Component-Based Deployment and Management of Services in Active Networks

Marcin Solarski[1], Matthias Bossardt[2], and Thomas Becker[1]

[1] Fraunhofer Institute for Open Communication Systems FOKUS
Berlin, Germany
{solarski,becker}@fokus.fhg.de
[2] Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, ETH
Zürich, Switzerland
bossardt@tik.ee.ethz.ch

Abstract This paper [1] describes a holistic approach towards the deployment and runtime management of services on active network nodes taken by the FAIN project. Both the underlying service model and the architectures supporting deployment and management are component oriented. The separation of service meta-information and implementation code allows for a very flexible way of service deployment management as it facilitates selective code distribution, fine-grained installation and instantiation. Active services are composed from a set of service components that can be selected on demand at deployment time and installed in any combination of the data, control, and management planes which enables realisation of arbitrary active services.

1 Introduction

Since the mid nineteen-nineties many efforts have been made to develop Active Networks technology [1] to enable more flexibility in provisioning services in networks. By defining an open environment on network nodes this technology allows to rapidly deploy new services which otherwise may need a long time and adaption of hardware.

There are two major approaches to Active Networks: on the one hand active packets (*capsules*) transmitting code along the data path which will be executed on the appropriate nodes and on the other hand a separated transmission of control and data packets. Whereas the first approach is suitable for deploying simple data path services (like new forwarding rules) with low latency, the latter one is more applicable for high-level, application-oriented services.

Deploying and managing high-level services requires an appropriate service model. While fully-fledged component-based service models are an integral part of many enterprise computing architectures (e.g. Enterprise JAVA Beans, CORBA Component Model, Microsoft's .NET), it is not the case in many approaches developed by the active networking community.

[1] This research is funded by the European Commission (IST-1999-10561), as well as by the Swiss Federal Institute of Technology (ETH) Zürich, and Swiss BBW (grant number 99.0533).

J. Sterbenz et al. (Eds.): IWAN 2002, LNCS 2546, pp. 87–98, 2002.

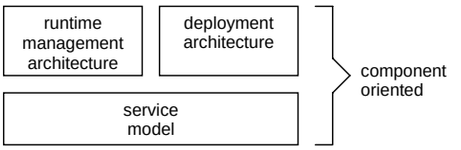

Fig. 1. Component oriented service model and supporting architectures

In this paper we present a *component-based* service model together with deployment and runtime management architectures based on this model (see figure 1). While the service model defines how services are described by potential recursive compositions of service components the deployment architecture defines how and when service components are transfered to active nodes and installed upon a service request. The runtime management architecture deals with the installation of serivce components in execution environments and the management of component instances.

The mentioned architectures have been implemented during the still ongoing FAIN project and used in the project test bed.

Section 2 describes the service model in more detail. Sections 3 presents the design and implementation of the runtime management architecture while section 4 presents the deployment architecture together with an example. In section 5 we show some related work and in section 6 we draw a conclusion and give an outlook on future work.

2 Service Model

This section describes the FAIN service model. The basic concept of the service model is the service *component*, a piece of self-contained software that is the smallest unit of deployment and management. A *service* is a unit of functionality that a service provider wants to offer to its customers. This functionality is realised by a combination of one or more service components.

The composition pattern is hierarchical in that service components may be recursively composed of sub-components. The goal of this approach is to enable flexible deployment and management of services at a suitable level of granularity, as well as to deal with services running in multiple execution environments with varied capabilities and underlying software technologies on possibly heterogeneous active nodes.

From the **deployment perspective** , a service component consists of a service descriptor and an optional reference to the service code stored in a file called *code module*. There are three classes of service components that differ in the terms of whether they consist of some service subcomponents and whether they directly refer to a code module: *(Simple) Implementation* – a service component without any dependencies. It contains just a reference to a code module; *Compound Implementation* – a service component consisting of subcomponents and having a reference to a code module; and *Abstract Implementation* – a service component consisting of subcomponents and having no reference to a code module.

From the **runtime perspective** a component is a unit of instantiation and activation. Runtime instances of components are identified by a unique name and are owned by

an identity defined by its name and credentials. Assigning an owner to each component instance allows to do access control and accounting based on identities. The configuration of a component instance is defined by a set of properties. The connections to the outside are described by the component's ports where components can offer arbitrary ports in addition to the basic ones, e.g. a bandwidth manager will offer an interface for reserving bandwidth in addition to the initial interface.

Components are accessed and interconnected by *ports*. Ports can be used for exchanging information or to model this exchange. Ports are also useful to express dependencies among components. A port has a name valid in the context of the holding component as well as a reference to the component itself. A port is described by a direction, an address (i.e. the endpoint for data exchange like an IP address, a memory address, an IOR, etc.), a format (i.e. the protocol used for data exchange like IP, ATM, IIOP, HTTP, etc.), and an optional type used for typed ports (e.g. a CORBA interface repository ID). The term "port" is used here to generalise from the notion of "interface" used in distributed object technology.

Each component offers a special port, namely its *iComponentInitial* interface. This interface is used for the initial access of a client to a component. Using this interface a client can retrieve the references to the other ports (and interfaces) supported by the component. In order to get access to a port of the component the client has to authenticate itself at this interface passing its credentials. The result is a reference to the requested specific interface tailored to the calling client. Components can offer arbitrary ports in addition to the basic ones, e.g. a bandwidth manager will offer an interface for reserving bandwidth in addition to the initial interface.

The management of components has two aspects. The component life cycle is managed using *component managers* as it is described in section 3. The other aspect is called *component configuration* and is a process of setting up or tuning the component behavior that may take place after the component is instantiated. The configuration of a component is described by properties as pairs of names and values. Properties can be used to define a component's behavior, e.g. a property may define a resource limit for a particular user or may restrict the access and usage of a component's interface. Interested clients can register for getting notified when particular properties change.

3 Node Management Architecture

This section describes the design and implementation of the runtime management architecture as it was developed during the FAIN project [13]. Since this architecture is based on the component-orineted service model its basic abstraction is as well the notion of components. Before presenting the desing in more detail a short introduction to the runtime environment for active services will be given.

The places where service code is executed are called *execution environments*. While this is common to all active network approaches the FAIN runtime architecture defines additionally the notion of *virtual environments*. Virtual environments (VEs) were introduced to abstract from the specifica of execution environments (EEs), for example, there are EEs implemented in hardware offering high performance, other EEs are implemented in interpreted languages focusing on flexibility.

A VE is owned by a service provider and may group several EEs assigned to the particular service provider. Thus, a VE is the "frontdoor" to the services running in the execution environment(s) of a service provider and used for their management. There is one special *privileged* VE owned by the node provider which holds the basic node services, like EE management, VE management, bandwidth management, etc. The priviledged is the main entry poitn for an active node.

When a service provider owns multiple VEs spread over several network nodes the VEs form a virtual active network. In order to identify which VEs belong to a virtual network they are tagged with a unique virtual network identifier.

3.1 Design

To support the component-based service model the design of the runtime management architecture also uses the component as the main abstraction. From the management viewpoint a component instance represents two aspects: firstly the functional aspect where the component is seen as a service instance and secondly the non-functional aspect where the component is seen as a resource.

The management architecture utilizes a special kind of component ports, namely *interfaces* (as known from CORBA [10]), and defines a basic set of interfaces which comprise the management API of the active node. They are used for example by the active service provisioning (ASP) module (see section 4.1) to install services on the node and later to create instances. Service instances use the same API to discover other services or resources. Because of the reflective nature of this API service instances can retrieve and modify metainformation about themselves.

The three basic types of interfaces used for the runtime management are *iTemplateManager*, *iComponentManager*, and *iComponentInitial*. The runtime management allows adding new ports to components so that they can publish their specific functionality. This allows for a very flexible way to construct services from basic components by combining and enhancing the already provided functionality.

In the following the abstractions defined by the runtime anagement will be described in more detail.

Template Manager. A template manager manages templates (e.g. JAVA classes, object files, etc.) from which component instances can be created. Management comprises the installation, removal, and updating of templates. This operations are available at the template manager's *iTemplateManager* interface. Template managers are implemented by VEs and EEs as the environments are the places where service components are installed. The active service provisioning (ASP) module will use the template manager of the priviledged VE to install a new template on the node and pass a description of the template.

A template description includes all the information which is necessary to create component instances. However, instead of having the template manager to deal with different instantiation methods the template description contains a factory for instances, called component manager. Further the template description includes the name and version of the template, the VE and EE identifiers, the path to the code archive of the

template, and optional additional properties defining template specific features. The VE and EE identifiers are used to determine the service provider for whom to install the template and the runtime environment for component instances (e.g. a JAVA virtual machine).

Component Manager. A component manager is used to manage instances of components associated to a specific template, thus it is acting as a component factory. Managing comprises the creation, activation, deactivation, discovery, deletion, and updating of instances. This operations are available at the component manager's *iComponentManager* interface.

The parameters for the creation of a component are specific to the type managed by the component manager (as defined by the respective template) and the result is the component's *iComponentInitial* interface. The component manager uses the parameters to check the possibility to create a new instance, i.e. the availability of the required resources.

Specific Component Managers. While template managers are only specific to the EE in which templates should be eventually installed the component managers are even more specific. For each type of components there needs to be a component manager. When a particular type of resource is represented by a type of component (e.g. a process represented by a component) there has to be the appropriate resource manager (e.g. a process manager). In order to define a manager one should specify the additional interfaces and operations if any, the properties supported for resource creation, and the dimensions and units supported for monitoring resources if applicable.

There are special managers for managing the basic services provided by an active node, namely security management, traffic management, packet demultiplexing, and management of execution environments as well as virtual environments.

Example. Figure 2 depicts an example snapshot of an active node. From left to right it shows the priviledged VE with the attached priviledged EE supporting *iTemplateManager* (iTM) interfaces. Inside the priviledged EE there are installed three templates with respective component managers supporting *iComponentManager* (iCM) interfaces: VE management, EE management, and management of an arbitrary resource "z". The VE manager created one VE, the EE manager created two EEs, and the arbitrary resource manager one resource instance. Both EEs and the resource instance are attached to the VE. Further, it can be seen that in EE1 there is installed a template "x" and in EE2 a template "y". From both templates an instance was created.

3.2 Implementation

To support the management of services and resources on an active node a framework was implemented comprising base classes for components, component managers, and template managers. The implementation was carried out in JAVA with a strong support for CORBA interfaces as communication ports (although components can implement

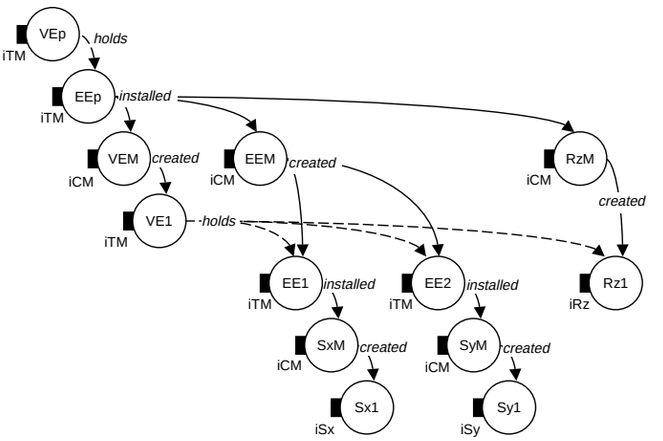

Fig. 2. Example snapshot of components on active node (iTM = iTemplateManager, iCM = iComponentManager, iSx = interface of service "x", iSy = interface of service "y", iRz = interface of resource "z")

whatever type of port is most appropriate). This framework facilitates the implementation of JAVA based services as well as services based on different technology by creating wrapping components.

Using the framework there were implemented a collection of basic services to support other services which would be deployed to the node dynamically. Those basic services live in the node provider's privileged execution environment and are thus able to access operating system resources. Other services living in the service providers' environments can use the functionality provided by the basic services after successfully accessing their interfaces.

4 Service Deployment Architecture

Service deployment is considered in this paper a process of making a service available in the active network to the service user. It involves determining the target environment, identifying the service components needed, mapping them to the target environment, fetching the code, as well as installing and activating service components in their target environments.

This section describes the FAIN service deployment architecture called Active Service Provisioning (ASP) system. The system is defined by its main functionalities in section 4.1. The design of the ASP system, including the architectural components and their interrelationship is the contents of section 4.2. Finally, the operation of the ASP system is explained in a concrete deployment scenario for an example transcoder service in section 4.3.

4.1 Functionality of the Active Service Provisioning System

In this section, the main functionality of the Active Service Provisioning system is described. The main actors communicating with the ASP system are:

Service Provider, or SP for short, composes services that include active components and deploys these components in the network via the Active Service Provisioning, and offers the resulting service to Consumers.

Active Network Service Provider, or ANSP for short, provides facilities for the deployment and operation of the active components into the network. Such facilities come in the form of an active middleware, support of new technologies etc.
The ANSP owns an active network offering one or more environments where active code from Service Providers can run.

These roles are described in the FAIN Enterprise Model in [12] in more detail. The main use cases of the ASP system are:

Releasing a service. The Service Provider who decides to offer his service in the active network has to release it in the active network. The service is released by making the service meta-information and service code modules available to the ASP system.

Deploying a service. After the service is released in the network, the Service Provider may want to deploy his service so that it can be used by a given service user. It means finding a target environment that is most suitable for the given user, determining a mapping of the service components to the available EEs of the target environment, downloading the appropriate code modules, and finally installing and activating them.

Removing a service. The Service Provider may request to remove a deployed service from the environment it was deployed in. The ASP identifies the installed service components and removes them from the EEs of the target environment.

Withdrawing a service. A service released in the active network may be withdrawn so that is is not available to be deployed any more. The ASP removes the service metainformation and discards the service code modules.

4.2 Design

The design of the ASP system follows a two tier approach. We distinguish between network and node level ASP system. The network level functionality consists of metainformation and code module management, as well as the selection of nodes that are to execute part of the service logic. On the node level, necessary service components are identified, and dependencies are resolved. What follows is a more detailed description of the ASP design, and how the functionality presented in section 4.1 is achieved by our implementation.

Service Descriptor. The service descriptor supports a service model as described in section 2. The first part of the service descriptor holds information about the developer or provider and the functionality of the corresponding service component. The second part is dependent on the class (cf. section 2) of service component described. For a *simple implementation* this part contains a reference to a code module and identifies

the target environment where the code module is to be installed. It further contains EE-specific information, which is used to perform EE-specific part of deployment process. The service descriptor of an *abstract implementation* holds information about required subcomponents and how they are to be bound to each other in order to perform the expected functionality. Finally, a *compound implementation* is a mixture of the two classes above, and hence contains both sets of information. The service descriptor is implemented in XML, which proved to be a very suitable technology for the task at hand. Furthermore, we developed an XML Schema to verify the structure and correct syntax of service descriptors.

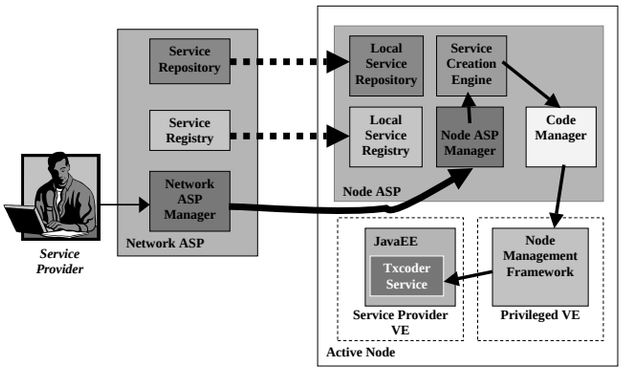

Fig. 3. ASP interaction when deploying an example service

Network Level ASP Design. The network level ASP system consists of three components depicted in figure 3: Network ASP manager, Service Registry and Service Repository.

The **Network ASP Manager** serves as an access component to the ASP system. In order to initiate the deployment of an particular service, a Service Provider contacts the Network ASP Manager and requests a service to be deployed as specified by the service descriptor.

The **Service Registry** is used to manage service descriptors. Service descriptors are stored on it, when a service component is released in the network. Network ASP Manager and the Service Creation Engine may contact the Service Registry to fetch service descriptors.

The **Service Repository** is a server for code modules. A code module is stored on the Service Repository, when a service descriptor referencing the particular code module is released in the network. The Code Manager, which is part of the node level ASP system, may fetch code modules from the Service Repository.

Node Level ASP Design. On the node level, the following components make up the ASP system as shown in the *node ASP* block of figure 3: Node ASP manager, Service creation engine and Code Manager.

The **Node ASP Manager** is the peer component to the network ASP manager on the node level. The network ASP manager communicates with the node ASP manager in order to request the deployment, upgrading and removal of service components.

The **Service Creation Engine** (SCE) selects appropriate code modules to be installed on the node in order to perform the requested service functionality. The service creation engine matches service component requirements against node capabilities and performs the necessary dependency resolution. More details about this mapping process can be found in [8], [9]. Since the service creation engine is implemented on each active node, active node manufacturers are enabled to optimize the mapping process for their particular node. In this way it is possible to exploit proprietary, advanced features of an active node. The selection of service components is based on service descriptors.

The **Code Manager** performs the execution environment independent part of service component management. During the deployment phase, it fetches code modules identified by the service tree from the service repository. It also communicates with Node Management to perform EE-specific part of installation and instantiation of code modules. The Code Manager maintains a database containing information about installed code modules and their association with service components.

4.3 Deploying an Example Service

This section describes a scenario in which an example active service is being deployed. After the structure of the service is presented, the details of the ASP components interactions are given.

Example Service. In order to evaluate our architecture, a transcoder service has been implemented [15] and deployed in the FAIN test bed (cf. figure 5). The service functionality is implemented in two code modules, a transcoder controller module and a transcoder engine module. Both are to be deployed on the same node. Three service descriptors are used to hold the corresponding meta-information. The first service descriptor holds information about an *abstract implementation* of a *transcoder*. In particular, it contains references to two sub-services – a transcoder controller and a transcoder engine – which make up the final transcoder service. Both sub-services, transcoder controller and transcoder engine, may be *abstract implementations* themselves. In the transcoder version we implemented, however, the sub-services are *simple implementations*. That is, each sub-service consists of service descriptor and associated code module. The service descriptors for *simple implementations* (cf. figure 4.3) holds both EE-unspecific and EE-specific information. The EE-unspecific part identifies, among others, the EE in which a particular code module is supposed to be executed, as well as the name and/or location of the code module. For our Java-based EE, the EE-specific part, which can be found in the *PROPERTIES* section, contains information such as codepath, and main class name, and others.

Figure 5 presents the configuration of the FAIN test bed and the system components involved in the deployment process. The FAIN test bed contains active nodes located at FhG, ETH and UPC, the FAIN Consortium partners' sites. On each of the active nodes, both the Node Management Framework and node-level Active Service Provisioning

```
<?xml version="1.0" encoding="UTF-8"?>
<SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:chameleon.xsd" xsi:type="IMPLEMENTATION">
   <DESCRIPTION>
      <SERVICE_NAME>transcoder_controller</SERVICE_NAME>
      <SERVICE_ID/>
      <PROVIDER>FAIN</PROVIDER>
      <VERSION>0.1</VERSION>
   </DESCRIPTION>
   <PROPERTIES>
      <PROPERTY>
         <KEY>mainClassName</KEY>
         <VALUE>org.ist_fain.services.transcoder_controller.
         TranscoderManager</VALUE>
      </PROPERTY>
      <PROPERTY>
         <KEY>mainCodePath</KEY>
         <VALUE>/usr/local/jmf-2.1.1/lib/jmf.jar:/usr/local/jmf-2.1.1/lib/
         sound.jar:/usr/local/jmf-2.1.1/lib</VALUE>
      </PROPERTY>
   </PROPERTIES>
   <ENVIRONMENT>
      <EE_NAME>JVM</EE_NAME>
      <EE_VERSION>1.3.1</EE_VERSION>
   </ENVIRONMENT>
   <CODE xsi:type="CODE_LOCATION">
      <CODEBASE>jvm.transcoder1.FAIN.transcoder1.jar</CODEBASE>
   </CODE>
</SERVICE>
```

Fig. 4. XML service descriptor example

systems are running. The network ASP components are located so that the Service Repository is at UCL in London, the Service Registry at FhG and the Network ASP Manager at UPC.

In an example scenario, the Service Provider requests deployment of his transcoder service. The service is composed of two service components that need to run colocated on a Java-enabled active node. The network ASP decides that the optimal target environment is the active node at ETH and sends a node-level deployment request to the node ASP on the target node. The node ASP processes the requests by resolving service dependencies, fetching the needed code modules and triggering the EE-specific installation process. Finally, a transcoder service instance is created.

Now the service is ready to use. The service provider may configure the transcoder to convert a JPEG video stream into a H.263 format and to forward the video data to a given receiver.

5 Related Work

The work of Marcus Brunner et al. on **Virtual Active Networks** [2] defines an architecture for the creation and management of services in an active network. However, it lacks the component-oriented service model allowing to flexibly combine functionality out of service components. The framework defined by the **IETF ForCES** [4] separates functionality found on a network node into forwarding and controlling elements. Further,

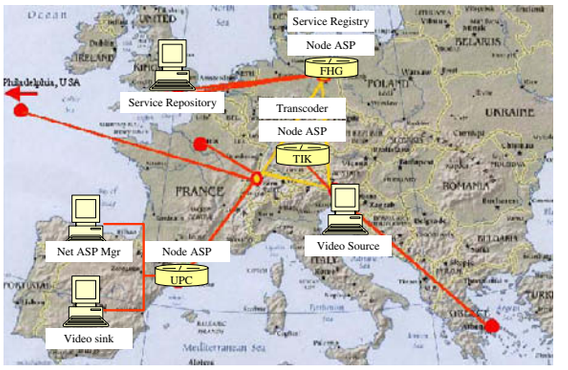

Fig. 5. Deploying the transcoder service – component distribution

ForCES aims at defining a model which describes how those elements are connected to form a self-contained network element.In comparison to our work ForCES seems to have a narrower scope focussing on the control and forwarding planes. **NetScript** [6] 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 **LARA active router architecture** developed at the University of Lancaster [5] is also based on components. This architecture allows to create services from components in a flexible and extensible way. However, this work focuses on the runtime aspects and doesn't define a mechanism to map a service description to interconnected component instances. The **AMNet project** now continued in the flexinet project [7] defines an architecture for programmable networks. Service modules can be loaded from a repository to active nodes where they run inside execution environments on top of a resource control layer. Although it is possible to combine modules out of multiple object code pieces there seems to be no explicit component model for services. The **CORBA Component Model** (CCM) [11] defines languages to specify components and their implementations. Further it defines a deployment and runtime environment for components. On the other hand it concentrates on enterprise applications and sticks to a client-server model. It doesn't allow to introduce new kinds of ports (e.g. stream oriented) for connecting components.

6 Discussion and Future Work

The approach presented in this paper covers the whole process of handling services in active networks beginning with a service model and stretching to deployment and runtime architectures. The choice of a component-based approach facilitates the fine-grained service description, deployment, and management. In the completeness of our approach, not focussing on one aspect while neglecting others, we see its novelty. Particular features are: out-of-band deployment, separation of service metadata and code, node-level service component dependency resolution, component-oriented service model and support for heterogeneous service implementations.

Our concepts have been designed and implemented as part of the FAIN architecture for active networks. First working prototypes can give a qualitative proof for the feasablitity of the design. A quantitative evaluation is pending and will be tackled during the ongoing last year of the project using the project's international testbed.

Besides the quantitative evaluation our future work will focus on extending our architecture with regard to its flexibility and robustness. On the node level, we intend to consider upgrades of running active services on the fly to achieve better availability of rapidly changing implementations of the services. Further, we target for an optimisaton of the network-wide service code distribution.

References

1. Tennenhouse, D.L., Wetherall, D.J.: Towards an Active Network Architecture. Computer Communication Review, Vol. 26, No. 2, April 1996
2. Brunner, M., Plattner, B., Stadler, R.: Service Creation and Management in Active Telecom Networks. Communications of the ACM, April 2001
3. 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
4. IETF ForCES Working Group: Forwarding and Control Element Separation, http://www.ietf.org/html.charters/forces-charter.html
5. Schmid, S., Finney, J., Scott, A.C., Shepherd, W.D.: Component-based Active Network Architecture. Proceedings of 6th IEEE Symposium on Computers and Communications, 3-5 July 2001
6. Da Silva, S., Florissi, D., Yemini, Y.: Composing Active Services in NetScript. Position paper, DARPA Active Networks Workshop, Tucson, AZ, March 9-10, 1998.
7. Fuhrmann, T., Harbaum, T., Schller, M., Zitterbart, M., AMnet 2.0: An Improved Architecture for Programmable Networks, submitted to IWAN'02, available from www.flexinet.de
8. Bossardt, M., Ruf, L., Stadler, R., Plattner, B.: A Service Deployment Architecture for Heterogeneous Active Network Nodes. Kluwer Academic Publishers, 7th Conference on Intelligence in Networks (IFIP SmartNet 2002), Saariselkä, Finland, April 2002
9. Bossardt, M., Ruf, L., Stadler, R., Plattner, B.: Service Deployment on High Performance Active Network Nodes. IEEE Network Operations and Management Symposium (NOMS 2002), Florence, Italy, April 2002
10. OMG: The Common Object Request Broker: Architecture and Specification. http://www.omg.org/cgi-bin/doc?formal/02-05-15.pdf, Mai 2002
11. OMG: CORBA Components final submission. http://www.omg.org/cgi-bin/doc?orbos/99-02-05
12. FAIN Deliverable D1: Requirements Analysis and Overall Architecture. FAIN Consortium, May 2001, pp. 11-18
13. FAIN Deliverable D4: Revised Active Node Architecture and Design. FAIN Consortium, May 2002
14. FAIN Deliverable D5: Revised Specification of Case Study Systems. FAIN Consortium, May 2002
15. FAIN Deliverable D6: Definition of Evaluation Criteria and Plan for the Trial. FAIN Consortium, December 2001