Enabling Re-configurability on Component-based Programmable Nodes

Jó Ueyama, Stefan Schmid, Geoff Coulson, Gordon S. Blair, Antônio T. Gomes, Ackbar Joolia, Kevin Lee

Computing Department
Lancaster University
LA1 4YR Lancaster, UK
{ueyama, sschmid, geoff, gordon, gomes, joolia, leek}@comp.lancs.ac.uk

Abstract

Recently developed networked services have been demanding architectures that accommodate an increasingly diverse range of applications requirements (e.g. mobility, multicast, QoS), as well as system requirements (e.g. exploiting specialized processing hardware). This is particularly crucial for architectures of network systems where the lack of extensibility and interoperability has been a constant struggle, hindering the provision of novel services. It is also clear that to achieve such flexibility these systems must support extensibility and re-configurability of the base functionality subsequent to the initial deployment. Based on our experience with middleware platforms, we argue that re-configurability of network services is best implemented by means of reflection.
In this position paper we present component-based approach to developing flexible networked systems, especially targeted at the Intel IXP1200 programmable networking environment and discuss how our approach can offer a more deployable, flexible and extensible networking infrastructure. We show the viability of our component model to re-configure services on the Intel IXP1200 platform. An application scenario is presented to validate our approach to enable re-configuration of services across different layers of an IXP1200-based router platform.

1 Introduction

An increasing number of recent applications (e.g. real-time, multimedia) and their underlying system infrastructures (e.g. workstations, PDAs, embedded systems, ad-hoc networks) have been requiring a flexible architecture to accommodate all the requirements necessary to run these applications as well as to inter-operate in a heterogeneous environment composed of different types of applications and hardware platforms. It is increasingly clear that to achieve this, we need an extensible and re-configurable architecture that is capable of loading and integrating new functionality at run-time. As an example of re-configurability, we could load and unload services on a network router and intelligently adapt its forwarding behaviour to various types of traffic and environments such as mobile or ad-hoc.

Unfortunately, although much research on providing an open architecture for networking systems have been carried out, we still lack a generic approach to develop and deploy new network services. Existing paradigms address configuration and re-configuration of services running on a particular level of a programmable networking system (e.g. open signalling for *control functions*, and active networks for *in-band packet processing*). But there is as yet no really comprehensive approach.

At the same time, component technology [21] has been widely cited as a suitable model for developing adaptive software due to its incrementally deployable nature [10]. For example, with component technology described in [10] one can add, replace and remove the constituent components residing in the same address space. Therefore, the use of component technology provides a means for deployment-time configurability and run-time re-configurability. However, although component-based architecture have been successfully used in many adaptive applications, early research into active networks has not truly adopted a component model [18]. Moreover, the majority of existing work (e.g. Vera [15] and Genesis [5]) omit support for dynamic re-configuration. RANN (*Reflective Active Network Node*) [22] introduces the use of reflection to support flexible configuration in active networks, but it only defines an architecture

where active nodes use reflection to better structure services. Essentially, RANN defines an architecture for configuration rather than re-configuration. Moreover, we argue that this work is partial; it only addresses the configurability within the execution environment, and not at lower or higher system levels. Furthermore, RANN is language specific (Java).

As a consequence, this paper presents the design and implementation of a component-based architecture for programmable networking software, which provides an integrated means of developing, deploying and managing such systems. The proposed architecture consists of a generic component model applied on *all levels* of the programmable network design space, which ranges from fine-grained, low-level, in-band packet processing functions to high-level signalling and coordination functions. The projected benefits of this approach are detailed in section 4. Configuration and re-configuration across this architecture is achieved by dynamically loading and unloading service components. Reflection is used to reify configurations of components and to support various types of meta-data to facilitate configuration and re-configuration.

The remainder of the paper is structured as follows. Section 2 looks in detail at existing component-based frameworks for generic applications and component-based technology to provide programmability in networks which leads to the motivations of our work. Subsequently, section 3 examines existing technology necessary for this architecture. It introduces our component model OpenCOM, the concept of component frameworks and finally covers the IXP1200 platform. Section 4 then reports on our globally applied component-based architecture to enable dynamic creation, deployment and management of services in programmable networking environments. Our approach and an overview of the design space of the programmable networking environment is detailed in the same section. The design and implementation of our model carried out to date is presented in section 5. Finally, section 6 draws general conclusions from this paper and the proposed architecture.

2 Analysis of Existing Work

We distinguish here component-based platforms for general purpose applications from component models tailored towards a specific application domain (such as active and programmable networks).

2.1 Generic Component-based Platforms

Although there exists a wide range of research systems (e.g. XPCOM, K-Component [11]) that implement component platforms for generic applications, we argue that all of these are suboptimal for programmable networks due to their lack of special support for the hardware platforms used in this context. These limitations are frequently related to the platform upon which the component is built for (e.g. the Java virtual machine). Another limitation regards the system layer at which the component model is targeted. For example, none of the existing component models consider the integration of low-level and high-level components running on different layers of the router hardware (e.g. in-band packet forwarding, or signaling). There exists work addressing low-level components providing an architecture to build component-based OSs (e.g. Think [12], Knit [17]). Nevertheless, typically these systems do not provide a uniform and globally applied framework to load and bind both assembly language-based components and high-level components.

2.2 Component-based Platforms for Programmable Networks

Aside from the component model for generic applications, there has been considerable research on component-based platforms for programmable routers (e.g. Click [16], NP-Click [20], VERA [15], NetBind [5], LARA++ [19]). Nevertheless few works support configuration and reconfiguration (i.e., adaptation, extension, evolution and removal) sufficiently. Most of them support the first configuration but do not support the subsequent re-configuration at runtime. Moreover, systems that implement re-configuration do not adequately support the management of system integrity over re-configuration operations (e.g. ensuring that firewall updates are applied consistently and universally). Furthermore, these works do not provide an *integrated* approach to configure and re-configure services across all layers of the programmable networking system (see section 4). For example, VERA limits re-configurability to in-band functions and

the hardware abstractions layer, whereas NetBind considers only in-band functions. LARA++, on the other hand, allows re-configurability on all layers, but lacks an uniform model to do so (i.e. different component models are used on the different layers).

3 Background

3.1 Component Frameworks

The concept of component frameworks (CFs) is applied to deal with component constraints and the dimensioning of the applicability of participating components. CFs are also applied to provide the *structure* allowing the use of components for a specific domain of application. A number of runtime CFs have been implemented as part of our past research (e.g. pluggable protocols, pluggable thread schedulers, and pluggable media filtering) [10]. CF were originally defined by [21] as *collections of rules and interfaces that govern the interaction of a set of components "plugged into" them.* In this sense, a CF embodies rules and interfaces that would make sense for a specific domain of application. For example, a CF for pluggable protocols consists of a variety of interfaces and rules to integrate plug-in protocols and ensure that they are stacked in an appropriate order. As another example, CFs can determine constraints to govern the interaction of a set of components: a CF can mandate that a packet scheduler component must always read its input from a packet classifier. Such constraints are useful to ensure meaningful re-configuration and therefore the system must provide support for expressing these constraints. Essentially, CFs provide the necessary support and conditions for components and also regulates the interaction rules between component instances for a specific domain of application. A component framework can come alone or interact and cooperate with other CFs (as long as it conforms to the rules governed on the host CF). Therefore, it is natural to design CFs themselves as components.

3.2 OpenCOM

Lancaster's OpenCOM [7] is a lightweight, efficient, flexible, and language-independent component model that was originally developed as part of previous research on configurable middleware [10]. OpenCOM is fine-grained in that its scope is intra-capsule (for capsule, see below) and it imposes a minimal overhead on cross-component invocation without compromising the overall performance. It is currently implemented on top of a subset of Mozilla's XPCOM platform.

OpenCOM relies on four fundamental concepts:

Capsule: a capsule (see figure 1) comprehends multiple physical address spaces within a single logical container: for example, a capsule could encapsulate both a Linux process in the IXP1200's control processor and one or more microengines. Encapsulating multi-address-spaces offers a powerful means of abstracting over heterogeneous but tightly-coupled hardware (e.g. PC, StrongARM and the microengines of the IXP1200 router platform - see section 3.3 and figure 3 for the IXP1200 architecture).

Interface: expresses a unit of service provision;

Receptacle: define a service requirement and is used to make the dependency of one component on another explicit;

Binding: is an association between a receptacle and an interface. In the original version of OpenCOM, bindings were restricted to receptacles and interfaces residing in the same address space. Currently, OpenCOM is being extended to support binding between receptacles/interfaces of different address spaces as well. Semantically, a connection represents a communication path between one receptacle and one interface. Bindings in the original OpenCOM were exclusively implemented in terms of vtables [3] (a vtable is essentially a table containing pointers to virtual functions). Currently, however, we are extending OpenCOM to support bindings implemented in a variety of ways.

On top of OpenCOM, we have designed a CF called the "Router CF" that is targeted specifically at packet-forwarding functions in routers. See figure 2. In this illustration the *meta-interfaces* support

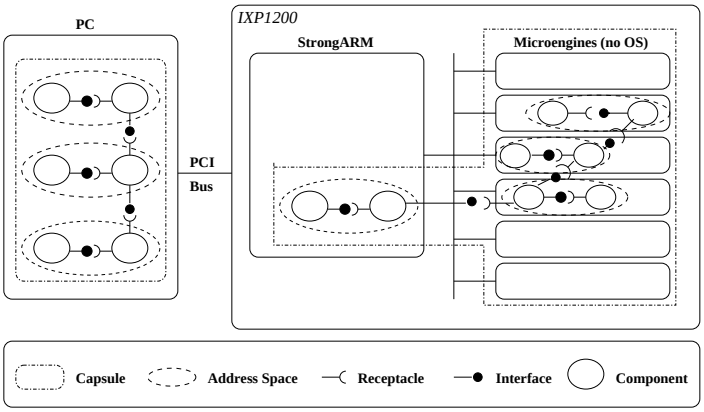

Figure 1: Multi-address-space capsules

inspection of the types of interfaces and receptacles declared by the target component. The meta-interfaces use the underlying XPCOM system to query the component's type library file and return the IID's (interface identifiers) of the interfaces it implements. Information obtained from this inspection can then be used to enable and guide subsequent re-configuration (e.g. to replace one component with another). Reflection is used as the means to obtain such information, which enables the re-configuration in our framework. Moreover, the obtained information is useful in enabling CFs to check the type of interfaces offered, enforcing the compliance of binding rules at runtime. The *controller* component (see figure 2) relies on the mentioned meta-interfaces in order to manage and configure the internal constituents of the respective CF. It is important to emphasize that the CF can be composed hierarchically of other CFs of the same type, in order to build a composite of CFs.

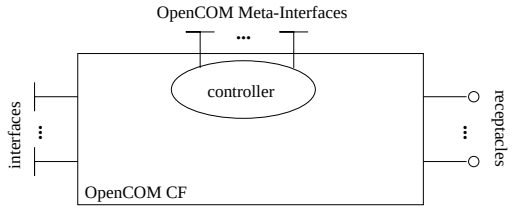

Figure 2: Generic CF for programmable routers

3.3 The Intel IXP1200 Router

The IXP1200 router [8, 13] is an Intel-proprietary architecture based on IXP1200 network processor (NPs). Its architecture combines a StrongARM processor with six independent 32-bit RISC processors called *microengines*, which have hardware multithread support. The StrongARM is the core processor and is primarily concerned with control and management plane operations, whereas microengines handle packet-forwarding. Our component model is being adapted for this platform. In terms of memory, the IXP1200 platform provides the *Scratch* or *Scratchpad* and also the static and dynamic RAM. Figure 3 illustrates the implementation environment of our prototype.

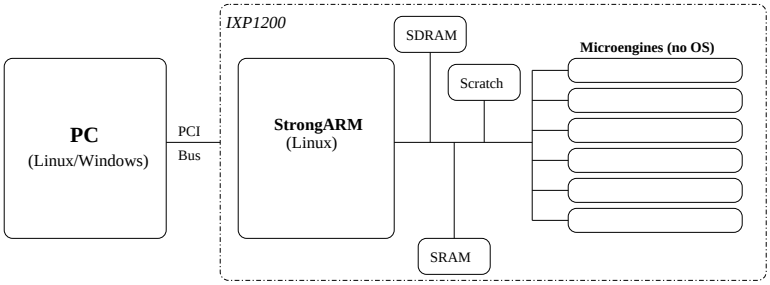

Figure 3: Simulation environment of our prototype

4 Our Approach

4.1 Characterizing Programmable Networking Environments

The design space of programmable networking [9] can be split into four layers or "strata". We use the term "stratum" rather than "layer" to avoid confusion with layered protocol architectures. The four strata (see figure 4) are described as follows:

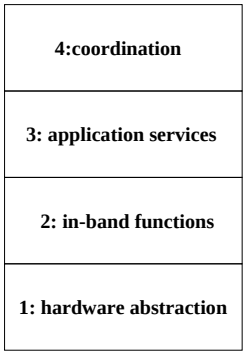

Figure 4: Design Stratas of Programmable Networking

Hardware Abstraction: The *hardware abstraction* stratum corresponds to the minimal operating system functionality needed to run applications on the higher levels. This functionality encompasses threads, memory, I/O, library loading and other services needed to support higher-level network programmability. Interfaces in this stratum are often implemented as *wrappers* around underlying native facilities in order to support heterogeneous platforms which allows transparent use of these services on a number of different router architectures, such as PC-based router or Intel IXP1200s which provides multiple processors (StrongARM and microengines) and distributed/hierarchical memory arrays.

In-Band Functions: This stratum consists of packet processing functions such as packet filters, checksum validators, classifiers, diffserv schedulers, and traffic shapers. Given that these functions are low-level, in-band and fine-grained (and therefore highly performance critical) they must be implemented extremely efficiently (i.e. machine instructions should be counted with care.

Application Services: The application services stratum encompasses coarser grained functions: for example, active networking execution environments [1]. Functions in this stratum are less performance critical and act on pre-selected packet flows in application specific ways (e.g. per-flow media filters).

Coordination: This stratum supports out-of-band signalling protocols which carry out distributed coordination, including configuration and re-configuration of the lower strata. This, for example, includes signalling protocols such as RSVP, or architectures that enable resource allocation in dynamic private virtual networks as employed by architectures like Genesis [4], Draco [14], or Darwin [6].

The main aim of our work is to provide a globally-applied component model, which can enable (re)configuration of services in *all* strata of the programmable networking design space. This yields a number of important benefits. The component model:

- *is simple and uniform* - it allows the creation of services in all strata and provides a uniform run-time support for deployment, inspection and (re)configuration;
- *enables bespoke software configurations* - by the composition of CFs in each stratum, desired functionality can be achieved while minimising memory footprint; trade-offs vary for different systems types (e.g. embedded, wireless devices; large-scale core routers);
- *facilitates ad-hoc interaction*–e.g. application or transport layer components can directly access (subject to access policies) "layer-violating" information from the link layer, which nowadays is considered as indispensable [2].

We aim to apply this approach in both PC-based router as well as specialised programmable routers (e.g. Intel IXP1200). This heterogeneity is fundamental to validate our claim of a generic model. We also strive to implement this model without compromising the overall performance.

4.2 Aims and Objectives

Our project aims to build a framework to enable re-configuration of services running on all levels (*strata*) of any programmable router. It attempts to mask underlying hardware heterogeneity, so that a PC-based router and an IXP1200-based router which has multiple processors (i.e. *microengines* and *StrongARM*) and distributed/hierarchical memory array, will look as similar as possible to other strata. Reconfigurability in this architecture will be carried out by means of reflection technology which is implemented in OpenCOM.

Initially, we are building a version of the Router CF described above for the IXP1200. Essentially, this CF offers an abstracted view of the IXP (i.e. an API) to both developers and deployers of programmable networking functionality. The CF embodies rules and constraints that component developers must follow in developing plug-in components.

5 Progress to Date

In line with the above-mentioned aims and objectives, our component model for programmable networking platforms enables *loading* and *binding* of components residing in all strata of the design space of the programmable networking environment. The core architecture contains essentially these two functionalities: *loading* and *binding*. The former loads a class (i.e. code) and also creates an instance of one component in the specified *capsule_id* and the latter accomplishes the binding between individual components (between interfaces and receptacles).

5.1 Generic Framework

Following the above-mentioned approaches and aims, we extended the core API of OpenCOM with the following methods:

- `load(capsule_id, component_guid);`
- `unload(capsule_id, component_guid);`
- `bind(interface_guid, receptacle_guid);`

- `unbind(interface_guid, receptacle_guid);`

In these methods, *capsule_id* specifies either the capsule where the component will be loaded or the capsule where the component resides (e.g. in the PC, StrongARM or a Microengine). *Component_guid*, *interface_guid* and *receptacle_guid* are component, interface and receptacle identifiers respectively. In our prototype, GUID is the "*globally unique identifier*" and it is used to locate a specific component, receptacle, and interface to be loaded or bound. For high-level components, we employ the ClassID (implemented in XPCOM) as our identifiers for components; and *InterfaceID* as interface and receptacle identifiers. For Microcode-based components, we have been using the name and the path where each component is located to identify the component and its interface and receptacle.

An example environment is depicted in figure 3. Note that this example employs three address spaces: PC, StrongARM and Microengines.

5.2 Re-Configuration: Loading and Binding using Reflective Techniques

The loading mechanism determines the physical placement of components to-be-deployed by taking into account factors such as resource usage as well as QoS and security constraints. Similarly, the binding mechanism wires the component into the appropriate place in the software router configuration. The implementation of the above functionality includes a number of wrapper functions that map each *load* and *bind* method to the corresponding mechanism to re-configure components in the requested address space.

Components running in this environment can typically be of any granularity, i.e., components can be coarse-grained (e.g. developed for the StrongARM or PC platform) or fine-grained, which in our case are typically coded in the Intel IXP1200 machine language (i.e. microcode). This enables us to write high-level services that rely on functionalities provided by low-level components.

In this sense, the binding mechanism can as well connect components of any type. Bindings between microengine and StrongARM components is (typically) carried out by using a shared-memory mechanism (exploiting the IXP1200s SDRAM and SRAM common memory). We have also been using the *Scratch memory* (for Scratch memory, see figure 3) to transfer integer values between these components (see figure 5 for details) as this provides the lowest transfer time (compared to SRAM and SDRAM). Finally, binding between two microengine components is achieved by changing the execution path from one component to another, i.e., by changing the branch instruction, pointing the execution path to the next component (see figure 6). This is implemented in the same way as in NetBind [5]. However, we claim that our implementation is more fine-grained as we do not rely on "pipelines" (a pipeline in NetBind is a set of assembly-based components that are initially connected and executed subsequently). Our component model loads, connects and executes units of single components instead of a set of them. As illustrated in figure 6, each component in Microcode can be composed of four threads that run in parallel.

The *controller* component contains information about the component configuration. It has the reflective capabilities to inspect and re-configure the constituents of the CF. The inspection and re-configuration mechanisms are achieved by the OpenCOM meta-interfaces implemented on each CF (see figure 2). Information on each CF is provided to the *controller* implemented on a CF hierarchically superior to the inspected CF. As pointed out before, a CF may be composed of other CFs in order to build a composite of CFs.

5.3 An Application Scenario

In order to validate our component model, we are now experimenting with a configuration of the Router CF which uses our globally applied component model across several layers of the IXP1200 programmable router environment. This scenario demonstrates how (re-)configuration and reflection can be used to extend the network service on a router at run-time. In particular, the example emphasizes the advantage of having a single component model, which allows configuration and (re-)configuration of the service components across several layers of the programmable network design space and across different processing hardware (i.e., network processors, PCs, etc.). It also shows how (re-)configuration can be carried out in dimensions that do not have to have been foreseen when the system was designed.

The Router CF configuration illustrated in Figure 7 is a typical configuration for an IP router. It consists of several low-level, in-band components on the "fast-path" of the router, namely a classifier and

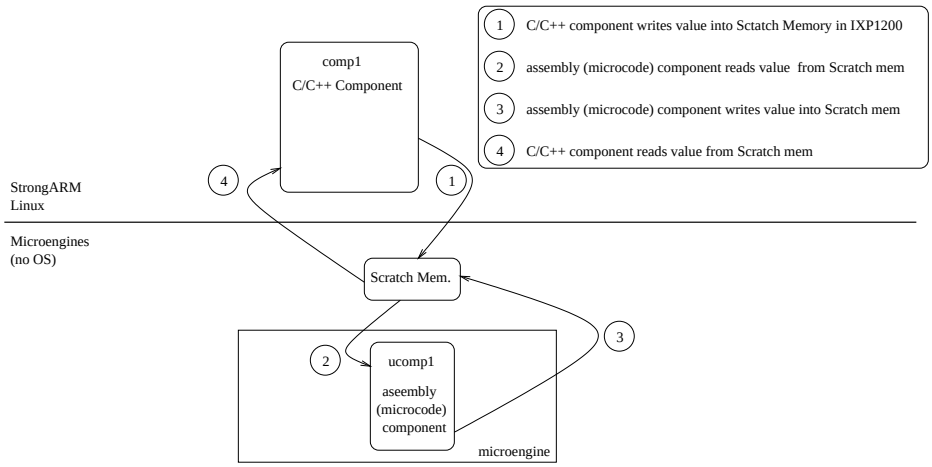

Figure 5: Binding between C/C++ components and assembly (microcode)-based components

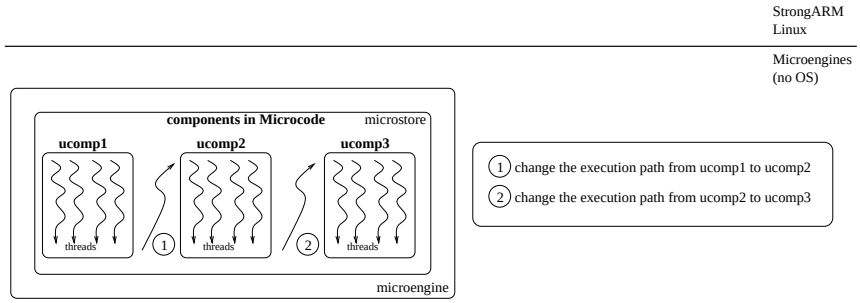

Figure 6: Binding between assembly (microcode)-based components

a forwarder, as well as a queuing and scheduling component, an application service-level component for the processing of IP options on the "slow-path", and a high-level Router CF in the control plane of the router (coordination stratum).

In the case of our IPX1200 based router architecture, we develop the above mentioned functionalities on different strata (in-band, application service and coordination layer) of our multi-strata programmable router architecture, but as component of the same component framework. In order to leverage the processing capabilities and performance of the different hardware layers, we target the different functionalities on the processing hardware best suited for them: thus, we implement the "fast-path" components in machine code for the microengines of the IPX1200 network processors. And for the IP options component on the routers "slow-path", we target on the Linux OS running on the StrongARM processor. And finally, the Routing component, which encompasses several different routing protocol components, we implement on the PC platform.

In order to illustrate the extensibility and flexibility of our approach, we include an IPv6 to IPv4 protocol translation component, which is added to the initial Router CF application at run-time. Such dynamic extensibility can be required to adapt a network environment to provide IPv6 support without restarting the network device.

Like the Router CF itself, the IPv6 to IPv4 translator is spread across different strata and thus areas of the IPX1200 router architecture. While the actual protocol translation takes place in the application services layer, the management component is established at the control plane (coordination stratum) of the router platform. The Translator is integrated with the existing functionality on the router through composition of an overall service CF. The controller component of the overall service CF integrates the

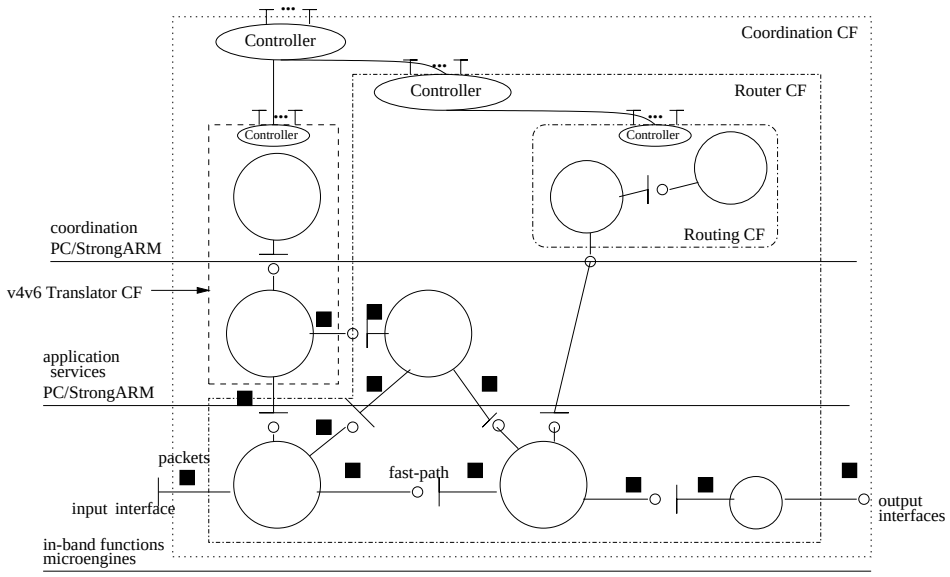

Figure 7: IPv4v6 translator application scenario

Translator by loading the composite and establishing the bindings with the Router CF interfaces.

As mentioned before, reflective techniques are used to accomplish this functionality (i.e. configuration and (re-)configuration of component-based services). Also, it is important to emphasize again that it is the uniform component model that enables components of different strata to interface with each other.

In terms of performance, we point out that the overhead seen by packets transversing the router is no more than would be incurred by Intel's standard software development environment (e.g. using Microace - Intel's component model for IXP1200 platform [13]) as the same bindings (i.e. linkages between software modules) are used in both case. Moreover, the Intel environment (Microace) is static [5], whereas the OpenCOM approach is soft and flexible.

6 Conclusions

In this paper we have proposed a component model that can potentially be applied at all all strata of the design space of programmable network platforms to create services by loading and binding involved components. We believe that the combination of components along with reflection and CFs offers a promising mechanism to configure and re-configure services in networking environments. A key strength of this model is the uniform framework to load and bind both high-level and low-level components. Therefore, loaded and bound components may reside in all strata of the programmable networking environment. As a consequence, we argue that our model facilitates fundamental re-configuration on programmable routers and hence greatly increases flexibility.

Furthermore, it is important to emphasize that our framework facilitates the extensibility and programmability of network processor based systems. These architectures are usually extremely complex and difficult to program and, as a consequence, re-configuration is hardly considered on these "primitive" environments. However, the provision of a generic framework for these architectures gives the programmer a friendly interface (abstraction) to create and consequently re-configure services based on low-level components. We do this by creating an "illusion" for the component developer that low-level components can be loaded and bound in the same way as high-level components.

7 Acknowledgements

Jó Ueyama would like to thank the National Council for Scientific and Technological Development (CNPq - Brazil) for sponsoring his scholarship at Lancaster University (Ref. 200214/01-2). The authors would like to thank the anonymous reviewers for helping to improve this paper. Finally, we would like to thank Paul Grace for his comments on this paper.

References

[1] ANTS. The ants toolkit. `http://www.cs.utah.edu/flux/janos/ants.html`, 2001.

[2] R. Braden, T. Faber, and M. Handley. From Protocol Stack to Protocol Heap — Role-Based Architecture. In *ACM SIGCOMM Computer Communication Review*, volume 33, No 1, January 2003.

[3] K. Brown. Building a Lightweight COM Interception Framework Part 1: The Universal Delegator. *Microsoft Systems Journal*, January 1999.

[4] A. Campbell, Meer G., M. Kounavis, K. Miki, J. Vicente, and D. Villela. The Genesis Kernel: A virtual network operating system for spawning network architectures. In *OPENARCH'99 - Open Architecture and Networking Programming*, New York, USA, March 1999.

[5] 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.

[6] P. Chandra, A. Fisher, C. Kosak, T.S.E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable Resource Management for Value-added Network Services. In *6th IEEE Intl. Conf. on Network Protocols (ICNP 98)*, Austin, Texas, USA, October 1998.

[7] M. Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM Middleware 2001*, Heidelberg, November 2001.

[8] D. Comer. *Network Systems Design using Network Processors.* Prentice Hall, 2003.

[9] G. Coulson, G. Blair, T. Gomes, A. Joolia, K. Lee, J. Ueyama, and Y. Ye. Position paper: A Reflective Middleware-based Approach to Programmable Networing. In *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[10] 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.

[11] J. Dowling and V. Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In *Reflection 2001*, Kyoto, Japan, September 2001. LNCS 2192.

[12] 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.

[13] Intel. Intel IXP1200. `http://www.intel.com/IXA`, 2002.

[14] R. Isaacs and I. Leslie. Support for Resource-Assured and Dynamic Virtual Private Networks. In *JSAC Special Issue on Active and Programmable Networks*, 2001.

[15] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *4th International Conference on Open Architectures and Network Programming (OPENARCH)*, April 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] A. Reid, M. Flatt, L Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, October 2000.

[18] S. Schmid. *A Component-based Active Router Architecture.* PhD thesis, Lancaster University, `http://www.mobileipv6.net/~sschmid/PhD_Thesis.ps`, December 2002.

[19] S. Schmid, T. Chart, M. Sifalakis, and A Scott. Flexible, Dynamic, and Scalable Service Composition for Active Routers. In *IWAN 2002 IFIP-TC6 4th International Working Conference*, volume 2546, pages 253–266, Zurich, Switzerland, December 2002.

[20] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA*, February 2003.

[21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, second edition, 2002.

[22] A. Villazón. A Reflective Active Network Node. In *IWAN*, pages 87–101, 2000.