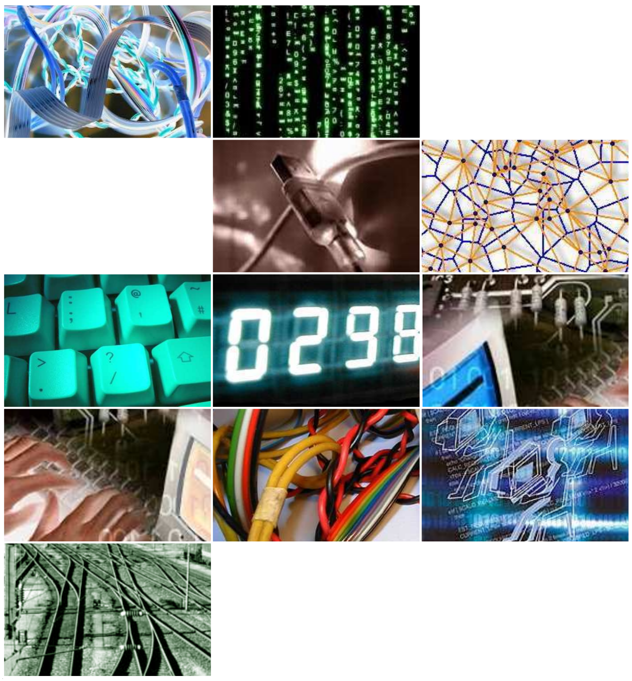
Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Wintersemester 2002/03

Diploma Thesis DA-2003.06

Service Composition for Active Networks



Authors:
Andreas Moser
Roman Hoog Antink

Supervisor:
Matthias Bossardt

Professor:
Prof. Dr. Bernhard Plattner

Co-Supervisor:
Lukas Ruf

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Within the area of active networks, the problem of installing and configuring software components in complex and heterogeneous node environments is a major issue. This thesis comprises design and implementation of an active node. A standard Linux installation on a PC has been used as platform. The framework managing the node is called service creation engine. Its task is to map an incoming node-independent service request to the particular node architecture. This includes resolving internal dependencies of the requested service by consulting a remote service server and installing the required software components in the available runtime environments of the node, called execution environments. Currently two execution environments have been integrated. The first is based on Java and allows service composition for userspace in a simple and flexible way. The second runs in kernelspace and features high performance needed for packet forwarding. It consists of the Click Modular Router which provides numerous different service components. Measures which allow these two execution environments to communicate with each other have been taken. By this means, services which run in both execution environments at the same time may be installed.

The active node has been designed in such a way that new execution environments may be added without the need for modifications of the service creation engine. In addition, the problem of demultiplexing packets coming from the network interface cards has also been addressed within the scope of this thesis.

Contents

List of Figures

List of Tables

Acknowledgments

We would like to thank Matthias Bossardt and Lukas Ruf for their reliable and dedicated supervision.
Lots of kudos to the Computer Engingeering and Networks Laboratory of the ETH for providing hardware and support.
Thanks to our wives and girlfriends Jérémie and Viola for bearing us during the stressful times of the thesis.
Special thanks to Yvonne Schnetzler, the designer of our marvelous cover picture.

Zurich, March 18th 2003

Andreas Moser
Roman Hoog Antink

Chapter 1

Introduction

The research of *Active Networks* is still ongoing. Whereas some standards have been already designed many issues still remain open. Active networks make possible things never seen before but they also introduce new problems that demand a solution in order to make active networks become a reasonable approach for network related applications.

Active networks are based on the infrastructure known at present and extend them with new functions. As depicted in figure 1.1 an active network may contain both so called *active nodes* (surrounded by bold boxes) and ordinary nodes (surrounded by rectangles). The bold dashed lines are network connections between nodes for application data transfer.

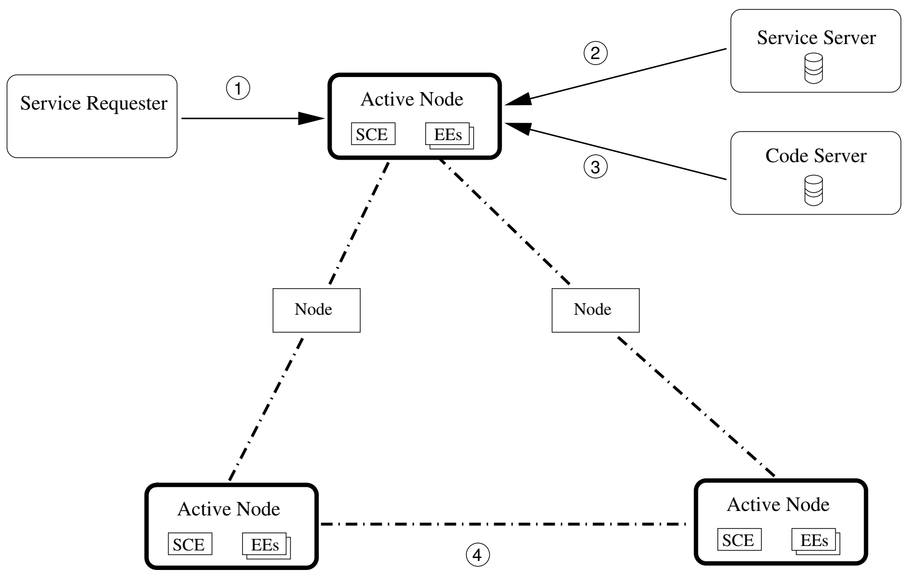


Figure 1.1: Active Network Overview

The purpose of an active node is to provide services to users. A service can be anything which processes network data. Typical applications are multimedia stream recoding, web caching, intrusion detection and so forth.

Services are installed upon user requests during runtime on active nodes. The program code which provides those services is downloaded and installed dynamically from the network. The numbers 1-4 in figure 1.1 show the order of events when a service is established and which components are required for doing so. Note that many different models of active nodes exist. Some details discussed in this chapter might be specific to our diploma thesis. The events 1-4 in figure 1.1 are:

1. The user requests the installation of a service on the active node.

2. The active node resolves all internal service dependencies. Services may be structured in a hierarchical way, thus they can consist of a host of subservices. The service server provides descriptions of services. By this means the active node is able to determine the structure and internal requirements of services.

3. The code of the specific services and subservices may be downloaded from the code server.

4. After the installation of the service, the application data flowing through the network is processed on each active node. As usual, packets travel through the network from source to destination. On their way they encounter active nodes where they might be manipulated, dropped or redirected depending on the installed services.

An active node basically consists of a local *service creation engine* (SCE) also referred as management EE in literature, and one or more *execution environments* (EEs) where services are running in. The SCE manages and installs the services on the local active node. The EEs are the “living rooms” for services.

Performance is an important issue on active nodes. Therefore the integration of a kernel-based EE was one of our objectives. We also integrated a Java-based EE which allows service composition in a flexible and simple way. Due to the typical low performance of Java applications, our Java EE’s task is basically to process non-time-critical *control* data[1]. In contrary, the kernel EE mainly processes time-critical *application* data.

In chapter 2 the architecture of the active node is explained.
Chapter 3 focusses the SCE which includes the resolution of service requests and installing services.
In chapter 4 the patform specific concepts and mechanisms needed for a successful service deployment are discussed.
Chapter 5 summarizes the achievements of our thesis, whereas chapter 6 states what could be done next.

[1]For example routing table information or scoring data.

Chapter 2

Architecture

Figure 2.1 shows the overview of an active node. This represents the basic architecture and starting point of the implementation of our active node. As development platform we used the Linux kernel. In the following, the particular components of the active node are described (see figure 2.1):

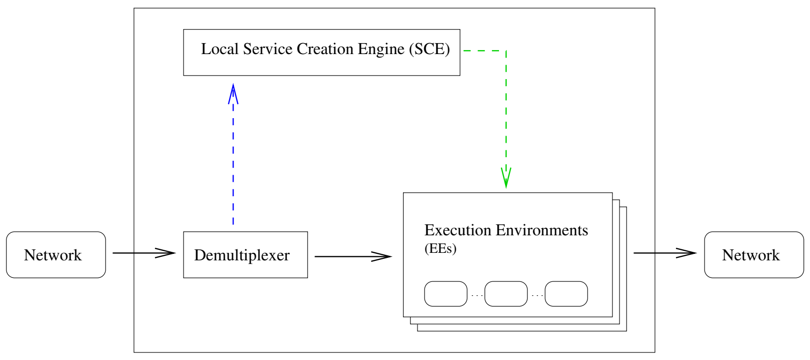


Figure 2.1: Active node architecture

Network The hardware network interface cards (NIC) where network packets come from and go to.

Demultiplexer Network packets coming from a NIC have to be directed to the appropriate destination service. This is done by the demultiplexer.

Service Creation Engine The management EE providing an interface for service composition and installation. It configures both the services in the EEs and the demultiplexer.

EEs The active node provides various EEs in which service components can run.

2.1 Proposals

Starting from this basic architecture, we evaluated several design proposals for the demultiplexer. The following two architecture proposals have been under consideration together with the final architecture presented in section 2.2.

2.1.1 Netfilter with Ipqmpd

As depicted in figure 2.2, the Click router environment sees incoming packets first. If they are meant for a userspace EE, they are passed to Netfilter, which in turn forwards them to the accurate userspace EE via Ipqmpd.

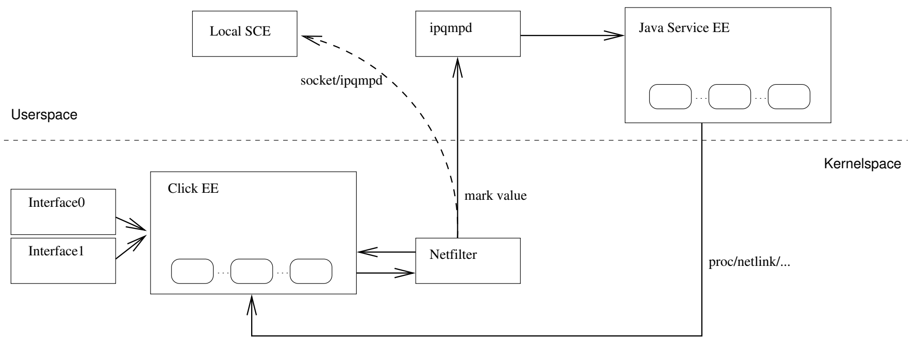


Figure 2.2: Architecture proposal with Ipqmpd

The necessity of also using the Click EE as demultiplexer makes this architecture less flexible and asymmetric. Furthermore, Ipqmpd[1] is still a *short time hack* and not very well documented.

2.1.2 Without Netfilter

Another possibility is shown in figure 2.3. Demultiplexing is done also with Click but there is no need for Netfilter. Packets meant for a userspace EE are passed over directly.

As the proposal with Ipqmpd this architecture might be optimal if Click is an obligate component of the active node, since packets destined for Click do not travel through an extra demux level. Even though Click is a well developed piece of software and many people are using it, Netfilter overtops in both criterions.

2.2 Final Architecture

Using Netfilter as demultiplexer, any combination of EEs may be running on the active node without depending of a special one which also provides the demulti-

[1]To be found at *http://www.advogato.org/proj/ipqmpd/*

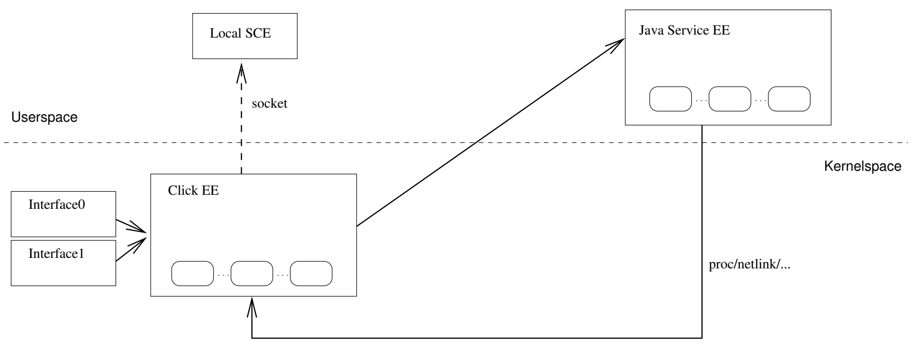


Figure 2.3: Architecture proposal without Netfilter

plexing. That way the active node may run with or without Promethos and Click. Figure 2.4 shows how the various EEs are integrated symmetrically.

Due to Netfilters modular design, it is a simple task to extend the demultiplexer for newly added EEs. Demultiplexing is done by means of a special Iptables target for each EE.

2.3 Inter-EE communication

In order to combine different EEs in a single active node and allow inter-EE communication, inter-EE adapter elements must be implemented. They contain the intelligence how to transfer data to or read data from other EEs. The configurators must insert these adapter elements into the configuration in order to establish interEE communication. Refer to section 4.2.2 for details about how this is done between our Java EE and Click.

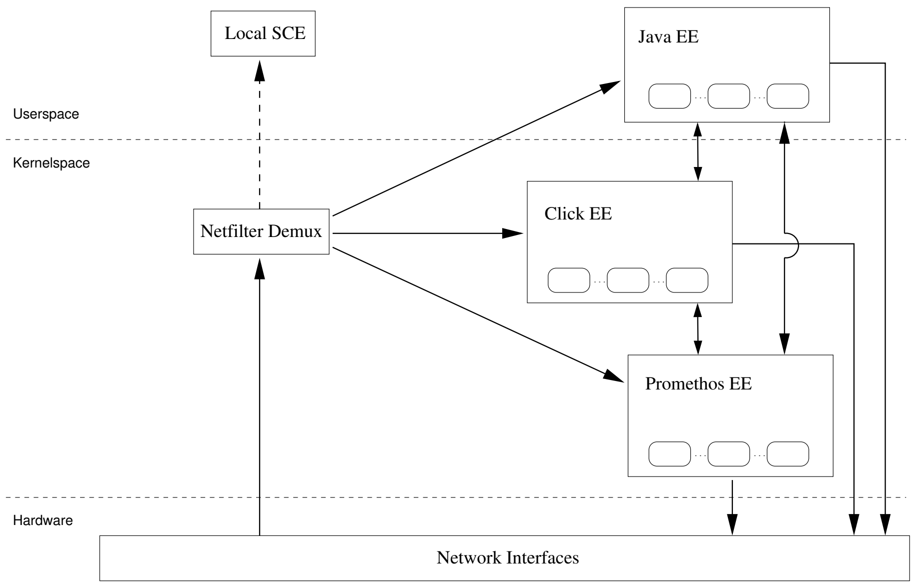


Figure 2.4: Final architecture

Chapter 3

Service Creation Engine

In this chapter the service creation engine (SCE) is described. Its task is to map a service request which is node-independent to the concrete node on which the SCE is running. Section 3.1 describes the order of events in terms of seven phases (I to VII). Section 3.2 is concerned about the installation of the service and is structured in terms of the involved components.

3.1 Building the Service Tree

Picture 3.1 shows an overview of the SCE components. As can easily be seen a centralized approach has been chosen. The *specification processor* represents the center of the framework. This means that it contains the whole intelligence about the processing of an incoming packet. The other components act as various services for the specification processor. More details can be taken from the following sections. All procedures are explained in the order of events which is more comprehensible than pure concept-oriented considerations.

3.1.1 Service Tree: General Introduction

In order to understand the succeeding sections the the concept of *service trees* has to be introduced first. The basic idea is to resolve a simple incoming service request into a more extensive service tree, which represents all possible implementations of a service as a whole. An example of a simple service tree is depicted in figure 3.2.

The two different node-types appearing are called *specifications* and *implementations*. A specification is a node which describes the relations of a set of *subservices*. Consequently specifications are just descriptions which have to be resolved, whereas implementations are final nodes and thus always leaves of the service tree. There are properties which both specifications and implementations have in common. Examples for these are: service name, provider and ports. Other properties are specific for specifications or implementations. For example specifications always contain a description of their subservices and some information on how they are

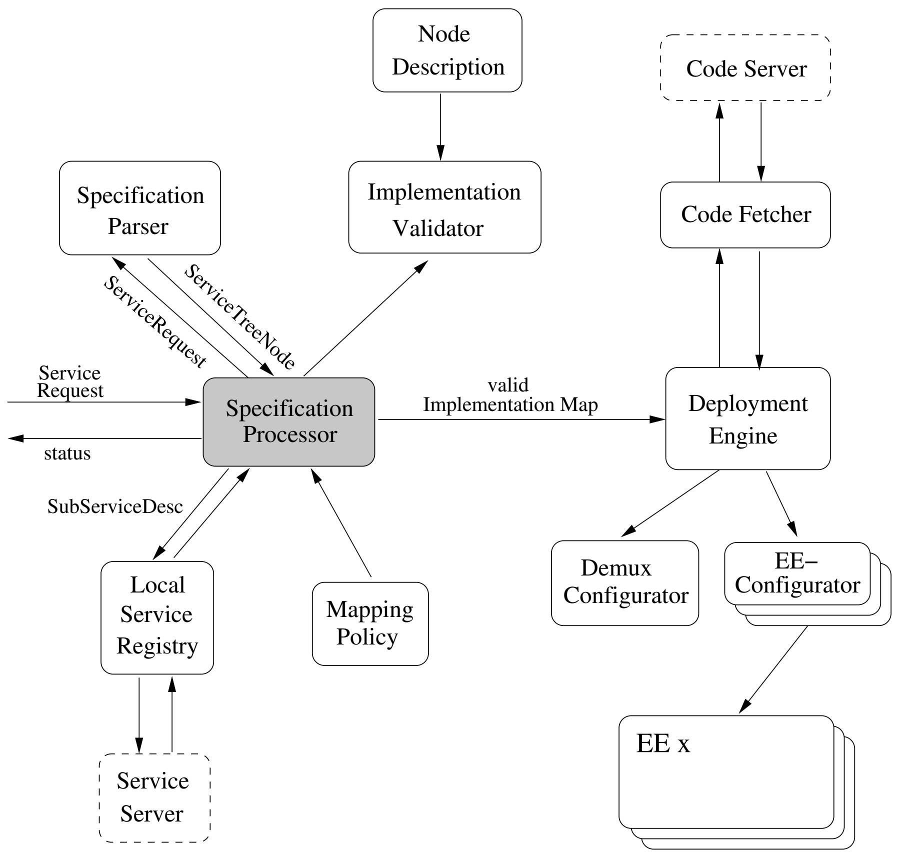


Figure 3.1: SCE overview

connected with each other. In contrast, implementations must provide declarations about the OS and execution environment they run in.

For resolving a service request, first the root node which always has to be a specification (see 3.1.2) must be resolved into its subservices. Referring to figure 3.2, the root node *SW Dictionary* is resolved into the subservices *Packet Processor*, *Dictionary* and *Queue*. For each of these there may be one or more nodes representing the according subservice. In the example the *Packet Processor* is available in two different versions. One is an implementation, the other one a specification node. In the next step those specifications are resolved which have been added to the tree in the former step. Doing this the service tree of figure 3.2 is transformed into the one shown in figure 3.3.

This procedure continues until all specifications are resolved and thus each branch ends up in an implementation as a leaf. Regarding the service tree in figure 3.3 there are two possible ways of deploying the service *SW Dictionary*. The primary consists of the implementations *Packet Processor*, *Dictionary* and *Queue*.

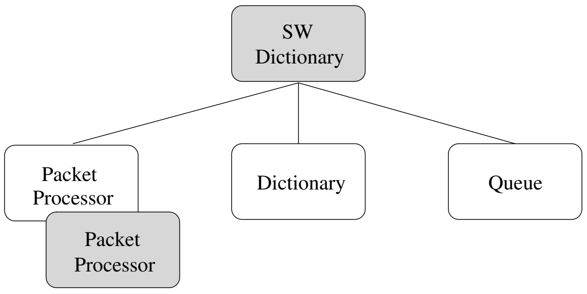


Figure 3.2: Example of a service tree

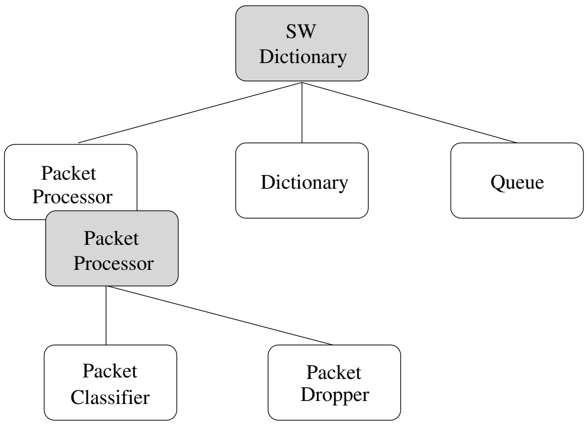


Figure 3.3: Resolved service tree

The secondary uses the same ones, with the difference that the *Packet Processor* is replaced by the *Packet Classifier* and the *Packet Dropper*. By this means a requested service can be implemented in possibly many ways, so the service creation engine has to select one for the deployment.

3.1.2 Phase I: Building a Service Tree

All activities of the service creation engine are initiated by an incoming *service request*. In earlier papers and theses, the *TIK Laboratory* has proposed to implement these requests using the *XML-Format*. We have decided to adopt this proposition mainly for two reasons: firstly each XML-file represents a logical tree structure which is a suitable approach for structuring the various declarations of a service request. Secondly XML is presently a widespread format so that the general acceptance and the availability of appropriate Java-APIs are ensured.

More details and some examples about the usage of these XML-requests are included in the User Guide, section B.3.

The incoming service request is either a specification or a *simple request* which can be resolved into a specification. Therefore the root node must always be a specification. The SCE does *not* allow implementations as root nodes.

Beginning the processing of a request the specification processor forwards the XML data to the *specification parser* without modifying it in any way (see figure 3.2). It is the task of the parser to read the incoming request, parse it and commute the resulting information into a Java-internal representation of *service tree nodes* being managed by a *service tree*. The specification processor is now able to read information from this node and thus begin resolving it. For this purpose the first subservice described in the root specification is read out and forwarded to the *local service registry*, which is responsible for fetching some XML data describing the requested subservice. Again, the incoming XML data is directly forwarded to the specification parser which repeats the same procedure as seen above. However, there is one difference: in the current case the XML data might include information about *more than one node*. This comes from the fact that a subservice might be deployed in various different ways. For example, the *Packet Processor* in figure 3.2 might be regarded. On all accounts, the parsed service tree nodes are added to the service tree as *children* or *children nodes*[1] of the root specification.

This procedure is done for each subservice until the service tree is expanded to the new *depth* of 1. At this new level each specification can be regarded as if it was the root specification and thus be resolved as described above. Implementations are left unmodified because they represent the leaves of the service tree. Therefore, the service tree is completed if all branches end up in an implementation.

There are two more concepts which need to be focussed within the scope of building the service tree. They are described in the following two subsections.

3.1.2.1 Parameters

Any node or service in the service tree may be configured by some parameters. For example, a service called *Filereader* may need parameters such as a *filename* or a *number of bytes* to be read. For this purpose we have introduced *two kinds of parameters* which can be specified in the XML description of each service:

Expected Parameters only have a name without a value set in the considered node. They expect the parent specification to set the value. For a successful deployment it is necessary that a specification passes the right number of expected parameters to each of its children nodes. In order to be able to do so, each service has a specific number of expected parameters which must be *"well-known"*. If there is more than one expected parameter which is passed from parent to child, the assignment is performed *order-sensitive*.

[1] In the following the terms *subservice*, *child* and *child node* are used exchangeably

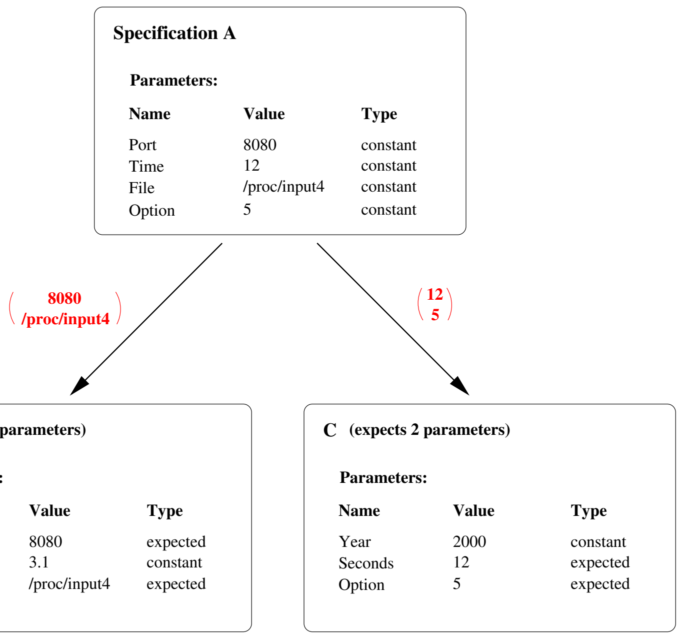


Figure 3.4: Concept of parameter passing

Constant Parameters have both name and value set in the considered node. The parent specification does not need to know about those parameters.

Specifications are allowed to pass both expected and constant parameters to their children nodes. For better illustration of this concept, figure 3.4 shows an example setting. For the deployment engine, only the *order* of the parameters is relevant. It has no knowledge about where they came from and how their values were set. This results in that the whole concept of parameter passing is only an issue of the processings of the specification processor. A second consequence is the fact that the *names* are not of global relevance but *only valid within the scope of the considered node*. An expected parameter can have different names in the parent and the child node. This is important because different subservices may use parameters with the same name but different meanings.

3.1.2.2 Demultiplexing Rules

Similar to parameters *demultiplexing rules* are attributes of services. They address the issue of which packets are or are not destined for this service. In order to be able to make this decision, the deployment engine must configure the demultiplexer and therefor needs some rules. In contrast to the *parameters* described in section 3.1.2.1, the Service Creation Engine can handle them in a simpler way: wherever demultiplexing rules appear in a service they are to be passed to all its children and children's children (for further details see section 4.1.2, page 37). As a result they stay in the nodes as one of various attributes and can be handled by the deployment engine.

3.1.3 Phase II: Node Validation

Though 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 service creation engine to do extensive checks and validation steps to ensure that the service is deployable on this node. According to this concept the deployment engine must only *deploy* a service and is not concerned about whether it is deployable at all.

The first and most simple check is the *node validation*. In the context of this step all implementations have to be tested whether they can be installed and run on this node. Specifications need not be examined because they only serve as construction plan.

The node validation includes checks for the following attributes:

- OS name
- OS version
- EE name
- EE version
- Types of the in- and outports

For doing so the specification processor recursively goes through the service tree and passes all implementations to the implementation validator (see figure 3.1) . This component is responsible for checking whether the attributes belonging to the passed implementation are supported on this node. As can be seen in figure 3.1 the implementation validator consults the node description for performing the tests itemized above. As return the specification processor receives the simple decision whether the passed implementation can be deployed or not. If not, the service tree node concerned is marked as *invalid*. At the same time the specification processor must verify whether the parent specification of the invalid node is still deployable. It is undeployable if the implementation concerned is the only available variant for a particular subservice. In this case, the considered specification also has to be flagged as invalid.

This recursive check for deployability goes back until the root node is reached. At that time, the decision can be taken whether the service *as a whole* can still be deployed. If negativ, the service requester gets a short message informing him about the failure. If positiv, the processing of the requested service can go on.

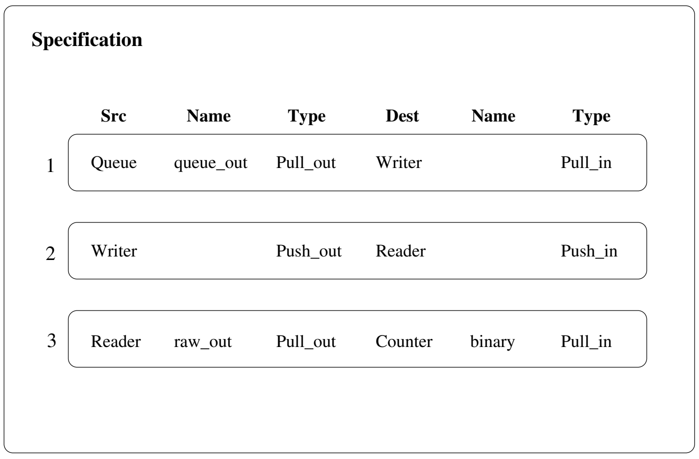
As a last step, provided the service is still deployable, the invalid nodes have to be removed. This is performed by a simple algorithm which recursively goes through the whole service tree and removes all nodes concerned. Removing means that the specifications lose their links to the invalid children.[2]

The removing algorithm does *not* perform any checks for deployability.

3.1.4 Phase III: Port Validation and Routing

At present, the nodes have been validated separately, leaving aside any interactions between them. In this section the next step called *port validation* is described. It is the first of two validation steps concerning the connections between implementation nodes. The second one is called *connection validation* and is illustrated in section 3.1.6.

3.1.4.1 Subservices



Specification

	Src	Name	Type	Dest	Name	Type
1	Queue	queue_out	Pull_out	Writer		Pull_in
2	Writer		Push_out	Reader		Push_in
3	Reader	raw_out	Pull_out	Counter	binary	Pull_in

Figure 3.5: Connection descriptors in a specification

The objective of the port validation is to check whether the service tree is consistent in respect of ports. The node validation described above (see 3.1.3) has validated that all port types of an implementation node exist within the scope of its execution environment. This time another problem is focussed: each specification node defines connections between its children nodes. This implies predications about *number, names and types* of the subservices' ports. The port validation is responsible for proving these predications.

See figure 3.5 for an example of such connection definitions. The regarded specification node contains a list of *connection descriptors*. Each of them specifies a *data path connection* or *transport connection* between two children nodes. Such

[2]In Java it is not possible to free the memory of those "removed" nodes.

connections always lead from an *outport of the source node* to an *inport of the destination node*. These ports are specified by their name (optional) and type (mandatory). Therefore two matching ports have to be found for each connection descriptor. If all these assignments can be fulfilled all children nodes of this specification can be connected in the specified way concluding in the validation of the ports.

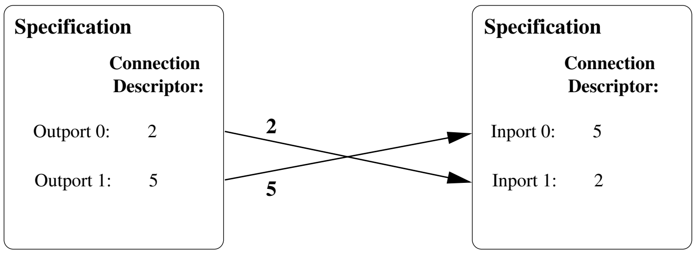


Figure 3.6: Storage of routing information

However, as can easily be seen, more than that has been achieved: beside the port validation, the problem of *routing* is solved as well. The assignment from a connection descriptor to an out- and an inport also means that those two ports have been connected to each other via the transport connection. It is for this reason that the assignment

$$\texttt{outport index} \rightarrow \texttt{transport connection} \rightarrow \texttt{inport index}$$

is not only theoretically proven but also stored for later use. Figure 3.6 illustrates how this routing information is filed in the nodes.

The algorithm performing the assignment from a connection descriptor to the ports of the child node is the following:

1. Assign all ports with names.
2. Assign ports after type. If there is more than one take the first one.
3. For the remaing ports check whether their type is declared *optional*.

Firstly, it is important to know that in- and outports are always treated separately. Therefore an inport and an outport can coexist although they have the same name.

Assigning ports by name or type always means that the according designation must match perfectly without any exceptions. Names have been introduced in case a node has more than one port with the same type but with a different function. Therefore the names must be globally “well-known” and serve as unique identifier.

In case a node has more than one port with the same type and all of them having the same function, the assignment is done based on the order of appearance (see second instruction).

If a child node has ports remaining unassigned after the second instruction it has to be checked whether their type is declared "*optional*". This feature is used in case there are ports which can remain unconnected, as for example `procfs` ports in Click elements. These are ports that typically stay unconnected. The check can be performed by consulting the implementation validator as described in section 3.1.3.

In order that the above algorithm can work, a set of conditions must be applied for port names and types. Those for the port names are the following:

- Ports of specification or implementation nodes can have a name but it is not mandatory.
- A connection descriptor of a specification does not have to specify a name for source and destination port. Even if the matching port of the child node does have a name.

For port types the following rules obtain:

- Each port of a specification or implementation node must have a type.
- Each connection descriptor of a specification must specify the type of both the source port and the destination port.

3.1.4.2 Specifications

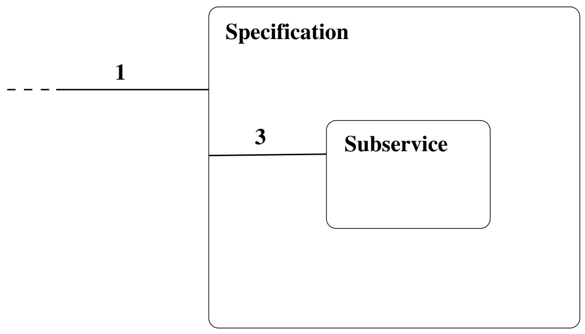


Figure 3.7: Routing problem with specifications

Until now one problem has been left aside: children nodes of specifications may be specifications again. This has consequences on the task of routing. For illustration of this problem see figure 3.7.

Specifications can be regarded as containers and its subservices as nodes within them. Therefore transport connections can be *inside* or *outside* of a specification. The challenge of routing now consists of assigning an outside connection to the correct inside connection. In the example of figure 3.7, the outside connection 1 has to be assigned to the inside connection 3. In order to ensure the uniqueness of this assignment *global identifiers* or *IDs* had to be introduced for transport connections. Consequently the routing problem in respect of specifications is solved by the assignment

`ID of outside connection` → `port index` → `ID of inside connection`.

As described in 3.1.4.1 outside connections are described in the parent specification. In contrast, inside connections are specified by connection descriptors of the considered specification node *itself*. Therefore, the step of port validation including assignment of connection descriptors to ports, has to be done not only for all children nodes, but also for the specification node itself.

The same procedure of removing invalid nodes as described in section 3.1.3 is followed after the port validation. Now the remaining nodes in the service tree are not only valid, but also contain all information needed for routing.

3.1.5 Phase IV: Building a Validation Map

After all nodes and ports have been validated and the routing information is available in all implementation nodes, there is no need for the service tree anymore. For the deployment engine does not expect any kind of tree from the specification processor but solely an *implementation map*. This is a linked list of implementations. Thus, the "vertical structure" of the service tree must be converted into a "horizontal" one. It is because some more checks have to be done that the implementation map cannot be created yet. Therefore we were confronted with the task to design such a "horizontal" structure called *validation map* which is suitable to the processing steps still to be done.

This section describes how the validation map has been designed and how it is created by the service creation engine.

3.1.5.1 Design of Validation Nodes

Figure 3.8 illustrates that each port of an implementation can be connected to possibly various different succeeding (for outports) or preceding (for inports) implementations. This results from the fact that there may be several possibilities for each subservice of a specification (see 3.1.1).

Designing the validation map can be reduced to the task of finding a suitable structure for the individual nodes. These so called *validation nodes* are the elementary nodes of the validation map. We found two possible approaches:

1. A new structure is introduced.
2. The existing implementations are used.

We have decided in favor of the second approach for one major reason: introducing a new structure would it make necessary to copy various data structures from

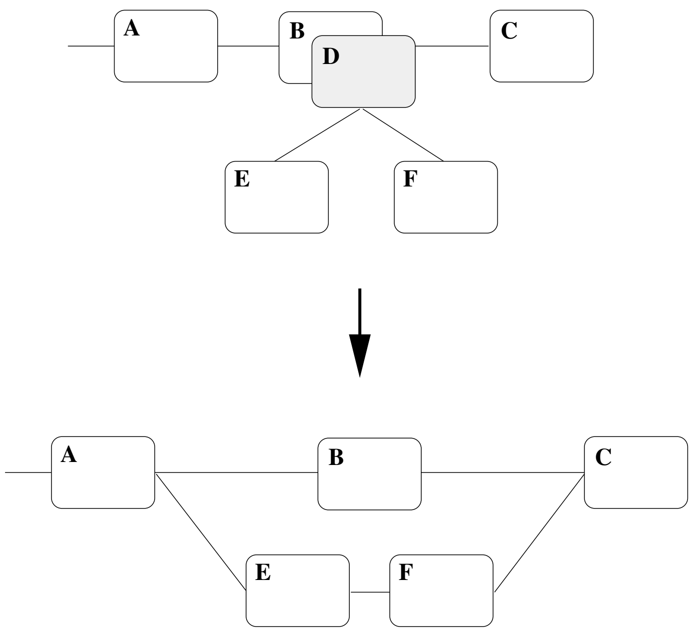


Figure 3.8: An implementation may have several successors

the implementations to the new nodes. Furthermore, it would always have been necessary to provide a static assignment from implementations to those nodes. Last but not least an expensive retransformation into implementation nodes would have been inevitable for building the final implementation map.

Choosing the variant of using the existing implementations includes that they have to be extended by some data structures. The main part of this additions is responsible for the connection management to succeeding nodes which are also called *successors*. Some smaller additions include a few flags indicating different states. But these are not relevant for our conceptual considerations.

3.1.5.2 Introduction to Algorithms Iterating the Validation Map

The whole service structured in such a "horizontal" way is called *validation map*. There are various processing steps in the following sections which need to go through the map. Thus, all those algorithms have been implemented with the same iteration logic which is described in this section. The *iteration logic* means the control path of the algorithm whereas the *processing logic* represents the data path which is the issue of each algorithms' section.

Figure 3.9 shows an example of a validation map. The root node at the left is no node in terms of specifications or implementations. It is a logical component of the validation map which has been introduced as interface for the access to the

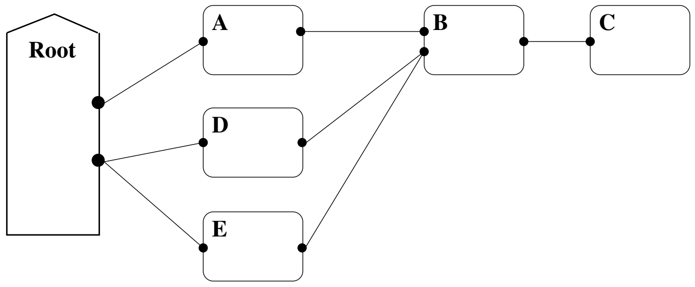


Figure 3.9: Validation Map

nodes. For this reason it is also called *root interface*. All algorithms iterating the validation map are initiated in the root interface.

It is important to be aware that the nodes `A`, `B` and `C` are the only possibilities for their service. In contrast, node `D` and `E` are two possibilities of the same service. This can be recognized by the fact that the latter ones come from the *same "outport"* of the root node and lead to the *same inport* of node `B`. The outports of the root node represent access points to the validation map. Section 3.1.5.5 and figure 3.16 explain these access points in more detail.

All algorithms iterating the validation map consist of the same structure which is outlined in the following:

```
1 Go through all outports i
2 Go through all possibilities j of outport i
3 Trigger the algorithm in the successor 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
```

For better illustration, the subsequent procedure is given, which is the result of applying the above algorithm to the example map in figure 3.9. *"Triggering a node"* means that the algorithm is activated in the appropriate node. *"Executing the logic"* concerns the processing logic explained above.

```
Root triggers A
A triggers B
B triggers C
C executes the logic and returns to B
B executes the logic and returns to A
A executes the logic and returns to Root
Root triggers D
D triggers B
```

```
B triggers C
C executes the logic and returns to B
B executes the logic and returns to D
D executes the logic and returns to Root
Root triggers E
etc.
```

It depends on the concrete algorithm whether the root node executes the processing logic at the end or not. Also it is the concern of each particular algorithm how to handle the multiple triggering of nodes with more than one preceder (nodes B and C in the example).

3.1.5.3 Expanding Validation Nodes

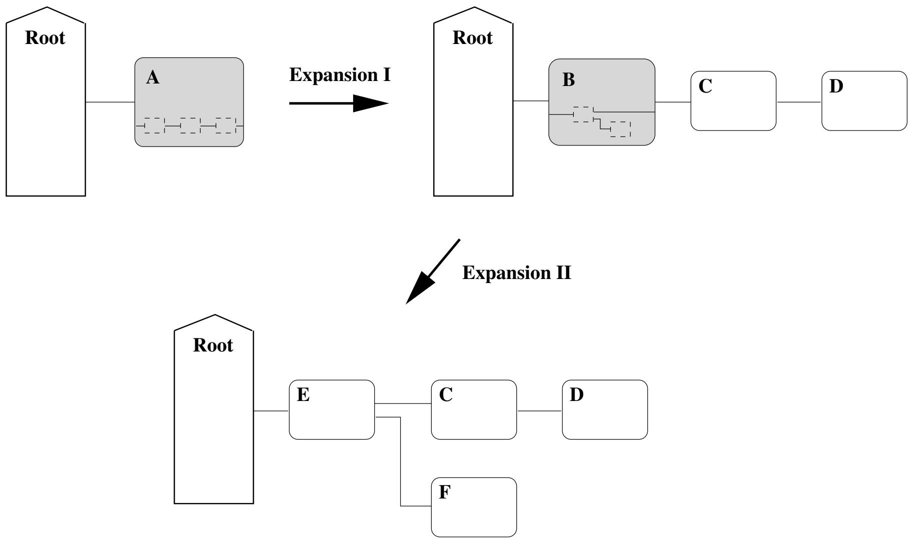


Figure 3.10: Building a Validation Map

Building the validation map turned out to be a challenging task. Remember that all routing information is basically available in the nodes. Our first idea was to build the validation map by one recursive algorithm. However, doing so we underestimated the complexity of routing nodes which are nested. Therefore we decided to choose an iterative approach. Regard figure 3.10 for the following explanations of this algorithm.

The basic idea is to *expand* all specifications step by step. Thus in the beginning the validation map consists of nothing but the root specification. In the first step this root node is expanded, which means that it is replaced by all its children nodes. Those need to be routed correctly for this purpose. The *expand algorithm* in turn expands all specifications of the resulting validation map. This algorithm is repeated until all specification nodes are expanded leaving the validation map to consist of nothing but connected implementations.

The expand algorithm is implemented as described in section 3.1.5.2 and therefore contains logic for iterating the validation map. In the following its actual processing logic is outlined:

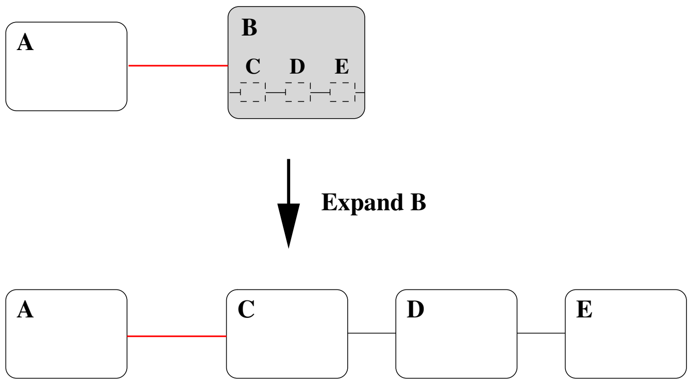


Figure 3.11: Updating connections to successors

1. Specifications have to be expanded which implies:
   - Update connnections to successors if they have been expanded.
   - Connect children nodes with each other.
   - Border nodes at the entry: update connection ID belonging to the appropriate inport.
   - Border nodes at the exit: hand connections to successors of this specification down to border nodes (inheritance) and update the connection ID belonging to the appropriate outport.
2. Implementations update their connections to successors if they have been expanded.

Apparently only *forward-leading* connection logic is stored. There is no need for iterating the validation map backward.

Both specifications and implementations have to update their connections to successors if those have been expanded. This situation is shown in figure 3.11. The red connection going out from node `A` has to be redirected from destination node `B` to `C`, provided the node `B` has been expanded the way shown. Figure 3.12 illustrates the actions to be done in respect of border nodes at the entry of specification nodes. The connection ID which node `A` has assigned to its only inport must be updated

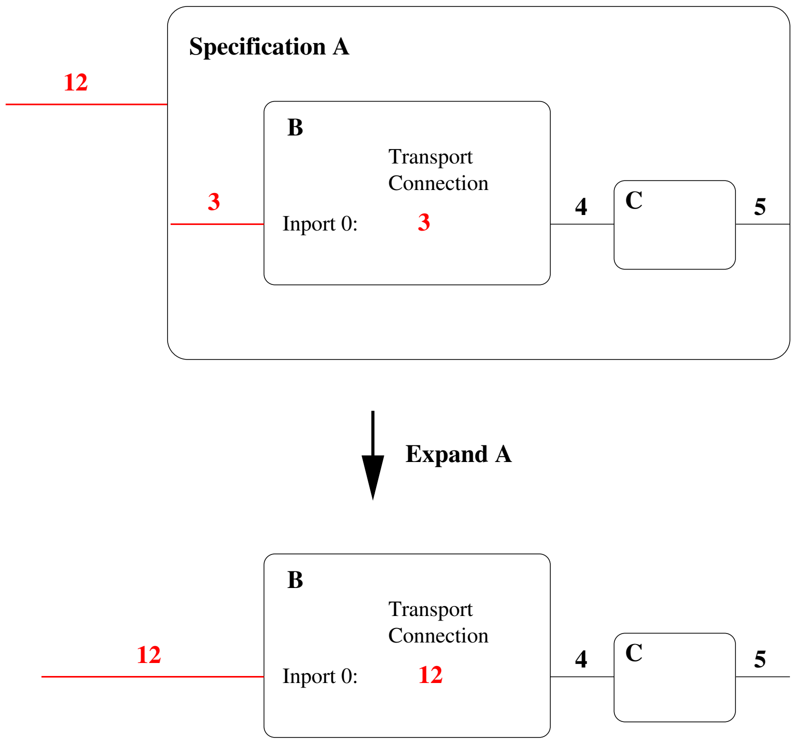


Figure 3.12: Actions on border nodes at the entry while expanding

from 3 to 12 because 3 was only the inside connection ID of the specification `A` used as temporary routing information (see 3.1.4.1 and 3.1.4.2).

The last situation to be considered is that of border nodes at the exit of specification nodes. It is shown in figure 3.13. Two actions have to be done: the first one is to provide that the border node `C` "inherits" the connection to node `D`. Similar to the situation of figure 3.12 the correct successor has to be found via the mapping of the inside connection ID 5 to the outside connection ID 12. The second action is to update node `C`'s assignment from its outport to the outgoing connection ID which is newly 12 instead of 5.

In section 3.1.5.2 at page 21 the problem has been mentioned that the algorithm is triggered multiple times in nodes with more than one preceder (no matter if it has one or more inports). This is shown in figure 3.14. The specification `C` is tried to be expanded by both nodes `A` and `B`. While the expansion can be performed as usual the first time (e.g. coming from node `A`) the second try (e.g. coming from node `B`) has to be treated in a special way:

- Node `C` does not have to be expanded once more. In order to inhibit a second expansion node `C` must memorize that it has already been expanded.

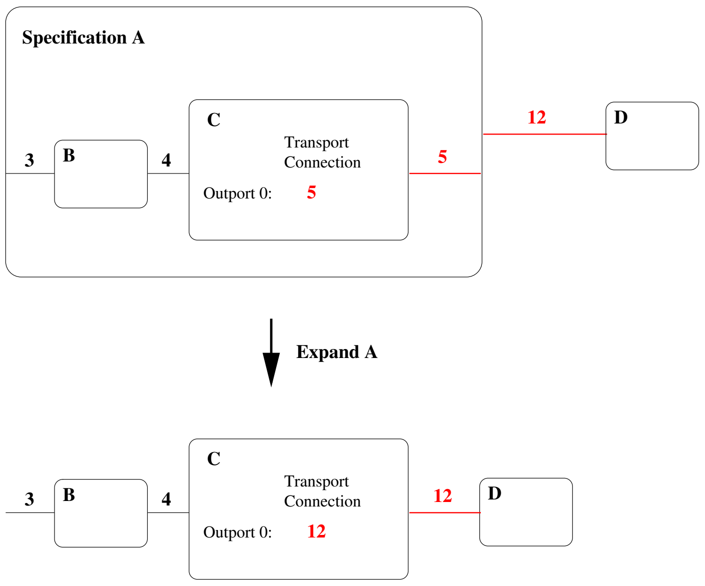


Figure 3.13: Actions on border nodes at the exit while expanding

- Node `B` nevertheless needs to get its new successors.

3.1.5.4 Entry and Exit Points

There is an issue which has been left aside so far: the deployment engine needs to know the entry and exit points of the whole service. An entry point is an inport which is not connected with an ordinary node. Instead it builds an entry to the whole service to be deployed. Exit points are outports with the analogical property at the exit of the service. Each service may have several entry or exit points. In the example shown in figure 3.15 the deployment engine would have to know that there

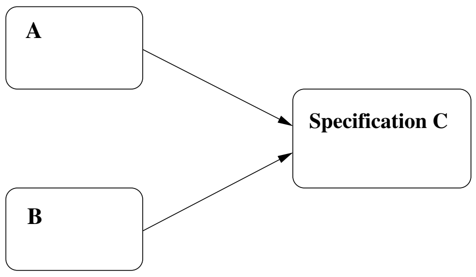


Figure 3.14: Special situation for the expand algorithm

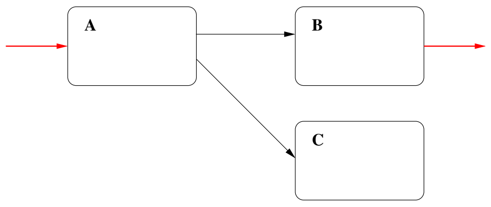


Figure 3.15: Entry and exit points

is an entry point at node `A` and an exit point at node `B` but none at node `C`.

This information arises from the root specification and is handed down to its children and children‘s children during the expand algorithm described above (see section 3.1.5.3).

3.1.5.5 “Floating” Nodes

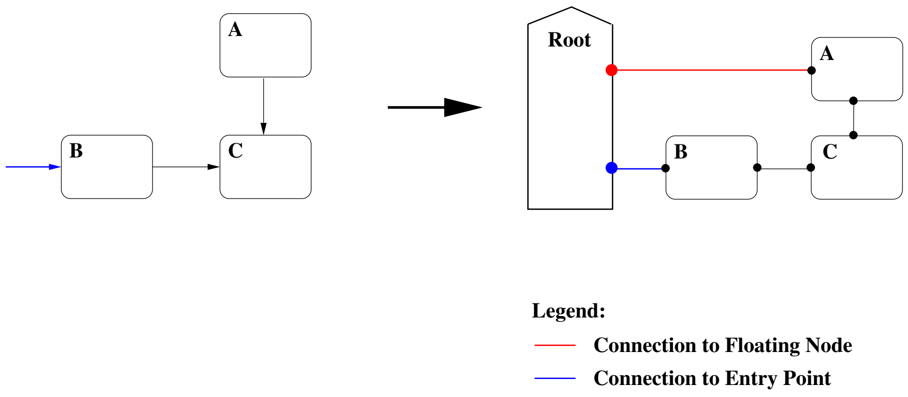


Figure 3.16: Access to floating nodes in the validation map

Nodes without inports or none of them connected to anything, are called *“floating” nodes*. Their only link to the other service components is via one or more connected outports. They need a special treatment because the validation map only contains forward-leading connections and thus they are omitted by iterating algorithms.

During the port validation (see section 3.1.4) floating nodes are recognized as subservices of the root specification. At that time they are registered in a dedicated data structure of the root specification. By this means they are detected when the root specification is expanded building the validation map. In order to get access to them, the root node of the validation map treats them like entry points. This means that an “outport” of the root interface is inserted and connected to the floating node (see also section 3.1.5.2, page 20). Therefore each “outport” of the root interface leads either to an entry point or a floating node. Refer to figure 3.16 for an illustration.

This special treatment of floating nodes has hardly any consequence for the algorithms iterating the validation map. The only algorithm which has been affected is the expand algorithm because floating nodes need to be connected to an “outport” of the root interface. For simplicity a special *expand algorithm for entry nodes* has been introduced. However, the differences to the ordinary expand algorithm are minimal.

3.1.6 Phase V: Connection Validation

One of the reasons why a validation map has been built, is the *connection validation*. Its objective is to remove all transport connections between implementations which cannot be deployed on this system.

Therefor, the implementation validator serves as an interface to the node description as seen e.g. in section 3.1.4.1. For this connection check the following parameters are needed of both the source and the destination implementation:

- EE name
- Types of the involved ports

Remember there are no longer specification nodes. The algorithm performing the connection validation again iterates the validation map as described in section 3.1.5.2. Its processing logic can be outlined like this:

- If a successor turns out to be invalid the connection leading to it is removed. A node is invalid if at least one connection going out from it is invalid and it has no more connections left for the affected outport.
- Validate all connections going out from the current node to its successors and remove all invalid ones.
- Check whether this node is still valid and memorize the result. The definition for invalid nodes is given in the first item.
- To the preceding node (where the iteration came from) the information is returned, whether this node is valid or not.

The problem of multiple executions of an algorithm (addressed in section 3.1.5.2) must also be solved here. It can be handled by memorizing the flag saying whether a node is valid or not (see item 3). Thus, it can be returned to each preceding node and it is not necessary to repeatedly continue the iteration through the validation map.

After the algorithm has finished there is no need to remove invalid nodes because all connections to invalid nodes have already been removed (see item 1 and 2).

3.1.7 Phase VI: Evaluating Routes

At this point all validations have been completed. The list of connected implementation nodes potentially yields a great number of possibilities how the requested service may be installed. It is the last task of the specification processor to select one of these and pass it to the deployment engine. In the following three sections this evaluation and selection is discussed.

3.1.7.1 The Idea

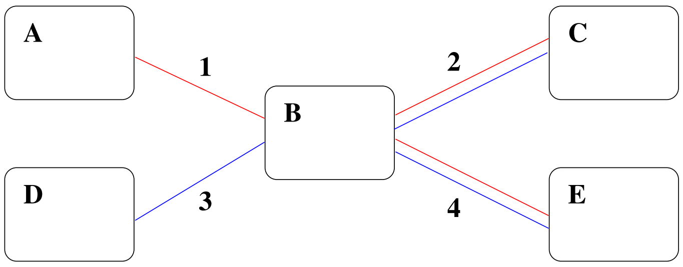


Figure 3.17: Routes of a validation map

The various possibilities of service installations are represented by the different available *routes* leading through the validation map. Regard figure 3.17 for an example. All red connections build one route, the blue ones build another. Through this the definition of a "route" can be recognized: It is the summary of all involved connections of one possible service installation. In the example of figure 3.17 the red route consists of the connections `1`, `2` and `4`. The blue one is built by the connections `3`, `2` and `4`. This can be written as:

```
Red route:  {AB, BC, BE}
Blue route: {DB, BC, BE}
```

Obviously the different routes can contain partially the same connections (`BC` and `BE` in the example).

The basic idea is now to first compute all routes contained in the validation map and store them. By this means it becomes possible to do the route evaluation "offline". The evaluation algorithm does not need to iterate the validation map on its own. Thus, if a new mapping policy is added (discussion of mapping policies in section 3.1.7.3) the new algorithm only has to implement an appropriate operation on this *route list*. Control path and data path are separated in an elegant way.

3.1.7.2 Computing All Routes

The algorithm computing all routes is the third and last algorithm which uses the iteration logic described in section 3.1.5.2. Its funcionality is described for node `A` in the example depicted in figure 3.18. As first step node `A` gets one route list

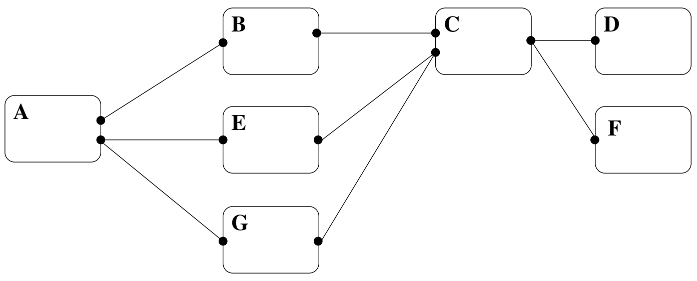
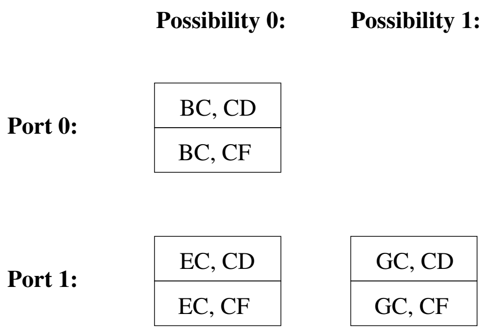


Figure 3.18: Example situation for computing routes

from each successor (node `B`, `E` and `G`). These route lists contain all possible routes "behind" the appropriate node. Figure 3.19 shows how they are sorted by outports and possible successor.



	Possibility 0:	**Possibility 1:**
Port 0:	BC, CD BC, CF	
Port 1:	EC, CD EC, CF	GC, CD GC, CF

Figure 3.19: Raw route lists

There are two processing steps necessary: *Merging* and *Combining*. The first step means that the different lists belonging to one outport are merged into one. The second in turn combines the routes of the resulting lists in a way that *one* more list remains at the end. See figures 3.20 and 3.21 for the list states after the first and the second step.

Note to figure 3.20: during the merging process the connection from the current node to the appropriate successor is also added.

Note to figure 3.21: the combine algorithm has to ensure that connections are omitted which are behind a node whose route list has already been added. In the

Port 0:	AB, BC, CD
	AB, BC, CF
Port 1:	AE, EC, CD
	AE, EC, CF
	AG, GC, CD
	AG, GC, CF

Figure 3.20: Route lists after merging

Node A:	AB, BC, CD, AE, EC
	AB, BC, CD, AG, GC
	AB, BC, CF, AE, EC
	AB, BC, CF, AG, GC

Figure 3.21: Route list after combining

example this problem occurs with node `C`. Like in the connection validation algorithm all nodes can memorize their route list once computed in order to inhibit redundant multiple executions.

Now the route list of the current node is complete and can be returned to the preceding node which in turn does the same. Note that the merging and combining procedure has also to be done in the root node of the validation map. After this the route list of the *whole service* is complete.

3.1.7.3 Selecting One Route

The route list of the whole service can now be evaluated. In order to select one out of the possible routes a *mapping policy* is needed. This policy contains the criterions based upon which the selection is done. The mapping policy component of figure 3.1) contains the currently configured policy. In the scope of our thesis we have implemented one mapping policy. It is called "*Minimize EE transitions*" and analyzes the available routes in respect of EE transitions. It selects the route which implies the least number of EE transitions. If more than one is found the first one is taken.

The data structures involved in the route selection are conceived in such a way that other mapping policies can be added easily. Note that the SCE's component "mapping policy" (see figure 3.1) does not contain the actual logic of a policy which iterates and evaluates the route list. This component is solely in charge of storing the currently configured policy in an encoded way.

3.1.8 Phase VII: Translation into an Implementation Map

In section 3.1.5 it has been mentioned that the deployment engine expects an implementation map from the specification processor. Therefore, after having done all necessary validation steps and having selected one possibility of implementing the requested service, the according route has to be translated into an implementation map.

As seen in section 3.1.5.1 the validation map consists of nothing but linked implementation nodes. And so does the implementation map. Therefore the translation from the selected route into an implementation map mainly implies 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 true for the implementation map.

Once the translation has been completed the valid implementation map is passed to the deployment engine.

3.2 Installing the Service

Now that the requested service has been expanded according to our active node capabilities and is validated, it can be installed.

3.2.1 Deployment Engine

The deployment engine gets an implementation map to be configured and is triggered by the specification processor. It calls the demultiplexer configurator. And furthermore it creates and starts the configurators of all EEs addressed in the implementation map.

```
<xsi:EE xsi:name="CLICK" xsi:configurator="click" [...]>
```

Table 3.1: Specification of EE configurator in node description

Each supported EE must specify the name suffix of their configurator class in the XML node description as shown in table 3.1. The deployment engine computes the name of the configurator corresponding to an EE by appending that suffix to the constant string `configurator_`. In addition the package path is prepended. For our two EEs this results in the configurator names:

- ch.ethz.ee.tik.chameleon.sce.Configurator_jee.class
- ch.ethz.ee.tik.chameleon.sce.Configurator_click.class

3.2.2 Demultiplexer Configurator

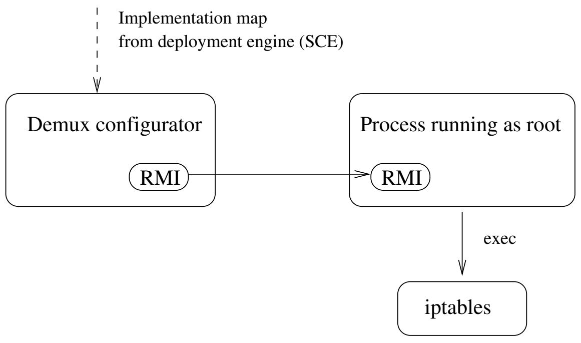


Figure 3.22: Demultiplexer configurator architecture

Figure 3.22 shows the architecture of the demultiplexer configurator based on RMI which is very similar to the Click configurators architecture (see section 4.2.1).

In both issues RMI has been chosen for the same reason: Iptables must run as system user *root*. The main tasks of the demultiplexer configurator are:

- Connect all service entry ports to the EE specific entry adapter element. These are all input ports of services where packet processing of newly arrived packets starts. Therefore the demultiplexer configurator inserts EE specific entry adapters (as specified in the node description) into the implementation map.
- Connect all service exit ports to the host network stack. This results in packets continuing their travel after being processed by the active node services.
- Set the required Iptables rules in order to connect the demultiplexer to the EE entry adapters.
- Set additional Iptables rules to prevent packets from circulating between the demultiplexer and the services after being processed[3].

Until now Click demux entry points must be push_in ports. If not the case, the demux configurator will insert the FromNetfilter element without checking for port semantics. Therefore the Click installer will fail later on. A solution to this problem would be to insert a SimpleQueue element between the FromNetfilter element and the entry port translating from push to pull semantics.

See chapter 4.1 for more details.

3.2.3 EE Configurators

A configurator receives an implementation map and installs those elements which belong to the EE the configurator is in charge of (native EE elements). Special care must be taken for native elements that are connected to elements from other EEs. Configurators must insert special adapter modules for this so called *inter-EE* communication.

As depicted in figure 3.23, this might lead to tricky port semantics. The procfs output port of the click service Counter is connected to the push input port of the JEE service ToFile. The node description must specify this kind of inter-EE connection as valid, since the JEE configurator inserts an adapter which has the accurate port semantics for this situation.

Connecting a Click procfs output port with a JEE pull input port might lead to problems since the JEE inter-EE adapter FromFile has a push output port (see figure 3.24). After insertion of elements by the configurator, the implementation map is not validated again. Therefore a Click procfs output to JEE pull input is not specified in the node description, in order to inhibit such situations.

Configurators always extend their base class *Configurator*. They provide a *configure* method called by the deployment engine.

[3]Packets passing through the demultiplexer are marked as *seen* by means of setting the socket buffers mark value.

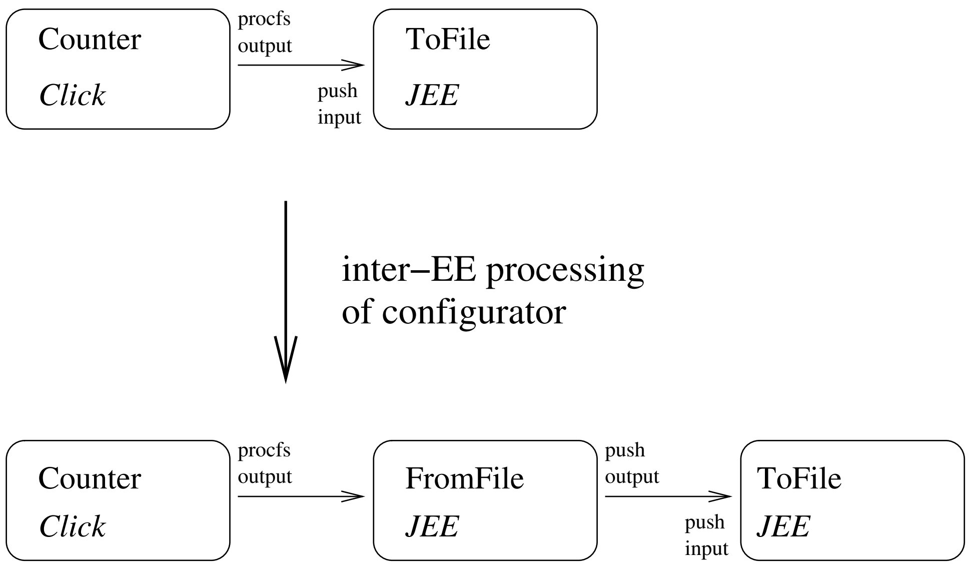


Figure 3.23: Inter-EE situation

Macro	expands to
`$INSTANCES$`	The instance name of the implementation.
`$EE$`	The name of the EE this implementation belongs to.
`$SERVICE$`	The service name of the implementation.

Table 3.2: Parameter expansion macros of configurator

Service parameters may consist of dynamic strings. We implemented a macro expansion for parameters. The *Configurator* base class provides a method *parseMacro* to expand macros in a parameter value string. It supports the macros listed in table 3.2.

However, by calling the *Configurator* base class' method *addMacro*, EE specific configurators may add further macros. This is done for example by the Click configurator for the `CLICKFSPATH` macro.

The EE specific configurators we implemented in our thesis are described in chapter 4.2.

EEs may provide their service's code modules locally or in a distributed manner on the code server. If code modules need to be downloaded from the code server the configurator has to use the code fetcher to do so.

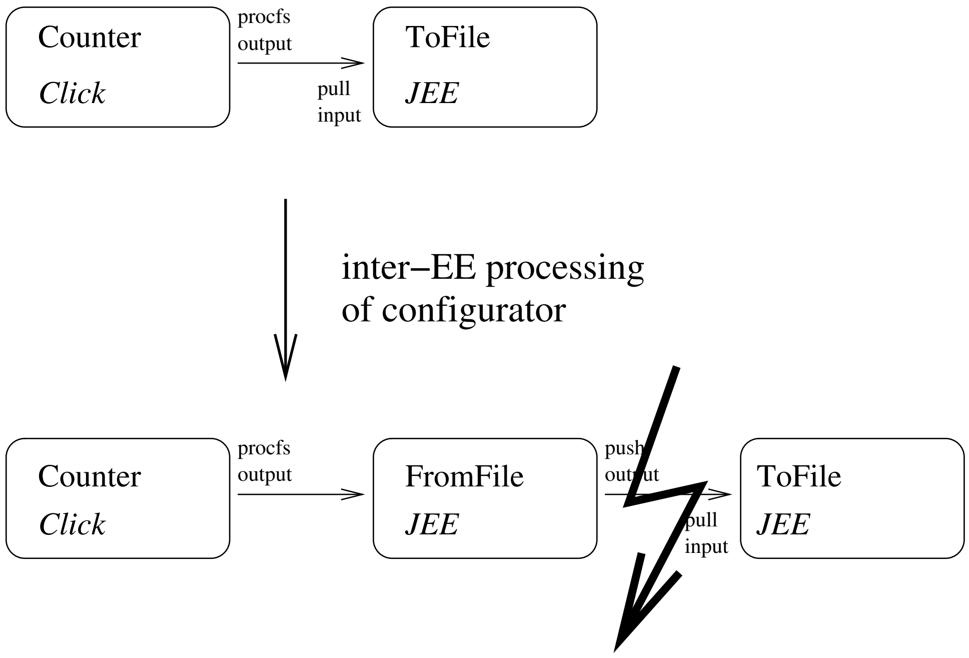


Figure 3.24: not working inter-EE situation

3.2.4 Code Fetcher

The code fetcher is the client application for the code server as depicted in the overview figure 3.1. Several code servers may be configured in the HostConstants class. The public method *fetch* takes an *Implementation* as argument and returns the binary code of the requested implementation after probing all configured code servers. The code module is taken from the first reachable code server providing it. Probing all known code servers has two advantages:

- Code modules may be spread over several code servers by EE, manufacturer or version etc. This may help solve licencing situations where a software manufacturer does not allow alternative download locations[4].
- Redundancy can be provided by copying code modules to different code servers.

[4]E.g. nVidia graphic card drivers for Linux

3.3 Utilities

The code server and the service server, described in this section, are not actually part of the SCE but serve as utilities for distributed service description and code storage management.

3.3.1 Code Server

The code server provides the code fetcher with code modules for the services. It is basically a multi-threaded TCP server. The server waits for requests that consists primarily of a single line containing the path of the code module. The response is plain binary without any status code. If the desired module could not be found, the connection is closed immediately. Thus clients can detect that case if no byte is sent back.

3.3.2 Service Server

The service server provides service descriptions.

It responds to requests from the local service registry, consisting of a single line containing the name of the desired service. The response is a service description list in XML format containing implementations and specifications.

If more than one description is available for the desired service, they are packed in one XML document and sent back. If no service description is available, nothing is returned.

Let me now shortly describe the internal functionality. The server reads all XML files from the configured[5] file system path, storing service descriptions in memory to be passed over to the local service registry when queried.

[5]in the HostConstants java class

Chapter 4

Active Node Platform

This chapter tells about the platform specific parts of the active node.

4.1 Demultiplexer

As already depicted in figure 2.4 demultiplexing of data packets to the running services is done by Netfilter. This offers great opportunities to integrate any userspace and kernelspace EE since Iptables supports passing packets to userspace. Furthermore new Iptables targets can be written easily due to Iptables' modular design.

4.1.1 Configuration

Configuring the demultiplexer is done by the SCE demux configurator. This sets various Netfilter rules in such a way that data packets concerning a certain service are passed to the running service's EE.

All examples in this chapter treat the Click-demux[1] case, since presently it is the only EE which can receive packets from Netfilter.

4.1.2 Architecture

Entry elements need to be connected with the demultiplexer, which is Netfilter/Iptables in our case. Each EE, which should receive packets from the demultiplexer, has to provide an entry adapter element and a demux Iptables target string. Such an adapter element is specified in the node description and is inserted by the demux configurator into the implementation map.

We assume an entry adapter element to be unique within an EE. Each entry element is connected to an output port of that one and only entry adapter element, as shown in figure 4.1. The output ports of the entry adapter elements must correspond with the right demux Iptables rules. This assignment is done by appending an ID to the Iptables demux target.

[1]This means connecting the demultiplexer to services running in the Click EE.

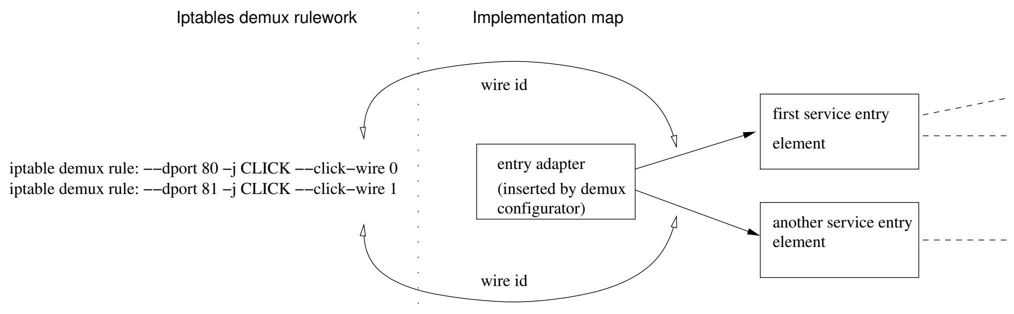

Iptables demux rulework
Implementation map
wire id
iptable demux rule: --dport 80 -j CLICK --click-wire 0
iptable demux rule: --dport 81 -j CLICK --click-wire 1
entry adapter (inserted by demux configurator)
first service entry element
another service entry element
wire id

Figure 4.1: EE entry adapter elements

A service description must communicate which packets it is interrested in by means of demux rules. A demux rule is needed for each entry port of the service. Demux rules are specified in XML service descriptions as option strings conforming to Iptables' syntax, e.g. in table 4.1. Demux rules spread over all three configuration levels (the request, the specifications and the implementations) are cumulated by the specificaton parser for the deployment engine during traversal of the service tree. Local demux rules (in respect of the currently iterated service tree node) always precede demux rules from higher service tree nodes. E.g. demux rules in an implementation will precede demux rules from the request.

```
...
<SERVICE xsi:type="...">
    ...
    <DEMUX_RULE>-p tcp --dport 80</DEMUX_RULE>
    ...
</SERVICE>
...
```

Table 4.1: Demux rule definition in XML node description

However to actually build a valid Iptables rule, the *Iptables target* is required as well. And since we support services with more than one entry port, we need to assign demux rules somehow to ports. This means, we need a target description string in the node description, telling what the EE specific Netfilter target looks like. For Click EE it is:

```
<xsi:EE xsi:name="CLICK" xsi:demux_target="-j CLICK --click-wire ">
```

Assignment of demux rules to entry ports is done with the wire ID. Whenever the demux configurator builds a rule, the current ID is simply appended to the rules target[2]. To complete our example, the resulting rule for a Click service demux rule

[2]That is why the trailing space in the target definition is required.

would be this: `-p tcp --dport 80 -j CLICK --click-wire 0`

Nevertheless it is not yet a fully qualified Iptables rule due to the lack of a table and chain specification. The demultiplexer places its rules into the mangle table for two reasons. First we need to mark packets which can only be done here. Secondly the mangle table has the highest Netfilter hook priority and hence sees packets first of all tables. See appendix C for more details about Netfilter.

```
*mangle
:PREROUTING ACCEPT [335:24525]
:INPUT ACCEPT [324:23405]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [322:27683]
:POSTROUTING ACCEPT [322:27683]
-A PREROUTING -m mark --mark !0x1d -j MARK --set-mark 0x1d
-A PREROUTING -m mark --mark !0x1d -p tcp --dport 80 -j CLICK --click-wire 0
COMMIT
```

Table 4.2: Problematic demux rules

The explanations so far would lead the attentive reader to the assumption that a full running demux configuration would run Iptables rules as shown in table 4.2. But without further counter measures we get stuck in a type of chicken-egg problem. Marking packets just before passing them to Click inhibits the second rule from taking any packets, due to the marking. Not marking packets at all would lead to packet loops between Netfilter, Click and the hosts network stack[3].

```
*mangle
:PREROUTING ACCEPT [335:24525]
:INPUT ACCEPT [324:23405]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [322:27683]
:POSTROUTING ACCEPT [322:27683]
:andemux - [0:0]
-A PREROUTING -m mark --mark !0x1d -j andemux
-A andemux -m mark --mark !0x1d -j MARK --set-mark 0x1d
-A andemux -p tcp --dport 80 -j CLICK --click-wire 0
COMMIT
```

Table 4.3: Final demux rules

The final solution is a user defined chain *andemux* in the *mangle* table as shown in table 4.3. At present packets get into our user chain only if they are not yet

[3]We use the Click element *ToHost* to pass packets back to the network stack. But then Netfilter sees those packets again.

marked. In the user chain they are marked just before being given to Click. If a packet does not match any demux rule in the user chain, it returns to the PREROUTING chain and its fate depends on other Iptables rules and the hosts network stack.

4.1.3 Demultiplexer Implementation for Click

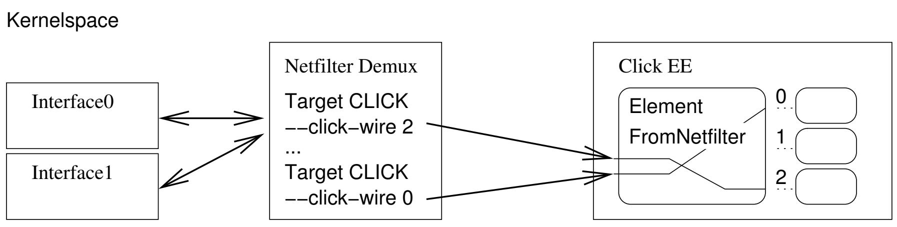


Figure 4.2: Handling packets from demux/Netfilter to Click

In order to pass packets from Netfilter to Click two small software components had to be developed. On the Netfilters side we wrote a new Iptables target `CLICK`. For Click a new element `FromNetfilter` was required which registers itself for packet reception with Netfilter by calling a global function `ipt_click_register`.

For the sake of simplicity only one element `FromNetfilter` may be configured at the same time. Otherwise we would have to multiply packets if several instances of the `FromNetfilter` element are expected to receive packets.

Without any further measures taken, this would mean that we are not able to distinguish which Iptables rule was triggered and all packets handled over to Click would be treated exactly the same way from then on. But by introducing the `wire` parameter to Iptables `CLICK` target, we can demultiplex at the Click side again. Since with the `wire` parameter, which is differently defined for each Iptables rule, the output number of the `FromNetfilter` element at the Click side can be chosen arbitrarily, illustrated by figure 4.2.

4.1.4 Promethos

Since Promethos fits seamlessly into Netfilter, no additional code is necessary to support the kernel EE Promethos. Promethos provides its own set of Iptable targets to receive packets from the Netfilter framework. As described in section 4.1.2 the demux configurator appends a wire ID to each demux rule. This might be undesirable for EEs other than Click. This problem could be solved by introducing a flag in the node description which tells whether an EE needs a wire ID appended. Another solution is proposed in chapter 6 on page 48.

4.2 EE Specific Configurators

The following sections describe the implemented EE specific configurators.

4.2.1 Click Configurator

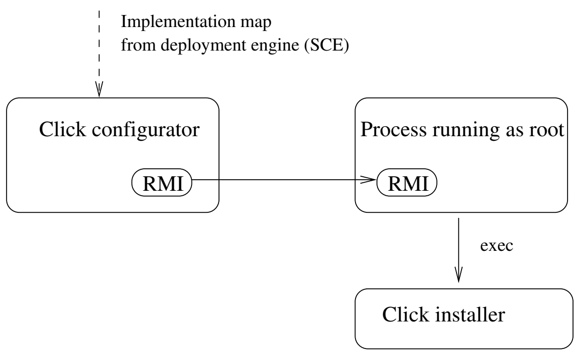


Figure 4.3: Click configurator architecture

The Click configurator parses the implementation map and compiles a Click configuration language file[4]. A Click configuration consists of:

Elements The elements a service consists of are specified with their instance name and service name. The instance name is configured by the user. The service name must be the name of an existing Click element and is specified in the XML service description.

Connections The connections specify which port of the source element is connected to which port of the destination element. Port names, existing in the SCE, have no meaning to Click but they are required for inter-EE communication as described in section 4.2.2.1.

Parameters Click parameters have no name. They are assigned by means of their position[5].

An example is given in table 4.4. The order of service parameters is preserved by the SCE. Comparatively in our example the user gave a service description to the SCE where the first parameter for the classifier element filters TCP/Port 80 traffic and the second filters TCP/Port 81 traffic.

Inter-EE adapters are not (yet) available for the Click EE. To pass network packets from kernel Click to userspace a userspace library, which is able to handle socket

[4]As described in [Click]

[5]Parameter names configured by the SCE are ignored.

```
# Elements syntax: <instance name>::<element name>(<parameter list>);
cl::Classifier("tcp port 80", "tcp port 81");
count1::Counter();
count2::Counter();
giveBack::ToHost("eth0");

# Connection syntax: <instance name>[<output port index>] -> \\
#                    <instance name>[<input port index>] ...;
cl[0] -> count1 -> [0] giveBack;
cl[1] -> count2 -> [1] giveBack;
```

Table 4.4: Click configuration example

buffers, is needed. We did not spend much time on that issue (see section 4.4.1) but concentrated on passing control information by means of the proc filesystem to userspace, as described in section 4.2.2.1.

Furthermore the Click installer[6] must be executed in order to tell the Click kernel module to implement the configuration. Since the Click installer must be run as system user *root*, we split the Click configurator into a server and a client, talking to each other by RMI as depicted in figure 4.3. Hence only the RMI server must run as *root*.

The RMI client simply calls a single method on the server side, passing along the compiled Click configuration, which is then saved into a temporary file where the Click installer reads and implements it.

4.2.2 JEE Configurator

The JEE configurator has to perform three main tasks:

1. Establish inter-EE communication links by inserting inter-EE adapters.
2. Download java byte code of all configured JEE elements from the code server.
3. Call RMI methods to install the JEE services.

In contrary to Click element parameters, JEE element parameters have no definite position but require a definite name. Parameter semantics are described in more detail in section 3.1.2.1.

Every JEE XML service implementation description has to specify the code location of the element. This is the path of the class file (containing java byte code) on the code server. Downloading the byte code is done by using the code fetcher (section 3.2.4).

[6]Provided with the Click distribution at [Click].

```
<PORTS>
    <OUT_PORT xsi:type="procfs_out" xsi:name="procfs_byte_count"/>
</PORTS>
<PARAM xsi:name="procfs_byte_count"
  xsi:value="$CLICKFSPATH$/$INSTANCE$/byte_count"/>
```

Table 4.5: Parameter expansion example

4.2.2.1 Inter-EE Configuration

Whenever the JEE configurator encounters a connection *from* another EE, it inserts a FromFile element, which has to read from a procfs file and forward the data to its push outport. But how does the FromFile element know from which procfs file it should read?

As described in section 3.1.2.1, service parameters may have names as do connection ports. The trick is to define a name for every Click procfs port, which corresponds to a service parameter of the same name containing the full procfs file path as its value. However parameters who's names start with `procfs_` are reserved for this purpose only and not passed further on to the EE. Table 4.5 shows an example of how that part of a service specification looks.

For inter-EE connections *to* another EE, the *ToFile* element is inserted. The same kind of file parameter assignment is done as described above.

4.3 Chameleon Java Execution Environment (J2E)

We adapted most of the JEE from [sdafann02]. The concept of elements and ports is the same. However our JEE runs in a separate process. Configuration is done with RMI. Figure 4.4 depicts the architecture.

One of our design goals was a single process standalone JEE. In order to separate the JEE from the configurator we wrote a RMI interface, providing a basic set of remote methods to manage the configuration of the JEE. The RMI interface provides the following methods:

add Adds a JEE service element to the current configuration. The element is represented as a byte array, containing a fully qualified java class object.

The instantiation and initialization of the element is done on the RMI server side. This approach does not require the element class to be serializable, thus we were not required to change the existing base classes much.

start Starts all service threads. Service elements may implement own java threads, as explained in section 4.3.1.

connect Connect two already added elements.

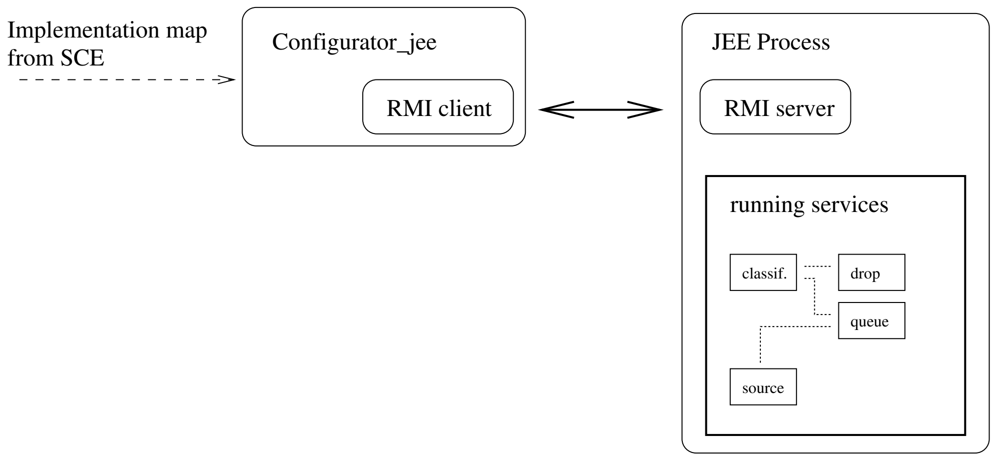


Figure 4.4: JEE architecture

reset Stop all service threads and remove all service elements from the configuration.

haveInstance Tell whether an element with the given instance name has already been added.

4.3.1 Elements Overview

The JEE supports two kinds of ports: *push* and *pull*. Their semantics are the same as described in [sdafann02].

We dropped the concept of compound containers and their special ports *link*, since we only support simple elements. Any service specifications are resolved into implementations[7] by the SCE's specification processor (see section 3.1.2), long before an implementation map reaches our JEE configurator.

The configuration string, passed to element constructors, can be analyzed by java Properties objects. This is prepared in our framework class `Container`. The XML service description of a JEE element must name its parameters[8].

JEE elements may run their own java threads, by implementing the Runnable java interface. Elements should not call the start() method on their own, since it is done by the JEE configurator for all elements at the same time. The threads (the `run()` method) should check for the stop flag, which will be set by the configurator as soon as an element is no longer used. By setting the running flag accordingly, elements must report their running status to the configurator. Both flags are part of the `Container` base class.

[7]Implementations are always leave nodes of a service tree.

[8]In contrary to the Click Router EE, where parameters are assigned according to their position.

4.3.2 Elements Provided

Name	Parameters	Ports	Purpose
FromFile	file, interval	1 OutPush	Reads data from a *file* every *interval* milli seconds.
ToFile	file	1 InPush	Writes data to a *file*.
Rate		1 InPush, 1 OutPush	Expects a byte counter value as string in the input packet and tells the current rate in byte/s to the output packet.
EMARate		1 InPush, 1 OutPush	Expects a byte counter value as string in the input packet and tells the *exponential moving average* rate in byte/s to the output packet.
Shaper	factor	1 InPush, 1 OutPush	Reads a single input number and multiplies it with the configured *factor*.

Table 4.6: Provided JEE elements

All these elements (see table 4.6) can not handle real socket buffers due to the lack of a userspace wrapper library. They process simple information patterns instead as typical control information. E.g. a single number in byte stream representation or the raw packet data without any kernel specific data structures or pointers.

4.3.3 Writing new JEE Elements

Your own element should...

- extend the base class ch.ethz.ee.tik.chameleon.jee.Container
- implement the constructor <element>(String name, String config)
- call the super class' constructor `super(name, in_port_type, number_of_in_ports, outport_type, number_of_outports, config)`
- read any configuration parameters from the protected Properties object of the Container class by calling its method `getProperty(String name)`
- implement `simpleAction(DataPacket p)` which returns the processed data packet

If your element needs its own java thread, implement the Runnable interface but do not call start() from your constructor. It will be called from the RMI server for all elements at the same time for the sake of synchrony.

4.4 Click Execution Environment

Other than the Click configurator we did not need any code to integrate Click into our SCE. However we thought of another inter-EE communication facility, described in the following section.

4.4.1 Click Netlink Module

We attempted to implement a netlink connection between Click and the JEE. This function requires two modules: a Click element which connects click with a netlink socket and a native java implementation for the userspace side.

The Click element[9] has not been tested acceptedly, nevertheless it is capable of transmitting and receiving packets via netlink. The native java library is not runnable[10].

We ceased development of this module, due to low priority and lack of time. Inter-EE communication is still possible by means of the *procfs* interface described in section 4.2.2.1.

[9]To be found in our archive in an/click_netlink/
[10]In our archive in an/java/jee_netlink/

Chapter 5

Conclusion

This chapter summarizes all goals achieved in respect of the three issues *service composition*, *EE integration* and the *demultiplexer*. As conclusion about the whole thesis we can state that every single component of the implemented active node platform has been designed as flexible and modular as possible. However, it was our experience that for practical reasons, not every detail can be abstracted and designed in a completely generic way. Decisions had to be made which are specific for our two EEs. It would be illusionary to assume that an active node platform could be implemented in such a generic way that any addition and modification of it could be done without the need of any implementation-specific measures.

Nevertheless, we think that our active node is a well designed and flexible framework which implies a maximum of usability and a minimum of complexity.

5.1 Service Composition

The service creation engine has been implemented as Java application running in userspace. In its idle state, the SCE waits for an incoming service request which is processed in turn. The major processing steps done by the SCE include the resolution of all internal dependencies, various validation steps, selection of one implementation by consulting the mapping policy and finally the installation of all service components in the available EEs.

For the sake of adaptility and readability, service requests and descriptions are formatted in XML. The resolution of internal dependencies leads to a hierarchical structure of the service components. This allows the processing and installation of arbitrarily complex services.

As overall function, the SCE is responsible for mapping the node-independent service request against the node capabilities. The latter are represented by the node description which are stored as an XML file. This concept results in a very flexible node administration in case of platform changes, like adding new EEs or upgrading the OS.

The final selection of one implementation can be influenced by specifying an appropriate mapping policy. Currently, one mapping policy is supported which minimizes the EE transitions.

Service descriptions and service code may be stored separately on different hosts in the network. As a result distributed resources and redundance can be provided.

5.2 EE Integration

We adapted the *Chameleon* Java EE from [sdafann02] in order to integrate it into our active node platform as a stand-alone process. Some features were stripped off since the enhanced SCE covers them and some minor functions were added to fit into our RMI-based EE architecture. Services may be implemented easily for this EE whereas performance is not of much importance like in any Java application.

The integration of the Click modular router as a kernel EE not only features high performance. The Click router project already provides a huge number of services. They may be used by solely editing appropriate XML service descriptions. Refer to [Click] for more details.

Furthermore, inter-EE communication has been implemented which makes services more flexible. During validation of the service tree, services which are provided by several EEs, may be selected based on the mapping policy described above.

5.3 Demultiplexer

The Netfilter/Iptables-based demultiplexer allows simple extensions for new EEs. Network packets addressing a particular service may be forwarded appropriately by means of the large and powerful rule set of Iptables.

Chapter 6

Known Bugs and Future Work

6.1 Major Issues

- Click does not allow mixed push/pull semantics for *agnostic* ports, as shown in figure 6.1. This constraint is not checked by the validator since it requires redesigning service description semantics.
- The Click EE can not be considered as stable since the active nodes kernel crashes upon a second service configuration. The reason for this could be a reinitialization bug in our FromNetfilter Click module or another bug in the official Click distribution. We could not investigate the reason for this bug due to lack of time and because the kernel oops[1] could not be written to a file for further analysis by means of *ksymoops*.

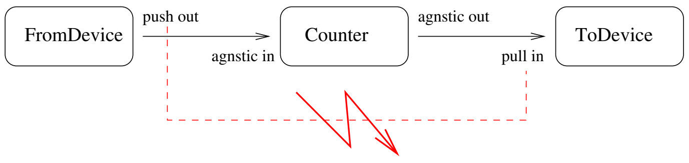


Figure 6.1: Mixed Click push/pull semantics with agnostic ports

6.2 Minor Issues

- The design of the SCE does not allow configuration of additional services nor selective removal of services from the running configuration. This feature would require significant changes to the SCE framework. Configurators

[1]A kernel error message which is issued whenever the kernel encounters an internal critical situation. This happens very rarely on original (not patched) kernels.

would neccesitate storing the current configuration and distinguishing services that must be newly created from services already running. The demultiplexer would have to perform a similar task, reconfiguring EE entry points as far as required. Special care has to be taken when adapter elements are inserted (at service entry and exit points or inter-EE borders).

- Since Iptables configuration commands are executed as root, passing Iptables filter rules from the XML service description and the node description directly and unchecked to the demux configurator bears a security risk (see section 3.2.2). A solution to this problem could be a new demux configuration language which is translated into Iptables commands by a strict parser, checking for shell escape characters.
- The demux adapter elements and also inter-EE adapters have to consist of a single implementation. For more complex EEs, which might be supported in the future, whole service specifications containing subservices could be of interest. Changes would be required in the demux configurator and the EE configurators, where adapter elements are inserted into the implementation map.
- The current way of service deployment does not revalidate a service tree after insertion of adapter elements by the EE configurators and the demux configurator. Bogus XML configurations might not lead to clearly stated error messages. Some configurators might simply fail to deploy services instead, even though the service request had been validated successfully by the SCE.

  A known bug consists of the fact that service entry points with pull semantics cause trouble which is not detected by the demux configurator, as explained in section 3.2.2.
- Some future EEs might need to avoid the wire ID appended to demux Iptables rules. An (untested) solution might be appending a '`#`' to the demux rule target specification in the node description. This should make the wire ID look like a comment to the shell, inhibiting Iptables from seeing it.
- As mentioned in section 11 on page 40 and section 4.4.1 the netlink interface for the JEE has not yet been finished.
- In some cases the SCE crashed when specifications are used as service requests. We had no time to care about it. The SCE works perfectly if simple requests are used.
- If a service contains a specification as floating node, the according specification must not have entry points. In other words, the specification node only contains floating nodes and their direct and indirect successors.
- In [SDA] the concept of funnels and tubes is introduced. Our SCE handles both as the same kind of transport connections.

- After changing any XML configuration files the SCE need to be restarted. This means services and EEs can not be added at runtime. Adding services at runtime needs minor changes to the code server which presently scans XML service description upon first instantiation only. Adding new EEs at runtime has been prepared already by means of loading the configurator classes dynamically as described in section 3.2.1. Minor changes to the service server are necessary to rescan service descriptions more often.

- The SCE has to iterate trees and maps at different times. Maybe this task could be unified by means of a single iterator which knows how to traverse all the main data structures. This approach already has been chosen in the configurators whereas our iterator *ImplementationMapIterator* is not resistant against changes of the data structures during traversal.

Bibliography

[SDA] Matthias Bossardt *A Service Deployment Architecture for Heterogenous Active Nodes*;
ETH DITET TIK, 2002.

[sdafann02] Florian Kaufmann *Service Deployment for a Java-based Active Network Node*;
ETH DITET TIK, 2002.

[Click] The Click Router Project *http://www.pdos.lcs.mit.edu/click/*

[Netfilter] Netfilter / Iptables Project *http://www.netfilter.org/*

Appendix A

Installation Guide

A.1 Requirements

- Linux Kernel 2.4.20 from http://www.kernel.org/
- Click router from http://www.pdos.lcs.mit.edu/click/ (CVS)
- Iptables userspace tools 1.2.7a from http://www.netfilter.org/

A.2 Installation steps

Note: you can find every software required on the CDROM archive additionally.

A.2.1 Preparing the kernel

1. Unpack the kernel sources to /usr/src/linux. Unpack also click and Iptables sources.
2. Make sure the following symbolic links point at these places in the unpacked kernel sources:

   ```
   /usr/include/linux -> /usr/src/linux/include/linux
   /usr/include/asm -> /usr/src/linux/include/asm-i386
   ```

3. Patch the kernel with the appropriate click patch to be found in the unpacked click source in `etc/linux-2.4.20-patch`.
4. Patch the kernel again with the Click-Netfilter patch `click_netfilter/kernel-2.4.20.diff` found in our software package.
5. Configure the kernel making sure the Netfilter CLICK-Target and both the MARK-Target and the MARK-match are enabled as module and install new kernel.

6. Patch the Iptables userspace tools with the click_netfilter patch from our software package.
7. Build and install new Iptables userspace tools.
8. Boot the new kernel.
9. Copy fromnetfilter.{hh,cc} and netlink.{hh,cc} into click/elements/local
10. Configure, build and install click. Pass `--enable-local` to the configure script.

A.2.2 Java environment

1. Make sure JDK 1.4 or higher is installed (from your Linux distribution).
2. Get and install *jdom* (betha 8) from http://www.jdom.org.
3. Find our java source code in our archive at *an/java*.
4. Get and install *xerces* (2.2.1) from http://xml.apache.org.
5. Get and install the XPath evaluator *jaxen* (1.0) from http://jaxen.sourceforge.net/
   Note: We needed jaxen-full.jar and lib/saxpath.jar to be in the CLASSPATH. *Jaxen* might get integrated into jdom soon.
6. Adjust your CLASSPATH environment variable so that it contains those jar archives and the top directory of the java source code.
7. Adapt ch/ethz/ee/tik/chameleon/util/{HostConstants,Constants}.java to your needs. You should adapt HostConstants.java to reflect your file paths. Or you can create a symlink from `an/xml` to `/tmp/an`. Make sure CODE_SERVER_BINARYS_PATH points to the directory where the JEE services are. This should be `an/java/jee/services`.
8. Build java software, using the makefile in the java top directory. Simply type *make*.

Appendix B

User Guide

B.1 Running the Active Node

The shell script *run.sh* in the java subdirectory launches the basic framework. It starts the following processes:

Service creation engine The SCE main process, waiting for TCP service requests. See section 3.

Code server The code server, maybe running on another host. See section 3.3.1.

Java EE The java EE (J2E), see chapter 4.3.

Service server Maintaining an XML based database of service descriptions. May run on another host. See section 3.3.2.

Click configuration server Runs as root as RMI server, see section 4.2.1.

Demux configurator Runs as root as RMI server, see section 3.2.2.

Now you may request services as shown in the example implementation *ch.ethz.ee.tik.chameleon.sce.ServiceRequester*. See also the sample service requests in xml/servicerequester/.

Note: *run.sh* also inserts the kernel module *ipt_CLICK.o* since Click depends on it even when there are no Iptables rules set using the target *CLICK*.

For the demonstration we used the script *an/demo/rundemo.sh*. See its code for the various startup modes. Each command line option fulfills a single task needed for the complete demonstration.

B.2 Integrating new EEs

To integrate a new EE, the following minimal tasks are required:

- Write a *Configurator* like those described in section 3.2.3. Make sure, it fulfills the minimal set of tasks:

  - Insert inter-EE adapters into the implementation map.
  - Download code modules by means of the code fetcher if the EE supports dynamic code loading.
  - Reset its EE to stop old services.
  - Install the new implementation map considering services, their parameters, and connections. Port assignment must be preserved. Refer to our implementations *Configurator_jee* and *Configurator_click* to see how node iteration is done correctly in order to fetch all connections.

- Add the new EE to the *node description*, see section B.3.1. Add a single *EE* element and as many *EE_CONNECTIONS* as required in order to reflect your EE's port semantics to the node description.

- Write *service descriptions* as explained in section B.3.2 for integration of your EE's services.

B.3 XML Files

This section describes the structure of all XML files needed to run services on the AN. It shows how to write new service descriptions and other features covered by XML configuration files.
Some notes about the XML graphs:

- Elements are surrounded by ellipses.
- Attributes are encircled by a hexagon.
- Doubly surrounded Elements may appear more than once.
- Dashed arrows point to optional XML nodes. Solid arrows mark nodes as mandatory.

B.3.1 Node Description

Table B.1 shows an example node description. The meanings of the XML nodes depicted in figure B.1 are:

NODE_DESCRIPTION The root element of the node description.

OS Compound element for the supported OSs of this node.

OS:name The name of our OS.

OS:version Our OS' version.

EE Compound element for an EE description.

EE:name The name of the EE.

EE:configurator The suffix of the Configurator class for this EE (see section 3.2.3).

EE:demux_target If the EE should get packets from the demux, it must specify the *Iptables target* passing packets to that EE (see section 4.1.2, page 37).

EE:version The EE's version.

PORT A port description of a single kind of ports.

Port:type The type name of that port.

Port:flag Optional port flags. Currently only *optional* is supported. A port with the optional flag set may be left unconnected in the implementation map as described in section 3.1.4.1 on page 16.

EE_CONNECTION Compound element for service connections. As normal case you have one per EE, describing intra-EE connections (fromEE and toEE both contain the current EE's name) and one per inter-EE connections.

fromEE The name of the originating EE of the listed connections.

toEE The name of the destination EE these connections go to.

CONNECTION A connection that comes from *fromEE* and goes to *toEE*.

fromPort The name of the kind of source port for a connection. Contains a valid port name, specified in the *EE* elements *type* attribute.

fromPort The name of the kind of destination port for a connection. Contains a valid port name, specified in the *EE* elements *type* attribute.

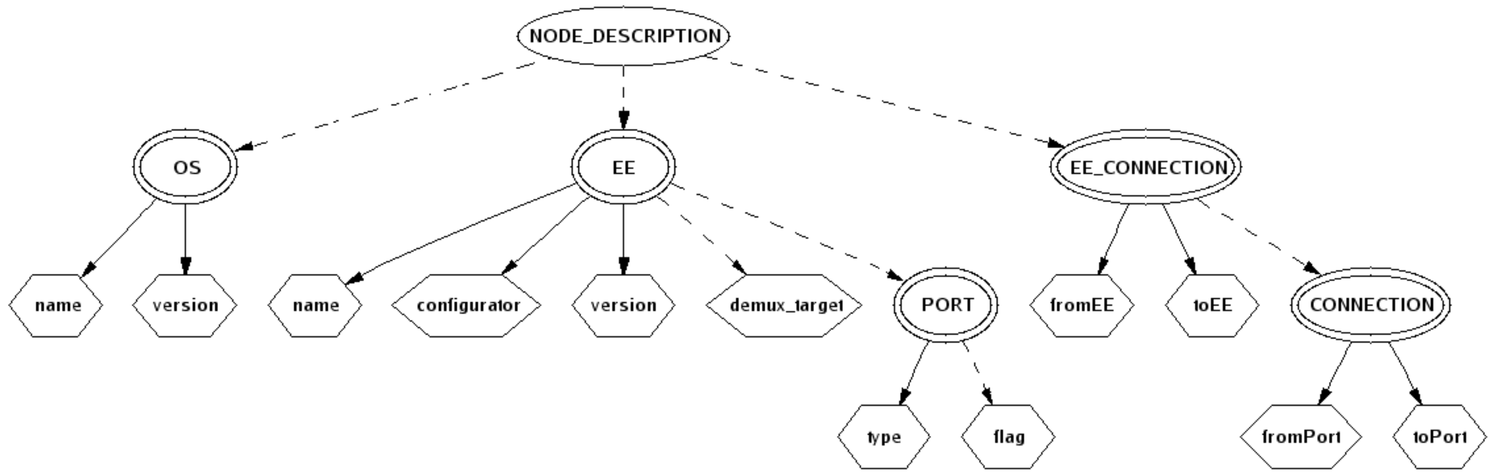


Figure B.1: XML node description graph

B.3.2 Service Descriptions

As mentioned in section 3.1.1, two different kinds of service descriptions are known, which may appear mixed in one file:

Specification A compound description, containing one or more subservices.

Implementation The atomic description of a service, representing a single service element of an EE.

Table B.2 shows an example of a specification, table B.3 shows an example of implementations.

B.3.2.1 Service Specification

Description of the elements (see figure B.2):

SERVICE_LIST XML root element.

SERVICE Compound element of a service.

SERVICE:type The type of this service. For specifications always *SPECIFICATIONS*.

DESCRIPTION Compound element.

servicename The name of this service.

PROVIDER The manufacturer of this service.

VERSION The service version.

DEMUX_RULE Demux rule pattern (Iptables rule options) for service entry points (see section 4.1.2 on page 37).

PORTS Compound element for the port list.

IN_PORT An input port and its type.

IN_PORT An output port and its type.

PARAM A service parameter. Parameter inheritance and handling is described in section 3.1.2.1.

PARAM:name The name of a parameter. Parameters are inherited by order. Some EEs need parameter names, some other discard it (e.g. Chameleon J2E).

PARAM:value The optional value of a parameter.

SUB_SERVICE Compound element.

SUB_SERVICE:name The service name of a subservice.

SUB_SERVICE:instance_name The uniq instance name of the service.

SUB_SERVICE:PARAM A parameter reference. Its name must correspond to an existing parameter among the service parameters.

TRANSPORT_CONNECTION Compount element for a connection between subservices.

SRC_PORT Source port of a transport connection.

DEST_PORT Destination port of a transport connection.

{SRC,DEST}_PORT:name Optional port name, used for routing (see section 3.1.4.1).

{SRC,DEST}_PORT:instance_name Instance name of the connected service.

```
<?xml version="1.0" encoding="UTF−8"?>
<NODE_DESCRIPTION xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:noNamespaceSchemaLocation="C:\Chameleon\XML\chameleon.xsd">
    <!−− configurator: name of java class configuring this EE
         name: name of this EE, corresponds to service descriptions
    −−>
    <xsi:OS xsi:name="Linux" xsi:version="2.4.20"/>
    <xsi:OS xsi:name="Linux" xsi:version="2.4.19"/>
    <xsi:OS xsi:name="Linux" xsi:version="2.4.18"/>
    <xsi:OS xsi:name="AS400" xsi:version="23.4.56.pre12 patchlevel 5"/>
    <xsi:EE xsi:name="JEE" xsi:configurator="jee" xsi:version="1">
        <xsi:PORT xsi:type="pull_in"/>
        <xsi:PORT xsi:type="push_in"/>
        <xsi:PORT xsi:type="pull_out"/>
        <xsi:PORT xsi:type="push_out"/>
    </xsi:EE>
    <xsi:EE xsi:name="CLICK" xsi:configurator="click" xsi:version="1"
            xsi:entry_module="FromNetfilter" xsi:exit_module="ToHost"
            xsi:demux_target="−j CLICK −−click−wire ">
        <xsi:PORT xsi:type="pull_in"/>
        <xsi:PORT xsi:type="push_in"/>
        <xsi:PORT xsi:type="pull_out"/>
        <xsi:PORT xsi:type="push_out"/>
        <xsi:PORT xsi:type="agnostic_in"/>
        <xsi:PORT xsi:type="agnostic_out"/>
        <xsi:PORT xsi:type="procfs_in" xsi:flag="optional"/>
        <xsi:PORT xsi:type="procfs_out" xsi:flag="optional"/>
    </xsi:EE>
    <xsi:EE_CONNECTIONS xsi:fromEE="CLICK" xsi:toEE="JEE">
        <xsi:CONNECTION xsi:fromPort="procfs_out" xsi:toPort="push_in"/>
    </xsi:EE_CONNECTIONS>
    <xsi:EE_CONNECTIONS xsi:fromEE="JEE" xsi:toEE="CLICK">
        <xsi:CONNECTION xsi:toPort="procfs_in" xsi:fromPort="push_out"/>
    </xsi:EE_CONNECTIONS>
    <xsi:EE_CONNECTIONS xsi:fromEE="CLICK" xsi:toEE="CLICK">
        <xsi:CONNECTION xsi:fromPort="pull_out" xsi:toPort="pull_in"/>
        <xsi:CONNECTION xsi:fromPort="push_out" xsi:toPort="push_in"/>
        <xsi:CONNECTION xsi:fromPort="pull_out" xsi:toPort="agnostic_in"/>
        <xsi:CONNECTION xsi:fromPort="push_out" xsi:toPort="agnostic_in"/>
        <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="agnostic_in"/>
        <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="push_in"/>
        <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="pull_in"/>
    </xsi:EE_CONNECTIONS>
    <xsi:EE_CONNECTIONS xsi:fromEE="JEE" xsi:toEE="JEE">
        <xsi:CONNECTION xsi:fromPort="pull_out" xsi:toPort="pull_in"/>
        <xsi:CONNECTION xsi:fromPort="push_out" xsi:toPort="push_in"/>
    </xsi:EE_CONNECTIONS>
</NODE_DESCRIPTION>
```

Table B.1: Example of a complete node description

```
<?xml version="1.0" encoding="UTF−8"?>
<SERVICE_LIST xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:noNamespaceSchemaLocation="C:\Chameleon\XML\chameleon.xsd">
<SERVICE xsi:type="SPECIFICATION">
        <DESCRIPTION xsi:servicename="bandwidth_checker" xsi:option="normal">
                <PROVIDER>ETH</PROVIDER>
                <VERSION>2.1</VERSION>
        </DESCRIPTION>
        <PORTS>
                <IN_PORT xsi:type="push_in"/>
                <OUT_PORT xsi:type="push_out"/>
        </PORTS>
        <PARAM xsi:name="file" xsi:value="/tmp/count"/>
        <SUB_SERVICE xsi:name="Counter" xsi:instance_name="counter">
        </SUB_SERVICE>
        <SUB_SERVICE xsi:name="ToFile" xsi:instance_name="result_writer">
                <PARAM xsi:name="file"/>
        </SUB_SERVICE>
        <TRANSPORT_CONNECTION>
                <SRC_PORT xsi:service_instance="this" xsi:type="push_in"/>
                <DEST_PORT xsi:service_instance="counter" xsi:type="agnostic_in"/>
        </TRANSPORT_CONNECTION>
        <TRANSPORT_CONNECTION>
                <SRC_PORT xsi:service_instance="counter" xsi:name="procfs_byte_count"
                   xsi:type="procfs_out"/>
                <DEST_PORT xsi:service_instance="result_writer" xsi:type="push_in"/>
        </TRANSPORT_CONNECTION>
        <TRANSPORT_CONNECTION>
                <SRC_PORT xsi:service_instance="counter" xsi:type="agnostic_out"/>
                <DEST_PORT xsi:service_instance="this" xsi:type="push_out"/>
        </TRANSPORT_CONNECTION>
</SERVICE>

</SERVICE_LIST>
```

Table B.2: Example of a service specification

```
<?xml version="1.0" encoding="UTF−8"?>
<SERVICE_LIST xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:noNamespaceSchemaLocation="C:\Chameleon\XML\chameleon.xsd">
<SERVICE xsi:type="IMPLEMENTATION">
    <DESCRIPTION xsi:servicename="FromFile" xsi:option="normal">
        <PROVIDER>ETH</PROVIDER>
        <VERSION>2.1</VERSION>
    </DESCRIPTION>
    <ENVIRONMENT>
        <OS>
            <NAME>Linux</NAME>
            <VERSION>2.4.19</VERSION>
        </OS>
        <EE>
            <NAME>JEE</NAME>
            <VERSION>1</VERSION>
        </EE>
    </ENVIRONMENT>
        <CODE_LOCATION>FromFile.class</CODE_LOCATION>
    <PORTS>
        <OUT_PORT xsi:type="push_out"/>
    </PORTS>
    <PARAM xsi:name="file"/>
    <PARAM xsi:name="interval" xsi:value="510"/>
</SERVICE>
<SERVICE xsi:type="IMPLEMENTATION">
    <DESCRIPTION xsi:servicename="ToFile" xsi:option="normal">
        <PROVIDER>ETH</PROVIDER>
        <VERSION>2.1</VERSION>
    </DESCRIPTION>
    <ENVIRONMENT>
        <OS>
            <NAME>Linux</NAME>
            <VERSION>2.4.19</VERSION>
        </OS>
        <EE>
            <NAME>JEE</NAME>
            <VERSION>1</VERSION>
        </EE>
    </ENVIRONMENT>
        <CODE_LOCATION>ToFile.class</CODE_LOCATION>
    <PORTS>
        <IN_PORT xsi:type="push_in"/>
    </PORTS>
    <PARAM xsi:name="file"/>
    <!−− PARAM xsi:name="debug" xsi:value="1"/ −−>
</SERVICE>
</SERVICE_LIST>
```

Table B.3: Example of service implementations

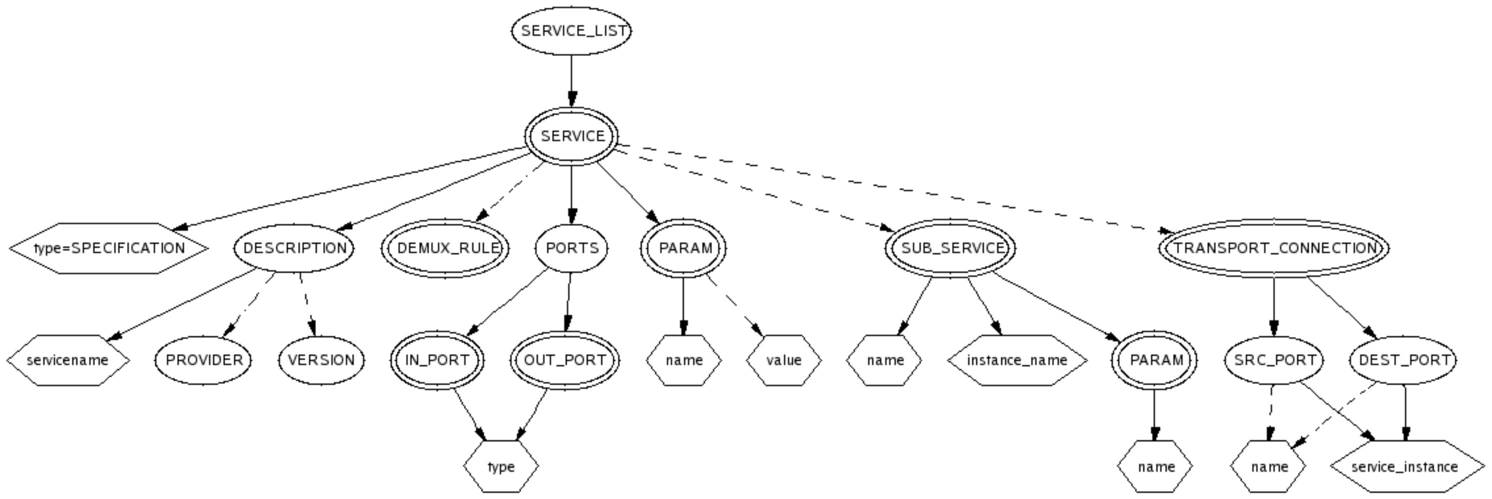


Figure B.2: XML service specification and service request graph

B.3.2.2 Service Implementation

Figure B.3 shows how an XML service implementation looks like. The following list only mentions the differences to the service specification in section B.3.2.1.

ENVIRONMENT Compound element.

OS, NAME, VERSION Must match the same fields in the node description.

EE, NAME, VERSION Must match an EE descriptor in the node description.

PARAM:value Parameters with given values in implementations are constant parameters. They can not be overwritten by service specifications.

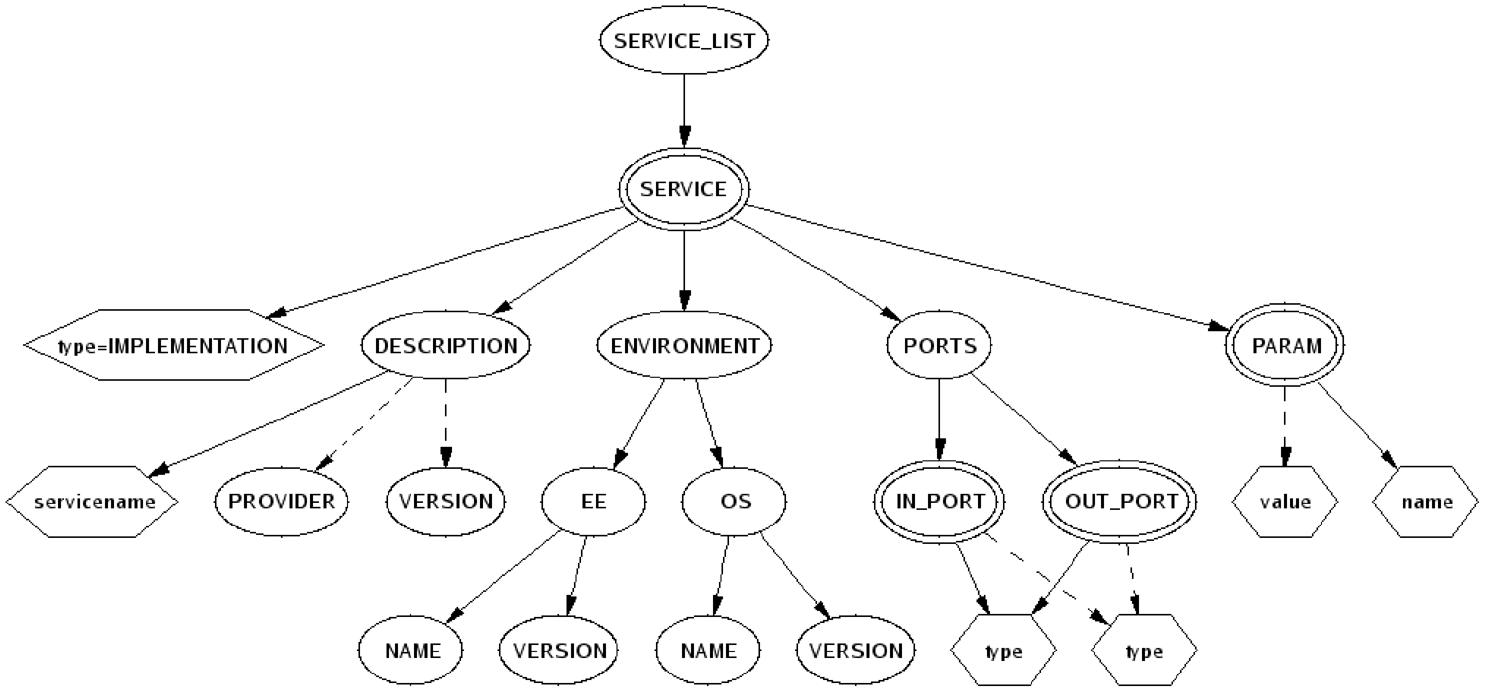


Figure B.3: XML service implementation graph

B.3.3 Service Request

Whenever the user wants to request the installation of a service, she has to submit a service request to the SCE. Service requests may appear in two different forms, as described in section 3.1.2. The simple form is self-explanatory and shown in figure B.4. The more complex form is just a full service specification, explained in section B.3.2. Table B.4 shows an example of a simple service request.

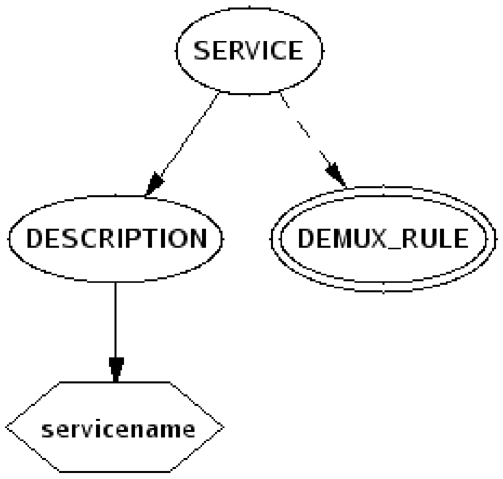


Figure B.4: XML simple service request graph

```
<?xml version="1.0" encoding="UTF-8"?>
<SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Chameleon\XML\chameleon.xsd">
        <DESCRIPTION xsi:servicename="bandwidth_checker"/>
        <DEMUX_RULE>-p tcp --dport 80</DEMUX_RULE>
</SERVICE>
```

Table B.4: Example of a simple service request

Appendix C

Introduction to Netfilter

Netfilter/Iptables is the official packet filter shipped with the linux kernel version 2.4. Its main advantages over the old Ipchains packet filter are the modular architecture and the less intrusive kernel interface making extensions easier and more stable.
A lot of documentation can be found on [Netfilter].

C.1 Netfilter/Iptables Concept

The Netfilter framework provides hooks for examining packets traveling through the local network node at different points in the kernel. You may think of hooks as of checkpoints where decisions are made about the packets fate or where packets get altered. Iptables uses these hooks in order to apply its filter rules.

C.1.1 Netfilter

One of the main goals of the design of Netfilter was changing as little code of the existing IP stack as possible. Figure C.1 shows the five checkpoints where packet inspection may take place.

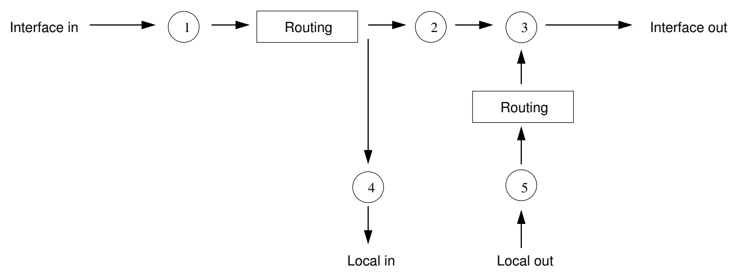


Figure C.1: Netfilter hooks

1. PREROUTING Packets coming from other machines over the network can be checked out here just before the routing decision is to be met. A typical application of this hook is DNAT[1]. See also section C.2.
2. FORWARD Packets that came from outside and which are destined for another host pass through this hook.
3. POSTROUTING All packets leaving our host are inspected by this hook. SNAT[2] and masquerading are done here usually. See also section C.2.
4. INPUT Packets destined for the local host pass this hook just before being given to the application layer.
5. OUTPUT This hook sees packets generated locally just before routing is done.

C.1.2 Iptables

As already known from other packet filters also Iptables holds the filter rules in chains. The user configures the rules in chains whereas the order of rules is a decisive factor. Since each packet is compared to the match criterions of each rule, starting at the first rule of a chain. If a rule matches, the rules action (target) is taken emmediately and rule processing stops for this packet with few exceptions.

Iptables introduces a specific concept of tables holding several chains. Each chain of a table is connected with a Netfilter hook, except user defined chains. Figure C.2 depicts the default tables and their chains.

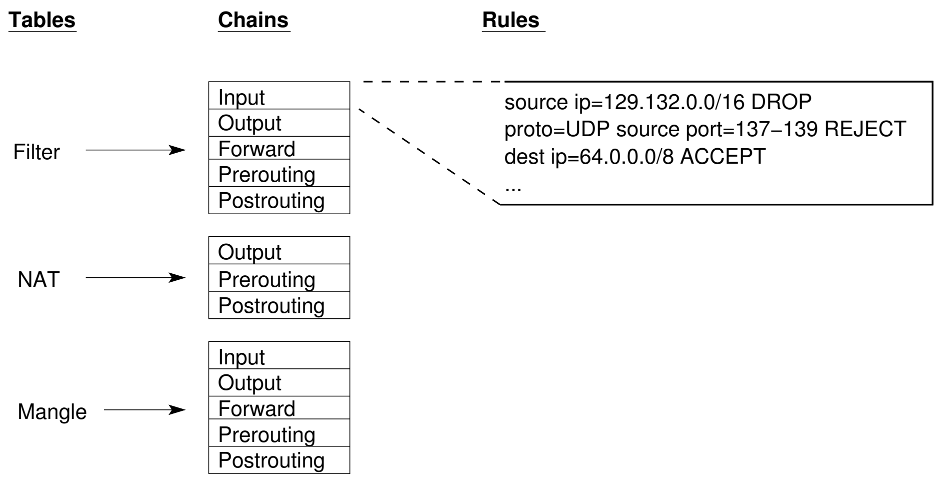


Figure C.2: Iptables: tables, chains and rules

[1]Destination Network Address Translation
[2]Source Network Address Translation

Since several tables hold Input chains, the chains of the same hook are hooked into Netfilter according to their tables priority defined by the implementation of Iptables. NAT chains are hooked *before* filter chains for example.

Filter is the standard table where packets mainly are rejected or accepted.

Nat The address of packets are changed here.

Mangle This table is meant for all other kinds of manipulations like changing the TTL value of an IP packet.

C.2 NAT

Network address translation means changing the source or target address of a packet in general. Two sub methods can be distinguished: SNAT and DNAT.

C.2.1 DNAT

DNAT changes the destination address of packets. This is usefull for example if a firewall should pass incoming connection to a server in the DMZ.

C.2.2 SNAT

SNAT changes the source address of outgoing packets. This method is usefull for network renumbering since that way hosts addresses appear in networks without really having such an address.

C.2.3 Masquerading

Masquerading is a special case of SNAT where the source address of outgoing packets are rewritten to the address of the outgoing interface of the firewall. That way hosts from the inner network are hidden to the outside world.

The routing of reply packets into the inner network is done automatically before they pass the prerouting hook.

This method is very usefull for large local networks needing internet access since all the inner hosts only need one public internet address which is given to the firewall.

C.3 Data structures

The following sections discuss some of the more important data structures of Iptables in kernel version 2.4.18.

The file net/ipv4/netfilter/ip_tables.c demonstrates the basic architecture of Iptables.

```
static LIST_HEAD(ipt_target);
static LIST_HEAD(ipt_match);
static LIST_HEAD(ipt_tables);
```

These three lists exist globally and uniquely.

tables Tables hold chains holding rules.

New tables are registered by the function ipt_register_table[3]. The hooking (section C.1.1) into Netfilter has to be done explicitly afterwards.

match This list holds data structures of all known matches that may be used by rules. Mainly these structures hold pointers to functions performing the actual match (e.g. comparing source address with given pattern).

target The target list is similar to the match list. If a packet matches some rule, the Iptables code looks into this list for the appropriate function which provides the rules action. The return value of those target functions determins the fate of the packet; e.g. whether it will be accepted or rejected.

[3] net/ipv4/netfilter/ip_tables.c:1378

Appendix D

Assignment

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Winter 2002/2003

Diplomarbeit
für

Roman Hoog Antink and Andreas Moser

Supervisor:	Matthias Bossardt
Co-Supervisor:	Lukas Ruf

Ausgabe: 4.11.2002
Abgabe: 18.3.2003

Service Composition for Active Networks

1 Introduction

Active networks contain programmable routers, which allow users to dynamically install programs. Such programs are typically part of a distributed service deployed in the network. Active routers enable the customization of packet handling in a very flexible way. Possible services include packet filtering, Web caches, as well as specialised multicasting protocols and video scaling.

An active network node is similar to a classical router in a legacy IP network. In both cases, the main task is to forward data packets. Additionally, an active node allows freely customizing the processing of data packets on the node. As a consequence, networks that include active network nodes may offer services in addition to the basic IP forwarding services of classical IP networks. A service is composed of service components that must be deployed on one or more active nodes.

At TIK, we developed a service deployment architecture for active network *nodes* [2], which allows to identify and deploy service components that must be executed on a specific node.[1]

[1]There are other approaches to network level service deployment. For a discussion of the service deployment design space see [1].

Service deployment includes installing and configuring software components that perform processing of the data packets. Service components run in execution environments. Many types of execution environments (EE) have been developed each of which is usually optimized 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 EEs. All this motivated us to develop a service deployment scheme that can cope with heterogeneous active nodes, i.e., with active nodes running sets of EEs that can vary from node to node. Our goal was to develop a scheme, where the service specification is node-independent, but service deployment and installation recognize the diversity of EEs, thereby allowing a service to exploit the particular functionality and performance features of active network nodes. The *service creation engine* (SCE), available on each active node, is responsible to carry out this mapping.

The current implementation features a Java-based EE. From a performance point of view, such an EE, however, is not sufficient for transport plane operations. That is, for packet processing at line speed. More efficient approaches for this type of task include EE that execute code in the kernel space.

Click router [5] enables simple configuration of the transport plane of a software router. Click executes code in kernel space and provides a variety of service components (Click terminology: elements). It lacks, however, several features of an actual EE. Nevertheless, it is an interesting candidate for an EE focussing on transport plane functionality.

2 Assignment

2.1 Objectives

The programs on active routers usually consist of several service components that must be installed and configured correctly in order to provide the requested service. Our active router prototype is able to compose services that are described in an XML-based description language. Currently, only service components written in Java are supported. Since these components are executed in a Java Virtual Machine (JVM), the JVM is called an execution environment (EE).

The goal of this project is to extend our prototype such that service components can be installed and configured in other execution environments as well. E.g. it is useful to support a kernel execution environment, which allows to achieve much better performance. Click router will be used as a base for a new kernel EE.

The project consists of three main parts.

- Preparation of the basic platform (nodeOS with demultiplexer).
- Extension of service creation engine (SCE).
- Design and implementation of a Click-based EE.

The node operating system (nodeOS) must contain an efficient demultiplexer allowing dispatching incoming packets to the appropriate EEs. Furthermore, it must be possible that service components in different execution environments can communicate with each other. Possible communication facilities offered by a Linux-based nodeOS include sockets and the proc filesystem.

The SCE must be extended to support more than one EE on a node. Therefore, it would be useful if node capabilities, such as number and types of EEs, were represented in a proper way (e.g. via an XML-based *node description*). Adding a new type of EE on a node should not require modifications of the SCE, but only of the node description. As is the case with a Click-based

EE, it might be required that EE-specific translator logic must be added. That is, the SCE internal representation of service components and their interconnection must be translated into one that is understood by the EE-specific configurator. The SCE should support adding translator logic in a standardized way. The mapping of a service description to an implementation should be based on matching the service requirements (described in the XML service descriptor) against the node capabilities (described in the XML node descriptor).

To build an Click-based EE, Click router must be modified in a way to be connected to the demultiplexer. Click elements should be stored on a code server. Furthermore, it must possible to easily extend the number and types of Click elements (service components) by making them available on a code server. The SCE must be able to retrieve Click elements from the code server and install them in the Click-based EE.

2.2 Tasks

- Get familiar with active networks and service deployment concepts. Read and understand the assigned papers, which deal with this topic [6, 7, 2].
- Get familiar with the semester thesis by Florian Kaufmann entitled 'Service Deployment for a Java-based Active Network Node'.
- Set up your work environment.
  - Install Linux (Kernel 2.4.17) on your computer.
  - Apply PromethOS patches (`http://www.promethos.org`).
  - Use CVS (`http://www.cvshome.org`) to manage your code.
- Get familiar with Click Router and XORP.
  - Read the corresponding papers [5, 3, 4].
  - Install and test the Click software.
- Prepare the active node platform
  - Consider using Netfilter (`http://www.netfilter.org`), PromethOS, and libipqmp (`http://www.gnumonks.org/projects/project_details?p_id=2`) for demultiplexing packets.
- Extend the service creation engine to deal with several EEs.
  - Provide an XML based description of node capabilities.
  - Extend the SCE to be able to match the node description against the service description.
  - Redesign the SCE to be extensible with other EEs.
- Design a new kernel EE.
  - Use Click as the basis of your design.
  - Connect Click to the demultiplexer.
  - Provide an interface between the Click-based EE and the SCE that allows to install and connect Click elements (Tip: the interface may use the click configuration file).
  - Store Click elements on a code server and retrieve them from there.
  - Provide XML service descriptors for some of the Click elements.

- Inter-EE communication between Click-based EE and Java-based EE.
  - Evaluate different potential communication facilities (sockets, procfs, etc.)
  - Implement necessary communication adapters for both EEs.
- Optionally: Come up with a concept to deploy services on routers with distributed processing elements (e.g. one network processor per interface card).
- Provide an evaluation plan and evaluate your work accordingly.
- Implement a short demo of your work.
- Document your work in a detailed and comprehensive way. We suggest you to continually update your documentation. New concepts and investigated variants must be described. Decisions for a particular variant must be justified.

3 Deliverables and Organisation

- If possible, students and advisor meet on a weekly basis to discuss progress of work and next steps. If problems/questions arise that can not be solved independently, the students may contact the advisor anytime.
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- At half time of the semester thesis, a short discussion of 15 minutes with the professor and the advisor will take place. The student has to talk about the major aspects of the ongoing work. At this point, the student should already have a preliminary version of the table of contents of the final report. This preliminary version should be brought along to the short discussion.
- At the end of the semester thesis, a presentation of 15 minutes must be given during the TIK or the communication systems group meeting. It should give an overview as well as the most important details of the work. Furthermore, it should include a small demo of the project.
- The final report may be written in English or German. It must contain a summary written in both English and German, the assignment and the time schedule. Its structure should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Related work must be correctly referenced. See `http://www.tik.ee.ethz.ch/flury/tips.html` for more tips. Three copies of the final report must be delivered to TIK.
- Documentation and software must be delivered on a CDROM.

Literatur

[1] Matthias Bossardt, Takashi Egawa, Hideki Otsuki, and Bernhard Plattner. Integrated service deployment for active networks. In James Sterbenz, Osamu Takada, Christian Tschudin,

and Bernhard Plattner, editors, *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.

[2] 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.

[3] Mark Handley, Orion Hodson, and Eddie Kohler. Xorp: An open platform for network research. In *Proceedings of First Workshop on Hot Topics in Networks*, Princeton, New Jersey, 2002.

[4] Eddie Kohler. *The Click modular router*. PhD thesis, Massachusetts Institute of Technology, November 2000.

[5] 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.

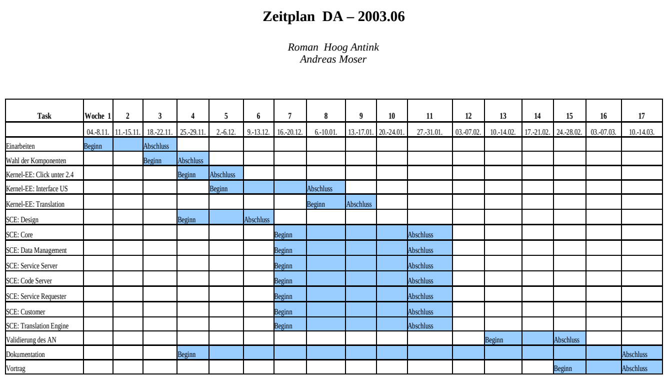
[6] Konstantinos Psounis. Active networks, applications,safety, security, and architectures. *IE-EE Communications Survey Magazine*, 2(1), 1999.

[7] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

Zürich, den 10.11.2002

Appendix E

Schedule



Zeitplan DA – 2003.06

Roman Hoog Antink
Andreas Moser

Task	Woche 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	04.-8.11.	11.-15.11.	18.-22.11.	25.-29.11.	2.-6.12.	9.-13.12.	16.-20.12.	6.-10.01.	13.-17.01.	20.-24.01.	27.-31.01.	03.-07.02.	10.-14.02.	17.-21.02.	24.-28.02.	03.-07.03.	10.-14.03.
Einarbeiten	Beginn	🟦	Abschluss														
Wahl der Komponenten			Beginn	Abschluss													
Kernel-EE: Click unter 2.4				Beginn	Abschluss												
Kernel-EE: Interface US					Beginn	🟦	🟦	Abschluss									
Kernel-EE: Translation								Beginn	Abschluss								
SCE: Design				Beginn	🟦	Abschluss											
SCE: Core							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Data Management							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Service Server							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Code Server							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Service Requester							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Customer							Beginn	🟦	🟦	🟦	Abschluss						
SCE: Translation Engine							Beginn	🟦	🟦	🟦	Abschluss						
Validierung des AN													Beginn	🟦	Abschluss		
Dokumentation				Beginn	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	Abschluss
Vortrag															Beginn	🟦	Abschluss

Index