A Service Deployment Architecture for Heterogenous Active Networks Nodes

Matthias Bossardt[1)], Lukas Ruf[1)], Bernhard Plattner[1)], and Rolf Stadler[2)]
[1)]*Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland*
{bossardt,ruf,plattner}@tik.ee.ethz.ch

[2)]*Center for Telecommunications Research, Columbia University, New York, USA,*
stadler@ctr.columbia.edu

Abstract In order to realise 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. The service specification is kept independent of any particular node architecture. During the service deployment phase, the service specification is resolved recursively 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 a video scaling service.

Keywords: Active networks, 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 multiprocessors, run service

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35584-9_19

O. Martikainen et al. (eds.), *Smart Networks*

components as kernel modules [1] and provide multiple concurrent execution environments.

Many types of execution environments have been developed each of which is usually optimised for a certain type of tasks. Active nodes typically provide more than one to support the whole spectrum of services. 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 recognise the diversity of execution environments, thereby allowing a service to exploit the particular functionality and performance features of active network nodes.

Our approach to service deployment on heterogeneous active network nodes, 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 controlling and managing the service components. 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 organised as follows. Section 2 outlines the service model. It is illustrated using the example of a video scaling service. Section 3 presents our approach to local service resolution and creation. The previously presented video scaling service is used to illustrate the deployment mechanism. Section 4 summarises related work. In section 5, we draw our conclusions and discuss future work.

2. SERVICE MODEL FOR ACTIVE NETWORK NODES

With Chameleon, we propose a service model that alleviates the development of services, and at the same time takes advantage of the flexibility and versatility offered by active networks. Services are modelled such as to support a flexible deployment on heterogeneous nodes, which is the main contribution of Chameleon. Typically, a service can be seen as a distributed application in the network. An active node provides the platform on which part of the service is run. In this paper, we focus on the node level.

The network level aspects exceed the scope of this paper. This section introduces the service model. We motivate our approach by presenting the requirements.

2.1 Requirements

Service models have been developed for various network and system architectures. Most models target the telecom environment [13], some of them end systems [2]. Similar to the work described in this paper, [7,9,12] feature models that focus on the node level. To a certain extent, they all allow for a recursive composition of services. However, to the best of our knowledge, they do not completely support our requirements as summarised below. For a more detailed discussion of the requirements, the reader is referred to [3].

2.1.1 Separating the Data Flow Interface from the Control Interface

The task of a router in traditional networks is to forward packets. 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 programming both the interconnection and configuration of processing components, as well as the actual processing in the components. 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. First, 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 an intuitive way to model a network service. Second, a data flow interface, which is separated from a control interface, facilitates the mapping of the service specification into an efficient implementation by mapping the data flow onto high performance communication facilities of the active node. An example for such a feature is described in [14], which proposes a zero-copy facility for packet processing.

The control interface allows controlling and managing a service component. It must provide a generic way to exchange control and monitoring information among processing components. The implementation of this interface depends on the specific execution environment and is also addressed by standardisation efforts [12].

2.1.2 Support of Different Active Node Environments

An active node may provide more than one execution environment. This is motivated by the following facts. First, each execution environment is usually optimised for a certain type of tasks. Some of them are rich in functionality, others are more restricted, but offer better security or better performance for certain tasks. Second, certain processing components are implemented for one specific execution environment. Finally, it is not realistic to assume that all active nodes provide the same set of execution environments [6], we require a service model to enable service specification that is independent of the node environment.

2.1.3 Support for Services Spanning Several Execution Environments

An optimum mapping of a service specification onto the node environment may result in an implementation that spans several execution environments. A packet classifier, for example, which processes packet at link speed, would be mapped to reconfigurable hardware execution environment [8], whereas a component that performs a routing algorithm, e.g. RIP, is more likely to be implemented in a CORBA execution environment. As a consequence, the service model must include abstractions for communication facilities between the service components. These communication facilities are execution environment specific. Moreover, the node may provide facilities that allow for communication between components in different execution environments.

2.2 Node Independent Service Model

This paper advocates a service model that is specialised for active networking. In order to fulfil the requirements from the preceding section, we need abstractions to model a service in a node independent way. We base the model on two basic abstractions, namely *composable containers* and *connectors*.

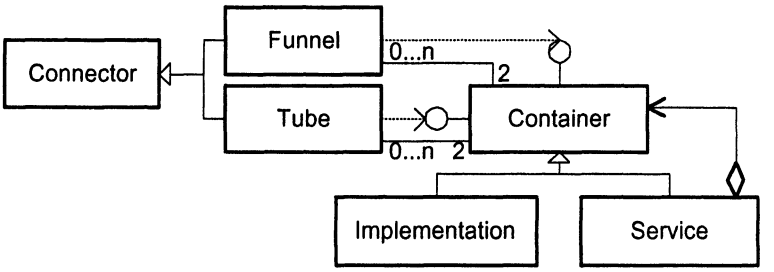

Figure 1. UML representation of the Chameleon service model

2.2.1 Composable Containers

We apply the *composition design pattern* [11], which allows a service to be structured in a hierarchical and recursive way. The UML diagram in Figure 1 shows that a *service* is composed of *containers*. We notice further that a *container* is a generalisation of a *service*, as well as of an *implementation*. A *container* defines the interfaces, which are common to both *service* and *implementations*. As a consequence, services and implementations can be handled the same way. A new service can be specified by composing existing services and implementations. An implementation, however, is not composed. It represents the primitive object of the composition.

> As a result of using the composition design pattern, a service instance can be represented as a tree of dependencies, which can be resolved recursively. The leaves of the tree are implementation objects, whereas the nodes are services.

An implementation represents both code modules that must be or are already installed in a specific execution environment, and services that are installed when the node is bootstrapped.

Since a container, is a generalisation of both a service and an implementation, it is up to the local mapping process at the active nodes to determine whether a specific container representing an implementation or a service. The resulting service tree may therefore be different for each node environment. We exploit this property when mapping a service description onto different node environments (see section 3).

2.2.2 Connectors

Figure 1 shows that *funnels* and *tubes* are a specialisation of a connector. Both are an abstraction for communication facilities between containers. Tubes bind two data flow interfaces, whereas funnels allow communication between control interfaces.

A tube transports packets and is unidirectional. Both end points of a tube have the same capabilities. One end point sends packets, whereas the other receives them.

A funnel, in contrast, binds two control interfaces of, possibly, different type. As a consequence a funnel is a more complex abstraction. Attributes of an *implementation's* control interfaces specify the implemented API. Based on this information the active node may provide adapters to enable communication between different types of control interfaces. It is assumed that control APIs specified by different standardisation efforts may coexist in this way. We do nót assume, however, that for each possible pair of control API standards an adapter can be provided.

In general, implementations of tubes and funnels are execution environment specific. The node, however, provides adapters and communication facilities that allow for an implementation of tubes and funnels across execution environment boundaries.

2.3 Modelling a Video Scaling Service

The goal of this section is to illustrate the applicability of our service model. We model a video scaling service as described in [4]. The service is based on a hierarchical encoding scheme, named WaveVideo [5]. The service measures the output queue length (congestion) of the WaveVideo stream. Based on this measurement, it selectively drops packets of the stream in order to adapt to the available bandwidth and to achieve a graceful degradation of the video quality.

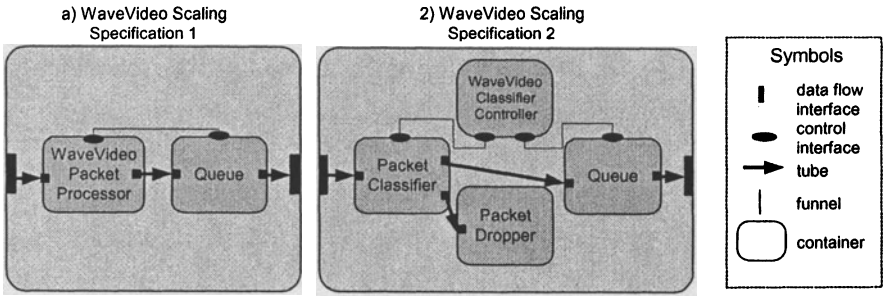

Figure 2. Visual representation of two specifications of the same service

Figure 2 shows a visual representation of two possible WaveVideo Scaling service specifications that may be registered in appropriate registries within the network. Specification b) contains more details than a), but both describe the same behaviour. Being more generic, specification a) allows for

more freedom during the mapping process on the node. We come back to this example in section 3.

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 is deployed within the node environment. In this paper, we investigate specifically service deployment on the node level. 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 request the deployment of new services. We will refer to the entity that requests a service as *service requester*.

A service request includes a specification that contains references to sub-services and a map of their interconnection, as shown in Figure 2. In our implementation, we use an XML document and a corresponding XML schema to represent such a service specification.

We consider a scenario where the service request is sent to a particular network node. This request contains a specification of the service to be installed locally. Such service specifications can be transported to a network node using a file transfer protocol, mobile agents, or encapsulating the specification in active packets.

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

3.1 The Local Service Creation Engine

On receiving a service request, the local service creation engine (see Figure 3) maps the request onto the local node environment. In the following, we describe the components of the local service creation engine in detail.

The **specification processor** is the heart of the local service creation engine. It controls the mapping process. The input to the specification processor is the service specification. When the mapping process terminates, an implementation map is generated. The implementation map is referring to

implementation objects (see Figure 1) that must be installed on the node and shows their interconnection.

The **specification parser** receives a service specification from the *specification processor*. The service specification is read and checked on syntactical correctness. Its output is a one level tree representation of the service specification.

The **local service registry** is used to lookup service specifications. The specification processor passes (sub-)service name and, optionally, service attributes to the local service registry. The local service registry matches the request with registered services or consults an external service registry. It subsequently returns a list of matching (sub-)service specifications to the specification processor. The local service registry contains also entries for services that are preinstalled in the execution environments when bootstrapping the active node.

The **mapping policy** contains rules that influence the decision process of the *specification processor* when selecting a service specification from the list generated by the local service registry. E.g. a policy might indicate that implementations for execution environment 1 are always preferred to those for execution environment 2.

The **node information base** contains information about the configuration of the active network node. The information includes the types of available execution environments on the active node. Furthermore, the node information base contains information about the types of funnels and tubes available. The node information base is part of the local management information base (MIB).

The **deployment engine** receives an implementation map from the specification processor and generates a specific implementation map for each execution environment. The execution environment specific implementation maps contain information about the code modules to be installed and configured, as well as their interconnection. If required, tube and funnel adapters, which allow for communication across execution environment boundaries, are inserted in these maps.

The **code fetcher** receives a list of code modules that are required to be installed on the node. The list is compiled by the *deployment engine* after the service mapping process. The code fetcher contacts the appropriate code repositories to download the code modules.

The **adapter repository** contains funnel and tube adapters, which may be requested by the deployment engine.

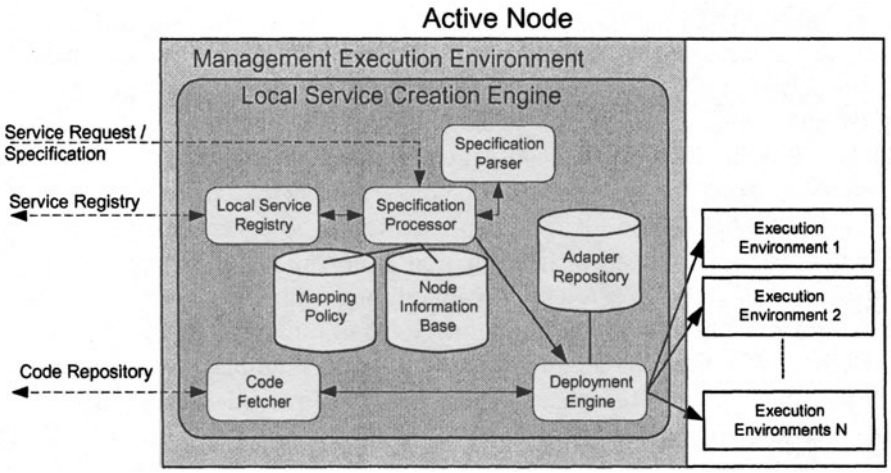

Figure 3. Active node with local service creation engine

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 *recursively* built, the leaves of which contain implementation objects. An implementation map containing these implementation objects and the description of their interconnection is passed to the deployment engine. In the second phase, the deployment engine builds a specific implementation map for each execution environment, triggers the fetching of the required code and the installation in the execution environments. In the following sections, we describe the mapping process in more detail.

3.2.1 Building the Service Tree

The process that builds the service tree is controlled by the *specification processor*.

Step 1. The *specification processor* receives a service specification from the service requester.

Step 2. The *specification processor* calls the *specification parser*, which transforms the service specification into a tree representation. The leaves of the tree are the sub-services the service is composed of. The tree is returned to the specification processor.

Step 3. The specification processor uses the *local service registry* to lookup specifications matching the (sub-)services referred by the leaves of the tree. As a result a list of service specifications for each leaf node is

passed to the *specification processor*. If a service specification points to an implementation, information about the location(s) where the code module can be retrieved from or a reference to a preinstalled service is returned. In the second case, in which a service specification points to (sub-)services, (sub-)service names and, optionally, attributes are returned.

Step 4. For each leaf of the service tree, the *specification processor* decides, which (sub-)service or implementation to insert in the tree. The decision process is based on information from the *mapping policy* and the *node information base*. If a leaf is mapped to an implementation, it must not be considered anymore in the remaining steps of building the service tree.

Step 5. If all leaves are mapped to implementations, the tree building process terminates. Otherwise steps 2--4 are repeated for the remaining leaves, which point to (sub-)services, in order to recursively build the service tree.

Step 6. The *specification processor* generates an implementation map and passes it to the *deployment engine*.

3.2.2 Installing the Mapped Service

The *deployment engine* controls the process of installing the mapped service.

Step 1. The *deployment engine* receives the implementation map from the *specification processor*. Based on this information, it constructs execution environment specific implementation maps, inserting references to tube and funnel adapters where required.

Step 2. The *deployment engine* contacts the *code fetcher* with a list of code modules and requests their download. It further retrieves the required tube and funnel adapters from the *adapter repository* and configures them.

Step 3. The *deployment engine* passes the specific installation maps and references to the code modules and adapters to the execution environments, where they are installed by execution environment specific mechanisms.

3.3 Mapping a Video Scaling Service

Applying the service specifications from Figure 2, we illustrate the mapping process as described above. An example in which the service is mapped onto two different node architectures can be found in [3].

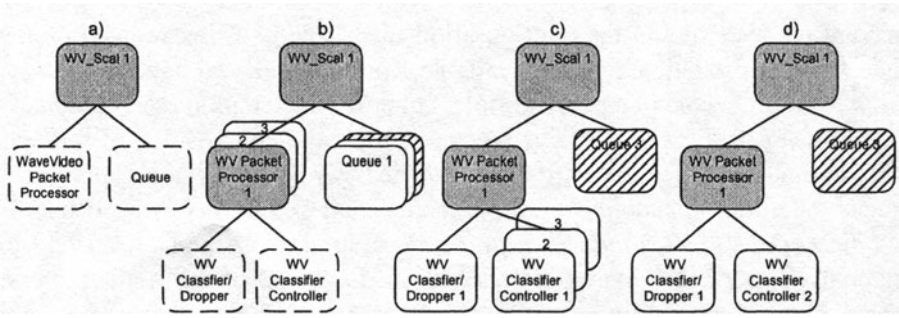

Figure 4. Four steps of resolving a WaveVideo Scaling service specification on a node

The active node gets a request to deploy a service according to the service specification from Figure 2a). Figure 4 illustrates this subsequent mapping process. The service specification is composed of two sub-services, as can be seen in Figure 4.1. The *container discovery* is used to lookup specifications of these sub-services. Three matching specifications of `WVPacketProcessor` as well as three specifications of `queue` are retrieved (cf. Figure 4.2). One `WVPacketProcessor` specification (grey box) points to two sub-services, whereas the remaining two (white boxes) point directly to implementations for specific execution environments. One `queue` instance is a preinstalled (sub-)service available on the node (hatched box), the other two are pointing to implementations as well. We notice in figure Figure 4.3 the *specification processor* decided - after consultation of the *node information base* and the *mapping policy* - to use `queue_3` and continues the recursive resolving process for the composed `WVPacketProcessor_1`. The resulting implementation consists in three implementation entities (cf. Figure 4.4). One is provided by the node (`Queue_3`), the other two instances are code modules (`WVClassifierDropper_1, WVClassifierController_2`) and must be downloaded by the *code fetcher*. The *deployment engine* requests the appropriate execution environments to install the code and binds the interfaces according to the implementation maps. The binding of the interfaces might require the insertion of adapters - abstracted by tubes and funnels - in order to cross execution environment boundaries.

4. RELATED WORK

Click Router [7] proposes a software architecture for configurable routers. Click Router can interpret a configuration file, and create and configure services. Services are composed of C++ elements that run in a kernel execution environment. Click Router assumes a direct mapping of the

element names listed in the configuration file to the ones instantiated on the node. All required elements are available from a node local code repository. Click Router does not support multiple, concurrent execution environments.

NetScript [9] uses a recursive mechanism for the composition of services from components. However, it does not deal with the service deployment process and does not support multiple, concurrent execution environments.

The *IEEE P1520 standards initiative* [12] aims at a standardised API to the control interface of network elements. In low level of the architecture, it uses a similar service composition mechanism. P1520, however, does not specify service deployment mechanisms.

TINA-C [13] specifies a network architecture for the telecommunication environment. Network services run on a Distributed Processing Environment (DPE). TINA-C computational objects provide an operational interface and a stream interface that allows separating the data flow from the control flow.

5. DISCUSSION AND FUTURE WORK

In this paper, we described a service model that 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, uses locally available information from the node information base and local policies for the mapping in order to take advantage of specialised execution environments provided by a node. The presented algorithm is greedy in the sense that on each level of building the service tree, a decision is made on which sub-service to use. Therefore, the algorithm may not find a globally optimum mapping. A simple extension, based on backtracking, would allow performing an exhaustive search of possible service mappings. We expect the mapping space to be relatively small. Therefore, we consider an exhaustive search to be a valid approach.

Future work will investigate the service deployment on a network level and its integration with the work proposed in this paper. Moreover, we will evaluate whether our assumption about the mapping space is valid. 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 evaluate Chameleon within the testbed of the IST-FAIN project (5th EC

Framework Program) [10], which aims at developing an architecture for future high performance active networks.

ACKNOWLEDGEMENTS

We gratefully acknowledge the feedback we received on an earlier version of this paper and during discussions with the FAIN project members. This work is funded by ETH Zürich and Swiss BBW under grant number 99.0533. It is part of ETH's contribution to the IST Project FAIN (5th EC Framework Program).

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] 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] Bossardt, M., Stadler, R. Service Deployment on High Performance Active Network Nodes. TIK Technical Report 122, ETH Zürich, Switzerland, 2001.

[4] Keller, R., Choi, S., Decasper, D., Dasen, M., Fankhauser, G. and Plattner, B. An Active Router Architecture for Multicast Video Distribution. Infocom 2000, Tel Aviv, Israel, 2000.

[5] Fankhauser, G., Dasen, M., Weiler, N., Plattner, B., Stiller, B. WaveVideo - An Integrated Approach to Adaptive Wireless Video. ACM Monet, Vol. 4 No. 4, 1999.

[6] Haas, R., Droz, P., Stiller, B. Distributed Service Deployment over Programmable Networks. DSOM 2001, Nancy, France, 2001.

[7] 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.

[8] 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.

[9] Da Silva, S., Florissi, D., and Yemini, Y., Composing Active Services in NetScript, position paper, DARPA Active Networks Workshop, Tucson, AZ, March 9-10, 1998.

[10] 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.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Addison-Wesley, 1995.

[12] Vincente, J., Denazis, S., Biswas, J. L-interface Building Block API. IP Working Group IEEE P1520, www.ieee-pin.org, March 2001.

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

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