Chameleon: Realizing Automatic Service Composition for Extensible Active Routers

Matthias Bossardt, Roman Hoog Antink, Andreas Moser, and Bernhard Plattner

Computer Engineering and Networks Laboratory *
Swiss Federal Institute of Technology, ETH
Zürich, Switzerland
bossardt@tik.ee.ethz.ch

Abstract. Complex network services can be constructed by composing simpler service components in a well defined way. To benefit most from such an approach, service components should be reusable for different services. Furthermore the composition must be performed automatically and customized to the service execution platform.
In this paper, we focus on node local aspects of service composition. We contribute design and implementation details of Chameleon, a system targeted at automatic service composition. Our system is based on (1) service descriptors containing meta-information about service components and (2) a service creation engine composing and installing services in a platform specific and automatic way. Target platforms are modeled as active nodes featuring Execution Environments (EEs) to serve as runtime environments for service components.
To validate our concepts, we implemented an active node. It features two different EEs, an EE based on Click router technology, which is suitable for forwarding plane services, as well as a general purpose Java-based EE. A demonstration service, which performs traffic shaping, is briefly presented to illustrate the concepts and their applicability.

1 Introduction

Active nodes promise to be a very flexible platform for dynamic deployment of network services. A wide variety of active network nodes (ANN) have been developed so far. In the beginning, active packet-based approaches were predominant. Active packets contain code or a reference to code, which is executed in Execution Environments (EEs) of active nodes at packet arrival. More recently research has focused on component-based approaches. In contrast to active packets, component-based service models are based on code modules, which are usually deployed before arrival of packets to be mangled using a specific service deployment infrastructure. A detailed discussion of both approaches and their properties can be found in [7].

In the component-based approach, active nodes provide one or more EEs to run service components (SCs). Distinctive features of EEs are the abstractions (API) they

* This work is partly funded by ETH Zürich and Swiss Bundesamt für Bildung und Wissenschaft (BBW) under grant number 99.0533. A subset of it is part of ETH's contribution and work as a partner in the European project IST-FAIN (IST-1999-10561).

N. Wakamiya et al. (Eds.): IWAN 2003, LNCS 2982, pp. 163–177, 2004.

provide to service components and the technology they are implemented in. We expect active nodes to feature two or more EEs. First, multiple EEs allow to separate transport, control and management plane of active nodes using different technologies and providing adequate APIs.[1] Second, it is possible to select SCs implemented in different technologies and optimize service composition.

In this paper we focus on service composition for a component-based service model. We provide a model that is generic enough to be implemented using a variety of EE technologies. Consequently, components in the fast path of a router (e.g. a forwarding table lookup) can be optimized for high performance, whereas control components (e.g. a routing algorithm) might be run in an EE optimized for control processes. Hence, the most appropriate technology can be selected for each service component, as long as the corresponding EE is available on the active node.

Components-based software and their composition have been studied in the context of distributed computing [2,1] or telecommunication networks [4]. They are based on middleware providing SCs with a homogeneous environment. Thus, a service can be implemented in the same way on different nodes. This approach results, however, in additional communication overhead among SCs at runtime, which is not acceptable for active networks where SCs must mangle packets at line speed.

In [8] we introduced Chameleon, a node-level service deployment architecture to cope with heterogeneity in active networks caused by active nodes featuring different sets of EEs. At the core of our architecture is the Service Creation Engine (SCE), which composes services by performing a node specific mapping of service descriptions to appropriate code modules. Furthermore, the SCE binds the code modules and configures both EEs and node operating system (nodeOS). As a result, we are able to describe services in the same way for all nodes, without introducing communication overhead at runtime. This comes at the cost of additional service deployment complexity, which is handled by the SCE.

Although this paper describes refinements of the Chameleon architecture, we focus on service composition facets of our system. We contribute the details of our approach to service description and composition. Moreover, we describe design and implementation of our prototype. A major requirement in the design and implementation of the SCE was to ensure that the same SCE can be used for many different active node designs/implementations. This paper presents how this is achieved by our design.

For our prototype implementation, we built two execution environments. One is based on Click router [10] and well suited to run transport plane service components. The second EE is based on a Java Virtual Machine (JVM) and useful for control plane service components. We implemented a traffic shaper service that uses both EEs and describe its deployment.

This paper is organized as follows. Section 2 describes our approach to service composition and the models involved. It further explains a formal language to describe the elements of our models. Section 3 discusses the design details of the service composition engine, while section 4 describes our prototype implementation. A sample service to

[1] The proper separation of forwarding (aka transport), control and management plane is a current concern of several major companies. More specifically, the IETF ForCES work group [9] intends to come up with a standardized protocol between transport and control components.

validate the implementation is briefly presented. Finally, we draw our conclusions in section 5.

2 Service Composition

Service composition allows building complex services from simpler *service components*. In that sense, service components take on the role as building blocks, similar to LEGO™ bricks to build toys. As a result service components can be reused to construct different services. For example a *packet classifier* may be used in a *diffServ* as well as in a *intServ* service implementation.

The adequate degree of granularity of service components has been discussed for a long time. High granularity (i.e. small service components) maximizes the reuse potential of service components, whereas low granularity (i.e. big service components) may allow for better performance optimizations and simpler design of components. As it is safe to assume that opinions on the appropriate degree of granularity continue to differ, our service model must not impose any constraints on granularity. Section 2.2 shows that we achieve this using a hierarchical composition mechanism.

A model of active services, as well as of platforms (active nodes) such services are executed on, is essential to define an appropriate description language for both, services and nodes. The description language is necessary to automatize service composition. This section presents service and node models, as well as their formal description.

2.1 Service Model

Our service model is based on *implementation service components (ISC)*, which are the entities of deployment. An ISC is a unit of code that performs a well-defined functionality (e.g. packet classification) when run in an execution environment (EE). A service is based on one or more connected service components.

Two ISCs running in the same EE communicate to each other via *ports* using EE-specific mechanisms. For example, in a JVM-based EE a component performs method calls on the other. A component may have zero or more ports.

Ports are defined by their `name` (optional) and `type`. There are two basic types of port semantics: *push*, and *pull* ports. A component's *egress* push port initiates the forwarding of data to an ingress push port of the *succeeding* service component. Using pull ports, however, it is the *ingress* pull port of a component that triggers the transfer of data from the *preceding* service component. Note that it is not allowed to bind a pull port to a push port, irrespective of the direction of data transfer.

An ISC may as well communicate with a component in a different EE, as long as the nodeOS provides adequate mechanisms. Such communications are completely transparent to ISCs. It is up to the node to provide adapters for the different EEs that render inter-EE communication transparent.

ISCs feature a configuration interface. Configurations can be passed to the component in the form of parameters.

2.2 Service Composition Model

While section 2.1 described our service model from a runtime perspective, this section focuses on composition of service components.

The composition model is based on two main abstractions, *implementation service components (ISCs)*, as explained in section 2.1, and *abstract service components (ASCs)*.[2] The purpose of *ASCs* is to model dependencies by grouping a number of service components and describing their interconnections. Hence, we are able to express that e.g. a (simplified) *diffServ* service consists of a *packet classifier* and a *priority scheduler* component, where data packets are first processed by the classifier and subsequently by the scheduler. In this example the *diffServ* component is an ASC, because it refers to other SCs in order to be implemented. *Packet classifier* and *priority scheduler* may be represented by ASCs or ISCs themselves. These ASCs, in turn, refer to other SCs, resulting in a tree structure of dependencies of arbitrary depth, with all leaves being ISCs. In this way, our model enables hierarchical service composition.

Two main advantages result from this model. First it is up to service implementors to decide which granularity of service components suits their requirements best. Second any service can be reused as a SC to construct a new service by defining a new ASC grouping and connecting it with other SCs.

The semantics of an SC is implicitly defined by its name. Different implementations with the same name may exist, as the same functionality can be implemented in several ways and technologies. As consequence it is possible to compose a service with a defined functionality in many different ways. It is up to standardization bodies to define the semantics assigned to an SC referred to by a specific name. This namespace may be organized in a hierarchical way to accommodate standardized *and* proprietary SCs.[3]

2.3 Service Description

To handle service composition automatically, a service description language formalizing the models is necessary. We propose a markup language based approach, which is implemented in XML to benefit from the wealth of available tools.

We require that each SC is described in an XML document that provides the necessary meta-information. Henceforth, we refer to these documents as *descriptors*. The following sections describe their content and structure. Henceforth, XML *elements* are graphically represented as ellipses, their *attributes* as boxes. Dashed ellipses stand for optional elements. Double ellipses signify one or more, or, if dashed, zero or more elements.

2.3.1 Elements of ISC Descriptions. Figure 1 presents the structure of an ISC descriptor, as indicated by the attribute `type=IMPLEMENTATION`. Hence, this descriptor contains meta-information about a piece of code that provides a defined functionality. As explained in section 2.2, the Element `servicename` encodes the semantics of the component. That is we assume that knowing the servicename of a component means

[2] We use the term *service component (SC)* for the generic case where both ISC *or* ASC apply.

[3] This is similar to the namespace of Management Information Bases (MIBs), where the name defines the semantics of a variable and proprietary extensions are possible.

knowing its functionality. Other elements define the type of runtime environment (EE and operating system) the component was implemented for. Thus it is possible to write ISCs providing the same functionality for different runtime environments. A list of ports, including their types, describe how the component can be connected to others. It is possible to define a set of default parameters for the component. Parameters are listed as name/value pairs. An optional element `CODE_MODULE` contains references to the files that contain the code module. The reference takes the format of an Uniform Resource Locator (URL) [6]. Omitting this element is only possible if a resource discovery system is in place that maps XML elements such as servicename, provider, etc. to URLs. Such a system, however, is out of scope of this paper.

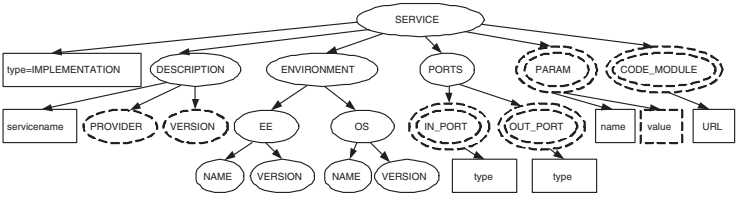


Fig. 1. Structure of ISC descriptor

2.3.2 Elements of ASCs Descriptions. In figure 2 the structure of an ASC descriptor is shown. The attribute `type=ABSTRACT` defines that the descriptor contains meta-information about an *abstract* service component. As for the ISC descriptor, the servicename defines the functionality of this component. There is a list of ports and their types to define how this component can be connected to others, as well as a list of default parameters in name/value notation.

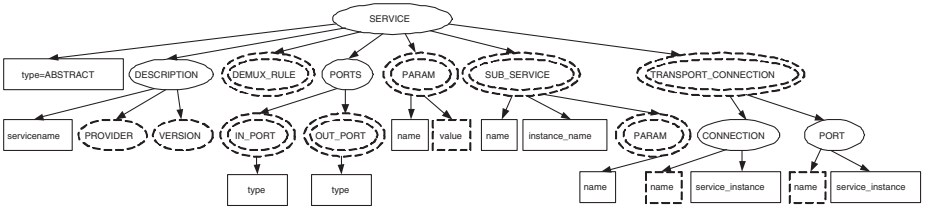


Fig. 2. Structure of ASC descriptor

An abstract service component descriptor has three distinctive elements: `SUB_SERVICE`, `TRANSPORT_CONNECTION`, and `DEMUX_RULE`. `SUB_SERVICE` is a list of references to service components the ASC depends on, which are identified by their `servicename`. Note that such references are ambiguous, as there exists probably more than one component with the same `servicename`. This ambiguity has been chosen deliberately to allow services to be composed in different ways. Section 3 discusses this issue

in detail. TRANSPORT_CONNECTION is a list of connections among sub-service components. In this way a topology of interconnected sub-service components is defined, which results in the functionality assigned to the servicename of the ASC. DEMUX_RULE contains a list of rules that define which packets entering the active node must be dispatched to the service component defined by the descriptor.

2.4 Active Node Model

Figure 3 shows the active node model our system is targeted at. The model is briefly presented here to pave the way for section 2.4.1, which deals with the formal description of an active node.

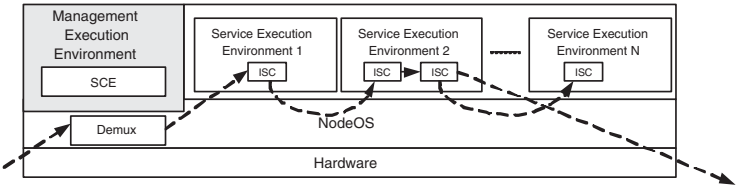


Fig. 3. Active Node Model

The node operating system (nodeOS) controls allocation of node resources (e.g. memory, CPU, storage, communication ports) to different EEs. Furthermore, it contains a demultiplexer, which forwards incoming packets to appropriate EEs.

There are two functionally different types of EEs on a node. First, there are *service EEs*, in which ISCs are run, providing a defined service to end-users attached to the network. Service EEs are often optimized for certain types of services components (e.g. control plane or transport plane components). Therefore they are implemented in many different technologies and provide different APIs to the service components. The set of service EEs a node provides is not defined.[4] Second, there is a *management EE*, which provides privileged access to the nodeOS to configure both, nodeOS and service EEs. The service creation engine (SCE) configures the demultiplexer and the service EEs.

2.4.1 Active Node Description. Based on the node model from figure 3 many different implementations of nodes are possible. To automate service composition, it is necessary to formally describe the capabilities of a node. Figure 4 presents the structure and content of a node descriptor.

The node descriptor contains an element describing the operating system of the node (OS) including its name and version. More important for service composition are the elements listing available EEs (EE) and inter-EE communication facilities (EE_CONNECTION). EEs are identified by their name and version. They are further characterized by the types of ports they provide. A reference to the EE-specific

[4] Selecting the "best" set of EEs is left to node operators or manufacturers. The offered set of EEs may serve as a means of differentiation among them.

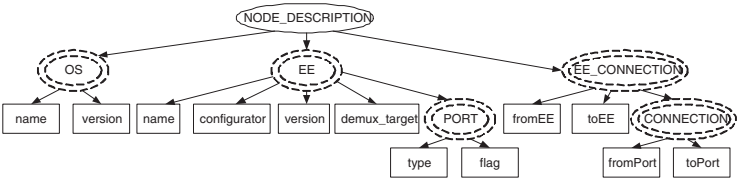


Fig. 4. Structure of active node descriptor

`configurator` is required by the SCE to load and configure the ISCs in the EE. Finally the `demux_target` defines the way an EE is attached to the demultiplexer. A node may provide facilities to allow communication among EEs. Such facilities are listed as `EE_CONNECTION`.

3 Automating Service Composition: The Service Creation Engine

This section presents the *service creation engine (SCE)*, a management subsystem that is able to interpret service and node descriptors. Based on these descriptors it performs automatic service composition. Hence, appropriate service components are selected according to node capabilities (e.g. types of EEs, communication facilities, etc.) and a mapping policy. The SCE performs for network services what compilers/linkers do for computer programs. While compilers produce executables for a specific computer architecture from source code, the SCE creates an executable service for a specific active node architecture from descriptors and code modules (so-called ISCs). Similar to cross-compilers, the SCE is capable of delivering different "executables" based on the target system being defined in the active node descriptor. Furthermore, the SCE provides means to load, bind and configure code components (ISCs).

3.1 SCE Overview

Figure 5 gives an overview of the SCE. The processing within the SCE involves two basic steps, which are carried out by the *composition engine* and the *deployment engine*, respectively. The composition engine is triggered by a service request issued by an authorized user of the node (e.g. the network operator). A service request has the format of an ASC descriptor. The composition engine interprets the request and – if necessary – retrieves SC descriptors of sub-components from a (possibly external) repository (descriptor server). Based on meta-information about the node, appropriate service components are selected. As a result we get a list of service components and information about their binding and configuration, which is passed to the deployment engine. The deployment engine parses the list and retrieves the necessary code modules from a (possibly external) repository (code server). Moreover, it interacts with EEs via EE-specific configurators to load, bind, and configure service components.

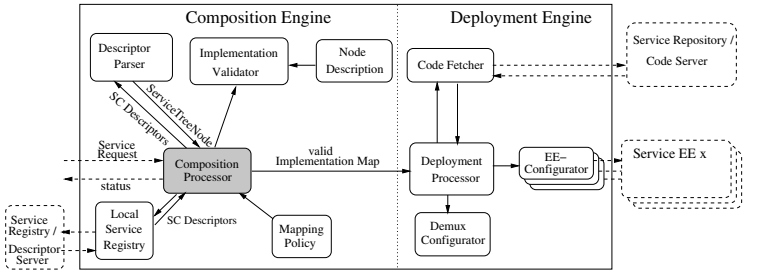


Fig. 5. SCE overview

3.2 Processing in the Composition Engine

As the core of the composition engine the composition processor coordinates the processing of descriptors and the composition of services. It utilizes several support modules to perform its task (see figure 5). This section discusses the processing within the composition engine chronologically.

3.2.1 Phase I: Dependency Resolution. The composition processor is triggered by a service request. Analyzing the initial ASC descriptor (included in the service request) with the help of the descriptor parser, one or more required sub-service components are identified. For each of those components one or more descriptors are fetched from the descriptor server. As a result we get a tree of all possible dependencies, which we call *service tree* because each node represents a service component. In the case of ASC descriptors other sub-service components are referenced and corresponding descriptors must be fetched. Hence more nodes are added to the tree. In the case of ISC descriptors, code modules are referenced and the resolution process stops for this branch. The service tree is finished when all leaves refer to ISCs.

It is important to note that each node may be represented by several SC descriptors. Hence the service tree may contain information about many possible combinations of ISCs that make up a service with the behavior defined in the service request. Further processing steps taken by the composition engine narrow down this set of combinations to one that is the most appropriate for the active node under consideration.

Parameter Handling. Any service component of the service tree (represented by tree nodes) may be configured by parameters. Two types of parameters are supported, which can be specified in the XML descriptors:

Required parameters only have a name without a value set in the considered node. They require the parent ASC to set the value. For a successful deployment it is necessary that an ASC passes the right number of required parameters to each of its children nodes. To do so, every service component sharing a common service name, has the same specific number of required parameters, which must be *"well-known"* (for example defined by a standardization body).

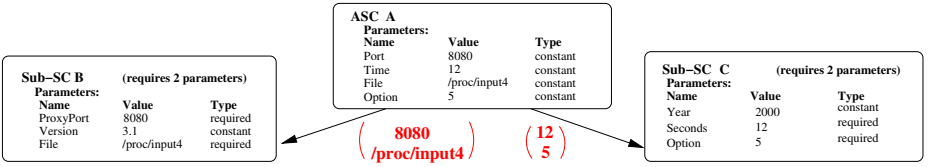


Fig. 6. Concept of parameter passing

Constant parameters have both name and value set in the considered node. The parent ASC does not need to deal with those parameters, but optionally overwrites their values.

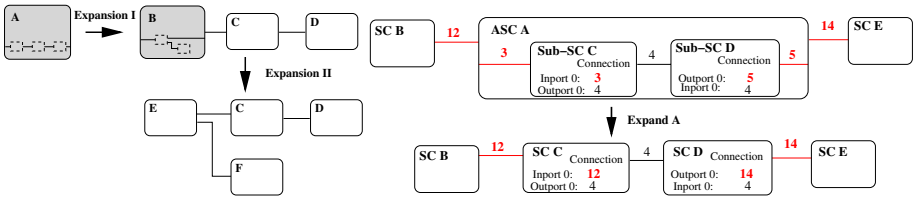
ASCs are allowed to pass both required and constant parameters to their children nodes. For better illustration of this concept, figure 6 shows an example setting, where SCs B and C are children of ASC A. The

Handling Demultiplexing Rules. Similar to parameters *demultiplexing rules* are associated with SCs. They filter the packets being destined for a given SC. To take this decision, the deployment engine must configure the demultiplexer and therefore needs adequate rules. In contrast to the *parameters*, the composition engine can handle them in a simpler way: wherever demultiplexing rules appear in a SC descriptor they are passed to all its children and children's children.

3.2.2 Phase II: Service Tree Node Validation. Although an extensive service tree has been built with possibly many ways to deploy it, it can not be passed to the deployment engine yet. It is the task of the composition engine to carry out extensive checks and validation steps to ensure that the service is deployable on the node under consideration. The node validation includes checks for the following attributes: OS name, OS version, EE name, EE version, types of the in- and outports.

To do so the composition processor recursively analyzes the service tree and passes all ISCs to the implementation validator (see figure 5). This component checks whether the attributes associated with the analyzed ISC are supported on a given active node by comparing the ISC descriptor to the active node descriptor. Invalid ISCs are removed from the service tree. Please note that removal of an ISC may lead to parent ASCs becoming invalid as well.

3.2.3 Phase III: Port Mapping. In phase II each service component in the service tree has been validated separately, leaving aside any bindings among them. Bindings are defined through ports and connections. In the previous phase, it was already checked whether the specified types of ports are supported by the respective EE. This phase extends the previous checks by mapping ports to connections. First, all named connection end-points (i.e. end-points specifying a port name) are mapped to the respective ports of an SC. Second, non-named connection end-points are mapped to non-named ports in the order of appearance in the descriptor. Third, it is checked that all unconnected ports have the attribute *optional*. SCs that fail to pass these checks are removed from the tree.



(a) expanding ASCs (b) routing connections

Fig. 7. Flattening the service tree

3.2.4 Phase IV: Service Tree Flattening. At this stage the service tree contains much information that is no longer relevant. Remember that the result of the processing in the composition engine is to be a list of ISCs including references to code modules and information about their configuration and binding. As a consequence, the hierarchical grouping of service components, which is achieved by ASCs, must be flattened. As several validation steps are carried out on the flattened data structure, we refer to it as *validation map*. All algorithms iterating the validation map consist of the same structure which is outlined in table 1.

Table 1. Validation map traversal

1	Go through all outports i
2	Go through all possibilities j of outport i
3	Trigger the logic of the algorithm in the successor node identified by (i,j)
4	Get the return of the successor triggered in 3
5	Execute the logic of the algorithm
6	Return the result of 5

To build the validation map all ASCs are expanded step by step (see figure 7(a)). In the beginning the validation map consists of nothing but the root ASC (A) of the service tree. In the first step this root node is expanded, which means that it is replaced by all its children nodes. The expand algorithm then expands all further ASCs until the validation map consists of (connected) ISCs only.

Figure 7(b) shows how connections are handled while expanding ASCs. Each connection between two SCs is identified by an unique ID. A port being attached to a given connection contains a reference to the connection ID. Let's now focus on the example in figure 7(b), where ASC A is to be expanded. Connections with ID 12, 3, 5, and 14 are affected by this operation. As a result of the previous port mapping operation, inport 0 of sub-SC C refers to connection 3, whereas outport 0 of sub-SC D refers to connection 5. To flatten the hierarchy, connections 12 and 3, respectively 5 and 14 must be merged. This is achieved by redefining the port mapping of ASC A's child components to refer

to the outside connections 12 and 14, respectively. Data structures related to ASC A, as well as connections 3 and 5 are no longer needed.

3.2.5 Phase V: Connection Validation. At this stage the validation map consists of connected ISCs only. The objective of the next validation map traversal is to remove all transport connections between ISCs which cannot be deployed on the active node under consideration. This is done by analyzing for each connection EE name and EE version of the two ISCs attached to it, as well as the port types involved. This information is compared to the active node descriptor to check whether the connections are supported by the active node.

3.2.6 Phase VI: Evaluating Routes. At this point validation is completed. The list of connected ISCs potentially yields a great number of possible installations of the requested service. The composition processor selects one of these and pass it to the deployment engine.

Therefore, the validation map is traversed to compute all possible routes. The list of routes can now be evaluated. To select one route a *mapping policy* is needed. This policy contains an algorithm the selection of components is based upon. Currently the mapping policy component of the SCE (see figure 5) enforces a policy to minimize the number of EE transitions. Given the modular design of the composition engine, other policies can be implemented and integrated easily with the SCE.

3.2.7 Phase VII: Creating the Implementation Map. As seen in section 3.2.4 the validation map consists of nothing but linked ISCs. So does the implementation map. Therefore the translation from the selected route to an implementation map consists of a transformation of some data structures. The main difference between these data structures is the fact that those of the validation map allow multiple successors for one outport. This is not the case for the implementation map. Once the translation has been completed the valid implementation map is passed to the deployment engine.

3.3 Processing in the Deployment Engine

The deployment engine (see figure 5) has two main tasks: it downloads, installs and configures ISCs as defined in implementation maps, and it configures active node facilities, such as the demultiplexer and inter-EE communication channels, accordingly. Thus, the deployment engine may need to insert adapters between ISCs to allow for communications across EE boundaries, which is transparent to the involved ISCs. Further, EEs are configured by the deployment engine to allow for transparent communication between the demultiplexer and ISCs as well.

3.3.1 EE Configurators. There is no standardized interface to configure EEs, neither now nor do we expect one for the future. We believe, however, that it is possible to agree on a common service model, like the one in section 2.1. For this reason, our architecture features EE-specific configurators, which translate the generic configuration commands

of the deployment engine, which depend on the service model only, into EE-specific ones.

For each EE on an active node, a corresponding configurator is registered with the SCE. In this way it is possible to cope with a wide variety of different EEs using the same SCE. Support for new EEs can be added to the SCE by simply updating the active node descriptor and installing an EE-specific configurator.

3.3.2 Demultiplexer Configurator. The demux configurator is kept simple. It gets a list of Netfilter style rules [3] and passes them to the demultiplexer. In our case no rule translation is necessary as demultiplexing is based on the Linux Netfilter subsystem [3].

4 Implementation

To validate our approach to automatic service composition, we implemented an active node prototype including our service composition system. Furthermore, a demonstration service was deployed on this node to serve as a proof-of-concept.

4.1 Service Creation Engine

The SCE is written in Java and (including EE-specific configurators) runs on its own JVM generally referred to as *management EE*. The configurators communicate with their respective service EE via Java Remote Method Invocation (RMI)[5]. RMI was selected for two reasons. First, it allows running the SCE in its own JVM, which results in a proper separation of the SCE from other node subsystems, such as service EEs and node OS. Second, it becomes simple to run the management EE, including SCE, on a physically distinct computer.Hence our implementation enables a physical separation of the management plane from forwarding/control planes. The overhead introduced by the RMI communication is not relevant in this context as only a limited amount of configuration information is exchanged.

We developed a common XML Schema [12] for service descriptors, which enables the SCE to check their syntactical correctness using a standard validating XML parser like Xerces [5].

4.2 Active Node Prototype

The prototype is based on a Linux 2.4.20 kernel. Its Netfilter [3] subsystem serves as the demultiplexer (demux). Each EE that is to receive packets from the demux has to provide an entry adapter element, as well as a target string. Adapter elements must be listed in the active node description and are inserted by the demux configurator into the implementation map.

To attach EEs to the demux a patch of the Netfilter subsystem of the kernel is necessary to provide a new target for data packets. Although this renders EE installation

[5] In the case of ClickEE, an RMI proxy was implemented to communicate with the proprietary *click installer*.

more complex, this solution was selected for performance reasons, as the entire demux is confined within the kernel space. The target string is required by SCE to compile Netfilter rules allowing to dispatch packets to proper service entry points.

Our node provides two service EEs: one, *ClickEE*, is based on the kernel internal Click router [10], whereas the other, *Chameleon Java EE (CJEE)*, runs on a Java Virtual Machine (JVM). Both support our component based service model, otherwise they are completely different. Click EE runs in kernel space, and implementations of the service components are compiled into the kernel. It needs a proprietary configuration file to be configured. This file is generated by the ClickEE specific configurator as a result of the deployment engines generic EE configuration commands. The CJEE on the other hand, runs in user space, and its service components may be retrieved from an external server. Its configuration interface implements the following methods: `add` service component, `connect` two components, `start` service thread, as well as a few others, which are not relevant here.

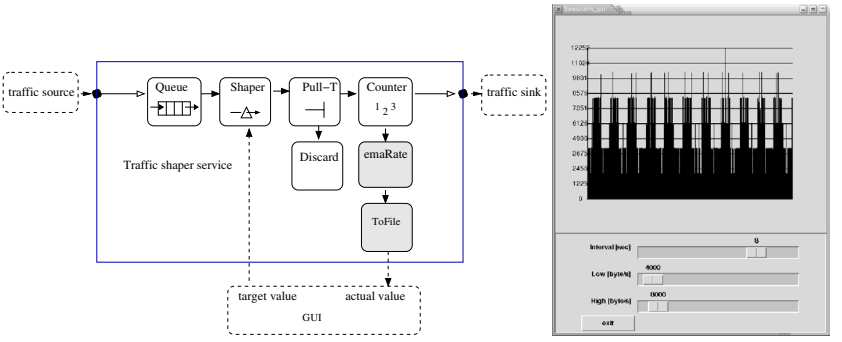
The node also provides inter-EE communication facilities. Our implementation supports the */proc file system (procfs)* and, to some extent, *Linux Netlink* for communication among service components in different EEs. Therefore we implemented communication adapters that allow both Click EE and CJEE service components to communicate with each other over the procfs and Linux Netlink.

4.3 Demonstration Service

To demonstrate the working of SCE and active node, we implemented a small service, which performs traffic shaping. The SCE deploys the service on our active node. Figure 8(a) shows a graphical representation of the implementation map resulting from the processing within the SCE's composition engine. Grey components are executed in the CJEE, whereas white ones run in the ClickEE. External components necessary for demonstration purposes are represented within dashed boxes. A simple GUI was implemented to set and modulate the *target value*, as well as to represent the temporal evolution of the *actual value* (see figure 8(b)).

The main goal of our system is to simplify service creation by providing means to compose services from components. Our experience with the demo service shows that this goal is achieved. To implement the service, we *reused* components provided by the developers of Click router and developed new ones for the CJEE. We had to write the descriptors for all the involved components. However, were such service descriptors to become a standard, we would expect the component developers to provide them as well. To glue components together, we wrote an ASC descriptor of our traffic shaping service, which is sent to the active node to trigger the deployment of the service.

The service was designed to demonstrate the major features of our system, which are *service composition, support for multiple EEs and inter-EE communication support*. This has been achieved. Figure 8(a) shows that the SCE composes the service from several service components, using different EEs. The composition algorithm selects, whenever possible, service components implemented in the faster EE (ClickEE). Communication adapters between *counter* and *emaRate*, inserted by the deployment engine, provide inter-EE communication support, which is transparent to the service components.



(a) service implementation map and external utilities (b) GUI screenshot

Fig. 8. Demo service: a traffic shaper

To carry out measurements all subsystems of our architecture are run as separate processes on the same active node, which is based on a Pentium IV 2 GHz CPU with 512 MB RAM. Communication among processes uses the same protocols and mechanisms as in a distributed set-up. The measurements do not reflect any delay caused by a network while fetching descriptors and code. Total processing of the SCE to compose and deploy the traffic shaper service takes on average $2.98s$ on a just booted node. Only $0.85s$ thereof are used by the composition engine. $2.13s$ are spent by the deployment engine to fetch code, and to load and configure components in the different EEs.

5 Conclusions

In this paper we presented design and implementation details of Chameleon, a system for automatic service composition. Our central contribution is the design and implementation of the service creation engine (SCE), which maps a node independent service description to a node specific implementation by composing appropriate service components. The resulting mapping depends on node capabilities described in the node descriptor. Based on our service and composition models, we further contributed an XML-based service description language. The language is used to write descriptors containing meta-information about service components. Descriptors are processed by the SCE to automatically select, bind, load, and configure service components for a specific node.

Finally, we implemented an active node prototype, including two distinctive EEs. This platform was used to demonstrate an example service, which was deployed using the SCE. Configuring EEs as different as the Chameleon Java EE and the Click-based EE demonstrated the ability of our approach to deal with a wide range of EEs.

The versatility of the Chameleon approach has further been shown with other active node implementations: In the European IST-FAIN project a subset of our system was

applied for node level service deployment [11]. Moreover Chameleon was used to deploy and configure services in the context of mobile networks [13].

Consequently, it is safe to state that our architecture consisting of description language and SCE provides the necessary means to cope with heterogeneity expected to be present in (partially) active networks.

References

1. Corba component model webpage. `http://www.omg.org/`.
2. Enterprise java beans webpage. `http://java.sun.com/products/ejb/`.
3. Netfilter webpage. `http://www.netfilter.org`.
4. Tina webpage. `http://www.tinac.com`.
5. Apache. Xerces2 webpage. `http://xml.apache.org/xerces2-j/`.
6. Tim Berners-Lee, Larry Masinter, and Mark McCahill. RFC 1738: Uniform resource locators (url), December 1994.
7. Matthias Bossardt, Takashi Egawa, Hideki Otsuki, and Bernhard Plattner. Integrated service deployment for active networks. In *Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN*, number 2546 in Lecture Notes in Computer Science, Zurich, Switzerland, December 2002. Springer Verlag.
8. Matthias Bossardt, Lukas Ruf, Rolf Stadler, and Bernhard Plattner. A service deployment architecture for heterogeneous active network nodes. In *IFIP International Conference on Intelligence in Networks (SmartNet)*, Saariselka, Finnland, April 2002. Kluwer Academic Publishers.
9. IETF. Forces working group. `http://www.ietf.org/html.charters/forces-charter.html`.
10. Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
11. Marcin Solarski, Matthias Bossardt, and Thomas Becker. Component-based deployment and management of services in active networks. In *Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN*, number 2546 in Lecture Notes in Computer Science, Zurich, Switzerland, December 2002. Springer Verlag.
12. W3C. Xml-schema webpage. `http://www.w3c.org/XML/Schema`.
13. Qing Wei, Karoly Farkas, Paulo Mendes, Christian Prehofer, Bernhard Pattner, and Nima Nafisi. Context-aware handover based on active network technology. In *Proceedings of the Fifth Annual International Working Conference on Active Networks IWAN*, Lecture Notes in Computer Science, Kyoto, Japan, December 2003. Springer Verlag.