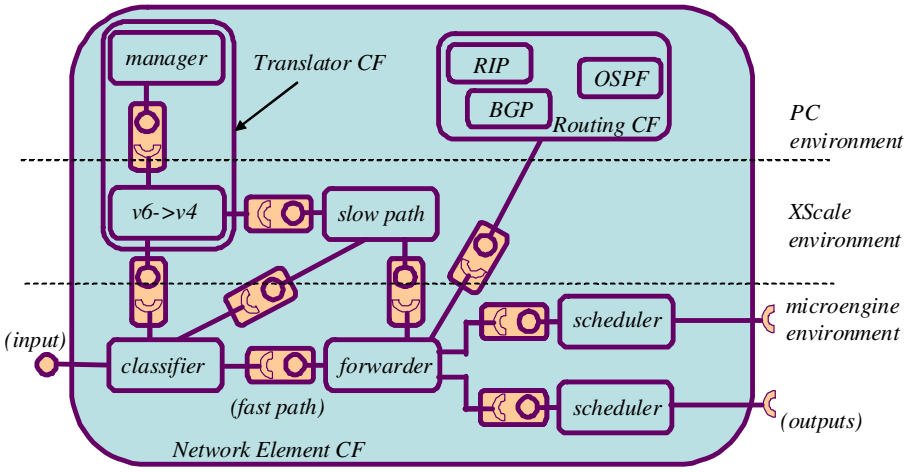

Fig. 6: IXP-based router reconfiguration scenario

Initially, the Translator CF accepts from the user a v6->v46 protocol translator. This is offered to the CF via a CF-specific interface which uses the interface meta-model to check the to-be-loaded component type for conformance to the CF's criteria for plug-ins. The Translator CF then employs a kernel-level meta-loader, as described in section 4.1, to transparently select an appropriate loader for the component type. The selected loader again uses the interface meta-model to check that the to-be-loaded component conforms to its own requirements (e.g. that microengine-hosted components support only single-integer arguments/return values as discussed above). The Translator CF then uses the OpenCom kernel's *getprop()* operation to locate the classifier's receptacle, and, by manipulating a CF-specific API that encapsulates and constrains the architecture meta-model, arranges for this to be bound to the translator. Here, the CF checks that the proposed binding does not violate its rules for structural integrity; having verified this, the CF asks the meta-binder to actually do the work. It could additionally add an interceptor to the binding to, say, monitor and log the number of IPv6 packets actually forwarded.

Environment Programmer Role The first decision to be taken by the OpenCom environment programmer role when deploying OpenCom on an Intel IXP-based router is the scoping of capsules—e.g. should there be separate capsules for the PC, XScale and microengines, or should a single capsule span the entire router? We chose a single all-encompassing capsule on the basis that this promotes the highest degree of integration between the different areas of the router hardware. The next decision is where to locate the capsule's primary caplet. Given that our reference kernel implementation runs under Linux, and that the microengines themselves are too primitive to support the kernel, the choice was between the host PC and the XScale processor. We chose the XScale because it is more 'central'—i.e. in more direct contact with the microengines; but the PC would also have been possible. Next, we must define a suitable set of extension caplets and corresponding component styles, loaders and binders. In the XScale and PC environments the caplets were implemented as standard OS processes, but in the microengine environment we designed a specialised caplet type (see below) that runs on each microengine. Note that other caplet-to-microengine mappings are possible. For example, it would have been possible to encapsulate all the microengines within a single caplet whose loader 'intelligently' assigns components to specific microengines on the basis of their current loading, and perhaps even migrates already-running components between microengines according to some resource management policy.

In the remainder of this section we focus on the above-mentioned microengine caplet, which is of most interest in illustrating the generality of our systems-building approach. The implementation of the microengine caplet itself is relatively straightforward: its only function is to establish a caplet channel from the primary caplet on the XScale

to the target microengine. We realised the caplet channel pattern in terms of libraries provided by Intel that allow one to directly access a microengine's microstore from the XScale processor. Thus no explicit caplet delegate is required. The *component style* employed by the microengine caplet is noteworthy in being strictly and severely constrained: components may have a maximum of *one* interface and *one* receptacle, and these furthermore may only support operations that accept and return a single integer. The main reason for these restrictions is that they enabled us to base the component style on an already-existing module convention adopted by Intel and the Netbind project [Campbell,02]. This considerably eased the task of developing a broad palette of functionality in terms of classifiers, filters, forwarders, schedulers etc. Our microengine component style extends the Intel/Netbind convention by incorporating an on-disc meta-data-enhanced format that supplements the bare executable with pointers to the corresponding IDL interfaces, and other meta-data. We actually designed *two* variants of the microengine component style: The first is a 'basic' version that supports only 'singleton' components. With this version, if we require, say, a pipeline of *n* filter components, the code of the filter needs to be loaded *n* times, which wastes scarce microstore memory. The second, more 'advanced', variant supports multiple instantiation of components. While this is more flexible, it has the corresponding disadvantage of being slightly less efficient. In particular, it assumes a level of indirection in bindings (cf. vtables); also, it involves separating out instance variables and modifying the instructions that access these, and it involves modifying inter-component calls to pass an instance identifier on the stack.

We now discuss the loader and binder plug-ins that we designed for the microengine caplet. The *loader* is of interest because it provides the illusion of dynamic loading/instantiation despite the fact that the microengine hardware only allows modification of its microstore memory when the microengine has been stopped [Intel,04]. The basic capability provided by the hardware is to stop the processor, access microstore locations, and then restart execution at a hard-wired microstore address. To achieve transparent loading it is therefore necessary to not only load the new component, but also to patch the hard-wired restart address so that subsequent execution jumps to the point at which it left off. In addition, to avoid microstore fragmentation when components are being repeatedly loaded and unloaded, the loader needs to occasionally relocate components within the microstore in a transparent manner. The loader is also responsible for enforcing, by inspecting the meta-data attached to an on-disc component template, the above-mentioned component style specific restrictions on interfaces and receptacles.

We have implemented a range of *binders* for the microengine caplet. The simplest of these is an intra-microengine binder for the above-mentioned 'basic' component style. This uses the caplet channel to access the microengine memory, and implements a binding by 'morphing' a jump instruction in the component supporting the receptacle, to refer to the designated entry point of the component supporting the interface (this technique was pioneered in the Netbind project [Campbell,02]). The necessary entry and exit point information is obtained (via the kernel's registry) from meta-data attached to the packaged component, which is transformed by the loader from relative to absolute offsets. The intra-microengine binder for the 'advanced' multi-instantiation component style instead uses the caplet channel to initialise a per-binding indirection table and instance variable vector. We have also implemented a range of *cross-caplet* binders that bind microengine-hosted components to *i*) components in other microengine caplets, *ii*) components in XScale caplets (running under Linux), and *iii*) components in host PC caplets (also running under Linux). These are considerably more complex than the intra-microengine binders discussed above. In particular, the latter two require stubs and skeletons to map the parameter and return values to PCI bus packets. The microengine-side stubs/skeletons are hand coded rather than being generated automatically from IDL (this is another reason for severely constraining the interfaces of microengine components).

Finally, we have implemented a range of loaders for the XScale and PC caplet types, together with binders that perform intra-caplet binding within these caplet types, and a binder that operates between the two. These are relatively straightforward and are implemented using delegators and delegates along the lines discussed in section 4.4.1.

7.2.3 Overall Evaluation

This case study offers a glimpse of the abstraction power of the OpenCom programming model, and also exemplifies the use of domain-specific CFs and reflective meta-models to facilitate reconfiguration and ad-hoc intervention at a high level of abstraction. Note particularly that the steps involved in the reconfiguration need not have been foreseen when the initial configuration was defined, and that they are entirely separate from the basic functionality of the components involved.

The case study is also useful in clarifying the benefits of the component-based programming model in 'taming' hostile and heterogeneous deployment environments such as IXP NPs. In particular, it can be seen that:

- The system programmer is completely shielded from the complex, low-level, heterogeneous and distributed nature of the NP-based router. Instead, she sees only components, loaders, binders, and caplets—all of which are operated on using standard and generic APIs. Even the presence of multiple caplets can be made transparent if the environment programmer has provided suitable meta-loaders/binders.

- Nevertheless, the implementation incurs *no* inherent in-band performance costs, as the component styles used are optimally tailored to their particular environments (e.g. the 'basic' microengine component style is based directly on that provided by Intel[1]), and the binders provided by the environment programmer employ optimal, IXP-specific, mechanisms such as address-patching and the use of nearest-neighbour registers.

This case study also illustrates how the third-party nature of the programming model supports the control and management of the software in the primitive microengine environment. For example, from the primary caplet on the XScale one can incrementally load/unload microengine-based and PC-based components in a completely uniform manner. Similarly, bindings can be uniformly created between a microengine component and another component located anywhere else on the router. The uniformity is most clearly seen when we consider that a microengine component can equally straightforwardly initiate the loading and binding of components on the XScale or the host PC. Furthermore, the reflective meta-models apply uniformly to components in all three caplet types.

To validate the performance of OpenCom in the NP context, we compared the throughput of our Network Element CF with that of standard monolithic software. In both cases an IXP2400 was configured with a simple 'bridging' program that ran on a single microengine and transferred packets between two 2.5 Gbps fibre ports. In the 'standard monolithic' case, the bridge was based on IXP2400 code released by Intel. In our case, the same code was refactored as two pipelined components: a receiver and a sender. We used the 'basic' component style. Subsequently, we experimented with adding additional 'null' components between the receiver and sender, and with using both of the component styles and binding types referred to above (i.e. the 'basic' and 'advanced' types). The throughput in the Intel case was 646.34 Mbps. The results for the OpenCom case are shown in Table 3.

Table 3: Throughput of an OpenCom-based network bridge

Number of components	*'Basic' binder/ component style*	*'Advanced' binder/ component style*
2 (sender/receiver)	646.34 Mbps	634.52 Mbps
5 (i.e. 3 null)	636.51 Mbps	614.51 Mbps
10 (i.e. 8 null)	631.68 Mbps	575.23 Mbps
20 (i.e. 18 null)	613.90 Mbps	442.10 Mbps

Note that the OpenCom case with the two components and the 'basic' binder adds *zero* overhead. Inevitably, overhead is incurred where additional components and/or the 'advanced' component style and binding type are deployed, but this appears to be well within tolerable bounds given the additional functionality. In terms of memory use, the per-component overhead of the null components was as follows: basic component style: 40 bits (i.e. one microengine instruction word); advanced component style: 120 bits (i.e. 3 instruction words). Overall, the results show that it is perfectly feasible in performance terms to use the OpenCom approach to build low-level embedded software such as NP software.

7.3 Case Study 2: Reflective Middleware

7.3.1 Introduction

[1] Obviously there is an additional overhead involved in the use of the 'advanced' multiply-instantiable component style; but given the extra functionality, this is to be expected. The key point is that if one compares like with like—i.e. if one uses singleton components—the overhead is no greater than in the Intel/ Netbind environments.

Middleware is distributed systems software that sits between (distributed) applications and an underlying network and end-system infrastructure. Its role is to shield applications from the complexity and heterogeneity of the underlying infrastructure by providing them with a distributed virtual machine. Due to diversification of both applications and infrastructures, recent years have seen a explosion in the types of middleware being employed. Examples include: traditional object-based middleware such as CORBA; component-based middleware such as Enterprise JavaBeans or the CORBA Component Model; Web Services and Grid middleware; asynchronous middleware based on events, tuples or publish-subscribe; adaptive middleware for distributed real-time systems or mobile computing; embedded systems middleware; and sensor network middleware. This diversity makes it increasingly attractive to approach middleware construction using a framework approach in which tailored platforms can be constructed and customised in terms of a re-usable and extensible set of components and CFs. We have built such a framework, called OpenORB [Coulson,02], [Parlavantzas,05], and have used it to instantiate a number of different middleware platforms as discussed in this section.

7.3.2 Programmer Role Considerations

System Programmer Role The system programmer is responsible for creating the OpenORB framework and populating it with a suitable set of CFs that can be combined by the user to build a variety of configurable and run-time reconfigurable middleware platforms. OpenORB (see figure 7) is structured as a top-level CF ('Top') that is itself composed of three layers called the *Resource Layer*, the *Communications Layer*, and the *Binding Layer*. Each of these layers can itself contain a potentially-extensible set of second-level CFs. The Top CF enforces the three layer structure by ensuring that each plugged-in CF (and its components) only has access to interfaces offered by components in the same or lower layers. Furthermore, it imposes policies concerning layer composition and dynamic changes in layer composition. The plugged-in CFs address more focused sub-domains of middleware functionality (e.g., binding establishment and thread management), and enforce appropriate sub-domain specific policies.

The Resource Layer CFs respectively handle buffers (the Buffer Management CF takes pluggable buffer allocation policies), transport connections (the Transport Management CF takes pluggable transport protocols), and thread management (the Thread Management CF takes pluggable user-level thread scheduling policies). Next, the Communications Layer contains Protocol and Media Stream CFs. The former accepts plug-in protocol components (e.g. GIOP or SOAP) which can be organised into stacks, and the latter accepts software codecs. Finally, the Binding Layer contains a Binding CF [Parlavantzas,03] that accepts *binding type* (BT) implementations (e.g., remote object invocation, streaming connections, publish/subscribe etc.). The Binding CF is the most complex of the CFs and is a crucial part of the OpenORB architecture because it determines the programming model offered to middleware users (e.g. standard CORBA or Web Services).

The OpenORB architecture fosters a considerable degree of component sharing across multiple middleware platforms. In particular, the Resource Layer provides buffer management, transport and thread scheduling components that can potentially be used by many platforms including diverse ones such as CORBA, Web Services and event based middleware. The layer composition policies in the Top CF determine exactly which plug-ins are to be applied. The typical level of re-use, however, tends to diminish in the higher layers.

Overall, the OpenORB implementation consists of about 50,000 lines of C++ divided into five CFs and around thirty components. We have used this to build a standard CORBA environment [Coulson,02], a platform for mobile computing [Grace,03] (which employs a Web Services API and a Dynamic Service Discovery CF in the Binding Layer) and a platform for Grid computing [Grace,05]. Full details of the overall OpenORB approach are available in [Parlavantzas,05].

Environment Programmer Role Unlike the programmable networking case study, OpenORB does not need to employ a large and complex set of caplets, loaders, and binders. This is because (with the exception of embedded and sensor network middleware) the underlying infrastructure tends to be accessed via relatively high-level OS APIs. However, this case study does require the OpenCom kernel to be ported to a range of end-systems running a range of OSs. Because of the size and simplicity of the kernel, this has proved straightforward: we have easily ported the kernel to a range of PCs and PDAs running both Windows (including CE) and a variety of Unix flavours. The main requirement in terms of caplets has been to support both native and interpreted component styles. Therefore, we have Java-based caplets to go with native kernels and vice versa. We employ straightforward OS mechanisms (e.g. Java class loaders and IPC mechanisms such as pipes) to implement suitable loaders and

cross-caplet binders that work within a single machine environment. We then build inter-machine communication in terms of middleware-level communications CFs that include RPC protocols etc.

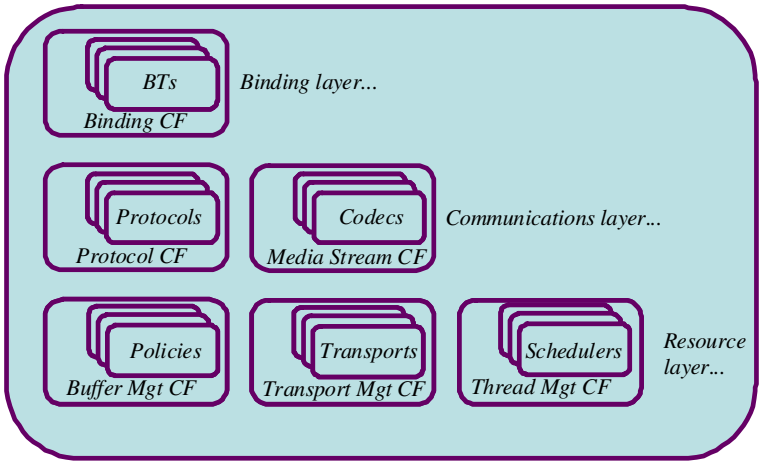

Fig. 7: The generic OpenORB architecture

7.3.3 Overall Evaluation

Unlike the programmable networking case study, the OpenORB case study emphasises the system programmer role more than the environment role; it demonstrates the practicability of using OpenCom to develop a large and complex system which has evolved over a number of years and been worked on by a significant number of people. It also validates the usefulness and practicality of the CF approach to building systems.

In terms of overhead, we have compared the performance of our OpenORB-based CORBA implementation with standard CORBA object request broker (ORB) implementations. The objective here is to evaluate the degree to which componentisation impacts performance. The ORBs chosen for comparison were Orbacus 3.3.4 and GOPI v1.2 (GOPI [Coulson,02a] is a non-component based CORBA implementation that we previously developed). All three ORBs employed CORBA GIOP v1.2. Orbacus is well known as one of the fastest and most mature CORBA-compliant commercial ORBs available and therefore represents the state of the art in CORBA performance. A comparison with GOPI, on the other hand, yields insight into the overhead of componentisation as the GOPI code was extensively re-used in our later component-based OpenORB implementation.

All three ORBs were tested 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 XP and the compiler was Microsoft's cl.exe version 12.00.8804 with flags /MD /W3 /GX /FD /O2. Our tests measured method invocations per second over the PC'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.

The results of timing a large number of round-trip invocations using the above setup are shown in figure 8. It can be seen that for packets smaller 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 bindings). In our test configuration the data path for each GIOP invocation involved 67 bindings, 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. This demonstrates that structuring middleware platforms in terms of dynamically-composable components does not necessarily incur a significant overhead.

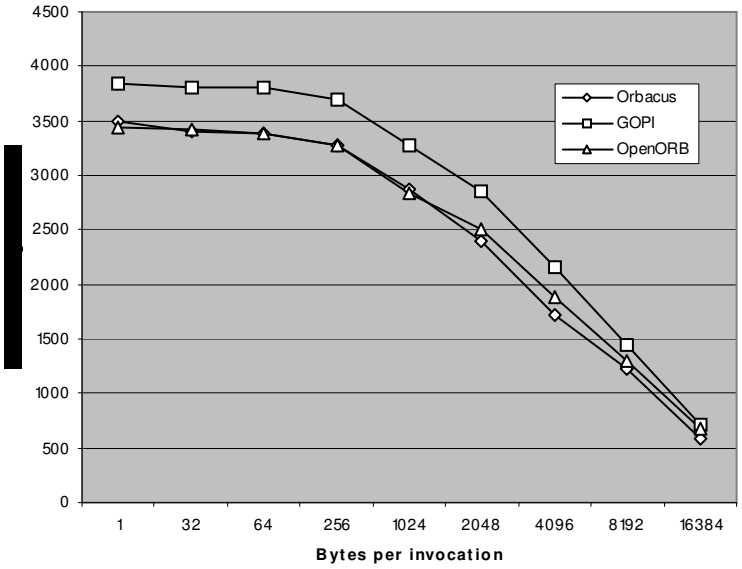

Fig. 8: Performance of OpenORB versus GOPI and Orbacus

Finally, based on our experience with both OpenORB and the non-component-based GOPI, we briefly comment on the relative development benefits of using a generic component model to build middleware. Although GOPI was already structured in a 'modular' fashion, the informality of the module interfaces was, as usual, a limiting factor in fostering reuse and independent development. In contrast, we found that the greater rigor imposed by OpenCom—and especially by the use of receptacles and language independent interfaces—considerably enhanced these factors as well as leading to, if anything, faster development time. In addition, we automatically benefited from the ability to mix programming languages, as well as to reconfigure component configurations dynamically.

8. Related Work

OpenCom can usefully be positioned against *three* areas of related work: i) application-level component models, ii) systems-level component models that are narrowly targeted at specific target domains and/or deployment environments, and iii) generic systems-level component models.

In terms of a comparison with application-level component models, OpenCom substantially differs from designs such as EJB [Sun,05], the CORBA Component Model [OMG,99], Microsoft's COM or .NET [Microsoft,05], and ICENI [Furmento,02], in being far more lightweight. OpenCom's capsule concept is superficially related to the 'container' concept espoused by many of these models, but the latter is far more complex and provides correspondingly richer functionality—e.g. in terms of policy specification for security, event-handling, transactions, and persistence. In contrast, OpenCom capsules—due to our goals of genericity, resource parsimony, and performance—are policy free and embody only minimal, low-level, functionality (i.e., loading-, binding-, and registry-related). OpenCom shares with these models an emphasis on third-party deployability of components. However, whereas software re-use is the primary reason for most of these systems to espouse third-party deployability, the main reason for third-party deployability in OpenCom is to facilitate system configurability and reconfigurability, and to enable primitive components in extension caplets to function as first-class players in the computational model (cf. the programmable networking case study). OpenCom also differs from EJB and ICENI in being language independent.

We now turn to component models which aspire to support the development of low-level systems, but which cannot reasonably be designated as *generic*. Here, we restrict our remarks primarily to arguing why these systems cannot be so designated. In the *embedded systems* area we have a wide range of component models such as Pebble [Magoutis,00], PECOS [Winter,02], PBO [Stewart,93], SaveCCM [Hansson,04] and Koala [Ommering,00]. Most of these are build-time only technologies—components are not visible at run time and therefore these systems do

not support dynamic reconfiguration. One area that some of these systems (i.e. PECOS and PECT) *do* support, however, that OpenCom does not natively support, is the specification (at build-time) of real-time constraints such as cycle time or worst case execution time. Such facilities are clearly important in certain real-time-critical areas. Our approach to providing such facilities, where needed, would be to provide a suitable 'real-time systems' CF rather than building-in real-time properties into the programming model itself. A further observation is that many of these embedded systems technologies (e.g. PBO, SaveCCM, and Koala) are tightly coupled to an specific underlying OS environment and/or are programming language specific. In the *OS building* area we have systems such as THINK [Fassino,02], OSKit [Ford,97] and MMLITE [Helander,98]. THINK has a similar component-based programming model to OpenCom and also employs the notion of plug-in bindings. However, it does not share OpenCom's 'kernel-style' architecture and it lacks any equivalent of OpenCom's other platform extension facilities (i.e. caplets and loaders) and so is unable to abstract over as broad a range of deployment environments. It also lacks OpenCom's reflective capabilities and has so far only been used to develop OSs in standard PC environments. OSKit is a Knit-based toolkit that consists specifically of OS related components. Thus it is Knit (see below) rather than OSKit itself that is the 'generic' model. MMLITE was an early attempt to adapt and apply Microsoft's COM as a vehicle for building operating systems. Although it did support a degree of reconfiguration, MMLITE had no specific support (e.g. in terms of reflection and CFs) to control and manage this; in addition, due to its reliance on COM, it was not suitable for primitive deployment environments. In the *programmable networking environments* area we have a wide range of component systems that might be exemplified by VERA [Karlin,01], MicroACE [Johnson,03], and Netbind [Campbell,02]. Many of these systems are targeted at primitive deployment environments; but they largely achieve this by defining very limited and specific component models. VERA, for example, is more a component framework than a generic component model in that it supports plug-in components only at pre-designated 'hot spots'. Similarly, MicroACE is specifically designed for Intel IXP network processors—its notion of 'component' is inherently 'split' so that a part of a component exists on a microengine and part on the control processor. Apart from the fact that MicroACE does not support run-time reconfiguration, this architecture is clearly inapplicable to other types of deployment environment. Again, Netbind, on which we have built our own programmable networking research (see section 7.2), is barely a component model—it deals with informal 'modules' rather than components and its binding mechanism is intimately tied to the IXP deployment environment. Finally, in the *middleware platforms* area we have systems such as LegORB [Roman,00], k-Components [Dowling,01] and various JavaBeans-based approaches (e.g. [Bruneton,00] and [Joergensen,00]). These systems tend to be flexible in terms of their support for dynamic reconfiguration, but none of them are suitable for deployment environments other than standard PC environments (or at best PDAs running a standard OS such as Windows CE).

Turning now to purportedly generic systems-level component models, we observe that as mentioned in the introduction, only a limited amount of work has so far been carried out on exploiting the component paradigm as a generic systems-building approach. The other main players in the field are Knit [Reid,00] and Fractal [Bruneton,04]. Knit was initially targeted primarily at operating systems (e.g. the OSKit system mentioned above), but it has also been successfully used to build software routers, although only in conventional PC environments. The main limitation of Knit is that it addresses purely *build-time* concerns: the component model is not visible at run time so there is no systematic support for dynamic component loading, still less managed reconfiguration. Fractal is much closer to OpenCom in its goals and approach. Like OpenCom, it supports bindings between interfaces as first class objects, and it takes seriously the need for dynamic reconfiguration. However, although Fractal is designed to be abstract and generic, it seems to lack some of the flexibility of OpenCom, which potentially limits its applicability. In particular, it mandates a specific component style that provides quite rich (and therefore costly) behaviour in several areas—i.e. it supports composite components, has an architecture that describes how components can be built in terms of multiple Java classes (or C files), and its components need to include a 'controller' part for lifecycle management. It is difficult to see how this relatively complex component style could, for example, be deployed in very primitive environments like network processor microengines or sensor motes. Like OpenCom, Fractal supports reflection. This is primarily architectural reflection which employs an XML-based architectural description language that is available at run time. OpenCom, on the other hand, supports an extensible range of reflective facilities, and leaves the choice of specific facilities to the environment programmer.

Finally, it is instructive to briefly compare OpenCom's kernel with the common notion of OS microkernels (see, e.g., [Rashid,89]). While there is an apparent similarity between the architecture diagram in figure 2 and that of a typical microkernel OS, it can easily be shown that the resemblance is only superficial: microkernels are responsible for such in-band tasks as interrupt handling, thread scheduling, message passing and paging. The

OpenCom kernel, on the other hand, only supports simple compositional functionality—i.e. loading and binding. The distinction is perhaps most clearly evident when one considers that it is straightforward to unload the kernel itself to save space (as long as no subsequent system reconfiguration is required).

9. Conclusions

We have presented a generic component-based approach to the construction of systems software, and have argued that this generic approach has significant advantages over the current crop of narrowly-targeted systems-building components models. In particular, we argue that our approach maximises the genericity and abstraction level of the component based programming model while at the same time supporting a principled approach to the support of a wide range of target domains and deployment environments. OpenCom offers a flexible, extensible and language-independent architecture that is based on a minimal component run-time kernel. Due to its simplicity, the OpenCom kernel can easily be deployed in a wide range of deployment environments and used to underpin the construction of both dynamic and static systems (in the latter case the kernel can, as mentioned above, be viewed simply as a highly flexible and configurable loader that is discarded once the system has booted).

On top of the kernel, the platform extensions layer adds the generic abstractions of caplets, loaders and binders. These are instrumental in adapting OpenCom to heterogeneous deployment environments and in 'taming' the idiosyncrasies of the deployment environment without over-abstraction (which would lead to poor performance and lack of accessibility to useful deployment-environment-specific features). As shown in section 7, a wide range of deployment environment idiosyncrasies can be exposed in a consistent and efficient manner through these abstractions. The result is a standard and general approach to system construction and reconfiguration based on a uniform model of third-party loading and binding (this is particularly well illustrated by the programmable networking case study). Another important element of our platform extensions design is the notion of differentiated programmer roles: environment programmers populate a given deployment environment with suitable caplets, loaders, and binders; and system programmers construct the target system in terms of these. This differentiation structures the development of systems and provides a principled approach to the task of bridging the "implementation gap" between an abstract programming model and a concrete deployment environment. Also in the extensions layer, the reflective meta-models provide principled means of reconfiguring systems in terms of inspection, adaptation and extension. Again, it is for the environment programmer to choose which of an (extensible) set of reflective meta-models should be made available in any given installation.

Component styles and component frameworks are two further features of our approach that strongly foster genericity. The ability (through caplets) to support an extensible set of component styles means that the component notion can be instantiated in almost any conceivable programming language and deployment environment (including microengines and sensor motes). Nevertheless, thanks to third-party loading and binding, the idiosyncrasies of different component styles are only visible to the developers who *write* components using some particular style. Other developers who *compose* (and reconfigure) systems in terms of these components can choose to remain oblivious to their internal heterogeneity. Our architecture is similarly agnostic in the area of component frameworks. Unlike, say, Fractal, which supports a particular instantiation of the CF concept (which employs an associated XML-based architecture description language), OpenCom fosters generality by leaving the CF notion loosely specified as an architectural 'pattern'. Nevertheless, this does not preclude the use of specific 'meta-frameworks' that can be used as optional extensions to generate CFs. For example, we have developed a meta-framework called 'Plastik' that employs an ADL to formally specify constraints on OpenCom CFs and automatically generates checking code to police these constraints at run time [Joolia,05].

In our present and future work we are continuing both to develop OpenCom and to evaluate it by building target systems in a wide variety of target domains and deployment environments. For example, a current EU-funded collaborative project is focusing on the use of an OpenCom-like component model in the development of real-time and embedded systems [Costa,07]. This project is also investigating the provision of fundamental support for security in OpenCom-like systems. The approach we are taking here is to provide a 'security mediator' in the extensions layer that offers a minimal Trusted Computing Base, and then to provide higher level mechanisms and policies in terms of CFs. We are also working on an extension of the above-mentioned Plastik meta-framework CF for the specification and policing of real-time properties. Finally, we are investigating the potential of aspect oriented systems development techniques in OpenCom-based real-time embedded environments as a more design-oriented approach to complex system configuration and reconfiguration.

Acknowledgements

The authors would like to acknowledge the UK EPSRC for funding the vast majority of this research. We would also like to gratefully acknowledge our students over the past few years who have contributed in numerous ways: in particular, Nelly Bencomo, Michael Clarke, Anita Cleetus, Nikos Parlavantzas, Rajiv Ramdhany and Nirmal Weerasinghe. Finally, we would like to acknowledge our colleagues in the EU-funded RUNES project (IST-004536) who contributed ideas in the later stages of the development of OpenCom. RUNES is currently developing a component-based middleware infrastructure for networked embedded systems that picks up and develops many of the concepts pioneered in OpenCom.

The OpenCom software is available for download at: http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/download.php.

References

[Bencomo,05] Bencomo N., Blair G., Coulson G., Grace P., Rashid A., "Reflection and Aspects Meet Again: Runtime Reflective Mechanisms for Dynamic Aspects", Proc. First Middleware '05 Workshop on Aspect Oriented Middleware Development (AOMD 05), Grenoble, France, 2005.

[Blair,04] Blair, G., Coulson, G., Grace, P., "Research Directions in Reflective Middleware: the Lancaster Experience", Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004) co-located with Middleware 2004, Toronto, Ontario, Canada, Monday, October 18th, 2004.

[Bruneton,00] Bruneton, E., Riveill, M., "JavaPod: an Adaptable and Extensible Component Platform", Proc. Reflective Middleware 2000, New York, 2000.

[Bruneton,04] Bruneton, E., Coupaye, T., Leclerc, M., Quema, V., Stefani, J-B, "An Open Component Model and its Support in Java", 7th Intl Symp. on Component-Based Software Engineering (ICSE-CBSE7), Edinburgh, Scotland, May 2004.

[Campbell,02] Campbell, A.T., Chou, S., Kounavis, M.E., Stachtos, V.D., and Vicente, J.B., "NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers", 5th IEEE International Conference on Open Architectures and Network Programming (OPENARCH' 02), June 2002.

[Clarke,01] Clarke, M., Blair, G.S., Coulson, G., "An Efficient Component Model for the Construction of Adaptive Middleware", IFIP/ACM Middleware 2001, Heidelberg, Nov 2001.

[Costa,07] Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G.P., Sivaharan, T., Weerasinghe, N., Zachariadis, S., "The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario", Proc. 5th Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'07), White Plains, New York, 19-23 March 2007.

[Coulson,02] Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", ACM Distributed Computing Journal, Vol 15, No 2, pp 109-126, April 2002.

[Coulson,02a] Coulson, G., Baichoo, S., Moonian, O., "A Retrospective on the Design of the GOPI Middleware Platform", ACM Multimedia Journal, Vol 8, No 3, pp 340-352, Dec 2002.

[Coulson,03] Coulson, G., Blair, G.S., Hutchison, D., Joolia, A., Lee, K., Ueyama, J., Gomes, A.T., Ye, Y., "NETKIT: A Software Component-Based Approach to Programmable Networking", ACM SIGCOMM Computer Communications Review (CCR), Vol 33, No 5, pp 55-66, October 2003.

[Coulson,04] Coulson, G., Blair, G.S., Grace, P., "On the Performance of Reflective Systems Software", Proc. International Workshop on Middleware Performance (MP 2004), April, 2004, Phoenix, Arizona; Satellite workshop of the IEEE International Performance, Computing and Communications Conference (IPCCC 2004), 2004.

[Dowling,01] Dowling, J., Cahill, V., "The K-Component Architecture Meta-Model for Self-Adaptive Software", Proc. Reflection 2001, LNCS 2192, 2001.

[Emmerich,02] Emmerich, W., "Distributed Component Technologies and their Software Engineering Implications", Proc. of the 24th International Conference on Software Engineering, Orlando, Florida, pp. 537-546, 2002.

[Fassino,02] Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., "THINK: A Software Framework for Component-based Operating System Kernels", Usenix Annual Technical Conference, Monterey (USA), June 10th-15th, 2002.

[Ford,97] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O., The Flux OSKit: A Substrate for Kernel and Language Research, Proc16th ACM Symposium on Operating Systems Principles, Saint Malo, France, ISBN 0-89791-916-5, pp38-51, ACM Press, 1997.

[Furmento,02] Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J., "ICENI: Optimisation of Component Applications within a Grid Environment", Journal of Parallel Computing, Vol 28 No 12, pp :1753-1772, 2002.

[Garlan,00] Garlan, D., Monroe, R.T., Wile, D., "Acme: Architectural Description of Component-Based Systems", in Foundations of Component-Based Systems, Leavens, G.T., and Sitaraman, M. (eds), Cambridge University Press, pp 47-68, 2000.

[Grace,03] Grace, P., Blair, G.S., Samuel, S., "ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability", Proc. Intl. Symposium on Distributed Objects and Applications (DOA 2003), Catania, Sicily, Italy, Nov 2003.

[Grace,05] Grace, P., Coulson, G., Blair, G.S., Porter, B., "Deep Middleware for the Divergent Grid", Proc. IFIP/ACM/USENIX Middleware 2005, Nov 2005.

[Hansson,04] Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M., SaveCCM -- a component model for safety-critical real-time systems, Euromicro Conference, Special Session on Component Models for Dependable Systems, IEEE, Sept. 2004.

[Helander,98] Helander, J., Forin, A., "MMLite: A Highly Componentized System Architecture", 8th ACM SIGOPS European Workshop, pp96-103, Sintra, Portugal, Sept 1998.

[Hughes,06] Hughes, D., Greenwood, P., Blair, G., Coulson, G., Pappenberger, F., Paul Smith, P., Keith Beven, K., "An Intelligent and Adaptable Grid-based Flood Monitoring and Warning System", Proc. UK eScience All Hands Meeting, 2006.

[Intel,04] Intel IXP1200/2400 Network Processors; http://www.intel.com/IXA.

[Joergensen,00] Joergensen, B.N., Truyen, E., Matthijs, F., and Joosen, W., "Customization of Object Request Brokers by Application Specific Policies", IFIP Middleware 2000, New York, April, 2000.

[Johnson,03] Erik J. Johnson, Aaron R. Kunze, "IXP2400/2800 Programming: The Complete Microengine Coding Guide", ISBN 097178616X, Intel Press, May 2003.

[Joolia,05] Joolia, A., Batista, T., Coulson, G., Tadeu A., "Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform", to appear Proc. 5th Working IEEE/IFIP Conference of Software Architecture (WICSA 5), Pittsburgh, Pennsylvania, USA, November, 2005.

[Karlin,01] Karlin, S., Peterson, L., "VERA: An Extensible Router Architecture", Proc. IEEE Conf. on Open Architectures and Network Programming, OPENARCH 2001, Anchorage, Alaska, pp 3-14, April 2001.

[Kon,02] Kon, F., Costa, F., Campbell, R., and Blair, G., "The Case for Reflective Middleware", Communications of the ACM, Vol 45, No 6, pp 33-38, June 2002.

[Maes,87] Maes, P., "Concepts and Experiments in Computational Reflection", Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, ACM Press, pp 147-155, 1987.

[Magoutis,00] Magoutis, K., Brustoloni, J.C., Gabber, E., Ng, W.T., Silberschatz, A., "Building Appliances out of Reusable Components using Pebble", Proc. SIGOPS European Workshop 2000, Kolding, Denmark, pp 211-216, Sept 2000.

[Microsoft,05] Microsoft, .Net Home Page, http://www.microsoft.com/net, 2005.

[Mozilla,05] Mozilla Organization, XPCOM project, http://www.mozilla.org/projects/xpcom, 2005.

[NPF,05] http://www.npforum.org/, 2005.

[NPF,05] Network Processing Forum, http://www.npforum.org/, 2005.

[OMG,95] Object Management Group, "The Common Object Request Broker: Architecture and Specification", 2.0 edition, July 1995.

[OMG,99] Object Management Group, CORBA Components Final Submission, OMG Document orbos/99-02-05, 1999.

[Ommering,00] van Ommering, R., van der Linden, F., Kramer, J., Magee, J., "The Koala Component Model for Consumer Electronics Software" IEEE Computer, Vol 33, No 3, pp78-85, ISSN 0018-9162, March 2000.

[Parlavantzas,03] Parlavantzas, N., Coulson, G., Blair, G.S., "An Extensible Binding Framework for Component-Based Middleware", Proc. 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), Brisbane, Australia, September 16-19, 2003.

[Parlavantzas,05] Parlavantzas, N., "Constructing Modifiable Middleware with Component Frameworks", PhD Thesis, Computing Dept., Lancaster University, 2005.

[Reid,00] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E., Knit: Component Composition for Systems Software, Proc. 4th Operating Systems Design and Implementation (OSDI}, pp 47-360, Oct 2000.

[Roman,00] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., "LegORB", Proc. IFIP/ACM Middleware'2000 Workshop on Reflective Middleware, IBM Palisades Executive Conference Center, NY, April 2000.

[Stewart,93] Stewart, D., Volpe, R., Khosla, P., Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects, Robotics Institute, Carnegie Mellon University report CMU-RI-TR-93-11, July 1993.

[Sun,05] Sun Microsystems, Enterprise JavaBeans Specification, http://java.sun.com/products/ejb/index.html, 2005

[Szyperski,98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[Winter,02] Winter, M., Genbler, T., Christoph, A., Nierstrasz, O., Ducasse, S,. Wuyts, R., Arevalo, G., Muller, P., and Stich, C., Schonhage, B., Components for embedded software: the PECOS approach, Proc. 2002 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '02), Grenoble, France, ACM Press, pp 19-26, 2002.

[Rashid,89] Rashid, R., Baron, B., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R., "Mach: A Foundation for Open Systems", Proc 2nd Workshop on Workstation Operating Systems (WWOS2), September 1989.

[Gamma,04] Gamma, E., Helm, R., Johnson, R., Vissides, J., "Design Patterns: Elements of Reusable Object-oriented Software", Addison Wesley, ISBN: 0201633612, 1994.