

Component-based System Software: A Generic Approach

Jó Ueyama¹, Francois Taiani², Geoff Coulson², Edmundo R. M. Madeira¹, Paul Grace²

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Avenida Albert Einstein, 1251 Caixa Postal 6176
13083-970 Campinas, SP - Brasil

²Computing Department
InfoLab21 South Drive Lancaster University
Lancaster LA1 4WA UK.

joueyama@ic.unicamp.br

Resumo. *A tecnologia de componentes é comumente utilizada para construir softwares no nível de aplicações. No entanto, essa tecnologia é menos utilizada para a construção de softwares no nível de sistemas. Além disso, a grande parte dos modelos de componentes para o nível de sistemas é destinada à construção de software de sistemas para um determinado domínio e plataforma. Este artigo apresenta o OpenCom, um modelo genérico de componentes destinado à construção de software de sistemas para uma variedade de domínios (por exemplo, middlewares, sistemas embutidos, SOs) e plataformas (por exemplo, PCs, sensores, roteadores). O artigo descreve o modelo e o uso do OpenCom em diversos estudos de caso, o que demonstra a sua flexibilidade e a generalidade desta tecnologia.*

Abstract. *Component-based software engineering has recently emerged as a promising solution to the development of system-level software. Unfortunately, current approaches are limited to specific platforms and domains. This lack of generality is particularly problematic as it prevents knowledge sharing and generally increases development costs. In this paper we present OpenCom, a generic component-based platform that is specifically designed to support a wide range of system software, both in terms of deployment environments (e.g. PDAs, embedded devices, network processor-based routers) and target domains (e.g. embedded systems, middleware, OSs, programmable networking environments). We discuss the fundamentals of OpenCom's programming model, present a performance evaluation, and illustrate the advantages of our model based on several case studies.*

1. Introduction

Component-based technologies are nowadays widely used to develop application-level software, as illustrated by the numerous component-based platforms available to application developers (e.g. Mozilla plugins, Enterprise Java Beans, .NET) [17, 15, 24].

Building on this success, a number of approaches have recently been proposed to apply component-based programming to system-level software. These approaches cover a wide range of systems, from embedded devices [8, 20] and Operating Systems [6, 9], through to programmable networks [16, 14, 2] and middleware platforms [5, 22].

These early researches have shown that components are well-suited for system software for two key reasons: first, they provide a high degree of modularisation, thus allowing complex system software to be divided into simple and easy-to-develop functional parts. This ‘divide and conquer’ approach facilitates the design, development and debugging of complex system software. Secondly, because of their inherent modularity, component-based platforms tend to be much more portable than monolithic systems. This is particularly advantageous for system software, whose portability is often inherently difficult to achieve and can provide a key competitive advantage in terms of reuse and software quality.

Unfortunately, these efforts on component-based system software tend to be highly non-generic. They are limited both in terms of *target domain* (e.g. embedded systems, middlewares, operating systems or programmable networking environments) and *deployment environment* (e.g. standard PC environments, network processors or micro-controllers). For example, OSKit [7], THINK [6], and MMLite [9] exclusively target component-based operating systems and cannot be used to realise programmable networking software or to implement a distributed middleware. Similarly, most approaches can only be deployed on conventional desktop machines, as opposed to less conventional environments such as PDAs and embedded devices. Those supporting embedded devices typically only support one type: Vera [14], NetBind [2] and NP-Click for instance are limited to the Intel IXP family of routers [11].

Because of this narrow focus, existing component models fail to support systems that span multiple deployment environments, and cannot be used to develop platforms that combine multiple target domains. As a consequence, different environments and different target domains require different programming models and different APIs. This lack of generality inhibits skill transfer, prevents component reuse across platforms, and generally drives development costs up.

In this paper, we propose a new component model called **OpenCom** that addresses the above deficiencies. Unlike our previous research (i.e. **OpenCOM** which was targeted to middleware domain), this work gives an infrastructure to construct system software targeted to a wider range of system domains including OSs, middlewares and network stacks. In addition, **OpenCom** gives the support to develop system software aimed at a variety of deployment environments (PC, routers, sensor nodes) at a minimal cost. More precisely our component model has the following three key properties:

- *Target system independence.* OpenCom does not recommend any policy or facility that is specific to a particular target system (e.g. real-time support or media-streaming support). When required, such features can be seamlessly added through a clear and principled extension mechanism.
- *Independence of the deployment environment.* OpenCom’s programming model can be easily ported to wide range of deployment environments, from standard PCs, set-top boxes, and resource-poor PDAs, through to embedded systems with no OS support, and high speed network processor-based routers. This portability results from an incremental design based on a core set of minimal features. This minimality ensures this core can be accommodated even within the most constrained devices.
- *Negligible overhead.* As our prototype evaluation shows, OpenCom incurs neg-

ligible performance overhead and has a very small memory footprint. This is particularly vital for deeply embedded software that runs on highly-constrained devices.

The remainder of this article is organised as follows: we first provide some background on component technology and discuss the corresponding research challenges (Section 2). We then move on to present OpenCom's architecture and discuss its fundamental programming concepts (Section 3). In Section 4 we present a detailed performance evaluation of OpenCOM based on a prototype that combines a standard PC and a network-processor board. We then discuss three case studies that demonstrate the genericity of our approach (Section 5). Finally, we review related work (Section 6) and conclude (Section 7).

2. Component Technology

This section introduces the main notions behind component-based software development. We start with basic concepts (components, interfaces, receptacles, bindings, and component frameworks), which serve as a key structuring mechanism in OpenCom.

As we go along, we will use this conceptual overview to discuss the research challenges pertaining to the use of components in system software. We wrap up this section by outlining the key differences between our previous work and the new general-purpose component model.

2.1. Components, Interfaces, Receptacles, Bindings, and Component Frameworks

OpenCom is a component-based programming framework and as such shares the same basic common concepts as most component programming models [25].

Components are central to these models and denote encapsulated units of deployment and functionality that interact with the outside world through *interfaces* and *receptacles*. Note that a component may have multiple interfaces and receptacles.

Interfaces are units of service provision which are supported by components. They define sets of operation signatures and their associated datatypes. In OpenCom, interfaces are expressed in OMG IDL to support language independence.

Receptacles are 'required interfaces' that define a unit of service requirement. Receptacles make explicit the dependency of one component on other components. Interfaces and receptacles are collectively termed as *interaction points*.

The association between a single interface and a single receptacle is represented as a *binding*. Different binding types are often identified, depending on the semantic of the association, and the particular interaction protocol used between the two service points.

Component frameworks (hereafter CFs) have evolved out of the basic notions of components, interfaces, receptacles and bindings as a structuring mechanisms to develop complex component-based applications. A Component framework is usually defined as "collections of rules and interfaces that govern the interaction of a set of components 'plugged into' them" [25]. A CF embodies rules, constrains, and interfaces that make sense in a specific application domain.

2.2. Research Challenges

A number of component technologies have been proposed in recent years to develop system-level software. Although seminal, these works suffer from a number of deficiencies, which we discuss here in more details.

First, these technologies tend to be tightly coupled to *a particular target domain* (e.g. embedded systems, programmable networking environments, or middleware platforms). They are optimised for the construction of a particular family of systems and usually focus on one particular challenge: maximising performance, minimising memory requirements, allowing dynamic reconfigurations. NetBind [2], for instance, is limited to programmable networks development and focuses essentially on dynamic data-paths construction.

Similarly, these technologies tend to support *one particular deployment environment* only. The PECOS component model for instance [26] is limited to field devices; VERA [14] and NetBind [2] focus on network processor-based routers. More crucially, most of the remaining approaches have so far only been deployed in a conventional PC-based environment (e.g. K-Component [5], Click [16]). In addition, some of them (e.g. Koala [20], and VERA [14]) strongly depend on the underlying operating system and thus cannot be easily ported to other environments.

In terms of adaptation, many of these approaches do not support *run-time reconfiguration*. This is particularly inhibiting in embedded mobile devices where resources are particularly constrained and resources availability evolve dynamically. This is also a strong limitation for long-running systems where unplanned functionalities often need to be inserted without any service interruption (e.g. the introduction of a new forwarder in a programmable router).

These technologies also have varying *memory footprint overhead* depending on the target system they were aimed at. This variability further restricts reuse and technology transfers. PC-based component models for instance tend to be rich in features (e.g. reflection, a large number of binding types). As a consequence, they have quite a high memory overhead and cannot be deployed in constrained environments [5]. Conversely, components models that target constrained devices tend lack the rich features that are expected in unconstrained environments and cannot be reused in a PC-based context.

In this paper, we propose to address these challenges with OpenCom, a generic component model that provides the necessary infrastructure to construct a wide range of systems for a variety of deployment environments. Thanks to its incremental design, OpenCom is inherently extensible to accommodate domain-specific facilities in a natural and explicitly-supported way. OpenCom also supports run-time reconfiguration, as illustrated with our case studies. Last, but not least, because of its minimalist core, OpenCOM incurs minimal overheads as shown in our performance evaluation section.

2.3. Discussion: OpenCOM versus OpenCom

It is important to highlight that Lancaster OpenCOM [4] refers to a previous component model targeted at middleware platforms, while OpenCom concerns our new general-purpose component model. The former, i.e., OpenCOM is built on top of Microsoft COM and has been successfully applied to constructing re-configurable middleware platforms.

However, the main limitation with OpenCOM is its inability to construct systems for a heterogeneous environment. For example, OpenCOM is unable to construct systems in several embedded devices because these devices often consist of multiple deployment environments, each of which requiring its own mechanism to deploy components. As an illustration of this, consider the IXP1200 router platform (see Figure 3) that consists of PC, StrongARM and microengine environments. OpenCOM is unable to deploy components in such a heterogeneous environment in a natural and explicitly-supported way. Another limitation which was identified in OpenCOM is the high memory footprint, which hinders its deployment in resource-constrained devices. The high memory overhead comes primarily from the Microsoft COM component model that is needed in OpenCOM.

On the other hand, OpenCom is designed to address the above mentioned shortcomings, and thus, it is aimed at constructing systems in a way that is independent of the deployment environment (e.g. heterogeneous environments like the IXP1200 routers which are resource constrained devices) and also aimed at constructing systems for a wide range of system domains (e.g. middlewares and operating systems). As a result, OpenCom consists of a more comprehensive programming model that is particularly targeted at overcoming the heterogeneity that is often found in embedded devices.

3. A General-Purpose Component Model

3.1. Architectural Overview

OpenCom uses an incremental design based on three key elements: (i) a *run-time kernel*, (ii) an *extension framework*, and (iii) two *reflective meta-models* (Figure 1).

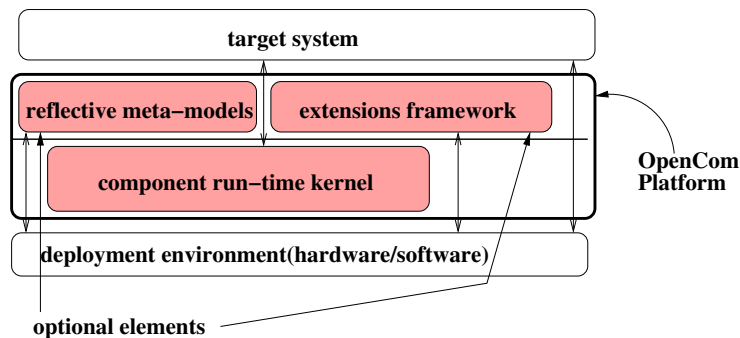


Figure 1. Overall OpenCom architecture

OpenCom's run-time kernel provides a core set of primitives that support a minimal yet extensible component-based programming model. All the remaining layers (the reflective meta-models, the extensions framework and the target system layers) extend the kernel and are implemented using OpenCom's components model, i.e. they are deployed and managed by the kernel. The arrows in Figure 1 indicate the possible interactions between the layers (e.g. the *target systems* is allowed to invoke functions implemented in the *component run-time kernel* layer).

In the remaining of this section we give further details of each element, starting with the kernel.

3.2. The OpenCom Kernel

The OpenCom kernel supports the life-cycle of components, i.e., the Kernel API provides primitives to load, unload, bind and unbind components. As explained above, the kernel is only capable of deploying a particular style of component called the *primary-style* component. In our prototype implementation, this primary style is the XPCOM component format [17]. This enable us to reuse any XPCOM component.

To enable the deployment of OpenCom in resource-constrained devices, the kernel follows a *microkernel-style* architecture, i.e., it only implements the minimal and required functionalities. It only supports the basic service of loading and binding components (through *load()*, *instantiate()* and *bind()*), since these functionalities are required by systems targeted at any domain and environment. This minimality also ensures that the kernel can easily be ported to many deployment environments.

The kernel can only deploy components in a single address space. The deployment of components in environments with multiple address spaces is only possible through the *extensions framework* which is covered in detail in the next section. By factoring out this functionality, we can make the OpenCom kernel lightweight and deployable in resource-constrained devices (e.g. a sensor node) where the support of multiple address spaces is often not required.

In order to support dynamic reconfiguration and reflective extensions, the kernel is capable of loading and binding components at runtime when they are required and of unloading and destroying them when no longer demanded.

In terms of component life-cycle, *load()* reads component meta-data from the library in which the requested component is packaged, and *instantiate()* utilises this meta-data to create an instance of this component. Component meta-data is kept in a repository called the kernel *registry*. The kernel *registry* stores the meta-data of all components, interfaces, receptacles and bindings.

An Interface is bound to a receptacle using the *bind()* operation. This operation enables the construction of systems by selecting components and connecting them up according to the needs of the target system. The disconnection between components is effected through the *destroy()* operation.

3.3. Reflective Meta-Models

As explained earlier, the *reflective meta-models* layer is itself a component framework and therefore an OpenCom component that is deployed (when needed) by the kernel. It is optional and implements the necessary functionality to inspect, adapt and extend the target system at runtime. In addition, since the meta-models are themselves components, their precise configuration can be tailored to the needs of the target system. Here we discuss in detail the two meta-models provided by default by the OpenCom architecture: the *architecture meta-model* and the *interface meta-model*.

The *architecture meta-model* uses structural reflection to represent the current topology of a system. In OpenCom, the architecture meta-model consists of a representation of components and their bindings. This representation allows the topological adaptation of a target system. The *interface meta-model* supports two different capabilities: first, it can discover the details of the interfaces and receptacles in terms of their

operation signatures. Secondly, the interface meta-model enables ‘dynamic invocations’ to be undertaken on dynamically-discovered interfaces. These capabilities enable components to invoke interface types which are unknown at coding time.

3.4. The Extensions Framework

As mentioned in Section 3.2, the kernel is only capable of deploying primary-style components. The extensions framework removes this limitation and allows the kernel to support heterogeneous types of components. This framework is required whenever components relying on incompatible technologies need to co-exist within the same application.

The extensions framework relies on three new abstractions: *caplets*, *loaders*, *binders* (Figure 2). A combination of these is necessary for each new type of component that is to be supported by the kernel. From an implementation point of view, caplets, loaders and binders are primary-style components and are deployed by the kernel. They provide a bridge to a new deployment environment.

The sum of all components, primary and non-primary, that are managed by the kernel belong to what we have termed as a *capsule*. We discuss each of these new terms in more detail below.

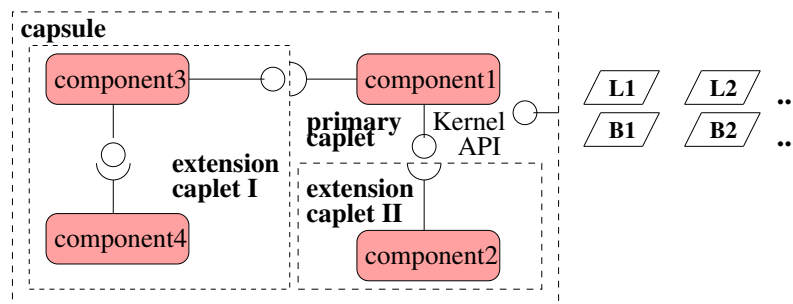


Figure 2. The elements of the OpenCom programming model (the loaders are represented as rectangles on the right)

Capsules are “component containers” that may encompass multiple address spaces and are managed by one single kernel. They provide a namespace in which all components and interfaces are individually identified by a unique identifier. Each capsule is associated with one specific OpenCom kernel that is responsible for deploying components in that particular capsule. Usually, an entire system will reside in one capsule, as in the example shown in Figure 3, where a capsule encompasses an entire IXP1200 router (PC, StrongARM processor and microengines). The notion of capsule helps reason about cases where multiple kernels must interact (for legacy or compatibility reasons, for instance).

Caplets are OpenCom’s representation of a “locus of deployment”. In an environment that supports virtual memory for instance, a caplet will typically be mapped to an address space. However, in more limited environments, several caplets may share the same single address space, with additional protections possibly integrated in the binders, the loaders, or the components themselves. Conceptually, caplets allow the kernel to support different technologies in the underlying deploying environment. They also provide an isolation mechanism between components which are mutually distrustful or have different privileges. They are similar to the notion of *domains* implemented in the THINK

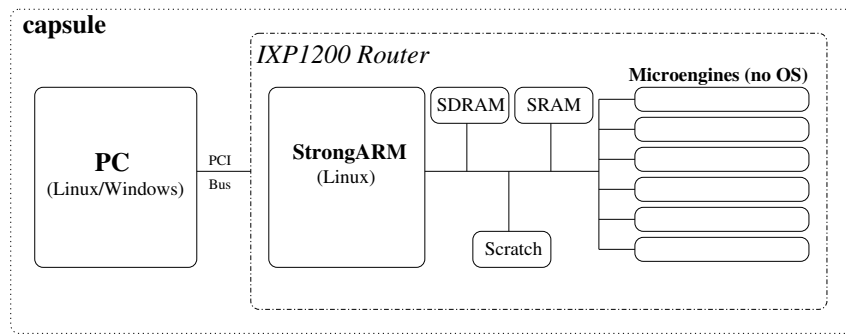


Figure 3. An example of capsule

component model [6]. For example, if a system requires deployment of both C++ and Java components, this is achieved by creating separate caplets for each technology domain.

Note that caplets and capsules differ in that caplets are explicitly created, unlike the capsule associated with each kernel. In this respect, a capsule should be regarded a *logical* component containers that may encompass several address spaces and hardware domains and may therefore contain several caplets.

Loaders encapsulate the complexity of loading components in a heterogeneous environment. Each loader supports a specific loading mechanisms, thus allowing a wide range of component styles to be managed by the same kernel.

Finally, *binders* are OpenCom components designed to provide a wide range of 'binding mechanisms'. Using binders, developers are free to implement a wide range of binding mechanisms that might be required in the underlying deployment environment. For example, they may implement a binder that creates connections between Java components or a binder that connects components written in Assembly language.

When the kernel starts, it only contains a single caplet, the *primary caplet*. At this point the caplet coincides with the enclosing capsule. Additional caplets termed *extension caplets* may then be progressively added to the system through the extensions framework to support new styles of components.

3.5. The OpenCom Programmer Roles: Creating and Using Extensions

OpenCom's programming model provides a strong separation of concerns between portability issues and functional development. In particular, OpenCom supports two distinct programmer roles:

- *Deployment programmers* ensure OpenCom's programming model is available in a given deployment environment. More precisely, deployment programmers port the kernel to a specific deployment environment and implement the required caplets, loaders and binders.
- *Target developers* develop target systems by using the OpenCom programming model. If needed, target developers can use the set of additional caplets, loaders and binders provided by deployment programmers.

4. Performance Evaluation

4.1. Overview

This section examines a quantitative evaluation of the inherent performance properties and incurred overhead by OpenCom. To obtain an accurate figure, we often computed the average of million measurements. For example, the loading time of OpenCom primary-style components was computed by calculating the average figure from over a million *load()* calls.

The experiments in this section were all carried out using a Dell Precision 340 series workstation with 4 x 1.6GHz Pentium CPUs, 512 MB of RAM, and running Linux Redhat 8.0. With regard to software, all the measurements were collected by using the kernel implemented in C++ for Linux. The primary-style components are all stored in the Linux Shared Object libraries, which are equivalent to Windows DLLs. It should be pointed out that some experiments were also carried out on the *Radsys* IXP1200 router environment [11]. This network processor-based router runs *BlueCat* embedded Linux as the OS.

4.2. Evaluation of the Kernel

The kernel for this evaluation was implemented in C++ using the GNU GCC compiler for Linux environments (Redhat Linux version 8.0). Primary-style components deployed by this kernel are stored in Linux *shared object* libraries, which enable components to be deployed dynamically at runtime. The kernel uses the C++ object creation mechanism for component creation. On top of the kernel optional extensions framework and reflective meta-models are also implemented in C++.

4.2.1. Memory Footprint

Table 1.a illustrates the memory overhead required by the kernel, extensions framework and reflective meta-models. As shown in the table, the kernel itself is lightweight and can enable its deployment in resource constrained devices such as a sensor node. For comparison purposes: MMLite [9] kernel requires 26KBytes of memory footprint which is close to that required by OpenCom.

| <i>Architectural layer</i> | <i>Memory footprint</i> | <i>Operation</i> | <i>OpenCom</i> | <i>C++</i> |
|----------------------------|-------------------------|------------------|----------------|--------------|
| Kernel | 32KB | Load | 9.8 μ s | 7 μ s |
| Extensions framework | 10KB | Instantiation | 0.47 μ s | 0.28 μ s |
| Reflective meta-models | 8KB | Time | | |

Table 1. a) Memory footprint; b) Loading and instantiation time

The memory footprint required to accommodate a single null primary-style component was measured as 24 bytes. The component in this experiment has no interface and receptacle. In addition, it has null initialisation and finalisation routines. The per-interface and per-receptacle memory overhead is 8 bytes with an additional 5 bytes per operation.

To obtain an accurate figure, several components (i.e. a hundred) were deployed to compute the memory overhead incurred by each null component. The memory overhead produced by one hundred components was then divided by 100, which provided the

overhead imposed by each deployed null component. The same calculation was applied to measure the per-interface and per-operation overhead mentioned above.

From the above results, the component memory footprint is quite comparable to that required for a single null C++ object which was measured as 20 bytes.

4.2.2. Component Loading and Instantiation Time

Table 1.b shows a comparison between the time taken to deploy OpenCom primary-style components and C++ objects. The time measured to load a single null primary-style component is $9.8\mu s$. This includes the time taken to read a shared object library from the disk storage and extract the information of each primary-style component stored in the library. The extracted information of primary-style components includes the constructor function, which is utilised to instantiate a component. This measurement was computed by calculating the average figure from over a million *load()* calls. This is quite comparable to that obtained when loading a null C++ object by means of the same shared object library. The time computed for C++ object was measured as $7\mu s$. This *loading* time is the measurement of the overhead taken to read a shared object library from the disk storage.

To instantiate a null already-loaded component, OpenCom took $0.47\mu s$ compared to $0.28\mu s$ taken to instantiate a null C++ object. The differences in this measurement can be attributed to the larger file size that is required by OpenCom components to accommodate information for the meta-data. In addition, it can also be attributed to the complex instantiation mechanism that is employed in OpenCom, which includes the interaction with the kernel registry (see figures on the table above).

4.3. Performance of the Extensions Framework Mechanisms

Performance evaluation of the extensions framework mechanisms was carried out in the IXP1200 router environment [11] (see Figure 3). This router is particularly interesting because it is *i*) heterogeneous (e.g. it has a number of processors, including the microengines that are specialised for packet processing); *ii*) resource-poor, having a small amount of memory; and finally, *iii*) performance constrained (i.e. packets must be processed at line speeds). In terms of the implementation, we have developed a caplet, loader and binder components to deploy Microcode components that run in the microengines. Microengines are RISC CPUs that are targeted at packet forwarding and processing, while Microcode is the Assembly language for the microengines.

4.3.1. Microcode Loader and Binder

To evaluate the performance of the implemented microcode loader and binder plugins, a comparison was made with that obtained by NetBind [2]. NetBind is a good point of comparison given that our microcode loader and binder employ an approach that was pioneered by their project. NetBind loads at the granularity of so-called *pipelines*. A pipeline is a pre-configured assembly of singleton microcode components. These pipelines constrain the modules to be deployed in a linear topology and are created whenever these modules have to be loaded and bound. Unlike NetBind, the implemented loader and

binder extensions allow the components to be deployed individually by creating any kind of arbitrary topology.

To compare the NetBind performance with that obtained by the implemented plugins, this experiment evaluates the imposed overhead to incrementally extend the linear topology of components in a single microengine. In particular, the time taken to include each microcode component in the existing pipeline is measured. The results given in Figure 4 show that NetBind incurs an increasing linear overhead. This results from having to rebuild the entire pipeline for each new configuration. Unlike Netbind, the implemented binder plugin incurs a constant overhead for each extension, as the components are bound incrementally. The binding overhead is minimal as it only involves changes in the branch instructions. It should be underlined that in this experiment, all the microengine components were deployed in one single microengine.

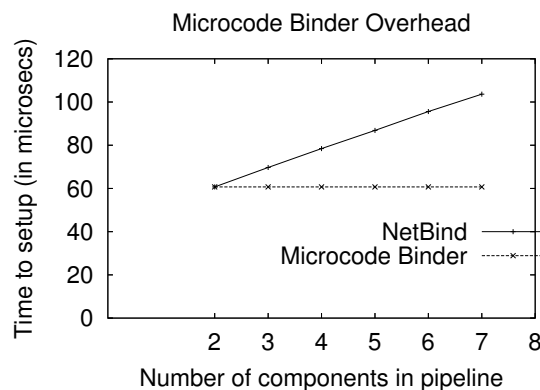


Figure 4. Time to make an incremental addition of a component to an existing configuration

5. Case Studies

In this section we present three case studies that demonstrate the suitability of our component model in a wide range of environments and system domains. The case studies also illustrate the capacity of the extension components (i.e. caplets, loaders and binders) to deploy a variety of component styles in diverse environments. This demonstrates that OpenCom is suitable for constructing systems that run in a demanding heterogeneous environment like a network processor based router.

5.1. Case Study I: Programmable Networking Environment on Network Processors

Network processors (hereafter called NPs) consist of a multi-processor programmable device that forwards and processes packets at high speed (typically gigabit/s). Like conventional CPUs, these processors are programmable. However, unlike conventional CPUs, NPs are optimised for packet forwarding and processing.

These environments offer an ideal case study to validate the proposed generic systems-building technology for the following reasons:

- NPs are widely known to be very difficult to program. For example, they have their own hardware architecture, often with no OS and rely on a specialised language

to program these devices. They also operate in a heterogeneous environment with a multiple processor and memory types.

- The developed system must be extremely efficient to meet the requirements set by these devices, namely to meet the imposed gigabit forwarding speeds.

This case-study applies the OpenCom programming model to construct systems in such environments, more precisely, in the Intel IXP1200 range of NPs [11]. This router is composed of *i*) a StrongARM CPU running Linux, which acts as the general control processor in the router; *ii*) an array of six so-called microengines RISC CPUs that are attached to each other and share three types of memory as shown in Figure 3 (i.e. Scratchpad, SDRAM and SRAM); and *iii*) a set of dedicated hardware elements (not illustrated in the figure above) such as the network interface and hash unit.

The *microengine caplet* provides a communication channel between the primary caplet in the StrongARM and the target microengine caplet. This channel is implemented through the libraries provided by Intel that support direct access from the StrongARM to the microengine microstore, and memories such as the Scratchpads, SDRAMs and SRAMs. Instructions and data can be exchanged between components in the StrongARM and components in the microengines by sharing these registers. For example, components in the microengines can place an integer value in an SDRAM register that can subsequently be retrieved by components running in the StrongARM.

The *microengine loader* embodies the capability to load and instantiate microcode components. The components that are deployable by OpenCom in the microengines are coded either in microengine-C (i.e. a subset of C for the IXP platforms) or Assembly. Both languages are very primitive and highly constrained. In particular, this loader enables the target developer to deploy components in the same fashion that takes place for the primary style components, despite the primitive underlying environment. Moreover, loading components in these environments is extremely complex and cannot be undertaken in a straightforward way. For example, it is only possible to load components into the microengines while they are in the *stop mode*. The stop mode is the mode where the microengines do not perform any instruction. In this particular context, the *instantiate()* operation activates a previously loaded component by starting its execution on the microengines.

Section 4.3.1 outlines a quantitative evaluation of the implemented microcode loader and binder. The graph in Figure 4 makes a comparison between the reconfiguration overhead incurred by our extensions and that by NetBind.

5.2. Case Study II: Middleware for Parallel Environments

In view of the current diversity of applications and deployment environments, it is desirable that middleware platforms are capable of being tailored to a particular applicability (e.g. mobile computing middleware, sensor network middleware, parallel environment middleware, etc.).

This case study applies the OpenCom programming model in the domain of middleware platforms for parallel environments called FlexPar. Essentially, it evaluates the use of OpenCom in constructing a flexible middleware that is capable of building parallel software at runtime. In addition, it also demonstrates that OpenCom is not only targeted

at constructing programmable networking environments, but is rather a generic form of technology that can be adopted to construct systems for a wide range of target domains.

In short, FlexPar is a reconfigurable middleware that is capable of constructing parallel software at runtime. It builds parallel software by deploying multithreaded components that make use of the CSP (Communicating Sequential Processes) [10] disciplines. CSP is a paradigm for concurrent programming that prevents common problems found in concurrency, such as deadlocks, and race hazards. FlexPar deploys components in JCSP [21] and *occam-pi* as both support the CSP paradigm. JCSP is a set of Java libraries that provides CSP disciplines for Java programmers. *occam-pi* is another CSP-based language that has lower performance overhead than that of Java. As a result, *occam-pi* is often adopted to construct parallel embedded systems in environments with low resources [12]. We implemented the FlexPar infrastructure in the following way:

- *Kernel*. In this case study, we coded the kernel in Java in order to simplify the deployment of components written in JCSP. The construction of the kernel proved to be straightforward, on account of its size and simplicity. The FlexPar kernel required a minimal memory footprint of 40 KBytes.
- *Caplets, loaders and binders*. Caplets, loaders and binders were implemented to support a range of component styles, including the primary (in C++) and interpreted (in Java and JCSP) styles. Thus, extensions (i.e. caplets, loaders and binders) to support primary-style components and Java-based components were employed in this case study. In addition, a caplet, loader and binder that support the deployment of components in *occam-pi* are now under construction.

The FlexPar architecture is structured as depicted in Figure 5. The architecture is composed of a kernel whose role is to deploy the JCSP and *occam-pi* loaders and binders. This kernel follows the microkernel style to enable its deployment in environments where resources are particularly scarce.

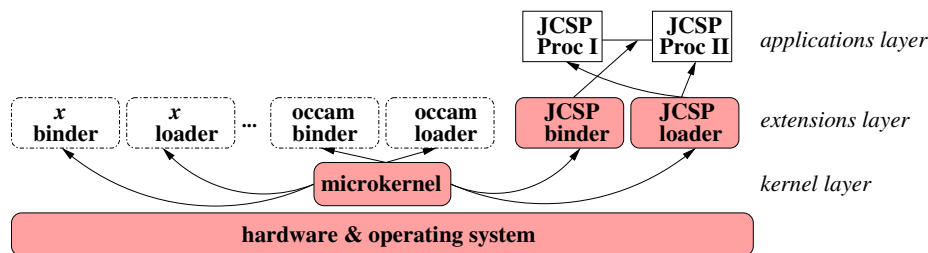


Figure 5. FlexPar architecture

It is important to mention that there is no bias for JCSP and *occam-pi* as the architecture is extensible and supports other parallel languages. This is illustrated in Figure 5 through *x loader* and *binder* that are designed to load and bind, respectively components written in a fictitious *x* language.

The OpenCom flexibility is also valuable to tailor middlewares to a wide range of target domains. As an illustration of this, a middleware for mobile computing called ReMMoc (Reflective Middleware for Mobile Computing) was constructed by applying the principles adopted by OpenCom (i.e. component-based software engineering, component frameworks, reflective meta-models). A middleware for grid computing [3] has also been implemented, which adopts the OpenCom approach.

As part of the FlexPar project, we are now constructing our *occam-pi* loader and binder extensions. We will also construct a cross-binder extension which will let us create connections between a JCSP and *occam-pi* component.

5.3. Case Study III: Software for Sensor Motes

OpenCom was employed in this case study to construct low-level software targeted at sensor motes. Sensor motes are very primitive environments that consist of a small circuit board that hosts a number of electronic sensors and a simple 8-bit microcontroller. This third case study demonstrates that OpenCom can be used to construct low-level software running in resource-constrained devices such as a sensor mote.

To deploy OpenCom in the sensor motes, the kernel had to be built on top of a simple microcontroller monitor program called *Contiki*. Contiki supports some simple facilities to load C code modules dynamically and bind them. The provision of the OpenCom kernel on top of Contiki makes it easier to program these primitive environments and also provides a standard and uniform programming model that is applied ubiquitously, including in the standard PCs.

Essentially, this kernel for Contiki leads to the following benefits:

- It provides the notion of components with multiple interfaces and receptacles, whereas in Contiki, there are only code modules with a single set of functions.
- It allows multiple instantiations of a single component, whereas Contiki only supports singleton modules.
- Finally, it offers the facility to rebind components, whereas in Contiki one has to destroy an instantiated component and reload it again, if it is necessary to create a binding to a different component.

This case study demonstrates that the OpenCom approach is also suited to low-level software targeted at primitive and resource-constrained devices, like a sensor mote. The implemented Contiki-kernel occupies only 30 Kbytes, which demonstrates that it is portable to a wide spectrum of deployment environments.

6. Related Work

We now briefly discuss several component models that we analysed with a focus on the flexibility of the architecture in constructing systems independent of the target domain and deployment environment. The analysed component models were classified into four distinct categories that were based on the domain of the target systems at which they were aimed: embedded systems, middleware platforms, operating systems and programmable networking environments.

In the first category a number of component models for embedded systems have been proposed (Koala [20], RoboCop [18], PECOS [26], and SaveCCM [8]). Unfortunately, none of them provides a generic programming model that can be used outside of their narrowly targeted area. They are primarily concerned with implementing a model that can ensure that the constraints for a real-time system are all fulfilled. For example, some of them (SaveCCM, PECOS, and RoboCop) provide a *component description language* that is particularly designed to ensure that all the informed constraints are fulfilled.

However, this description language cannot be applied widely to other target domains since it is narrowly-targeted at real-time systems.

In the second category, component models for middleware platforms have been proposed (K-Component [5], DPRS [22], Fractal [1], and OMG's CORBA Component Model (CCM) [19]). These component models demonstrated that they have poor support for deploying multiple component styles, particularly components written in Assembly language. This precludes the adoption of these component models for primitive environments such as an embedded device. In addition to this, it is evident from this analysis that most of the above component models do not aim to deploy components in environments that are resource-constrained. This is normally reflected in the higher memory footprint overhead that is imposed by most of the analysed component models.

In the third category, an investigation was made on component models targeted at operating systems. This investigation included THINK [6], MMLite [9] and OSKit [7]. It was noted that most of the component models are primarily targeted at providing modular OS functionality and that this hinders the adoption of such technologies for constructing systems for other domains. Another factor is that some of the technologies investigated rely on heavy-weight platforms that preclude their use, particularly where memory resources are poor. For example, 2K is built on top of CORBA, whereas OSKit relies on a subset of Microsoft COM. This hinders the use of both 2K and OSKit in resource-poor embedded devices.

Finally, we also analysed the component technologies targeted at programmable and active networks. This includes models such as NetBind [2], VERA [14] Shalaby's router [23], ACE [11], and Click [16]. This analysis suggested that these component models for networking environments are all narrowly-targeted at a particular router platform and unable to be employed to construct systems targeted at other domains. This reflects the goals and nature of these component models: they are usually targeted at solving one particular challenge in the domain of computer networks. For example, NetBind is a component model that is aimed at constructing data paths that are reconfigurable at runtime. ACE proposes a component model that is particularly targeted at deploying components on the Intel IXP routers.

7. Conclusions and Further Work

This paper described the design of the OpenCOM component model, a novel component technology for building system software. The primary focus of the design is to provide a generic facility for building system software that is independent of the deployment environment and the target domain at which it is aimed. This general-purpose component model is beneficial as it enables programmers to rely on a single tool to build systems targeted at a variety of domains and/or deployment environments. OpenCom proposes a flexible, extensible and language independent component model with a minimal kernel. Since the kernel is minimal, it can be easily ported to other deployment environments. Finally, OpenCom has been applied to the needs of both present and future work. For example, the Runes (Reconfigurable Ubiquitous Networked Embedded Systems) is an EU-funded project that aims to build an architecture for networked embedded systems that encompasses dedicated radio layers, networks, middlewares, and specialised simulation and verification tools. This project aims to employ OpenCom as the programming plat-

form which lies at the heart of the Runes architecture. Plastik [13] is a meta-framework that attempts to integrate an architecture description language (ADL) to OpenCom.

Acknowledgements

Jó Ueyama would like to thank the National Council for Scientific and Technological Development (CNPq - Brazil) for sponsoring his PhD scholarship at Lancaster University (Ref. 200214/01-2). The first and fourth author would also like to thank FAPESP for funding the FlexPar research project (Ref. 2006/06576-8).

References

- [1] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.
- [2] A.T. Campbell, M.E. Kounavis, D.A. Villela, J.B. Vicente, H.G. de Meer, K. Miki, and K.S. Kalaichelvan. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *5th IEEE International Conference on Open Architectures and Network Programming (OPENARCH'02)*, June 2002.
- [3] G. Coulson, P. Grace, G.S. Blair, L. Mathy, D. Duce, C. Cooper, W. Yeung, and W. Cai. Towards A Component-Based Middleware Framework for Configurable and Reconfigurable Grid Computing. In *WETICE*, pages 291–296, 2004.
- [4] G. Coulson, Blair G.S., M. Clarke, and N. Parlavantzas. The Design of a Highly Configurable and Reconfigurable Middleware Platform. *ACM Distributed Computing Journal*, 15(2):109–126, April 2002.
- [5] J. Dowling and V. Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In *Reflection 2001*, Kyoto, Japan, September 2001. LNCS 2192.
- [6] J.P. Fassino, J.B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX 2002 Annual Conference*, June 2002.
- [7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51. ACM Press, 1997.
- [8] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. SaveCCM – A Component Model for Safety-critical Real-time Systems. In *Euromicro Conference, Special Session Component Models for Dependable Systems*. IEEE, September 2004.
- [9] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *8th ACM SIGOPS European Workshop*, pages 96–103, Sintra, Portugal, September 1998.
- [10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [11] Intel. Intel IXP1200. <http://www.intel.com/IXA>, 2002.
- [12] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*,

SBES 2007
XXI Simpósio Brasileiro de Engenharia de Software

volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

- [13] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable. In *5Th Working IEEE/IFIP Conference on Software Architecture*, Pittsburg, Pennsylvania, November 2005.
- [14] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *4th International Conference on Open Architectures and Network Programming (OPENARCH)*, April 2001.
- [15] Microsoft Corporation. .Net Home Page. <http://www.microsoft.com/net>, 2001.
- [16] R. Morris, Kohler E., J. Jannoti, and M. Kaashoek. The Click Modular Router. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Charleston, SC, USA, December 1999.
- [17] Mozilla Organization. XPCOM Project. <http://www.mozilla.org/projects/xpcom>, 2001.
- [18] J. Muskens and M. Chaudron. Prediction of Run-Time Resource Consumption in Multi-task Component-Based Software Systems. In Springer Verlag, editor, *Proceeding of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004.
- [19] OMG. The CORBA Component Model. orbos/99-07-01.
- [20] R. Ommering, F. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. 33(3):78–85, March 2000.
- [21] P.H. Welch and J.M.R. Martin. A CSP Model for Java Multithreading. In P. Nixon and I. Ritchie, editors, *Software Engineering for Parallel and Distributed Systems*, pages 114–122. ICSE 2000, IEEE Computer Society Press, June 2000.
- [22] M. Roman and N. Islam. Dynamically programmable and reconfigurable middleware services. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 372–396, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
- [23] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, a Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible routers for active networks, 2002.
- [24] Sun Microsystems. Enterprise Java Beans Specification Version 1.1. <http://java.sun.com/products/ejb/index.html>.
- [25] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
- [26] M. Winter, T. Genbler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Muller, C. Stich, and B. Schonhage. Components for Embedded Software: the PECOS Approach. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA, 2002. ACM Press.