Service Deployment on High Performance Active Network Nodes

Matthias Bossardt[*]   Rolf Stadler[†]

TIK Technical Report 122
September, 2001

Abstract

In order to realize service deployment on high-performance active nodes, the problem of installing and configuring software components in complex, heterogeneous node environments must be addressed. The paper presents our approach to this problem, called Chameleon, which includes two main aspects. First, the service model we propose is based on service components with two interfaces–a data flow interface for programming packet flows and a control interface for programming the control plane. Second, the service specification is kept independent of any particular node architecture. During the service deployment phase, the service specification is resolved on each node offering the service and is driven by node-specific parameters. The result of this resolution is a tree of service components, which can differ among different types of nodes. Our solution allows a service to take full advantage of specific node features, such as those related to performance or security. The design is illustrated using an active adaptable reliable multicast service.

Keywords – Active network, service model, service deployment

1   Introduction

Service deployment on active network nodes includes installing and configuring software components that perform processing in the data path. Service deployment is difficult on high-performance nodes due to their complex architectures. They are often based on a multiprocessors, run service components as kernel modules and execute multiple concurrent execution environments.

Many types of execution environments have been developed. Some of them are rich in functionality, others are more restricted, but offer better security or better performance for certain tasks. As a consequence, active nodes typically provide more than one type of execution environment. Also, different types of active nodes usually support different sets of execution environments. All this motivates us to develop a service deployment scheme that can cope with heterogeneous active nodes, i.e., with active nodes running sets of execution environments that can vary from node to node. Our goal is to develop a scheme, where the service specification is node-independent, but service deployment and installation recognize the diversity of execution environments, thereby allowing a service to exploit the particular functionality and performance features of active network nodes.

Multiple approaches to active networking have been proposed over time. Early work in this field was based on the so-called capsule model [3], which implies *inband service*

---

[*]Swiss Federal Institute of Technology (ETH), Computer Engineering and Networks Laboratory (TIK), Zurich, Switzerland, bossardt@tik.ee.ethz.ch

[†]Columbia University, Center for Telecommunications Research (CTR), New York, USA, stadler@ctr.columbia.edu

deployment. Capsules are packets that contain both program code *and* data. When arriving on an active node, the capsule code is executed in an execution environment. The capsule model has proved to be useful, if the service complexity is low, bandwidth and code installation overhead are not critical, or the amount of state on a node must be kept small. To exploit the fact that, in most active services, the packets require the execution of the same code on almost every node, the plug-in model [1] has been proposed. In this model, active packets do not carry code, but only a reference to so-called router plug-ins. When arriving on a node, a packet is processed by the router plug-in it is pointing to. If the referred plug-in is not available on the node, it is downloaded from a code server and installed. This mechanism implies *out-of-band* service deployment. It is generally believed that out-of-band service provisioning will be used for future active telecom environments. For this reason, our work is based on the plug-in concept.

Our approach to service deployment in heterogeneous active networks, called *Chameleon*, has two important aspects. First, the service model we propose is based on components with two types of interfaces–a data flow interface for programming packet flows and a control interface for programming the control plane. A service is structured as an arbitrary tree of such components. Second, the service specification is independent of any particular node architecture. During the service deployment phase, the service specification is resolved on each node offering the service. The resolution and service creation process depends on the locally available set of execution environments. This way, the service can take advantage of the specific node features.

This paper is organized as follows. Section 2 outlines the Chameleon service model. It is illustrated using the example of an adaptive reliable multicast service. Section 3 presents our approach to local service resolution and creation. It is shown how the adaptive reliable multicast service is deployed in two different active node environments. Section 4 summarizes related work. In section 5 we draw our conclusions and discuss future work.

This work is part of the authors' contribution to the IST-FAIN project (5th EC Framework Program) [6], which aims at developing an architecture for future high performance active networks.

2   The Chameleon Service Model for Active Network Nodes

With Chameleon, we propose a new service model that alleviates the development of services, and at the same time takes advantage of the flexibility and versatility offered by active networks. This section introduces the service model. We motivate our approach by presenting the requirements. Further a sample service is described to illustrate the use of our service model.

2.1   Requirements

Service models have been developed for various network and system architectures. Most models target telecom environment [8], some of them end systems [2]. None of them is suitable for active networks, which impose new requirements on a service model.

2.1.1   Separating the Data Flow Interface from the Control Interface

The task of a router in traditional networks is to forward packets to the destination. An incoming packet traverses several processing stages before it leaves the router. An active node is the equivalent of a router in active networks. Packet forwarding remains the main task of an active node. However, the distinctive feature of an active node is that packet processing in the node is freely programmable. An active node allows to program both the interconnection and configuration of processing components (control plane), as well as the processing in the components itself (data plane). Because of the flow based nature

of packet processing in active nodes, we introduce an explicit data flow interface. Several benefits originate from a service model that supports such an interface.

1. An explicit data flow interface supports representation of an active service as a directed graph, where the vertices correspond to packet processing components and where the edges represent packet flows. Data flow diagrams are a intuitive way to model a network service. For example, a simplified IP forwarding service can be modeled by the following sequence of components: *CheckIPHeader*, *LookupIPForwardingTable*, *DecrementTTL*, *CalcChecksum*, and *OutputQueue*.

2. A data flow interface, which is separated from a control interface, facilitates the mapping of the service specification into an efficient implementation. In traditional networks, routers are highly specialised machines for packet forwarding. As a consequence, routers distinguish between control plane and data plane. The data plane works at link speed and forwards packets efficiently, whereas the control plane may implement more complex functionality (e.g. route discovery protocols), while running at a much slower time scale. Being the analogous entity to routers in active networks, active nodes must be optimized for packet forwarding as well. Therefore the distinction between control and data plane remains valid for active nodes. The existence of a data flow interface allows to separately program the two planes and simplifies the mapping of services onto execution environments that offer high performance data plane features. An example for such a high performance data plane feature is described in [10]. It proposes an operating system facility that allows to forward data among different service components without copying it from one memory location to another. The data flow interface can be directly mapped onto such a facility.

3. An explicit data flow enables easy integration of "hardwired" service components. Active nodes may provide certain service components that are implemented in a nonprogrammable way. It is very simple to abstract such a component with our service model and to integrate it into the data flow of a more complex service. An example is shown in section 2.3, where standard IP multicast is extended to provide a more complex adaptive reliable multicast service.

The control interface must provide a generic way to exchange control and monitoring information among processing components. The requirements for such an interface depend on the specific execution environment. In certain execution environments, the control interface may provide read/write access to state information of service components only. For example, a *scheduler* can read the current length of a *queue*. On the other hand, an execution environment that provides CORBA methods allows a much more complex control interface implementation.

2.1.2 Support of Different Active Node Environments

A high-performance active node may provide more than one execution environment. This is motivated by the facts that 1) each execution environment is usually optimised for a certain type of tasks, and 2) certain processing component are implemented for one specific execution environment. Since it is not realistic to assume that all active nodes provide the same set of execution environments, we require a service model that enables service specification that is independent of the node environment.

2.1.3 Support of Adaptive Services

Adaptive services allow to adjust the service to changes in network topology and traffic patterns. Adaptive functionality includes replacing and reconfiguring control and data path processing components, which is facilitated by active network technology. With such

mechanisms available, it is possible to create services that, at runtime, replace service components. For example, consider a routing service for ad hoc networks that replaces the routing algorithm based on the current mobility pattern of the active node. In addition to that, the services can be automatically deployed on demand. Examples for this type of adaptive services include packet caches, that are only installed when and where loss occurs. Furthermore, it is possible to dynamically update the version of a service component. We conclude that our service model must support adaptive services in order to completely exploit the potential benefits of active networks.

2.2   Node Independent Service Model

In order to fulfill the requirements from section 2.1, we need abstractions to model a service in a node independent way. A traditional component model such as the Corba Component Model [14] or JavaBeans [13] is not adequate for our purpose, for the following reasons:

- The separation of control and data plane, which is essential for a high performance active node, is not supported by a traditional component model. In such a model, components communicate through a sophisticated infrastructure (e.g. CORBA Object Request Broker). Because of its generality it lacks the performance that is required on the data plane.

- Execution environments, such as kernel or reconfigurable hardware execution environments, do not support traditional component models. Of course, service components running in this type of environment could be represented by proxies running in an execution environment that supports a traditional component model. Our service model avoids such proxies and the resulting communication overhead.

- Some processing components need only exchange very little information among each other, e.g. the current queue length for a queue component. There is no need to support remote method invocation or even more complex middleware services. Enforcing the use of a CORBA-like interface would be technically too expensive.

This paper advocates a new service model that is specialised for active networking. The contributions of our service model include the following.

- Abstractions that enable node independent service specification.

- A container for service logic that is optimised for active networking.

- An interface design that allows the separation of control and data plane.

- An interface design that enables an optimised mapping of the service description onto the node environment.

Three building blocks are sufficient to create a service model for active networks, namely *container, tube,* and *funnel*. The service logic is packaged into containers. Containers communicate with each other through tubes and funnels. Tubes represent highly specialised facilities for data plane communication, whereas funnels transport control information among containers. Figure 1 shows the relationship among the components of the service model, which are described in detail below.

2.2.1   Container

A container is an abstract class that generalises the *code* and *service specification* classes. The container declares interfaces that are common to both subclasses. As a result *code* and *service specifications* can be handled and accessed in a uniform manner. Figure 1 shows this relationship among the entities of the service model.
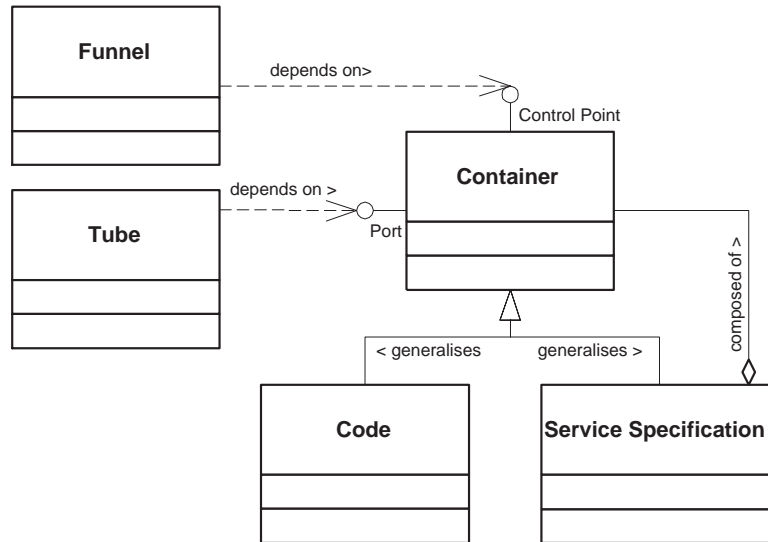
Figure 1: Chameleon Service Model

A *service specification* is a composition of containers, which represent either service specifications or code. As a result, services are structured as trees. Note that the Chameleon service model gives additional flexibility to a service designer, compared to service models that follow a layered approach [7]. Using the Chameleon service model, a service developer can extend a service defined by tree A with additional components, which results in a new service defined by a tree that is an extension of A.

Code objects are the leaves of the service tree. Examples of code include binaries for the kernel execution environment, a Java class for a Java-based execution environment, or a synthesised VHDL for an execution environment based on reconfigurable hardware [5].

Container feature two different types of interfaces.

Ports represent the data flow interface. Ports are either input or output ports. An output port is connected to a tube, which in turn connects to exactly one input port. A container has zero or more input respectively output ports.

Control points represent the control interface. There are two approaches to realizing a generic control interface. First, all containers use a uniform interface to communicate with each other [14].

Second, service components implement the control points in a way that is optimised for a specific execution environment. Adapters are provided on demand in order to enable control communication among different execution environments. Control point implementation A can be adapted to implementation B, if the functionality of A is a subset of the functionality of B. Figure 2 shows an example of such an interface implementation hierarchy.

Our service model uses the second approach, which allows for very efficient control plane implementations at the cost of increased service deployment complexity. Consider two containers providing service A rsp. B, both running in the same execution environment X. X supports read/write access to data structures as control point implementations. Control communications between containers A and B can be done very efficiently within X. On the other hand, consider a second execution environment Y that supports CORBA methods as control point implementation. Assume

that container A runs in X, whereas B runs in Y. In this case, an adapter is provided by the node that maps the control point functionality of execution environment X to the functionality of Y.
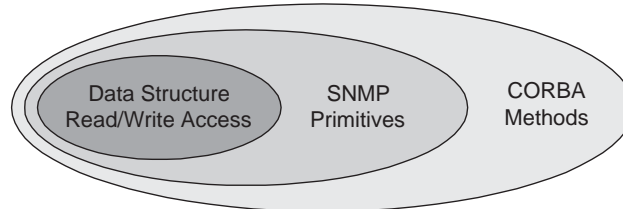

Figure 2: Example of Control Point Implementation Hierarchy

2.2.2   Tube

A tube enables the data flow cross execution environment boundaries. Tubes transport packets and are unidirectional. Tubes are similar to UNIX pipes, which allow data to flow from one process to another. However, with tubes the nature of the end points depend on the execution environment. It may be a process in the case of a Java execution environment, or a I/O unit of a reconfigurable hardware execution environment. Furthermore, the end points are located in different execution environments.

2.2.3   Funnel

A funnel provides an infrastructure for the exchange of control information across execution environment boundaries. Funnels are for the control information, what tubes are for the data flow. In addition to that, funnels adapt a simple control point implementation to a more complex one.

2.3   Modeling an Adaptive Reliable Multicast Service

We model an adaptive reliable multicast service using the Chameleon service model. The goal of this section is to illustrate the applicability of our service model to describe a adaptive service. We use a graphical representation for the service model. In our current implementation, we use an XML-based specification language.

In [9], the authors present an Active Reliable Multicast (ARM) service. ARM is based on standard IP multicast, as described in [11], [12]. In order to achieve reliability, receivers generate NACK messages to signal packet. NACK message is sent to the source, which in turn resends the lost packet. ARM extends standard IP multicast with the following functionality.

- Routers cache multicast data packets for possible retransmission. On receiving a NACK message, the router checks whether the requested packet is in its cache. If this is the case, the requested packet is retransmitted.

- NACK implosion [9] is avoided by dropping duplicate NACK messages. Only one NACK per multicast subtree is forwarded to the source.

- Using partial multicast, routers retransmit packets only to those receivers that requested them.

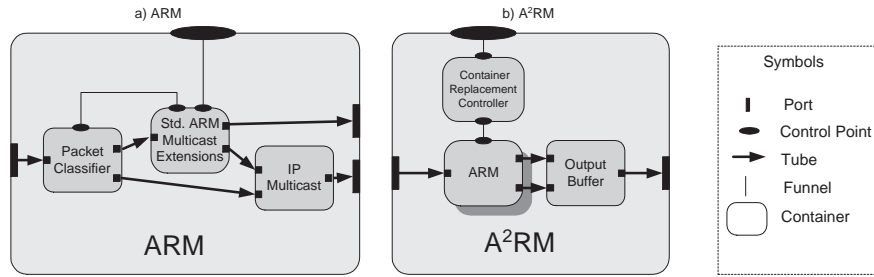Figure 3a) shows an example of an ARM service model.

Figure 3: Specifications of Reliable Multicast Services

2.3.1 Making ARM Adaptive

In [9], the authors claim that ARM needs only to run in a few strategic locations in the network in order to perform well. Since these strategic locations are difficult to determine and probably change frequently, ARM is a premium candidate to be extended with adaptive behaviour. The result is an *Adaptive* Active Reliable Multicast (A^2RM). In A^2RM router decide based on local measurements whether to perform the ARM multicast extensions (data caching, NACK fusion/suppression, and partial multicast). Making the service adaptive consists in adding container that controls the *ARM* container (see figure 3b)). This additional container reads the number of NACKs received and compares it against two thresholds. If the *standard ARM multicast extensions* are running and the number of NACKs falls below the low threshold, the former are replaced with a container that simply counts and forwards NACKs. If the number of NACKs, on the other hand, increases above the high threshold the container containing the NACK counter is replaced by the ARM multicast extensions. As a result, router resources are only used to provide ARM when and where required. We notice in figure 3b) that the *ARM* container is shown with a shadow indicating that it may be replaced at runtime.

3 Service Deployment on Active Network Nodes

Service deployment in a network takes place on two levels: 1) on the network level, where network nodes are identified that will run the service, and 2) on the node level, where software will be deployed within the node environment. In this paper, we investigate specifically local service deployment on active nodes. Service deployment on the network level is left as future work.

In traditional networks, a service is specified and deployed by a centralised entity, e.g. the network manager. In active networks, however, users, if permitted, or services themselves can create service specifications and deploy new services. We will refer to the entity that specifies the service as *service specifier*.

We consider a scenario where the service specification is sent to a particular network node. Local service specifications can be transported to a network node using a file transfer protocol, using mobile agents, or encapsulating the specification in active packets.

The remainder of this section describes our approach to a the local service creation architecture, and the mechanism that maps a service specification to the local node environment.

3.1 The Local Service Creation Engine

Once a service specification has been received by a node, the local service creation engine maps the specification onto the local node environment (see figure 4). In our implemen-

tation, we use an XML document and a corresponding schema to specify a service. The local service creation engine consists of a node independent and a node specific part. The *node independent* part parses the service specification and resolves the hierarchy. This part of the local service creation engine is portable.

There is a small number of *node specific* components within the local service creation engine. These components provide information about the execution environments available on the node. Furthermore, one of these components, named configurator, installs and configures the container instances in the execution environments.

In the following, we describes the components of the local service creation engine in detail.

Specification resolver is the heart of the local service creation engine. It controls the mapping process. The input of the specification resolver is the service description. When the mapping process terminates, the *configurator* is called in order to install and configure the mapped service.

Specification parser receives a service specification from the *specification resolver*. The service specification is read and checked on syntactical correctness. It's output is a one level tree representation of the service specification.

Container discovery is used to search the network for containers. The search key identifies the service that is provided by the container. Containers may be found in *container repositories* that reside on specific container servers or on active network nodes (including the requesting node itself). This component returns a list of references to containers.

Configurator dynamically installs code in the execution environments. Furthermore, it sets up tubes and funnels to enable communications among service components that reside in different execution environments. This component is node specific.

Mapping strategy contains rules that allow the *specification resolver* to decide which container instance is the most appropriate for the node environment. E.g. a rule might specify that containers for the kernel execution environment are always preferred to Java containers.

Node information base contains information about the configuration of the active network node. The information includes the type(s) of available execution environments, as well as the type(s) of funnels and tube(s) available to allow communication among different execution environments. The node information base is part of the local management information base (MIB).

3.2   Mapping the Service Specification onto the Node Environment

The mapping of the service specification onto the node environment involves two phases. In the first phase, a service tree is built, the leaves of which contain code objects. In the second phase, the service tree is traversed from the leaves towards the root. The code is fetched from code repositories and installed in the execution environments. Then the containers are interconnected by tubes and funnels.

3.2.1   Building the Service Tree

1. The *specification resolver* receives a service specification from the service specifier.

2. The *specification resolver* calls the *specification parser*, which constructs a tree that reflects the information contained in the service specification. The leaves of the tree are the containers the service is composed of.
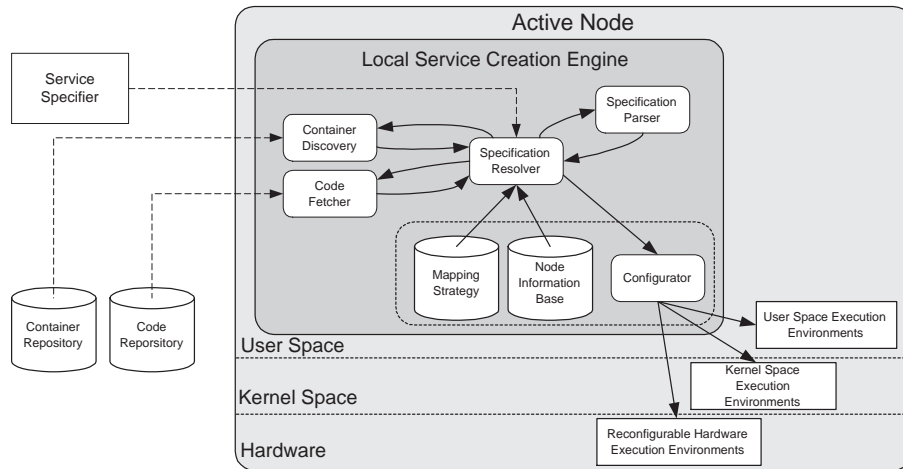
Figure 4: Local Service Creation Engine

3. The *container discovery* searches the network for instances of the leaf containers. As a result a list of container instances is handed to the *specification resolver*. If a leaf container points to code, the list contains information about the location(s) where the container code can be retrieved from. In the second case, in which a leaf container points to a service specification, the resulting list may contain one or more instances of service specifications.

4. For each leaf container the *specification resolver* decides, which instance to insert in the tree. The decision process is based on information from the *mapping strategy* and the *node information base*. If a leaf is mapped to code or to a "hardwired" service, the leaf must not be considered anymore in the remaining steps of building the service tree.

5. If all leaves are mapped to code or to "hardwired" services the tree building process terminates. Otherwise steps 2–4 are repeated for the remaining leaves.

3.2.2 Installing the Mapped Service

1. The service tree is traversed from the leaves towards the root.

2. For each visited leaf, the *code fetcher* retrieves the code from a *code repository*.

3. The *configurator* installs the code into the execution environment.

4. For each visited tree node, the *configurator* sets up the connections among the children tree nodes as described in the container that is represented by the currently visited node.

5. The process terminates after the root has been visited.

3.3 Mapping an Adaptive Reliable Multicast Service

We use the service specification from section 2.3 to illustrate the functionality of the local service creation engine. The service specification is applied to two different active node environments. We show that in both cases the service specification is deployed exploiting the specific features of the active nodes. The exact mapping depends on the active node environment. As an example, we give the mapping for two different types of active nodes and describe the process in detail for one node.

3.3.1   Node Environments

Active node 1 is a software router, which we currently run at ETH (see figure 6a)). It is based on a modified version of Linux that provides a kernel execution environment on the Intel IXP1200 network processor. In the user space we run a Java execution environment. Tubes between the execution environments are implemented as domain sockets. Funnels make use of the `/proc` file system to exchange control information among containers across the execution environment boundaries.

Active node 2 is based on a conventional IP router that is connected to a legacy PC running a Java execution environment (see figure 6b)). The IP router provides hardwired IP multicast and packet filter services. The control plane functions of the router can be accessed form a Java execution environment on a PC through an SNMP interface. The data plane can be accessed by setting filter parameters of the routers, process these packets on the PC and send the resulting packets back to the router.
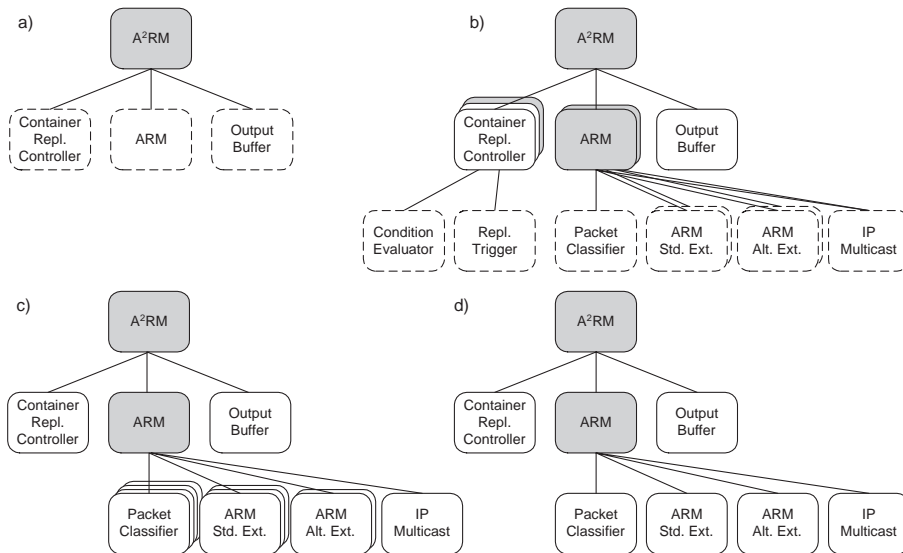

Figure 5: Four Steps of Resolving an A^{2}RM Service Specification on a Node

3.3.2   Mapping A^{2}RM onto the Node Environments

The Active node receives the A^{2}RM service specification and generates the initial service tree with the A^{2}RM container at its root as shown in figure 5a). The grey box indicates that A^{2}RM is composed of several services. In fact, the A^{2}RM service specification contains references to three containers, which are represented with a dashed frame in figure 5a).

The *container discovery* searches the network for appropriate container instances. Figure 5b) shows that three instances of the *container replacement controller*, two instances of *ARM* and one instance of the *output buffer* were found. Two of the *container replacement controller* are references to executable code (white boxes), whereas the third is composed of two other containers. We notice also that one instance of *ARM* is composed of three containers, whereas the other is composed of four. Both instances point to a *IP multicast* container.

Since a suitable "hardwired" output buffer instance is available at the node, no other instance has been searched for. Figure 5c) shows that of the two available version of *ARM*, the *specification resolver* decided to take the one that is composed of four containers. The *container discovery* searches the network again. It finds a couple of instances for each of

these containers. We notice in figure 5d) that the *specification resolver* has chosen the appropriate instances. Since all container instances point to code (white boxes), the mapping process terminates. The *code fetcher* retrieves the code from the *code repositories*. The *configurator* subsequently installs and configures the code in the execution environments and sets up the required tubes and funnels. Figure 6a) shows the resulting mapping for *active node 1*. In figure 6b), we see the resulting mapping of the same service onto the environment of *active node 2*.
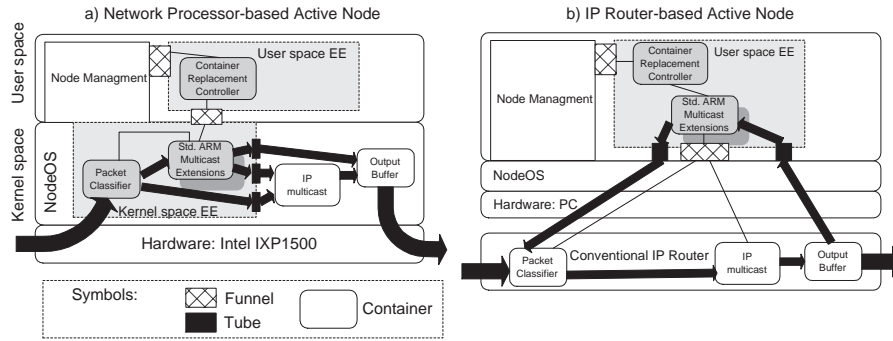

Figure 6: Adaptive Multicast Services Deployed on Two Different Node Environments

4   Related Work

Click Router [4] proposes a software architecture for configurable routers. Click Router can interpret service specifications an create and configure services. Services are composed of C++ elements that run in a kernel execution environment. Click Router does not support multiple execution environments and assumes homogeneous node architectures.

The *IEEE P1520 standards initiative* [7] proposes a different approach to service composition than outlined in this paper. The P1520 interfaces are strictly layered and named V-(value added), U-(user), L-(lower) and CCM-(connection control and management) interface. The IP sub working group produced draft documents about the architecture of the L-interface. This interface provides a representation of physical elements of the node as Virtual Network Devices. Virtual Network Devices are composed of lower level building blocks. The initiative proposes three levels of abstraction: base building blocks, resource building blocks, and service-specific building blocks. Building blocks of a specific level are composed of building blocks of the level below only. P1520 restricts the hierarchy of service components to three levels with defined functionality. The Chameleon service model allows for an arbitrary hierarchy without layering, which is required to support the heterogeneous node environments. Base building blocks interact with physical router elements through the CCM-interface. This interface allows only to control and manage physical elements. P1520 aims at programming the control plane. However, it does not allow to program the data plane, which is a distinctive feature of our service model for active networks. P1520 does not support adaptive services, because building blocks are not dynamically replaceable. Furthermore, P1520 does not specify service deployment mechanisms.

TINA-C [8] specifies a network architecture for the telecommunication environment. Network services run on a Distributed Processing Environment (DPE), which is often implemented as a CORBA Object Request Broker (ORB). TINA-C computational objects provide an operational interface, which connects to the ORB and allows to control and manage a component. Furthermore, TINA-C include a stream interface that allows to separate the data from the control plane. As a consequence of the operational interface, TINA-C computational objects require a CORBA-like DPE infrastructure.

5 Discussion and Future Work

In this paper, we described a service model that enables services components to program the data *and* control plane of an active node. Furthermore, it is possible to specify adaptive services, which dynamically replace service components. The service model facilitates service deployment on multi-execution environment nodes. The service specification is locally mapped to the node environment in order to exploit the specific features of the different execution environments. Moreover the service specification supports heterogeneous networks because it is node independent. As a consequence the same service specification can be sent to different types of active nodes. The local service creation engine, which maps the service specification to the node environment, requires only a small part of node-dependent code. A node information base and local rules for the mapping strategy allows to exploit the specific features of the execution environments provided by a node.

Future work will investigate the service deployment on a network level and its integration with the work proposed in this paper. Also, we will extend and generalize the XML schema that we use in our prototype implementation for the specification of services. Furthermore, we plan to realize and evaluate Chameleon within the testbed of the IST-FAIN project (5th EC Framework Program) [6], which aims at developing an architecture for future high performance active networks.

6 Acknowledgments

References

[1] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B. Router Plugins - A Software Architecture for Next Generation Routers. IEEE/ACM Transactions on Networking, February 2000.

[2] B. Stiller, B., Class, C., Waldvogel, M., Caronni, G., Bauer, D. A Flexible Middleware for Multimedia Communication: Design, Implementation and Experience, IEEE Journal on Selected Areas in Communication, vol. 17 no. 9, September 1999.

[3] Tennenhouse, D.L. and Wetherall, D.J. Towards an Active Network Architecture. Computer Communication Review, Vol. 26, No. 2, April 1996.

[4] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M.F. The Click Modular Router. ACM Transactions on Computer Systems 18(3), August 2000, pages 263-297.

[5] Lockwood, J.W., Naufel, N., Turner, J.S., and Taylor, D.E. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001), Monterey, USA, February 2001.

[6] Galis, A., Plattner, B., Moeller, E., Laarhuis, J., Denazis, S., Guo, H., Klein, C., Serrat, J., Karetsos, G., Todd, C. A Flexible IP Active Networks Architecture. International Working Conference on Active Networks (IWAN 2000), Tokyo, Japan, October 2000.

[7] Lin, P., et al. Programming Interfaces for IP Networks, A White Paper. IP Sub Working Group IEEE P1520, www.ieee-pin.org, June 1999.

[8] TINA-C. www.tinac.com.

[9] Lehman, L.H., Garland, S.J. and Tennenhouse, D.L. Active Reliable Multicast. IEEE Infocom '98, San Francisco, April 1998.

[10] Druschel, P., Peterson, L.L., Fbufs: A High-Bandwidth Cross-Domain Transfer Facility, Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, Dec. 1993.

[11] Deering, S.E. Host extensions for IP multicasting. IETF RFC 1112, August 1989.

[12] Fenner, W. Internet Group Management Protocol, Version 2. IETF RFC 2236, November 1997.

[13] Hamilton, G.(editor). JavaBeans, V1.01. Sun Microsystems, Mountain View, USA, 1997. http://java.sun.com/products/javabeans/docs/beans.101.pdf.

[14] CORBA Components final submission, http://www.omg.org/cgi-bin/doc?orbos/99-02-05