A Re-configurable Component Model for Programmable Nodes

Jó Ueyama[1], Stefan Schmid[1], Geoff Coulson[1], Gordon S. Blair[1],
Antônio T. Gomes[2], Ackbar Joolia[1], Kevin Lee[1]

[1] Computing Department
Lancaster University
LA1 4YR Lancaster, UK
{ueyama, sschmid, geoff, gordon, joolia, leekj}@comp.lancs.ac.uk
[2] Departamento de Informática,
PUC-Rio,
R. Marquês de São Vicente, 225 – Gávea – 22453-900, Rio de Janeiro, RJ, Brasil
atagomes@telemidia.puc-rio.br

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. 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. In this paper we present a component model that addresses these concerns. We also discuss the application of the component model in network processor-based programmable networking environments and discuss how our approach can offer a more deployable, flexible and extensible networking infrastructure.

1. Introduction

An increasing number of recent applications (e.g. real-time, multimedia) and their underlying systems (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 clear that in order to achieve this accommodation, 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 in 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 tend to only 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*).

At the same time, component technology [15] has been widely cited as a suitable approach for developing adaptive software due to its incrementally deployable nature [7]. For example, with the component technology described in [7] one can add, replace and remove the constituent components residing in the same address space. Therefore, the use of component technology potentially provides the 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 programmable networks has not truly adopted a component model [12]. Moreover, the majority of existing work in this area (e.g. Vera [9] and Genesis [3]) omit support for dynamic re-configuration.

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 networking design space, which ranges from fine-grained, low-level, in-band packet processing functions to high-level signalling and coordination functions. Benefits of this proposed approach are detailed in section 3. Configuration and re-configuration across this architecture is achieved by dynamic loading and unloading of service components. Further, reflection is used to reify configurations of components and to support vari-

ous types of reconfiguration.

The remainder of the paper is structured as follows. Section 2 examines existing technology that underpins our approach. It introduces our component model OpenCOM, the concept of component frameworks and finally covers the Intel IXP1200 network processor based router platform. Section 3 then presents our "globally applied" approach to network programmability, and section 4 discusses our implementation efforts so far. Subsequently, section 5 presents an application scenario that illustrates our approach. Finally, section 6 surveys and analyses related work; and section 7 draws general conclusions from this paper and the proposed architecture.

2. Background

2.1. OpenCOM

Lancaster's OpenCOM [4] is a lightweight, efficient, flexible, and language-independent component model that was developed as part of previous research on configurable middleware [7]. OpenCOM adopts a computational model based on interfaces, receptacles and connections and has basic reflective facilities built in.

OpenCOM relies on five fundamental concepts:

Capsules: a capsule is a logical "component container" that may encompass multiple address spaces (although capsules do not cross machine boundaries). For example, a capsule could encapsulate multiple Linux processes, or different hardware domains on a network processor-based router. Encapsulating multiple address spaces offers a powerful means of abstracting over tightly-coupled but heterogeneous hardware (e.g. the PC, StrongARM and microengines of an Intel IXP1200 router platform – see section 2.3 and figure 1).

Components: components serve as programming language-independent units of deployable functionality. One builds systems by loading components into capsules, and then composing these with other components (by binding their interfaces and receptacles; see below).

Interfaces: interfaces of components are expressed in a language independent interface definition language and express a unit of service provision; multiple interfaces can be supported per component.

Receptacles: define a service requirement and are used to make the dependency of one interface on another explicit (and hence one component on another);

Bindings: Bindings express an association (a communication path) between one receptacle and one interface residing in the same capsule.

OpenCOM is currently implemented on top of a subset of Mozilla's XPCOM [11] which is a cross platform component model that can run over a large number of architectures. OpenCOM is inherently language-independent and employs binary-level component bindings, which has the potential to provide the performance needed in networking systems.

A crucial aspect of OpenCOM that is heavily exploited in our programmable networking research is its support for *plug-in loaders* and *plug-in binders*. Essentially, loading and binding are viewed as components frameworks (see below) in the OpenCOM architecture, and it is possible to extend the architecture with many and various implementations on loading and binding. We return to this topic in section 4.

2.2. Component Frameworks

CF were originally defined by [15] as "*collections of rules and interfaces that govern the interaction of a set of components 'plugged into' them*". In our sense, a CF embodies rules and interfaces that would make sense for a specific domain of application. For example, a packet-forwarding CF might accept packet-scheduler components as plug-ins; or a media-stream filtering CF might accept various media codecs as plug-ins [7]. A number of runtime CFs have been implemented as part of our past research (e.g. pluggable protocols, thread management–offering pluggable schedulers, buffer management, media filtering and extensible binding types) [7]. (In the rest of the paper we use the shorthand "plug-in" to refer to a component that is plugged into a CF.)

Crucially, CFs can impose constraints that govern the way a set of plug-ins inter-relate. For example, in a packet forwarding CF, a CF can mandate that a packet scheduler component must always read its input from a packet classifier. Such constraints are useful to facilitate meaningful re-configuration and therefore the system must provide support for expressing these constraints.

A component framework can be standalone or can 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.

In our programmable networking research we have designed a generic "Router CF" on top of OpenCOM that enables the flexible configuration and reconfiguration of software routers. This is described in detail in [6]. The use of the Router CF is illustrated in section 5.

2.3. The Radisys Intel IXP1200 Router

The IXP1200 router [5, 8], on which we have implemented our OpenCOM component model, is an Intel proprietary architecture based on IXP1200 network processors (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 the microengines handle packet forwarding. The IXP1200 is illustrated in Figure 1.

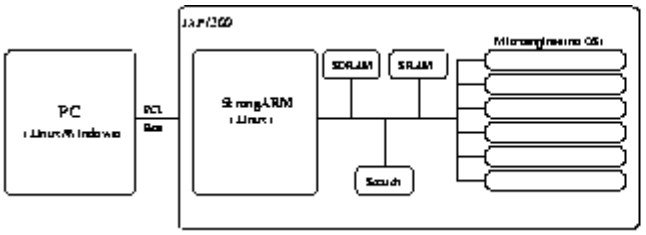

Figure 1. IXP1200 hardware environment of our prototype

3. Our Approach to Building Programmable Networking Systems

3.1. The Design Space of Programming Networking

The design space of programmable networking [6] can be split into four layers or strata. We prefer the term "stratum" to "layer" to avoid confusion with layered protocol architectures. The four strata are described as follows:

Hardware Abstraction: The *hardware abstraction* stratum corresponds to the minimal operating system-like functionality needed to run software on the higher levels.

In-Band Functions: This stratum consists of in-band packet processing functions such as packet filters, checksum validators, classifiers, diffserv schedulers, and traffic shapers.

Application Services: The application services stratum encompasses coarser-grained functions in the sense of programs running under active networking execution environments [1].

Coordination: This stratum supports out-of-band signalling protocols (e.g. RSVP) which carry out distributed coordination, including configuration and re-configuration of the lower strata.

3.2. Benefits of a Globally-Applied Component-Based Approach

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

- *is simple and uniform* – it allows the creation of services on all strata and provides a uniform run-time support for deployment, inspection of current configurations, 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 in protocol layer can directly access (subject to access policies) "layer-violating" information from other, non-adjacent, layers, which is increasing considered useful [2].

In our work we apply this approach to both network processor based routers and commodity PC-based router. 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. Implementation

As mentioned, our component model offers support for *loading* and *binding* of components residing in all strata of the design space of the programmable networking environment. We provide this functionality as part of our core architecture. The loading mechanism uses plug-in loaders to create an instance of one component in the specified container and the binding mechanism uses plug-in binders to accomplish the binding between individual components (between interfaces and receptacles).

As a consequence, we have implemented plug-in loaders that load components into Windows address spaces, Linux address spaces, and IXP1200 microengines. In the general case, the programmer may

either select a specific loader manually, or (more commonly) elect for transparency and let OpenCOM make the choice. In the former case, the programmer would use OpenCOM's reflective capabilities [7] to make the alternatives visible, and then interact with a specific loader. In the latter case, the selection is made on the basis of attributes attached to both components and loaders (e.g. a "*CPU-type*" or "*OS-type*" attribute). Loaders themselves may espouse a further level of choice (which may also be attribute driven) of which address space to load into. For example, a microengine loader (that comprehends each IXP1200 microengine as a separate address space) might make a choice of which microengine to use for a particular load request by taking into account factors such as resource usage, QoS, and security/ safety constraints. Furthermore, it is possible, using a "placement" meta-model supported by the microengine loader, to manually control this placement if desired.

In addition, we have implemented the following set of plug-in binding components (or *binders*):

- *vtable-based* – This binder was implemented as part of the original OpenCOM platform. It operates only in the Linux environment (on the host PC or StrongARM) and enables the binding of any component generated by a compiler whose binaries employ the vtable function-call convention.
- *shared memory* – We have developed a microengine-specific binder that uses shared memory (i.e., scratch memory, static and dynamic RAM - SRAM and SDRAM; see figure 1), to bind components that reside in different microengines. We also have a shared memory binder that binds a microengine-based component to a Linux-based component; and another that binds two components running in different Linux processes.
- *branch instruction* – This binder enables bindings between components on the same microengine. Essentially, a component is bound to another (cf. Netbind [3]) by rewriting a branch instruction so that execution jumps to the desired target.

The above functionality is made available through the following *capsule API*:

- `load(address_space, component);`
- `load(loader, address_space, component);`
- `unload(address_space, component);`
- `bind(receptacle, interface);`
- `bind(binder, receptacle, interface);`
- `unbind(receptacle, interface);`

In addition, the following meta-interfaces expose a meta-model of the loader and binder CFs:

- `install_loader(component, loader);`
- `remove_loader(component);`
- `install_binder(component, binder);`
- `remove_binder(component);`

These meta-interfaces are used to add/ remove loaders and binders to alter the set available to callers of *load()*, *unload()*, *bind()*, and *unbind()*.

5. Application Scenario

To demonstrate our approach, we present a configuration of our recently-implemented Router CF (that was mentioned in section 2.2) which covers strata 2, 3, and 4 (the OpenCOM runtime itself deals with stratum 1 by wrapping the underlying OS with CFs for thread management, buffer pool management etc.). The below scenario demonstrates how OpenCOM's (re)configuration capabilities can be used to extend the network services on a router at run-time. In addition, the scenario emphasizes the benefits of a single, uniformly-applied, component model, which allows configuration and reconfiguration of service components across several strata of the programmable network design space and across different hardware environments (i.e., a PC and an IXP1200-based router). It also shows how reconfiguration can be carried out in dimensions that have not been foreseen when the system was designed.

The Router CF configuration illustrated in Figure 2 (minus the dotted box) is a typical configuration for IP forwarding. It consists of several in-band components (stratum 2) on the "fast-path" of the router, namely a classifier and a forwarder, as well as scheduling components, an application service-level component (stratum 3) for the processing of IP options on the "slow-path", and a "routing protocol" CF in the control plane of the router (stratum 4).

To best exploit the capabilities of the different hardware elements of the IXP1200, we target the above functions at the hardware best suited to them. Thus, we deploy the "fast-path" components on the microengines, the IP options component on the StrongARM, and the routing protocol CF on the PC. Note that we can additionally exploit the multi-address-space capsule feature of OpenCOM to address security/ safety issues. For example, we can load untrusted components into separate address spaces (within the same capsule) so that they cannot maliciously or accidentally disrupt or crash the whole system.

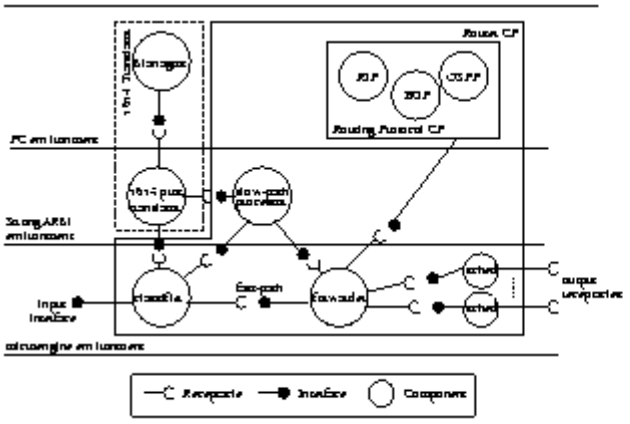

Figure 2. IPv6v4 translator application scenario

To illustrate run-time reconfigurability, we dynamically install IPv6-to-IPv4 protocol translation functionality (collectively called a "translator"), into the initial configuration (note that, like the Router CF itself, the translator is spread across different layers of the router architecture: while the actual protocol translation takes place on the StrongARM, management is performed on the PC). Such dynamic extensibility might be required to adapt to a network environment providing IPv6 support without needing to restart the network device. For example, if our system was running on a mobile PDA, we would only require IPv6 functionality when we become attached to a fixed network. When operating in a wireless network environment, we can save memory by omitting this functionality.

To integrate the translator we first attempt to load its two consituent components into the appropriate address spaces. This is achieved transparently (based on a "CPU-type" attribute attached to the components) by the loader CF. Furthermore, the CF checks that the components being loaded conform to its rules. We then obtain a new receptacle on the classifier, and arrange for this to be bound to the translator. An appropriate binder is selected transparently (by the binding CF). We could also use OpenCOM facilities [7] to ensure that the translator has adequate resources (e.g. in terms of its thread priorities, and buffer pool availability) to perform with a required level of QoS. Note that none of these steps need to have been foreseen when the initial configuration was defined, and that they are entirely decoupled from the basic functionality of the components involved.

6. Related Work

Although there exists a whole variety of research on component-based platforms for programmable routers (e.g. Click [10], NP-Click [14], VERA [9], NetBind [3], LARA++ [13]), few works support configuration and reconfiguration (i.e., adaptation, extension, evolution and removal of components) sufficiently. Most of them support initial configuration but do not support subsequent runtime re-configuration. Moreover, systems that do 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). We address this issue by designing appropriate CFs. 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 3). For example, VERA limits re-configurability to in-band functions and the hardware abstraction layer, whereas NetBind considers only in-band functions. LARA++, on the other hand, allows re-configurability on all strata, but lacks an uniform model to do so (i.e. different component models are used on the different layers). Villazón [16] introduces the use of reflection to support flexible configuration in active networks, but this work only addresses an architecture in which active nodes use reflection better to structure services. Essentially, the work defines a reflective architecture for configuration rather than (re-)configuration.

7. Conclusions

In this paper we have proposed a component model based on a single framework for all strata of the design space of programmable network platforms that facilitates the creation of services by flexibly loading and binding relevant components at a potentially fine granularity. We believe that the combination of components, reflection, and CFs offers a promising approach to the configuration and reconfiguration of services in networking environments. A key strength of our model is the uniform framework to load and bind both Linux and assembly-based components on heterogeneous hardware. 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 approach facilitates the extensibility and programmability of network processors based systems. These architectures are usually hard to program and, as a consequence, re-configuration is hardly considered on these "primi-

tive" 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 on assembly-based components. We do this by creating an "illusion" for the component developer that assembly-based components can be loaded and bound in the same way as Linux components.

8. 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). We would also like to thank Intel Corp for their generous donation of equipment, and the UK EPSRC for funding the bulk of our research.

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] A. Campbell, M. Kounavis, D. Villela, J. Vicente, H. de Meer, K. Miki, and K. 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.

[4] M. Clarke, G. 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.

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

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

[7] G. Coulson, B. 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.

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

[9] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *4th International Conference on Open Architectures and Network Programming (OPENARCH)*, April 2001.

[10] R. Morris, K. E., J. Jannotti, and M. Kaashoek. The Click Modular Router. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Charleston, SC, USA, December 1999.

[11] Mozilla Organization. XPCOM Project. `http://www.mozilla.org/projects/xpcom`, 2001.

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

[13] 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.

[14] 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.

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

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