A Globally-Applied Component Model for Programmable Networking

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

[1] Computing Department
Lancaster University
LA1 4YR Lancaster, UK
{ueyama, geoff, gordon, schmid, joolia, leek}@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. We argue that currently developed software frameworks for active and programmable networking do not provide a truly generic approach to the development, deployment, and management of services. Furthermore, current systems are typically targeted at a particular level of the programmable networking design space (e.g. at low-level, in-band, packet forwarding; or at high-level signaling) and/or at a particular hardware platform. In addition, most existing approaches, while they may address the initial *configuration* of systems, neglect dynamic *re*configuration of running systems. In this paper we present a reflective component-based approach that addresses these limitations. We show how our approach is applicable at all system levels, can be applied in heterogeneous hardware environments (specifically, commodity PC-based routers and network processor-based routers), and supports both initial configuration and dynamic reconfiguration. We especially address the latter point; we show the viability of our approach in (re)configuring services on an Intel IXP1200 network processor-based router.

1 Introduction

Although significant progress has been made in programmable networking, numerous research challenges remain. In particular, there remain important issues clustered around *configurability* and *heterogeneity*. Programmable networking systems must be highly configurable and, moreover, run-time reconfigurable, to meet requirements for dynamic fine-grained deployment, 24x7 operation, managed software evolution, dynamic quality of service (QoS) and resource management, and configurable security [19]. Similarly, these systems must be easily deployable in complex, multi-programming language, multi-operating system, and multi-hardware platform environments, and offer transparency and portability without sacrificing performance.

N. Wakamiya et al. (Eds.): IWAN 2003, LNCS 2982, pp. 202–214, 2004.

In recent years significant progress has been made in the design and implementation of generic *component-based* systems-building methodologies (e.g. [THINK [11], OpenCOM [10], Knit [18]]) which, because of their emphasis on fine-grained configurability, reconfigurability, and system heterogeneity, have interesting implications, we believe, for programmable networking research. Based on these observations, we have initiated a project that is attempting to apply component-based principles to programmable networking environments.

Deriving from this project, this paper presents the design and implementation of a component-based architecture for programmable networking systems, which provides an integrated means of developing, deploying and managing such systems. The proposed architecture consists of i) a generic, reflective, component model that, we argue, can be uniformly applied at *all levels* of the programmable network design space (i.e., from fine-grained, low-level, in-band packet processing to high-level signaling and coordination), and ii) an extensible architecture of component frameworks that are built in terms of the generic component model and support plug-in functionality in diverse areas of the programmable network design space. The claimed benefits of the proposed architecture are detailed in section 3.

The paper is structured as follows. First, section 2 provides background to our approach: it presents the basics of our reflective component model, introduces the concept of component frameworks, and provides a brief overview of the network processor-based router environment in which we are primarily working. Section 3 then presents our "globally applied" approach to network programmability, and section 4 discusses our results and 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 offers conclusions.

2 Background

2.1 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 (see below for definition of "capsule") and it imposes minimal overhead on cross-component invocation. It is currently implemented on top of a subset of Mozilla's XPCOM platform [16].

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 micro-engines 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 unit of service requirement on a component and are used to make explicit the dependency of one component on others.

Bindings: are associations between receptacles and interfaces in the same capsule: a binding represents a communication path between one receptacle and one interface. Bindings in the original OpenCOM implementation [10] 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 (see section 4).

Importantly, OpenCOM also supports a range of built-in *reflective meta-models* which form the basis of configuration and reconfiguration in our approach. In particular, it supports an *architecture* meta-model that represents compositions of components as a graph, and allows the programmer to manipulate this graph to effect corresponding changes on the underlying systems (e.g. in terms of inserting/ deleting components and making/ breaking bindings). It also supports an *interception* meta-model that enables the insertion of arbitrary code within bindings that is executed when a call is made across the binding; and an *interface* meta-model that allows the programmer to introspect on available interface and receptacle types on a component. There is also a *resources* meta-model that represents types and quantities of resources dedicated to various components or sets of components. More details on the reflective meta-models are given in [10].

OpenCOM deploys a singleton per-capsule runtime, which manages a repository of component types and provides interfaces for the creation and deletion of components and for binding/ unbinding. All create/delete/bind/unbind requests are reflected in the above-mentioned architecture meta-model.

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 below.

2.2 Component Frameworks

Although necessary, the component model's explicit representation of dependencies and its reflective meta-models are not in themselves sufficient for the management of reconfiguration. In particular, their genericity precludes *specific*

competencies in imposing and policing domain-imposed constraints on reconfiguration. For example, they cannot prevent the nonsensical replacement of an H.263 encoder with an MPEG encoder, or mandate that a packet scheduler must always receive its input from a packet classifier. Such semantic constraints are essential if we are to ensure meaningful configuration and reconfiguration, and therefore the system must provide support for their expression and enforcement.

To add the necessary dimension of specificity and constraint, we apply the notion of *component frameworks*. These were originally defined by Szyperski [22] as "collections of rules and interfaces that govern the interaction of a set of components 'plugged into' them" (see figure 4). More concretely, component frameworks (hereafter, CFs) are targeted at a specific domain and embody 'rules and interfaces' that make sense in that domain. 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 [10]. (In the rest of the paper we use the shorthand "plug-in" to refer to a component that is plugged into a CF.)

Essentially, CFs serve as "life-support environments" for components in a particular domain of application. They contain arbitrary CF-specific state, embody shared services for plug-ins, and actively police their plug-ins to ensure that they conform to their domain-specific rules and interfaces (e.g. interfaces can be inspected at run-time using the interface meta-model). CFs can support multiple instances of multiple types of plug-in, and plug-ins can either be independent of each other or can be bound together in arbitrary configurations (as long as these conform to the rules imposed by the host CF).

Overall, our component-based approach is strongly predicated on the benefits of reflection and CFs: reflection provides an open and flexible architecture by supporting the inspection, adaptation and extension of underlying component topologies, while CFs provide runtime structure for domain-specific configurations of components and encapsulate domain-specific rules and constraints.

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 [9]. The use of the Router CF is illustrated in section 5.

2.3 The Radisys Intel IXP1200-Based Router

The Radisys Development Platform, on which we are basing the bulk of our router implementation work, is based on the Intel IXP1200 network processor. This is an Intel-proprietary architecture that conforms to the Intel IXA architecture [12]. It is attached to a host PC and combines a StrongARM processor (running Linux) with six independent 32-bit RISC processors called microengines, which have hardware multithread support, and are used for fast-path processing. There are also various types of memories, buses, and specialised hardware assists available on the processor. The outline architecture is illustrated in 1.

Programming support on the router is very primitive, especially in the microengine environment. This environment does not run an operating system and all

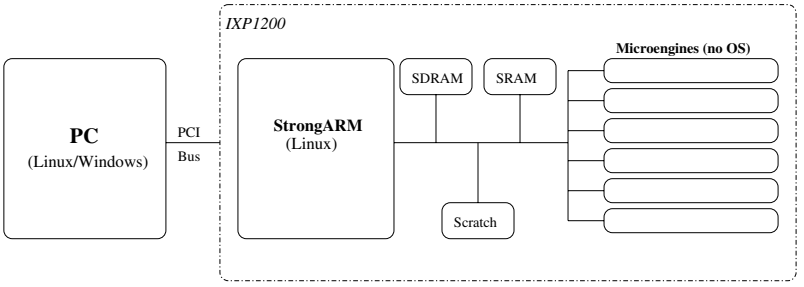

Fig. 1. Testbed–PC and IXP1200 router

resource allocation and inter-process communication has to be manually managed. The current Intel-provided programming environment has no support for dynamic reconfiguration.

3 Our Approach to Building Programmable Networking Software

3.1 The Design-Space of Programmable Networking

We conceptually partition the global design space of programmable networking [9] into four layers or *strata*. We use the term "stratum" rather than "layer" to avoid confusion with layered protocol architectures. The four strata are illustrated in figure 2.

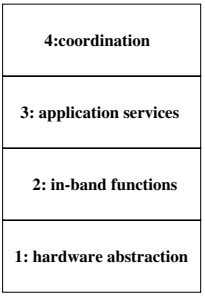

Fig. 2. Stratification of the Programmable Networking Design Space

The *hardware abstraction* stratum contains necessary hardware and operating system functionality such as threads, memory, I/O, and library loading. Services in this stratum are often implemented as wrappers around underlying native facilities in order to support heterogeneous platforms. The *in-band functions* stratum consists of packet processing functions like packet filters, checksum validators, classifiers, diffserv schedulers, and traffic shapers. Given that these are

low-level, in-band and fine-grained (and therefore highly performance critical) performance is a key concern in this stratum. The *application services* stratum encompasses coarser-grained functions (in the active networking execution environment sense [1]). These are less performance critical as they act on pre-selected packet flows in application specific ways (e.g. per-flow media filters). Finally, the *coordination* stratum supports out-of-band signalling protocols which carry out distributed coordination, including configuration and reconfiguration of the lower strata. It includes, for example, routing protocols, signalling protocols such as RSVP, or architectures that allow resource allocation in dynamic private virtual networks (e.g. Genesis [4], Draco [13], or Darwin [6]).

3.2 Benefits of a Globally-Applied Component-Based Approach

We argue that our proposed approach of uniformly applying the same component model, supported by the notions of reflection and CFs, yields the following potential benefits:

- *a simple and uniform model* (based on OpenCOM)–we provide a simple and uniform programming model for thecreation of services in all strata, and also a uniform run-time environment for deployment, and (re)configuration; a key aspect here is the separation of concerns between building systems (using the basic component model) and managing/ reconfiguring them (using a combination of reflective meta-models and CFs).
- *enables bespoke software configurations*–according to the composition of CFs in each stratum, desired functionality can be achieved while minimising memory footprint; trade-offs will vary for different system 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, e.g., the link layer; this kind of "layer-breaking" is of growing interest in the research community [2].

We are applying our approach in PC-based routers as well as the Intel IXP1200 environment discussed above. This heterogeneity is fundamental to validate our claim of a generic model. We also strive to implement this model without compromising performance so that we can reasonably apply the approach in the lower as well as the higher strata (see section 4).

4 Implementation

To pursue the approach, we are extending OpenCOM to enable highly configurable *loading* and *binding* of components in all strata of the programmable networking design space. As mentioned in section 2.1, the core architecture supports these two functions, loading and binding, as CFs.

The API for loading and binding is as follows (*capsule_id* specifies the capsule where the component will be loaded or already resides–see figure 3; *Component_guid*, *interface_guid* and *receptacle_guid* are globally unique IDs for components, interfaces and receptacles respectively):

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

These API calls are IDL-specified for language independence, and the CFs underlying them are system-independent. However, underlying these CFs are system-dependent plug-in loader and binder components which are dynamically installed/ removed by means of a meta-interface on the CFs. As explained below, the ability to transparently access this system-dependent functionality from a system-independent API is key to the power and generality of our design.

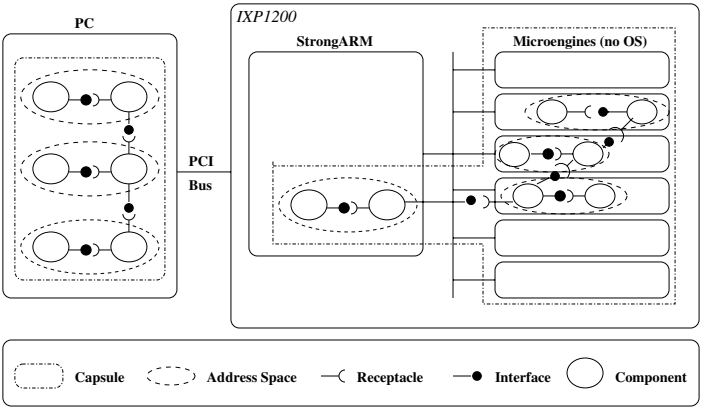

Fig. 3. Multi-address-space capsules

We have implemented plug-in loader components (or *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 the CF make the choice. In the former case, the programmer would use the architecture metamodel 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 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 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 [5]) by rewriting a branch instruction so that execution jumps to the desired target.

As with the loader CF, the programmer may either select a binder manually, or elect for transparency and let the binder CF make the choice on the basis of attributes and heuristics.

Importantly, it is not necessary in our architecture to execute the OpenCOM runtime on the microengines (which would, in any case, be infeasible). Instead, the pluggable loader/ binder frameworks running in a Linux process on the StrongARM control processor subsume all the microengine specifics. These are then encapsulated within specific plug-in loaders/ binders. The end result is that the programmer has the benefit of full transparency while retaining the full generality of the programming model regardless of which environment his/her code is running in.

We have not yet carried out a comprehensive performance evaluation of the IXP1200-specific loaders and binders. We observe, however, that the overhead of establishing and reconfiguring bindings is entirely "out-of-band" and does not impact data flowing between components. The major factor impacting the overhead of in-band inter-component communication is the choice of binding mechanism involved. As we are using essentially the same mechanisms as other well-evaluated systems (i.e. Netbind and MicroACE [12]) there is no reason to expect that performance should suffer. The one OpenCOM-specific feature that might significantly impact performance is the *number* of inter-component bindings involved – which is a function of the granularity of components. Again, based on evaluations of previous fine-grained systems such as Click [15] we have no a-priori reason to believe that fine-grained componentisation is necessarily problematic.

5 Application Scenario

To demonstrate our approach, we present a configuration of our recently-implemented Router CF (that was mentioned in 2.1) which covers strata 2, 3, and 4 (the OpenCOM runtime itself deals with statum 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 4 (minus the dotted box) is a typical configuration for IP forwarding. It consists of several low-level, 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).

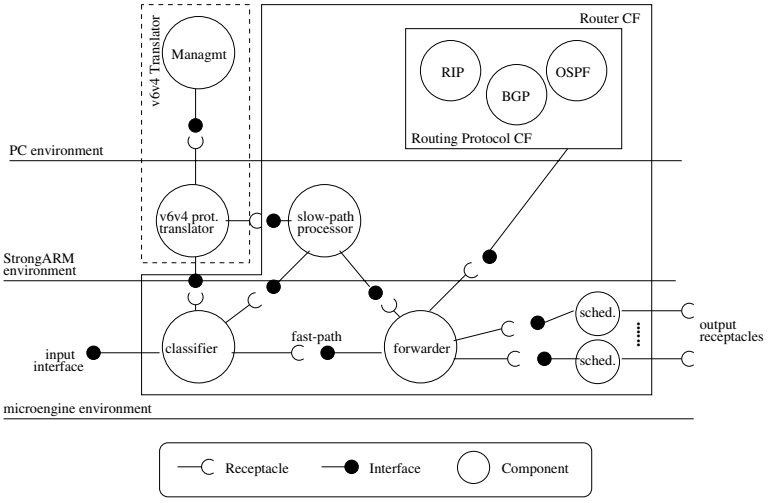

Fig. 4. IPv6v4 translator application scenario

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.

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 (e.g. the interface meta-model is used to ensure that they support appropriate interfaces/ receptacles). We then obtain a new receptacle on the classifier, and, by manipulating the architecture meta-model, arrange for this to be bound to the translator. An appropriate binder is selected transparently (by the binding CF). We could also use the resources meta-model 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. And, we could additionally add an interceptor to the binding to count the number of IPv6 packets actually forwarded. 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

The various NodeOS implementations (e.g. the Scout-based implementation reported in [17]) address generic system support for active and programmable networking. However, they do not focus primarily on building programmability in terms of componentisation – rather they are targeted at the support of coarse-grained execution environments (mainly strata 2 and 3) which themselves may or may not internally support componentisation.

Recent work at FOKUS, Berlin [21] discusses a flexible component-based architecture for programmable routers. Like our work, this aims at language independence and system heterogeneity. However, the initial implementation work has been in a Java environment which has to date precluded applying the approach in the fast-path, and in network processor-based routers. In addition, the work focuses of the managment of dynamic deployment rather than unplanned reconfiguration.

VERA [14] is a strata 2 and 3 extensible router architecture that explicitly supports adding new components, such as packet forwarders, to routers. VERA is also deployed on network processor-based routers (specifically, Intel IXP1200-based). However, VERA's architecture is limited in its flexibility: extensions can only be added at pre-defined "hooks" provided by the system. In addition, key elements of the architecture itself (e.g. the router and hardware abstractions, as well as the distributed router OS) can not be removed or changed. Furthermore, VERA's component model does not address the provision of services belonging to all strata of the networking design space.

NetBind [5] proposes an approach to construct in-band packet-forwarding paths on a network processor-based router (again, based on the IXP1200), and to reconfigure forwarding paths dynamically. Low latency in these paths, despite the possibility of changing them at run-time, is one of the outstanding features of NetBind. This is achieved by patching branch instructions at the machine code level which involves minimal overhead (we borrow this technique from Netbind; see section 4). Nevertheless, NetBind is not a generic framework for adding new services on network processor-based routers; e.g., it does not address strata 3 or 4. Instead, it aims to tackle solely the construction of dynamic data paths. In addition, like VERA, there are many parts of the architecture which can neither be configured at deploy-time, nor reconfigured at run-time.

Click [15] is an extensible component-based router targeted at PC-based environments. A Click router is constructed by selecting from a library of components called "elements" that carry out fine-grained tasks, and which are aggregated into a graph structure. Click offers extensibility by providing a straightforward and flexible means of defining new configurations; but, crucially, it does not support dynamic reconfiguration. Although Click was not initially deployed on network processor based routers, NP-Click [20] is a recent implementation for such environments (but this still suffers from the same lack of dynamic reconfiguration).

Villazón [23] 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.

Overall, while there has been substantial research addressing the need for configurability in active and programmable networks, few approaches address both configuration and *re*configuration in a fully general and comprehensive manner. For example, some systems, like VERA, support reconfiguration, but only in pre-determined ways. Furthermore, most of these systems are *partial*, addressing either high-level concerns (e.g. the Villazón work) or low-level concerns (e.g. Netbind). None of them proposes an integrated architecture allowing configuration and reconfiguration of services running in all strata.

7 Conclusions

In this paper we have proposed a generic component-based model for programmable networking systems that enables (re)configuration of systems using reflective techniques and CFs. A key strength of this model is that it is based on a platform and language-independent approach, which can be applied across different network processing hardware. Furthermore, we argue that the proposed framework can be applied to configure and reconfigure component-based services on all levels (strata) of the design space of a programmable network system.

We believe that such a globally-applied component model has the potential to greatly facilitate the (re)configuration of services, as a single, unified programming model is used to compose and adapt services across the different strata of the design space.

Furthermore, we expect our framework to considerably facilitate the programmability and reconfigurability of network processor-based systems. These architectures are widely acknowledged to be very difficult to program [8] and, as a consequence, reconfiguration is hardly considered on these "primitive" platforms. However, the provision of an OpenCOM-based programming model for these architectures gives the programmer a friendly interface (abstraction) with which to orchestrate low-level functions in a uniform manner, and also facilitates the imposition, via the CF concept, of domain-specific constraints on these routers.

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 equipament, 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`, 2002.
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 Networking. 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.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.
12. Intel. Intel IXP1200. `http://www.intel.com/IXA`, 2002.
13. R. Isaacs and I. Leslie. Support for Resource-Assured and Dynamic Virtual Private Networks. In *JSAC Special Issue on Active and Programmable Networks*, 2001.
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. 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.
16. Mozilla Organization. XPCOM Project. `http://www.mozilla.org/projects/xpcom`, 2001.
17. L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.
18. 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.
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. M. Solarski, M. Bossardt, and T. Becker. Component-based Deployment and Management of Services in Active Networks. In *Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN*, Zurich, Switzerland, December 2002.
22. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
23. A. Villazón. A Reflective Active Network Node. In *IWAN*, pages 87–101, 2000.