ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Sandro Ribolla

Mapping Services Onto Hierarchy-extended Active Network Nodes

Diploma Thesis DA-2004-13
April 2004 to October 2004

Supervisor: Lukas Ruf
Professor: Bernhard Plattner

Zusammenfassung

Moderne Router Architekturen bestehen aus einer Vielzahl von Prozessoren, welche diverse Tasks verrichten. Im ersten Teil dieser Master Thesis geht es darum, einen Algorithmus zu entwickeln, welcher auf effiziente Art und Weise einen Service-Graphen, bestehend aus mehreren Service-Chains, auf einen Prozessor-Graphen abbildet. Die Effizienz des Algorithmus wird anhand einer Beispielstruktur gemessen und mit einem so genannten 'Exhaustive-Search-Algorithmus' verglichen. Im zweiten Teil der Master Thesis soll das PromethOS NP-extended Execution Environment des Netzwerkknotens optimiert werden, so dass die Zahl der Klassifikationen für einen Flow reduziert werden kann. Durch diese Reduktion erhofft man sich eine verbesserte Performance für den PromethOS NP Knoten.

Modern router architectures are built of many processors, which execute different tasks. In the first part of this master thesis the ambition is to design an algorithm, which maps a service graph containing some service chains onto a processor graph in an efficient manner. The efficiency of the algorithm is measured and compared to a so-called exhaustive search algorithm. In the second part of the master thesis the PromethOS NP-extended execution environment is to be optimized in such a way as to reduce the number of classifications of a flow. By this reduction you expect a better performance of the PromethOS NP node.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel soll dem Leser einen kurzen Überblick über die Thematik und die Problemstellung dieser Master Thesis liefern.

1.1 Motivation

Netzwerkdienste benötigen eine enorme Rechenleistung von Routern, um Datenpakete in kurzer Zeit bearbeiten zu können. Damit diese Rechenleistung bereitgestellt werden kann, beinhalten Router diverse Prozessoren, welche in einer hierarchischen Struktur organisiert sind. Auf diese Prozessoren können Services installiert werden, die selektiv Pakete verarbeiten. Ein Beispiel für einen solchen Dienst ist die Datenverschlüsselung. Die Erweiterung der Rechenleistung hat den Nachteil, dass die Komplexität der Verwaltung der Prozessoren zunimmt.

Am Institut für Technische Informatik und Kommunikationssysteme (TIK) der ETH Zürich (ETHZ) wurde im Rahmen der Semesterarbeit von Amir Guindehi COBRA [3] entwickelt. Bei COBRA handelt es sich um eine modulare und erweiterbare Routerarchitektur. Später wurde das Projekt in PromethOS [5] unbenannt. Basierend auf dem Linux Kernel 2.4 kann die PromethOS-Plattform dynamisch erweitert werden. PromethOS wurde später im Rahmen des Projekts zu PromethOS NP [6], [7] weiterentwickelt, damit die Plattform von Netzwerkprozessoren Gebrauch machen konnte. Diese Netzwerkprozessoren liefern die Rechenleistung, welche für die Netzwerkdienste gebraucht werden.

1.2 Problemstellung

Die offizielle Problemstellung kann im Anhang 2 nachgeschlagen werden. Diese Master Thesis ist in zwei Teile gegliedert: Im ersten Teil soll ein effizienter Algorithmus implementiert werden, der einen Service-Graphen auf einen Prozessor-Graphen abbildet. Im zweiten Teil soll das PromethOS NP-extended Execution Environment optimiert werden.

1.2.1 Teil 1: Mapping-Algorithmus

Im Paper [6] ist die Definition eines Service-Graphen beschrieben. Ein Service-Graph besteht aus einer bestimmten Zahl von verknüpften Service-Chains. Eine Service-Chain beinhaltet wiederum eine bestimmte Zahl von Plugins. Ein Plugin wird durch ein Modul im Kernel repräsentiert. Das Mapping von einem Service-Graphen entspricht dem Zuordnen seiner Service-Chains zu einzelnen Prozessoren des Prozessor-Graphen, auf denen die Service-Chains dann installiert werden. Will man den Service optimal auf den Prozessor-Graphen mappen, müssen die Prozessoren so gewählt werden, dass sie einerseits den Anforderungen der Service-Chains genügen und andererseits einen minimalen Weg (Delay) zwischen den einzelnen Prozessoren aufweisen. Die Schwierigkeiten der Implementierung eines solchen Mapping-Algorithmus liegen bei den folgenden Punkten:

1. Die Zeit für das Ausführen des Mapping-Algorithmus sollte in einem vernünftigen Rahmen bleiben. Dieser Rahmen wird durch die Spezifikation des Service-Graphen vorgegeben.
2. Da die Zahl der Prozessoren im Prozessor-Graphen sehr gross sein kann, muss ein intelligenter Algorithmus implementiert werden, welcher Prozessoren eventuell vorselektiert und gruppiert, so dass der Rechenaufwand nicht zu gross wird.
3. Es muss garantiert werden können, dass die gefundene Lösung auch korrekt ist und der Algorithmus für alle Graphen funktioniert.
4. Es müssen Regeln und Parameter definiert werden, wie ein Service beschrieben werden soll, so dass der Algorithmus später leicht in den PromethOS NP Knoten eingebunden werden kann.

1.2.2 Teil 2: Optimierung des PromethOS NP-extended Execution Environments

Im zweiten Teil dieser Master Thesis geht es darum, das bestehende PromethOS NP-extended Execution Environment zu optimieren. Konkret möchte man die Anzahl an Klassifikationen für Flows reduzieren, um so eine bessere Performance zu erreichen. Die Schwierigkeiten dieser Optimierung liegen einerseits beim Einarbeiten in den bestehenden Code, da dieser aus einer Vielzahl komplexer Codedateien besteht. Andererseits muss ein Designvorschlag für die Implementierung gemacht werden, welcher sich nahtlos in das bestehende Execution Environment eingliedert, ohne die bestehende Funktionalität einzuschränken. Diese Implementierung wird ebenfalls als nicht trivial eingeschätzt.

1.3 Gliederung dieses Berichts

Das Kapitel 2 definiert die Schnittstellen zum PromethOS NP Knoten und wie diese im Mapping-Algorithmus verwendet werden. Diese Schnittstellen sollen eine spätere Einbindung des Algorithmus in den PromethOS NP Knoten erleichtern.

Im Kapitel 3 werden die Lösungsansätze präsentiert und erläutert. Weiter wird die daraus resultierende Implementierung erklärt. Am Schluss des Kapitels wird auf die Evaluation der entwickelten Mapping-Algorithmen eingegangen.

Das Kapitel 4 befasst sich mit der Optimierung des Mapping-Algorithmus. Es werden verschiedene Lösungsansätze präsentiert, welche die Performance des Algorithmus verbessern sollen. Am Ende des Kapitels wird erneut auf die Evaluation der optimierten Algorithmen eingegangen und ein Vergleich mit den nicht optimierten Algorithmen gezogen.

Das Kapitel 5 beinhaltet die Optimierung des PromethOS NP-extended Execution Environment. Es werden zwei Lösungsansätze zur Reduktion der Zahl der Klassifikationen vorgestellt. Weiter wird die Implementierung des gewählten Lösungsansatzes beschrieben und am Ende des Kapitels evaluiert.

Das Kapitel 6 bildet den Abschluss dieser Master Thesis, indem die erreichten Ziele zusammengefasst werden und ein Ausblick auf weiterführende Arbeiten gewagt wird.

Der Anhang enthält begleitende Informationen zu dieser Arbeit, wie die offizielle Aufgabenstellung, der zu Beginn der Arbeit aufgestellte Zeitplan und das Layout des Codes.

1.4 Danksagung

Ich danke allen Personen, welche mich im Laufe dieser Master Thesis in irgendeiner Form unterstützt haben.

Weiter möchte ich mich bei Herrn Professor Dr. Bernhard Plattner bedanken, dass er mir diese Arbeit ermöglicht hat.

Speziell bedanken möchte ich mich bei Lukas Ruf für seine einmalige Betreuung. Sein Engagement, seine Kompetenz und seine Geduld haben mir sehr geholfen diese Arbeit erfolgreich abzuschliessen.

Kapitel 2

Ausgangslage

Dieses Kapitel soll einen Überblick liefern über die Datenstrukturen, welche im Mapping-Algorithmus benötigt werden. Dabei werden die Datenstrukturen so gewählt, dass sie die reellen Ressourcen des PromethOS NP Knotens möglichst detailliert implementieren. Umso realistischer diese Schnittstellen implementiert werden, desto einfacher ist eine spätere Einbindung der Algorithmen in den PromethOS NP Knoten.

Als erstes werden die Informationen zusammengefasst, welche man für den virtuellen Aufbau des Graphen benötigt. Diese Informationen beziehen sich ausschliesslich auf die 'Hardware' des Knotens. Danach folgen die Anforderungen an die Definition eines Service-Graphen. Die Ressourcen, welche der Service-Graph benötigt, sind im Abschnitt 2.2 aufgelistet. Weiter unterscheidet man zwischen statischen und dynamischen Ressourcen: Statische Ressourcen sind Ressourcen, welche für jede Ausführung des Mapping-Algorithmus gleich bleiben. Dynamische Ressourcen verändern ihren Wert von Mapping zu Mapping.

2.1 Hardware: Routerarchitektur

Im Paper [6] ist beschrieben, wie der Node-Manager die Ressourcen des Routers in einer 'Resource Information Database' (RID) verwaltet. Diese Informationen aus der RID werden benötigt, um den Prozessor-Graphen für den Mapping-Algorithmus aufzubauen. Die RID muss die folgenden statischen und dynamischen Informationen beinhalten:

2.1.1 Statische Ressourcen

Die erste statische Ressource, welche benötigt wird, ist die Grösse des Speichers (Memory), welcher pro Prozessor verfügbar ist. Weiter wird pro Prozessor die Id und der Typ des Prozessors gespeichert. Im weiteren Verlauf dieser Arbeit wird häufig der Begriff 'Port-Nummer' verwendet. Damit ist die Id des Prozessors gemeint. Die 'Processing-Time' einer CPU entspricht der Zeit, die der Prozessor benötigt, um eine bestimmte Arbeit zu verrichten und gehört auch zu den statischen Ressourcen. Der Einfachheit halber ist die Processing-Time pro Prozessor für jeden Arbeitsaufwand gleich gross. Das heisst, dass die Zeit, die ein Prozessor braucht um ein Programm der Grösse 512kB auszuführen, gleich gross ist, wie bei einem Programm der Grösse 256kB. Diese Vereinfachung kann problemlos gemacht werden, da sie alle Prozessoren gleichermassen betrifft und somit keinen Einfluss auf das Mapping ausübt. Die Information, wie die Prozessoren des Routers untereinander verbunden sind, gehört ebenfalls zu den statischen Ressourcen. Diese Information wird in der Graphentheorie meistens in Form einer Adjadanzmatrix gespeichert und genau so wird auch hier vorgegangen. Als letzte statische Ressource benötigt man die Informationen über die Bandbreite und den Delay der Kommunikationslinks.

Die statischen Ressourcen:

- Memory (Grösse)

- CPU's (Id und Typ)
- Processing-Time einer CPU
- Kommunikationslinks zwischen den Prozessoren (Adjadanzmatrix)
- Bandbreite und Delay der Kommunikationslinks

2.1.2 Dynamische Ressourcen

Zu den dynamischen Ressourcen zählt der momentan verfügbare Speicherplatz. Man muss den momentan verfügbaren Speicherplatz kennen, um entscheiden zu können, ob man die Service-Chain in den Speicherbereich des Prozessors laden kann. Weiter wird das Wissen benötigt, welche Prozessoren bei Beginn des Mapping-Algorithmus verfügbar sind. Dieses Wissen ermöglicht, dass der Aufbau des Graphen auf die momentan verfügbaren Prozessoren beschränkt werden kann und so der Graph kleiner wird, als wenn man auch die nicht verfügbaren Prozessoren berücksichtigen würde. Ein solches Vorgehen führt zu einer besseren Performance beim Mapping, da der Mapping-Algorithmus eine geringere Knotenzahl abarbeiten muss. Zuletzt muss die momentan verfügbare Bandbreite der Kommunikationslinks gespeichert werden, damit eine Überlastung der Links verhindert werden kann.

Die dynamischen Ressourcen:

- momentan verfügbares Memory (Grösse)
- momentan verfügbare CPU's (Id und Typ)
- momentan verfügbare Bandbreite der Kommunikationslinks

Anhand der statischen und dynamischen Ressourcen ist der Prozessor-Graph eines PrometheOS NP Knotens vollständig beschrieben.

2.2 Software: Service-Graph und Service-Chains

Ein Service-Graph entspricht einem gerichteten Graphen. Die Knoten dieses Graphen entsprechen einer Service-Chain und die Links symbolisieren die notwendige Bandbreite, die für diese Service-Chain aufgebracht werden muss. Der ganze gerichtete Graph stellt den Ablauf der einzelnen Service-Chains dar. Ein Beispiel für einen Service-Graphen ist im Kapitel 3 in der Abbildung 3.2 dargestellt.
Zusätzlich zu diesem Ablauf benötigt man Informationen, die von der Spezifikation des Service-Graphen abhängen. Diese Spezifikation wird durch den Benutzer beim Erstellen des Service-Graphen festgelegt. Es handelt sich dabei um Informationen wie zum Beispiel den Eingangs-Port und den Ausgangs-Port des Service-Graphen.

Es können folgende statische und dynamische Ressourcen für die Service-Chains definiert werden:

2.2.1 Statische Ressourcen

Als erste statische Ressource benötigt man den Typ des Prozessors, auf den die Service-Chain gemappt werden kann. Dabei wird nur ein gültiger Typ erlaubt. Es ist also nicht möglich, dass mehrere Prozessortypen für eine Service-Chain in Frage kommen. Dieser Umstand soll den Mapping-Algorithmus weiter vereinfachen. Falls eine Service-Chain nur auf einen bestimmten Prozessor gemappt werden darf, muss eine solche Information ebenfalls gespeichert werden. Dies geschieht mit Hilfe eines Eintrags für die Port-Nummer. Eine grosse Anzahl der Ressourcen, welche eine Service-Chain benötigt, kann nicht exakt abgegrenzt werden. Vielmehr werden durch eine 'Schätzung' die minimale und maximale Grenze bestimmt. Um den Algorithmus zu vereinfachen wird diese Schätzung aber noch vernachlässigt und es wird von exakten Werten für die Ressourcen ausgegangen. Zu diesen Ressourcen zählt die Grösse

der Service-Chain, welche im Speicher des Prozessors gebraucht wird. Der Speicherbedarf ist von der Grösse des Source-Code der einzelnen Plugins, welche sich in der Service-Chain befinden, sowie von der Grösse der Service-Chain selbst abhängig. Diese Grösse wird im weiteren Text mit 'Processing-Cost' bezeichnet. Weiter zählt die Bandbreite einer Service-Chain zu den 'unscharfen Ressourcen'. Die Bandbreite einer Service-Chain entspricht der Bandbreite, welche ein Link aufweisen muss, damit er die Daten der Service-Chain weiterleiten kann. Schliesslich fehlt noch die Ressource, welche die Zeiteinschränkung der Service-Chain speichert.

Die statischen Ressourcen:

- Prozessortyp (z.B Host-Prozessor oder Packet-Prozessor)
- Port-Nummer (Id)
- Definition der Grenzen der 'unscharfen' Ressourcen:
- minRessourcen$\leq$zugewieseneRessourcen$\leq$maxRessourcen
- Diese 'unscharfen' Ressourcen bestehen aus: Memory, Bandbreite und Zeit)

2.2.2 Dynamische Ressourcen

Unter den dynamischen Ressourcen für den Service-Graphen und die Service-Chains versteht man nicht das gleiche, wie für die dynamischen Ressourcen des Prozessor-Graphen. Unter dynamischen Ressourcen für den Service-Graphen und die Service-Chains versteht man die Ressourcen, welche pro Service-Chain während des Mapping-Algorithmus in Gebrauch sind, um ein Paket zu verarbeiten. Diese Information kann für Statistiken verwendet werden, hat aber auf den Mapping-Algorithmus keinen Einfluss und wird auch nicht weiter verwendet.

Die dynamischen Ressourcen:

- Memory
- Bandbreite
- CPU
- Zeit

```
struct node {
   int type;
   int id;
   int procUnits;
   int procTime;
   int forwardTime;
};
```

(a) Knotenstruktur

```
struct link {
   int bandwith;
   int delay;
};
```

(b) Linkstruktur

Abbildung 3.4: Notwendige Strukturen für die Repräsentation des Prozessor-Graphen

Graphen verwendet werden. Im letzten Abschnitt wird der Aufbau des Layered-Graphen erklärt.

Strukturen: Prozessor-Graph

Wir benötigen zwei Strukturen für die Repräsentation eines Prozessor-Graphen: eine Struktur, welche einen Prozessor repräsentiert und eine Struktur, welche einen Link symbolisiert. Die Parameter der Strukturen kann man aus den Schnittstellen-Definitionen von Kapitel 2 herleiten.

Prozessor-Struktur Für die Prozessor-Struktur ergeben sich die folgenden Parameter: Typ, Id, Processing-Units, Processing-Time und Forwarding-Time. Der Prozessor-Typ identifiziert, um welchen Typ von Prozessor es sich handelt. Die Id enthält einen eindeutigen Integer-Wert, welcher im ganzen Prozessor-Graphen einmalig ist. Die Processing-Units wiederspiegeln die Rechenkapazität eines Prozessors, welche mit dem 'Instruction-Memory' einer CPU vergleichbar ist. Die Processing-Time entspricht der Zeit, welche vom Prozessor benötigt wird, um eine bestimmte Arbeit zu erledigen. Wie im Abschnitt 2.1.1 schon erläutert wurde, ist diese Processing-Time unabhängig von der Grösse der Arbeit. Unter der Forwarding-Time versteht man die Zeit, welche benötigt wird, wenn ein Prozessor ein Paket nur weiterleitet (forwarded) und nicht verarbeitet. Die Processing-Time und die Forwarding-Time eines Prozessors haben den grössten Einfluss auf das Mapping, da man diese beiden Faktoren minimal halten möchte. In Abbildung 3.4(a) sieht man die Datenstruktur eines Prozessors in 'C-Notation'. Um alle Prozessoren des Prozessor-Graphen speichern zu können, wird ein Array $nodeSet$ verwendet, welches Einträge mit der Prozessor-Struktur speichern kann. Im $nodeSet$ Array sind alle Prozessoren des Prozessor-Graphen nach Id sortiert gespeichert. Falls auf einen Prozessor zugegriffen wird, also z.B. wenn sich die Processing-Units ändern, wird mittels der Id des Prozessors auf das $nodeSet$ zugegriffen und der Eintrag aktualisiert.

Link-Struktur Die Parameter eines Links kann man ebenfalls aus Kapitel 2 interpretieren. Die beiden Parameter: Bandbreite und Delay beschreiben die Eigenschaften eines Links ausreichend. In Abbildung 3.4(b) ist die Datenstruktur der Links dargestellt. Für die Verwaltung der Links wird die Adjadanzmatrix verwendet. Die Adjadanzmatrix speichert für jeden Knoten i den Nachbarsknoten j, welcher durch genau einen Link(i,j) erreichbar ist. Es existieren für jeden Link im Prozessor-Graphen genau zwei Einträge in der Adjadanzmatrix.

Adjadanzmatrix Es wird eine n*n-Matrix $adjMatrix$ erstellt, wobei n die Anzahl Prozessoren im Prozessor-Graphen ist. Ein Eintrag in $adjMatrix$ entspricht der Link-Struktur. Gibt es einen Link zwischen dem Knoten i und dem Knoten j mit der Bandbreite b und dem Delay d, so wird an der Stelle i, j der Adjadanzmatrix ein Eintrag gemacht mit $adjMatrix[i][j].bandwith = b$ und $adjMatrix[i][j].delay = d$. Gibt es keinen Link vom Knoten i zum Knoten j wird die Bandbreite und der Delay an dieser Stelle der Adjadanzmatrix auf 0 bzw. -1 gesetzt.

Das folgende Beispiel soll die Verwendung der Datenstrukturen verdeutlichen.

Beispiel Gegeben ist der Basis-Graph aus Abbildung 3.5. Die Anzahl Knoten des Basis-Graphen ist 7. Das bedeutet, dass es sieben Einträge im Array $nodeSet$ gibt. Weiter gibt es drei

```
struct service {
   int countServiceChains;
   int inPort;
   int outPort;
   serviceChain * servChain;
}
```

Abbildung 3.7: Die Datenstruktur für einen Service-Graphen

```
struct serviceChain{
   int maxBandwith;
   int procType;
   int portNr;
   int maxProcCost;
};
```

Abbildung 3.8: Die Datenstruktur für eine Service-Chain

sen. Diese Prozessoren werden anhand der Werte, welche in $inPort$ und $outPort$ gespeichert werden, selektiert. Als letzter Parameter enthält die Service-Graph-Datenstruktur ein Array $servChain$, welches die im Service-Graphen enthaltenen Service-Chains in der richtigen Reihenfolge speichert.

Service-Chain Die Datenstruktur für die Spezifikation einer einzelnen Service-Chain ist in Abbildung 3.8 dargestellt. Mittels dem Parameter $maxBandwith$ wird die maximale Bandbreite spezifiziert, welche von der Service-Chain benötigt wird. Der Wert von $procType$ legt den Prozessor-Typ fest, auf dem die Service-Chain installiert werden kann. Falls die Service-Chain auf einen ganz bestimmten Prozessor gemappt werden soll, kann mit dem Wert für $portNr$ die Prozessor-Id übergeben werden, so dass der Mapping-Algorithmus die Service-Chain auf den gewünschten Prozessor abbildet. Ein Wert von $portNr = -1$ symbolisiert, dass die Service-Chain auf einem beliebigen Prozessor vom Typ $procType$ installiert werden kann. Die maximal benötigten Processing-Ressourcen werden im Parameter $maxProcCost$ übergeben. Die Service-Chain kann nur auf Prozessoren installiert werden, welche die benötigten $maxProcCost$ auch zur Verfügung stellen können.

Beispiel Das Beispiel soll die Verwendung der beiden Datenstrukturen nochmals erläutern und so das Verständnis erleichtern. Der zu mappende Service-Graph aus Abbildung 3.9 besteht aus drei Service-Chains. Die Datenpakete, die für den Service-Graphen bestimmt sind, sollen beim Eingangs-Port 4 den Prozessor-Graphen 'betreten' und beim Ausgangs-Port 3 ihn wieder verlassen. Die erste Service-Chain benötigt Processing-Ressourcen in der Grösse von 30, die zweite 50 und die letzte 35. Wie schon erwähnt wurde, ist die erste und die letzte Service-Chain immer an einen Prozessor gebunden. Die zweite Service-Chain soll auf einen beliebigen Prozessor vom Typ 3 gemappt werden. Die benötigte Bandbreite ist für alle drei Service-Chains 120. Aus dieser Spezifikation resultieren die Datenstrukturen aus Abbildung 3.10.

Damit sind die notwendigen Informationen vollständig und der Layered-Graph kann erzeugt werden.

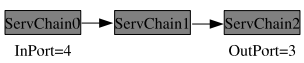


Abbildung 3.9: Ein einfaches Beispiel für einen Service-Graphen

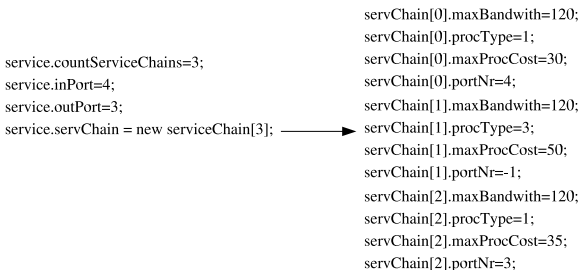


Abbildung 3.10: Die Datenstrukturen für den Service-Graphen aus Abbildung 3.9

Layered-Graph

Defintion: Intra-Layer-Link Unter Intra-Layer-Link versteht man einen Link des Basis-Graphen. Intra-Layer-Links sind reale bidirektionale Links, deren Bandbreite und Delay im Basis-Graphen definiert sind.

Defintion: Inter-Layer-Link Ein Inter-Layer-Link ist ein virtueller Link, der zwei Layer im Layered-Graphen unidirektional miteinander verbindet. Es handelt sich um virtuelle Links, da sie eigentlich einen reellen Prozessor darstellen. Die Bandbreite eines Inter-Layer-Links entspricht den Processing-Units des korrespondierenden Prozessors. Der Delay symbolisiert die Processing-Time des Prozessors.

Der Aufbau des Layered-Graphen kann in zwei Schritte unterteilt werden: Zuerst werden die Intra-Layer-Links für jeden Layer in der Matrix $adjMatrix$, welche neu die Links des Layered-Graphen enthält, gespeichert. In einem zweiten Schritt wird die Matrix, durch die Inter-Layer-Links ergänzt. Die resultierende Matrix hat die Grösse $n = service.countServiceChains * countNode$, wobei $countNode$ die Anzahl Prozessoren im Basis-Graphen darstellt.

Die Intra-Layer-Links sind für jeden Layer identisch. Dieses Wissen machen wir uns zu Nutzen, indem wir die Einträge der $adjMatrix$ einfach für jeden Layer an die richtige Stelle kopieren. Der Link von Knoten i zum Knoten j im Basis-Graphen entspricht dem Link vom Knoten $i + m * countNode$ zum Knoten $j + m * countNode$ auf Layer m im Layered-Graphen. Die adjMatrix des Basisgraphen wird $service.countServiceChains - 1$-mal kopiert. Damit sind die Intra-Layer-Links für jeden Layer vollständig implementiert.

Das Erzeugen der Inter-Layer-Links funktioniert auf ähnliche Weise: Für jede Service-Chain des Service-Graphen wird überprüft, ob die Service-Chain nur auf einem bestimmten Prozessor installiert werden darf. Wenn dies der Fall ist, wird an der richtigen Stelle in der $adjMatrix$ der Inter-Layer-Link eingefügt. Wenn z.B. für die erste Service-Chain $service.servChain[0]$ nur der Prozessor mit Id=4 für ein Mapping in Frage kommt, wird an der Stelle $adjMatrix[0 * countNode + service.servChain[0].portNr][1 * countNode + service.servChain[0].portNr]$ ein Eintrag eingefügt. Der Bandbreite wird der Wert der Processing-Units des Prozessors und dem Delay der Wert der Processing-Time zugewiesen. Damit ist der Inter-Layer-Link vollständig in den Layered-Graphen eingefügt worden.
Ist die Service-Chain nicht an einen bestimmten Prozessor gebunden, wird die obige Prozedur für jeden Prozessor gemacht, welcher den benötigten Typ aufweist.

Die Theorie soll anhand des folgenden Beispiels nochmals erläutert werden.

Beispiel Gegeben sind der Service-Graph aus Abbildung 3.9 und der Basis-Graph aus Abbildung 3.5. Die Adjadanzmatrix $adjMatrix$ nach der Ausführung der beiden Schritte ist in Abbildung 3.11 aufgeführt. Dargestellt sind dabei nur die Bandbreiten der Links. Die Struktur der $adjMatrix$ für die Delays sieht aber identisch aus. Der korrespondierende Layered-Graph kann der Abbildung 3.12 entnommen werden. Speziell zu erwähnen gilt es die Tatsache, dass der Inter-Layer-Link der letzten Service-Chain nicht in den Layered-Graphen eingefügt wird. Das

Performance-Messungen

Die Performance-Messungen sind aus Zeitgründen einfach gehalten. Um zu überprüfen, wieviel Speicher verwendet wird, benutzt man das Linux-Tool 'top' (table of processes). Die Zeitmessungen werden mit der Funktion 'gettimeofday' aus der 'sys/time.h' Bibliothek durchgeführt.

Da die Grösse des verbrauchten Speicherbereichs und die benötigte Zeit von der Grösse des Basis-Graphen und der Grösse des Service-Graphen abhängt, wird eine Messung zuerst für weitere Basis-Graphen mit steigender Prozessorzahl durchgeführt und danach die Grösse des Service-Graphen variiert.

In einer ersten Messung wird die Anzahl Knoten im Basis-Graph Schritt für Schritt verdoppelt. Es existieren von jedem Prozessortyp gleich viele Prozessoren. Gemessen wird die Grösse des benutzten virtuellen Speichers, sowie die Zeit für die Generierung des Layered-Graphen. Die Resultate sind in den Files 'layGraph_node_mem_2.dat' und 'layGraph_node_time_2.dat' im Verzeichnis '/ribolla/code/exhauSearch/performance' gespeichert.

In der Abbildung 3.14(a) sind die Resultate für den Speicherbedarf an virtuellem Speicher graphisch dargestellt. Der Speicherbedarf an virtuellem Memory scheint mit zunehmender Knotenzahl exponentiell zu steigen.
In der Abbildung 3.14(b) sind die Resultate für den Zeitaufwand graphisch dargestellt. Der Zeitaufwand scheint ebenfalls mit zunehmender Knotenzahl exponentiell zuzunehmen.

In einer zweiten Messung bleibt die Anzahl Knoten im Basis-Graph konstant $countNode = 96$. Im Unterschied zur ersten Messung wird der Service-Graph schrittweise um eine Service-Chain erweitert. Gemessen wird die Grösse des benutzten virtuellen Speichers, sowie die Zeit für die Generierung des Layered-Graphen.

Das Ergebnis für die zweite Messung ist identisch zur ersten Messung. Es spielt folglich keine Rolle, ob man die Knotenzahl des Basis-Graphen oder die Zahl der Service-Chains erhöht. Entscheidend ist, dass die Knotenzahl im Layered-Graphen zunimmt und darum die Adjadanzmatrix grösser wird und als Konsequenz mehr Speicher benötigt. Die gleiche Aussage gilt für den Zeitverbrauch.

```
Start Node: 4   Dest Node: 17

Pfad 0 : 4  5  6  4  11  9  7  8  15  16  19  17
Pfad 1 : 4  5  6  4  11  9  7  8  15  17
Pfad 2 : 4  5  6  4  11  9  7  14  15  16  19  17
Pfad 3 : 4  5  6  4  11  9  7  14  15  17
Pfad 4 : 4  5  6  4  11  9  8  7  14  16  19  17
Pfad 5 : 4  5  6  4  11  9  8  15  14  16  19  17
Pfad 6 : 4  5  6  4  11  9  8  15  17
Pfad 7 : 4  6  5  4  11  9  7  8  15  16  19  17
Pfad 8 : 4  6  5  4  11  9  7  8  15  17
Pfad 9 : 4  6  5  4  11  9  7  14  15  16  19  17
Pfad 10 : 4  6  5  4  11  9  7  14  15  17
Pfad 11 : 4  6  5  4  11  9  8  7  14  16  19  17
Pfad 12 : 4  6  5  4  11  9  8  15  14  16  19  17
Pfad 13 : 4  6  5  4  11  9  8  15  17
Pfad 14 : 4  11  9  7  8  15  16  19  17
Pfad 15 : 4  11  9  7  8  15  16  19  18  20  19  17
Pfad 16 : 4  11  9  7  8  15  16  19  20  18  19  17
Pfad 17 : 4  11  9  7  8  15  17
Pfad 18 : 4  11  9  7  14  15  16  19  17
Pfad 19 : 4  11  9  7  14  15  16  19  18  20  19  17
Pfad 20 : 4  11  9  7  14  15  16  19  20  18  19  17
Pfad 21 : 4  11  9  7  14  15  17
Pfad 22 : 4  11  9  8  7  14  16  19  17
Pfad 23 : 4  11  9  8  7  14  16  19  18  20  19  17
Pfad 24 : 4  11  9  8  7  14  16  19  20  18  19  17
Pfad 25 : 4  11  9  8  15  14  16  19  17
Pfad 26 : 4  11  9  8  15  14  16  19  18  20  19  17
Pfad 27 : 4  11  9  8  15  14  16  19  20  18  19  17
Pfad 28 : 4  11  9  8  15  17
Pfad 29 : 4  11  12  9  7  8  15  16  18  20  19  17
Pfad 30 : 4  11  12  9  7  8  15  17
Pfad 31 : 4  11  12  9  7  14  15  16  18  20  19  17
Pfad 32 : 4  11  12  9  7  14  15  17
Pfad 33 : 4  11  12  9  8  7  14  16  18  20  19  17
Pfad 34 : 4  11  12  9  8  15  14  16  18  20  19  17
Pfad 35 : 4  11  12  9  8  15  17
Pfad 36 : 4  11  12  13  11  9  7  8  15  16  19  17
Pfad 37 : 4  11  12  13  11  9  7  8  15  17
Pfad 38 : 4  11  12  13  11  9  7  14  15  16  19  17
Pfad 39 : 4  11  12  13  11  9  7  14  15  17
Pfad 40 : 4  11  12  13  11  9  8  7  14  16  19  17
Pfad 41 : 4  11  12  13  11  9  8  15  14  16  19  17
Pfad 42 : 4  11  12  13  11  9  8  15  17
Pfad 43 : 4  11  13  12  9  7  8  15  16  18  19  17
Pfad 44 : 4  11  13  12  9  7  8  15  17
Pfad 45 : 4  11  13  12  9  7  14  15  16  18  19  17
Pfad 46 : 4  11  13  12  9  7  14  15  17
Pfad 47 : 4  11  13  12  9  8  7  14  16  18  19  17
Pfad 48 : 4  11  13  12  9  8  15  14  16  18  19  17
Pfad 49 : 4  11  13  12  9  8  15  17
Pfad 50 : 4  11  13  12  11  9  7  8  15  16  19  17
Pfad 51 : 4  11  13  12  11  9  7  8  15  17
Pfad 52 : 4  11  13  12  11  9  7  14  15  16  19  17
Pfad 53 : 4  11  13  12  11  9  7  14  15  17
Pfad 54 : 4  11  13  12  11  9  8  7  14  16  19  17
Pfad 55 : 4  11  13  12  11  9  8  15  14  16  19  17
Pfad 56 : 4  11  13  12  11  9  8  15  17
pfadzahl: 57

************************************
minimaler Pfad:  4  11  9  8  15  17
Pfad delay:  41
```

Abbildung 3.15: Der Output des Exhaustive-Search-Algorithmus

Links auf allen Layern aktualisiert. Dafür wird zuerst überprüft, ob es sich um einen Inter-Layer-Link oder um einen Intra-Layer-Link handelt. Diesen Test braucht man, da Inter-Layer-Links nur in eine Richtung zeigen und darum nur einen Eintrag in der Adjadanz-Matrix besitzen, Intra-Layer-Links aber in beide Richtungen zeigen und folglich zwei Einträge besitzen. Als nächster Test wird überprüft, auf welchen Layern der Link existiert, da Inter-Layer-Links nur auf bestimmten Layern vorhanden sind. Danach wird der Eintrag bzw. werden die Einträge in der Adjadanzmatrix um die benötigte Bandbreite der Service-Chain vermindert. Falls man in der Rekursion einen Schritt zurückgeht, wird die Bandbreite wieder hergestellt, wie sie vor dem Rekursionsschritt war.

Am Ende der Rekursion sind alle gültigen Pfade in einem Array gespeichert. Der Algorithmus gibt alle gültigen Pfade aus und wählt den optimalen (schnellsten) Pfad aus.

Beispiel

Gegeben sei der Layered-Graph aus Abbildung 3.12. Der Exhaustive-Search-Algorithmus durchläuft diesen Layered-Graphen so oft, bis er alle möglichen gültigen Pfade berechnet hat. Der resultierende Output des Algorithmus kann der Abbildung 3.15 entnommen werden. Man sieht den Start- und den Endknoten und die 57 Pfade mit den dazugehörigen Knotensequenzen. Als Beispiel ist in Abbildung 3.16 der gültige Pfad mit der Nummer 21 dargestellt. Das optimale Mapping, welches der Algorithmus berechnet hat, ist die Knotensequenz $4, 11, 9, 8, 15, 17$ mit dem minimalen Delay von 41, wie aus dem Resultat der Abbildung 3.15 herausgelesen werden kann. Das lässt sich nun folgendermassen interpretieren: Da unsere Beispielarchitektur aus Abbildung 3.1 sieben Prozessoren beinhaltet, kann man die Knotensequenz $4, 11, 9, 8, 15, 17$ umschreiben auf $4, 4, 2, 1, 1, 3$, wenn man die Einträge $modulo 7$ rechnet. Das bedeutet nun, dass die erste Service-Chain auf den Prozessor 4 gemappt wird, die zweite auf den Prozessor 1 und die letzte auf den Prozessor 3. In Abbildung 3.17 ist der optimale Pfad visualisiert. Dieses Mapping erreicht einen minimalen Delay für den Service-Graphen.

3.2.2 Implementierung

Algorithmus

```
void exhaustiveSearch(service service1)
01 init actNode=startNode
02 if(actNode!=destNode)
03   ∀ nodes i ∈ LayeredGraph
04     if(node i ∈ Neighbour(actNode))
05       if(Link(actNode,i)=InterLayerLink)
06         if(Link(actNode,i).bandwith>service1.serviceChain.maxProcCost)
07           ∀ Layers search Link(m,n) corresponding to Link(actNode,i)
08             Link(m,n).bandwith=Link(m,n).bandwith-service1.serviceChain.maxProcCost
09           actNode=i
10           add actNode to actPath
11           exhaustiveSearch(service1)
12           remove actNode from actPath
13           ∀ Layers search Link(p,q) corresponding to Link(old-actNode,actNode)
14             Link(p,q).bandwith=Link(p,q).bandwith+service1.old-serviceChain.maxProcCost
15       if(Link(actNode,i)=IntraLayerLink)
16         if(Link(actNode,i).bandwith>service1.serviceChain.maxBandwith)
17           ∀ Layers search Link(m,n) corresponding to Link(actNode,i)
18             Link(m,n).bandwith=Link(m,n).bandwith-service1.serviceChain.maxBandwith
19             Link(n,m).bandwith=Link(n,m).bandwith-service1.serviceChain.maxBandwith
20           actNode=i
21           add actNode to actPath
22           exhaustiveSearch(service1)
23           remove actNode from actPath
24           ∀ Layers search Link(p,q) corresponding to Link(old-actNode,actNode)
25             Link(p,q).bandwith=Link(p,q).bandwith+service1.old-serviceChain.maxProcCost
26             Link(q,p).bandwith=Link(q,p).bandwith+service1.old-serviceChain.maxProcCost
27 else
28   calculate delay cost of actPath
29   calculate forwarding cost of actPath
30   if((delay(actPath)+forwading(actPath)) < (delay(minPath)+forwading(minPath)))
31     minPath=actPath
```

Beschreibung

Zuerst werden die, durch den Algorithmus verwendeten Variablen initialisiert (Zeile 01). Damit die Initialisierungen nicht bei jeder Rekursion wieder ausgeführt werden, wird jedesmal überprüft, ob $init = 1$ ist. Die Initialisierung übergibt den Startknoten vom Service $service1.inPort$ der Variable $actNode$, welche den aktuellen Knoten speichert. Weiter wird die Variable $servChainCount$ mit Null initialisert. $servChainCount$ zählt die bereits gemappten Service-Chains. Dieses Wissen wird benötigt, damit man weiss, auf welchem Layer man sich momentan befindet und so auch die richtige Service-Chain verwendet. In der Variable $destNode$ wird die Id des Zielknotens gespeichert. Korrekterweise speichert man die Id des Zielknotens, welche dieser auf dem letzten Layer besitzt. Um die Anzahl gültiger Pfade zu zählen, wird die Variable $pathCount$ benötigt. Am Schluss der Initialisierung speichert man im Array $usedNodeIdSet$ an der ersten Stelle den Startknoten, da dieser der erste Knoten ist, welcher vom Algorithmus abgearbeitet wurde. Danach beginnt der eigentliche Rekursionsschritt. Dieser wird wie folgt implementiert:

Die Adjadanzmatrix $adjMatrix$ wird an der Stelle $actNode$ nach Nachbarn durchsucht (Zeile 03-04). Wenn ein Nachbar gefunden wurde, wird zuerst überprüft, um was für eine Art Nachbarn es sich handelt. Dabei kommen zwei Möglichkeiten in Frage: Entweder ist es ein Intra-Layer- oder ein Inter-Layer-Nachbar. Warum diese Unterscheidung gemacht werden

muss, wird später noch erwläutert.
Handelt es sich um einen Inter-Layer-Nachbar folgt der Test, ob der Link eine genügende Bandbreite aufweist (Zeile 06). Ist dieser Test erfolgreich, wird dieser Link auf allen Layern aktualisiert (Zeile 08). Für diese Aktualisierung muss zuerst festgestellt werden, auf welchen Layern der Link überhaupt existiert (Zeile 07). Ist diese Aktualisierung abgeschlossen, wird der Nachbar selektiert (Zeile 09). Da es sich um einen Inter-Layer-Nachbar und folglich um einen Inter-Layer-Link gehandelt hat, folgt daraus, dass die aktuelle Service-Chain auf einen Prozessor gemappt wurde. Die Folge davon ist, dass $servChainCount$ inkrementiert wird. Weiter wird auch $countLevel$ um eins erhöht, da eine weitere Rekursionsstufe folgt. Damit am Ende des Algorithmus der Pfad durch den Layered-Graphen rekonstruiert werden kann, wird der benutzte Knoten i an der Stelle $countLevel$ ins $usedNodeIdSet$ eingefügt (Zeile 10). Danach folgt der rekursive Funktionsaufruf (Zeile 11).

Handelt es sich aber um einen Intra-Layer-Nachbar (Zeile 15) wird die Variable $servChainCount$ nicht inkrementiert, da ja auch keine Service-Chain gemappt wurde. Ein weiterer Unterschied zwischen Intra-Layer- und Inter-Layer-Nachbarn ist, dass die Bandbreite eines Intra-Layer-Links an zwei Stellen der $adjMatrix$ aktualisiert werden muss, da es sich um einen bidirektionalen Link handelt (Zeile 17-19).

Diese rekursive Ausführung dauert solange, bis entweder der Zielknoten erreicht wurde $actNode = destNode$ oder keine neuen Nachbarn für $actNode$ gefunden werden.

Im Falle, dass der Zielknoten erreicht wurde (Zeile 27), wird der aktuelle Pfad ausgegeben. Die Ausgabe ist einfach, da alle gültigen Pfade im Array $allPath$ gespeichert sind. In dieses Array werden die Id's des aktuellen $usedNodeIdSet$ Arrays übertragen. Weiter werden auch die Forwarding-Kosten der Knoten des $usedNodeIdSet$ gespeichert. Dieses Umschreiben der Daten vom $usedNodeIdSet$ ins $allPath$ Array ist notwendig, da bei jeder Rekursion das Array $usedNodeIdSet$ verändert wird. Wichtig zu Wissen ist auch, dass der Algorithmus das Array $usedNodeIdSet$ nicht jedesmal neu formatiert, sondern einfach die Daten überschreibt und die aktuell gültige Grösse des Arrays anhand des Wertes von $countLevel$ ermittelt. Weiter wird der Delay des aktuellen Pfades berechnet (Zeile 28). Zuerst addiert man für $n \geq 1$ den Delay vom Link von Knoten $n-1$ zum Knoten n für alle Knoten n ausser dem letzten auf. Für den letzten Knoten $n = countLevel$ werden nur noch die Processing-Kosten addiert. Damit sind die Delay-Kosten aber noch nicht vollständig. Was noch fehlt sind die Forwarding-Kosten, falls welche auftreten (Zeile 29). Für deren Berechnung muss zuerst überprüft werden, ob Knoten überhaupt Datenpakete forwarden. Knoten, welche nur forwarden, besitzen zwei Vorgängerknoten, welche sich auf demselben Layer befinden, wie der Knoten selbst. Man muss also nur die Id dieser beiden Vorgängerknoten überprüfen, ob sie sich auf dem gleichen Layer befinden, dann weiss man, ob ein Knoten forwarded oder nicht. Der korrekte Layer lässt sich anhand der Id berechnen: $layer = id/countNode$. Für jeden Knoten im Pfad, der Datenpakete nur forwarded, wird im $nodeSet$ die korrespondierende $forwardTime$ abgefragt und aufaddiert. Die resultierenden totalen Forwarding-Kosten werden am Schluss zu den Delay-Kosten addiert. Anhand dieser Delay-Kosten wird überprüft, welches der momentane minimale Pfad ist (Zeile 30-31). Dieser wird am Ende des Algorithmus ausgegeben.

Falls keine neuen Nachbarn für den aktuellen Knoten $actNode$ gefunden werden oder die obige Ausgabe abgeschlossen wurde, wird ein Rekursionsschritt zurück gemacht. Dies bedeutet konkret, dass zuerst das $countLevel$ dekrementiert wird und dann der aktuelle Knoten $actNode$ wieder aus dem aktuellen Pfad entfernt wird (Inter-Layer-Link: Zeile 12, Intra-Layer-Link: Zeile 23). Als nächstes wird die Bandbreite des verwendeten Links auf allen Layern wieder hergestellt (Inter-Layer-Link: Zeile 13-14, Intra-Layer-Link: Zeile 24-26). Dafür wird der Link auf allen Layern gesucht, ob er existiert und falls dies der Fall ist, wird die im vorherigen Rekursionsschritt benutzte Bandbreite zu der momentanen Bandbreite addiert. Dies entspricht der Ausgangslage, wie sie vor dem Rekursionsschritt war.

Performance

Um den Speicherbedarf und den Zeitaufwand des Exhaustive-Search-Algorithmus zu analysieren, wird das Tools 'top' und die Funktion 'gettimeofday' verwendet. Es werden zwei Messungen gemacht :

Messung 1 In der ersten Messung werden die Bandbreiten der Links des Basis-Graphen aus Abbildung 3.18(a) in jedem Durchgang der Messung verändert. Ein Durchgang umfasst folgende Schritte:

1. Link(1,3) = Link(0,1) = Link(0,2) = i
2. Link(1,2) = Link(2,4) = i
3. Führe Algorithmus aus
4. Speichere Messresultate
5. Link(1,3) = Link(0,1) = Link(0,2) = i
6. Link(1,2) = Link(2,4) = i-50
7. Führe Algorithmus aus
8. Speichere Messresultate

Dieser Ablauf wird für i $\in$ I = $\{$50, 100, 150, 200, 250, 300$\}$ jeweils einmal ausgeführt.

Während den Messungen hat sich gezeigt, dass die Knotenzahl und die Linkzahl nicht die Kriterien sind, welche den Speicherbedarf und den Zeitaufwand des Algorithmus beeinflussen. Der Grund dafür liegt zum Beispiel darin, dass man Knoten im Basis-Graphen hat, welche nicht erreichbar sind, da ihre Links ausgelastet sind. Das gleiche gilt für die Links: Wenn man neue Links in den Basis-Graphen einfügt, deren Bandbreite aber zu klein ist für die benötigte Bandbreite der Service-Chain, ist, obwohl die Linkzahl grösser wird, der Speicherbedarf und der Zeitaufwand immer noch gleich gross.

Man verwendet zwei andere Bezugsgrössen, um eine Aussage über den Speicherbedarf und den Zeitaufwand machen zu können: die Rekursionstiefe und die Anzahl gültiger Pfade. Diesen Bezugsgrössen kann ein direkter Zusammenhang zum Speicherbedarf und dem Zeitaufwand nachgewiesen werden.

Die Daten von dieser Messung sind im File 'exhaustSearch_countLev_time.2.dat' gespeichert.

In der Abbildung 3.25(a) sieht man den Zusammenhang der Rekursionstiefe und der Ausführungszeit des Algorithmus.

Messung 2 In der zweiten Messung werden die Bandbreiten der Links des Basis-Graphen aus Abbildung 3.18(a) wie für den ersten Test in jedem Durchgang der Messung verändert. Ein Durchgang umfasst die folgenden Schritte:

1. Link(1,3) = Link(0,1) = Link(0,2) = **Link(1,5)** = i
2. Link(1,2) = Link(2,4) = i
3. Führe Algorithmus aus
4. Speichere Messresultate
5. Link(1,3) = Link(0,1) = Link(0,2) = **Link(1,5)** = i
6. Link(1,2) = Link(2,4) = i-50
7. Führe Algorithmus aus

3.3 Extended-Shortest-Path-Algorithmus

Die Aufgabe des Extended-Shortest-Path-Algorithmus ist dieselbe Aufgabe wie für den Exhaustive-Search-Algorithmus: Das Finden des kürzesten Pfades von einem Startknoten zu einem Zielknoten durch den Layered-Graphen. Im Gegensatz zum Exhaustive-Search-Algorithmus möchte man aber nicht alle möglichen Pfade berechnen, sondern durch einen 'intelligenten' Algorithmus direkt den kürzesten Pfad finden. So könnte eventuell Ausführungszeit gespart werden.

Der Extended-Shortest-Path-Algorithmus basiert auf dem Shortest-Path-Algorithmus von Dijkstra. In den folgenden Abschnitten wird zuerst der Shortest-Path-Algorithmus von Dijkstra erklärt und danach die Erweiterungen zum Extended-Shortest-Path Algorithmus erläutert. Am Ende der Theorie folgt jeweils ein kurzes Beispiel.

3.3.1 Dijkstra's Shortest-Path-Algorithmus

Der Shortest-Path-Algorithmus von Dijkstra findet den kürzesten Pfad von einem Startknoten aus zu einem Endknoten in einem Graphen. Der Algorithmus formt den Graphen zu einem sogenannten 'Spanning-Tree' um. Dieser Spanning-Tree enthält alle Punkte des Graphen und enthält keine Zyklen. Der Spanning-Tree beginnt beim Startknoten und berechnet für jeden Knoten die kürzeste Distanz zum Startknoten. Dafür werden folgende Datenstrukturen benötigt:

Datenstrukturen

- Graph $G = (V, E)$, wobei V ein Set von Knoten und E ein Set von Kanten darstellt,
- Zwei Sets von Knoten:
  - Das Set von Knoten S, dessen kürzester Pfad vom Startknoten schon berechnet worden ist
  - Die restlichen Knoten $V - S$.
- Zwei weitere Datenstrukturen:
  - Ein Array d mit den kürzesten Distanzen von jedem Knoten zum Startknoten
  - Ein Array pi mit dem Vorgängerknoten für jeden Knoten im Graphen.

Algorithmus

```
void dijkstra(Graph g, Node startNode)
01 ∀ nodes v ∈ g
02   d[v]=∞
03   pi[v]=none
04 d[startNode]=0
05 S={startNode}
06 while(V-S not empty) 07   find node q=min(V-S)
08   add node q to S
09   ∀ nodes v ∈ Neighbour(q)
10     if(d[v] > d[q]+LinkCost(q,v)
11       d[v]=d[q]+LinkCost(q,v)
12       pi[v]=q
```

Beschreibung

Zuerst werden die beiden Arrays d und pi initialisert (Zeile 01-03). Der Startknoten $startNode$ hat die Distanz $d[startNode] = 0$ zu sich selber (Zeile 04). Er ist der erste Knoten, von dem aus die Distanzen zu seinen Nachbarsknoten berechnet werden. Damit man weiss, dass er

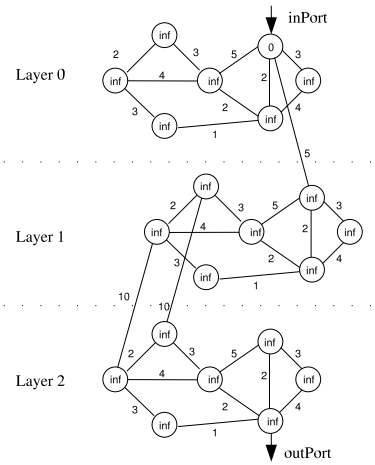


Abbildung 3.26: Die Initialisierung des Layered-Graphen nach Dijkstra's Shortest-Path-Algorithmus

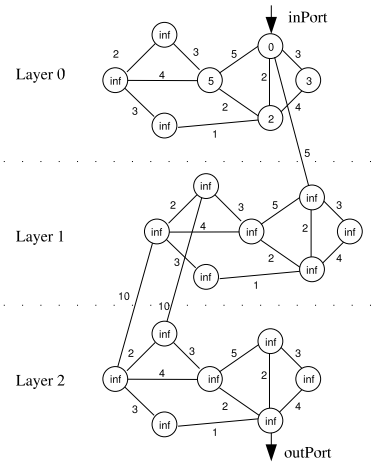


Abbildung 3.27: Der Layered-Graph nach dem zweiten Schritt

schon abgearbeitet worden ist, wird er in das Set S eingefügt (Zeile 05). Danach beginnt eine Schleife, die solange durchlaufen wird, bis das Set $V - S$ leer ist (Zeile 06). In der Schleife wird der Knoten q aus $V - S$ gesucht, der den kleinsten Wert in d gespeichert hat (Zeile 07). Hat man diesen Knoten gefunden, wird er ins Set S geschrieben (Zeile 08). Nun sucht man alle Nachbarn v von q und vergleicht, ob man durch das Benutzen des Knotens q einen besseren Wert für $d[v]$ erreichen kann. Ist dies der Fall, wird dieser Wert in $d[v]$ abgelegt und der Knoten q als Vorgängerknoten von v in $pi[v]$ gespeichert (Zeile 09-12).

Beispiel

In diesem Beispiel soll der Ablauf des Shortest-Path-Algorithmus von Dijkstra nochmals erläutert werden. Gegeben ist der Layered-Graph aus Abbildung 3.12. Im ersten Schritt des Algorithmus wird der Layered-Graph initialisiert. Dabei symbolisieren die Zahlenwerte innerhalb der Knoten den Wert d vom Knoten. In Abbildung 3.26 ist der Graph nach der Initialisierung dargestellt.

In einem zweiten Schritt werden die kürzesten Distanzen vom Startknoten aus zu seinen direkten Nachbarn bestimmt. Diese Werte werden in d gespeichert. Das Resultat kann in Abbildung 3.27 überprüft werden.

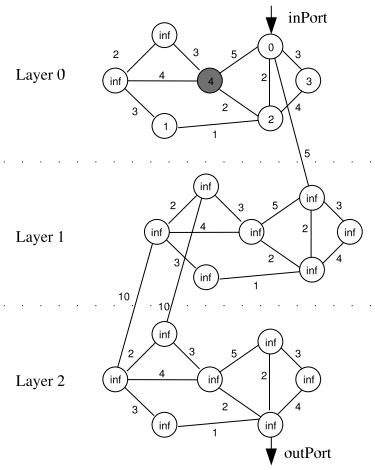


Abbildung 3.28: Der Layered-Graph nach dem dritten Schritt

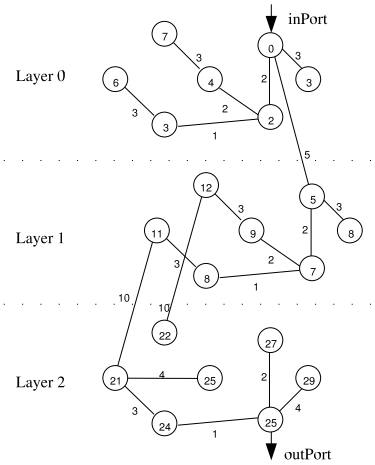


Abbildung 3.29: Der resultierende 'Spanning-Tree'

Als nächster Schritt wird der Knoten aus $V - S = \{\ Knoten2, Knoten5, Knoten6\ \}$ ausgewählt, der die kürzeste Distanz zum Startknoten hat: Knoten 5 mit $d[5] = 2$. Von diesem Knoten aus wird nun überprüft, ob einer der Nachbarsknoten via diesen Knoten günstiger erreicht werden kann. Im Beispiel hat der Knoten 2 vor dem Schritt die kürzeste Distanz $d[2] = 5$ zum Startknoten. Zieht man den Vergleich mit $d[5] + LinkCost(5, 2) = 4$ sieht man, dass der Knoten 2 via den Knoten 5 schneller erreichbar ist als vom Knoten 4 aus. Dies hat zur Folge, dass $d[2] = 4$ gesetzt wird und der Vorgängerknoten vom Knoten 2 $pi[2]$ von 4 auf 5 geändert wird. Diese Änderung ist in Abbildung 3.28 dokumentiert.

Der Algorithmus dauert so lange, bis alle Knoten abgearbeitet worden sind. Als Resultat erhält man einen 'Spanning-Tree', wie er in Abbildung 3.29 dargestellt ist.

3.3.2 Erweiterungen zum Extended-Shortest-Path-Algorithmus

Dem Shortest-Path-Algorithmus von Dijkstra fehlen zwei Eigenschaften, damit er als Mapping-Algorithmus genutzt werden kann:

1. Er muss berücksichtigen, dass Knoten auch Forwarding-Eigenschaften besitzen. Das heisst, wenn ein Knoten nicht als Inter-Link-Knoten verwendet wird, kommen zusätzlich zu den Delay-Kosten des Links noch die Fowarding-Kosten des Knotens dazu. Erst anhand dieser Summe kann entschieden werden, ob ein Umweg über einen anderen Knoten

tatsächlich günstiger ist als der momentan bestehende Weg.

2. Dijkstra's Shortest-Path-Algorithmus berücksichtigt keine Einschränkungen bei den Kapazitäten der Links. Dem Algorithmus fehlt ein Test, ob die benötigte Bandbreite für die Service-Chain die verfügbare Bandbreite des Links nicht überschreitet. Dieser Test muss immer durchgeführt werden, wenn ein Knoten über einen Link erreicht werden soll. Ebenfalls muss eine Aktualisierung der Bandbreite stattfinden, falls ein Link benutzt wurde. Diese Aktualisierung muss innerhalb des gesamten Layered-Graphen stattfinden, da Links auf verschiedenen Layer einen einzigen realen Link symbolisieren können.

Forwarding-Kosten

Um die Forwarding-Kosten zu berücksichtigen, muss man zuerst feststellen können, wann ein Knoten q forwarded und wann nicht. Die Tatsache, dass ein Knoten forwarded, beinhaltet die Konsequenz, dass sich der Vorgängerknoten $pi[q]$, der aktuelle Knoten q und der Nachfolgeknoten v alle auf dem selben Layer L_i befinden. Ist dies nicht der Fall, kann man mit Sicherheit sagen, dass der aktuelle Knoten nicht forwarded. Mit diesem Wissen kann man zwei Fälle unterscheiden:

1. Der aktuelle Knoten q forwarded. Das bedeutet, dass für einen Vergleich der momentan kürzesten Distanz für jeden Nachbar v vom aktuellen Knoten mit der Distanz, die sich ergibt, wenn man einen Weg über den aktuellen Knoten wählt, die Forwading-Kosten miteinbezogen werden müssen.

2. Der aktuelle Knoten q forwarded nicht. Als Folge davon haben die Forwarding-Kosten keinen Einfluss auf die Berechnung der kürzesten Distanz.

Diese neue Funktionalität benötigt folgende Erweiterung des Dijkstra's Shortest-Path-Algorithmus:

```
void extended_Shortest_Path(Graph g, Node startNode)
01 ∀ nodes v ∈ g
02   d[v]=∞
03   pi[v]=none
04 d[startNode]=0
05 S={startNode}
06 while(V-S not empty) 07   find node q=min(V-S)
08   add node q to S
09   ∀ nodes v ∈ Neighbour(q)
**********************NEW!**********************************
+1    if((v,q∈ L_i)and(q,pi[q]∈ L_i))
+2      if(d[v] > d[q]+LinkCost(q,v)+ForwardCost(q))
+3        d[v]=d[q]+LinkCost(q,v)+ForwardCost(q)
+4        pi[v]=q
+5    else
************************************************************
10      if(d[v] > d[q]+LinkCost(q,v))
11        d[v]=d[q]+LinkCost(q,v)
12        pi[v]=q
```

Capacity-Tracking

Um die Einschränkungen der Kapazitäten der Links nicht zu verletzten, wird das Verfahren des Capacity-Trackings verwendet, welches Sumi Y. Choi in ihrem Paper [1] beschrieben hat. Bei diesem Verfahren wird, bevor man einen Link dem kürzesten Pfad zu einem Knoten hinzufügt, der Pfad vom aktuellen Knoten zum Startknoten zurückverfolgt und dabei jedes Mal, wenn dieser Link verwendet wird, die Bandbreite aufsummiert. Am Schluss wird beim Startknoten überprüft, ob die Summe der Bandbreite die verfügbare Bandbreite des Links nicht

überschreitet. Ist dies der Fall, wird der neue Link dem kürzesten Pfad hinzugefügt. Ist die Summe der Bandbreite aber zu gross, kann der Link nicht mehr verwendet werden und der Knoten ist nicht über diesen Link erreichbar.

Der erweiterte Dijkstra's Shortest-Path-Algorithmus sieht folgendermassen aus:

```
void extended_Shortest_Path(Graph g, Node startNode)
01 ∀ nodes v ∈ g
02   d[v]=∞
03   pi[v]=none
04 d[startNode]=0
05 S={startNode}
06 while(V-S not empty) 07   find node q=min(V-S)
08   add node q to S
09   ∀ nodes v ∈ Neighbour(q)
10     if(d[v] > d[q]+LinkCost(q,v))
**********************NEW!*********************************
+1       sumBandwith=0
+2       testNode=q
+3       while(testNode≠startNode)
+4         testNode=pi[testNode]
+5         if Link(testNode,pi[testNode])=Link(q,v)
+6           sumBandwith=sumBandwith+Link(testNode,pi[testNode]).bandwith
+7       if(sumBandwith≤Link(q,v).bandwith
************************************************************
11         d[v]=d[q]+LinkCost(q,v)
12         pi[v]=q
```

3.3.3 Implementierung

Die benötigten Datenstrukturen sehen folgendermassen aus:

bestCurrDist Dieses Array speichert für jeden Knoten die momentan kürzeste Distanz zum Startknoten.

predNode Für jeden Knoten wird in diesem Array der Vorgängerknoten gespeichert, über den man den Knoten auf dem kürzesten Pfad erreichen kann.

usedNodeIdSet Damit man weiss, welche Knoten schon benutzt wurden, schreibt man deren Id in das usedNodeIdSet-Array.

usedNodeCount Diese Hilfsvariable zählt die benutzten Knoten. Ist usedNodeCount gleich der Anzahl Knoten im Layered-Graphen wird der Algorithmus beendet, da jeder Knoten abgearbeitet wurde.

Algorithmus

```
void extendedShortestPath(service service1)
01 init predNode[]
02 init bestCurrDist[]
03 actNode=startNode
04 while(usedNodeCount≠#Nodes∈LayeredGraph)
05   add actNode to usedNodeSet
06   ∀ Nodes i ∈ Neighbour(actNode) and not ∈ usedNodeSet
07     if(Link(actNode,i)=InterLayerLink)
08       if(Link(actNode,i).bandwith≥servChain.maxProcCost)
09         if(bestCurrDist[i]=not defined or bestCurrDist[i]>bestCurrDist[actNode]+Link(actNode,i).delay)
```

```
10        testNode=actNode
11        sumBandwith=servChain.maxBandwith
12        while(testNode≠startNode)
13          if(Link(testNode,predNode[testNode])=Link(actNode,i) or
13          Link(predNode[testNode],testNode)=Link(actNode,i))
14            sumBandwith=sumBandwith+servChain.maxBandwith
15          testNode=predNode[testNode]
16        if(sumBandwith≤Link(actNode,i).bandwith)
17          bestCurrDist[i]=bestCurrDist[actNode]+Link(actNode,i).delay
18          predNode[i]=actNode
19    else
20      if(Link(actNode,i).bandwith≥servChain.maxProcCost)
21        if(Level(actNode)=Level(predNode[actNode]) and Level(i)=Level(actNode))
22          if(bestCurrDist[i]=not defined or
22          bestCurrDist[i]>bestCurrDist[actNode]+Link(actNode,i).delay+actNode.forwardTime)
23            testNode=actNode
24            sumBandwith=servChain.maxBandwith
25            while(testNode≠startNode)
26              if(Link(testNode,predNode[testNode])=Link(actNode,i) or
26              Link(predNode[testNode],testNode)=Link(actNode,i))
27                sumBandwith=sumBandwith+servChain.maxBandwith
28              testNode=predNode[testNode]
29            if(sumBandwith≤Link(actNode,i).bandwith)
30              bestCurrDist[i]=bestCurrDist[actNode]+Link(actNode,i).delay+actNode.forwardTime
31              predNode[i]=actNode
32        else
33          if(bestCurrDist[i]=not defined or
33          bestCurrDist[i]>bestCurrDist[actNode]+Link(actNode,i).delay)
34            testNode=actNode
35            sumBandwith=servChain.maxBandwith
36            while(testNode≠startNode)
37              if(Link(testNode,predNode[testNode])=Link(actNode,i) or
37              Link(predNode[testNode],testNode)=Link(actNode,i))
38                sumBandwith=sumBandwith+servChain.maxBandwith
39              testNode=predNode[testNode]
40            if(sumBandwith≤Link(actNode,i).bandwith)
41              bestCurrDist[i]=bestCurrDist[actNode]+Link(actNode,i).delay
42              predNode[i]=actNode
43  minVar=∞
44  ∀ Nodes j ∈ LayeredGraph and not ∈ usedNodeSet
45    if(bestCurrDist[j]<minVar and bestCurrDist[j] ≠ undefined)
46      minVar=bestCurrDist[j]
47      actNode=j
48 bestCurrDist[destNode]=bestCurrDist[destNode]+destNode.procTime
```

Beschreibung

Zuerst werden die beiden Arrays initialisiert (Zeile 01-02). Der Startknoten hat den momentan kürzesten Weg 0 und sein Vorgängerknoten wird auf -1 gesetzt, was bedeuten soll, dass er keinen Vorgängerknoten besitzt. Da alle anderen Knoten auch noch keine Vorgängerknoten besitzen, werden auch diese Vorgängerknoten auf $predNode[i] = -1$ gesetzt. Ihre momentan kürzeste Distanz wird auf $bestCurrDist = -1$ gesetzt, damit der Algorithmus erkennt, dass noch keine momentan kürzeste Distanz für diese Knoten berechnet wurde.

Der Algorithmus startet nun beim Startknoten (Zeile 03) und sucht nach einem Nachbarn, welcher noch nicht abgearbeitet wurde (Zeile 06). Für diesen Nachbarn wird überprüft, ob es sich um einen Inter-Layer oder einen Intra-Layer-Nachbarn handelt (Zeile 07). Diese beiden Fälle müssen wie in Abschnitt 3.2 differenziert werden.

Handelt es sich um einen Inter-Layer-Nachbarn, wird getestet, ob die verfügbare Bandbreite des Links der notwendigen Bandbreite genügt (Zeile 08). Ist dieser Test erfolgreich, wird berechnet, ob man den Knoten i via den aktuellen Knoten $actNode$ schneller erreichen kann (Zeile 09). Ist dies der Fall oder ist die kürzeste Distanz $bestCurrDist[i]$ vom Knoten i zum Startknoten $startNode$ noch gar nicht definiert, könnte die berechnete $bestCurrDist[i]$ als kürzeste Distanz vom Knoten i zum Startknoten $startNode$ eingesetzt werden. Bevor dies aber gemacht werden kann, muss kontrolliert werden, ob keine Bandbreiten überlastet werden. Dies geschieht mit dem, in Abschnitt 3.3.2 beschriebenen, Capacity-Tracking. Zuerst wird eine Hilfsvariable $testNode$ definiert, welche man verwendet, um den kürzesten Pfad von unten nach oben zu durchlaufen (Zeile 10). Weiter benötigt man die Variable $sumBandwith$, welche die im kürzesten Pfad verwendeten Bandbreiten des Links $Link(actNode, i)$ aufsummiert (Zeile 11). Das Capacity-Tracking wird solange durchgeführt, bis vom $actNode$ aus der Startknoten erreicht wird (Zeile 12). Immer wenn ein Link $Link(testNode, pred[testNode])$ dem aktuellen Link $Link(actNode, i)$ entspricht (Zeile 13), wird die Bandbreite $sumBandwith$ um den Wert erweitert, den der Link für die Service-Chain aufbringen musste (Zeile 14). Ist der Startknoten erreicht worden, wird überprüft, ob die total benötigte Bandbreite $sumBandiwth$ die verfügbare Bandbreite des Links nicht übersteigt (Zeile 16). Ist noch genug Bandbreite vorhanden, um den Link zu benutzen, wird jetzt die neue kürzeste Distanz $bestCurrDist[i]$ vom Knoten i zum Startknoten eingesetzt (Zeile 17). Zusätzlich wird der Vorgängerknoten $actNode$ des Knotens i im Array $predNode$ eingetragen (Zeile 18).

Ist der Nachbar ein Intra-Layer-Nachbar (Zeile 19), wird wie beim-Inter-Layer Nachbar überprüft, ob genügend Bandbreite vorhanden ist (Zeile 20). Ist dies der Fall, werden weiter zwei Fälle unterschieden: Der aktuelle Knoten $actNode$ leitet Pakete nur weiter (Forwarding Knoten) oder beim aktuellen Knoten handelt es sich um einen Processing Knoten, auf den eine Service-Chain gemappt wird. Dieser Test wird folgendermassen umformuliert: Ein Knoten, der ein Forwarding Knoten ist, befindet sich auf dem gleichen Layer, wie sein Vorgängerknoten und sein Nachfolgeknoten. Um zu bestimmen, ob der Knoten $actNode$ forwarded, wird also überprüft, ob sich der Vorgängerknoten $predNode[actNode]$, der aktuelle Knoten $actNode$ und der mögliche Nachfolgeknoten i alle auf dem gleichen Layer befinden (Zeile 21). Diese zwei Fälle müssen unterschieden werden, da, falls ein Knoten forwarded, zusätzlich zu den Link-Kosten noch die Forwarding-Kosten berücksichtigt werden müssen. Diese Forwarding-Kosten haben, wie die Link-Kosten einen direketen Einfluss auf die Berechnung von $bestCurrDist$ (Zeile 22 und Zeile 30). Die Berechnung der kürzesten Distanz zum Startknoten ist, bis auf einen kleinen Unterschied im Capacity-Tracking, für beide Fälle (Forwarding Knoten: Zeile 20-31, Processing Knoten: Zeile 32-42) identisch zur Berechnung bei einem Inter-Layer-Nachbarn. Der kleine Unterschied besteht darin, dass bei einem Intra-Layer-Nachbarn ein Link in beide Richtungen überprüft werden muss, damit das Capacity-Tracking korrekt ist. Der Grund dafür liegt nahe: Intra-Layer-Links sind bidirektional, Inter-Layer-Links hingegen unidirektional.

Sind für den aktuellen Knoten die Distanzen zu all seinen Nachbarn überprüft und allenfalls aktualisiert worden, wird der nächste Knoten j selektiert. Dies geschieht aufgrund des minimalen Wertes von $bestCurrDist[j]$. Es wird also der Knoten j selektiert, welcher noch nicht abgearbeitet wurde (Zeile 44) und die kleinste Distanz zum Startknoten aufweist (Zeile 45). Knoten, deren kürzeste Distanz noch nicht berechnet worden ist, können nicht selektiert werden. Hat man diesen Knoten j gefunden, wird er als neuer aktueller Knoten $actNode$ verwendet (Zeile 47).

Um den Layered-Graphen um ein Layer zu verkleinern, wurde der letzte Inter-Layer-Link (entspricht der letzten Service-Chain) nicht im Layered-Graphen eingetragen. Darum wird die Processing-Zeit $destNode.procTime$ des Zielknotens $destNode$ zu der kürzesten Distanz $bestCurrDist[destNode]$ vom Zielknoten zum Startknoten addiert (Zeile 48).
Das Resultat des Algorithmus ist ein reduzierter Graph (Spanning-Tree), der keine Zyklen mehr

Idee

Die Ausgangslage für die Methode ist ein reduzierter Graph, bei dem der Zielknoten nicht erreicht wurde. Von diesem Zielknoten aus werden Nachbarn gesucht, für die mittels Dijkstra's Algorithmus eine kürzeste Distanz zum Startknoten berechnet wurde. Gibt es solche Nachbarn, wird für jeden von ihnen der Link zwischen Nachbar und Zielknoten als kritisch markiert und der Shortest-Path-Algorithmus nochmals ausgeführt. Der Shortest-Path-Algorithmus wird um die Funktionalität erweitert, dass er kritische Links nicht selektieren kann. Hat der Zielknoten aber keine Nachbarn, deren kürzeste Distanz zum Startknoten berechnet worden ist, wird einer der Nachbarn ausgewählt. Von ihm aus werden dann wieder alle Nachbarn bestimmt, deren Shortest Path schon berechnet wurde. Dies wird für jeden Nachbar ausgeführt. Das Abbruchkriterium für diese Exhaustive Search Methode muss noch definiert werden. Eine Möglichkeit wäre, dass man die Suchtiefe beschränkt. Die Exhaustive-Search-Methode würde dann nur bis zu dieser Suchtiefe nach einem erreichbaren Knoten suchen. Das Resultat dieser Methode sind mehrere kürzeste Pfade vom Start zum Zielknoten, von denen der kürzeste selektiert wird.
Der Extended-Shortest-Path-Algorithmus mit Capacity-Tracking und Reverse-Exhaustive-Search sieht dann folgendermassen aus:

```
void extended_Shortest_Path(Graph g, Node startNode)
01 ∀ nodes v ∈ g
02   d[v]=∞
03   pi[v]=none
04 d[startNode]=0
05 S={startNode}
06 while(V-S not empty) 07   find node q=min(V-S)
08   add node q to S
09   ∀ nodes v ∈ Neighbour(q)
************************************NEW*************************************
+1     if(Link(q,v)≠critical)
*****************************************************************************
11       if(d[v] > d[q]+LinkCost(q,v))
12         sumBandwith=0
13         testNode=q
14         while(testNode≠startNode)
15           testNode=pi[testNode]
16           if Link(testNode,pi[testNode])=Link(q,v)
17             sumBandwith=sumBandwith+Link(testNode,pi[testNode]).bandwith
18         if(sumBandwith≤Link(q,v).bandwith
19           d[v]=d[q]+LinkCost(q,v)
20           pi[v]=q
```

Der Aufruf des Algorithmus hat folgende Struktur:

```
01 extendedShortestPath(g,startNode)
02 critNode=destNode
03 while(bestCurrDist[destNode]=undefined)
04   find critLayer L_c
05   find Node i ∈ neighbour(critNode) and bestCurrDist[i]≠undefined
06   ∀ Layers L_j, j<c
07     Link(critNode,i).critical=true
08     Link(i,critNode).critical=true
09   extendedShortestPath(g,startNode)
10   ∀ Layers L_j, j<c
11     Link(critNode,i).critical=false
12     Link(i,critNode).critical=false
```

```
Modus ist Shortest Path
Startknoten: 3
New bestCurrDist[9] = 4
New predNode[9] = 3
No valid Path found!
```

Abbildung 3.33: Der Output für das Programm 'extendShoPathTest1_1'

Der Extended-Shortest-Path mit Capacity-Tracking und Reverse-Exhaustive-Search wird zuerst einmal ausgeführt (Zeile 01). Dann wird überprüft, ob der Zielknoten erreicht wurde (Zeile 03). Falls nicht, wird nach einem Nachbarsknoten i des Zielknotens $destNode$ gesucht, dessen kürzeste Distanz zum Startknoten schon berechnet wurde (Zeile 05). Danach werden die Links vom Zielknoten zum Nachbarn i des Zielknotens auf allen Layern überhalb des kritischen Layers $critLayer$, auf dem sich der kritische Knoten $critNode$ befindet, als kritisch markiert(Zeile 06 - Zeile 08) und der Algorithmus noch einmal gestartet (Zeile 09). Am Schluss (Zeile 10 - Zeile 12) werden die im vorherigen Schritt als kritisch gesetzten Links wieder als unkritisch markiert.

3.3.6 Evaluation

In diesem Abschnitt soll zuerst der Algorithmus wie in Abschnitt 3.2.3 anhand eines Beispiels verifiziert werden und dann bezüglich Speicherbedarf und Zeitaufwand untersucht werden.

Verifikation

Um die Funktionsweise des Algorithmus zu verifizieren, wird vom Basis-Graphen aus Abbildung 3.18(a) und dem Service-Graphen von Abbildung 3.18(b)ausgegangen. Die Parameter des Basis-Graphen und des Service-Graphen sind die gleichen wie in Abschnitt 3.2.3.

In Abbildung 3.19 ist der Layered-Graph dargestellt, welcher aus den gemachten Spezifikationen resultiert.

Die gesammten Informationen sind im Programm 'extendShoPathTest1_1' im Verzeichnis '/ribolla/code/exhauSearch/test' gespeichert. Falls der Leser den Test selber durchführen möchte, kann er dies durch das Ausführen des Programms mit dem Parameter 2 tun. Der Befehl lautet:

./extendShoPathTest1 2

Die Überprüfung des Extended-Shortest-Path-Algorithmus erfolgt indem, dass für jeden Berechnungsschritt die neue momentan kürzeste Distanz zum Startknoten $bestCurrDist$ und der neue Vorgängerknoten $predNode$ ausgegeben werden. In der Abbildung 3.33 sieht man den Output des Algorithmus für den obigen Test. Da absichtlich die Bandbreiten der Links des Basis-Graphen zu klein gewählt wurden, nämlich 10 anstatt 50, wird vom Startknoten aus nur der Knoten 9 erreicht. Darum wird auch nur für diesen Knoten die momentan kürzeste Distanz zum Startknoten $bestCurrDist$ und der Vorgängerknoten $predNode$ bestimmt. Mit der aktuellen Konfiguration des Basis-Graphen findet der Algorithmus korrekterweise keine Lösung.

In einem zweiten Programm 'extendShoPathTest1_2' wird die Konfiguration des Basis-Graphen nun wie folgt abgeändert:

- Link(1,3).bandwith = 50
- Link(1,2).bandwith = 50
- Link(2,5).bandwith = 50

Daraus folgt der neue Layered-Graph, wie man ihn in Abbildung 3.21 sieht.

```
Modus ist Shortest Path
Startknoten: 3
New bestCurrDist[1] = 4
New predNode[1] = 3
New bestCurrDist[9] = 4
New predNode[9] = 3
New bestCurrDist[2] = 10
New predNode[2] = 1
New bestCurrDist[7] = 6
New predNode[7] = 9
New bestCurrDist[8] = 12
New predNode[8] = 7
New bestCurrDist[13] = 14
New predNode[13] = 7
New bestCurrDist[5] = 16
New predNode[5] = 2
New bestCurrDist[11] = 18
New predNode[11] = 8
New bestCurrDist[14] = 16
New predNode[14] = 13
New bestCurrDist[17] = 22
New predNode[17] = 14
 17  14  13  7  9  3
predNode:  -1  3  1  -1  -1  2  -1  9  7  3  -1  8  -1  7  13  -1  -1  14
bestCurrDist zum Zielknoten:  26
```

Abbildung 3.34: Der Output des Programms ’extendShoPathTest1_2’

Durch die obigen Änderungen wurde der Layered-Graph so verändert, dass genau ein Pfad durch den Layered-Graphen existiert. Wenn man den Output vom Programm ’extendShoPathTest1_2’ betrachtet, sieht man am Resultat, dass der Algorithmus einen gültigen Pfad findet. Der gefundene Pfad entspricht der Knotenfolge $\{17,14,13,7,9,3\}$ in umgekehrter Reihenfolge. Dieser Pfad verwendet die Links, deren Bandbreite von 10 auf 50 erhöht wurden. Im Array $predNode$ sieht man anhand der Indizes an den Stellen, wo sich eine ’-1’ befindet, dass für diese Knoten keine kürzeste Distanz zum Startknoten gefunden wurde.

Anhand des Outputs von Abbildung 3.34 werden die Berechnungsschritte dem Programm-Code aus Abschnitt 3.3.3 zugeordnet. Dies soll helfen zu zeigen, dass der Algorithmus korrekt funktioniert. In Abbildung 3.35 sieht man diese Zuordnung Zeile für Zeile. Die Initialisierung der Strukturen für den Algorithmus und die Selektion des neuen aktuellen Knotens $actNode$ wurden dabei vernachlässigt.

In einem dritten Programm ’exhaustSearchTest1_3’ werden die Bandbreiten aller Links des Basis-Graphen auf 50 gesetzt. Dies hat zur Folge, dass nun alle Knoten erreichbar sind. Dies kann man dem Array $predNode$ aus der Abbildung 3.36 entnehmen. Es sind keine ’-1’ Einträge im Array vorhanden, mit Ausnahme des Eintrags beim Knoten 3. Der Knoten 3 entspricht dem Startknoten und hat deshalb korrekterweise keinen Vorgängerknoten.

Performance

Um den Speicherbedarf und den Zeitaufwand des Extended-Shortest-Path-Algorithmus zu analysieren, wird wie für den Exhaustive-Search-Algorithmus das Tool ’top’ und die Funktion ’gettimeofday’ verwendet. Bei den folgenden Messungen wird die Implementierung der Reverse-Exhaustive-Search Funktionalität aus Abschnitt 3.3.5 entfernt, damit sie die Messungen nicht beeinflusst. Es werden zwei Messungen gemacht :

Messung 1 In der ersten Messung werden die Bandbreiten der Links des Basis-Graphen aus Abbildung 3.18(a) in jedem Durchgang der Messung verändert. Ein Durchgang umfasst folgende Schritte:

1. Link(1,3) = Link(0,1) = Link(0,2) = i
2. Link(1,2) = Link(2,4) = i

	Modus ist Shortest Path
	Startknoten: 3
Zeile 21 – 31	New bestCurrDist[1] = 4
Zeile 21 – 31	New predNode[1] = 3
Zeile 07 – 18	New bestCurrDist[9] = 4
Zeile 07 – 18	New predNode[9] = 3
Zeile 21 – 31	New bestCurrDist[2] = 10
Zeile 21 – 31	New predNode[2] = 1
Zeile 33 – 42	New bestCurrDist[7] = 6
Zeile 33 – 42	New predNode[7] = 9
Zeile 21 – 31	New bestCurrDist[8] = 12
Zeile 21 – 31	New predNode[8] = 7
Zeile 07 – 18	New bestCurrDist[13] = 14
Zeile 07 – 18	New predNode[13] = 7
Zeile 21 – 31	New bestCurrDist[5] = 16
Zeile 21 – 31	New predNode[5] = 2
Zeile 21 – 31	New bestCurrDist[11] = 18
Zeile 21 – 31	New predNode[11] = 8
Zeile 21 – 31	New bestCurrDist[14] = 16
Zeile 21 – 31	New predNode[14] = 13
Zeile 21 – 31	New bestCurrDist[17] = 22
Zeile 21 – 31	New predNode[17] = 14
	17 14 13 7 9 3
	predNode: -1 3 1 -1 -1 2 -1 9
	7 3 -1 8 -1 7 13 -1 -1 14
Zeile 51	bestCurrDist zum Zielknoten: 26

Abbildung 3.35: Die Zuordnung des Outputs von Programm ’extendShoPathTest1_2’ zum Programm-Code des Extended-Shortest-Path-Algorithmus

```
Modus ist Shortest Path
Startknoten: 3
New bestCurrDist[1] = 4
New predNode[1] = 3
New bestCurrDist[9] = 4
New predNode[9] = 3
New bestCurrDist[0] = 10
New predNode[0] = 1
New bestCurrDist[2] = 10
New predNode[2] = 1
New bestCurrDist[7] = 6
New predNode[7] = 9
New bestCurrDist[6] = 12
New predNode[6] = 7
New bestCurrDist[8] = 12
New predNode[8] = 7
New bestCurrDist[13] = 14
New predNode[13] = 7
New bestCurrDist[4] = 16
New predNode[4] = 2
New bestCurrDist[5] = 16
New predNode[5] = 2
New bestCurrDist[10] = 18
New predNode[10] = 8
New bestCurrDist[11] = 18
New predNode[11] = 8
New bestCurrDist[12] = 16
New predNode[12] = 13
New bestCurrDist[14] = 16
New predNode[14] = 13
New bestCurrDist[16] = 22
New predNode[16] = 14
New bestCurrDist[17] = 22
New predNode[17] = 14
 17  14  13  7  9  3
predNode:  1  3  1  -1  2  2  7  9  7  3  8  8  13  7  13  -1  14  14
bestCurrDist zum Zielknoten:  26
```

Abbildung 3.36: Der Output für das Programm ’extendShoPathTest1_3’

3. Führe Algorithmus aus
4. Speichere Messresultate
5. Link(1,3) = Link(0,1) = Link(0,2) = i
6. Link(1,2) = Link(2,4) = i-50
7. Führe Algorithmus aus
8. Speichere Messresultate

Dieser Ablauf wird für i $\in$ I = $\{$50, 100, 150, 200, 250, 300$\}$ jeweils einmal ausgeführt.

In der Hoffnung, dass man in später den Vergleich zwischen dem Exhaustive-Search-Algorithmus und dem Extended-Shortest-Path-Algorithmus ziehen kann, werden die Rekursionstiefe und die Pfadzahl als Bezugsgrössen verwendet. Weil diese beiden Grössen für den Extended-Shortest-Path nicht definiert sind, werden die Konfigurationen des Basis-Graphen und des Service-Graphen verwendet, welche eine bestimmte Rekursionstiefe und Pfadzahl für den Exhaustive-Search-Algorithmus bewirken. Im weiteren Verlauf dieser Messungen wird darum von diesen beiden Bezugsgrössen gesprochen, obwohl sie für den Extended-Shortest-Path nicht existieren.

Die Resultate ergaben, dass die Performance des Extended-Shortest-Path-Algorithmus unabhängig von der Rekursionstiefe und der Pfadzahl ist. Es wurden für alle Rekursionstiefen ein konstanter Wert von 0.15 ms gemessen. Der gemessene Wert für den Speicherbedarf lag konstant bei 848kB.

Messung 2 In der zweiten Messung wird die Abhängigkeit des Extended-Shortest-Path-Algorithmus von der Anzahl Knoten untersucht. Dafür wird die Knotenzahl eines Basis-Graphen konstant erhöht. Der Basis-Graph hat folgende Eigenschaften:

- Jeder Knoten ist mit jedem anderen Knoten über einen Link verbunden
- Es existieren drei Prozessortypen (Knotentypen)
- Von jedem Prozessortyp gibt es gleich viele Prozessoren (Knoten)
- Alle Links, ausser den Links zum Zielknoten, haben die gleiche Bandbreite 50 und den gleichen Delay 2
- Damit es keinen gültigen Pfad wird die Bandbreite der Links zum Zielknoten auf 10 gesetzt
- Für die Prozessoren gelten die Angaben aus Abschnitt 3.2.3

Der zu mappende Service ist der Service Graph aus Abbildung 3.18(b).

Die Anzahl Knoten im Basis-Graphen entspricht n $\in$ N = $\{$9, 45, 99, 210, 399, 810, 1500$\}$. Für jedes n $\in$ N wird die Messung einmal durchgeführt.

Die Daten für den Speicherbedarf in Abhängigkeit von der Knotenzahl sind im File 'extendSP_Node_Mem_2.dat' im Verzeichnis '/ribolla/code/exhauSearch/performance' gespeichert. Im File 'extendSP_Node_Time_2.dat' befinden sich die Resultate für den Zeitaufwand in Abhängigkeit von der Knotenzahl. In der Abbildung 3.37 sieht man die Visualisierung der Resultate.

Fazit

Aufgrund der verschiedenen Referenzgrössen lässt es sich nur schwer einen Vergleich zwischen dem Exhaustive-Search-Algorithmus und dem Extended-Shortest-Path-Algorithmus zu ziehen. Jeder Algorithmus hat für bestimmte Topologien eines Prozessor-Graphes seine Vorteile gegenüber dem anderen Algorithmus.

verhindern. Die Tatsache, dass mit diesem Lösungsansatz speziell für grosse Service-Graphen selten eine Lösung gefunden wird, auch wenn eine Lösung existieren würde, wird in den Papern nicht erwähnt.

Weiter wird im Paper [1] der Ansatz des ’Capacity-Tracking’-Algorithmus vorgestellt. Dieser Ansatz dient als Basis für den Ansatz des Capacity-Trackings von Abschnitt 3.3.2. Die Funktionsweise ist prinzipiell dieselbe. Im Paper konzentrieren sich Sumi Y. Choi und Jonathan Turner aber auf die Anwendung auf so genannte ’unicast sessions’. Darunter versteht man einen Layered-Graphen, welcher nur einen Inter-Layer-Link pro Layer aufweist. In dieser Master Thesis wurde diese Einschränkung aufgehoben und das Capacity-Tracking so umformuliert, dass es auf beliebige Layered-Graphen angewendet werden kann.

Sumi Y. Choi und Jonathan Turner haben als Beispiel für einen Algorithmus, welcher den kürzesten Pfad durch den Layered-Graphen finden soll, den Shortest-Path von Dijkstra erwähnt. Es wurden aber keine Hinweise gemacht, wie dieser Algorithmus implementiert werden muss, so dass er auf den Layered-Graphen angewandt werden kann, ohne dass die Einschränkungen der Bandbreiten verletzt werden.

Kapitel 4

Optimierungen

In diesem Kapitel wird die Optimierung des Exhaustive-Search-Algorithmus und des Extended-Shortest-Path-Algorithmus erläutert. Es werden verschiedene Lösungsansätze präsentiert und implementiert.

Damit die Schnittstellen zum PromethOS NP klar definiert sind, sollen die Definitionen des Basis-Graphen und des Service-Graphen in seperaten Header Files implementiert werden. Diese Änderung soll eine spätere Einbindung des Mapping-Algorithmus in den PromethOS NP erleichtern. Das Design und die Implementierung findet man in Abschnitt 4.1.

In einem ersten Lösungsansatz werden die Arrays durch Pointer ersetzt und anstatt einer Struktur, welche den gesammten Layered-Graphen speichert, werden zwei Datenstrukturen implementiert, wovon die eine die Intra-Layer-Links und die andere die Inter-Layer-Links speichert. Die Implementierung und die Evaluation für diesen optimierten Layered-Graphen findet man im Abschnitt 4.2.

Anhand des optimierten Layered-Graphen wird die Funktionsweise des Exhaustive-Search-Algorithmus und des Extended-Shortest-Path-Algorithmus den neuen Datenstrukturen angepasst. Die Lösungsansätze, die Implementierungen, sowie die Evaluation findet man im Abschnitt 4.3 für den optimierten Exhaustive-Search-Algorithmus und im Abschnitt 4.4

Weiter wird die Möglichkeit zur Knotenreduktion im Layered-Graphen untersucht, um so ebenfalls eine Verbesserung der Performance zu erreichen. Die Ideen und Resultate basieren auf dem Paper [4]sind in Abschnitt 4.5 aufgeführt.

4.1 Header Files

In der Abbildung 4.1 sieht man die beiden Schnittstellen ’graph.h’ und ’service.h’. Sie liefern den Mapping-Algorithmen einen Zugang zu den Definitionen des Basis-Graphen und des Service-Graphen. Diese Definitionen sind in den beiden Files ’graph.cc’ und ’service.cc’ implementiert. Bei der Einbindung der Algorithmen in den PromethOS NP fallen diese beiden Files weg und werden durch den Zugriff auf die Ressource-Information-Database (RId) ersetzt.

4.1.1 Die Schnittstelle ’graph.h’

Diese Schnittstelle definiert die verwendeten Strukturen für die Knoten und für die Links. Weiter kann die Anzahl der drei Prozessortypen hier festgelegt werden. Es werden drei Funktionen zur Verfügung gestellt:

makeNodes() Definiert die Knoten im Basis-Graphen und initialisert sie mit einer Id und den gewünschten Parametern.

4.2.1 Optimierter Layered Graph

Der optimierte Layered-Graph besteht neu aus zwei Datenstrukturen. Dies sind die Strukturen 'link** intraLinks' und 'node*** interLinks'. $intraLinks$ speichert die Links des Basis-Graphen. Zusammen mit der Struktur $interLinks$, welche die Inter-Layer-Links speichert, bilden sie den Layered-Graphen.

Die Funktion 'makeIntra()', welche von der 'graph.h' Schnittstelle zur Verfügung gestellt wird, erzeugt die $intraLinks$ Struktur. Das Vorgehen ist identisch wie beim Layered-Graphen aus Abschnitt 3.1, mit dem Unterschied, dass $intraLinks$ ein Array von Pointern darstellt, welches Strukturen vom Typ 'link' beinhaltet. Der Speicherbereich wird dynamisch alloziert und nach dem Abarbeiten des Mapping Algorithmus wieder freigegeben. Die Grösse des Speicherbereichs entspricht $countNode * countNode *$ **sizeof**$(structlink)$.

Die Funktion 'makeInter()', welche ebenfalls von der 'graph.h' Schnittstelle zur Verfügung gestellt wird, erzeugt ein Array von Pointer $interLinks$, welches Pointer auf Knoten speichert. Die Knoten werden mit der Schnittstellen-Funktion 'makeNodes()' generiert und in einem Array $nodeSet$ vom Typ 'node' gespeichert. Auf diese Elemente des Arrays $nodeSet$ zeigen die Pointer von $interLinks$. Der Speicherbereich wird ebenfalls dynamisch alloziert und hat die Grösse $countServiceChains * countNode *$ **sizeof**$(structnode*)$.

Der Aufbau von 'interLinks' sieht folgendermassen aus:

```
void makeInter()
01 ∀ Service-Chains i ∈ Service-Graph G_S
02   if(Service-Chain is port mapped on portNr pNr)
03     interLinks[i][pNr]=&nodeSet[pNr]
04   else
05     ∀ Nodes n ∈ Basis Graph G_B
06       if(nodeSet[n].type = procType)
07         interLinks[i][n]=&nodeSet[n]
```

Für jede Service-Chain des Service-Graphen wird überprüft, ob sie auf einen bestimmten Prozessor mit der $Id = portNr$ gemappt werden soll, oder ob sie auf allen Prozessoren vom Typ $procType$ installiert werden darf. Trifft der erste Fall zu, wird an der Stelle [aktuelle Service-Chain][Portnummer] von $interLinks$ ein Pointer auf das Array $nodeSet$ an der Stelle $portNr$ eingefügt. Im zweiten Fall wird für jeden Prozessor n, welcher dem Prozessortyp $procType$ entspricht, an der Stelle [aktuelle Service-Chain][Prozessortyp] von $interLinks$ ein Pointer auf das Array $nodeSet$ an der Stelle n eingefügt.

Evaluation

Anhand des Basis-Graphen aus Abbildung 3.5 und des Service-Graphen aus Abbildung 3.9 mit den Parametern aus Beispiel 3.1.2 soll die korrekte Funktionsweise zur Erzeugung des optimierten Layered-Graphen gezeigt werden.

Wenn man den Output des Programms 'optLayGraph' in Abbildung 4.2 analysiert, sieht man, dass der Layered Graph korrekt erzeugt wurde. Die Intra-Layer-Links sind durch $intraLinks$ korrekt erzeugt worden. Um die Pointer der Datenstruktur $interLinks$ zu überprüfen, wurden die $procUnits$ der Knoten, auf welche die Pointer zeigen, ausgegeben. Man sieht, dass diese an den richtigen Stellen von $interLinks$ eingefügt wurden. Die letzte Zeile von $interLinks$ enthält keine Pointer, bzw. alle Pointer dieser Zeile zeigen auf 'NULL'. Dies ist korrekt, da ein Pointer in der Zeile i der Datenstruktur $interLinks$ bedeutet, dass es einen Inter-Layer-Link von Layer i zu Layer $i + 1$ gibt. Da der Service-Graph aus drei Service-Chains besteht, existieren drei Layer. Folglich kann kein Inter-Layer-Link auf dem Layer 3 existieren.

Um den optimierten Layered-Graphen mit der alten Version des Layered-Graphen zu

Dies hat zur Folge, dass der Optimierte Exhaustive-Search-Algorithmus im Vergleich zum Exhaustive-Search-Algorithmus aus Abschnitt 3.2 wesentliche Änderungen erfährt.
Ein Rekursionsschritt umfasst neu die folgenden Aufgaben:

Finde erreichbaren Nachbarn: Zuerst wird in der $intraLink$-Matrix der erste Intra-Layer-Nachbar gesucht. Danach wird überprüft, ob der Link zum Nachbarn eine genügend grosse Bandbreite aufweist. Die minimale Bandbreite, die der Link aufweisen muss, entspricht der maximalen Bandbreite der aktuellen Service-Chain. Wird kein Intra-Layer-Nachbar gefunden, wird in der $interLink$-Matrix nach einem Inter-Layer-Nachbarn gesucht. Gibt es keine erreichbaren Nachbarn, wird der Rekursionsschritt beendet und bei der vorherigen Rekursionsstufe weitergesucht. Damit man den gewählten Pfad am Ende des Algorithmus nachvollziehen kann, wird der selektierte Nachbar in einem Array, welche die Sequenz der gewählten Knoten enthält, gespeichert.

Ist der Nachbar der Zielknoten: Handelt es sich beim gewählten Nachbarn um den Zielknoten, wird der gültige Pfad als Knotensequenz im aktuellen Pfad gespeichert. Weiter werden für die Sequenz von Knoten die Delays der Links aufsummiert. Zusätzlich überprüft der Algorithmus ob Knoten in dieser Sequenz sind, welche Pakete nur weiterleiten (forward) müssen. Diese 'Forwarding-Zeiten' werden zu der Summe der Delays addiert und das ergibt das Entscheidungskriterium für den optimalsten Pfad. Nämlich der Pfad mit der kleinsten Zeit (Delay+Forwarding). Danach wird verglichen, ob der aktuelle Pfad einen Delay besitzt, der kleiner ist als der Delay des minimalen Pfades. Ist dies der Fall, wird der aktuelle Pfad als minimaler Pfad gespeichert. Am Ende wird der Rekursionsschritt beendet und beim vorherigen Rekursionsschritt weitergemacht.

Aktualisiere Bandbreite: Bei jedem Rekursionsschritt wird die Bandbreite des verwendeten Links aktualisiert. Im Falle eines Intra-Layer-Links wird die korrespondierende Bandbreite des Links in der $intraLink$ Matrix um die maximale Bandbreite der aktuellen Service-Chain vermindert. Handelt es sich um einen Inter-Layer-Link wird die korrespondierende Bandbreite des Links in der $interLink$ Matrix um die maximalen 'Processing' Kosten der aktuellen Service-Chain vermindert. Falls man in der Rekursion einen Schritt zurück geht, wird die Bandbreite wieder hergestellt, wie sie vor dem Rekursionsschritt war.

Am Ende der Rekursion ist der optimale Pfad im minimalen Pfad gespeichert.

4.3.2 Implementierung

Algorithmus

```
void opt_exhaustiveSearch()
01 init actNode=startNode
02 init actLayer=0
03 if((actNode!=destNode)or(actLayer!=service1.countServiceChains-1))
04  ∀ nodes i ∈ BasisGraph
05   if(node i ∈ Neighbour(actNode))
06    if(intraLinks(actNode,i)≥service1.servChain[actLayer].maxBandwith)
07     intraLinks(actNode,i).bandwith=intraLinks(actNode,i).bandwith-
07     service1.servChain[actLayer].maxBandwith
08     intraLinks(i,actNode).bandwith=intraLinks(i,actNode).bandwith-
08     service1.servChain[actLayer].maxBandwith
09     actNode=i
10     if(currCountLevel[interLinkCount]>countLevel-1)
11      usedCountLevel[currCountLevel[actLayer]][currActNode[actLayer]]=-1
12     add actNode to usedNodeIdSet
13     countLevel++
14     opt_exhaustiveSearch()
15     countLevel–
16     get actNode from usedNodeIdSet
17     intraLinks(actNode,i).bandwith=intraLinks(actNode,i).bandwith+
```

```
17        service1.servChain[actLayer].maxBandwith
18        intraLinks(i,actNode).bandwith=intraLinks(i,actNode).bandwith+
18        service1.servChain[actLayer].maxBandwith
19    if(interLinks(actLayer,actNode)!=NULL)
20      if(usedCountLevel[countLevel][actNode]!=1)
21        if(interLinks(actLayer,actNode).procUnits≥
21        service1.serviceChain[servChainCount].maxProcCost)
22          interLinks(actLayer,actNode).procUnits=interLinks(actLayer,actNode).procUnits-
22          service1.serviceChain[servChainCount].maxProcCost
23          usedCountLevel[countLevel][actNode]=1
24          currActNode[actLayer]=actNode
25          currCountLevel[actLayer]=countLevel
26          actlayer++
27          add actNode to usedNodeIdSet
28          countLevel++
29          opt_exhaustiveSearch()
30          countLevel–
31          actlayer–
32          get actNode from usedNodeIdSet
33          interLinks(actLayer,actNode).procUnits=interLinks(actLayer,actNode).procUnits+
33          service1.serviceChain[servChainCount].maxProcCost
34 else
35   actPath.length=countLevel+1
36   copy nodes i from usedNodeIdSet to actPath
37   calculate delay and forwarding cost of actPath
38   if(actPath.delay<minPath.delay)
39     minPath=actPath
```

Beschreibung

Zuerst werden alle Variablen des Algorithmus initialisiert (Zeile 01-02). Die Variable $actNode$ speichert den aktuellen Knoten und wird mit dem Startknoten initialisiert, welcher mit $service1.inPort$ übergeben wird. Die Variable $actLayer$ speichert den aktuellen Layer. Dies ist notwendig, da durch die Aufteilung der $adjMatrix$ auf die zwei Matrizen $interLinks$ und $intraLinks$ die Information über den Zusammenhang zwischen der Knoten Id und des Layers, auf dem sich dieser Knoten befindet, ebenfalls aufgeteilt worden ist. Das bedeutet, dass man anhand des aktuellen Knotens $actNode$ nicht auf den aktuellen Layer schliessen kann, wie das beim Exhaustive-Search-Algorithmus aus Abschnitt 3.2 noch möglich war. Die Folge davon ist, dass man die Variable $actLayer$ benötigt. Weiter werden die Arrays $interLinkFin$ und $currCountLevel$ initialisiert. Ihre Verwendung wird später in diesem Abschnitt erläutert. Das gleiche gilt für die Variablen $interLinkCount$, $currActNode$ und die Matrix $usedCountLevel$. Im Array $usedNodeIdSet$ werden die Knoten gespeichert, welche für die aktuelle Rekursion bisher benutzt wurden.
Nach der Initialisierung beginnt die eigentliche Rekursion. Für alle Knoten im Basis-Graphen wird zuerst nach einem Intra-Layer-Nachbarn gesucht (Zeile 05). Hat man einen Intra-Layer-Nachbarn gefunden, wird überprüft, ob die Bandbreite des Links zu diesem Nachbarn grösser als die maximal benötigte Bandbreite der aktuellen Service-Chain ist (Zeile 06). Die aktuelle Service-Chain, lässt sich leicht anhand des Wertes von $actLayer$ ermitteln. Ist die Bandbreite ausreichend gross, wird der Link ausgewählt und die Bandbreite aktualisiert (Zeile 07-08). Da es sich um einen Intra-Layer-Link handelt, muss die Bandbreite des Links in beide Richtungen aktualisiert werden. Nach der Aktualisierung des Links wird der neue aktuelle Knoten $actNode$ selektiert (Zeile 09). Damit man am Ende der Rekursion nachvollziehen kann, welche Knoten verwendet wurden, speichert man den aktuellen Knoten $actNode$ im Array $usedNodeIdSet$ (Zeile 12). In der Variable $countLevel$ wird die aktuelle Rekursionsstufe gespeichert. $countLevel$ dient in mehreren Fällen als Hilfsvariable. Sie wird zum Beispiel dazu verwendet, um den aktuellen Knoten $actNode$ im Array $usedNodeIdSet$ zu speichern. Vor jedem Rekursionsaufruf (Zeile

```
Start Node: 3    Dest Node: 5

 3  3
pfadzahl: 0
**********************************
minimaler Pfad:
Pfad delay:  10000
```

Abbildung 4.4: Der Output von 'opt_mapping kompiliert mit 'graph_one.cc'

14 und Zeile 31) wird $countLevel$ um 1 erhöht und nach dem Rekursionsaufruf um 1 vermindert. Zusätzlich wird nach dem Rekursionsaufruf der aktuelle Knoten $actNode$ geladen, sowie die Bandbreite des Links rekonstruiert, so dass die Situation, wie sie vor dem Rekursionsschritt war, wieder korrekt geladen wird (Zeile 16-18).
Findet man für einen Rekursionsschritt keinen Intra-Layer-Nachbarn, wird als Nächstes ein Inter-Layer-Nachbar gesucht (Zeile 19). Wird ein solcher gefunden, muss überprüft werden, ob die Processing-Units $procUnits$ des korrespondierenden Knotens mindestens so gross sind, wie die maximalen Processing Kosten $maxProcCost$ der aktuellen Service-Chain (Zeile 21). Ist dies der Fall, werden die Processing-Units des Knotens aktualisiert (Zeile 22) und der aktuelle Knoten $actNode$ ins Array $usedNodeIdSet$ eingefügt (Zeile 27). Weiter wird der aktuelle Layer $actLayer$ um 1 erhöht (Zeile 26), da es sich beim benutzten Link um einen Inter-Layer-Link handelt und man sich darum ein Layer höher befindet. Geht man diesen Rekursionsschritt wieder zurück, wird der aktuelle Layer um 1 vermindert (Zeile 31) und die Variablen wieder so hergestellt, wie sie vor dem Rekursionsschritt waren.
Da der Algorithmus keine Kontrolle darüber hat, ob ein Rekursionsschritt von einem Intra-Layer-Nachbarn oder von einem Inter-Layer-Nachbarn ausgelöst wurde, muss man diese Kontrollfunktion explizit in den Algorithmus einbinden. Die Variable $usedCountLevel$ speichert, ob ein Inter-Layer-Link benutzt werden darf. Ein Inter-Layer-Link, der schon mal benutzt worden ist, darf erst wieder verwendet werden (Zeile 20), wenn sich in einer Rekursionstiefe, die kleiner ist als die Rekursionstiefe in welcher der Inter-Layer-Link benutzt worden ist, ein Knoten ändert (Zeile 10-11). In den Variablen $currActNode$ und $currCountLevel$ werden die Informationen gespeichert, welches der aktuelle Knoten $actNode$ und wie tief die Rekursionstiefe $countLevel$ zum Zeitpunkt des Inter-Layer-Links waren (Zeile 24-25).
Die Rekursion ist abgeschlossen, wenn kein Intra-Layer-Nachbar und kein Inter-Layer-Nachbar gefunden werden konnten oder wenn der Zielknoten und der Ziellayer erreicht worden sind (Zeile 03).
Ist der Zielknoten und der Ziellayer erreicht worden (Zeile 34), wird zuerst die Rekursionstiefe gespeichert. Sie entspricht der Länge des aktuellen Pfades um 1 erhöht (Zeile 35). Danach werden die einzelnen Knoten des $usedNodeIdSet$ in den aktuellen Pfad $actPath$ kopiert und der Delay des Pfades inklusive Forwarding-Kosten berechnet (Zeile 36-37). Falls der aktuelle Pfad $actPath$ schneller ist als der momentan schnellste Pfad $minPath$, wird dieser in $minPath$ gespeichert (Zeile 38-39).

4.3.3 Evaluation

In diesem Abschnitt soll der Algorithmus anhand eines Beispiels verifiziert werden und dann bezüglich Speicherbedarf und Zeitaufwand untersucht werden.

Verifikation

Um die Funktionsweise des Algorithmus zu verifizieren, wird der Agorithmus analog zur Evaluation aus Abschnitt 3.2.3 an verschiedenen Basis-Graphen getestet. Es werden die selben drei Tests ausgeführt. Beim ersten Test sollte erneut kein gültiger Pfad gefunden werden. Beim zweiten Test müsste genau ein Pfad, beim letzten Test alle Pfade berechnet werden können.
Der Basis Graph des ersten Tests ist in der Datei 'graph_one.cc' gespeichert. Kompiliert man den Algorithmus 'opt_mapping' mit dieser Datei und führt ihn danach aus, erhält man den Output aus Abbildung 4.4. Wie man sieht, wird korrekterweise kein gültiger Pfad gefunden.

Zusätzlich zu den oben erwähnten Optimierungen, kann die Anzahl an Layer des Layered-Graphen um ein Layer reduziert werden. Der Grund dafür ist, dass im ersten Layer keine Intra-Layer-Links verwendet werden, da die erste Service-Chain immer auf einen bestimmten Prozessor gemappt wird. Diese Reduktion führt dazu, dass weniger Pfade berechnet werden müssen. Daraus resultiert erneut ein kleinerer Zeitaufwand und ein geringerer Speicherbedarf.

4.4 Optimierter Extended-Shortest-Path

4.4.1 Grundlagen

Der optimierte Extended-Shortest-Path-Algorithmus muss den neuen Datenstrukturen des optimierten Layered-Graphen angepasst werden. Die Funktionsweise bleibt dieselbe, wie für den Extended-Shortest-Path-Algorithmus aus Abschnitt 3.3. Die Ideen von Dijkstra's Shortest-Path-Algorithmus und den dazugehörigen Erweiterungen aus Abschnitt 3.3.2 werden ebenfalls implementiert. Einzig auf die Implementierung der Reverse Exhaustive-Search Funktionalität wird verzichtet.
Die Idee zur Einbindung der neuen Datenstrukturen $intraLinks$ und $interLinks$ in den optimierten Extended-Shortest-Path-Algorithmus sieht folgendermassen aus:

void **opt_extendedShortestPath()**
01 actNode=startNode
02 actLayer=0
03 while((actNode≠destNode)**and**(actLayer≠lastLayer))
04 add actNode to usedNodeSet
05 if(all Nodes i ∈ Basis Graph for actLayer determined)
06 ∀ nodes i ∈ Basis Graph for actLayer
07 find Inter-Layer-Link(i,j)
08 determine bestCurrDist[j]
09 determine predNode[j]
10 select actNode
11 set all Nodes *not determined*
12 actLayer=actLayer+1
13 else
14 ∀ nodes i ∈ Basis Graph for actLayer
15 find Intra-Layer-Links(actNode,i)
16 determine bestCurrDist[i]
17 determine predNode[i]
18 select actNode
19 set actNode *determined*

Der Algorithmus beginnt beim Startknoten $startNode$ auf dem Layer $actLayer = 0$ (Zeile 01-02). Solange der Zielknoten $dedstNode$ auf dem letzten Layer $lastLayer$ nicht erreicht worden ist (Zeile 03), wird folgender Ablauf ausgeführt:
Zuerst wird der aktuelle Knoten als 'abgearbeitet' markiert (Zeile 04). Danach werden alle Knoten des Layers 0 abgearbeitet, wie man es vom Extended-Shortest-Path von Abschnitt 3.3 kennt und so ein 'Spanning-Tree' vom Basis-Graphen des ersten Layers (Layer 0) erzeugt (Zeile 13-19). Sind alle Knoten abgearbeitet und bestimmt (engl. determined), können für das erste Layer alle verfügbaren Inter-Layer-Links benutzt werden und so die Endknoten der Inter-Layer-Links bestimmt werden (Zeile 05-11). Sind alle Inter-Layer-Links benutzt worden, wird das nächste Layer analog abgearbeitet.
Das folgende Beispiel soll den Ablauf nochmals verdeutlichen.

Beispiel Gegeben ist der Layered Graph aus Abbildung 3.12. In der Abbildung 4.9 sieht man wie der Algorithmus Layer für Layer den 'Spanning-Tree' generiert.

4.4.2 Implementierung

Algorithmus

```
void opt_extendedShortestPath()
01 init predNode[]
02 init bestCurrDist[]
03 actNode=startNode
04 while((usedNodeCount≠#Nodes∈LayeredGraph)or(actLayer≠lastLayer))
05   add actNode to usedNodeSet
06   if(usedNodeCount==countNode-1)
07     actLayer=actLayer+1
08     ∀ nodes i ∈ Basis Graph
09       if(interLinks(actLayer,i)≠NULL)
10         if(interLinks(actLayer,i)->procUnits≥service1.servChain[actLayer].maxProcCost
11           testNode=actNode
12           sumBandwith=service1.servChain[actLayer].maxProcCost
13           while(testNode≠startNode)
14             if((testNode%countNode=i)and(predNode[testNode]%countNode=actNode))
15               sumBandwith=sumBandwith+
15               service1.servChain[predNode[testNode]/countNode].maxProcCost
16             if(Inter-Layer-Link)
17               testNode=testNode-countNode
18             else
19               testNode=predNode[testNode]
20           if(sumBandwith≤interLinks(actLayer,i)->procUnits)
21             bestCurrDist[actLayer*countNode+i]=bestCurrDist[(actLayer-1)*
21             countNode+i]+interLinks(actLayer,i)->procTime
22             predNode[actLayer*countNode+i]=actLayer*countNode+i
23             if(bestCurrDist[actLayer*countNode+i]<minVar
24               minVar=bestCurrDist[actLayer*countNode+i]
25               actNode=i
26           else
27             bestCurrDist[actLayer*countNode+i]=2000
28             predNode[actLayer*countNode+i]=actLayer*countNode+i
29   usedNodeCount=1
30   else
31     ∀ nodes i ∈ Basis Graph
32       if(intraLinks(actLayer,i).delay≠-1)
33         if(node i not already used in actLayer)
34           if(intraLinks(actNode,i).bandwith≥service1.servChain[actLayer].maxBandwith
35             if((Level(actNode)=Level(predNode[actNode]))and
35             (predNode[actNode]≠-1))
36               if((bestCurrDist[actLayer*countNode+i]=-1)or
36               (bestCurrDist[actLayer*countNode+i]>bestCurrDist[actLayer*countNode
36               +actNode]+intraLinks[actNode][i].delay+nodeSet[actNode].forwardTime))
37                 Capacity-Tracking
38             else
39               if((bestCurrDist[actLayer*countNode+i]=-1)or
39               (bestCurrDist[actLayer*countNode+i]>bestCurrDist[actLayer*countNode+
39               actNode]+intraLinks[actNode][i].delay))
40                 Capacity-Tracking
41     minVar=∞
42     ∀ Nodes j ∈ Basis Graph and not already used in actLayer
43       if((bestCurrDist[actLayer*countNode+j]<-1)and
```

```
44      (bestCurrDist[actLayer*countNode+j]≠-1))
45        minVar=bestCurrDist[actLayer*countNode+j]
46        actNode=j]
47    usedNodeCount=usedNodeCount+1
48 bestCurrDist[destNode]=bestCurrDist[destNode]+nodeSet[destNode].procTime
```

Beschreibung

Zuerst werden alle Variablen initialisiert. Speziell zu erwähnen sind die Arrays $predNode$ und $bestCurrDist$, welche die Resultate des Algorithmus speichern (Zeile 01-02). Damit man einen Layer sparen kann, wird der aktuelle Knoten auf den Eingangsknoten des Services $service1.inPort$ gesetzt (Zeile 03) und der Wert der Processing-Time des Prozessors, auf den die erste Service-Chain gemappt wird, in $bestCurrDist[startNode]$ gespeichert.
Der Algorithmus bestimmt zuerst für jeden Knoten des aktuellen Layers $actLayer$ den Vorgängerknoten $predNode$ und die momentan kürzeste Distanz zum Startknoten $bestCurrDist$. Im Detail sieht dies so aus, dass für jeden Knoten i im Basis-Graphen überprüft wird, ob er einen Link zum aktuellen Knoten $actNode$ hat (Zeile 31-32). Existiert ein Link zum aktuellen Knoten, wird kontrolliert, ob der Knoten i schon abgearbeitet worden ist (Zeile 33). Ist dies nicht der Fall, folgt ein weiterer Test, mit dem die, von der Service-Chain benötigte Bandbreite, mit der vorhandenen Bandbrei-te verglichen wird. Ist die verfügbare Bandbreite genug gross (Zeile 34) wird überprüft, ob der aktuelle Knoten die Aufgabe hat Pakete nur weiterzuleiten, also zu forwarden, oder ob er Pakete verarbeitet (Zeile 35). Ein Knoten, der Pakete nur weiterleitet, befindet sich auf dem selben Layer, wie sein Vorgängerknoten. Darum vergleicht man die Layer, um zu bestimmen, ob ein Knoten forwarded.
Handelt es sich beim aktuellen Knoten um einen Knoten, der Pakete nur weiterleitet, werden die Forwarding-Kosten beim Vergleich von der aktuellen $bestCurrDist$ mit der neuen $bestCurrDist$, die über einen Umweg via dem aktuellen Knoten entsteht, berücksichtigt (Zeile 36). Ist die neue $bestCurrDist$ via dem aktuellen Knoten kürzer als die bestehende, wird zuerst das 'Capacity-Tracking' ausgeführt (Zeile 37), bevor die neue $bestCurrDist$ und der neue $predNode$ gespeichert werden. Das Link Capacity-Tracking entspricht dem Link Capacity-Tracking, welches auch für Inter-Layer-Links ausgeführt wird (Zeile 11-28). Das Link Capacity-Tracking für Inter-Layer-Links wird weiter unten erläutert. Der Unterschied des Link Capacity-Tracking für Intra-Layer-Links zu dem für Inter-Layer-Links besteht darin, dass die Intra-Links bidirektional sind und folglich bei einem Capacity-Tracking für den Link(a,b) auch der Link(b,a) berücksichtigt werden muss.
Handelt es sich beim aktuellen Knoten nicht um einen Knoten, der Pakete nur weiterleitet, können die Forwarding-Kosten weggelassen werden (Zeile 38). Ist die neu berechnete $bestCurrDist$ kürzer als die bestehende (Zeile 39), wird ebenfalls zuerst das Capacity-Tracking ausgeführt, um zu verhindern, dass keine Bandbreite überlastet wird. Das Capacity-Tracking (Zeile 40) ist identisch zum oben erwähnten Capacity-Tracking für Knoten, die nur forwarden.
Ist ein Knoten abgearbeitet worden, wird der nächste aufgrund der kürzesten Distanz zum Startknoten selektiert. Es können aber nur Knoten ausgewählt werden, die auf diesem Layer noch nicht abgearbeitet worden sind (Zeile 42-47).
Sind alle Knoten eines Layers abgearbeitet und die $predNode$ sowie die $bestCurrDist$ bestimmt (Zeile 06), wird der aktuelle Layer $actLayer$ um 1 erhöht (Zeile 07) und für jeden Knoten des Basis-Graphen auf diesem Layer (Zeile 08) nach möglichen Inter-Layer-Nachbarn gesucht (Zeile09). Hat man einen solchen Knoten gefunden, wird zuerst überprüft, ob seine Processing-Units ausreichend sind für die aktuelle Service-Chain (Zeile 10). Ist dies der Fall, wird das Capacity-Tracking gestartet.
Als erstes wird der Hilfsvariable $testNode$ der aktuelle Knoten $actNode$ zugewiesen (Zeile 11) und die zweite Hilfsvariable $sumBandwith$ der benötigten Processing-Units der aktuellen Chain gleichgesetzt (Zeile 12). Danach wird vom aktuellen Knoten aus schrittweise der Vorgängerknoten selektiert (Zeile 17 oder 19) und überprüft ob der aktuelle Link(actNode,i), den man verwenden möchte, schon vorher einmal benutzt worden ist (Zeile 14). Ist der Link schon mal benutzt worden, wird die $sumBandwith$ um den Wert der Processing-Units, welcher von der damaligen Service-Chain benötigt worden ist, erhöht (Zeile 15). Danach wird der Vorgängerknoten des Vorgängerknotens selektiert und wieder überprüft, ob der Link(actNode,i) schon mal benutzt worden ist. Dies wird so lange gemacht, bis der Startknoten $startNode$ erreicht

```
Modus ist Shortest Path
Startknoten: 3
Exit - Kein Pfad moeglich
```

Abbildung 4.10: Der Output von 'opt_mapping' kompiliert mit 'graph_one.cc'

wird(Zeile 13). Hat man den Startknoten erreicht, wird die summierte Bandbreite $sumBandwith$ mit den verfügbaren Processing-Units des korrespondierenden Prozessors verglichen (Zeile 20). Sind die verfügbaren Processing-Units ausreichend, werden die neue $bestCurrDist$ und der neue $predNode$ gespeichert (Zeile 21-22). Danach wird der neue aktuelle Knoten aufgrund des kleinsten Wertes für $bestCurrDist$ unter allen Knoten des Basis-Graphen, für den aktuellen Layer ausgewählt (Zeile 23-25). Reichen die verfügbaren Processing-Units nicht aus (Zeile 26), wird als $currBestDist$ der Wert 2000 gespeichert (Zeile 27). Dies verhindert ein Blockieren des Algorithmus, erfordert aber zusätzlich, dass am Schluss das Resultat überprüft wird, ob im Mapping ein Knoten mit $bestCurrDist = 2000$ verwendet worden ist.
Sind alle Knoten abgearbeitet worden, wird die $bestCurrDist$ des Zielknotens noch um den Wert der Processing-Time des Prozessors, auf den die letzte Service-Chain gemappt wird, erhöht (Zeile 48). Dies muss der Algorithmus am Schluss noch nachholen, da man den letzten Layer, wie schon den ersten, eingespart hat, um die Performance des Algorithmus zu verbessern.

4.4.3 Evaluation

Verifikation

Um die Funktionsweise des Algorithmus zu verifizieren, wird der Agorithmus analog zur Evaluation aus Abschnitt 3.2.3 an verschiedenen Basis-Graphen getestet. Es werden erneut die selben drei Tests ausgeführt. Beim ersten Test sollte erneut kein gültiger Pfad gefunden werden. Beim zweiten Test müsste wiederum ein Pfad und beim letzten Test alle Pfade berechnet werden.
Der Output, den der Algorithmus für den ersten Test liefert, ist in Abbildung 4.10 aufgeführt. Wie man sieht, wird korrekterweise kein gültiger Pfad gefunden.

Für den zweiten Test wird der Algorithmus mit der Datei 'graph_two.cc' kompiliert. Daraus resultiert der Output von Abbildung 4.5. Anhand des Outputs kann man erkennen, dass genau ein gültiger Pfad gefunden wurde. Vergleicht man die Knotensequenz des gültigen Pfades mit der Knotensequenz des gültigen Pfades aus Abbildung 3.34, sieht man, dass die beiden Sequenzen das gleiche Mapping symbolisieren. Der Optimierte Extended-Shortest-Path findet den gleichen gültigen Pfad wie der nicht optimierte Algorithmus. Weiter sieht man anhand der Werte in $predNode$, dass die beiden Algorithmen denselben 'Spanning-Tree' erzeugen, mit dem Unterschied, dass der optimierte Extended-Shortest-Path-Algorithmus einen Layer weniger benötigt.

Anhand des Outputs von Abbildung 4.11 werden die Rekursionsschritte dem Programm Code aus Abschnitt 4.4.2 zugeordnet. Dies soll helfen zu zeigen, dass der Algorithmus korrekt funktioniert. In Abbildung 4.12 sieht man diese Zuordnung Zeile für Zeile.

Im letzten Test wird der Basis Graph aus der Datei 'graph_three.cc' verwendet. Den resultierenden Output für den Algorithmus sieht man in der Abbildung 4.13. Mit Hilfe des Outputs kann man erkennen, dass zwei gültige Pfade gefunden werden. Dies entpricht der gesamten Anzahl an gültiger Pfade. Zieht man den Vergleich zum nicht optimierten Algorithmus, sieht man, dass die beiden gültigen Pfade die selben Pfade repräsentieren wie die gültigen Pfade aus der Abbildung 3.36. Der Wert des berechneten Delays stimmt ebenfalls bei beiden Varianten des Algorithmus überein.

19 if(SPW(G_u^f,a,i)+SPW(G_S,i,j)+SPW(G_v^f,j,b)<SPW(G,a,b))
20 SPW(G,a,b)=SPW(G_u^f,a,i)+SPW(G_S,i,j)+SPW(G_v^f,j,b)
21 if u=v and SPW(G_u^f,a,b)<SPW(G,a,b)
22 SPW(G,a,b)=SPW(G_u^f,a,b)

4.5.4 HEPV Creation and Maintenance

HEPV Creation

Gegeben seien die Partitionen eines Graphen. Zuerst werden alle Fragmente mittels Shortest-Path-Algorithmus codiert. Dies geschieht auf Zeile 2 des Creation-Algorithmus. Für jedes Grenzknoten-Paar eines jeden Fragmentes (Zeile 3) wird ein Link im Super-Graphen erzeugt (Zeile 11 bis 14), falls es einen Pfad zwischen diesen Knoten gibt (Zeile 4) und dieser Link noch nicht im Super-Graphen existiert. Gibt es schon einen solchen Link wird er aktualisiert (Zeile 7 bis 9). Am Schluss wird der Super-Graph mittels Shortest-Path-Algorithmus codiert.

void HEPVCreate(G_0^f,G_1^f,...,G_{p-1}^f)
01 ∀(0 ≤ u < p)
02 ENCODE(G_u^f)
03 ∀(Node i, Node j ∈ BORDER(G_u^f))
04 if SPW(G_u^f,i,j)< ∞
05 if Link(i,j) exists already
06 if Link(i,j).weight>SPW(G_u^f),i,j)
07 Link(i,j).fragmentId=G_u^f
08 Link(i,j).nextHop=ShortestPath(G_u^f,i,j).nextHop
09 Link(i,j).weight=SPW(G_u^f,i,j)
10 else
11 add Link to Super-Graph Link Set
12 Link(i,j).fragmentId=G_u^f
13 Link(i,j).nextHop=ShortestPath(G_u^f,i,j).nextHop
14 Link(i,j).weight=SPW(G_u^f,i,j)
15 ENCODE(G^S)

HEPV Update

Immer wenn ein Linkgewicht eines Fragments ändert, wird der korrespondierende Link im Super-Graphen ebenfalls beeinflusst. Es ist offensichtlich, dass wir nur die EPV der direkt beeinflussten Fragmente und den Super-Graphen ändern müssen, während die anderen Fragmente unbeeinflusst bleiben.
Zuerst werden die geänderten Fragmente neu codiert (Zeile 2). Danach muss der HEPVUpdate Algorithmus für jedes Grenzknoten-Paar der geänderten Fragmente (Zeile 3) das neue Link-Gewicht im Super-Graphen bestimmen: In der Schleife (Zeile 5 bis Zeile 10) wird für jedes Grenzknoten-Paar der geänderten Fragmente der neue Shortest-Path gefunden, indem jedes Fragment, welches dieses Grenzknoten-Paar besitzt, berücksichtigt wird.
Als Parameter erhält die HEPVUpdate-Funktion die geänderten Fragmente. Am Schluss wird der ganze Super-Graph wieder codiert.

void HEPVUpdate($G_{v_0}^f$,$G_{v_1}^f$,...,$G_{v_{q-1}}^f$)
01 ∀ G_u^f ∈ { $G_{v_0}^f$,$G_{v_1}^f$,...,$G_{v_{q-1}}^f$ }
02 ENCODE(G_u^f)
03 ∀ (Node i, Node j ∈ BORDER(G_u^f)
04 LinkWeight(G^S,i,j) = ∞
05 ∀ Fragments G_u^f der Partition
06 ∀ (Node i, Node j ∈ BORDER(G_u^f) and ∈ BORDER($G_{v_0}^f$) or ... or BORDER($G_{v_{q-1}}^f$))
07 if SPW(G_u^f,i,j) < ∞ and LinkWeight(G^S,i,j) > SPW(G_u^f,i,j)

08 Link(G^S,i,j).fragmentId=G_u^f
09 Link(G^S,i,j).nextHop=ShortestPath(G_u^f,i,j).nextHop
10 Link(G^S,i,j).weight=SPW(G_u^f,i,j)
11 ENCODE(G^S)

Das folgende Beispiel soll die Funktionalität verdeutlichen:

Beispiel In Abbildung 4.19 wird der Delay des Links 0-1 im ersten Fragment G_0^f von 1 auf 6 geändert. Der Ablauf der Funktion sieht nun folgendermassen aus:

1. Zuerst wird das geänderte Fragment G_0^f neu codiert. Die geänderten Werte der neuen Codierung sieht man in Abbildung 4.21; Sie sind rot markiert.
2. Für jedes Grenzknoten-Paar (i,j), welches im geänderten Fragment G_0^f vorkommt, wird das Gewicht des Links zwischen diesen Grenzknoten auf $\inf$ gesetzt. Im Beispiel sind das folgende Grenzknoten-Paare: (1,3),(1,4) und (3,4). Die Links gelten natürlich auch in entgegengesetzter Richtung. Das Ergebnis dieses Funktionsschrittes kann man der Abbildung 4.22(a) entnehmen.
3. Im nächsten Schritt wird für jedes Grenzknoten-Paar, welches im geänderten Fragment und eventuell in einem anderen vorkommt, der kürzeste Pfad zwischen den beiden Grenzknoten berechnet.

   Im Beispiel ist also für:

   (a) u=0 das Grenzknoten-Paar (1,3) im geänderten Fragment G_0^f und ebenso im Fragment G_u^f enthalten. In diesem Fall ist es leicht nachvollziehbar, da es sich zweimal um das gleiche Fragment G_0^f handelt. SPW(G_0^f,1,3) $< \infty$ and LinkWeight(G^S,1,3) $>$ SPW(G_0^f,1,3) ist erfüllt, darum wird der Super-Graph folgendermassen geändert:
   - Link(G^S,1,3).fragmentId=G_0^f
   - Link(G^S,1,3).nextHop=ShortestPath(G_0^f,1,3).nextHop=Knoten 0
   - Link(G^S,1,3).weight=SPW(G_0^f,1,3)=9

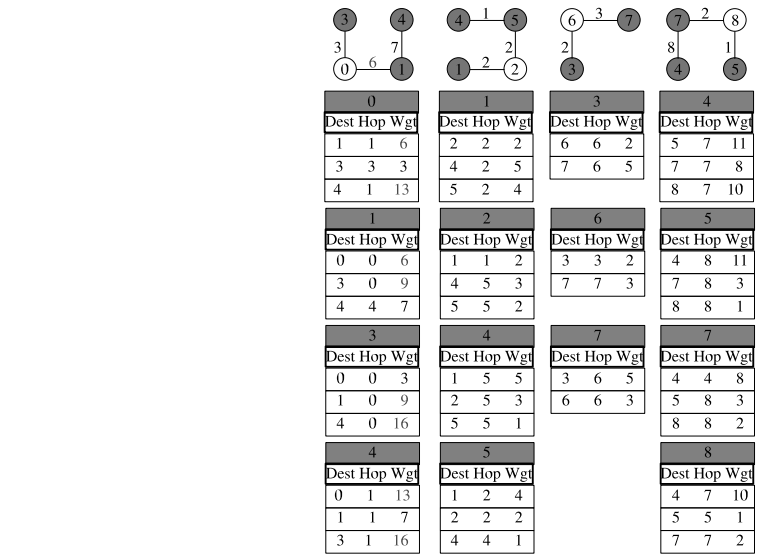
   Diese Änderungen sind in der Abbildung 4.22(b) eingetragen.

   (b) u=0 das Grenzknoten-Paar (1,4) im geänderten Fragment G_0^f und ebenso im Fragment G_u^f enthalten. Auch in diesem Fall ist es logisch, da es sich zweimal um das gleiche Fragment G_0^f handelt. SPW(G_0^f,1,4) $< \infty$ and LinkWeight(G^S,1,4) $>$ SPW(G_0^f,1,4) ist erfüllt, darum wird der Super-Graph folgendermassen geändert:
   - Link(G^S,1,4).fragmentId=G_0^f
   - Link(G^S,1,4).nextHop=ShortestPath(G_0^f,1,4).nextHop=Knoten 4
   - Link(G^S,1,4).weight=SPW(G_0^f,1,4)=7

   Diese Änderungen sind in der Abbildung 4.22(c) eingetragen.

   (c) u=1 das Grenzknoten-Paar (1,4) im geänderten Fragment G_0^f und ebenso im Fragment G_u^f enthalten. In diesem Fall handelt es sich um das erste und das zweite Fragment. SPW(G_1^f,1,4) $< \infty$ and LinkWeight(G^S,1,4) $>$ SPW(G_0^f,1,4) ist erfüllt, darum wird der Super-Graph folgendermassen geändert:
   - Link(G^S,1,4).fragmentId=G_1^f
   - Link(G^S,1,4).nextHop=ShortestPath(G_1^f,1,4).nextHop=Knoten 2
   - Link(G^S,1,4).weight=SPW(G_1^f,1,4)=5

   Diese Änderungen sind in der Abbildung 4.22(d) eingetragen.
4. Im letzten Schritt wird der vollständige Super-Graph codiert. Diese Codierung sieht man in Abbildung 4.23.



0			
Dest	Hop	Wgt	
1	1	6	
3	3	3	
4	1	13	

1			
Dest	Hop	Wgt	
2	2	2	
4	2	5	
5	2	4	

3			
Dest	Hop	Wgt	
6	6	2	
7	6	5	

4			
Dest	Hop	Wgt	
5	7	11	
7	7	8	
8	7	10	

1			
Dest	Hop	Wgt	
0	0	6	
3	0	9	
4	4	7	

2			
Dest	Hop	Wgt	
1	1	2	
4	5	3	
5	5	2	

6			
Dest	Hop	Wgt	
3	3	2	
7	7	3	

5			
Dest	Hop	Wgt	
4	8	11	
7	8	3	
8	8	1	

3			
Dest	Hop	Wgt	
0	0	3	
1	0	9	
4	0	16	

4			
Dest	Hop	Wgt	
1	5	5	
2	5	3	
5	5	1	

7			
Dest	Hop	Wgt	
3	6	5	
6	6	3	

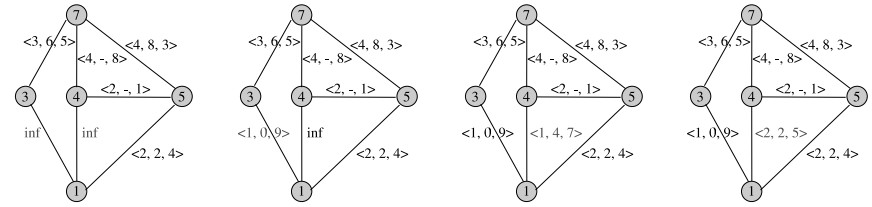
7			
Dest	Hop	Wgt	
4	4	8	
5	8	3	
8	8	2	

4			
Dest	Hop	Wgt	
0	1	13	
1	1	7	
3	1	16	

5			
Dest	Hop	Wgt	
1	2	4	
2	2	2	
4	4	1	

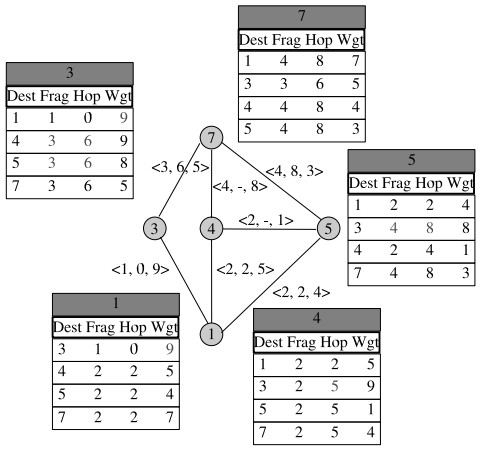
8			
Dest	Hop	Wgt	
4	7	10	
5	5	1	
7	7	2	

Abbildung 4.21: Die neue Codierung des ersten Fragments



(a) 1. Funktionsschritt (b) 2. Funktionsschritt (c) 3. Funktionsschritt (d) 4. Funktionsschritt

Abbildung 4.22: Änderungen am Super-Graphen



3				
Dest	Frag	Hop	Wgt	
1	1	0	9	
4	3	6	9	
5	3	6	8	
7	3	6	5	

7				
Dest	Frag	Hop	Wgt	
1	4	8	7	
3	3	6	5	
4	4	8	4	
5	4	8	3	

5				
Dest	Frag	Hop	Wgt	
1	2	2	4	
3	4	8	8	
4	2	4	1	
7	4	8	3	

1				
Dest	Frag	Hop	Wgt	
3	1	0	9	
4	2	2	5	
5	2	2	4	
7	2	2	7	

4				
Dest	Frag	Hop	Wgt	
1	2	2	5	
3	2	5	9	
5	2	5	1	
7	2	5	4	

Abbildung 4.23: Neue Codierung des Super-Graphen

```
makeService();
makeNodes();
makeIntra();
makeInter();

ShortestPath();

for(int i=0;i<service1.countServiceChains;i++){
   delete interLinks[i];
}
delete interLinks;
```

(a) Der bisherige Ablauf

```
makeService1();
makeNodes();
makeIntra();
makeInter();

ShortestPath();

for(int i=0;i<service1.countServiceChains;i++){
   delete interLinks[i];
}
delete interLinks;

makeService2();
makeNodes();
makeInter();

init=1;
ShortestPath();

for(int i=0;i<service1.countServiceChains;i++){
   delete interLinks[i];
}
delete interLinks;
```

(b) Der neue Ablauf

Abbildung 4.25: Der bisherige und der neue Ablauf der Aufrufe der Mapping Algorithmen

4.6.2 Implementierung

Erweiterung der Schnittstelle 'service.h'

Damit parallele Service-Chains möglich sind, muss die Schnittstelle 'service.h' erweitert werden. Damit die beiden Algorithmen nur minimal verändert werden müssen, wird 'service.h' um eine neue Funktion erweitert, welche die Parallelität implementiert. Diese neue Funktion 'makeService2()' macht nichts anderes, als dass sie die erste parallele Service-Chain durch die zweite ersetzt. Die erste und die letzte Service-Chain bleiben dabei unverändert. Das Resultat davon ist, dass der erste Service-Graph durch den zweiten ersetzt wird.

Erweiterung des Aufrufs der Mapping Algorithmen

Der bisherige Ablauf des Aufrufs der Algorithmen ist in Abbildung 4.25(a) abgebildet. Für die, durch die Aufteilung des Service-Graphen entstandenen, neuen Service-Graphen müssen die Algorithmen jeweils einmal aufgerufen werden. Zwischen dem Aufruf des Algorithmus für das Mapping des ersten Service-Graphen und dem Aufruf für das Mapping des zweiten Service-Graphen müssen die Ressourcen, welche durch den ersten Service-Graphen verbraucht werden im Layered-Graphen berücksichtigt werden. Konkret sind davon nur die Intra-Layer-Links betroffen, da die erste und die letzte Service-Chain zwischen den Service-Graphen nicht wechseln und die zweite bzw. dritte Service-Chain auf verschiedenen Prozessoren installiert werden und darum nicht die selben Inter-Layer-Links benutzen können. In der Abbildung 4.25(b) ist der neue Ablauf des Aufrufs der Algorithmen dargestellt.
Wenn man den Ablauf des Aufrufs von Abbildung 4.25(b) analysiert, erkennt man, dass die Intra-Layer-Links nur einmal mittels 'makeIntra()' erzeugt werden. Der Grund dafür ist der, dass die Bandbreite der Intra-Layer-Links, welche man für das Mapping des ersten Service-Graphen benötigt, dem Mapping des zweiten Service-Graphen nicht mehr zur Verfügung steht. So wird eine Überlastung der Links verhindert. In der Abbildung 4.25 wird nur der Extended-Shortest-Path-Algorithmus erwähnt. Die gemachten Aussagen gelten aber ebenso für den Exhaustive-Search-Algorithmus.

Auf die Evaluation der Fork- und Join-Operation musste aus Zeitgründen verzichtet werden. Die Funktionsweise wurde aber während der Implementierung fortlaufend überprüft. Der

interessierte Leser kann anhand der Files im Verzeichnis ’/ribolla/code/parallelServChains’ die Evaluation nachholen.

Kapitel 5

Optimierung des NP-extended PromethOS Execution Environments

In diesem Kapitel wird die Optimierung des NP-extended PromethOS Execution Environments erläutert.
Im Abschnitt 5.1 wird auf den bestehenden Programmcode eingegangen. Es wird die bisherige Funktionsweise und der Aufbau erläutert.
Danach werden im Abschnitt 5.2 die genaue Problemstellung und die Lösungsansätze vorgestellt.
Weiter wird im Abschnitt 5.3 die Implementierung eines Lösungsansatzes erläutert.
Am Ende des Kapitels wird im Abschnitt 5.4 die Verifikation der Implementierung anhand eines Beispiels dokumentiert.

5.1 Ausgangslage

In diesem Kapitel werden die für diese Aufgabe wichtigen Codefiles erläutert und auf ihre Funktionsweise eingegangen. Es soll dem Leser die notwendigen Grundlagen liefern, um die in Abschnitt 5.2 und Abschnitt 5.3 enthaltenen Gedankengänge und Schlussfolgerungen verstehen zu können.

5.1.1 Netfilter Framework

Netfilter ist ein Framework zur Paketbehandlung. Jedes Protokoll, welches von Netfilter unterstützt wird, definiert sogenannte 'Hooks'. An jedem dieser Hooks wird das Protokoll Netfilter aufrufen und ihm die Paketnummer und die Hooknummer übergeben. Teile des Kernels können sich an den verschiedenen Hooks registrieren.
Wenn ein Paket Netfilter erreicht, wird überprüft, ob sich jemand für diesen Hook registriert hat. Ist dies der Fall, bekommen alle, die sich für diesen Hook registriert haben, der Reihe nach die Möglichkeit das Paket zu untersuchen, vielleicht es zu verändern, zu verwerfen (DROP), es durchzulassen (ACCEPT) oder Netfilter zu beauftragen, das Paket für den Userspace einzureihen. Diese verschiedenen Möglichkeiten, die verwendet werden können, wenn ein Paket einen Test erfüllt, nennt man 'Target' einer Regel 'Rule'. Man unterscheidet, zwischen 'built-in Targets', 'extensions' und 'user-defined chains'.

Built-in Target Es gibt zwei sehr einfache Built-in Targets: DROP und ACCEPT. Wenn eines der beiden als Target verwendet wird, werden keine weiteren Regeln konsultiert.

User-defined chains Zusätzlich zu den drei 'Built-in Chains' (INPUT, OUTPUT und FORWARD) kann der User weitere Chains definieren.

PromethOS Netfilter-Target: ipt_PROMETHOS.c, ipt_PROMETHOS.h

Einträge in der 'promethos'-Tabelle zeigen auf ein spezielles Netfilter-Target, welches 'PROMETHOS' heisst. Dieses Target implementiert die Verwaltungstabellen in Form von Hashtabellen, die nötig sind, um zu vermerken, welche PromethOS-Plugins geladen sind und wieviele Instanzen von ihnen initialisiert wurden. Dazu implementiert das Target zwei Funktionen. Die erste wird von jedem sich initialisierenden PromethOS-Plugin zur Registrierung aufgerufen. Die zweite wird von jedem beendenden PromethOS-Plugin zur Deregistrierung aufgerufen. Im Code 'ipt_PROMETHOS.c' sind das die beiden Funktionen 'promethos_register' und 'promethos_unregister'. Die 'promethos_register'-Funktion reserviert den notwendigen Speicher für den Hashtabellen Eintrag und fügt das neue Plugin in die Hashtabelle ein. Die 'promethos_unregister' Funktion löscht alle Instanzen eines Plugins, entfernt danach das Plugin aus der Hashtabelle und gibt den Speicherbereich wieder frei. Die Funktionsweise der beiden Funktionen sieht folgendermassen aus:

promethos_register(name, target_func, config_func, reconfig_func, print_func, ctrl_func)
01 Alloziiere Speicherbereich fr neues promethos_target $*t$
02 Ausgabe: 'PROMETHOS: promethos_register name called...'
03 *t->promethos_name = name
04 *t->promethos_target_func = target_func
05 *t->promethos_config_func = config_func
06 *t->promethos_reconfig_func = reconfig_func
07 *t->promethos_print_func = print_func
08 *t->promethos_ctrl_func = ctrl_func
09 Füge das Target *t in die Hash Tabelle ein

promethos_unregister(name)
01 Ausgabe: 'PROMETHOS: promethos_unregister name called...'
02 // Entferne alle Instanzen von diesem Plugin von der instance hash
03 Durchsucht die Instanzhashtabelle nach Einträgen mit promethos_name=name
04 Für jede dieser Instanzen wird nochmals überprüft ob es sie wiklich gibt
05 Ist dies bestätigt, wird sie gelöscht
06 Ausgabe: 'PROMETHOS: promethos_unregister (promethos_name: removing instance instance_no
04 Entferne plugin von der plugin hash
05 t = find_target_hash($\&$promethos_table, name)
06 remove_target_hash($\&$promethos_table, name)
07 vfree(t)

Das Target erstellt zwei Dateien im '/proc'-Dateisystem des Kernels. Diese werden benötigt, damit das Userspace-Tool 'iptables' Kernelfunktionen ausführen kann. Die '/proc'-Dateien sind keine reellen Files, sondern dienen als Interface zum Kernel. Die beiden Dateien werden in der init-Funktion generiert. Zusätzlich zum Erzeugen der Dateien werden die Funktionen definiert, welche ausgeführt werden, wenn entweder lesend oder schreibend auf das File zugegriffen wird. Die Implementierung der init-Funktion beinhaltet folgende Punkte:

01 Ausgabe: 'PROMETHOS: init() called...'
02 Registriere das PromethOS-Target
03 Erzeuge die Datei 'instance' im /proc Filesystem
04 Ausgabe: 'PROMETHOS: init() - create_proc_entry succeeded...'
05 Setze die Lese-Funktion auf pf_instance_read
06 Erzeuge die Datei 'management' im /proc Filesystem
07 Ausgabe: 'PROMETHOS: init() - create_proc_entry succeeded...'
08 Setze die Lese-Funktion auf pf_status_read
09 Setze die Schreib-Funktion auf pf_status_write
10 Ausgabe: 'PROMETHOS: init() succeeded...'

Falls das Target nicht mehr benötigt wird, müssen das Target deregistriert und die '/proc' Files wieder gelöscht werden. Dies geschieht mittels der fini-Funktion:

01 Ausgabe: 'PROMETHOS: fini() called...'
02 Unregistriere das PromethOS-Target 03 Lösche die Datei 'instance' im /proc Filesystem 04 Lösche die Datei 'management' im /proc Filesystem 05 Ausgabe: 'PROMETHOS: fini() succeeded...'

Die Datei 'instance' ist erforderlich, falls man ein Plugin neu instanziert und deshalb eine Instanznummer benötigt. Die Datei 'management' wird dazu verwendet, dass man das Plugin neu konfigurieren, bzw. dem Plugin beliebige Daten zukommen lassen kann. Durch Schreiben in die 'management' Datei im /proc-Filesystem, kann durch Angabe der Instanznummer und der zu übergebenden Daten ein Plugin jederzeit neu konfiguriert werden. Weiter ist es möglich, durch Lesen der 'management'-Datei beim PromethOS-Target nachzufragen, welche Plugins in wievielen Instanzen alloziert sind.
Die Lesefunktion des 'instance'-Files im /proc-Filessystem ist auf pf_instance_read gesetzt. Das bedeutet, immer wenn man einen Lesezugriff auf die 'instance'-Datei ausführt, wird die Funktion pf_instance_read gestartet. Die Implementierung dieser Funktion ist folgendermassen umgesetzt:

01 next_instance=1
02 Ausgabe: 'PROMETHOS: pf_instance_read - returning instance next_instance'
03 Speichert die Instanznummer in den Buffer und gibt die Länge zurück

Die Lesefunktion des 'management'-Files ist durch die Funktion pf_status_read implementiert:

01 len = instancePrint(buffer, length, $\&$promethos_table, $\&$promethosi_table)
02 Ausgabe: 'PROMETHOS: pf_status_read - returning len bytes'

wobei die 'instancePrint'-Funktion folgendermassen definiert ist:

instancePrint(buffer, length, target_table, instance_table)
01 Sucht alle Plugins $plugin$ in der target_table
02 Solange es noch ein Plugin gibt und der Buffer noch nicht voll ist
03 Ausgabe: 'PROMETHOS: instancePrint promethos_name'
04 falls es eine promethos_print_func gibt
05 Sucht alle Instanzen $instance$ in der instance_table
06 Solange es noch eine Instanz gibt und der Buffer noch nicht voll ist
07 Fülle Buffer mit promethos_instance
08 Gib den vollen Buffer zurück

Die Schreibfunktion 'pf_status_write' der 'management'-Datei sieht wie folgt aus:

01 Ausgabe: 'PROMETHOS: pf_status_write - accepting count bytes'
02 Parsed die Zeichen, die mittels Buffer übergeben wurden.
03 Aus dem Parsing erhält man ev. den Namen, die Instanz Nr. und ein Config-String
04 Sucht das Plugin und überprüft, ob die Instanz gültig ist
05 Ausgabe: 'PROMETHOS: reconfiguring plugin name instance'
06 Falls dies der Fall ist, wird die promethos_reconfig_func mit dem Config-String ausgeführt

PromethOS Shared-Library: libipt_PROMETHOS.c

Das Userspacetool 'iptables' musste erweitert werden, um die neuen Optionen für das neue PromethOS-Target zu implementieren. Dies geschah, indem dem iptables-Programm eine weitere Shared-Library zugefügt wurde. Diese Shared-Library wird durch die Datei 'iptables/extentions/libipt_PROMETHOS.c' implementiert. Im weiteren Text wird deren Implementation im Detail erläutert.

Die Shared-Library implementiert prinzipiell zwei Teile: Zum Einen überprüft sie allfällige Komandozeilenparameter auf ihre Korrektheit, zum Anderen lädt sie allfällig noch nicht geladene PromethOS-Plugins, die sich dann automatisch beim PromethOS-Target registrieren und somit dem PromethOS-Framework zur Verfügung stehen. Weiter implementiert sie Funktionen zur Ausgabe der gewünschten Informationen zu den Plugins. Die Implementierung umfasst folgende Funktionen:

help Diese Funktion wird verwendet, um die verschiedenen möglichen Optionen anzuzeigen.

check_loaded_module Die Funktion check_loaded_module wird von der 'load'-Funktion ausgeführt, um zu überprüfen, ob ein Modul schon geladen ist. Das geschieht, indem die Funktion das /proc/modules-File ausliest und nach dem zu ladenden Modul sucht. Hat es das Modul gefunden, wird der Wert 1 zurückgegeben. Falls das Modul nicht geladen ist, gibt die Funktion 0 zurück.

load_module Wird von der Funktion final_check aufgerufen. Falls die Funktion check_loaded_module 0 zurückgibt, lädt die Funktion load_module mittels 'INSMOD' das Modul, welches momentan in der globalen Variable 'promethos_name' gespeichert ist.

init Diese Funktion könnte dazu verwendet werden, zusätzlichen Platz in der 'ipt_entry_match'-Struktur zu initialisieren und setzt bestimmte nfcache-Bits. Für PromethOS wird das aber nicht verwendet.

parse Pro Parameter, welcher im Userspacetool iptables verwendet wird, ruft iptables die Funktion parse() auf und übergibt die Position des Parameters. Aufgrund der Position können dann verschiedene Funktionen ausgeführt werden. Damit überprüft werden kann, wie oft eine Option schon ausgeführt worden ist, wird diese Information im Pointer 'flags' gespeichert.

final_check Diese Funktion wird aufgerufen, falls die Command-Line vollständig geparsed wurde. Ihr wird der flags-Pointer übergeben, so dass die Möglichkeit besteht, nochmals zu überprüfen, ob alle Optionen richtig ausgeführt wurden.

print Diese Funktion wird vom iptables code verwendet, um die zusätzliche 'match'-Information für eine Regel auszugeben.

save Die Funktion ist das Pendant zur Funktion 'parse'. Sie wird von 'iptables-save' verwendet, um die Optionen, welche die Regel generiert haben, zu rekonstruieren.

und die Makro Funktion:

_init In der _init() Funktion registriert sich die Shared-Library als neues Target bei dem iptables-Hauptcode. Es wird eine Struktur mit den Zeigern auf die in der Shared-Library implementierten Funktionen und Strukturen übergeben.

Ein Test-Plugin: promethos_TEST.c

Dieser Abschnitt soll die Implementierung eines PromethOS-Plugins, anhand des Plugins 'Test' erläutern. Das Plugin 'Test' wird im weiteren Text als Demo-Plugin verwendet.

Ein PromethOS-Plugin muss die PromethOS Header Datei einbinden und folgende Funktionen definieren:

- laod / unload
- target
- config / reconfig
- print

load In der load-Funktion des Plugins können weitere Initialisierungen für die Plugin-Implementation vorgenommen werden. Diese Funktion wird genau einmal beim Laden des Plugin-Moduls aufgerufen und ist somit geeignet, allfällige globale Initialisierungen für das ganze Plugin vorzunehmen.

unload Die unload-Funktion des Plugins wird genau einmal beim Entfernen des Plugins aus dem Speicher aufgerufen. Hier können allen Instanzen gemeinsame Ressourcen freigegeben werden.

target Die target-Funktion wird für jedes Paket, das den entsprechenden Flow-Definitionen in der PromethOS-Tabelle der Instanz entspricht, aufgerufen. Die Funktion erhält sowohl den sk_buff des Paketes, wie auch die Hook-Nummer, bei der das Paket gesehen wurde, das Input-, wie auch das Output-Interface und die Instanznummer der empfangenden Plugininstanz.

config Die config-Funktion wird für jede Instanz eines PromethOS-Plugins mit dem entsprechenden Konfigurationsstring und der Instanznummer aufgerufen. Hier können Plugininstanzen allfällige Initialisierungen, die jeder Instanz eigen sind, vornehmen.

reconfig Die reconfig-Funktion wird aufgerufen, wenn ein User-Prozess in die /proc/promethos/net/management-Datei schreibt und die entsprechende Plugininstanz angibt. Diese Funktion ist geeignet, einer Instanz zu ermöglichen, zur Laufzeit neu konfiguriert zu werden. Es können auch andere Laufzeitfunktionalitäten implementiert werden.

print Die print-Funktion wird aufgerufen, um eine beliebige Ausgabe zu machen.

Das Einbinden der PromethOS-Header-Datei geschieht durch:

```
#include <linux/netfilter_ipv4/promethos.h>
```

Als Nächstes müssen die load- und unload-Funktionen implementiet werden.

```
static int __init load(void) {
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: load() called...\n');
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: load() succeded...\n');
  return 0;
}

static void __exit unload(void) {
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: unload() called...\n');
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: unload() succeded...\n');
}
```

Die eigentliche Paketverarbeitung eines PromethOS-Plugins erfolgt in seiner target-Funktion. Sie wird für jedes Paket, das den Flow-Definitionen der entsprechenden Plugininstanz in der PromethOS-Tabelle entspricht, aufgerufen.

```
unsigned int promethos_target_test(struct sk_buff **pskb, unsigned int hooknum, const
struct net_device *in, const struct net_device *out, unsigned long instance) {
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: target() called for
  plugin TEST#%lu for hock %u\n', instance, hooknum);
  return IPT_CONTINUE;
}
```

Der Rückgabewert der Funktion entspricht dem Rückgabewert der Netfilter-Target-Funktion und kann die folgenden Werte annehmen:

IPT_CONTINUE , falls das Traversieren der Tabellen weitergehen soll .

NF_ACCEPT , falls das Traversieren der Tabellen weitergehen soll.

NF_DROP , falls das Paket weggeworfen werden soll.

NF_STOLEN , falls das Paket vom Plugin übernommen wird.

NF_QUEUE , falls das Paket für den Userspace eingereiht werden soll.

NF_REPEAT , falls der Hook nochmals aufgerufen werden soll.

Die Initialisierung der einzelnen Plugininstanzen erfolgt in der config-Funktion. Sie wird für jede Instanz einmal aufgerufen. Hier initialisiert sich die entsprechende Instanz und erhält auch den auf der Kommandozeile angegebenen Konfigurationsstring.

```
unsigned int promethos_config_test(const char *config, unsigned long instance) {
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: config() called for
  plugin TEST#%lu with config '%s'\n', instance, config);
  return 1;
}
```

Um eine Plugininstanz zur Laufzeit neu konfigurieren zu können, implementiert jedes Plugin eine reconfig-Funktion. Sie wird aufgerufen, wenn ein Userprozess auf die Datei /proc/promethos/net/management schreibt.

```
unsigned int promethos_reconfig_test(const char *config, unsigned long instance) {
  PROMETHOS_DEBUG_CALL('PROMETHOS_TEST: reconfig() called for
  plugin TEST#%lu with config '%s'\n', instance, config);
  return 1;
}
```

Alle diese im obigen Text erklärten Funktionen müssen dem PromethOS-Framework bekannt gegeben werden. Dies erfolgt durch zwei spezielle Makros, die in der PromethOS-Header-Datei definiert werden.

Mit dem promethos_init-Makro werden Pluginname, wie auch load-, target-, config- und reconfig-Funktionen definiert.

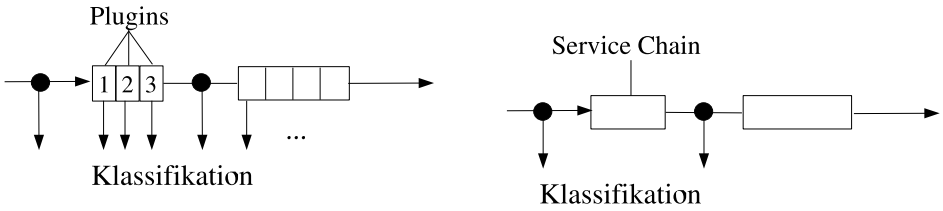
```
promethos_init(
        'TEST',
        load,
        promethos_target_test,
        promethos_config_test,
        promethos_reconfig_test,
        NULL,
        NULL
);
```

Mit dem promethos_exit-Makro wird die unload-Funktion definiert.

```
promethos_exit(unload);
```

5.2 Design - Lösungsvorschlag

In diesem Abschnitt werden zwei mögliche Varianten des Designs für die Optimierung des PromethOS NP-extended Execution Environmnets erläutert und eine davon als Lösungsvariante selektiert.



(a) Bisherige Klassifikation (b) Neue Klassifikation

Abbildung 5.3: Optimierung der Klassifikation

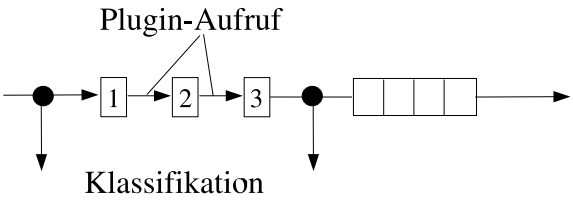


Abbildung 5.4: Design 1

5.2.1 Problemstellung

In der Abbildung 5.3(a) sieht man wo die 'Klassifikation' momentan stattfindet. Unter Klassifikation versteht man den Vergleich eines 'Flows' mit einer 'Rule'. Die Klassifikation wird für jedes PromethOS-Plugin ausgeführt. Dieser Umstand benötigt sehr viel Zeit. Durch die Einführung der neuen Datenstruktur der Service-Chains soll dieser Zeitaufwand verkleinert werden, indem nur noch eine Klassifikation pro Service-Chain durchgeführt wird. In der Abbildung 5.3(b) kann man die neue Datenstruktur sehen und wo die Klassifikation durchgeführt wird.

5.2.2 Lösungsvorschläge

1. Möglichkeit

In der Abbildung 5.4 sieht man die erste Möglichkeit für das Design der Service-Chain. Die Idee, die hinter dem Design steckt, ist diejenige, dass die Plugins sich selber aufrufen können. Das bedeutet, dass die Klassifikation nur für das erste Plugin stattfindet. Danach ruft das erste Plugin das zweite auf, das zweite Plugin das dritte, usw. Der Benutzer definiert die Reihenfolge und die Anzahl der Plugins, welche für ein bestimmtes Target ausgeführt werden sollen. Als Target wird nur das erste Plugin übergeben.

2. Möglichkeit

In der Abbildung 5.5 ist die zweite Möglichkeit eines Designs der Service-Chain dargestellt. Die Service-Chain entspricht einer Liste von Plugins, welche die Plugins in der richtigen Reihenfolge enthält. Der Benutzer definiert, welches Plugin an welcher Position in die Liste eingefügt wird. Als Target wird die ganze Liste übergeben. Das Target ist verantwortlich für die Ausführung jedes Plugins in der Liste.

Selektionsentscheid

Aufgrund der klaren Struktur und klaren Rückgabewerten hat man sich für den zweiten Designvorschlag entschieden. Ein weiterer Grund ist, dass das Target für die korrekte Ausführung der Plugins in der richtigen Reihenfolge verantwortlich ist. Da Menschen (Benutzer) häufig Fehler machen, ist das sicher ein Vorteil gegenüber des ersten Designvorschlags, wo der Benutzer durch die Spezifikation der Plugins die korrekte Ausführung der Plugins implementiert.

5.3.2 Neue Funktionen in ’ipt_PROMETHOS.c’

Damit man vom Userspace Tool ’iptables’ Kernelfunktionen ausführen kann, wird ein neues File im ’/proc’ Filessystem erzeugt. Das File ’chainstat’ wird für die Verwaltung der Service-Chains verwendet. Weiter wird die Funktion ’pf_chain_write’ dem Schreibzugriff auf das File zugewiesen. Das bedeutet, dass immer, wenn jemand auf das File ’chainstat’ schreibt, die Funktion ’pf_chain_write’ ausgeführt wird. Die Funktion ’pf_chain_write’ implementiert alle Befehle, die für die Verwaltung der Service-Chain notwendig sind.

Die Befehle lauten:

i plugin Erzeugt eine neue Instanz des Plugins ’plugin’.

c chain Erzeugt eine neue Service-Chain ’chain’ und eine neue Instanznummer. Falls die Service-Chain ’chain’ schon existiert, wird nur eine neue Instanznummer generiert.

a chainInst pluginInst Fügt der Service-Chain mit der Instanznummer ’chainInst’ das Plugin mit der Instanznummer ’pluginInst’ hinzu.

l chainInst Listet alle Plugins der Service-Chain mit der Instanznummer ’chainInst’ auf.

r chainInst pluginInstA pluginInstB Ersetzt in der Service-Chain mit der Instanznummer ’chainInst’ das Plugin mit der Instanznummer ’pluginInstA’ durch das Plugin mit der Instanznummer ’pluginInstB’.

p chainInst pluginInst Entfernt in der Service-Chain mit der Instanznummer ’chainInst’ das Plugin mit der Instanznummer ’pluginInst’.

d chainInst Löscht die Service-Chain mit der Instanznummer ’chainInst’.

Die Befehle werden ausgeführt, indem sie durch einen ’echo’-Befehl in das File ’chainstat’ geschrieben werden. Die Ausführung eines beliebigen Befehls ’command’ sieht folgendermassen aus:

```
echo command > /proc/promethos/net/chainstat
```

Auf eine detaillierte Beschreibung der Implementierungen der Befehle musste aus Zeitgründen verzichtet werden. Der Code wurde aber absichtlich ausführlich kommentiert, damit er trotz fehlender Beschreibung leicht verständlich ist.

Die Funktionalität von ’target()’ muss um die Implementierung der Service-Chain erweitert werden. Anhand des Wertes für ’type’, welcher in ’targinfo’ gespeichert ist, kann erkannt werden, um was für ein Target es sich handelt. Ist ’type’ gleich 1 handelt es sich um ein gewöhnliches Plugin. Hat ’type’ den Wert 2 handelt es sich um eine Service-Chain. Der Fall des Plugins ist schon implementiert. Im Falle einer Service-Chain, muss für jedes Plugin, welches sich in der Service-Chain befindet, die Funktion ’promethos_target_func()’ ausgeführt werden.

Für die Funktion ’checkentry()’ wird nur für den Fall, dass es sich beim Eintrag um ein Plugin handelt, etwas ausgeführt. Diese Implementierung für das Plugin ist ebenfalls scho im Source-Code vorhanden.

5.4 Evaluation

Aus Zeitgründen kann nur eine Verifikation durchgeführt werden. Eine Performance-Messung hatte keinen Platz mehr in dieser Arbeit.

5.4.1 Verifikation

Um die Funktionsweise der Service-Chain zu demonstrieren und zu zeigen, dass die Erweiterung um die Service-Chain-Funkion vollständig in das PromethOS NP-extended Execution Environment eingebettet werden konnte, wird das Skript 'demo.sh' als Beispiel ausgeführt. In diesem Skript wird zuerst ein herkömmliches PromethOS-Plugin erzeugt und mittels einem ICMP-Request bzw. ICMP-Reply die Funktionsweise des PromethOS-Targets getestet. In einem nächsten Schritt wird eine Service-Chain erstellt und mit drei Plugininstanzen gefüllt. Um zu zeigen, dass das PromethOS-Target die Service-Chain-Erweiterung korrekt implementiert, wird erneut ein ICMP-Request, bzw. ICMP-Reply ausgeführt. Danach wird gezeigt, dass die Funktionen 'replace', 'purge' und 'delete' korrekt funktionieren.
Im folgenden Abschnitt, sind die Resultate für die einzelnen Befehle aufgeführt und kommentiert.

Resultate

Zuerst wird mittels dem Userspace-Tool 'iptables' das PromethOS-Plugin 'TEST' mit der Konfiguration 'config test' instantiert und beim 'PREROUTING'-Hook registriert.

iptables -t promethos -A PREROUTING -s 127.0.0.1 -j PROMETHOS –plugin **TEST** –autoinstance –config 'config test'

Der 'iptables'-Aufruf führt zuerst die 'init'-Funktion aus und lädt dann das PromethOS-Target 'ipt_PROMETHOS'. Als Nächstes wird der Kernel für eine Instanznummer für das Plugin TEST angefragt. Er gibt die Instanznummer 1 zurück. Es wurde eine neue Instanz des Plugins TEST mit der Instanznummer 1 erzeugt. Am Schluss wird das Plugin geladen und steht dem Kernel zur Verfügung.

Die Ausgabe für den Userspace sieht folgendermassen aus:

```
Using /lib/modules/2.4.27-promethos/kernel/net/ipv4/netfilter/promethos_TEST.o
libipt_PROMETHOS: init() called...
libipt_PROMETHOS: init() succeded...
libipt: typedef 1 OK
libipt_PROMETHOS: parse() trying to load module ipt_PROMETHOS
libipt_PROMETHOS: autoinstance, requesting unique instance from kernel.
libipt_PROMETHOS: got unique instance 1 from kernel.
libipt_PROMETHOS: load_module() trying to load module promethos_TEST
# About to execute '/sbin/insmod promethos_TEST'
```

Im Kernel werden mehrere Operationen durchgeführt:

Als Erstes werden vier Files im /proc-Filessystem generiert. Diese Files werden als Schnittstellen zum Kernel verwendet. Auf die Anfrage von 'iptables' für eine Instanznummer wird die Funktion 'pf_instance_read' ausgeführt und gibt die Instanznummer 1 zurück. Danach werden das Plugin TEST und die PromethOS-Tabelle 'PROMETHOS_TABLE' registriert. Die Funktion 'checkentry' führt am Ende die Konfiguration des Plugins TEST aus.

```
localhost kernel: PROMETHOS: init() called...
localhost kernel: PROMETHOS: init() - create_proc_entry('/proc/promethos/net/instance') succeeded...
localhost kernel: PROMETHOS: init() - create_proc_entry('/proc/promethos/net/management') succeeded...
localhost kernel: PROMETHOS: init() - create_proc_entry('/proc/promethos/net/chain') succeeded...
localhost kernel: PROMETHOS: init() - create_proc_entry('/proc/promethos/net/chainstat') succeeded...
localhost kernel: PROMETHOS: init() succeded...
localhost kernel: PROMETHOS: pf_instance_read - returning instance 1
```

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
PROMETHOS  all  --  localhost.localdomain  anywhere          PROMETHOS TEST#1

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Abbildung 5.8: Die PromethOS-Tabelle mit dem Plugin TEST

localhost kernel: PROMETHOS: (version independent) promethos_init(TEST) called...
localhost kernel: PROMETHOS_TEST: load() called...
localhost kernel: PROMETHOS_TEST: load() succeded...
localhost kernel: PROMETHOS: promethos_register(TEST) called...
localhost kernel: PROMETHOS: (version independent) promethos_init(TEST) succeded...
localhost kernel: PROMETHOS_TABLE: init() called...
localhost kernel: PROMETHOS_TABLE: registering table promethos...
localhost kernel: PROMETHOS_TABLE: registering ipt_ops[0]...
localhost kernel: PROMETHOS_TABLE: registering ipt_ops[1]...
localhost kernel: PROMETHOS_TABLE: registering ipt_ops[2]...
localhost kernel: PROMETHOS_TABLE: registering ipt_ops[3]...
localhost kernel: PROMETHOS_TABLE: registering ipt_ops[4]...
localhost kernel: PROMETHOS_TABLE: init() succeded
localhost kernel: PROMETHOS: checkentry() called...
localhost kernel: PROMETHOS: configuring plugin TEST instance 1.
localhost kernel: PROMETHOS: config() dispatching to plugin TEST#1
localhost kernel: PROMETHOS_TEST: config() called for plugin TEST#1 with config 'config test'
localhost kernel: PROMETHOS: checkentry(TEST): adding plugin instance 1

Um die Funktionsweise des PromethOS-Targets für einzelne Plugins zu verifizieren, wird ein ICMP-Request bzw. ICMP-Reply mittels 'ping' ausgeführt.

ping -c 1 127.0.0.1

Als Resultat ruft der Kernel die 'target'-Funktion des Plugins TEST mit der Instanznummer 1 auf. Das geschieht zweimal: einmal für den ICMP-Request und einmal für den ICMP-Reply.

localhost kernel: PROMETHOS: target() called for plugin TEST#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#1 localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#1 for hock 0
localhost kernel: PROMETHOS: target() called for plugin TEST#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#1
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#1 for hock 0

Dass das Plugin TEST mit der Instanznummer 1 am richtigen Hook registriert worden ist, sieht man in Abbildung 5.8, nachdem der folgende Befehl ausgeführt wurde.

iptables -t promethos -L

Somit ist gezeigt worden, dass die Funktionsweise des PromethOS-Targets für die Plugins auch nach der Erweiterung für die Service-Chains einwandfrei funktioniert.

Als Nächstes wird gezeigt, dass die Implementierung der Erweiterung des PromethOS-Targets auch für die Service-Chains korrekt funktioniert.

Mittels ’iptables’ wird eine neue Service-Chain ’servChain’ erzeugt und wie das Plugin am PREROUTING-Hook registriert.

iptables -t promethos -A PREROUTING -s 127.0.0.1 -p icmp -j PROMETHOS –chain **servChain**

Dieser Befehl verursacht einen Schreibzugriff mittels ’echo ’c servChain” auf das File ’chainstat’ des /proc Filesystems, welcher wiederum den Aufruf der Funktion ’pf_chain_write’ auslöst.

libipt_PROMETHOS: init() called...
libipt_PROMETHOS: init() succeded...
libipt_PROMETHOS: Chain name servChain copied...
libipt_PROMETHOS: Command echo c servChain > /proc/promethos/net/chainstat starting...
libipt_PROMETHOS: Command succeeded...
libipt_PROMETHOS: Got unique instance nr 1 from kernel.

Die Funktion ’pf_chain_write’ hat zur Folge, dass der Kernel zuerst überprüft, ob die Service-Chain ’servChain’ bereits existiert und falls nicht, wird die Service-Chain erzeugt. Das Erzeugen der Service-Chain wird durch einen Eintrag in die Hashtabelle der Service-Chains abgeschlossen. Wie immer nach einem ’iptables’-Aufruf, wird für jeden Eintrag der PromethOS-Tabelle die Funktion ’checkentry’ ausgeführt.

localhost kernel: PROMETHOS: pf_chain_write() - Chain servChain doesn’t exist - creating...
localhost kernel: PROMETHOS: pf_chain_write() - vmalloc succeeded
localhost kernel: PROMETHOS: pf_chain_write() - Chain inserted...
localhost kernel: PROMETHOS: pf_chain_write() - Chain creation chain servChain succeeded...
localhost kernel: PROMETHOS: pf_chain_instance_read - returning 2 bytes
localhost kernel: PROMETHOS: checkentry() called...
localhost kernel: PROMETHOS: checkentry() called...

Damit ein Plugin in die Service-Chain eingefügt werden kann, muss es geladen und instanziert sein. In diesem Beispiel wird auf das Laden verzichtet und das bereits geladene Plugin TEST nochmals instantiert. Mittels ’my_config_1’ kann die neue Instanz konfiguriert werden. Im Output des Kernels sieht man, dass eine neue Instanznummer 2 (Instanznummer 1 ist schon vergeben) generiert und die Konfigurationsfunktion ’config’ der Plugininstanz aufgerufen wird.

echo ’i TEST my_config_1’ > /proc/promethos/net/chainstat
localhost kernel: PROMETHOS: pf_chain_write - accepting 19 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Instantiate plugin
localhost kernel: PROMETHOS: pf_status_write() - Configuration defined...
localhost kernel: PROMETHOS: pf_status_write() - Plugin Name: TEST
localhost kernel: PROMETHOS: pf_status_write() - Plugin Config: my_config_1
localhost kernel: PROMETHOS: pf_chain_write() - New plugin instance TEST#2 created...
localhost kernel: PROMETHOS: pf_chain_write() - config() dispatching to plugin TEST#2...
localhost kernel: PROMETHOS_TEST: config() called for plugin TEST#2 with config ’my_config_1’
localhost kernel: PROMETHOS: pf_chain_write() - config() succeeded.

Nun kann die neue Plugininstanz in die Service-Chain eingefügt werden. Der Kernel vergrössert den Speicherbereich der Service-Chain um einen Eintrag und fügt die Plugininstanz an der ersten freien Stelle (Position 0) ein. Da die Service-Chain vorher noch keinen Eintrag

hatte, versucht der Kernel vergebens den Speicherbereich des vorherigen Eintrags freizugeben.

echo ’a 1 2’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 6 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Append to chain
localhost kernel: PROMETHOS: pf_status_write() - Trying to append Plugininstance 2 to Chain servChain#1...
localhost kernel: PROMETHOS: pf_chain_write() - Try to vmalloc memory ...
localhost kernel: PROMETHOS: pf_chain_write() - Vmalloc memory succeeded...
localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#2 to chain servChain#1
localhost kernel: PROMETHOS: pf_chain_write() - Trying vfree old mem space...
localhost kernel: PROMETHOS: pf_chain_write() - vfree not succeceded for 0. time(s) ...
localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#2 to chain servChain#1 at position 0 succeeded

In den nächsten Zeilen, wird dieses Vorgehen für zwei weitere Instanzen des Plugins TEST wiederholt.

echo ’i TEST my_config_2’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 19 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Instantiate plugin
localhost kernel: PROMETHOS: pf_status_write() - Configuration defined...
localhost kernel: PROMETHOS: pf_status_write() - Plugin Name: TEST
localhost kernel: PROMETHOS: pf_status_write() - Plugin Config: my_config_2
localhost kernel: PROMETHOS: pf_chain_write() - New plugin instance TEST#3 created...
localhost kernel: PROMETHOS: pf_chain_write() - config() dispatching to plugin TEST#3...
localhost kernel: PROMETHOS_TEST: config() called for plugin TEST#3 with config ’my_config_2’
localhost kernel: PROMETHOS: pf_chain_write() - config() succeeded.

echo ’a 1 3’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 6 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Append to chain
localhost kernel: PROMETHOS: pf_status_write() - Trying to append Plugininstance 3 to Chain servChain#1...
localhost kernel: PROMETHOS: pf_chain_write() - Try to vmalloc memory ...
localhost kernel: PROMETHOS: pf_chain_write() - Vmalloc memory succeeded...
localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#3 to chain servChain#1
localhost kernel: PROMETHOS: pf_chain_write() - Trying vfree old mem space...
localhost kernel: PROMETHOS: pf_chain_write() - vfree succeceded for 1. time(s) ...
localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#3 to chain servChain#1 at position 1 succeeded

echo ’i TEST my_config_3’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 19 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Instantiate plugin
localhost kernel: PROMETHOS: pf_status_write() - Configuration defined...
localhost kernel: PROMETHOS: pf_status_write() - Plugin Name: TEST
localhost kernel: PROMETHOS: pf_status_write() - Plugin Config: my_config_3
localhost kernel: PROMETHOS: pf_chain_write() - New plugin instance TEST#4 created...
localhost kernel: PROMETHOS: pf_chain_write() - config() dispatching to plugin TEST#4...
localhost kernel: PROMETHOS_TEST: config() called for plugin TEST#4 with config

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
PROMETHOS  all  --  localhost.localdomain  anywhere          PROMETHOS TEST#1
PROMETHOS  icmp --  localhost.localdomain  anywhere          PROMETHOS servChain#1

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Abbildung 5.9: Die PromethOS-Tabelle mit dem Plugin TEST und der Service-Chain servChain

’my_config_3’
localhost kernel: PROMETHOS: pf_chain_write() - config() succeeded.

echo ’a 1 4’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 6 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Append to chain
localhost kernel: PROMETHOS: pf_status_write() - Trying to append Plugininstance 4 to Chain servChain#1...
localhost kernel: PROMETHOS: pf_chain_write() - Try to vmalloc memory ...
localhost kernel: PROMETHOS: pf_chain_write() - Vmalloc memory succeeded...
< localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#4 to chain servChain#1
localhost kernel: PROMETHOS: pf_chain_write() - Trying vfree old mem space...
localhost kernel: PROMETHOS: pf_chain_write() - vfree succeceded for 2. time(s) ...
localhost kernel: PROMETHOS: pf_chain_write() - Appending Plugin TEST#4 to chain servChain#1 at position 2 succeeded

Wenn man die PromethOS-Tabelle in der Abbildung 5.9 überprüft, sieht man, dass die Service-Chain korrekt am PREROUTING-Hook eingefügt worden ist.

iptables -t promethos -L

Um zu kontrollieren, ob die Plugininstanzen auch tatsächlich in die Service-Chain eingefügt worden sind, wird der Inhalt der Service-Chain ausgegeben. Wie man sieht, sind die Plugininstanzen korrekt eingefügt worden.

echo ’l 1’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 4 bytes
localhost kernel: PROMETHOS: pf_chain_write() - List chain
localhost kernel: PROMETHOS: pf_chain_write() - List chain servChain#1 :
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#2 Position: 0
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#3 Position: 1
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#4 Position: 2

Damit man die Funktionsweise der Service-Chain-Erweiterung des PromethOS-Targets testen kann, wird erneut ein ICMP-Request und ein ICMP-Reply mittels ’ping’ erzeugt. Im Output des Kernels sieht man deutlich, wie die ’target’-Funktionen der drei Plugininstanzen der Service-Chain der Reihe nach aufgerufen werden. Aus oben erwähntem Grund wird jede ’target’ Funktion zweimal aufgerufen.

ping -c 1 127.0.0.1

localhost kernel: PROMETHOS: target() called for plugin TEST#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#1
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#1 for hock 0
localhost kernel: PROMETHOS: target() called for chain servChain#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#2
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#2 for hock 0
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#3
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#3 for hock 0
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#4
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#4 for hock 0
< localhost kernel: PROMETHOS: target() called for plugin TEST#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#1
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#1 for hock 0
localhost kernel: PROMETHOS: target() called for chain servChain#1
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#2
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#2 for hock 0
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#3
< localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#3 for hock 0
localhost kernel: PROMETHOS: target() dispatching to plugin TEST#4
localhost kernel: PROMETHOS_TEST: target() called for plugin TEST#4 for hock 0

In den folgenden Zeilen wird die Funktionsweise der ’replace’-, ’purge’- und ’delete’- Befehle demonstriert und verifiziert. Da der Output des Kernels den Ablauf sehr detailliert zeigt, wird auf einen zusätzlichen Kommentar verzichtet.

echo ’r 1 2 3’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 8 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Replace in chain
localhost kernel: PROMETHOS: pf_status_write() - Trying to replace Plugininstance 2 with Plugininstance 3 in Chain servChain#1
localhost kernel: PROMETHOS: pf_chain_write() - Replacing Plugininstance 2 with Plugininstance 3 in Chain servChain#1...
localhost kernel: PROMETHOS: pf_chain_write() - Replaced Plugin TEST#2 with Plugin TEST#3

echo ’l 1’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 4 bytes
localhost kernel: PROMETHOS: pf_chain_write() - List chain
localhost kernel: PROMETHOS: pf_chain_write() - List chain servChain#1 :
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#3 Position: 0
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#3 Position: 1
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#4 Position: 2

echo ’p 1 4’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 6 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Purge from chain
localhost kernel: PROMETHOS: pf_chain_write() - Trying to purge Plugin TEST#4 from chain servChain#1
localhost kernel: PROMETHOS: pf_chain_write() - Purging Plugin TEST#4 succeeded...

echo ’l 1’ > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 4 bytes

localhost kernel: PROMETHOS: pf_chain_write() - List chain
localhost kernel: PROMETHOS: pf_chain_write() - List chain servChain#1 :
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#3 Position: 0
localhost kernel: PROMETHOS: pf_chain_write() - Plugin: TEST#3 Position: 1

echo 'd 1' > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 4 bytes
localhost kernel: PROMETHOS: pf_chain_write() - Delete chain
localhost kernel: PROMETHOS: pf_chain_write() - Deleting chain: servChain#1 ...
localhost kernel: PROMETHOS: pf_chain_write() - Deleting chain succeeded ...

echo 'l 1' > /proc/promethos/net/chainstat

localhost kernel: PROMETHOS: pf_chain_write - accepting 4 bytes
localhost kernel: PROMETHOS: pf_chain_write() - List chain
localhost kernel: PROMETHOS: pf_chain_write() - Not found Chain_Instance 1 in Chain Instance Hash Table

iptables -t promethos -F
rmmod promethos_TEST

localhost kernel: PROMETHOS: (version independent) promethos_exit(TEST) called...
localhost kernel: PROMETHOS: promethos_unregister(TEST) called...
localhost kernel: PROMETHOS: promethos_unregister(TEST): removing instance 1
localhost kernel: PROMETHOS: promethos_unregister(TEST): removing instance 2
localhost kernel: PROMETHOS: promethos_unregister(TEST): removing instance 3
localhost kernel: PROMETHOS: promethos_unregister(TEST): removing instance 4
localhost kernel: PROMETHOS_TEST: unload() called...
localhost kernel: PROMETHOS_TEST: unload() succeded...
localhost kernel: PROMETHOS: (version independent) promethos_exit(TEST) succeded...

rmmod iptable_promethos

localhost kernel: PROMETHOS_TABLE: fini() called...
localhost kernel: PROMETHOS_TABLE: unregistering ipt_ops[0]..
localhost kernel: PROMETHOS_TABLE: unregistering ipt_ops[1]..
localhost kernel: PROMETHOS_TABLE: unregistering ipt_ops[2]..
localhost kernel: PROMETHOS_TABLE: unregistering ipt_ops[3]..
localhost kernel: PROMETHOS_TABLE: unregistering ipt_ops[4]..
localhost kernel: PROMETHOS_TABLE: unregistering promethos table...
localhost kernel: PROMETHOS_TABLE: fini() succeded

rmmod ipt_PROMETHOS

localhost kernel: PROMETHOS: fini() called...
localhost kernel: PROMETHOS: fini() succeded...

Fazit

Anhand dieser Resultate kann man sagen, dass die Erweiterung des PromethOS-Targets um die Funktionalität der Service-Chains vollständig in das bestehende PromethOS NP-extended Execution Environment eingebunden werden konnte.

Kapitel 6

Zusammenfassung und Ausblick

Dieses Kapitel bildet den Abschluss dieser Dokumentation. Der Zusammenfassung der bisherigen Kapitel folgen eine Übersicht über die erreichten Ziele und schliesslich der Ausblick auf mögliche weiterführende Arbeiten.

6.1 Zusammenfassung

Im Kapitel 1 wurde die Problemstellung und die Motivation dieser Arbeit geschildert. Die Problemstellung besteht darin, dass ein effizienter Algorithmus für das Mapping von Service-Chains auf einen Prozessor-Graphen entwickelt und evaluiert werden sollte. In einem zweiten Teil der Problemstellung, sollte das PromethOS NP-extended Execution Environment optimiert werden. Konkret wollte man die Zahl der Klassifikationen von Flows reduzieren.

Im Kapitel 2 wurden die Ausgangslage beschrieben und die Schnittstellen zum reellen PromethOS NP Knoten spezifiziert, so dass eine spätere Einbindung des Algorithmus in den PromethOS NP Knoten einfach gemacht werden kann.

Im Kapitel 3 wurden die Ideen und die Implementierung von zwei Algorithmen präsentiert. Zunächst wurde der Lösungsansatz des Layered-Graphen dokumentiert und implementiert. Bei diesem Ansatz handelt es sich um eine Umformulierung des Problems: 'Finde das optimale Mapping eines Service-Graphen auf einen Prozessor-Graphen' zum Problem: 'Finde den kürzesten Pfad von einem Startknoten zu einem Endknoten durch den Layered-Graphen'. Auf dem Fundament des Layered-Graphen wurde der Exhaustive-Search-Algorithmus entwickelt, welcher alle möglichen Pfade durch den Layered-Graphen berechnet. Weiter wurde ebenfalls auf dem Fundament des Layered-Graphen der Extended-Shortest-Path-Algorithmus implementiert. Dieser Algorithmus basiert auf dem Shortest-Path-Algorithmus von Dijkstra und erzeugt einen so genannten 'Spanning-Tree', welcher das Mapping symbolisiert. Diese beiden Algorithmen wurden evaluiert und miteinander verglichen. Am Ende des Kapitels wurde der Vergleich des Lösungsansatzes zum Ansatz von Sumi Y. Choi und Jonathan Turner gezogen, den sie im Paper [2] und im Paper [1] veröffentlicht haben.

Im Kapitel 4 wurden die beiden Algorithmen und der Layered-Graph optimiert, so dass eine bessere Performance für den Speicherbedarf und den Zeitaufwand erreicht werden konnte. Die Implementierungen wurden evaluiert und mit den nicht optimierten Implementierungen der Algorithmen verglichen.

Im Kapitel 5 wurde zuerst die Ausgangslage anhand des Netfilter-Frameworks und des PromethOS NP-extended Execution Environments aufgeführt. Danach wurden zwei mögliche Lösungsvorschläge vorgestellt und eine davon selektiert. Dieser Lösungsansatz der Service-Chain ist implementiert und anhand eines Beispiels verifiziert worden. Dieses Kapitel dokumentiert die einzelnen Implementierungsschritte, sowie die anschliessende Verifikation.

6.2 Erreichte Ziele

Die wichtigsten Ziele, welche im Laufe dieser Arbeit erreicht wurden, sind:

- Es wurden zwei Algorithmen implementiert, welche einen Service-Graphen korrekt auf einen Prozessor-Graphen abbilden. Das Mapping, welches durch die Algorithmen berechnet wird, entspricht dem Mapping mit dem kleinsten Delay.
- Die Algorithmen basieren auf wohl definierten Interfaces, welche es ermöglichen, die Algorithmen leicht in den PromethOS NP Knoten einzubinden.
- Durch die Optimierung der Algorithmen, konnte der Speicherbedarf und der Zeitaufwand markant reduziert werden. Ob man akzeptable Werte erreicht hat, wird sich mit einer Evaluation der Algorithmen auf dem reellen PromethOS NP Knoten mit einem reellen Service noch herausstellen.
- Das PromethOS NP-extended Execution Environment wurde durch die Service-Chain-Funktionalität erweitert. Es können sämtliche wünschenswerte Funktionen für eine Service-Chain ausgeführt werden, wie zum Beispiel das Hinzufügen oder das Löschen von Plugins.

6.3 Ausblick

Dieser Abschnitt befasst sich mit den Ansatzpunkten, welche diese Arbeit für zukünftige Arbeiten bietet.

Die Algorithmen wurden bis jetzt nur anhand von fiktiven Werten evaluiert. In einer zukünftigen Arbeit könnte die Evaluation auf einem reellen PromethOS NP Knoten für einen reellen Service-Graphen durchgeführt werden. Erst dann wird sich zeigen, ob die durchgeführten Optimierungen schon ausreichend sind, damit man die Algorithmen in der Praxis anwenden kann.

Damit diese Evaluation durchgeführt werden kann, müssen die Algorithmen vollständig in den PromethOS NP Knoten eingebunden werden.

Weiter besteht die Möglichkeit neue Algorithmen zu entwerfen, oder die bestehenden Algorithmen weiter zu optimieren.

Für das PromethOS NP-extended Execution Environment müssen die Laufzeittests nach der Erweiterung durch die Service-Chain Funktionalität noch durchgeführt werden. Die Resultate können danach mit den Laufzeittests für das PromethOS NP-extended Execution Environment ohne Service-Chain Funktionalität verglichen werden. Dadurch könnte eine Aussage über den Performancegewinn gemacht werden.

Anhang A

Aufgabenstellung

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Sommersemester 2004

Diplomarbeit

für

Sandro Ribolla

Betreuer: Lukas Ruf
Stellvertreter: Matthias Bossardt

Ausgabe: 05.04.2004
Abgabe: 04.08.2004

Mapping Services Onto Hierarchy-extended Active Network Nodes

1 Einführung

Moderne Router Architekturen basieren verwenden eine Prozessorhierarchie, um die benötigte Verarbeitungskapazität skalierbar zur Verfügung zu stellen, die benötigt wird, um mit dem Zuwachs der Netzwerkbandbreite Schritt halten zu können. Bis anhin wurde diese Prozessorhierarchie statisch programmiert, dh. der Service Code (z.B. Paketfilterung) wurde zur Startzeit des Routers auf den einzelnen Prozessoren installiert. Active Networking [7] vereinfacht den Prozess des Code Updatings, indem es Konzepte und Mechanismen zur dynamischen Programmierung von Aktiven Netzwerkknoten zur Verfügung stellt.
Code wird in sog. Execution Environments (EEs) installiert, instantiert und ausgeführt.Am Insitut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich wurden ein Linux Kernel Space EE für Einzelprozessoren [4] und eine Erweiterung für Netzwerkprozessor-basierte Knoten [5, 6] in den Projekten *PromethOS* resp. *PromethOS NP* entwickelt.
Services werden als Graph von Service Komponenten spezifiziert. Diese Komponenten können auf verschiedenen Prozessoren der Prozessorhierarchie installiert werden, so dass ein Service die gesamte Hierarchie umspannen kann. Die Identifikation geeigneter Prozessoren im on-line Fall ist jedoch äusserst schwierig, wenn der angewendete Ressource Allokationsalgorithmus einerseits skalieren und andrerseits eine gute Lösung liefern soll. In [1, 2] wurde ein Ansatz mit einem Layered Graph untersucht, auf welchem in dieser Arbeit aufgebaut werden soll, indem pro Heterogenitätsstufe in der Prozessorhierarchie ein Layered Graph aufgebaut wird.

2 Aufgaben: Mapping Services Onto Hierarchy-extended Active Network Nodes

Im Rahmen dieser Diplomarbeit soll einerseits das Problem der Ressource Allokation untersucht und gelöst, andrerseits das Konzept der Service Chains [5] im PromethOS NP Execution Environment umgesetzt werden.

Die vorliegende Implementation des Linux Kernel Space EE von PromethOS NP liefert den Proof of Concept. Um die in [5] beschriebene, effiziente Architektur der Service Chains im Linux Kernel zu erreichen, muss das vorliegende EE dahingehend optimiert werden, dass die Funktionalität der Programmable Distributors und des Service Chain Control Bus unterstützt wird. Die Service Chains sollen sowohl an den Fast Path gleich wie auch an den normalen, Slow Path angehängt werden können.

3 Vorgehen

- Richten Sie sich eine Entwicklungsumgebung (GNU Tools) unter Linux ein.
- Machen Sie sich vertraut mit den Unterlagen (Dokumentation und Source Code) zum PromethOS NP EE.
- Lesen Sie sich in die bestehenden Ansätze zur Ressource Allokation [1, 2] ein.
- Erstellen Sie einen Zeitplan, in welchem Sie die von Ihnen zu erreichenden Meilensteine Ihrer Arbeit identifizieren.
- Untersuchen Sie verschiedene Varianten zur Lösung des Ressource Allokation Algorithmus. Als Vergleich zu Ihrer Lösung können Sie einen Exhaustive Search einsetzen.
- Entwickeln Sie eine Architektur für die Erweiterung des Linux Kernel Space EEs für die Service Chains wie unter 2 beschrieben.
- Implementieren Sie Ihre Architektur der Service Chains und den Ressource Allokationsalgortihmus als Service Mapper.
- Verifizieren, evaluieren und demonstrieren Sie das Erreichte.
- Dokumentieren Sie die Resultate ausführlich.

Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte und untersuchten Varianten vor dem definitiven Variantenentscheid ausführlich schriftlich festzuhalten.

4 Organisatorische Hinweise

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Professor Plattner.
- Am Schluss der Arbeit muss eine Präsentation von **20 Minuten** im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentation soll die Arbeit Interessierten praktisch vorgeführt werden.
- Die Arbeit muss regelmässig auf dem PromethOS SVN-Server `<https://svn.promethos.org:8443/svn/PromethOS>` gesichert werden. Es ist darauf zu achten, dass die **richtige Branch** verwendet wird.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Ihre Arbeit soll die Erkenntnisse und Implementationen der Diplomarbeiten von Michel Dänzer `<michel@daenzer.net>` und Adrian von Bidder `<avbidder@fortytwo.ch>` verwenden.

- Die Dokumentation ist mit dem Satzsystem LaTeX zu erstellen. Illustrationen müssen mit einem OpenSource Programm unter Linux erstellt werden.
- Es ist ein mit Bindespiralen gebundener Schlussbericht über die geleisteten Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung , einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigen, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind.
  Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt und eingesetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL) [3].
- Für die Arbeit werden jedoch Informationen eingesetzt, welche nur durch das Unterzeichnen eines NDA (Non-Disclosure Agreement) mit IBM Corp. erhalten wurden. Die Arbeit als ganzes (Programmcode und Dokumentation) sowie alle Informationen, die unter das NDA fallen, dürfen nur an Dritte weitergegeben werden, wenn eine schriftliche Einwilligung von IBM Corp. vorliegt. Falls NDA-Informationen auf einem anderen Rechner als `<www.promethos.org>` gespeichert werden, muss sichergestellt werden, dass kein unberechtigter Zugriff auf diese möglich ist.
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.

Literatur

[1] S. Choi and J. Turner. Configuring Sessions in Programmable Networks with Capacity Constraints. In *Proc. of IEEE ICC*, May 2003.

[2] S. Choi, T. Wolf, and J. Turner. Configuring Sessions in Programmable Networks. In *Proc. of IEEE Infocom*, Apr. 2001.

[3] GNU General Public License v2. http://www.gnu.org/copyleft/gpl.html, June 1991.

[4] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proc. of 4th Annual Int. Working Conf. on Active Networking (IWAN), Zürich, Switzerland*, number 2546 in Lecture Notes in Computer Science. Springer Verlag, Heidelberg, Dec. 2002.

[5] L. Ruf, R. Keller, and B. Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In *Proc. of 2004 ACS/IEEE Int. Conf. on Pervasive Services (ICPS'2004), Beirut, Lebanon*. IEEE, Jul. 2004.

[6] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner. Towards High-performance Active Networking. In *Proc. of 5th Annual Int. Working Conf. on Active Networking (IWAN), Kyoto, Japan*, number 2982 in Lecture Notes in Computer Science. Springer Verlag, Heidelberg, Dec. 2003.

[7] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications*, January, 1997.

Zürich, den 05.04.2004

Anhang C

Code Layout

-ribolla
---doku
------FinalDoku
------pics
------Zwischenpräsentation
---code
------exhauSearch
----------mapping.cc
----------performance
----------test
------optMapping
----------opt_mapping.cc
----------performance
----------test
------parallelServChains
----------opt_mapping_parallel.cc
---iptables
------extensions
----------libipt_PROMETHOS.c
---linux-kernel
------net
----------ipv4
--------------netfilter
--------------------ipt_PROMETHOS.c
------include
----------linux
--------------netfilter_ipv4
--------------------ipt_PROMETHOS.h

Literaturverzeichnis

[1] S. Choi and J. Turner. Configuring Sessions in Programmable Networks with Capacity Constraints. In *Proc. of IEEE ICC*, May 2003.

[2] S. Choi, T. Wolf, and J. Turner. Configuring Sessions in Programmable Networks. In *Proc. of IEEE Infocom*, Apr. 2001.

[3] A. Guindehi. COBRA Component Based Routing Architecture. Technical Report SA-2001-30, Jul. 2001.

[4] N. Jing, Y. Huang, and E. Rundensteiner. Hierarchical Optimization of Optimal Path Finding for Transportation Applications. In *IEEE Network*, Jul. 1996.

[5] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. Number 2546, Dec. 2002.

[6] L. Ruf, R. Keller, and B. Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. IEEE, Jul. 2004.

[7] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner. Towards High-performance Active Networking. Number 2982, Dec. 2003.