NETKIT: A Software Component-Based Approach to Programmable Networking

Geoff Coulson, Gordon Blair,
David Hutchison, Ackbar
Joolia, Kevin Lee, Jo
Ueyama, Antonio Gomes,
Yimin Ye
Computing Dept.,
Lancaster University
Lancaster LA1 4YR, UK
+44 1524 593054

geoff@comp.lancs.ac.uk

ABSTRACT

While there has already been significant research in support of openness and programmability in networks, this paper argues that there remains a need for generic support for the integrated development, deployment and management of programmable networking software. We further argue that this support should explicitly address the management of run-time reconfiguration of systems, and should be independent of any particular programming paradigm (e.g. active networking or open signaling), programming language, or hardware/ operating system platform. In line with these aims, we outline an approach to the structuring of programmable networking software in terms of a ubiquitously applied software component model that can accommodate all levels of a programmable networking system from low-level system support, to in-band packet handling, to active networking execution environments to signaling and coordination.

General Terms

Management, Design, Reliability, Experimentation, Standardization.

Keywords

Programmable networking, components, reflection, middleware.

1. INTRODUCTION

There are steadily increasing demands for openness and programmability in today's networks. In particular, both network operators and users want to be able to dynamically introduce new mechanisms into the network with ease and convenience. Examples of such mechanisms are quality of service (QoS) elements like intserv/ diffserv/ MPLS/ RSVP/ RED/ ECN; in-band media-stream filters; network address translators; firewalls and other security mechanisms; and application-level routers (e.g. for multicast or peer-to-peer networking).

The requirement for openness and programmability is further underlined by the desire to dynamically deploy emerging services like ubiquitous computing, ad-hoc networking, dynamic private virtual networks, and e-Science Grids. Furthermore, there is an associated requirement for *manageability* of such mechanisms and services so that they can be flexibly configured (including deployment, instantiation and initialisation) and *re*configured (including run-time adaptation, extension, evolution and removal) with ease and convenience.

The view expressed in this paper is that, while there has already been significant research in support of such requirements, there remains a need for *generic programming model support* to facilitate programmable networking. Ideally, this support should be programming language-, platform-, and even *paradigm*-independent (see below) and should explicitly facilitate the management of both configuration and reconfiguration as defined above.

The approach we are pursuing is to apply the notion of *software components* [39] to the programmable networking environment. According to Szyperski [39], software components *i*) have formally specified interfaces, *ii*) are packaged and distributed in binary form, and *iii*) can be dynamically deployed in address spaces. Unlike other research that advocates a component-based approach (e.g. [28] and [37]) we envisage components being uniformly applied at *all* levels of the programmable networking environment from fine-grained, low-level, in-band packet processing functions, to high-level signaling and coordination functions. In outline, we envisage on-demand component loading/ unloading and binding/ unbinding services as the basis of both configuration and reconfiguration.

The remainder of this paper is structured as follows. First, §2 provides an overview and analysis of the field of programmable networks. Next, §3 presents our generic component-based approach to programmable networking together with a discussion of the potential benefits of the approach. Then, in §4, we discuss current design and implementation work in our recently initiated NETKIT project that follows the component-based approach. As

the project is at a relatively early stage, the discussion is in terms of work-in-progress rather than definitive results. Finally, §5 discusses some related work (in addition to that surveyed in §2), and §6 presents our conclusions.

2. PROGRAMMABLE NETWORKING RESEARCH

2.1 The Design Space of Programmable Networking

The design space of programmable networking can be broadly represented in terms of the (highly abstract) reference architecture depicted in Figure 1.

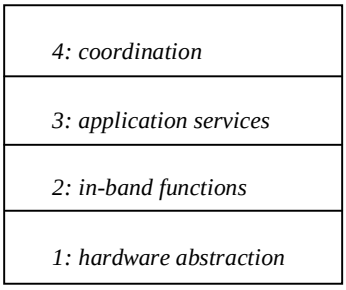

Figure 1: A reference architecture for programmable networking.

In this architecture, a *hardware abstraction* stratum (we use the term 'stratum' rather than 'layer' to avoid confusion with layered protocol architectures) contains the minimal operating system (OS) functionality (e.g. threads, memory allocation, and access to network hardware) that must be available on any participating node (e.g. router) to support higher-level network programmability. Services in this stratum typically try to mask underlying hardware heterogeneity so that, say, a standard PC-based router and a specialised programmable router (e.g. a router based on the Intel IXP1200 [23] network processor which provides multiple processors and distributed/ hierarchical memory arrays) will look as similar as possible to the higher strata. Furthermore, the nature of the stratum 1 services largely determines the QoS (e.g. predictability, throughput and latency) capabilities of programmable networking software in the higher strata.

Second, an *in-band functions* stratum comprises packet processing functions (e.g. packet filters, checksum validators, classifiers, diffserv schedulers, shapers, etc.) that touch all or most packets. As these functions are inherently low-level, in-band, and fine-grained, this is a highly performance critical area in which machine instructions must be counted with care.

Third, an *application services* stratum comprises coarser-grained 'programs'—in the active networking execution-environment sense [1]—that are less performance critical and act on pre-selected packet flows in application specific ways (e.g. per-flow media filters).

Finally, a *coordination stratum* includes or supports out-of-band signaling protocols that perform distributed coordination (e.g. configuration, reconfiguration) of the lower strata. Examples are RSVP, or protocols that coordinate resource allocation on a set of routers participating in a dynamic private virtual network, as employed by systems like Genesis [6], Draco [24] or Darwin [10].

2.2 Current Paradigms

Historically, there have been two main paradigmatic approaches to the provision of openness and programmability in networks: First, in the *active networking* paradigm (see, e.g., [1], [15], [34], [16], [17], [18], [19]) 'active packets' called carry programs that execute on 'active nodes', often in a Java-based execution-environment. Second, in the *open signaling* paradigm (see, e.g., [6], [10], [24]), routers export 'control interfaces' through which they can be remotely (re)configured by out-of-band, application specific, signaling protocols. More recently, a third approach—we'll call it out-of-band active—has become popular (see, e.g. [7], [11], [13], [22], [28]). In this approach, downloadable modules are dynamically installed onto routers through some (often unspecified) out-of-band mechanism. These systems vary in their support for kernel vs. user space modules, and whether or not in-band functions can be reconfigured.

Overall, active networking is the most dynamic of the three approaches and can operate on the finest time scales. However, it is not as easy to deploy as the other approaches, is perceived as more prone to security threats, and tends to be language specific (often Java). While being coarser grained and less dynamic, the open signaling approach is typically easier to deploy (especially for complex services like dynamic private virtual networks), easier to secure, and typically performs better than Java-based active networking systems (especially at the level of fundamental QoS elements like intserv or diffserv). The out-of-band active approach is between the two classic approaches in terms of both deployability and security vulnerability.

Combining the above analysis with that of §2.1, it is interesting to observe that much programmable networking research addresses only a *subset* of the concerns implied in Figure 1. In particular:

- active networking research tends to focus on stratum 1 (e.g. the Scout implementation of NodeOS [34], [35]) and stratum 3 (the performance requirements of stratum 2 typically cannot be met in a Java-based execution-environment, and stratum 4 coordination is typically left to the 'application')[1];
- open signaling approaches focus mostly on strata 2 and 4 (typically, router control interfaces enable stratum 2 configurability but do not support stratum 3 functions and completely hide stratum 1);

[1] Some active networking implementations (e.g., the Scout NodeOS implementation reported in [35], and the Lancaster work on LARA++ [11]) do address stratum 2 as well as strata 1 and 3. However, there is typically a distinction drawn between an in-kernel 'fast path' environment for 'default' packet handling, and a less efficient, user-space, environment for configurable/ extensible packet handling code. While the performance deficit is not so great as in Java-based execution environments, it remains true that the custom path suffers in terms of performance while the fast path suffers in terms of flexibility. It is not so necessary to face this trade-off in the open signalling approaches discussed next.

It can also be observed that most out-of-band active systems address only stratum 2 and/ or stratum 3 concerns (sometimes stratum 1 is partially addressed as well). For example, the Click modular router [28], the NetBind component binding system [7], Washington University's pluggable router framework [13], and the IEEE P1520 router component model [22] are all targeted at stratum 2. (Click employs a fine grained C++-based component model with flexible support for the configuration of packet scheduling, route lookup and queue drop modules etc.; NetBind is similar in concept but is lower-level and targeted at network processors; the Washington work is a framework for pluggable per-flow modules in the NetBSD environment; P1520 is working towards a standardised, language-independent, component model for modular routers.) Slightly more generally, the Knit system [37] supports stratum 2 (and stratum 1 also) in the form of a component model that has been used for both OS and in-band packet handling functions. However, Knit is supported only on conventional workstation architectures, not on specialised programmable routers. The VERA extensible router architecture [27] supports stratum 1 and stratum 2 on a wider range of router architectures but offers a far less general and flexible component model.

Overall, what appears to be missing from the state-of-the-art is a generic framework that is *both* paradigm-independent *and* equally applicable to all strata of the reference architecture.

2.3 Run-time Reconfiguration

It can also be strongly argued that support for *run-time reconfiguration* is inadequately addressed by current research. For example, while the above-cited component models support the initial configuration of components, none of them explicitly support the subsequent reconfiguration of a running system (e.g. to accommodate newly discovered services in a ubiquitous computing environment; to reconfigure an ad-hoc network; or to adjust the resources allocated to a dynamic private virtual network). Furthermore, systems that *do* allow reconfigurability (e.g. most active networking systems) still fail to adequately support the *management of system integrity* over reconfiguration operations (e.g. ensuring that firewall updates are applied universally and consistently; or that a change in a source media-filter type is accompanied by a compatible change at the sink; or that allocating more resources to one dynamic private virtual network does not lead to starvation in another).

There has been some work on the use of *reflection* to address such management related issues. For example, [21] describes reflective support for checking the integrity of coordination/ control code being downloaded into an execution environment, and [40] further supports some degree of dynamic reconfiguration of downloaded control code. More recently, [46] supports reconfiguration through dynamic linking, but not in the context of a principled reflective component model. On the other hand, [47] provides a reflective component model but focuses on a flexible deployment architecture rather than on fine-grained reconfiguration.

However, this work is again *partial*; it typically addresses only execution environment and coordination strata concerns (i.e. strata 3 and 4 in the reference architecture), and is programming language specific (Java).

2.4 Summary

Overall, we argue that while there has been significant research in programmable networking, most work to date has focused on specific and limited areas of the overall design space. This lack of recognition of the 'big picture' has led to a proliferation of programmable networking solutions that are on the one hand partial and on the other hand incapable of being easily combined to produce more comprehensive solutions. More specifically, there has been insufficient attention paid to the development of 'integrated' solutions that are capable of offering:

- a language-, platform- and paradigm-independent programming model that can be uniformly applied in all four strata of the reference architecture without unacceptable compromise (e.g. in terms of performance), and
- flexible support for both the configuration (e.g. deployment, instantiation, initialisation) and run-time reconfiguration (e.g., adaptation, extension, evolution, removal) of mechanisms and services in all strata.

Our approach to the provision of such an integrated solution is detailed in the rest of this paper.

3. TOWARDS A COMPONENT-BASED APPROACH TO PROGRAMMABLE NETWORKING

3.1 Support for Components

3.1.1 A Component-Based Computational Model

To realise the software component concept in the programmable networking environment, we first need a *component-based computational model* that satisfies the particular demands of that environment. As the basis of NETKIT, we employ an abstract, minimal, generic, language-independent, component-based computational model that is derived from our previous work on component-based middleware [8].

The key concepts embodied by the computational model are: *component*, *interface*, *receptacle*, *binding*, and *capsule*. These are illustrated in Figure 2 which shows two components inside a capsule (dotted lines). The component at the top left supports two interfaces (small circles) and one receptacle (small cup). This receptacle is bound to one of the interfaces of the bottom right component.

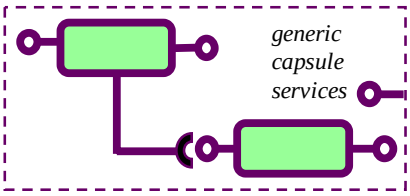

Figure 2: The component-based computational model.

Components can support any number of interfaces and receptacles. Interfaces are strongly typed and consist of a set of datatype definitions and operation signatures; they are defined in a programming-language-independent interface definition language such as OMG IDL or Microsoft IDL (we use OMG

IDL). Receptacles are 'anti-interfaces': whereas an interface expresses a unit of service *provision*, a receptacle expresses a unit of service *requirement*. (The term 'receptacle' is also employed by the CORBA Component Model [36]. The concept itself appears in various other component models under various names.) Receptacles are used to make explicit a dependency of one component on another. For example, if a component relies on a service of type *S*, it would declare a receptacle of type *S'* that would be bound at run-time to an interface instance of type *S* (which would be provided by some other component). The fact that dependencies are explicitly represented means that when a component is dynamically loaded it is possible to determine what other components and interfaces must be present for it to work correctly. This is a crucial enabler for 'third-party' configuration and dynamic reconfiguration of component topologies.

Bindings are associations between receptacles and interfaces that reside in the same capsule (and are type compatible). They are assumed to be implemented minimally and with negligible or low overhead. The viability of the component model at fine granularities, particularly in demanding areas like in-band packet processing, is heavily dependent on the degree of this overhead which must be comparable to, or less than, the overhead of a function call in a language like C. It is important to note that, as bindings are abstract, there is no prescription of a particular underlying implementation. This fact is heavily exploited in our current implementation work, as discussed in §4.2, which employs multiple alternative implementations of binding.

Finally, *capsules* provide a run-time environment for a set of component instances that are mutually participating in bindings. Capsules are typically, but not necessarily (see section §4.2 below), implemented as address spaces. The central role of capsules is to provide generic services for dynamically loading and unloading components, and for creating and destroying bindings. As well as being available from within the capsule in a third-party manner, these services can be made available from outside the capsule to support *external* third-party loading and binding[2]. This is useful to enable bootstrapping and third-party management of capsules (possibly from a remote site). In the programmable networking environment, it must additionally be possible to render the (un)loading and (un)binding of components subject to *security constraints* (i.e. to constrain who has rights to deploy, use, bind, reconfigure, etc.) and *safety constraints* (i.e. limits on what components can do to their host node). While *policy* in these areas is clearly application dependent, basic security and safety *mechanisms* should be built into the component model itself (e.g., the capsule) wherever possible and appropriate.

Although they may appear superficially similar, capsules are very different from active networking 'execution environments' (e.g. [ANTS,01]). Capsules are a minimal bootstrapping facility and are neutral with respect to programming language and API (beyond the very minimal load/ unload, bind/ unbind 'meta-API' outlined above). Capsules form the basis of a generic component model that, in turn, serves as the basis for any desired programmable networking functionality (including the construction of execution environments, which in our architecture would be implemented as component frameworks—see §3.3).

3.1.2 Portability Considerations

Portability is a crucial issue for us; we need to deploy the component model on a wide range of hardware platforms, from standard PCs to a variety of specialised programmable router platforms.

The obvious approach to portability is to define a single 'standard' OS-level API that all hardware platforms must support. Unfortunately, this simple approach has major drawbacks. First, some platforms will suffer sub-optimal performance because the abstractions employed by a necessarily 'lowest common denominator' API will tend to map better to some platforms than others (e.g. abstractions that implicitly assume shared memory may be hard to implement efficiently in a distributed memory environment). Second, a standard API precludes the exploitation of specialised platform-specific hardware (e.g. the availability of 'microengine' processors—as on the Intel IXP1200—or direct access to I/O ports). And, third, the work involved in porting a comprehensive API is likely to be significant in itself.

To avoid these difficulties, we adopt an approach to portability that is strongly influenced by radical micro-kernel architectures like L2 [30] and Think [41]. More specifically, we define two levels of portability. The first level comprises the component model itself; this is kept as simple as possible, and relies on an absolute minimum of system support so that it can be readily ported. Essentially, all that is needed is a sufficient implementation of capsules, including the capability to load/ unload executables and make/break bindings. The second level, which comprises all further system-oriented and hardware specific functionality (stratum 1) is then implemented *in terms of the component model itself*. This includes platform specifics like network card APIs, as well as generic OS-oriented APIs for threads, buffers, inter-capsule communication, etc.

A key benefit of this approach, apart from facilitating porting, is that only those stratum 1 services that are *actually required* on any particular platform need be ported and deployed. At the same time, thanks to the component model's explicit representation of dependencies, services that are not initially needed can be brought in later if requirements change/ evolve.

3.2 Reflection: Basic Support for Reconfiguration

Beyond the capability to construct component configurations (as provided by the basic component model outlined above), there is the further requirement, identified in §2.3, to support run-time *reconfiguration* of components in a generic and principled way. This breaks down into two areas: *adaptation* (to change behaviour along dimensions that are foreseen at deployment time), and *extension* (to add new behaviour unforeseen at deployment time). Furthermore, there is an associated

[2] This implies that the capsule's loader must include simple protocol support for remote access. We provide a 'bootstrapping' TCP/IP implementation on each NETKIT enabled router for this purpose. To provide more comprehensive remote access, our approach, based on our previous work [8], would be to deploy CFs in the capsule that provide appropriate middleware functionality.

requirement to first be able to *inspect* current configurations as the basis of subsequent adaptation and extension.

We employ the notion of reflection [31] to support such inspection, adaptation, and extension. Essentially, reflection is a pattern for opening up 'black box' systems to inspection, adaptation and extension. In abstract terms, this is achieved by invoking a so-called *meta-interface* on the system (see figure 3) to yield one or more *meta-models* of the system that can be inspected, adapted and extended. A defining feature of reflection is that these meta-models (which are said to reside at the *meta-level*) relate to the underlying system (referred to as the *base-level*) in a *causally connected* manner. This means that a change made to a meta-model implicitly causes a corresponding change in the underlying system, and vice versa. As an example, a topological graph-like meta-model (as in figure 3) could be used to explicitly represent the implicit topology of a composition of components—e.g. a fine-grained component-based packet forwarder à la Click [28]. Thanks to causal connection, when the graph is manipulated, e.g. by deleting or redirecting an arc, the underlying configuration is changed correspondingly (e.g. in terms of bindings).

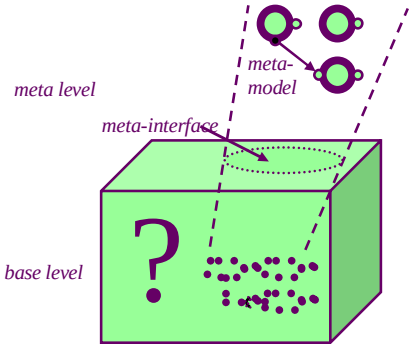

Figure 3: The concept of reflection.

Examples of reflective meta-models that we employ in our current work are as follows:

- an *architecture* meta-model which provides inspection, adaptation and extension of component compositions (as above),
- an *interception* meta-model which supports pre- and post-method call interception of invocations being made across bindings,
- an *interface* meta-model which supports the navigation of interfaces and receptacles on a component (cf. MS COM's 'IUnknown' convention), and inspection of interface/ receptacle signatures (cf. standard Java reflection in which interfaces can be discovered and inspected at run-time), and
- a *resources* meta-model that represent types and quantities of resource dedicated to various components or sets of components].

Detailed discussions of the first three of these meta models can be found in [2]. Detail on the resources meta-model is available in [3].

3.3 Component Frameworks: Constraining Reconfiguration and Providing Structure

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 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, and also to provide *structure* for domain-specific component configurations, we apply the notion of *component frameworks*. These were originally defined by Szyperski [39] 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, we might employ a *protocol CF* that embodies knowledge, in the form of appropriate rules and interfaces, about the configuration (and reconfiguration) of the 'plugged-in' protocols that it hosts (e.g. "you may not place an IP component on top of a TCP component"). Similarly, a packet-forwarding CF might accept packet-scheduler plug-ins; or a media-stream filtering CF might accept various media codecs as plug-ins.

Essentially, CFs serve as 'life-support environments' for components in a particular domain or application area. 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 reflection). 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).

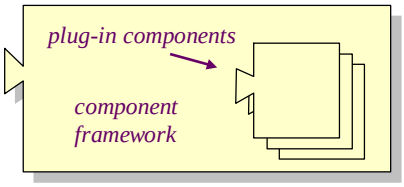

Figure 4: The concept of component frameworks.

CFs themselves are packaged as components. One implication of this is that, like any other component, CFs can be loaded/ unloaded dynamically. Another implication is that we can *nest* CFs to gain the benefits of hierarchical composition. For example, we have previously built a whole middleware infrastructure as a nested set of CFs [8].

To support reconfigurability that is consistent with domain-specific constraints, CFs can also provide CF-specific reflective meta-models that embody domain specific semantics. These are

typically layered on top of one or more of the generic meta-models mentioned at the end of §3.2. For example, a protocol CF could constrain an architecture meta-model to accept only linear topologies. In addition, CFs often require their plug-ins to support pre- and post- reconfiguration operations so that the host CF can ensure that they are in a dormant state before being reconfigured and can secure their state over reconfiguration operations.

3.4 Potential Benefits

The most obvious potential benefit of the proposed approach is that its ubiquitously-applied component model promises a uniform environment for the development, configuration, and reconfiguration of programmable networking software at all levels of the system and at any appropriate granularity and using any appropriate programming language. For example, functions as diverse as in-band packet handling and signaling can be developed, deployed, configured and reconfigured in a common manner and can rely on common support (such as dynamic remote instantiation, reflective services, and generic mechanism level security and safety support). In addition, the approach is, in principle, sufficiently general to accommodate any of the currently popular programmable networking paradigms (active networking, open signaling or application-level active networking). Essentially, all of these (e.g. an active networking EE) can be implemented in terms of CFs. Also, because components are language independent, portable, and (hopefully) can be applied at a wide range of granularities, they offer a solid basis for the incremental deployment of *existing* programmable networking software into a common component-based environment.

At a more detailed level, the fact that they are explicitly aware of their dependencies means that components can be (automatically) loaded on demand by their host CF so that only functionality that is actually needed at any given time need be resident on each node. Thus, a JVM instance (wrapped as a component) need only be loaded when the first Java component is deployed in a given address space; or a stratum 1 threading component need only be loaded if some component requires threads. This conserves resources and enables routers with limited capabilities to participate more effectively in programmable networking environments.

In general, the approach facilitates bespoke software configurations—by selecting appropriate 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).

The approach also facilitates analysing and operating on per-node software as a single composite—e.g. we can use the architecture meta-model to check consistency, integrity, security, etc; and can uniformly reconfigure and evolve the node's software base as needed (e.g. to load new functionality on demand, or unload functionality when no longer required; or juggle node resources between different activities); we can also instrument any part of the system in a uniform manner (using interceptors).

Furthermore, the approach helps organise ad-hoc interaction between layers—as all software is structured in terms of a uniform component model, any part of the system has the basic capability to talk to any other part (barring access control, and security etc. concerns) in a principled way (cf. [48])—e.g. application or transport layer components can straightforwardly obtain 'layer-violating' information from, e.g., the link layer (this is increasingly recognised as indispensable in mobile environments); furthermore, such links can be established in an ad-hoc, dynamic, manner.

Finally, reflection and CFs together promise significant benefits in terms of the management of configuration and reconfiguration. Generic meta-models can provide multiple views of component configurations and support 'principled' runtime inspection and reconfiguration along multiple alternative 'dimensions'. And where it is important to temper this power to honour domain-specific constraints, CF-specific meta-models can be used to appropriately constrain reconfiguration operations. Additionally, CFs simplify component development and assembly through design reuse and guidance to developers, encourage lightweight components (plug-ins), and increase the understandability and maintainability of systems. Most crucially, because CFs embody semantics and impose constraints relating to their area, they can play a leading role in maintaining *integrity* in the face of reconfiguration.

4. IMPLEMENTATION

4.1 Overview

Our implementation and evaluation of the NETKIT approach to programmable networking is still at an early stage. In this section, we describe our implementation work to date. §4.2 discusses work on deploying the component model, while §4.3 discusses a prototype stratum 2/3 CF.

To evaluate its claimed support for heterogeneity, we are currently working to deploy the NETKIT approach not only in standard PC-router environments, but also in Intel IXP1200 network processors-based routers [23], and in embedded, wireless and mobile devices [42]. This heterogeneity is crucial in validating the claimed generality of our approach. In all cases, the challenge is to maintain as much commonality as possible without compromising either (re)configurability or performance.

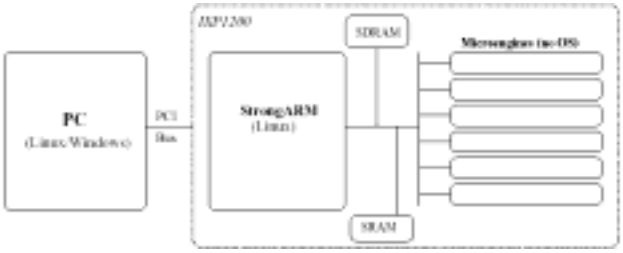

Figure 5: Schematic architecture of an IXP1200-based router.

In this paper, we focus mainly on the Intel IXP1200 implementation environment. As sketched in figure 5, the IXP1200 features an exotic hardware architecture comprising multiple processors—both a StrongARM control processor and primitive Intel-proprietary 'microengine' processors—together with various distributed/ hierarchical memory arrays.

4.2 Component Model Implementation

Our component model implementation, called *Maya*, is currently built on top of a subset of the Mozilla's XPCOM component model [45]. However, we are progressively moving away from the XPCOM dependency by applying the portability principles outlined in §3.1.2. For example, we are wrapping the stratum 1 level support provided by XPCOM into independent CFs. More importantly, we are structuring the component model run-time itself in terms of a number of CFs as follows:

- a multi-address-space capsule CF,
- a plug-in loader CF,
- a plug-in binder CF.

The multi-address-space capsule CF takes address spaces as plug-ins, resulting in a per-capsule run-time environment that comprehends multiple address spaces. For example, a capsule could encapsulate both a Linux process on the IXP1200's control processor, and one or more microengines (each microengine is associated with a single address space). Encapsulating multiple address spaces in capsules offers a powerful and general means of abstracting over tightly-coupled but heterogeneous hardware: the components within the capsule do not need to know that their execution environment differs from that of their peers, and they can uniformly operate on their peer components, and be operated on, using a common set of meta-models.

Building on multi-address-space capsules, the plug-in loader and plug-in binder CFs support (as plug-ins) multiple alternative implementations of component loading and binding respectively. In particular, these plug-ins can provide third-party loading/ binding in (intra-capsule) address spaces other than the one from which they were invoked. This builds on the transparency offered by the multi-address-space capsule concept and makes such a capsule a truly unified component support environment. For example, a component running in a Linux address space can initiate the loading of a component onto a microengine—without necessarily knowing that the component will be placed on a microengine—and then bind itself to the newly-loaded component without being aware that the latter is in any way different from itself. Happily, this transparency entails no change to the component model: it simply leverages its existing third-party loading/ binding concept.

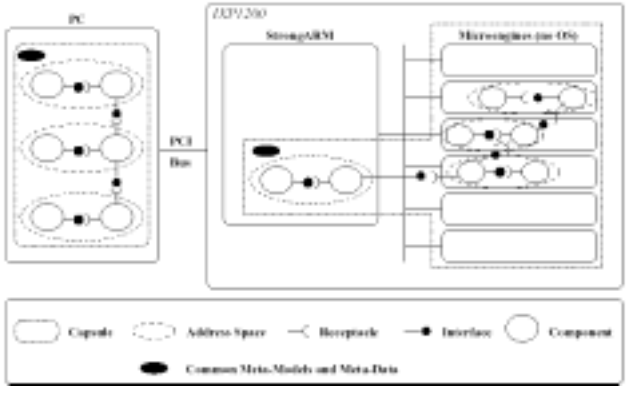

Figure 6: Multi-address-space capsules, loaders and binders

Figure 6 illustrates the multi-address-space capsule and plug-in loader/ binder concepts in the IXP1200 environment: it shows a multi-address-space capsule that encapsulates a Linux process address space and six microengine address spaces. Within this capsule are a number of components that are loaded and bound using in-capsule plug-in loaders and binders. The figure also shows a capsule in the PC environment that encapsulates three Windows address spaces, each of which contains a number of communicating components (this latter will be revisited in §4.3).

Transparency of loader/ binder selection is achieved by providing a standard set of polymorphic capsule APIs (i.e., *load(), unload(), bind()* and *unbind()*). On each call of these APIs, an appropriate plug-in is chosen on the basis of runtime configuration information. The choice of a loader, for example, might be based on attributes attached to the to-be-loaded component, such as target processor-type, target OS-type etc. Similarly, a binder might be selected on the basis of the hosting address spaces of the to-be-bound interface and receptacle. For example, to bind two components on separate microengines, a binding implementation based on shared scratch memory might be (transparently) selected. Where more control is required, and where multiple possibilities exist (e.g., where there is a choice of multiple microengines on which to load a component), transparency of plug-in selection can be foregone by means of a CF-specific meta-interface.

As well as providing a simple and consistent programming model, implementing loading and binding as plug-ins considerably simplifies the task of porting the Maya runtime to exotic architectures such as network processors. Returning to the above StrongARM/ microengines example, we simply employ a standard, generic, Linux capsule implementation. It is only the architecture-specific plug-in functionality (loaders and binders) that need to be microengine-aware. We expect the following to be a common deployment pattern: a 'primary' address space hosts the Maya runtime and 'secondary' address spaces present limited functionality to their hosted components. For example, a component hosted in a ('secondary') microengine address space will typically not have access to loaders and binders (i.e. the functionality underlying *load(), bind()* etc. will, for such components, be null). The approach also means, of course, that the dedicated fast-path packet-processing parts of the architecture are free of the performance and memory burden of the runtime. We emphasise again, though, that all this (i.e. notions of 'primary' and secondary' address spaces etc.) is entirely transparent to the Maya programmer.

As well as the default intra-capsule vtable-based bindings (we inherited these from Maya's XPCOM implementation base), we are currently developing a range of IXP1200-specific plug-in binding types. These are based on *i*) register transfers; *ii*) modifying branch instructions (cf. NetBind [7]); *iii*) shared memory mediated links involving either scratch memory or the additional static or dynamic RAM provided by the IXP1200; *iv*) paths over the various buses provided by the IXP1200.

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 [7] and Intel's MicroACE [23]) there is no reason to expect that performance should suffer. The one Maya-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 [28] we have no a-priori reason to believe that fine-grained componentisation is necessarily problematic.

4.3 Component Framework Developments

Our initial focus in the CF area has been on the design of a simple, but non-trivial, programmable networking-oriented CF that exercises many of Maya's configuration and dynamic reconfiguration features (including: multi-address-space capsules, plug-in loaders and binders, dynamic insertion of components based on the architecture meta-model; run-time type checking and interface discovery; the resources CF; and interceptors). Specifically, we have designed a stratum 2 and 3 'Router CF' which accepts, as plug-ins, Maya components that perform arbitrary user-defined packet-forwarding functions. Figure 7 illustrates one possible instantiation of the CF; however, the CF is capable of instantiating a very wide and general range of router configurations as long as these conform to a minimal set of CF-imposed rules.

In particular, the following set of rules, enforced at component load time by the CF using Maya's architecture and interface meta-models, must be adhered to by plugged-in components:

- plugged-in components must support specific packet-passing interfaces/ receptacles (called *IPacketPush* and *IPacketPull*: these respectively enable push- and pull-oriented inter-component communication [28]);
- plugged-in components may (optionally) support an *IClassifier* interface which exports an operation *register_filter()* that is used to install packet-filters; the intended semantic is that an installed packet-filter directs outgoing packets to particular outgoing *IPacketPush* or *IPacketPull* interface(s) that are named in the packet-filter specification; installing packet-filters may entail creating additional instances of these interfaces, which is possible using the standard Maya programming model;
- plugged-in components may be composite, in which case all their internal constituents must (recursively) conform to the CF's rules; additionally, composite components are expected to contain a so-called controller component that manages and configures the other internal components (see figure 7).

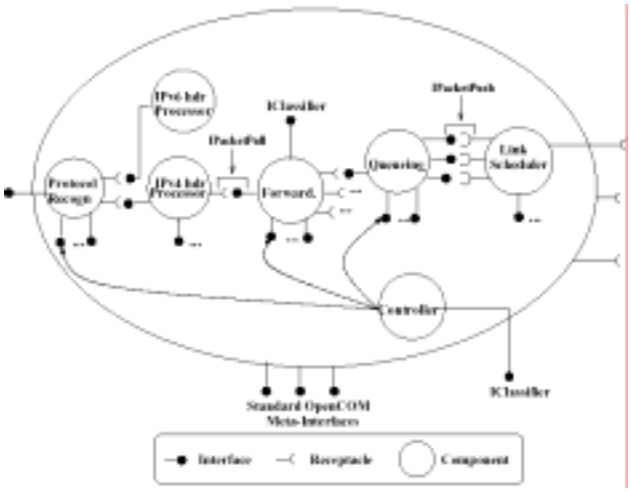

Figure 7: A composite that conforms to the Router CF

The CF also supports the definition of 'structural rules', expressed in terms of a simple XML schema, that constrain the reconfiguration of, and thus the internal topology of, composite components. Furthermore, these rules can be added or removed dynamically. Addition/ removal of rules is policed by an ACL managed by the composite's controller; the rules themselves are interpreted and enforced within an interceptor that is attached to calls of Maya's *bind()* primitive.

The Router CF also addresses safety/ security issues. To prevent untrusted plugged-in components (e.g. per-application components that act on a particular preselected packet flow) from maliciously tampering with the code/ data of other components in the same capsule, or from accidentally taking down the whole of the router capsule by crashing, we exploit Maya's support for multi-address-space capsules (see figure 6). In particular, a specialised plug-in loader is used which, if it determines that a to-be-loaded component is potentially malicious or otherwise dangerous, instantiates a new 'secondary' address space and loads the component into that (alternatively, if such an address space is already in place from a prior load, then this may be used) [43]. Such 'secondary' address spaces are barred from themselves accessing loading and binding services so that components loaded into them cannot initiate any such activities. When these untrusted components need to be bound to others in the 'primary' address space, a companion plug-in binder (having validated the legality of the binding) transparently deploys the appropriate inter-process communication mechanisms as discussed above.

Finally, the Router CF heavily exploits Maya's resources meta-model so that composites (subject to access constraints) can control the resourcing of designated tasks (e.g. packet forwarding, route lookup), especially in terms of threads, and map these flexibly to their constituent components.

The design of the Router CF is now fairly mature and we are implementing it in both PC-based and IXP1200-based routers. We hope to be able to validate its performance and flexibility in the near future. Interestingly, the IXP1200 implementation will bring to the fore the issue of component 'placement': in the PC implementation, we already, as described above, choose to place

components in different address spaces according to security/ safety considerations; in the IXP environment we additionally need to situate components (whether on the control processor or on some specific microengine) according to performance, memory availability, and load-balancing considerations. We consider that the CF itself should embody the 'intelligence' to transparently manage this placement, but with the possibility to control/ override this via a 'placement' CF built into a microengine loader.

5. FURTHER RELATED WORK

§2.2 has already discussed related work in the various programmable networking paradigms. That section also discussed stratum 2 component models for programmable networking. In this section we round off these discussions by briefly surveying related work in the area of software components in general and component based middleware in particular.

MMLite [20] is a component-based operating system built using MS COM components. It offers limited support for dynamic reconfiguration through a 'mutation' mechanism which enables the replacement of a component implementation at run-time. However it has no *framework* (e.g. in terms of reflection and CFs) to support and facilitate this replacement. Think [41] is another lightweight component model that is targeted at the construction of system software. It is close to Maya in its goals but has so far only been used in operating system implementation.

In the middleware environment, other researchers have investigated lightweight and flexible component architectures—like us, they aim to build the middleware itself in terms of components as opposed to merely supporting components on top of monolithic middleware. Prime examples are the University of Illinois' DynamicTAO [29] and LegORB [38]. These are flexible ORBs that employ a dependency management architecture that relies on a set of 'configurators' that maintain dependencies among components and provide hooks at which components can be attached or detached dynamically. Maya supports a similar capability but as an integrated part of the component model. Another example is work at Syddansk University on building real-time control middleware in terms of JavaBeans [26]. Again, none of this work has yet been applied in the programmable networking environment.

Finally, the OMG's CORBA Component Model (CCM) [36] is aimed at facilitating the deployment of distributed applications in an enterprise environment. Its central aim is to reduce the time to market for server-side code by providing a configurable server-side *container architecture* that supports generic non-functional concerns like transactions, persistence and lifecycle management. Other related solutions are Microsoft's DCOM and .NET [33], and Sun's Enterprise Java Beans. Although these technologies hold significant promise in the enterprise environment, they are not directly applicable to programmable networking environments because their container architectures carry significant overhead in terms of performance and memory footprint. In addition, some of them (i.e. EJB and .NET) operate only in a bytecode execution environment.

6. CONCLUSIONS AND FUTURE WORK

We believe that a fine-grained, reflective, language-independent component model, as discussed here, offers significant potential as the basis of an 'integrated' approach to the structuring of programmable networking software.

Apart from the potential benefits outlined in §3.4, we see our work as having potentially great applicability in the specific area of programming support for *network processors*. It is widely acknowledged that these architectures are difficult to program and that there is little or no commonality in programming environments across these machines due to their extreme architectural heterogeneity [44]. We believe that our component-based approach is a promising way of providing at least a degree of design portability across these architectures. A components-and bindings-based model seems to fit many such architectures, and the approach discussed in §4 of implementing loading and binding functionality as architecture-specific plug-ins to a generic component model runtime seems to have potential in exploiting and unifying a wide diversity of processing environments and internal communication mechanisms (the latter by means of plug-in binders). Furthermore, it is easy to see how network processor-specific hardware assists can be presented to the programmer as components. For example, a hardware checksummer can be presented as just another component to plumb in; the fact that it is implemented in hardware just means that the component implementation is effectively null (additionally, a binding to the checksummer 'component' could transparently map to whatever hardware-specific mechanism is needed to invoke the physical checksummer).

Finally, in addition to the IXP1200-related future work mentioned in §5, we are currently working with Columbia University to re-engineering their Genesis system [6]. This is a distributed service layer that supports the creation of dynamic private virtual networks, each potentially with its own semantics (addressing, routing, QoS, etc.). Apart from the opportunity to investigate the componentisation of an existing programmable networking system with a view to enhancing its deployability and (re)configurability, this is also particularly interesting to us as an exemplar of a richly-functioned stratum 4 system to complement our existing work in the other three strata.

7. REFERENCES

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

[2] Blair G.S., Coulson G., Robin P. and Papathomas, M., "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Davies N.A.J., Raymond K. & Seitz J. (Eds.), The Lake District, UK, pp. 191-206, 15-18 September 1998.

[3] Blair, G.S., Costa, F., Coulson, G., Duran, H., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., and Stefani, J.B., "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol 1616, pp115-134, 1999.

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

[5] Butler, R., Engert, D., Foster, I., Kesselman, C., Tuecke, S., Volmer, J., Welch V., “A National-Scale Authentication Infrastructure”, IEEE Computer, Vol 33, No 12, pp 60-66, 2000.

[6] Campbell, A.T., Kounavis, M.E., Villela, D.A., Vicente, J.B., de Meer, H.G., Miki, K., Kalaichelvan, K.S., “Spawning networks”, IEEE Network Magazine, Vol 13, No 4, pp. 16-29, July/Aug 1999.

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

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

[9] Coulson, G., Moonian, O., “A Quality of Service Configurable Concurrency Framework for Object Based Middleware”, Concurrency and Computation: Practice and Experience (to appear), 2002.

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

[11] Schmid, S.,Chart, T., Sifalakis, M, Scott, A.C., “Flexible, Dynamic and Scalabale Service Composition for Active Routers”, Proc. IWAN 2002, Zurich, Dec. 2002.

[12] Clark, M., Blair, G.S., Coulson, G., Parlavantzas, N., “An Efficient Component Model for the Construction of Adaptive Middleware”, Proc. IFIP/ACM Middleware 2001, Heidelberg, Nov 2001.

[13] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins: A Software Architecture for Next Generation Routers”, Proc. ACM SIGCOMM 98, 1998.

[14] Engler, D.R., Kaashoek, M.F., O’Toole, J., “Exokernel: An Operating System Architecture for Application-Level Resource Management”. Proc. 15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO, USA, pp 251-266, Dec 1995.

[15] Fry, M., Ghosh, A., “Application Level Active Networking”, Proc. 4th Intl. Workshop on High Performance Protocol Architectures (HIPPARCH '98), June 98.

[16] Merugu, S., et al, “Bowman and CANEs: Implementation of an Active Network”, Proc. 37th Conference on Communication, Control and Computing, Monticello, Illinois, September 1999.

[17] Yemini, Y., da Silva, S., “Towards Programmable Networks”, Proc. IFIP/IEEE International Workshop on Distributed Systems: Operations and Management”, Italy, October 1996.

[18] Hicks, M.W., Moore, J.T, Alexander, D.S., Gunter, C.A., Nettles, S., “PLANet: an Active Internetwork”, Proc. IEEE INFOCOM (3), pp 1124-1133, 1999.

[19] Schwartz, B., et al, “Smart Packets for Active Networks”, Proc. OPENARCH 1999, March 1999.

[20] Helander, J., Forin, A., “MMLite: A Highly Componentized System Architecture”. Proc. 8th ACM SIGOPS European Workshop, pp 96-103, Sintra, Portugal, September 1998.

[21] Hjalmtysson, G. “The Pronto Platform - A Flexible Toolkit for Programming Networks Using a Commodity Operating System”, Proc. IEEE Conf. on Open Architectures and Network Programming, OPENARCH 2000, Tel-Aviv, Israel, March 2000.

[22] IEEE P1520 Proposed IEEE Standard for APIs for Networks, http://www.ieee-pin.org/.

[23] Intel IXP1200; http://www.intel.com/IXA.

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

[25] Jones, N.D., “An Introduction to Partial Evaluation”, ACM Computing Surveys, 28(3), pp. 480-504, Sept 1996.

[26] Joergensen, B.N., Truyen, E., Matthijs, F., and Joosen, W., “Customization of Object Request Brokers by Application Specific Policies”. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.

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

[28] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F., “The Click Modular Router”, Proc. ACM SOSP 1999, pp 217-231, Dec 1999.

[29] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB”. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.

[30] Liedtke, J., “On µ-Kernel Construction”, Proc. 15th ACM Symposium on Operating System Principles (Copper Mountain Resort, CO., Dec. 3-6). ACM Press, New York, NY, pp. 237-250,1995.

[31] Maes, P., “Concepts and Experiments in Computational Reflection”, Proc. OOPSLA'87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.

[32] Mozilla Organization, XPCOM project, 2001, http://www.mozilla.org/projects/xpcom.

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

[34] NodeOS Interface Specification, AN Node OS Working Group, http://www.cs.princeton.edu/nsg/papers/nodeos.ps, Jan 2001.

[35] Peterson, L., Gottlieb, Y., Hilber, M., Tullmann, P., Lepreau, J., Schwab, S., Dandekar, H., Purtell, A., Hartman, J., "An OS Interface for Active Routers", IEEE Journal on Selected Areas in Communications, special issue on Active Networks, March 2001.

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

[37] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E., "Knit: Component Composition for Systems Software", Proc. OSDI 2000, pp 347-360, Oct 2000.

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

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

[40] Villazón, A., "A Reflective Active Network Node", Proc. 2nd Intl. Working Conf. on Active Networks (IWAN 2000), Tokyo, Japan, Oct 2000.

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

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

[43] Schmid, S., "A Component-based Active Router Architecture", Lancaster University PhD Thesis, http://www.mobileipv6.net/~sschmid/PhD_Thesis.ps, 2002.

[44] Comer, D., Peterson, L., "Network Systems Design Using Network Processors", ISBN 0131417924, Prentice-Hall, 2003.

[45] Mozilla Organization, XPCOM project, 2001, http://www.mozilla.org/projects/xpcom.

[46] Bos, H., Samwel, B., "The OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking", Proc. IWAN 2002, Zurich, Dec 2002.

[47] Solarski, M., Bossardt, M., Becker, T., "Component-based Deployment and Management of Services in Active Networks", Proc. IWAN 2002, Zurich, Dec 2002.

[48] Braden, R., Faber, T., Handley, M., "From Protocol Stack to Protocol Heap–Role-Based Architecture", ACM SIGCOMM Computer Communication Review, Vol 33 No 1, January 2003.