Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Christoph Jossi, Mischa Weise

Communication and Management Infrastructure for Services on Active Router

Student Thesis SA-2004-27
Summer Term 2004

Tutor: Matthias Bossardt

Supervisor:
Prof. Dr. Bernhard Plattner

12.7.2004

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Zusammenfassung

Aktive Knoten zeichnen sich dadurch aus, dass sie zusätzlich zum Routen von Paketen flexible, so genannte aktive Services anbieten. Diese Services bestehen aus einem Graph von einzelnen Komponenten, welche in Execution Environments (EE) ausgeführt werden. Verschiedene EEs sind für bestimmte Aufgaben einer Komponente verschieden gut geeignet. Meist besteht deswegen ein Service aus Komponenten, die in verschiedenen EEs ausgeführt werden. Aus diesen Gründen ist die Kommunikation zwischen den EEs essentiell für eine flexible Servicegestaltung.
Diese Arbeit befasst sich zum einen mit Komponenten zum Management von aufgesetzten Services auf einem Aktiven Knoten, sowie zum anderen mit der Einführung eines Managements der Inter-EE-Kommunikation und der Implementation einer Inter-EE-Kommunikationsmethode über Netlink-Sockets.
Bis anhin konnte ein einmal auf einem Aktiven Knoten aufgesetzter Service zur Laufzeit weder reparameterisiert werden, noch wieder vom Knoten entfernt werden. Diese Arbeit beschreibt und implementiert ein Repository, welches sich die wichtigen Informationen eines Services merkt, sodass er jederzeit reparameterisiert oder entfernt werden kann. Gleichzeitig dient dieses Repository auch zur Verwaltung von servicebezogenen Ressourcen.

Inhaltsverzeichnis

Liste der Abbildungen und Tabellen

Danksagung

Wir möchten unserem Betreuer Matthias Bossardt danken für die Stunden, welche er für unsere Arbeit aufgewendet hat.

Kapitel 1

Einleitung

Aktive Netzwerke sind Netzwerke, welche programmierbare Router, so genannte Aktive Knoten[1], beinhalten. Der Zweck eines Aktiven Knoten ist die Bereitstellung von Services für den User, welche eine frei konfigurier- und programmierbare Paketverarbeitung beinhalten. Typische Anwendungen sind Paketfilterung, Load-Balancing, Web-Caches, Video-Scaling, Intrusion-Detection und vieles mehr.
Ein Service selbst besteht aus einem Graphen von so genannten Service-Komponenten, die elementare Teilaufgaben erfüllen – Abb. 1.1 veranschaulicht dies:

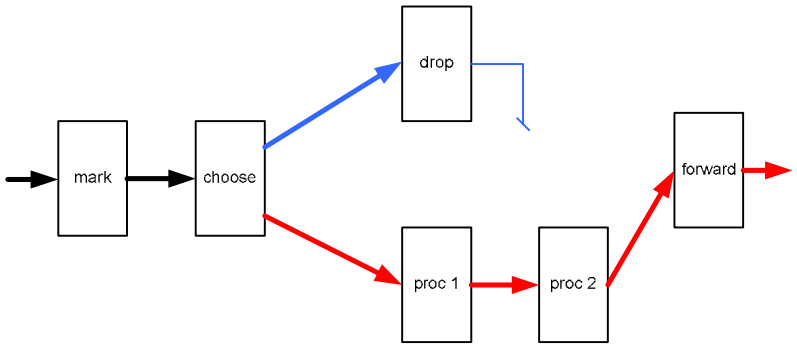

Abbildung 1.1: Schematisches Servicebeispiel

Der Programmcode, welcher die Service-Komponenten darstellt, wird dynamisch aus dem Netzwerk geladen und auf dem Knoten in so genannten Execution Environments (EEs), also Ausführungsumgebungen, ausgeführt. In Abb. 1.2 wird der Aufbau eines Aktiven Knoten dargestellt: Die Pakete werden von der Hardwareebene an den Demultiplexer (Demux) weitergeleitet, welcher die Aufgabe hat, Pakete an den entsprechenden Zielservice umzuleiten.

[1] Hier oft auch mit der Englischen Bezeichnung Active Node oder kurz Node erwähnt.

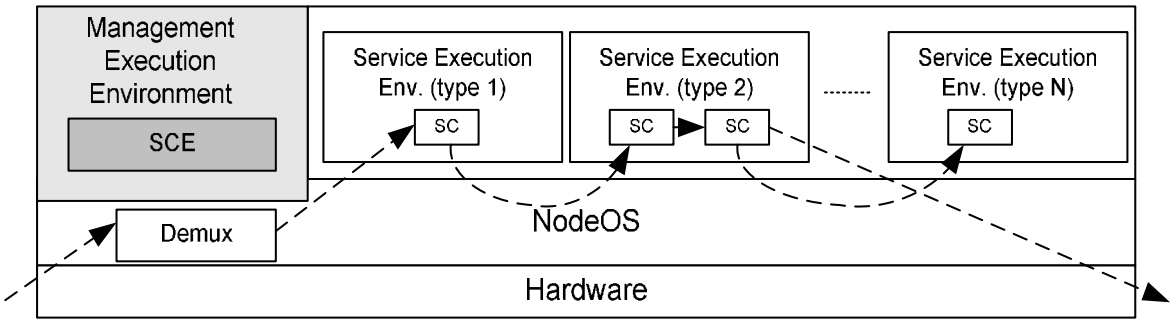

Abbildung 1.2: Aufbau eines Aktiven Knoten

Ein bestehender Service wird zur Laufzeit auf einen Benutzer-Request hin aufgesetzt; das geschieht durch Senden eines Service-Requests an den Aktiven Knoten. Damit auf diesem Knoten Services aufgesetzt werden können, muss er mit einer Service-Deployment-Architektur ausgestattet sein. Dieser Arbeit liegt der Chameleon Node zu Grunde, welche am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich entwickelt wurde:
In einer Management Ausführungsumgebung wird die Service Creation Engine (SCE) ausgeführt. Sie wird aktiv, wenn ein Service-Request beim Knoten eintrifft und installiert die Komponenten des angefragten Services, sofern dies machbar ist. Ein solcher Service-Request ist in XML geschrieben und knotenunabhängig. Die SCE hat die Aufgabe diesen Request auf die Architektur des Knoten abzubilden:

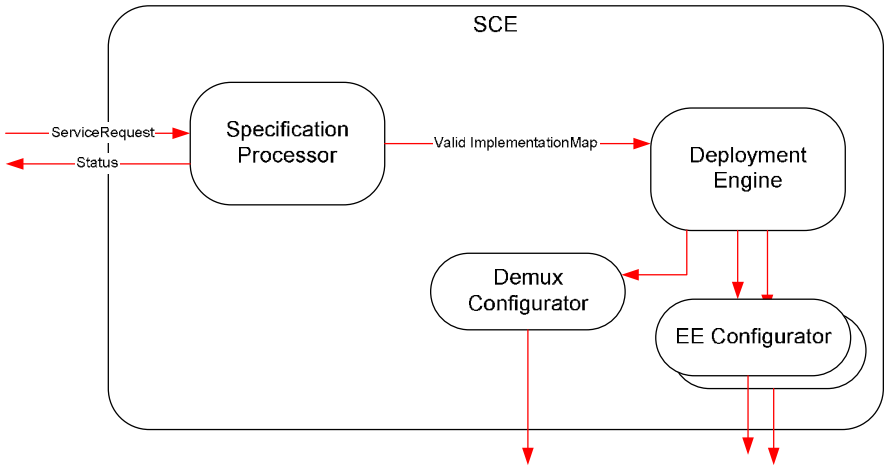

Abbildung 1.3: vereinfachte Deployment-Struktur

Der Specification Processor, ein Teil der SCE, evaluiert welche Service-Komponenten erforderlich sind, um den verlangten Service auf dem Knoten zu installieren und ob diese Komponenten überhaupt auf dem Knoten ausgeführt werden können. Das Resultat dieser Verarbeitung ist die Implementation Map, welche eine

validierte Abbildung des angeforderten Services auf die Knoten-Architektur ist. Daraufhin lädt ein anderer Teil der SCE, die DeploymentEngine, anhand der ImplementationMap den Code der identifizierten Komponenten in die jeweiligen EEs und konfiguriert die EEs mit Hilfe der EE-Konfiguratoren entsprechend. Wie ganz genau ein Service auf einem Aktiven Knoten aufgesetzt und ausgeführt wird, lässt sich in [da] nachlesen.

Die Verwendung mehrerer EEs macht eine effiziente und verlässliche Kommunikation zwischen den EEs erforderlich. Diese Inter-EE-Kommunikation ist daher wichtig, damit ein Service mit Komponenten in verschiedenen EEs überhaupt seine Aufgabe erfüllen kann.

In dieser Arbeit haben wir das Management der Inter-EE-Kommunikation ausgebaut um diverse Kommunikationsmethoden zu unterstützen. Des Weiteren haben wir eine dieser Methode, nämlich die Kommunikation über Netlink-Sockets, implementiert.
Gleichzeitig haben wie die Management Umgebung um ein Repository erweitert, welches Informationen über aufgesetzte Services speichert und Konzepte entworfen, wie man diese Informationen benützen kann. Ein Reparameterisierer erlaubt es einen Service zur Laufzeit mit neuen Parametern zu versehen und ein Remover ermöglicht es einen Service wieder vom Aktiven Knoten zu entfernen.

Kapitel 2

Erweiterung der Chameleon - Architektur

Dieses Kapitel beschreibt kurz die Chameleon-Architektur, welche erweitert werden soll. Daraufhin wird beschrieben, welche Erweiterungen der Chameleon-Architektur ins Auge gefasst wurden und aus welchem Grund.

2.1 Ausgangslage

In der gegenwärtigen Implementation beinhaltet Chameleon neben der Management EE mit der SCE noch ein Java-basiertes und eine Kernel-basiertes EE. Die Java-basierte EE ist leistungstechnisch nicht hinreichend für Operationen auf Transportebene. Für diese Aufgabe ist eine Kernel-basierte EE viel besser geeignet. Als Kernel-basierte EE wurde der Click Modular Router [click] gewählt. Die Click EE ermöglicht die einfache Konfiguration eines Software-Routers. Sie führt Code im Kernelspace aus und stellt eine grosse Anzahl von Service-Komponenten zur Verfügung (in Click-Terminologie: Elemente).
In den ursprünglichen Implementationen von Chameleon vor unserer Arbeit, wurde noch kein Schwerpunkt auf das Management von aufgesetzten Services gelegt. Ein Service wird durch ein Service Request angefordert und, sofern möglich, auf dem Node automatisch aufgesetzt und ausgeführt. Daraufhin besteht zur Laufzeit des Nodes keine Möglichkeit den Service zu stoppen, zu entfernen oder den laufenden Service mit anderen Parametern zu versehen. Es ist auch nicht möglich mehrere Services gleichzeitig auf dem Node laufen zu lassen. Kurz gesagt, es existieren keinerlei Informationen über einen Service auf dem Node, und ist ein Service einmal aufgesetzt, dann verhält sich der Node statisch bis er ausgeschaltet und neu gestartet wird.

Wie schon erwähnt unterstützt der Chameleon Node momentan zwei verschiedene EEs. Zum einen die JavaEE und zum anderen die ClickEE. Durch den Gebrauch von Komponenten in verschiedenen EEs können die Services optimiert werden. Damit ein

Service reibungslos ablaufen kann, ist es wichtig, dass auch die Kommunikation zwischen Service-Komponenten, welche sich in verschiedenen EEs befinden sichergestellt ist. Momentan unterstützt der Chameleon Node keine allgemein brauchbare Technologie, um zwischen den EEs zu kommunizieren. Ungenügende Lösungen über das proc-Filesystem und über Netlink-Sockets wurden bereits ansatzweise getestet. Bisher galt aber, auch die Inter-EE-Kommunikation wird in einer Service-Request angefordert. Dies ist natürlich nicht praktisch, denn man weiss im Vornherein ja nie, wo welche Service-Komponenten ausgeführt werden und kann somit auch noch nicht festlegen, wo Inter-EE-Kommunikation gebraucht wird. Auch widerspricht es dem Sinn des Node-unabhängigen Service-Requests.

2.2 Node Management

Wie erwähnt fehlen der SCE Informationen über laufende Services und genaue Vorgänge in den EEs. Dies äussert sich hauptsächlich in zwei Punkten: Erstens sind keine Informationen über aufgesetzte Services vorhanden und zweitens kann die SCE die begrenzten Ressourcen der EEs nicht kontrollieren. Die Nachteile dieser Situation sollten offensichtlich sein.
Beim Aufsetzen eines Services müssen nun alle wichtigen Informationen gesammelt und in einer vernünftigen Struktur abgespeichert werden. Eine solche Struktur stellt das von uns implementierte Repository dar: Es liefert einerseits die Möglichkeit jegliche Ressource-Informationen abzuspeichern, sowie andererseits eine Datenstruktur, auf welche Services eindeutig abgebildet werden können. Das Sammeln und Abspeichern der Service-Informationen und die dadurch notwenigen Konvertierungen erledigt eine von uns neu geschrieben Klasse in der SCE.
Als neue Komponenten eines Node-Managements wollen wir dem User die Möglichkeit bieten mit Hilfe der Informationen im Repository mit laufenden Services zu interagieren. Das Ändern des Service, zum Beispiel das Anpassen an geänderte Bedingungen, soll einem User ermöglicht werden. In unserer Arbeit sind wir den ersten Schritt gegangen und bieten die Möglichkeit die Parameter eines Services zur Laufzeit zu ändern. Dazu haben wir Schnittstellen implementiert und das Konzept eines Reparameterisierers entwickelt, welcher einen Service zu seiner Laufzeit mit neuen Parametern versehen kann. Gleichzeitig haben wir uns überlegt, dass davon der Schritt zu einem Remover, welcher einen aufgesetzten Service vollständig vom Node entfernt, nur ein kleiner, aber dennoch nicht trivialer, Schritt ist.

2.3 Inter-EE-Kommunikation

Durch die Verwendung von mehreren EEs auf einem Node, können Services optimiert werden. Wenn nun ein Service aufgesetzt wird, welcher in verschiedenen EEs Service-Komponenten ausführt, so sehen wir uns mit dem Problem der Inter-EE-

Kommunikation konfrontiert. Bereits vor unserer Arbeit wurde die Notwendigkeit dieser Kommunikation erkannt und es fanden auch schon Bemühungen [sa, da] statt um dieses Problem zu lösen. Auf dem momentanen Node, welcher unter dem Betriebssystem Linux läuft, sind deshalb auch schon Kommunikation über das proc-Filesystem und über Netlink-Sockets betrieben worden. Die bisherigen Lösungsansätze sind jedoch nicht genügend. Die vorhandene Implementation ist nicht funktional aufgeteilt und auch nicht generell einsetzbar, da sie in die Service-Requests eingebettet ist. Unser Ziel war es, einen allgemein anwendbaren Algorithmus zu finden, mit welchem sich die Inter-EE-Kommunikation zwischen allen EEs und auf allen Nodes auflösen lässt. Wir wollten nicht speziell unseren Node unterstützen, sondern eine allgemeine Lösung finden, welche allen möglichen Anforderungen genügt.

Um unser Management der Inter-EE-Kommunikation zu testen, brauchen wir natürlich auch mindestens eine spezielle Kommunikationsmethode. Wie bereits angesprochen, existiert weder eine allgemein nutzbare Lösung über das proc-Filesystem noch über Netlink-Sockets; dies wollen wir ändern. Die Methode, die wir zur Verfügung stellen wollen, nutzt die Netlink-Sockets. Diese Sockets sind grundsätzlich eine Infrastruktur um Kommunikation zwischen Userspace und Kernelspace zu betreiben. Sie kann aber auch missbraucht werden, um nur im Userspace zu kommunizieren. Die vor unserer Arbeit implementierte Kommunikation über Netlink-Sockets nutzt die Eigenschaften von Netlink nicht korrekt und zusätzlich wollen wir natürlich, dass diese Kommunikationsmethode allgemein einsetzbar ist und auch den Ansprüchen der Inter-EE-Kommunikation genügt, wie wir sie bereits oben angesprochen haben.

Kapitel 3

Node Management

Dieses Kapitel beschreibt wie genau das Node Management erweitert wurde. Darin wird der Aufbau des Repository erläutert, seine wichtigsten Schnittstellen und die Interaktion mit der SCE aufgeführt. Weiter wird das Konzept eines Reparameterisierers aufgezeigt und dessen Schnittstellen und Anforderungen beschrieben, sowie notwendige Überlegungen zu einem Service-Remover aufgezeigt.

3.1 Architektur

Zu Beginn überfliegen wir nochmals kurz, wie ein Service auf einem Aktiven Knoten aufgesetzt wird: Ein Knoten-unabhängiger Service-Request (in XML) trifft ein, wird durch die SCE verarbeitet, validiert und in eine Node-spezifische Implementation Map umgewandelt. Mit deren Hilfe wird dann der Code der benötigten Service-Komponenten in die EEs geladen. Daraufhin konfigurieren die Konfiguratoren die Komponenten und ihre Verbindungen innerhalb der EEs.

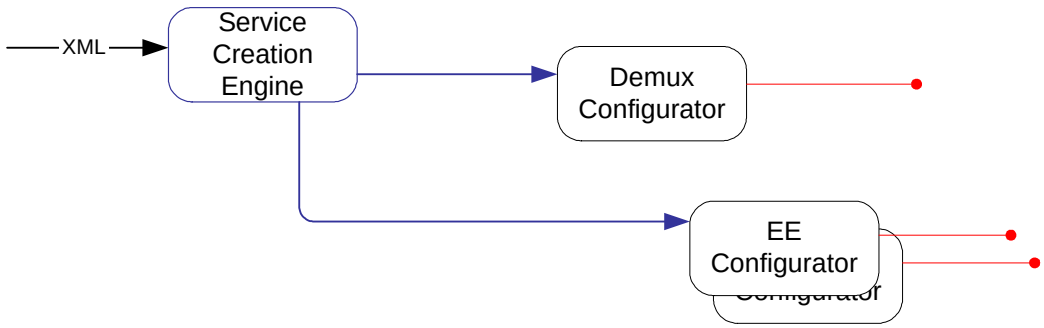

Abbildung 3.1: Ursprünglicher Zustand des Aktiven Knoten

Nun hat diese ursprüngliche Architektur in Abb. 3.1 ein grosses Defizit. Wie schon erwähnt, lässt sich ein einmal aufgesetzter Service nicht mehr ansprechen, also weder

entfernen, noch an neue Parameter anpassen. Dieses Defizit soll gelöst werden, indem der Node um ein Repository erweitert wird. Darin sollen, während der Installation eines Services durch die SCE, Informationen über den gerade zu installierten Service abgespeichert werden und während der ganzen Laufzeit des Services verfügbar sein. Dafür wurde die SCE durch die Helferklasse ToRepository erweitert, welche alle Interaktion zwischen der SCE und dem Repository erledigt. (siehe Abb. 3.2).

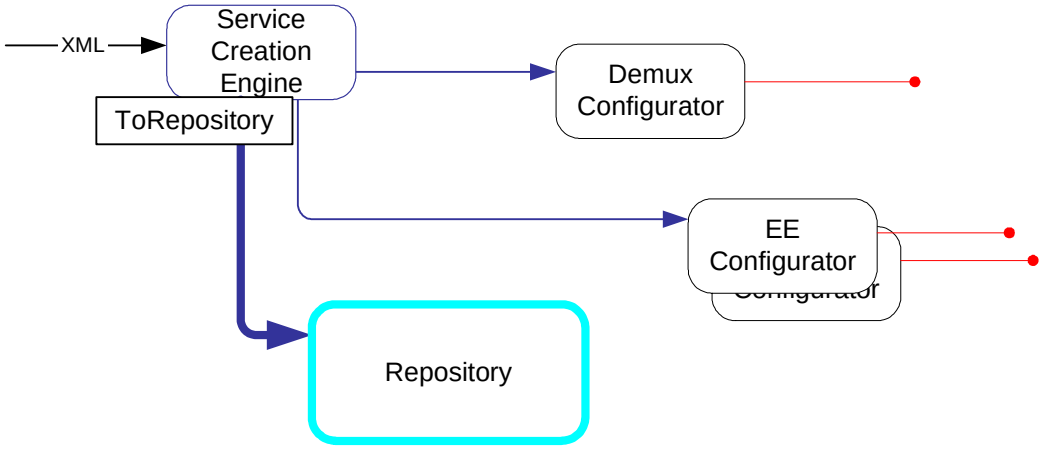

Abbildung 3.2: Erweiterung um ein Repository

Nun sollen diese im Repository gespeicherten Informationen natürlich auch angewendet werden. Der Reparameterisierer sollte mit Hilfe dieser Informationen in der Lage sein, die Parameter laufender Services zu ändern. Als Beispiel könnte der Scaling-Faktor bei einem Video-Scaling Service bei Überlastung des Netzes zur Laufzeit angepasst werden.

Für diese Erweiterung besteht die Notwendigkeit eines Message Handlers (siehe Abb. 3.3): Es muss entschieden werden, ob ein Service-Request in XML oder ein Reparameterisier-Request vorliegt. Dieser Request muss an die richtige Stelle weitergeleitet werden. Dazu ist die Ausarbeitung eines Protokolls nötig, welches einen Header mit Informationen für den Message Handler und eine Payload, sowie die Interaktion mit dem Absender des Requests vorschreibt. Diese Payload kann dann entweder ein Service-Request, ein noch in seiner Form zu bestimmender Reparameterisier-Request enthalten oder eine andere, neue Funktion des in Zukunft erweiterten Node-Managements ansprechen.

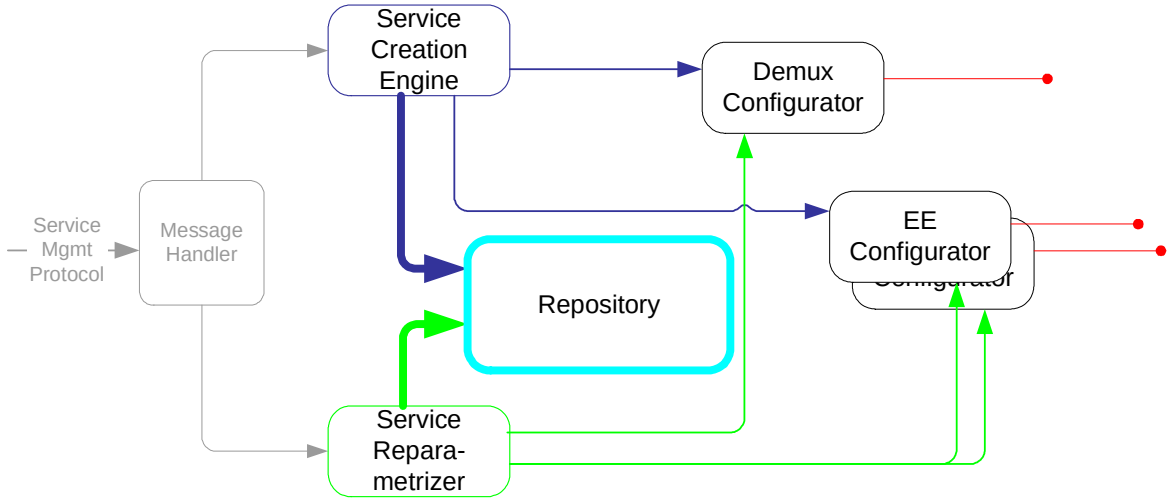

Abbildung 3.3: Management Erweiterungen und Message Handler

3.2 Repository

Das Repository wurde durch die Java-Klasse Repository implementiert. Diese Klasse muss zwei Funktionen erfüllen: Einerseits muss sie Informationen über Services speichern können, andererseits soll sie kritische Node-Ressourcen verwalten können.

Dazu kann sie eine Ansammlung von Service-Informationen, welche durch die Java-Klasse Service implementiert sind, aufnehmen. Ein bestimmter Service wird durch eine ihm zugewiesene und im Repository eindeutige ServiceID, abgespeichert und seine Information kann mit dieser ServiceID so jederzeit angesprochen werden

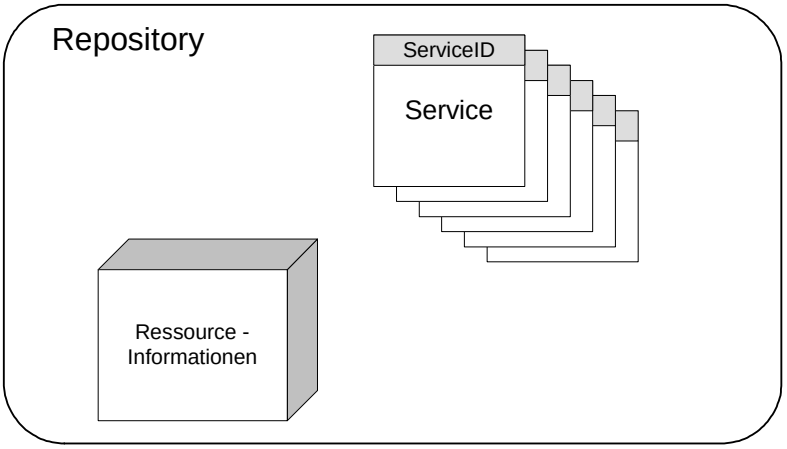

Abbildung 3.4: Aufbau des Repository

Ressource-Informationen werden abgespeichert, indem ein ganzes Objekt an das Repository übergeben wird, welches die Informationen enthalten muss. Das Objekt muss mit einem eindeutigen Namen bezeichnet werden und kann dadurch jederzeit angesprochen werden.

3.2.1 Ressourcenverwaltung

Die Idee hinter der Ressourcenverwaltung ist, dass sie sehr flexibel auf zukünftige Ansprüche anpassbar sein muss. Das Repository löst diese Anforderung, indem für jede beliebige Art von Ressourcenverwaltung generell Platz im Repository angeboten wird. An dieser Stelle muss ein zukünftiger Entwickler seine Ressourcen-Informationen als Java-Objekt abspeichern und wieder auslesen. Folgende Methoden werden für die Ressourcenverwaltung vom Repository zur Verfügung gestellt:

Signatur	Beschreibung
Object setRessourceInfo(String **type**, Object **obj**)	Speichert Ressource-Informations-Objekt ins Repository: **type** identifiziert den Typ der Ressource, muss eindeutig sein **obj** Ressourcen-Objekt, welches die Ressource-Informationen enthält *return* altes Objekt an dieser Stelle oder NULL, falls neuer Eintrag erstellt
Object getRessourceInfo(String **type**) throws RepositoryException	Holt Ressource-Informations-Objekt aus dem Repository: **type** identifiziert den eindeutigen Typ der Ressource *return* verlangtes Ressourcen-Objekt Excep. Falls kein Ressourcen-Objekt vom Type **type** vorhanden ist.

Tabelle 3.1: Interface der Ressourcenverwaltung

In Anhang B.2 stehen noch einige Tipps für das Einfügen einer neuen Ressourcen-Information.

3.3.2 Serviceverwaltung

Als erste Idee wurde von uns ins Auge gefasst, dass ganz simpel die ganze Implementation Map eines Services abgespeichert wird und mit Methoden zur Änderung von enthaltender Information versehen wird. Das wurde von uns insofern

verworfen, da die Klasse Implementation Map eine für diesen Zweck übermässig komplizierte Struktur besitzt. Andererseits werden durch unsere Implementation die meisten Informationen eins zu eins aus der Implementation Map ausgelesen und ins Repository gespeichert.
Das Problem bei der Serviceverwaltung ist, dass die EE-Konfiguratoren nicht in der gleichen Java-VM wie die SCE laufen, sondern auf einer anderen Java-VM – zu einem späteren Zeitpunkt eventuell sogar auf einem anderen Node. Die Verbindung zwischen diesen VMs geschieht via RMI. Das stellt uns insofern vor eine Herausforderung, dass in der einen VM die SCE und das Repository laufen und in einer anderen VM die Service-Komponenten aufgesetzt werden. Für die Interaktion mit laufenden Services müssen wir deshalb eine Möglichkeit haben, direkt auf Service-Komponenten in den EE zugreifen zu können. Dies geschieht mit einem Handler auf die instanzierte Service-Komponente in der EE. Dieser wird im Repository gespeichert. Somit hat man alle Informationen im Repository, die man braucht: Alle Informationen der ImplementationMap, sowie eine Möglichkeit direkt auf die Instanzen der Service-Komponenten zuzugreifen.

Im jetzigen Zustand, wo nur ein Service aufgesetzt werden kann, ist dieser Handler direkt der Wert der Variable ‚instanceName' der Service-Komponente. Dieser steht in der Implementation Map und kann dadurch als Vereinfachung schon vor dem Aufsetzen ins Repository gespeichert werden.
Bei einer Erweiterung der Node-Architektur, damit mehrere Services gleichzeitig laufen könnten, muss man auf folgendes achten:

- Entweder muss dieser Handler als Rückgabewert des Konfigurators an die DeploymentEngine zurückgegeben werden
- Oder der Konfigurator selbst muss diesen Handler ins Repository schreiben.

Unsere Präferenz ist klar die erste Variante. Die DeploymentEngine schreibt sowieso ins Repository und sie hat gleichfalls schon einen Kommunikationskanal mit den Konfiguratoren via RMI offen. So braucht man einerseits keine direkte Kommunikation via RMI zwischen Repository und Konfiguratoren zu implementieren und man muss sich andererseits keine Gedanken machen über mögliche Konflikte, wenn zwei Konfiguratoren auf den gleichen Service im Repository zugreifen.
Folgend noch eine Auflistung der wichtigsten Methoden, welche für die Service-Verwaltung im Repository zentral sind:

Signatur	**Beschreibung**
int addNewService()	Erstellt ein neues, leeres Service-Objekt im Repository *return* Eindeutige ServiceID mit welcher der neu erzeugte, leere Service im Repository angesprochen werden kann.

Signatur	Beschreibung
Service getService (int **id**) throws RepositoryException	Holt einen bestimmten Service aus dem Repository: **id** — Eindeutige ServiceID mit welcher ein gespeicherter Service im Repository angesprochen wird. *return* — Im Repository gespeicherter Service, welcher die ServiceID **id** trägt. Excep — Falls die ServiceID keinen Service bezeichnet
removeService (int id) throws RepositoryException	Entfernt einen bestimmten Service aus dem Repository: **id** — ServiceID des zu entfernenden Services Excep — Falls das Entfernen fehlgeschlagen ist oder die ServiceID keinen Service bezeichnet
setDeployed(int **id**) throws RepositoryException	Markiert den Service im Repository als aufgesetzt: Der Service ist vollständig und ohne Fehler auf dem Knoten aufgesetzt: **id** — ServiceID des zu markierenden Services **Excep.** — Falls die ServiceID keinen Service bezeichnet
boolean isDeployed(int **id**) throws RepositoryException	Überprüft, ob dieser Service vollständig und ohne Fehler auf dem Knoten läuft: **id** — ServiceID des Services Excep**.** — Falls die ServiceID keinen Service bezeichnet
Iterator getServiceIterator()	Liefert einen *Iterator*, mit dem über alle Services im Repository iteriert werden kann.
int totalDeployedServices()	Liefert die Anzahl der auf dem Knoten laufenden Services.

Tabelle 3.2: Interfaces zur Serviceverwaltung vom Repository

3.2.2.1 Service

Die Klasse Service stellt einen ganzen Service dar, wie er auf dem Node aufgesetzt ist. Dazu enthält sie alle serviceweiten Informationen eines Services. Sie enthält eine Reihe von Component-Klassen, die wiederum je eine Service-Komponente darstellt.

Hauptsächlich folgende Methoden werden zur Interaktion mit der Klasse Service verwendet:

Signatur	Beschreibung
ListIterator getIterator()	Liefert einen *ListIterator*, mit dem über alle Service-Komponenten iteriert werden kann.
int numComponents()	Liefert die Anzahl Service-Komponenten, welche dieser Service enthält.
setDemuxRules(Vector **demuxRules**)	Speichert die Demultiplexer-Regeln zu diesem Service. **demuxRules** Demultiplexer-Regeln
Vector getDemuxRules()	Liefert die Demultiplexer-Regeln zu diesem Service.
addComponent(Component **elem**)	Fügt Informationen über eine Service-Komponente zu einem Service hinzu: **elem** Component, welche zur Service-Information hinzugefügt werden soll
getComponent(String **instanceName**) throws RepositoryException	Liefert Informationen über eine Service-Komponente **instanceName** Bezeichnung der Service-Komponente Excep. Falls die Bezeichnung keine im Service inbegriffene Service-Komponente bezeichnet

Tabelle 3.3: Interfaces zum Abspeichern von Service Informationen

3.2.2.2 Component

Die Klasse Component enthält alle Informationen, welche für das Instanzieren einer Service-Komponente in ihrem EE benötigt werden.

Mit folgenden Methoden werden Informationen von Service-Komponenten abgespeichert oder ausgelesen:

Signatur	Beschreibung
Vector getParams()	Liefert die Parameter des Services als *Vector*, welcher zwei Vektoren enthält: 1. enthält alle Parameter-Namen 2. enthält die jeweils zugehörigen Parameter-Werte

Signatur	Beschreibung
setParam(Vector **paramNames**, Vector **paramValues**)	Speichert die Parameter des Services **paramNames** Vektor aller Parameter-Namen **paramValues** Vektor aller zugehörigen Werte
String getCodeLocation	Liefert den Ort mit dem Code der Service-Komponente mit folgender Syntax: [internet address]:/[path][filename] Beispiel: jee/queue.class
setCodeLocation(String **codeLocation**)	Speichert den Ort des Service-Komponenten Codes. **codeLocation** String mit Ort des Codes
Vector getConnections()	Liefert die Verbindungs-Informationen zu anderen Komponenten als *Vector*, welcher vier Vektoren enthält: 1. enthält alle incoming Ports 2. enthält alle outgoing Ports 3. enthält alle zugehörigen Typen zu 1. 4. enthält alle zugehörigen Typen zu 2.
setConnections(Vector **iP**, Vector **oP**, Vector **iPT**, Vector **oPT**)	Speichert die Verbindungs-Informationen zu anderen Komponenten dieser Service-Komponente **iP** alle incoming Ports **oP** alle outgoing Ports **iPT** alle zugehörigen Typen zu **iP** **oPT** enthält alle zugehörigen Typen zu **oP**

Abbildung 3.4: Methoden zur Abspeicherung von Service-Komponenten Information

In der nachfolgenden Tabelle stehen die Methoden, wie die Handler auf die in den EEs instanzierten Service-Komponenten abgespeichert und ausgelesen werden:

Signatur	Beschreibung
Object getInstanceHandler()	Liefert den Handler auf diese Service-Komponente, die in einem EE läuft. *return* Eindeutiger Handler auf Komponente in EE.

Signatur	Beschreibung
setInstanceHandler(Object **handler**)	Speichert einen Handler auf eine Service-Komponente, die in einem EE läuft. **handler** Eindeutiger Handler auf Komponente in EE

Abbildung 3.5: Methoden zum Abspeichern von Handlern auf Service-Komponenten

3.2.3 Interaktion zwischen Repository und SCE

Das Repository ist eine Einheit auf gleicher Ebene mit der SCE. Das heisst, das Repository wird gleichzeitig mit der Service Creation Engine instanziert. Daraufhin wird es bei den Aufrufen von Specification Processor und Deployment Engine weitergegeben. Somit ist abgesichert, dass immer und überall dasselbe Repository verwendet wird.
Die Interaktion zwischen der SCE und dem Repository übernimmt die Klasse ToRepository. Sie stellt die Funktionen ‚convert' und ‚revert' zur Verfügung:

Signatur	Beschreibung
int convert(ImplementationMap **implMap**, Repository **rep**) throws RepositoryException	Speichert die Informationen in der ImplementationMap ins Repository und liefert eindeutigen Handler auf die zugehörigen Service-Information im Repository. **implMap** Die Implementation Map des Services *return* Handler als ServiceID Excep. Falls beim Eintragen ins Repository ein Fehler passiert.
ImplementationMap revert(Repository **rep**, int **serviceID**) throws RepositoryException	Liest Informationen eines Services aus dem Repository und konvertiert diese in eine Implementation Map. **rep** Repository **serviceID** Rückzubildender Service *return* Eindeutiger Handler auf Komponente in EE. Excep. Falls beim Auslesen aus dem Repository ein Fehler passiert.

Abbildung 3.6: Methoden zum Abspeichern von Handlern auf Service-Komponenten

Das Abspeichern eines Service geschieht folgendermassen: Der Specification Processer beauftragt die Deployment Engine mit dem Aufsetzten des Services. Danach, falls keine Fehler aufgetreten sind, setzt der Specification Processer das Deployed-Flag beim von der Deployment Engine im Repository abgespeicherten Service. In der Deployment Engine selbst wird vor dem Aufsetzen des Services via die Konfiguratoren alle wichtige Information der Implementation Map ins Repository geschrieben und das war's auch schon.

3.3 Reparameterisierer

Wir haben uns überlegt, dass ein Reparameterisierer zwei Arten von Reparameterisieren unterstützen sollte:

- Direktes Reparameterisieren: Der Reparameterisierer ändert mit Hilfe der Handler im Repository die Parameter direkt bei der instanzierten, in ihrem EE laufenden Service-Komponente.
- Neuaufsetzten des Services: Der Reparameterisieren erzeugt eine Implementation Map aus den Informationen im Repository und ergänzt sie mit den neuen Parametern. Dann entfernt er den Service mit Hilfe des Removers und setzt den Service mit der angepassten Implementation Map neu auf.

Der Reparameterisierer sollte mit einer Kopie des Services arbeiten, damit, falls etwas schief läuft, nicht die Informationen im Repository verloren gehen. Nach der erfolgreichen Reparameterisation muss man dann nur noch diese Kopie mit den neuen Parametern ins Repository speichern.

3.3.1 Schnittstelle zu Repository

Alle wichtigen Schnittstellen zum Repository sind schon vorfolgend in der Beschreibung des Repository erwähnt worden. Es muss möglich sein aus den Informationen des Repositorys wieder die Implementation Map des Services zu generieren, dazu muss die ‚revert'-Methode in der Klasse ToRepository noch implementiert werden.

3.3.2 Schnittstelle zu Konfiguratoren

Eine Schnittstelle zu den Konfiguratoren braucht der Reparameterisierer um mit Hilfe der Handler auf die instanzierten Service-Komponente zuzugreifen oder um die reparameterisierte Implementation Map direkt an die Konfiguratoren zu schicken. Um dies ausführen zu können, brauch der Reparameterisator eine Kommunikation

via RMI mit den Konfiguratoren. Für diese Zwecke enthält die Basis-Klasse aller Konfiguratoren (Configurator.class) zwei abstrakte Methoden: ‚reparamFull' und ‚reparamSingle'. Diese Methoden müssen in den Konfiguratoren Configurator_jee.class und Configurator_click.class implementiert werden.

Falls einmal später noch zusätzliche EEs zu unserer Architektur aufgenommen werden, muss man die Beschreibung in [da] dahingehend erweitern, dass die Konfiguratoren dieser EEs natürlich obengenannte Funktionen auch implementieren müssen.

Kapitel 4

Management der Inter-EE-Kommunikation

In diesem Kapitel werden wir vorab nochmals kurz eine genaue Problemstellung dieser Teilaufgabe formulieren. Danach wollen wir eine kurze Diskussion über die Inter-EE-Kommunikation führen. Dabei richten wir unser Augenwerk auf die Anforderungen an eine solche Kommunikation. In der Folge stellen wir das gewählte Design vor, sowie die Entscheidungen, welche wir getroffen haben, um den Anforderungen gerecht zu werden. Im letzten Teil wollen wir noch kurz auf Probleme bei der Implementation zu sprechen kommen.

4.1 Problemstellung und Anforderungen

Der existierende Chameleon-Node kann momentan nur beschränkt Inter-EE-Kommunikation betreiben. Dabei liegen die Probleme vor allem in der ungenügenden Implementation von Kommunikationsmethoden, wie auch in der fehlenden Funktionalität der SCE. So wird der Inter-EE-Kommunikation keinerlei Beachtung geschenkt und in der Folge auch nicht speziell behandelt. Wir wollen nun die SCE um diese Funktionalität erweitern, um danach auf allen möglichen Nodes die Kommunikation zwischen den EEs sicherzustellen. Es ist an dieser Stelle wichtig zu begreifen, dass wir in diesem Kapitel nicht die vielen verschiedenen betriebssystemspezifischen Kommunikationsmethoden besprechen wollen, wie beispielsweise Netlink-Sockets. Vielmehr sind wir an einer allgemeinen Lösung interessiert, welche alle diese Kommunikationsmethoden unterstützt, sie jedoch nicht selbst implementiert. Dabei sei auf das nächste Kapitel verwiesen, in welchem wir eine spezifische Methode implementieren.

Wie bereits erwähnt, wollen wir alle denkbaren Kommunikationsmethoden unterstützen. Die in die SCE einzufügende Funktionalität kann logischerweise nicht alle diese Methoden implementieren. Dies beginnt schon bei der Anzahl der momentanen Möglichkeiten, sowie der Entstehung neuer Möglichkeiten in der

Zukunft. Dennoch wollen wir alle Methoden unterstützen. Somit hat jeder Betreiber eines Chameleon-Nodes die Möglichkeit, selber weitere Kommunikationsarten hinzuzufügen, ohne dass er dabei Änderungen an der SCE vornehmen muss.
Ein zweiter Punkt, welcher natürlich zur Debatte stand, war die Frage, wo wir die Funktionalität einbauen wollten. Wir setzten uns daraufhin das Ziel, möglichst wenig Änderungen am Code der bestehenden SCE zu machen.
Weiter wollten wir es einfach machen, neue Kommunikationsmethoden hinzuzufügen oder auch festzulegen, welche in welchem Fall verwendet werden soll.

4.2 Design

Zu Beginn beschäftigten wir uns mit der Frage, wo wir die zusätzlich notwendige Funktionalität in die SCE einbauen wollten. Für uns war klar, dass dies sicherlich nach dem Specification Processor ist. Schliesslich liefert er uns die validierte Implementation Map, nach welcher die Deployment Engine den Service generiert. Bis zu diesem Zeitpunkt spielt es noch keine Rolle, wie kommuniziert werden soll. Ausserdem wollten wir auch nicht die Funktionalität der Deployment Engine nicht erweitern, da diese genau das kann, wofür ihr Name steht. Wir entschieden uns, die zusätzlich verlangte Funktionalität in eine separate Klasse einzufügen, welche gleich nach dem Specification Processor aufgerufen wird. Als wichtigster Parameter übergeben wir die Implementation Map, welche in der Folge modifiziert wird.

Um der Deployment Engine eine gültige Implementation Map zu überreichen, müssen die Kommunikationselemente wie Service-Komponenten aussehen. Ihre spezielle Eigenschaft ist, dass sie entweder nur einen Inport oder nur einen Outport haben. Diese speziellen Service-Komponenten wollen wir von nun an Adapter nennen. Für einen Kommunikationskanal wird nun also je ein Adapter in jede EE plaziert. Diese beiden Adapter kommunizieren miteinander über vorhandene Interprozesskommunikation, welche vom Betriebssystem zur Verfügung gestellt wird.

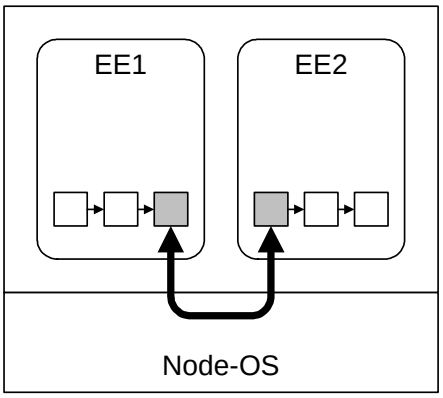

Abbildung 4.1: Inter-EE-Kommunikation läuft über das Betriebssystem ab

Wir haben bereits über die Notwendigkeit gesprochen, neue Kommunikationsarten einzufügen. Damit dies einfach möglich ist, wollen wir die Aufgaben, die schlussendlich von unserer Klasse erfüllt werden müssen in zwei Teile auftrennen. Zum einen gibt es den Teil, welcher von allen Methoden gebraucht wird. Auf der anderen Seite spezielle Funktionalität, welche ganz unterschiedliche Formen annehmen könnte. Somit muss man danach nur die Seite der speziellen Funktionalität zur Verfügung stellen und nicht weiteren redundanten Code hinzufügen. Zusätzlicher Vorteil dieser Variante ist, dass auf der Seite der speziellen Funktionalität auch spezielles Wissen über die Kommunikationsmethode vorhanden ist. Beispiele für allgemeine Funktionalität sind das Detektieren von Inter-EE-Kommunikation sowie das Laden eines Ressourcenverwaltungsobjektes aus dem Repository.
Nun stellte sich noch die Frage wie wir unserer Klasse mitteilen wollen, welche Kommunikationsmethoden auf diesem Node zur Verfügung stehen. Dafür eignet sich die Node-Description am besten. In Abbildung 4.2 sehen wir einen kurzen Auszug aus der Node-Description.

```
< xsi:EE_CONNECTIONS xsi:fromEE="CLICK" xsi:toEE="JEE">
        <xsi:CONNECTION xsi:fromPort="agnostic_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>
```

Abbildung 4.2: Auszug aus der Node-Description

Wie wir sehen können wird bereits hier beschrieben, welche Verbindungen zwischen den einzelnen Service-Komponenten möglich sind. Das kommt uns sehr gelegen. Denn bereits hier sehen wir die Definition der Inter-EE-Kommunikation (Zeilen 1-6).

Alles was wir nun noch zu tun brauchen, ist genauer zu definieren, wie diese Kommunikation stattfinden soll. Die Möglichkeit, welche wir vorschlagen und auch schlussendlich implementiert haben, sieht folgendermassen aus.

```
<xsi:fromEE="CLICK" xsi:toEE="JEE">
    <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="push_in" >
        <xsi:ADAPTER xsi:name="Netlink" xsi:configurator="Inter_EE_Com_Netlink">
            <xsi:PARAMETER xsi:name="netlink_family" value="7"/>
        </xsi:ADAPTER>
        <xsi:ADAPTER xsi:name="Procfs" xsi:configurator="Inter_EE_Com_Procfs"/>
    </xsi:CONNECTION>
</xsi:EE_CONNECTIONS>
```

Abbildung 4.3: Überarbeiteter Auszug aus der Node-Description

Wir sehen aus der bespielhaften Node-Description, dass die Kommunikation von einer Service-Komponente in der ClickEE zu einer in der JavaEE mit zwei Adaptertypen gelöst werden kann. Man könnte nun über Netlink-Sockets kommunizieren oder über das proc-Filesystem. Des weiteren sehen wir, dass wir noch Parameter übergeben können, welche vom Node abhängig sind. Mit Hilfe des Konfigurators können wir festlegen, von welcher spezifischen Klasse die Funktion addAdapter aufgerufen werden soll. Dies gibt uns die Möglichkeit mehrere verschiedene Funktionen zur Verfügung zu stellen. In Abbildung 4.4 sehen wir zusammengefasst die neue Node Description.

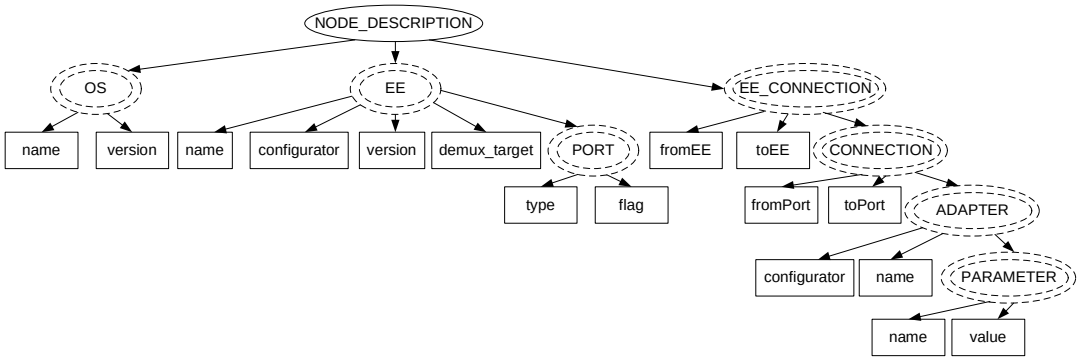

Abbildung 4.4: Graph der Node-Description

4.3 Implementation

Im Graphen in Abb. 4.6 sehen wir den Algorithmus, welcher von der Funktion SolveInterEE durchlaufen wird. Grundsätzlich werden wir von jeder Service-Komponente alle ausgehenden Ports durchlaufen und die Inter-EE-Kommunikation detektieren. Wurde so ein Kommunikationskanal gefunden, so interessieren wir uns für die möglichen Methoden, welche uns der Node zur Verfügung stellt um diese Kommunikation zu betreiben. Gibt es mehr als eine, so müssen wir natürlich eine Wahl treffen. Zusätzlich wird aus dem Repository eine Ressourcenverwaltungsobjekt geladen. In nächsten Schritt werden alle Parameter aus der Node-Description ausgelesen. Mit diesen zwei Schritten wollen wir verhindern, dass die spezifische Klasse Information über das Repository oder die Node-Description erhält, welche sie grundsätzlich nicht braucht. Im letzten Schritt wird dann noch die Funktion ‚addAdapter' der spezifischen Klasse aufgerufen.

Diese spezifische Klasse führt dann noch den Rest der anliegenden Arbeit aus. Die übrige Arbeit ist im folgenden Graphen in Abb. 4.5 nochmals kurz zusammengefasst. Wir möchten an dieser Stelle darauf hinweisen, dass der Graph keinerlei Verzweigungen enthält. So können wir die spezifische Klasse so einfach wie möglich gestalten. Sie hat nur die folgenden drei Aufgaben zu erfüllen. Als erstes muss sie die übergebenen Parameter parsen. Danach erzeugt die Klasse die Adapter nach den Regeln welche ihr bekannt sind. Im letzten Schritt fügt sie noch die Adapter in die Implementation Map ein und verbindet sie mit den Nachbarn. Wieso wir den letzten Teil nicht wieder in der Funktion SolveInterEE machen, wird im nächsten Kapitel klar werden.

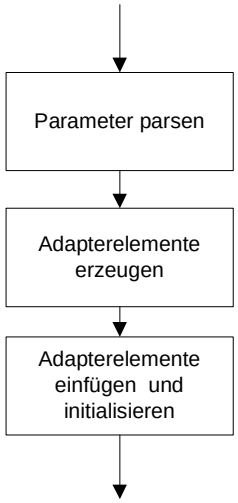

Abbildung 4.5: Aufgaben der Funktion addAdapter

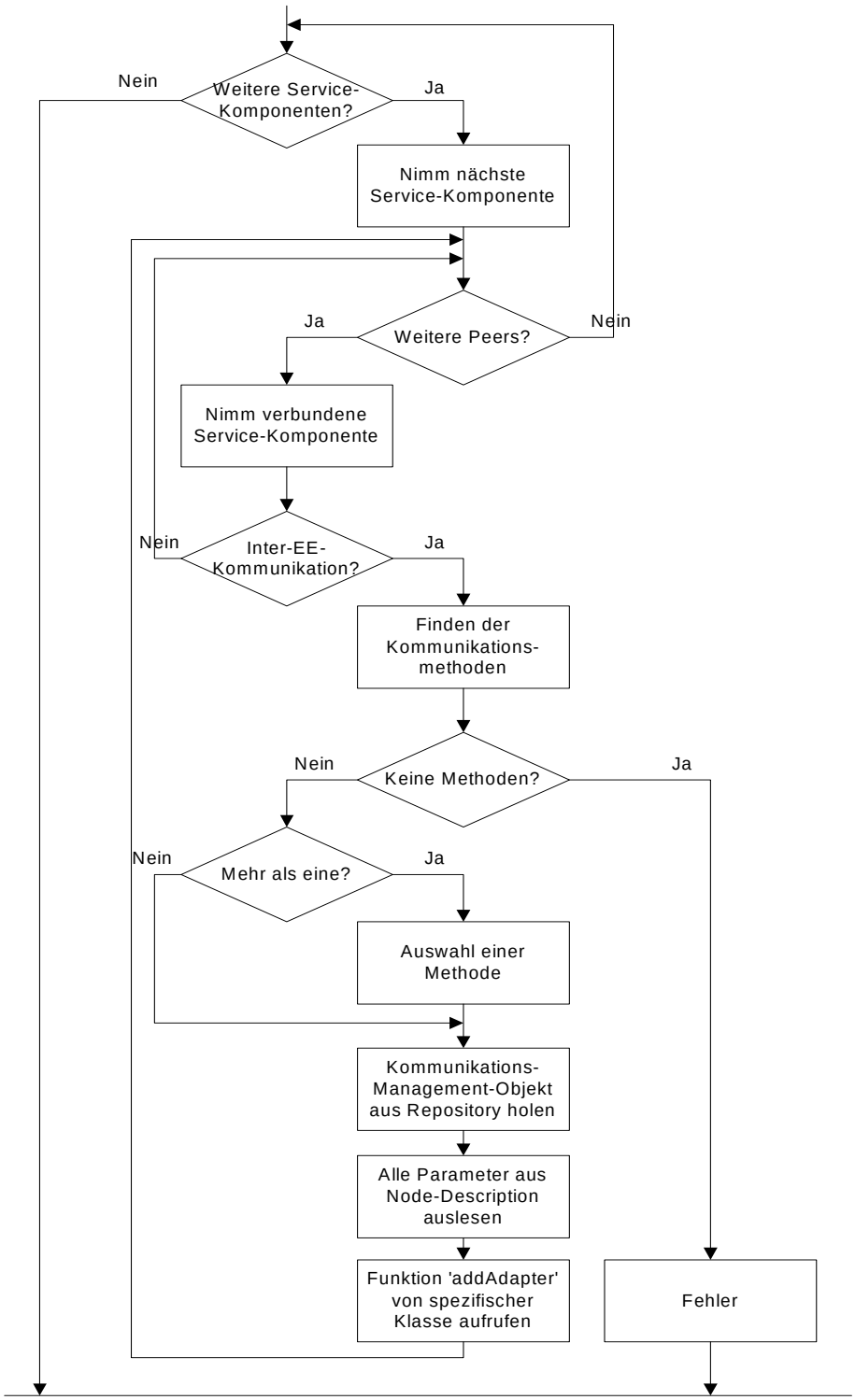

Abbildung 4.6: Algorithmus der Funktion SolveInterEE

Kapitel 5

Inter-EE-Kommunikation über Netlink-Sockets

In diesem Kapitel werden wir eine mögliche Implementation der Inter-EE-Kommunikation besprechen. Während wir zu Beginn nochmals kurz auf die Problemstellung eingehen, wollen wir nachher das Design festlegen, welches natürlich von den Eigenschaften der Netlink-Sockets abhängt. Mit der nachfolgenden Implementation wollen wir dann noch unsere Lösung präsentieren.

5.1 Problemstellung und Anforderungen

Auf unserem Node existieren momentan zwei verschiedene EEs. Im Userspace kann man mit der JavaEE arbeiten und im Kernelspace mit der ClickEE. Soll nun ein Service auf beide EEs verteilt werden, so brauchen wir eine Möglichkeit, zwischen den beiden Randkomponenten zu kommunizieren. Linux bietet uns für eine Kernelspace-Userspace-Kommunikation die Netlink-Sockets an. Mit Hilfe dieser Technologie wollen wir also die Kommunikation zwischen den EEs sicherstellen. Wir wollen eine Lösung finden, welche langfristig brauchbar ist. Das bedeutet, dass wir keine Einschränkungen im Bezug auf die Anzahl der Kommunikationskanäle haben wollen.

5.2 Design

Das Design ist schon weitgehend vorbestimmt. So haben wir bereits festgelegt, dass wir für jeden Kommunikationskanal jeweils einen Adapter in jeder EE platzieren wollen. Die Adressierung eines Netlink-Sockets besteht aus zwei Werten. Einer Netlink-Family und einer PID. Dies wird grundsätzlich als Process-ID referenziert, ist aber mehr im Sinne einer Port-ID zu verstehen. So muss denn auch das Paar (Netlink-Family;PID) einmalig sein. Diese Port-ID ist aber nicht vom eigentlichen Prozess abhängig. Per Definition kann pro Netlink-Family genau ein Socket im Kernelspace

bestehen, nämlich jener mit PID=0. Die Rolle der Netlink-Family kann folgendermassen definiert werden. Sie bestimmt das Protokoll, welches verwendet wird. Das bedeutet im Weiteren, dass wir eigentlich nur eine Netlink-Family verwenden sollten. Würden wir das nicht tun, so würden kostbare Ressourcen des Nodes verbraucht werden. Wir stehen also grundsätzlich vor dem Problem, nur einen Socket im Kernelspace einrichten zu können. Die folgenden Abbildungen zeigen uns die zwei verfügbaren Designs, welche uns zur Verfügung stehen.

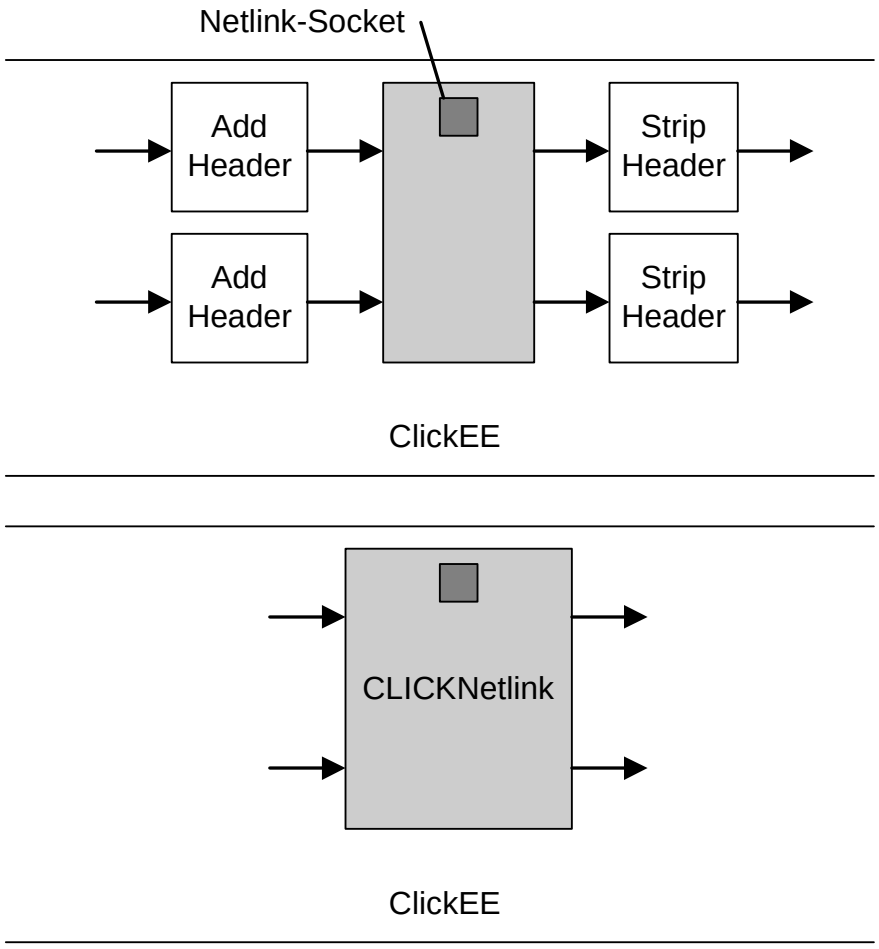

Abbildung 5.1: Zwei Designs für den Adapter in der ClickEE

Bei der ersten Variante wird ein zentrales Element (grau markiert) eingefügt, welches den Socket implementiert und gemäss dem vorher eingefügten Header (in AddHeader) das Packet weiterleitet. Ein eingehendes Packet wird ebenfalls gemäss eingefügtem Header (diesmal natürlich bereits in der JavaEE angefügt) auf einen Outport geleitet, wo dann in einem weiteren Element (StripHeader) der Header entfernt wird.

Bei der zweiten Variante wird das Mapping einer eingehenden Verbindung direkt vom zentralen Element ausgeführt. So werden beispielsweise alle Packete eingehend von Port 1 auf den Socket 4 geschickt. Ähnlich funktioniert der Ablauf beim Eintreffen eines Packetes aus dem Userspace. Dieses wird nun auf einen Outport gemappt.
Wir wollen nun Vor- und Nachteile der beiden Varianten besprechen. Der Vorteil der Variante eins ist, dass das zentrale Element keinerlei Aufgaben besitzt, ausser gemäss dem Header das Packet zu verschicken. Die vorhergehenden und nachfolgenden Elemente hingegen müssen aber wissen, wer ihr Peer ist. Die zweite Lösung entspricht mehr einer zusammengefassten Lösung. Alle drei Elemente werden in einem verpackt. Dadurch muss nun dieses zentrale Element alle Verbindungen kennen.
Wir haben uns für die kompaktere Lösung entschieden. Früher oder später muss ein Mapping stattfinden. Es schien uns nicht sinnvoll, mit zu vielen Elementen die Übersicht in der ClickEE zu beeinträchtigen. Anstatt das Mapping auf mehrere verschiedene Komponenten zu verteilen, ist es einfacher, diese in einem zentralen Element zu verwalten, wo sie dann auch einfach manipuliert werden können.
Da wir in der ClickEE sowieso nur ein Element haben, muss dieses die ganze Funktionalität von Push und Pull sowie senden und empfangen vornehmen. In der JavaEE hingegen können diese vier Funktionalitäten auseinander gehalten werden. Der Einfachheit halber werden wir vier dort verschiedene Service-Komponenten generieren, welche jeweils eine dieser Funktionalitäten einbinden.

5.3 Implementation

Die Implementation der Inter-EE-Kommunikation über Netlink-Sockets gestaltete sich ziemlich anspruchsvoll. Wir sahen uns konfrontiert mit den drei Programmiersprachen C, C++ und Java. Die erste kleine Herausforderung fand bei der Einbettung der Netlink-Sockets in die JavaEE statt. Da Java keine betriebssystemspezifischen Technologien unterstützt und somit auch die Netlink-Sockets nicht, mussten wir die Sockets in einer Shared Library, welche wir in C programmiert haben, einbinden. Die Funktionsaufrufe werden dabei über das Java Native Interface (JNI) gemacht. Um nur eine Shared Library einbinden zu müssen, wurde sämtliche Funktionalität im Bezug auf die Netlink-Sockets in diese Shared Library eingebunden. Beispielsweise muss ein Adapter, welches eingehende Pakete von der ClickEE weiterpusht, keine Messages senden können. Gemäss Shared Library ist das aber möglich.
Nun werden wir diese Shared Library kurz etwas genauer untersuchen. Um nicht ununterbrochen den Socket zu pollen, werden wir einen blockierenden readmsg-call machen. Dies bedeutet im Gegenzug, dass wir einen neuen Thread kreieren müssen. Da alle Kommunikations-Service-Komponenten die gleiche Shared Library benutzen, müssen nötige Informationen in zusätzlich alloziertem Memory gelagert werden. Der Pointer zu diesem Memory wird jeweils der Kommunikations-Service-Komponenten

zurückgegeben. Alsbald nun eine Message auf dem Socket ankommt, kommt der readmsg-call zurück. Die Message wird leicht manipuliert und danach wird sie der Funktion processMsg übergeben, welche ebenfalls wieder über das JNI aufgerufen wird. Was ab nun mit der Message geschehen soll, regelt die Service-Komponente. Um das Memory sowie die Sockets wieder freizugeben, muss die Funktion ‚uninitialise' aufgerufen werden. Diese existiert auch in den Kommunikations-Service-Komponenten und sollte bei einer Deinstallierung des Services aufgerufen werden.
Das Senden einer Message über den Netlink-Socket ist relativ simpel. So lösen eingehende Pakete auf der Kommunikations-Komponente den Aufruf der Funktion sendmsg der Library aus, welche ihrerseits die Funktion sendmsg des Sockets aufruft.

Die Adapter in der JavaEE haben so, wie sie momentan implementiert sind, nur einen Ausgang oder einen Eingang. Die Shared Library ist aber fähig auch Komponenten zu bedienen, welche mehrere Ein- und/oder Ausgänge haben. In den folgenden Abbildungen sehen wir die UML-Sequenz-Diagramme der bisher implementierten Adapter NlToJeePush und JeeToNlPush.

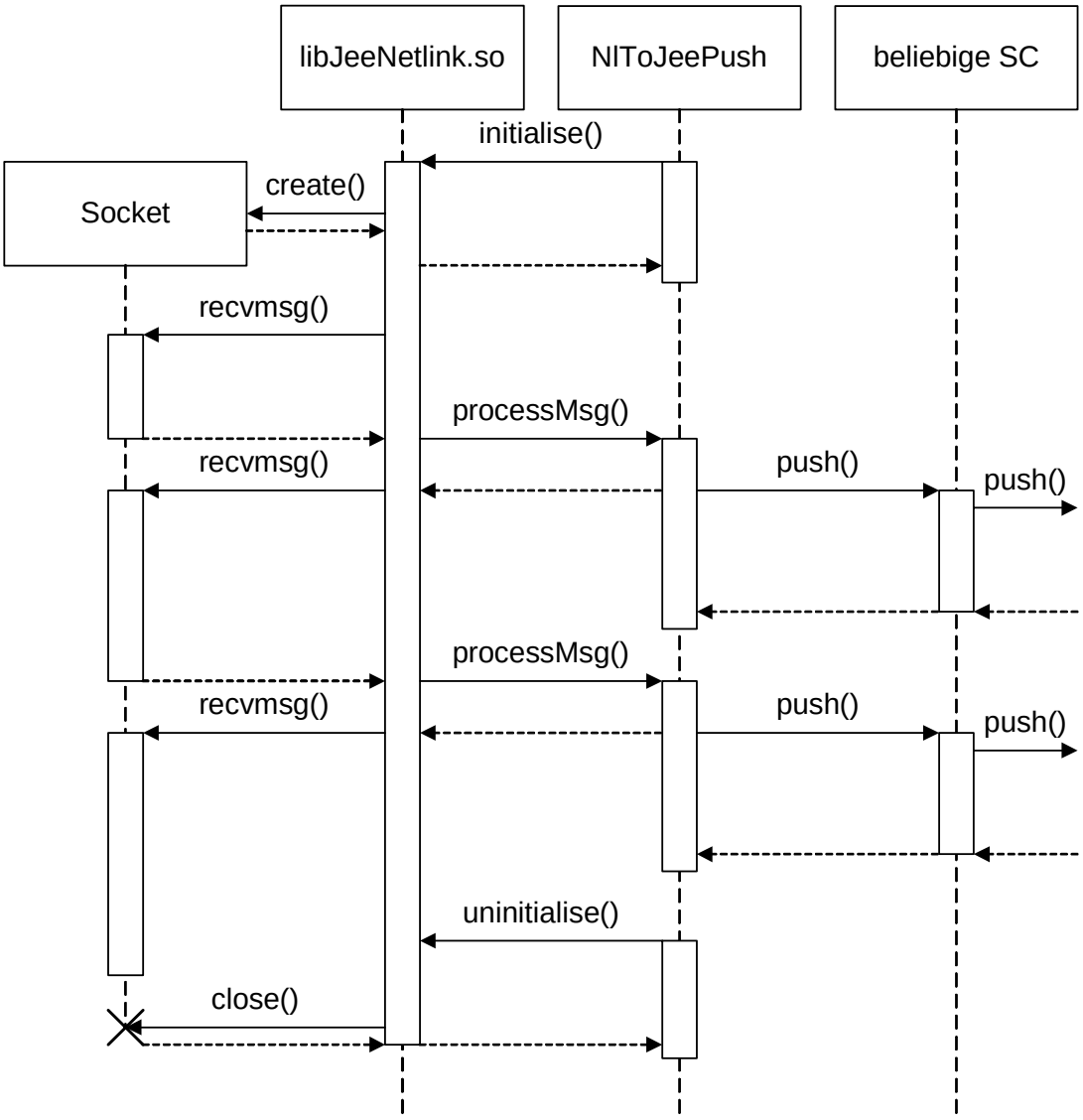

Abbildung 5.2: UML Sequenzdiagramm der Klasse NlToJeePush

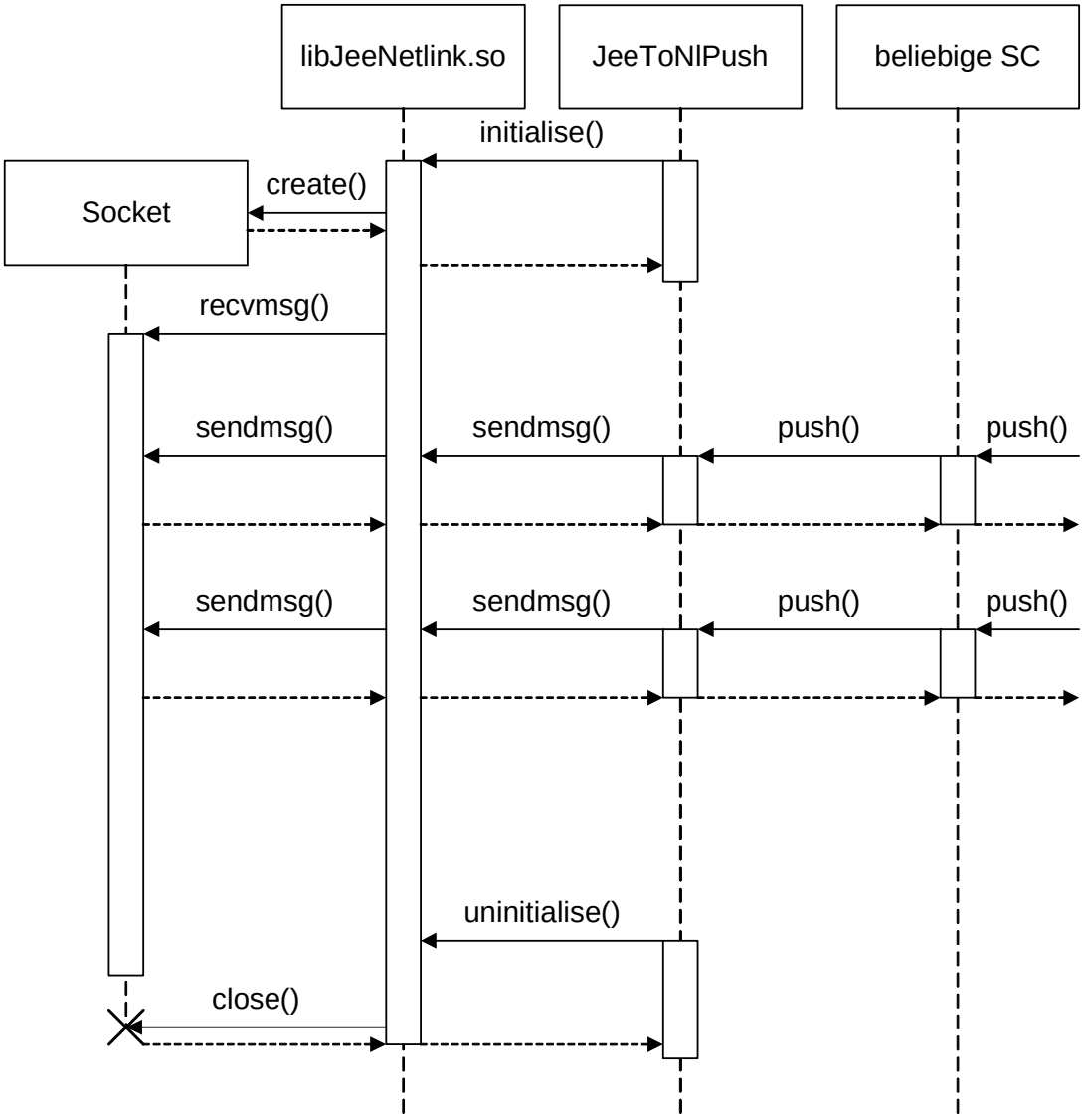

Abbildung 5.3: UML Sequenzdiagramm der Klasse JeeToNlPush

Um den Socket im Kernel zu realisieren, wurde er ebenfalls in einer Service-Komponente plaziert. Das Problem betreffend Mapping haben wir folgendermassen gelöst. Beim Konfigurieren wird die Netlink-Family festgelegt, wie auch alle Kommunikationskanäle. Ein Kommunikationskanal wird mit dem Tupel (fromPid, fromPort, toPid, toPort) eindeutig definiert. Auch diese Kommunikations-Service-Komponente unterstützt bislang nur push-Verbindungen, sie ist aber für die Erweiterung eingerichtet.
Werden nun diese Adapter in der ClickEE wie auch in der JavaEE eingefügt, so kann ein solcher Service danach folgendermassen aussehen:

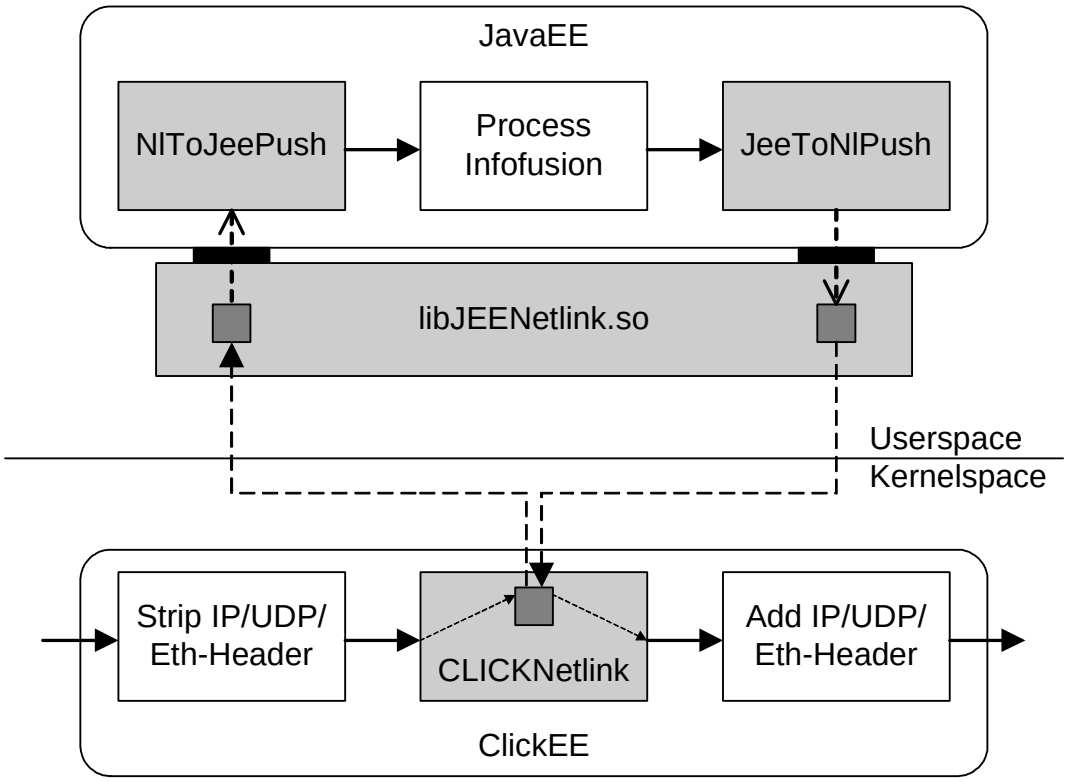

Abbildung 5.4: Einfacher Beispielservice

Probleme ergaben sich schlussendlich noch, wenn mehr als ein Adapter die gleiche Shared Library laden will. Das mehrmalige Laden einer Library hat grundsätzlich keine Auswirkung. Dies muss jedoch mit dem gleichen ClassLoader gemacht werden. Da nun jede Service-Komponente mit einem neuen ClassLoader erzeugt wurde, mussten wir den JEEServer anpassen. Nachdem auch dieses Problem gelöst war, funktionierten die Adapter einwandfrei. Im Anhang weisen wir dies auch noch mit einem Beispielservice aus, welcher das Beispiel aus Abbildung 5.4 noch ein bisschen erweitert.

Kapitel 6

Fazit und zukünftige Arbeiten

6.1 Fazit

Die Implementation des Repository und seine Integration in die bestehende Architektur sind ausgeführt und das Abspeichern eines Services ins Repository funktioniert. Nun muss das Repository noch seine Tauglichkeit in einer Anwendung aufzeigen. Dazu sind Konzept und Überlegungen zu einem Reparameterisierer und seine Interaktion mit der bestehenden Architektur ausgearbeitet und liegen in dieser Dokumentation vor.

Die Detektion der Inter-EE-Kommunikation, sowie das Einfügen der Adapterelemente funktioniert reibungsfrei. So werden alle Service-Komponenten richtig miteinander verbunden. Auch der später beschriebene Beispielsservice lief ohne Probleme. Diese entstanden dann, wenn ein weiterer Service angefordert wurde. Auf der einen Seite lief noch ein Service, welcher Ressourcen von Netlink benötigte und auf der anderen Seite wurde bereits weitere Ressourcen vergeben. Bei den Ressourcen an sich gab es noch keine Überschneidung, wohl aber später beim Aufsetzen eines Services, als der erste installierte einfach abgeschossen wurde. Deshalb haben wir in der Klasse Solve_Inter_EE_Communication zusätzlichen Code im Konstuktor eingefügt. Hier muss noch eine saubere Lösung gefunden werden, um den einen Service sauber zu entfernen. Dazu braucht es möglicherweise noch einmal einen Eingriff in die Klasse Solve_Inter_EE_Communication, bzw. einen Aufruf dieser Klasse beim Entfernen von Services. Solange weiterhin automatisch ein Service gelöscht wird, wenn der nächste angefordert wird, ist es schwierig, genauere Aussagen zu machen.

6.2 Zukünftige Arbeiten

- In einer direkt an unsere Arbeit folgende Arbeit könnte der Reparameterisierer implementiert und mit einem Beispielservice getestet werden. Für eine vollständige Funktionalität nach unserm Konzept müssten dabei auch noch ein Management Protokoll und ein Management Handler erarbeiten werden.
- Jetzt da die Inter-EE-Kommunikation reibungslos via Netlink-Sockets läuft könnte man einen Service implementieren, welcher diese Möglichkeiten voll ausschöpft.
- Da die Inter-EE-Kommunikation noch keine Pull-Verbindungen unterstützt, könnte man solche Adapter programmieren und Anpassungen vornehmen, damit diese unterstützt werden.
- Man könnte weitere Inter-EE-Kommunikations Methoden hinzufügen wie zum Beispiel das proc-Filesystem
- Last but not least: Es könnte mal ein wenig aufgeräumt werden.
    - Entfernen von nicht gebrauchtem Code. (zB: Code um die Inter-EE-Kommunikation nach früherer Methode zu realisieren, vor allem in JEEServer und ClickServer)
    - Sauberes Install-Skript schreiben
    - …

Literaturverzeichnis

[da] R. Hoog Antink and A. Moser. Service composition for active networks. Master's thesis, ETH Zurich, Switzerland, 2003.

[click] 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.

[sa] Julio Perez. Active network service mit netlink. Semester thesis at ETH Zurich, Switzerland, 2004.

[netlink] Gowri Dhandapani, Anupama Sundaresan. Netlink Sockets – Overview. http://qos.ittc.ukans.edu/netlink/html/netlink.html.

Anhang A

Installation Guide

This installation guide is manly from [da], with our comments added:

A.1 Requirements

- Linux Kernel 2.4.20 from http://www.kernel.org/
- Click router from http://www.pdos.lcs.mit.edu/click/ (CVS) Click 1.4pre
- 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/kernel-2.4.20. Make the following an symbolic link
    a. from /usr/src/linux to the directory
    b. /usr/include/linux -> /usr/src/linux/include/Linux
    c. /usr/include/asm -> /usr/src/linux/include/asm-i386
2. Unpack also Click and Iptables sources
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.
5. Configure the kernel making sure the Netfilter CLICK-Target and both the MARK-Target and the MARK-match are enabled as module, compile and install new kernel.
6. Patch the Iptables userspace tools with the click netfilter patch.
7. Build and install new Iptables userspace tools.
8. Boot the new kernel.

9. Copy fromnetfilter.{hh,cc} and clicknetlink.{hh,cc} into click/elements/local and copy clicknetlink.h into the linux header directory
10. Use Version 2.95 of the g++-Compiler: “export CXX=g++-2.95”. Configure, build (make install) and install click (click-install). 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 (beta 10) from http://www.jdom.org.
3. Find the java source code in our archive at an/java.
4. Get and install xerces (2.6.2) from http://xml.apache.org.
5. Get and install the XPath evaluator jaxen (1.0-FCS) from http://jaxen.sourceforge.net/
6. For Netlink-Socket Inter-EE-Kom. Copy libJEENetlink.so into /usr/local/lib/
7. Adjust your CLASSPATH environment variable (in the run-scrips and the makefile) so that it contains those jar archives and the top directory of the java source code.
8. Adapt ch/ethz/ee/tik/chameleon/util/{HostConstants,Constants}.java to your needs. You should adapt HostConstants.java to reflect your file paths. Make sure CODE_SERVER_BINARYS_PATH points to the directory where the JEE services are. This should be an/java/jee/services.
9. Build java software, using the makefile in the java top directory. Simply type make.

Anhang B

User Guide

B.1 Erstellen eines Reparameterisierers

- Der Reparameterisierer sollte als neues Modul implementiert werden, daher muss man die Liste der Module im Makefile erweitern.
- Implementation der Methode ‚revert' in ToRepository.class für das Reparameterisieren mit Neuaufsetzen.
- Implementation der Schnittstellen mit den Konfiguratoren für das direkte Reparameterisieren.

Daraufhin muss natürlich noch die Funktion des Reparameterisierers selbst implementiert werden und sich die genauen Details der Implementation überlegt werden:

- Wie sieht ein Reparameterisier-Request aus?
- Wie wird entschieden welche Art des Reparameterisierens angewendet wird?
- …

B.2 Einfügen von Ressourcen-Information ins Repository

- Mit der Hilfe der Methoden *setRessourceInfo* und *setRessourceInfo* ist es ein leichtes ein Java-Objekt, welches die Ressourcen-Informationen enthält ins Repository einzufügen und daraufhin wieder auszulesen.
- Bei Programmen, welche nicht in Java geschrieben sind, muss zusätzlich noch eine Java-Klasse geschrieben werden, welche die Ressource-Informationen in ein Java-Objekt umwandelt und mit dem Programm interagiert.

B.3 Integrieren einer neuen Inter-EE-Kommunikations Methode

Um eine neue Kommunikationsmethode zu unterstützen, muss folgendermassen vorgegangen werden. Wir beginnen mit dem Eintrag in der Node-Description. Beispielsweise wollen wir nun die Kommunikation über das proc-Filesystem hinzufügen. Diese Kommunikation soll von Push-Komponenten in der ClickEE nach Push-Komponenten in der JavaEE führen. In den Abbildung sehen wir Ausschnitte der Node-Description vor und nach der Modifikation.

```
< xsi:EE_CONNECTIONS xsi:fromEE="CLICK" xsi:toEE="JEE">
        <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="push_in" />
        <xsi:CONNECTION xsi:fromPort="push_out" xsi:toPort="push_in" />
</xsi:EE_CONNECTIONS>
```

Abbildung B.1: Ausschnitt der Node-Description vor der Modifikation

```
<xsi:fromEE="CLICK" xsi:toEE="JEE">
    <xsi:CONNECTION xsi:fromPort="agnostic_out" xsi:toPort="push_in" >
        <xsi:ADAPTER xsi:name="Procfs" xsi:configurator="Inter_EE_Com_Procfs"/>
    </xsi:CONNECTION>
    <xsi:CONNECTION xsi:fromPort="push_out" xsi:toPort="push_in" >
        <xsi:ADAPTER xsi:name="Procfs" xsi:configurator="Inter_EE_Com_Procfs"/>
    </xsi:CONNECTION>
</xsi:EE_CONNECTIONS>
```

Abbildung B.2: Ausschnitt der Node-Description nach der Modifikation

Die gemachten Einträge bedeuten nun folgendes. Es existiert eine Möglichkeit der Inter-EE-Kommunikation zwischen einer Push-Komponenten in der ClickEE und einer Push-Komponenten in der JavaEE. Der Name dieser Kommunikationsmethode ist Procfs. Nun muss eine Klasse, welche den Names des Konfigurators trägt, in diesem Falle "Inter_EE_Com_Procfs" die Funktion ‚addAdapter' zur Verfügung stellen, welche folgende Signatur hat:

addAdapter(ImplementationPort outport,
ImplementationPort inport,
LinkedList parameterNames,
LinkedList parameterValues,
ProcfsRessourceMgmt mgmt)

oder

addAdapter(ImplementationPort outport,
ImplementationPort inport,
LinkedList parameterNames,
LinkedList parameterValues)

Die Bedeutung dieser Parameter ist in der folgenden Tabelle zusammengefasst:

Name	**Typ**	**Bedeutung**
outport	ImplementationPort	outport der ClickEE-Komponente
inport	ImplementationPort	inport der JavaEE-Komponente
parameterNames	LinkedList	Namen aller übergebenen Parameter aus der Node-Description
parameterValues	LinkedList	Werte all dieser Parameter (als Strings)
Mgmt (optional)	ProcfsRessourceMgmt	Ressourcenmanagement dieser Methode falls nötig

Tabelle B.1: Methoden zur Inter-EE-Kommunikations Einbindung

In diesem Falle ist die Länge der beiden Linked Lists jeweils null, da keine Paramter übergeben werden müssen. Auch das Ressourcenmanagement ist in diesem Falle wahrscheinlich nicht nötig. Die Zuweisung eines einmaligen Pfades kann sicher mit einer Kombination aus den zu verbindenden Service-Komponeneten ermöglicht werden. Die Aufgaben der Funktion addAdapter bestehen im Generieren der beiden Adapterimplementationen sowie in deren Initialisierung. Weiter fügt sie schlussendlich noch die Adapter in die ImplementationMap ein.

Braucht man für eine bestimmte Kommunikation eine Ressourcenverwaltung, so muss diese unter dem Namen „ProcfsRessourceManagement“ zu finden sein. Dieses Management wird automatisch im Repository deponiert und wieder geladen, wenn erneut diese Kommunikationsmethode eingefügt würde.

Anhang C

Bekannte Bugs

Uns haben vor allem folgende Bugs gestört:

- Der Module-count von click.o (mit lsmod) geht beim Aufsetzten eines Service-Request bis auf 13 und kommt nach Beenden der EEs nur zurück auf 11. Dadurch zeigt Click ohne Neustart des PCs seltsames Verhalten. Dies führt bei erneuter Aufsetzung eines Services oft zu einem Reboot oder einer Kernel-Panik führt. Die Ursache dieses Fehlers zu eruieren und aus der Welt zu schaffen wäre nicht schlecht.

- Beim Anfordern eines Services werden den Service-Komponenten Namen verteilt. Diese Namen werden nicht vorsichtig genug gewählt, und so kann es vorkommen, dass in der Konfigurationsdatei der ClickEE verschiedene Komponenten mit demselben Namen existieren.

Diese Auflistung ist nicht vollständig, die in den vorgehenden Arbeiten [sa, da] beschriebenen Bugs sind immer noch vorhanden.

Anhang D

Demoservice

Der Beispielservice, welchen wir anforderten, weicht nur leicht vom Service unseres Vorgängers[2] ab. Bei einem eingehenden Packet wird zuerst der Ethernet-Header entfernt (1) und in einem zweiten Schritt der IP-Header geprüft (2). Die eigentliche Information Fusion wird in der Service-Komponente 7 betrieben. Um unser Beispiel etwas aufzuwerten, wollen wir zeigen, dass das anwenden mehrerer Verbindungen über Netlink-Sockets möglich ist. So haben wir jeweils zwei Service-Komponenten in der JavaEE (3+5) wie auch in der ClickEE (4+6) eingefügt.
Nach dem Auflösen der Service-Request, sieht der Service folgendermassen aus. Alle grauen Komponenten stellen Adapter dar. Sie wurden von der SCE eingefügt, um den Anforderungen der Inter-EE-Kommunikation zu genügen. Der Service lief reibungslos ab, benötigt aber bei mehrmaligem Anfordern ein sauberes Entfernen des vorher installierten Services.

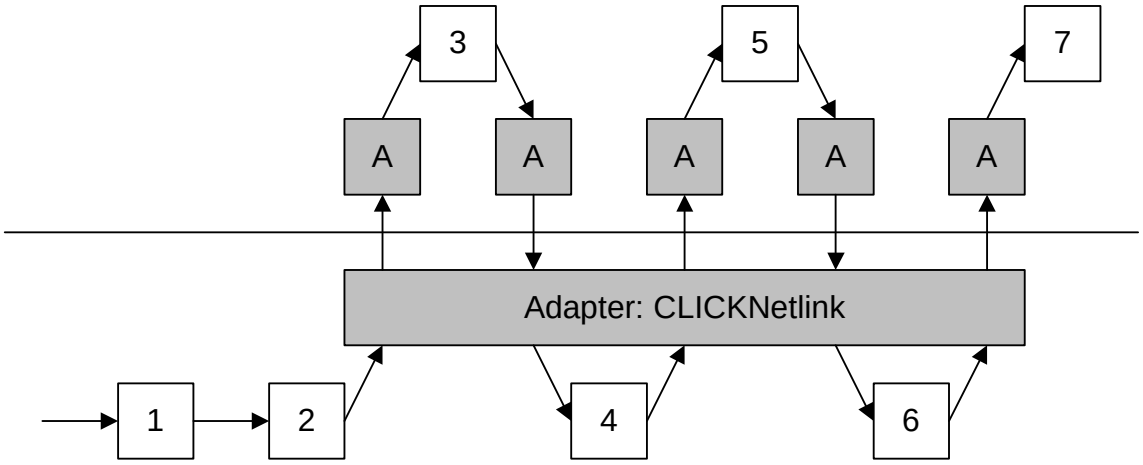

Abbildung 1 - Beispielservice

[2] Julio Perez [sa]

ZEITPLAN SEMESTERARBEIT SS 04

Woche / Thema	1 29.3-4.4	2 5.-11.4	3 12.-18.4	4 19.-25.4	5 26.4-2.5	6 3.-9.5	7 10.-16.5	8 17.-23.5	9 24.-30.5	10 31.5-6.6	11 7.-13.6	12 14.-20.6	13 21.-27.6	14 28.6-4.7
Einarbeiten (Active Networks, Netlink, Service Deployment)														
Netlink					**Design**	**Impl.**	**Deploy**							
Repository								**Design**	**Impl.**	**Test**				
SCE überarbeiten						**Design**	**Impl.**	**Test**						
Reparametrisizer									**Design**	**Impl.**	**Test**			
Dokumentation														
Demo/Präsentation											**Demo**			
Milestones				**Node läuft**			**Netlink läuft**			**Repo-sitory läuft**	**Repara-metrisizer läuft**	**Präsen-tation ist fertig**		**Doc fertig**

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Summer 2004

Semester Thesis

für

Christoph Jossi, Mischa Weise

Supervisor: Matthias Bossardt

Ausgabe: 29.3.2004
Abgabe: 2.7.2004

Service Management on Chameleon

---

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, load balancing as well as specialised multicasting protocols and video scaling.[1]

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.

Service deployment includes installing and configuring software components that perform processing of the data packets. Such components run in execution environments. Many types of execution environments (EE) have been developed. Some EEs are optimized for transport plane operations, where as others target the control or even the management plane. Active nodes typically provide more than one to support the whole spectrum of services.

[1] This list is not exhaustive.

In our lab, we developed a service deployment architecture for active network *nodes* [3], which allows to identify and deploy service components that must be executed on a specific node. This task is carried out by the Service Creation Engine (SCE), which matches service descriptors against a node descriptor. Service descriptors contain the requirements of a service (component), while the node descriptor describes the facilities offered by the node.

The SCE first finds all possible implementations of a service on a given node. In a second step, it evaluates these solutions with an algorithm and selects the best implementation. It subsequently controls the EE configurators to load, install and bind the service components [2, 4].

The current node implementation features a Java-based and a kernel based EE. From a performance point of view, the Java-based EE, is not sufficient for transport plane operations. More efficient approaches for this type of task include EE that execute code in the kernel space. Therefore, we integrated a Click router [5] based EE into our system. Click EE enables simple configuration of the transport plane of a software router. It executes code in kernel space and provides a variety of service components (Click terminology: elements). Additionally, the node provides inter-EE communication facilities in the form of Netlink sockets (partly implemented) and the proc file system, which allow service components in different EEs to communicate.

In this thesis, the management facilities of the Chameleon node should be extended to keep track of the installed service components. Several concurrently running services should be supported. Furthermore, Netlink communication between service components must be finished.

2 Assignment

2.1 Objectives

The goal of this thesis is to develop a node management component, which keeps track of the installed service components. Each of the installed service components should be addressable in order to allow for its reconfiguration or removal. The management component must be integrated with Chameleon node. As second priority, the node management component should also support several concurrently running services.

Netlink communication, which is partly implemented, must be fully supported.

2.2 Tasks

- Get familiar with active networks and service deployment concepts. Read and understand the assigned papers, which deal with this topic [3, 1, 2].
- Get familiar with the concepts related to the Chameleon node [2, 4, 6].
- Set up your work environment, i.e. a Chameleon node, as described in [4, 6] and run one of the sample services. In particular it is important to understand the working of the SCE.
- Implement the missing Netlink adapters [6]. Adapt the node descriptor accordingly.
- Enhance the SCE to support all adapter types that are listed in the node description.

- Come up with a design for a node management component to keep track of the installed service components. Be aware that the design should be such that it can be extended to support several concurrently installed services. You must sketch how this could be done.
- Implement the proposed node management component.
- *Optional:* Implement support for several concurrently installed services.
- Implement a demonstration 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, student 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 student 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, the assignment and the time schedule. Its structure should include an introduction, and a complete documentation of the produced software. 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 *Proceedings of the Fourth Annual International Working Conference on Active Networks, IWAN 2002*, number 2546 in Lecture Notes in Computer Science, Zurich, Switzerland, December 2002. Springer Verlag.

[2] Matthias Bossardt, Roman Hoog Antink, Andreas Moser, and Bernhard Plattner. Chameleon: Realizing automatic service composition for extensible active routers. In *Proceedings of the Fifth Annual International Working Conference on Active Networks, IWAN 2003*, number 2982 in Lecture Notes in Computer Science, Kyoto, Japan, December 2004. Springer Verlag.

[3] 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.

[4] R. Hoog Antink and A. Moser. Service composition for active networks. Master's thesis, ETH Zurich, Switzerland, 2003.

[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] Julio Perez. Active network service mit netlink. Semester thesis at ETH Zurich, Switzerland, 2004.

Zürich, den 20.3.2003