XML-basierte Kommunikation in Remos
Bericht zur Semesterarbeit

Nikolaos Kaintantzis

Sommersemester 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Remos steht für Resource Monitoring System. Es analysiert Netzwerke betreffend ihrer Topologie und Auslastung. Remos besteht aus zwei Komponenten: Einem Modeler, welcher die Schnittstelle für andere Applikationen anbietet, und verschiedenen Typen von Kollektoren, welche das Netzwerk analysieren. Diese beiden Komponenten kommunizieren über Sockets in einem proprietären ascii-basierten Protokoll.

Ziel dieser Arbeit ist es, das Protokoll durch ein XML-basiertes Protokoll zu ersetzen. Ebenfalls soll über HTTP-Verbindungen kommuniziert werden. Die Hauptmotivation liegt darin, anderen Applikationen leichteren Zugriff auf die Kollektoren zu bieten. Das Schreiben einer solchen Applikation ist ebenfalls Teil dieser Semesterarbeit.

Mit dem Wechsel des Protokolls ist auch eine neue Abstraktionsstufe eingeführt worden: Das Comm-Paket. Im propriäteren Protokoll sind Modeler und Kollektoren selbst für das Verpacken der zu sendenden und das Entpacken der empfangenen Information zuständig. Somit ist das Protokoll untrennbar von Modeler und Kollektor. Im Fall des Modelers muss eine Klasse, im Fall der Kollektoren müssen zusätzlich einige Hilfsklassen beim Protokollwechsel neu geschrieben werden.
Das Comm-Paket bietet für das Verpacken Builder-Klassen und für das Entpacken Reader-Klassen an. Die zu übermittelnde Information wird einem Builder über Set-Methoden übergeben. Über eine Get-Methode kann die als Kommunikations-String verpackte Information bezogen werden. Einem Reader wird ein Kommunikations-String übergeben. Durch die Get-Methoden des Readers kann die übermittelte Information wieder extrahiert werden.
Die Kapitel 2 bis 4 beschreiben, wie die XML-basierte Kommunikation umgesetzt worden ist.

Die Umstellung auf HTTP-Verbindungen hat Konsequenzen auf die Schnittstellen innerhalb der Pakete. Sie ist gemeinsam mit der Umstellung auf Servlets durchgeführt worden. Die Umstellung auf Servlets hat starke Auswirkungen auf die internen Abläufe der Kollektoren (siehe Kapitel 5). Deshalb

wurde nur der SNMP-Kollektor umgestellt. Ein Redesign sämtlicher Kollektoren hätte den Rahmen dieser Semesterarbeit gesprengt.

Die direkt kommunizierende Applikation (siehe Kapitel 6) ist als Java-Servlet geschrieben worden. Je nach Benutzer-Wunsch öffnet sie eine Socket- oder eine HTTP-Verbindung zu einem Kollektor. Die XML-Anfrage generiert sie mit Hilfe des Comm-Pakets. Die empfangene XML-Antwort wird mit einem XSL-Dokument in HTML transformiert und dem Benutzer im Browser angezeigt.

Am Ende der Arbeit wurde eine kleine Performance-Messung durchgeführt (siehe Kapitel 7). Die XML-Version skaliert der Messung nach gleich gut wie die propriätere Version. Bei Anfragen mit wenig Knoten ist das Verpacken und Entpacken der Kommunikations-Strings der Flaschenhals, was sich in eine zehn mal höhere Antwortzeit wiederspiegelt. Die Antwortzeit ist aber immer noch im Millisekunden-Bereich.

Kapitel 1

Aufgebenstellung der Semesterarbeit

Einführung

Remos ist ein REsource MOnitoring System, das einer Applikation erlaubt, Netzwerkinformation über eine netzwerkunabhängige Schnittstelle zu erfragen. Das Remossystem besteht grundsätzlich aus 2 Komponenten: dem *Modeler* und den *Kollektoren*. Der Modeler implementiert die Schnittstelle zur Applikation und nimmt die Anfragen der Applikation entgegen. Er schickt diese Anfragen an die Kollektoren weiter, welche die Information sammeln und an den Modeler zurückgeben. Diese Information ist noch in einer rohen Form und wird im Modeler in Datenstrukturen (Graph mit Knoten und Kanten) umgewandelt, welche die Applikation dann verwerten kann.

Das Protokoll zwischen Kollektor und Modeler ist proprietär (ASCII-basiert). Im Zuge von Remos Erweiterungen sollen auch neue Komponenten direkt auf die Kollektoren zugreifen können. Unsere Evaluation hat gezeigt, dass der Datenaustausch mittels HTTP als Protokol und XML zur Datenrepräsentation die geeignetste Variante darstellt. In einer Vorstudie hat Nathalie Kocher einen Prototypen entwickelt und erste Erfahrungen gesammelt. Aufgrund dieser Erfahrungen soll nun die definitive Implementation erfolgen.

Aufgabenstellung

Implementieren Sie die Kommunikation zwischen den Kollektoren und dem Remos Modeler mit HTTP und XML. Entwerfen Sie dazu eine XML Daten-

struktur, welche die auszutauschende Information darstellt. Implementieren Sie danach die Kommunikation zwischen dem Modeler und dem Kollektor mittels HTTP. Implementieren Sie die Kollektorseite einmal mit Sockets und einmal mit Servlets. Achten Sie darauf, dass die Änderungen möglist lokal sind und achten Sie auf Erweiterbarkeit.

Verifizieren Sie die Richtigkeit der neuen Kommunikation, z.B. mit dem Visualizer package. Schreiben Sie weiter auch eine Webapplikation, welche direkt mit dem Kollektor kommuniziert und die erhaltene Information darstellen kann.

Zuständiger Professor:	Prof. T. Gross	Ausgabe: 26. März 2001
Zuständiger Assistent:	Roger Karrer	

Kapitel 2

Analyse

Im ersten Teil dieser Arbeit ist die Kommunikation in Remos analysiert worden. In diesem Kapitel werden die für diese Arbeit relavanten Strukturen und Konzepte von Remos vorgestellt. Die Beschreibungen und Diagramme konzentrieren sich auf die an der Kommunikation beteiligten Pakete, Klassen und Methoden.

2.1 Interne Struktur von REMOS

Der Modeler sowie die Kollektoren befinden sich in eigenen Java-Paketen (siehe Abbildung 2.1). Der Modeler kann mit dem SNMP-Kollektor und dem Master-Kollektor kommunizieren. Hierfür öffnet er ein Socket zum Rechner und Port des jeweiligen Kollektors. Der Master-Kollektor verwaltet eine Menge von SNMP- und WAN-Kollektoren. Eintreffende Anfragen teilt er in Anfragen auf, die die einzelnen Kollektoren eigenständig beantworten können. Die Antworten sammelt er, fasst sie zusammen und leitet sie zum Modeler weiter.

Nebst der Kommunikation zwischen Modeler und einem bestimmten Kollektor, gibt es also eine kollektorinterne Kommunikation zwischen einem Master-Kollektor und den Kollektoren, die er verwaltet.
Beide Kommunikationen greifen beim SNMP-Kollektor auf dieselben Schnittstellen zu. Wird die Kommunikation zwischen Modeler und SNMP-Kollektor angepasst, so muss auch die Kommunikation zwischen Master- und SNMP-Kollektor geändert werden. Beide Kommunikationen sind somit nicht zu trennen und müssen gemeinsam auf XML umgestellt werden.

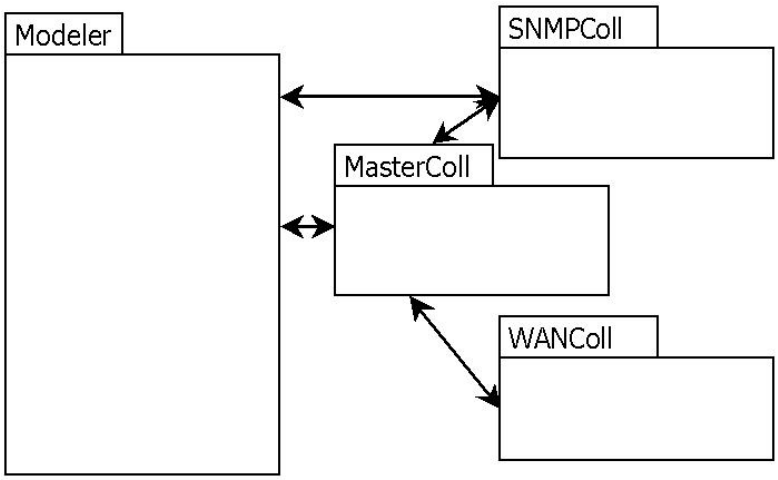

Abbildung 2.1: Kommunikation zwischen den Paketen

2.2 Kommunikations-Klassen in Remos

In den Paketen übernehmen einzelne Klassen die Kommunikation.
Im Modeler-Paket sind dies die Protokoll-Klassen (siehe Abbildung 2.2). Der `CollectorCommunicator` öffnet eine Socketverbindung zum Kollektor und generiert aus dem Socket einen `BufferedReader` und einen `PrintWriter`. Diese Ströme zum Lesen und Schreiben übergibt er seiner referenzierten `V2Protocol`-Klasse, welche die Kommunikation durchführt.

Jeder Kollektor hat eine Coll- und eine Comm-Klasse (siehe Abbildung 2.3). Die Coll-Klasse dient zum Starten des Kollektors, damit er auf einem bestimmten Port hört. Die Comm-Klasse übernimmt die Kommunikation. Im Gegensatz zum Modeler gibt es hier keine Versionierung. Am meisten vermisst man gemeinsame Superklassen. Die Vielfalt der Kollektoren ist wahrscheinlich im Stillen gewachsen.

2.3 Propriäteres Protokoll

2.3.1 Kommunikation zwischen Modeler und Kollektor

In Tabelle 2.1 ist die Kommunikation zwischen Modeler und Kollektor aufgeführt. Die Kommunikation beginnt mit einem Handshake, womit beide

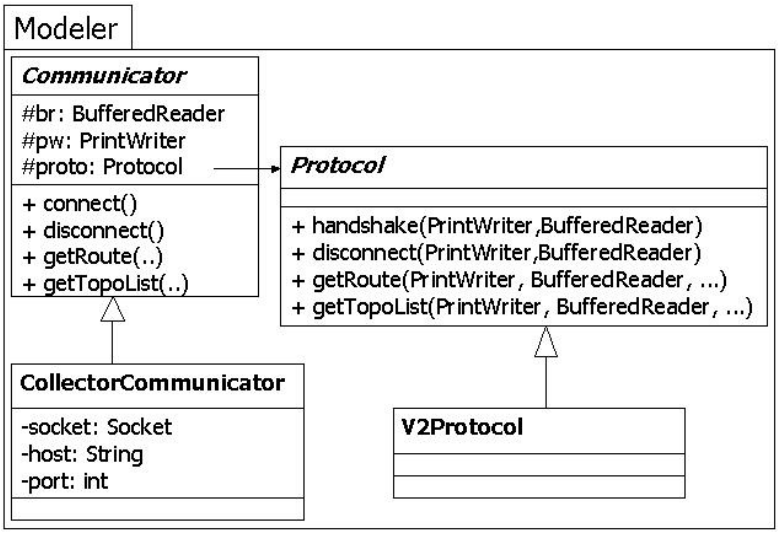

Abbildung 2.2: Kommunikationsklassen des Modelers

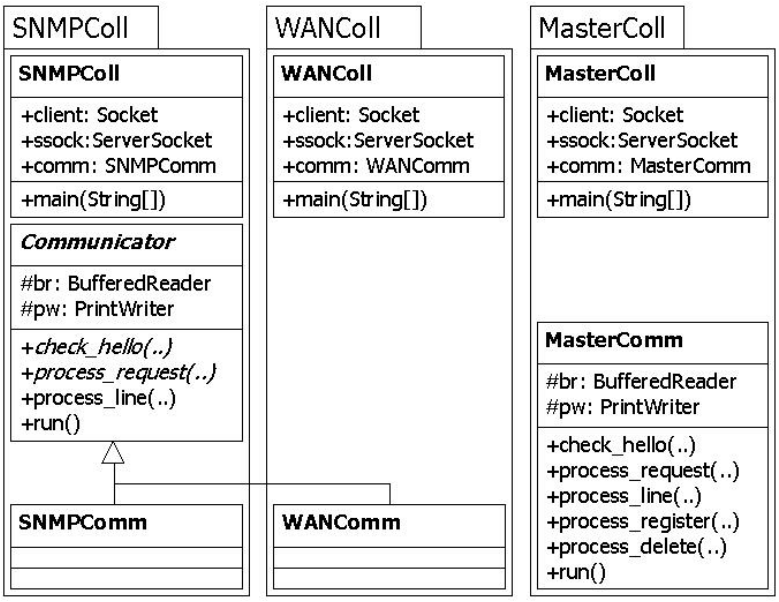

Abbildung 2.3: Kommunikationsklassen der Kollektoren

Operation	Richtung	Kommunikation
Handshake	→	HELLO REMOS MODELER <version>
	←	HELLO REMOS COLLECTOR <version>
Disconnect	→	CLOSE
Suicide	→	SUICIDE
GetRoute	→	REQUEST ROUTE <timeframe> <src-node-id> <dst-node-id>
	←	HEREIS ROUTE ENDOFROUTE ¶ ((<Node> | <Link> | <Error>) ¶)* ENDOFROUTE
Get Topo-List	→	REQUEST TOPOLOGY <timeframe> LIST <#nodes> <node-id-1> ... <node-id-n>
	←	HEREIS TOPOLOGY ENDOFTOPOLOGY ¶ ((<Node> | <Link> | <Error>) ¶)* ENDOFTOPOLOGY

Abkürzung	Kommunikation
<Node>	node <nodename><ipaddr><nodetype><speed>
<Link>	link <ipaddr[:ifnum]><ipaddr[:ifnum]> ¶ <direction> ¶ <latency> ¶ <bandwith> ¶ <max_bandwith> ¶
<Error>	error <nodename><ipaddr><nodetype><speed><error_nr>
<direction>	directed | undirected
<nodetype>	compute | switch | vswitch

Tabelle 2.1: ASCII-basiertes Protokoll zwischen Modeler und Kollektor

Operation	Richtung	Kommunikation
Register	→	REGISTER <colltype><num><net_1><mask_1> ...<net_n><mask_n>
Delete	→	DELETE <colltype><num><net_1><mask_1> ...<net_n><mask_n>

Tabelle 2.2: ASCII-basiertes Protokoll zwischen den Kollektoren

beteiligten Komponenten sicherstellen, dass sie kompatibel zueinander sind. Zuerst sendet der Modeler seinen Handshake-String samt seiner Versionsnummer und erwartet eine Antwort vom Kollektor. Beim *Disconnect* und *Suicide* wird keine Antwort erwartet. Bei einem *Route*-Request wird nebst dem Timeframe ein Start- und ein End-Knoten übergeben. Als Antwort erwartet man alle Knoten, die auf dem Weg zwischen den angegebenen Knoten liegen, sowie die Links, die diese verbinden. Ebenfalls möglich sind Error-Nodes (siehe Abschnitt 2.3.3). Beim *Topology*-Request will man wissen, wie eine gewisse Liste von Knoten im Netzwerk zueinander stehen. Als Antwort erwartet man wieder eine Menge von Knoten und Links. Die genaue Darstellung der Knoten sind wiederum der Tabelle 2.1 zu entnehmen.

2.3.2 Kommunikation zwischen den Kollektoren

Bei der Kommunikation zwischen den Kollektoren treten zwei neue Operationen auf (siehe Tabelle 2.2). Register meldet einen Kollektor beim Master-Kollektor an und Delete löscht eine getätigte Registrierung.

2.3.3 Errorhandlling

Exceptions beim Modeler werden als RemosExceptions an die Kundenapplikationen weitergegeben. Exceptions beim den Kollektoren werden unterschiedlich behandelt. Verursacht ein Knoten eine Exception (z.B. weil er dem Kollektor unbekannt ist) wird der Knoten zurückgesandt, beginnend mit dem Schlüsselwort “error” und endend mit der Fehlernummer (siehe auch Tabelle 2.1). Anderweitige Exceptions werden entweder abgefangen und ignoriert oder ebenfalls über die Socketverbindung gesendet Auch sie beginnen mit dem Schlüsselwort “error”. Der Modeler kann die beiden Fehler darin unterscheiden, dass die letzteren nicht mehr auftreten können, wenn das Senden der Antwort begonnen hat. Sieht also der Modeler das Schlüsselwort “REQUEST”, können nur noch Error-Nodes entstehen.

Fehler in der Protokollstruktur werden von den an der Kommunikation beteiligten Komponenten selbst erkannt, indem die Logik des Protokolls im Programmablauf der Empfangs-Methoden fest kodiert ist.

2.3.4 Protokolldiskrepanz

In der Analyse des Protokolls fiel auf, dass die Kollektoren *SET*s und *EDGE*s empfangen können, obwohl solche Anfragen vom Modeler nicht gesendet werden können. Auch kollektorintern werden sie nicht gesendet. Zwar ist der Begriff "Edge" für die Konfiguration der Kollektoren wichtig, doch diese Edges werden als Knoten übertragen.

2.4 Programmierstil

Am Rand sei hier bemerkt, dass der Modeler an der ETH und die Kollektoren an der CMU entwickelt wurden. Dies fällt beim Programmierstil auf.

Beim Modeler gibt es ein sehr gutes Information-Hiding und eine klare Aufgabenteilung der Klassen. Der Programmkode ist mittels Javadoc dokumentiert. Bei den Kollektoren ist alles public (oder package visible), was für klassenübergreifende Zugriffe auf Variablen genutzt wird, die teilweise static sind. Das Wissen über die Kommunikation ist sogar in Hilfsklassen kodiert, welche Kommunikations-Strings statt Objekte speichern.

Kapitel 3

Konzept

Dieses Kapitel beschäftigt sich mit der Kommunikation mittels XML. Es wird das neue XML-Protokoll vorgestellt und Varianten zum Generieren und Lesen von XML-Strings[1] diskutiert.

3.1 XML-Protokoll

In diesem Abschnitt wird gezeigt, wie das XML-Protokoll entworfen worden ist. Es gibt zwei unterschiedliche Typen von XML-Strings. Anfragen richten sich nach request.dtd und die Antworten nach reply.dtd. Diese Aufteilung ist nötig, da die Strukturen, für Anfragen und Antworten unterschiedlich sind. Die kompletten Definitionen sind im Anhang B vorzufinden.

Die Struktur der Kommunikation ist in den DTD-Dokumenten definiert. Ein erhaltener XML-String kann mit dem entsprechenden DTD-Dokument validiert werden. Validierende XML-Parser bieten diese Funktionalität an. Somit muss die Korrektheit der Protokoll-Syntax nicht mehr in Remos überprüft werden.

3.1.1 XML-Request

Jeder Request beginnt mit dem XML-typischen Header, gefolgt vom Wurzel-Tag.

[1] Für diese Arbeit ist entschieden worden, XML-Strings zu senden. Diese Entscheidung wird in Abschnitt 8.5 diskutiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE request SYSTEM "http://www.cs.inf.ethz.ch/~karrer/
    remos/request.dtd">
<request type="..." version="3.0">
...
</request>
```

Im neuen Protokoll wird auf ein explizites Handshake verzichtet. Die Kommunikation kann also direkt mit einer Anfrage beginnen. Die Kompatibilität wird durch das Attribut "version" sichergestellt, welches im Request-Tag vorhanden sein und laut DTD den Wert 3.0 haben muss. Das Attribut "type" kann den Wert route, topology, close, suicide, set, edge, register oder unregister annehmen. (Unregister wird neu anstatt delete verwendet, dient also zum Abmelden eines Kollektors.)

Innerhalb des `<request></request>`-Bereichs liegt die eigentliche Anfrage. Gültige Einträge sind Timeframe-, Node-ID- und Kollektoren-Tags.

Die erste Zeile einer Anfrage besteht immer aus einem Timeframe-Eintrag.

```
<timeframe>...</timeframe>
```

Im ASCII-Protokoll hatten lediglich die *Route*- und *Topology*-Anfragen ein Timeframe (siehe Tabelle 2.1). Fürs neue Protokoll ist entschieden worden, dass jede Anfrage ein Timeframe haben muss. Durch diese Entscheidung wird sichergestellt, dass jede Anfrage einen Inhalt hat. Das bedeutet, dass die letzten Zeichen einer Anfrage das schliessende Request-Tag (`</request>`) ist. Dies ist für die socketbasierte Kommunikation wichtig, in welcher das Ende einer Anfrage erkannt werden muss.

Dieser Zeile folgt eine Menge von Node-Ids oder ein Kollektor-Eintrag. Eine Node-Id kann fakultativ einen Typ "src" für Quelle oder "dst" für Ziel haben. Dies wird wie folgt dargestellt:

```
<nodeid type="src">rif1.ethz.ch</nodeid>
```

Ein Kollektor-Eintrag wird für *Register*- und *Unregister*-Anfragen verwendet. Nähere Angaben sind dem DTD im Anhang B zu entnehmen.

3.1.2 XML-Reply

Ähnlich zur Anfrage beginnt die Antwort mit dem XML-typischen Header, gefolgt vom Wurzel-Tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE request SYSTEM "http://www.cs.inf.ethz.ch/~karrer/
    remos/reply.dtd">
<reply type="..." version="3.0">
...
</reply>
```

Auch bei der Antwort stellt das DTD die Kompatibilität sicher, indem es vom Versions-Attribut den Wert 3.0 verlangt. Das Attribut "type" kann den Wert route oder topology annehmen. Innerhalb des `<reply></reply>`-Bereichs können nur nodes, links oder errors auftreten. Ein Knoten besteht aus seinem Namen, Typ, IP-Adresse und Speed, ein Link aus seiner Latenz, Bandbreite, maximaler Bandbreite, den IP-Adressen seiner Knoten und die Richtung. Ein Error besteht aus einem Knoten und einer Fehlernummer (siehe auch ASCII-Protokoll in Tabelle 2.1).

Jede Komponente, die einen bestimmten Wertebereich haben muss, ist als Attribut zu einem Tag definiert. So ist zum Beispiel der Typ eines Knotes, welcher nur drei vorgegebene Werte annehmen kann, als Attribute in Node-Tag definiert. Im DTD sind die Wertebereiche der Attribute festgehalten.
Komponenten, die einen beliebigen Wert annehmen können, sind als eigene Tags definiert. Zum Beispiel kann der Name eines Knotens eine beliebige Zeichenkette sein.
An dieser Stelle ein kleines Beispiel.

```
<reply type="route" version="3.0">
    <node type="compute">
        <name>rif1.ethz.ch</name>
        <ip>129.132.179.11</ip>
        <speed>-1.0</speed>
    </node>
    <node type="vswitch">
        <name>C_Vrif1_1</name>
        <ip>10.132.0.1</ip>
        <speed>1250000.0</speed>
    </node>
    <link direction="undirected">
        <ip typ="src">129.132.179.11</ip>
        <ip typ="dst">10.132.0.1</ip>
        <latency>1.0</latency>
        <bw>-1.0</bw>
        <maxbw>1250000.0</maxbw>
```

```
    </link>
</reply>
```

3.2 Varianten der XML-Kommunikation

Die XML-Strings müssen sowohl auf Seiten des Modelers wie auf Seiten der Kollektoren generiert werden, um sie zur Gegenseite zu senden. Ein empfangener XML-String muss nach relevanter Information durchsucht werden können. Wo und wie sollen diese XML-Strings generiert werden? Hierzu stehen mehrere Varianten zur Diskussion.

3.2.1 XML-String generieren

Zeilenweise generieren

Bei dieser Variante wird der XML-String zeilenweise generiert (siehe Abbildung 3.1). Wenn die zu übermittelnde Information im Programmfluss vorliegt, wird sie in eine XML-Zeile verpackt und gesendet. Dieses Vorgehen wäre ähnlich zum vorhandenen ASCII-Protokoll. Die Stellen, wo Informationen übermittelt werden, wären die selben.

Am Beispiel der Anfrage

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE request SYSTEM "http://www.cs.inf.ethz.ch/~karrer/
       remos/request.dtd">
3: <request type="route" version="3.0">
4:     <timeframe>-1.0</timeframe>
5:     <nodeid type="src">rif1.ethz.ch</nodeid>
6:     <nodeid type="dst">raf30.ethz.ch</nodeid>
7: </request>
```

würden die ersten zwei Zeilen zu Beginn des Sende-Prozesses generiert und gesendet, da sie überall identisch sind. Die Zeilen 3 bis 6 würden in der Methode, die für *Route*-Anfragen zuständig ist, versendet und die letzte Zeile kurz vor Ende des Sende-Prozesses.

Die Nachtteile dieser Kommunikation liegen auf der Hand. In den einzelnen Paketen muss der Entwickler selbst besorgt sein, dass die XML-Syntax eingehalten wird und z.B. keine Anführungszeichen vergessen werden. Dies muss

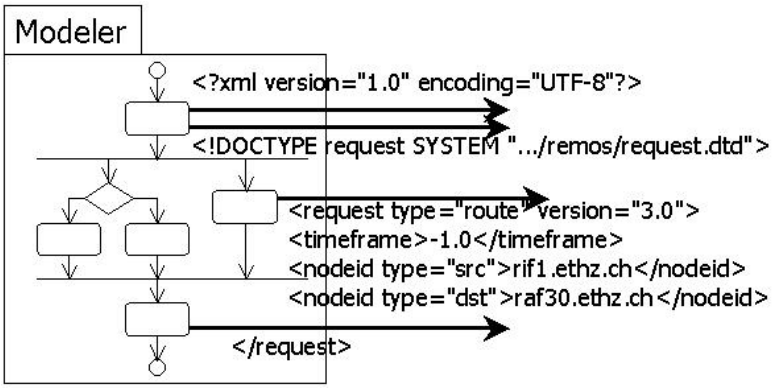

Abbildung 3.1: XML-String zeilenweise generieren

immer wieder an verschiedenen Stellen gemacht werden. Zudem ist die XML-Struktur fest im Programmablauf verdrahtet. D.h. wenn jemand eine anderes XML-Format wünscht, muss an vielen Stellen Kode angepasst werden. Beim jeden Senden könnte eine Exception auftreten. Das Exception-Handling wäre über viele Stellen im Kode verteilt.

DOM-Baum serialisieren

Bei der Variante DOM-Baum (siehe Abbildung 3.2) wird die Information in Knoten durch sequentielles Einfügen gesammelt. Auf der Baumstruktur können bereits eingefügte Knoten nachträglich manipuliert oder umgehängt werden. Am Ende kann dieser Baum serialisiert und in einen XML-String umgewandelt werden.

Bei dieser Variante muss man sich nicht mehr um die XML-Syntax selber kümmern. Der XML-String wird automatisch generiert. Dennoch bleibt ein Nachteil bestehen: Die XML-Struktur des Protokolls ist immer noch fest im Programmablauf verdrahtet. Wenn sich das XML-Format ändert, muss sich auch der DOM-Baum ändern. Dieser wird im Modeler und im Kollektor generiert, also müssen auch diese angepasst werden.

XML-Builder-Hilfsklassen

Die Idee hinter den Hilfsklassen – je eine für Anfragen und eine für Antworten – ist, dem Modeler und den Kollektoren zu verbergen, wie das XML-Protokoll strukturiert ist, indem XML-unabhängige Methoden angeboten werden. Intern kann eine beliebige Datenstruktur zur Verwaltung aufgebaut werden.

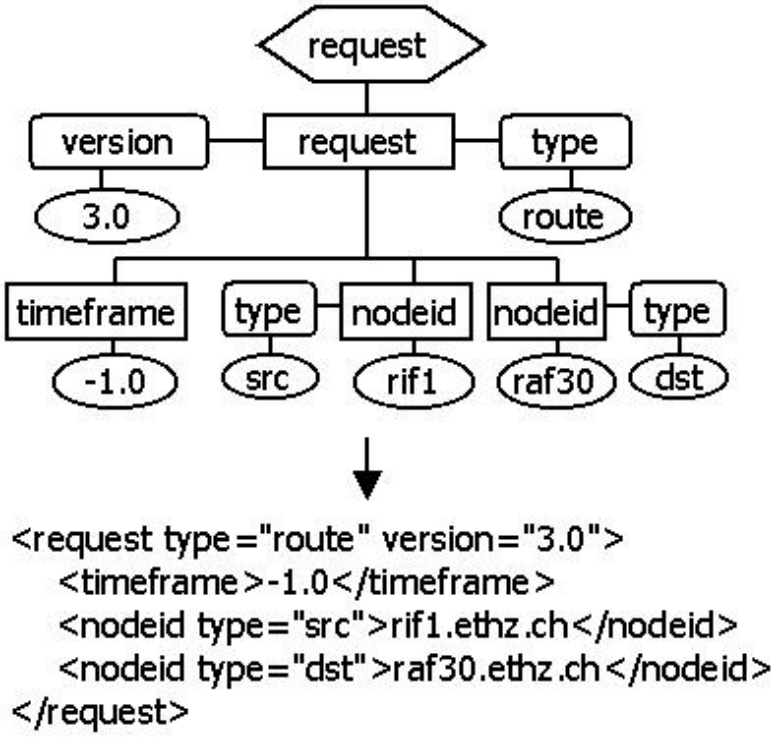

Abbildung 3.2: DOM-Baum serialisieren

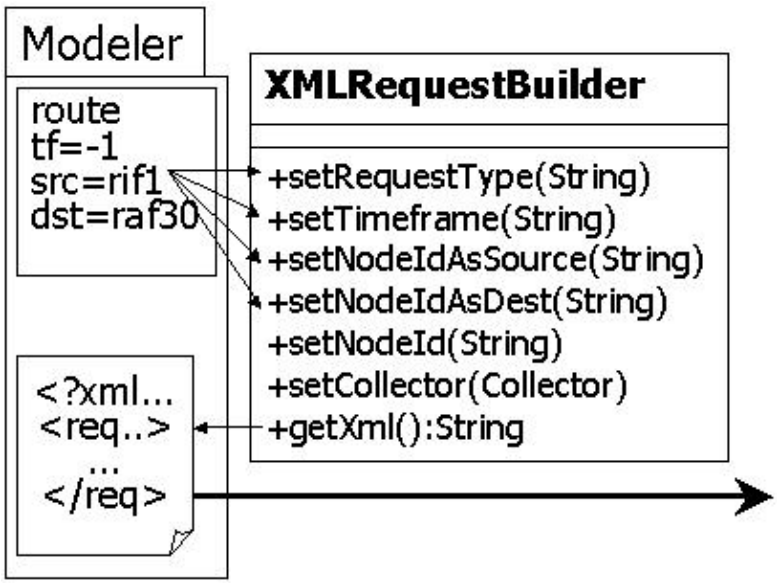

Abbildung 3.3: Funktionsweise des Request-Builders

Der DOM-Baum drängt sich auf, da er ein W3C-Standard ist und viele Parser-Hersteller DOM anbieten.

In dieser Arbeit ist die Lösung mit den XML-Builder-Klassen implementiert worden, da sie Protokoll-Einzelheiten verbergen. Die Verwendung eines neuen Protokolls oder eines neuen Parsers lassen Modeler und Kollektoren unberührt, was bei den anderen Lösungen nicht der Fall ist.

3.2.2 XML-String lesen

ASCII-Strom zeilenweise abarbeiten

Ähnlich zur proprietären Version soll Zeile für Zeile eingelesen werden und diese nach Schlüsselwörtern durchsucht und die Daten extrahiert werden. Dieses Vorgehen hätte den Nachteil, dass sich die lesende Applikation um die XML-Syntax kümmern müsste, damit sie weiss, wann Schlüsselwörter oder Daten kommen. Ebenfalls wäre die XML-Struktur fest im Programm einkodiert.

DOM-Baum instanzieren

Mit dem XML-String wird ein DOM-Baum erstellt. Dieser wird mit Hilfe eines Parsers traversiert und die Informationen nach Bedarf extrahiert. Die Nachteile bei dieser Variante liegen darin, dass die Baum-Form (also indirekt die XML-Struktur) bekannt sein muss, damit die Informationen gefunden werden. Ebenfalls führt ein Wechsel des Parser-Herstellers, mit welchem der Baum traversiert wird, zu Änderungen in den beteiligten Klassen, da jeder Hersteller ein anderes API zur Verfügung stellt. Der Aufbau eines DOM-Baumes empfiehlt sich erst, wenn Informationen mehrfach oder unabhängig von der Reihenfolge im XML-String gelesen resp. manipuliert werden. Da dies hier nicht der Fall ist, wäre der Aufbau eines Baums ein zu grosser Overhead.

SAX-Events abfangen

Beim Parsen des Dokumentes mit einem SAX-Paser werden Events wie "start-Element" oder "endElement" ausgelöst, auf welche im Modeler und den Kollektoren reagiert werden kann. So wurden die Daten im Prototyp [6] vor dieser Arbeit extrahiert. Diese Variante erfordert Wissen über die Namen der verwendeten Tags, empfiehlt sich aber, wenn die Information nur einmal gelesen werden muss. Aber auch hier ist ein Hersteller-Wechsel mit Änderungen an den Kollektoren und dem Modeler verbunden.

XML-Reader-Hilfsklassen

Diese Hilfsklassen kapseln die Struktur der XML-Strings ab. Sie bieten Methoden zum Zugriff auf kommunikationsspezifische Objekte, die im XML-String verpackt sind. Sie dienen als Schnittstelle zwischen Applikation und XML-Struktur. Da die Hilfsklassen auf die Reihenfolge und die Anzahl der Methoden-Zugriffe keinen Einfluss haben, muss eine Datenstruktur zur Verwaltung der im XML-String vorhandenen Information aufgebaut werden. Der DOM-Baum drängt sich auch hier auf.

Analog zur XML-Generierung sind die Hilfsklassen implementiert worden, da sie eine höhere Abstraktion bieten. Parser-Wechsel und Änderungen im Protokoll haben somit keinen Einfluss auf Modeler und Kollektoren.

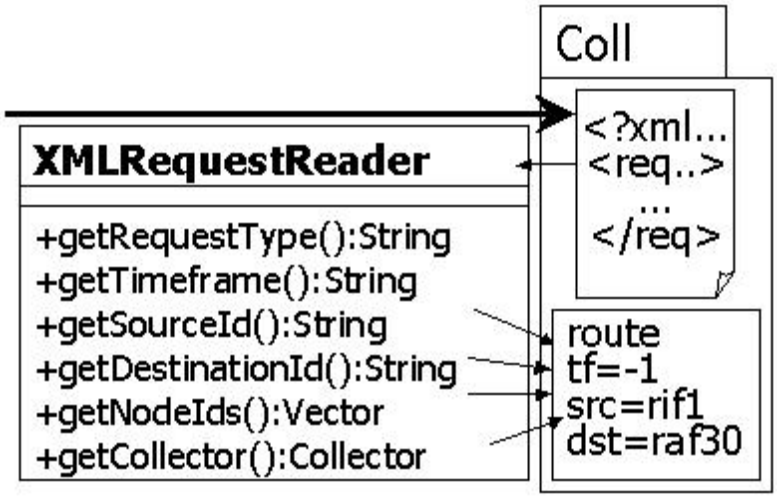

Abbildung 3.4: Funktionsweise des Request-Readers

3.3 Error Handling

Die vorhandene Fehlerbehandlung ist übernommen worden. D.h. Exceptions im Modeler werden als RemosExceptions zur Kundenapplikation weitergereicht. Bei den Kollektoren werden durch Knoten verursachte Fehler als Error-Nodes innerhalb der XML-Struktur übergeben. Der Modeler kann entscheiden, was es mit diesen Knoten anstellen will (z.B: ignorieren). Andere Fehler in den Kollektoren werden im alten Stile (Zeile beginnend mit dem Schlüsselwort “error”) über die Socketverbindung rausgeschrieben. Beim Empfänger tritt eine XML-Execption auf (da kein XML-Dokument empfangen worden ist), mit der vorherigen Fehlermeldung als Inhalt auf. Der Grund wieso darauf verzichtet worden ist, Fehler in einem XML-Format zu übertragen. liegt darin, dass beim Instanzieren eines Builders (also beim Erstellen des XML-Strings) eine Exception auftreten kann. Also könnte das Erzeugen einer Fehlermeldung in XML, wieder zu einer Fehlermeldung führen. Es wäre also unter Umständen gar nicht möglich, einen Fehler als XML-String zu generieren. Einen fest kodierten String zu versenden, hätte den Nachteil, dass keine Informationen über die Fehlerart versendet werden können. Einen XML-String im Modeler resp. in den Kollektoren zusammenzusetzen, hätten den Nachteil, das Protokoll-Logik im Modeler oder in den Kollektoren eingebaut werden muss.

Fehler im Protokoll werden durch eine Validierung des Dokuments erkannt. Beim Instanzieren eines Readers wird der XML-String mit dem referenzierten

DTD überprüft. Ist das Dokument nicht gültig, wird eine Exception geworfen, welche nach den oben beschriebenen Mechanismus behandelt wird. Die referenzierten request.dtd und reply.dtd sind unter `http://www.cs.inf.ethz.ch/~karrer/remos/` veröffentlicht worden.

Kapitel 4

Implementation

Dieses Kapitel befasst sich mit dem Einbinden der in den vorherigen Kapiteln beschriebenen Konzepte in Remos. Neu eingeführt worden, ist das Comm-Paket, welches die Protokoll- (also XML-) Strukturen abkapselt. Es wird erklärt, wie das Comm-Paket funktioniert und welche freien Java-Bibliotheken verwendet wurden. Des weiteren wird beschreiben, welche bestehenden Klassen angepasst werden mussten, damit die Kommunikation über das Comm-Paket abgewickelt werden kann.
Die Funktionstüchtigkeit ist mit Hilfe von Tests überprüft worden. Dabei haben sich in dieser Arbeit Komponenten-Tests als sehr hilfreich erwiesen.

4.1 Comm-Paket

Das Comm-Paket bietet Unterstützung bei der Kommunikation. Die XML-Hilfsklassen, die im Kapitel zuvor behandelt worden sind, befinden sich im neuen Comm-Paket. Daneben existieren Objekte wie Link, Node und Collector, die übertragen werden können (siehe Abbildung 4.1). Zwischen Modeler und Kollektor hat es vor dieser Arbeit keine gemeinsame Datenstrukturen gegeben.

Am Beispiel einer Route-Anfrage mit Timeframe -1.0 von rif1 nach raf30, beides Rechner an der ETH Zürich, soll der Aufbau und die Übertragung der Anfrage erklärt werden. Dieses Beispiel wird in der Abbildung 4.2 illustriert.

Der Modeler instanziert einen XMLRequestBuilder aus dem Comm-Paket und fügt über Set-Methoden die zu übermittelnde Information ein. Der XML-RequestBuilder baut intern einen DOM-Baum auf, an den er Blätter und

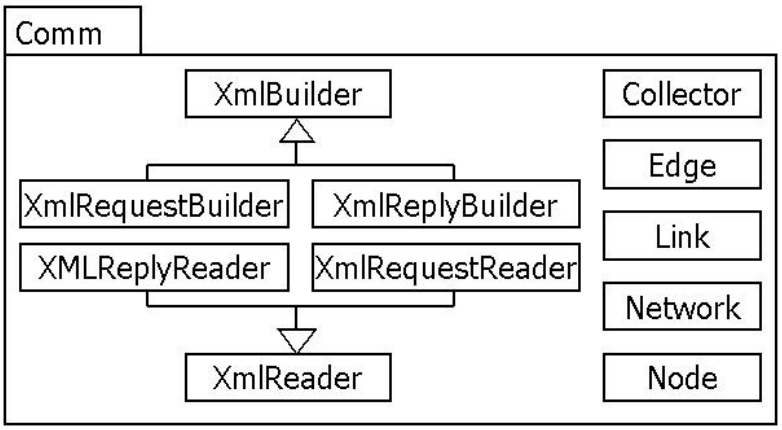

Abbildung 4.1: Klassen innerhalb des Comm-Paketes

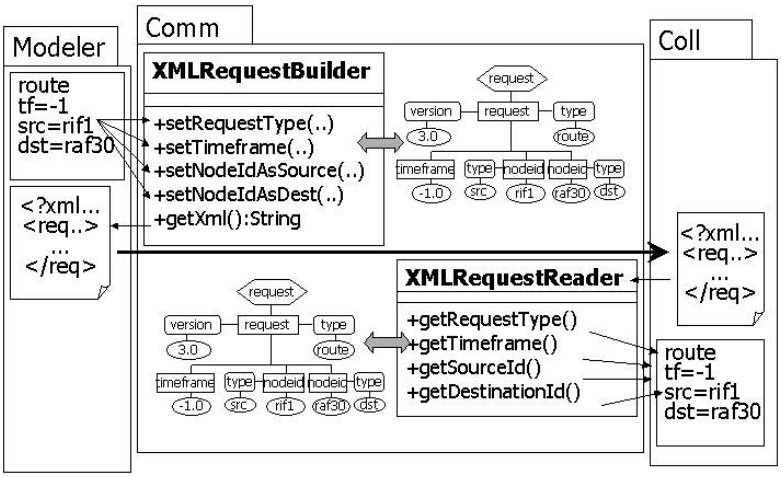

Abbildung 4.2: Übermitteln eines Requests

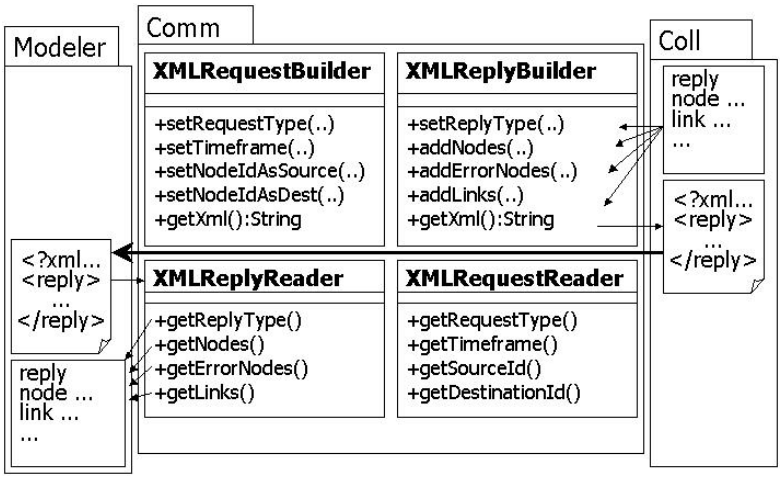

Abbildung 4.3: Übermitteln eines Replys

Knoten einfügt resp. manipuliert. Sobald der Modeler alle zu übermittelnden Informationen eingefügt hat, verlangt er via GetXml-Methode einen String, welcher die eingefügten Informationen als XML-Struktur beinhaltet. Diesen schickt er zum Kollektor.

Der Kollektor instanziert mit dem erhaltenen String einen XMLRequestReader. Dieser verifiziert das Dokument auf seine Richtigkeit und generiert sich intern einen DOM-Baum. Der DOM-Baum wird benutzt, um dem Kollektor die gewünschte Informationen zu liefern. Der Kollektor benutzt hierfür die Get-Methoden des XMLRequestReaders.

Analog funktioniert das Übermitteln einer Antwort (siehe Abbildung 4.3). Der Kollektor füllt einen XMLReplyBuilder mit Informationen, die er übermitteln möchte. Sobald dies geschehen ist, bezieht er den XML-String und schickt ihn zum Modeler. Der Modeler instanziert einen XMLReplyReader und extrahiert mit dessen Hilfe die Informationen.

4.2 Änderungen in Remos

Nebst dem neuen Comm-Paket gibt es Änderungen in anderen Paketen. Im Modeler (siehe Abbildung 2.2) gibt es eine neue Ableitung der Protokoll-Klasse – das `V3Protocol` – welches nun von `CollectorCommunicator` an Stelle vom `V2Protocol` referenziert wird.

Bei den Kollektoren (siehe Abbildung 2.3) sind die Klassen Communicator,

SNMPComm, WANComm und MasterComm neu geschrieben worden, da sie wie im Abschnit 2.2 beschrieben, für die Kommunikation zuständig sind. CollInfo und Edge, weitere Hilfsklassen, die versteckt mit der Kommunikation zu tun haben, sind ebenfalls angepasst worden. Es wurde darauf verzichtet die Kollektoren neu zu organisieren resp. zu entwerfen, da es den Rahmen dieser Arbeit gesprengt hätte.

4.3 XML-Parser

Im Comm-Paket wird der DOM-Parser von XERCES verwendet. (XERCES ist ein Unterprojekt von APACHEs XML-Projekt.) Der Hauptgrund für die Verwendung von XERCES liegt darin, dass hier gute Dokumentation (siehe Literaturverzeichnis) und Anwendungsbeispiele vorhanden sind. IBMs XML-Parser basiert ebenfalls auf XERCES.

4.4 Tests

4.4.1 Komponenten-Tests

Aufgrund der Grösse und Komplexität vom Remos haben sich die durchgeführeten Komponenten-Tests als sehr wichtig erwiesen. Vor der Einbindung des Comm-Paket in die Kommunikation, sind die einzelnen Klassen auf ihre Funktionalität getestet worden. Diese Testklassen befinden sich (im Zustand des letzten Tests) ebenfalls im Comm-Paket.

TestXmlReplyBuilder testet, nachdem alle Methoden des Builders benutzt wurden, ob dieser einen korrekten XML-String zurückgibt.

TestXmlReplyReader testet, ob jede Information, die sich in einem XML-String befindet, korrekt extrahiert wird. Beim Starten kann man via Parameter bestimmen, ob ein Route- oder Topology-Reply benutzt werden soll.

TestXmlRequestBuilder führt mehrere Test hintereinander durch. Es werden Methoden des Builders benutzt und der XML-String auf seine Korrektheit überprüft.

TestXmlRequestReader testet, ob jede Information, die sich in einem XML-String befindet, korrekt extrahiert wird. Beim Starten kann man

via Parameter bestimmen, ob ein Topology-, Route-, Suicide-, Close-, Register-Request benutzt werden soll.

4.4.2 Gesammt-Tests

Nebst dem Benutzen der vorhandenen Applikationen wurde eine Testklasse (TestProtocol im Modeler-Paket) geschrieben, die an den Modeler Anfragen stellt und dessen Antwort anzeigt. Die Resultate können auf Plausibilität oder auf Gegenvergleich mit der Remos-Version, die das proprietäre Protokoll benutzt, verifiziert werden.

Kapitel 5

HTTP-Servlets

5.1 Motivation

In den letzten Kapiteln ist gezeigt worden, wie die Kommunikation auf XML umgestellt worden ist. Ein weiterer Teil der Semesterarbeit ist es, die Kommunikation auf HTTP-Verbindungen und die Kollektoren auf Servlets umzustellen. Es bietet sich an dies in einem Schritt zu tun und HTTP-Servlets zu verwenden.

Der Vorteil von Servlets liegt darin, dass sie nicht manuell gestartet werden müssen. Sobald eine Verbindung zu einem Servlet verlangt wird, sorgt die Java-Servlet-Umgebung, dass das Servlet gestartet wird, falls es nicht schon läuft. Allerdings muss daran gedacht werden, die Java-Servlet-Umgebung vorgängig zu starten.
HTTP-Verbindungen erlauben einem interessierten Entwickler, die URLs von Kollektoren anzugeben, statt zuvor Rechnernamen und Sockets. Dies mag zwar kein überwiegender Vorteil sein, doch die Kombination von Servlets und HTTP-Verbindungen erlaubt eine Anfrage direkt in die URL des Browsers einzugeben. D.h. für schnelle Tests genügt der Browser als Applikation, denn die Browser stellen vermehrt auch XML dar.

5.2 Konzept und Implementierung

Die Umstellung auf HTTP-Servlets bringen grundlegende Veränderungen in den internen Schnittstellen und Strukturen, weshalb entschieden worden ist, die Umstellung nur für Modeler und den SNMP-Kollektor durchzuführen.

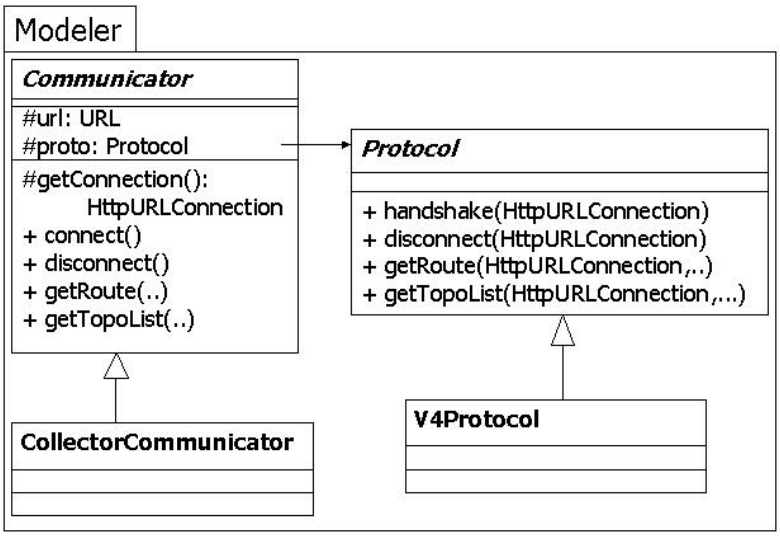

Abbildung 5.1: Neue Struktur des Modelers wegen HTTP

Die internen Schnittstellen im Modeler (siehe Abbildung 2.2) und in den Kollektoren (siehe Abbildung 2.3) sehen vor, dass aus der Verbindung ein BufferedReader und ein PrintWriter generiert werden und den beteiligten Kommunikationsklassen und -Methoden übergeben werden. Zwar lassen sich auch aus HTTP-Verbindungen BufferedReader und PrintWriter generieren. Da aber bei einer HTTP-Verbindung zuerst eine Anfrage und dann die Antwort folgen muss, wird der PrintWriter invalidiert, sobald der BufferedReader instanziert worden ist. Dies bedeutet eine Änderung der Schnittstellen innerhalb der Pakete. Beim Modeler hätte dies zur Folge, dass alle älteren Protokoll-Klassen (`V1Protocol`, `V2Protocol` und `V3Protocol`) nicht mehr funktionieren, da auf die Input-Parameter BufferedReader und PrintWriter in den Methoden der abstrakte Klasse `Protocol` verzichtet werden muss.

Deshalb wurden alle Klassen in ein neues Projekt kopiert und mit der Implementierung einer Prototypversion begonnen. In der Prototypversion wurden so wenig Änderungen der Schnittstellen wie nötig durchgeführt. Abbildung 5.1 zeigt die neuen Strukturen beim Modeler. In der Klasse `Protocol` wurden die Methoden handshake und disconnect beibehalten, auch wenn sie im `V4Protocol` nicht gebraucht werden und deshalb leer sind.
Neu speichert der `Communicator` eine URL. Die Methode getConnection liefert aus der URL eine neue Verbindung, denn eine Verbindung kann genau einmal benutzt werden. Diese Verbindung wird dem Protokoll übergeben.

Im SNMP-Kollektor (siehe Abbildung 5.2) hat sich zusätzlich die Art und

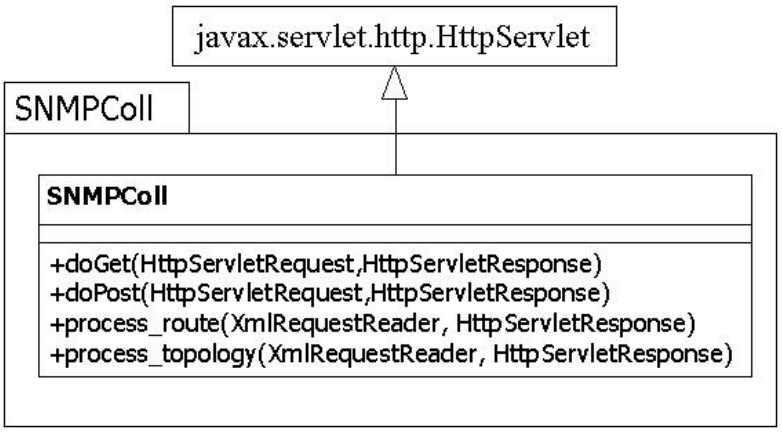

Abbildung 5.2: HTTP-Servlet-Struktur des SNMP-Kollektors

Weise, wie er gestartet wird, verändert. Neu wird er erst bei Bedarf durch die Java-Servlet-Umgebung gestartet. Die Klasse `SNMPColl` muss deshalb von `HttpServlet` abgeleitet werden. HttpServlet-Klassen müssen aber auch die Kommunikation durchführen. Aus diesem Grund sind die Klassen `SNMPColl`, `Communicator` und `SNMPComm` in einer neuen `SNMPColl`-Klasse zusammengefasst. Die doGet- resp doPost-Methode wird automatisch aufgerufen wenn ein Get- resp. ein Post-Request an die URL des Kollektors gestellt wird. In HttpServletRequest befinden sich die Parameter zum Request. Der XML-String ist dem Parameter "xml" zugewiesen. Die XML-Antwort kann in den HttpServletResponse geschrieben werden.

5.3 Folgen

5.3.1 Folgen für die Applikation

Die Folgen für den Modeler sind lediglich die Änderung der Schnittstellen (Das Verwenden vom HttpURLConnectios statt BufferedReader und PrintWriter und die neue Methode getConnection, welche HttpURLConnectios zur weiteren Verwendung erstellt). Für die Kollektoren sind die Folgen schwerwiegender. Am meisten schlägt die Änderung zu Servlets ins Gewicht. Die Kollektoren sind keine Threads mehr und müssen neu strukturiert werden. Servlets haben zum Beispiel andere Lebenszyklen und werden anders instanziert als Threads.

Das An- und Abmelden von Kollektoren beim MasterKollektor muss demzu-

folge auch neu entworfen werden. Bei der Socket/Thread-Lösung sind zuerst der Master-Kollektor gestartet worden und dann die andern Kollektoren. In deren Konfigurationsfiles steht, wo sich der Master-Kollektor befindet und auf welchen Port er hört. Da neuerdings die Servlets nicht durch einen Administrator gestartet werden, ist dieses Vorgehen nicht mehr möglich. Eine denkbare Lösung wäre das Schreiben eines Administrations-Formulars für den Master-Kollektor, in welches man die URLs von Kollektoren angibt oder wieder entfernt.

In der socketbasierten Lösung gibt es auf Kollektorenseite einen "Reaper", welcher nach dem Verbleiben von Kollektoren schaut. Alle Verbindungen zum Reaper müssten entfernt werden.

5.3.2 Folgen für die Kommunikation

Das Kommunikationsprotokoll ist identisch geblieben. Das Comm-Paket hat sich nicht verändert. Durch die Delegierung der Kommunikationslogik zum Comm-Paket ist eine getrennte Versionierung von XML-Kommunikation und Protokoll-Klassen möglich. Dies ist auch hier geschehen, das Protokoll hat die Version 3.0, die Protokoll-Klasse im Modeler hat die Version 4.

Der *Suicide*-Request muss bei der Serlvet-Implementation ignoriert werden. Beendet ein Servlet sich selbst, so wird die ganze Java-Servlet-Umgebung mitgezogen.

Kapitel 6

Test-Applikation

6.1 Motivation

Die Offenlegung der DTDs des Kommunikations-Protokolls erlaubt das Entwickeln neuer Applikationen, die direkt auf die Kollektoren (ohne Umweg über Modeler) zugreifen. Dadurch lassen sich zum Beispiel Überwachungs-Werkzeuge für einzelne Netzwerksegmente, auf denen Kollektoren laufen, schreiben.

Im Rahmen dieser Semesterarbeit soll gezeigt werden, dass das Schreiben solcher Applikationen einfach möglich ist. Es ist eine Applikation geschrieben worden, die an den Kollektor eine Anfrage stellt und dessen Antwort in eine einfach lesbare Form anzeigt.

6.2 Konzept

Statt die XML-Anfrage selbst zu formulieren, wird der XmlRequestBuilder aus dem Comm-Paket benutzt (siehe Abbildung 6.1). Der String wird an einen Kollektor gesendet, der zuvor vom Benutzer ausgewählt wurde. Die XML-Antwort des Kollektors wird nicht im Programm analysiert, sondern mit Hilfe eines XSL-Transformators und einem XSL-Dokument in eine HTML-Antwort umgewandelt. Damit das HTML in einem Browser angezeigt werden kann, ist die Applikation als Servlet geschrieben worden.

Durch das Verwenden des Comm-Pakets kann also Arbeit beim Entwickeln neuer Applikationen gespart werden. Es muss nicht mehr analysiert werden, wie ein XML-String auszusehen hat. Das Comm-Paket kann auch zum Verarbeiten der Antwort verwendet werden. Für diese Test-Applikation wurde

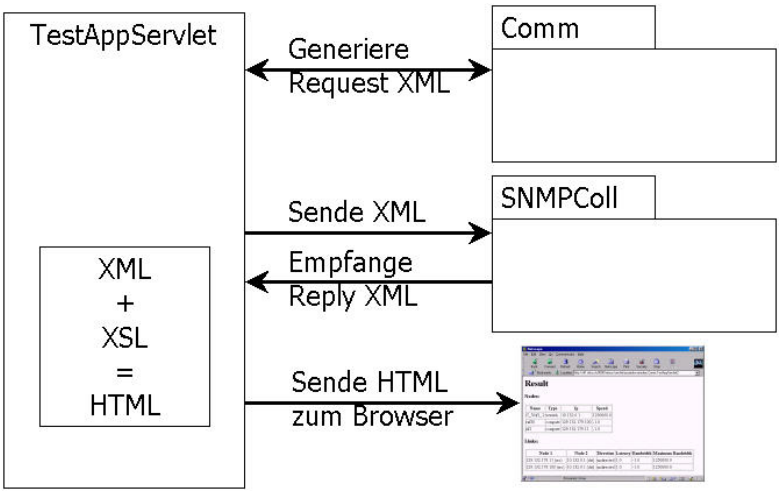

Abbildung 6.1: Programm-Ablauf der Test-Applikation

aber darauf verzichtet. Da der gewünschte Output HTML ist, läst sich durch die Verwendung von XSL-Transformationen weitere Flexibilität gewinnen. Das Layout ist im XSL-Dokument festgelegt und kann dort geändert werden. Die Test-Applikation ist also sehr schlank. Sie kümmert sich weder um die XML-Syntax noch um das Layout der Antwort.

6.3 Implementation

Die Applikation besteht aus 2 Servlets (TestAppServlet1 und TestAppServlet2), die sich im Comm-Paket befinden und der Datei `index.html` als Eintrittspunkt in die Applikation.

In `index.html` (siehe Abbildung 6.2) muss sich der Benutzer entscheiden, welche Protokoll-Version er verwenden möchte und welchen Typ seine Anfrage hat.

Diese Angaben werden zum ersten Servlet übermittelt, welches ein angepasstes Formular präsentiert. Wenn der Benutzer das V4-Protokoll verwenden möchte (siehe Abbildung 6.3), muss er die URL des Kollektors angeben. Möchte er das V3-Protokoll verwenden (siehe Abbildung 6.4), muss er Rechnername und Port angeben.

Im zweiten Teil des Formulars müssen bei einer *Route*-Anfrage (siehe Abbildung 6.3) Start- und End-Knoten, bei einer *Topology*-Anfrage (siehe Abbildung 6.4) alle Knoten der Anfrage angegeben werden.

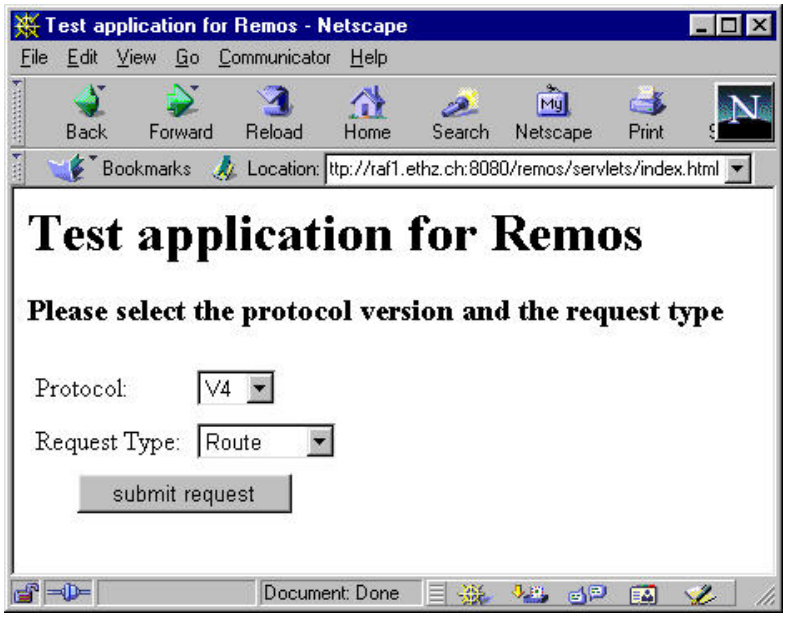

Abbildung 6.2: Test-Applikation: Protokoll und Request-Wahl

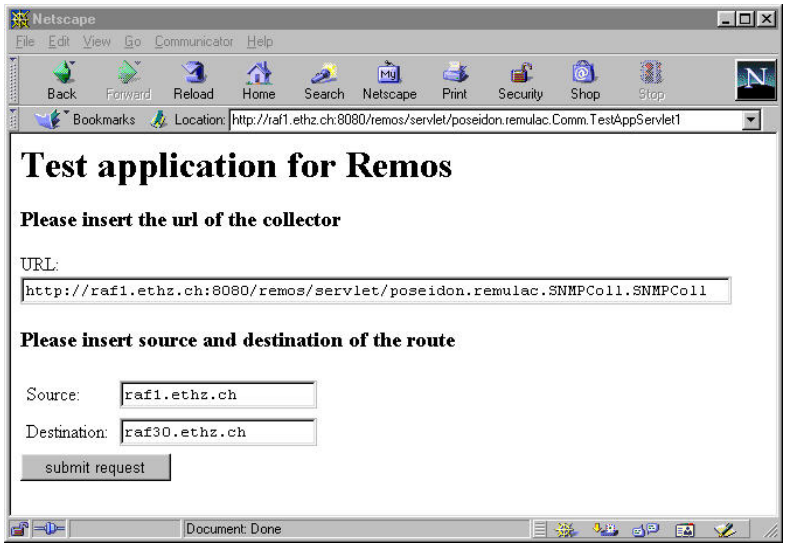

Abbildung 6.3: Test-Applikation: V4-Protokoll und Route-Request

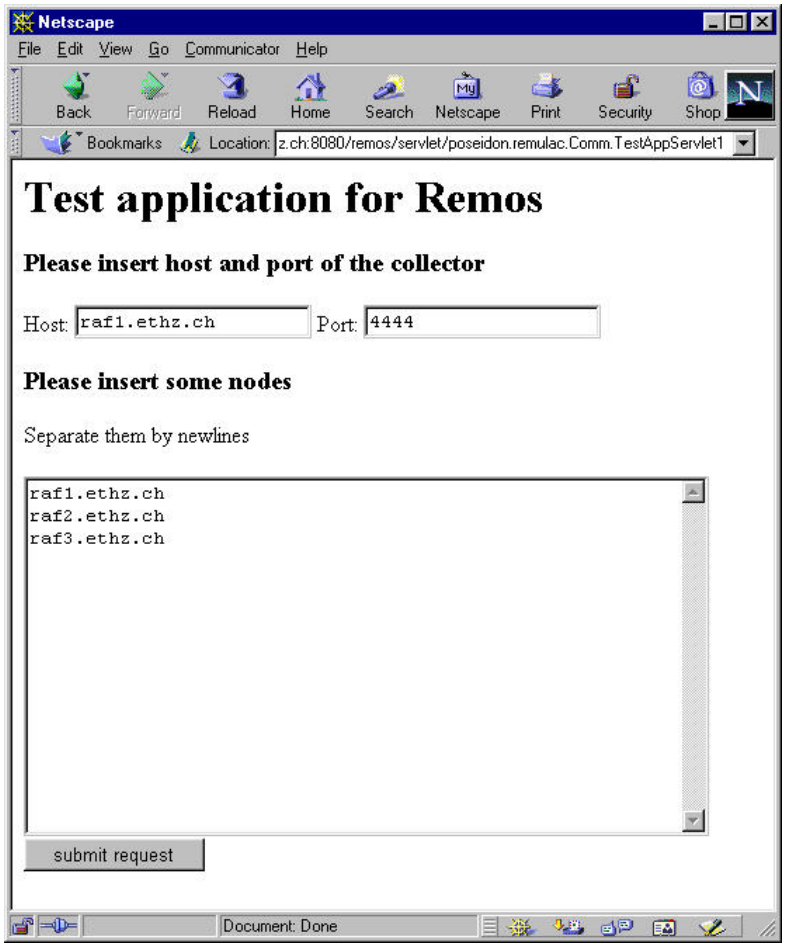

Abbildung 6.4: Test-Applikation: V3-Protokoll und Topology-Request

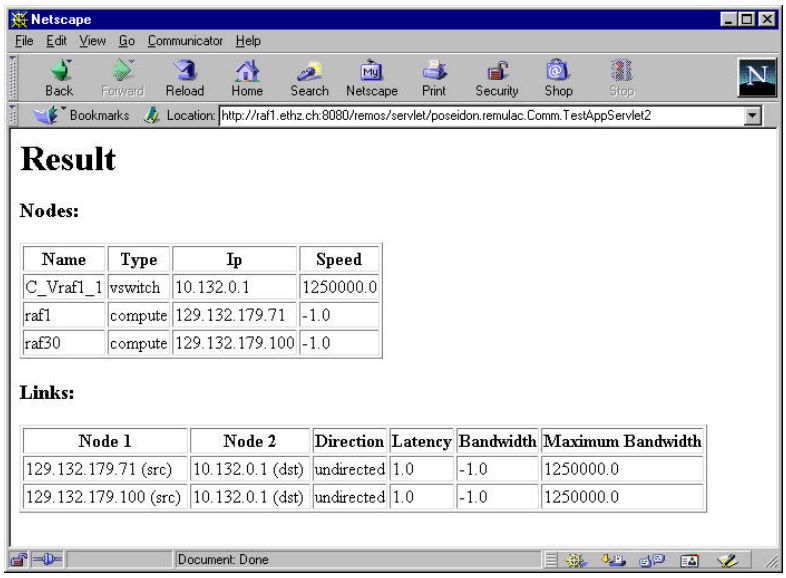

Abbildung 6.5: Test-Applikation: HTML-Antwort

Das zweite Servlet nimmt alle Inputs und erstellt mit Hilfe eines XMLRequestBuilders aus dem Comm-Paket die Anfrage. Die XML-Antwort wird mit einer XSL-Datei zu einer HTML-Antwort mit zwei Tabellen, je eine für alle Knoten und eine für alle Links, transformiert.

Kapitel 7

Performance-Analyse

In der ascii-basierte Kommunikation werden wenige Schlüsselwörter übertragenen. Die XML-Strings haben ein grösseres Volumen und werden aufwendiger generiert. Zur Erinnerung: Ein Builder aus dem Comm-Paket verwaltet die Information in einem DOM-Baum und serialisiert diesen zu einem XML-String. Es ist deshalb zu vermuten, dass die Kommunikation mittels XML langsamer ist und schlechter skaliert.

Dieses Kapitel vergleicht die Gesamtantwortzeiten des ascii-basierten Protokolls mit dem socket-basierten XML-Protokoll anhand von *Topology*-Requests an einen SNMP-Kollektor. Beim XML-Protokoll wird analysiert, welche Operationen wieviel Zeit benötigen.

Alle Aussagen in diesem Kapitel müssen mit weiteren Messungen untermauert werden. In dieser Arbeit musste kein Augenmerk auf die Performance gerichtet werden. Die Messungen wurden aus Interesse durchgeführt.

7.1 Vergleich ASCII- mit XML-Protokoll

Für den Vergleich wurden *Topology*-Requests an einen SNMP-Kollektor gesendet. Es ist die Zeit gemessen worden, bis der Modeler die Antwort in seine Daten-Strukturen umgewandelt hat. Die Anzahl Knoten des *Topology*-Requests wurde auch variiert, um eine Aussage über die Skalierbarkeit der beiden Protokolle zu tätigen. Bei 16 Knoten lagen 50 Messungen vor. Bei den anderen Knotenmengen wurden so viele Messungen durchgeführt, wie für nötig befunden (bis z.B. sichergestellt war, dass es sich um keine Ausreisser handelt).

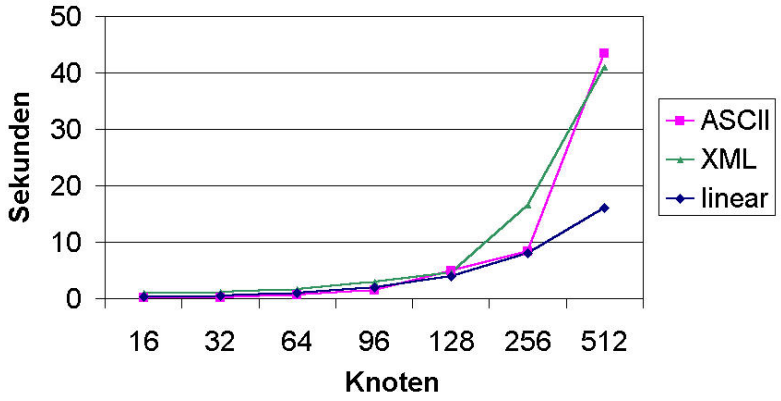

Abbildung 7.1: Skalierbarkeit von ASCII- und XML-Protokoll

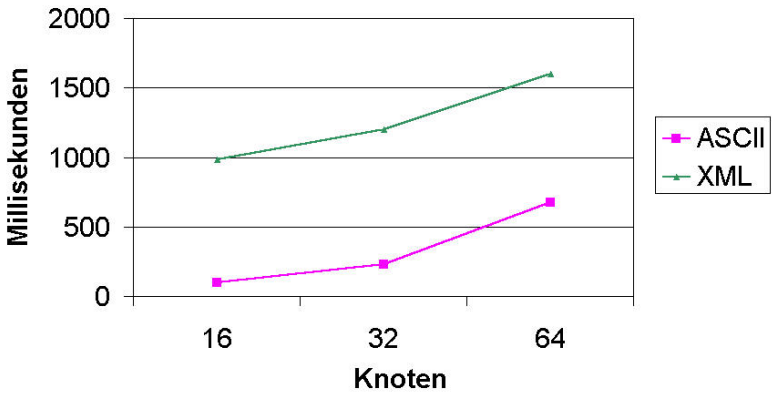

Abbildung 7.2: ASCII- und XML-Protokoll bei wenig Knoten

Wie in Abbildung 7.1 zu sehen, skalieren beide Protokolle ähnlich. Abbildung 7.2 verwendet Millisekunden und zeigt die drei Messungen mit den weinigsten Knoten. Während das ASCII-Protokoll bei 16 Knoten 85 Millisekunden benötigt, sind beim XML-Protokoll 870 Millisekunden nötig, bis der Modeler die Antwort erhält und auswertet. Das XML-Protokoll braucht in diesem Fall zehn mal länger.

7.2 Evaluation XML-Protokoll

Zur Evaluation des XML-Protokolls sind beim Modeler und beim Kollektor bei den wichtigsten Zuständen die Zeit gemessen worden. Dadurch kann be-

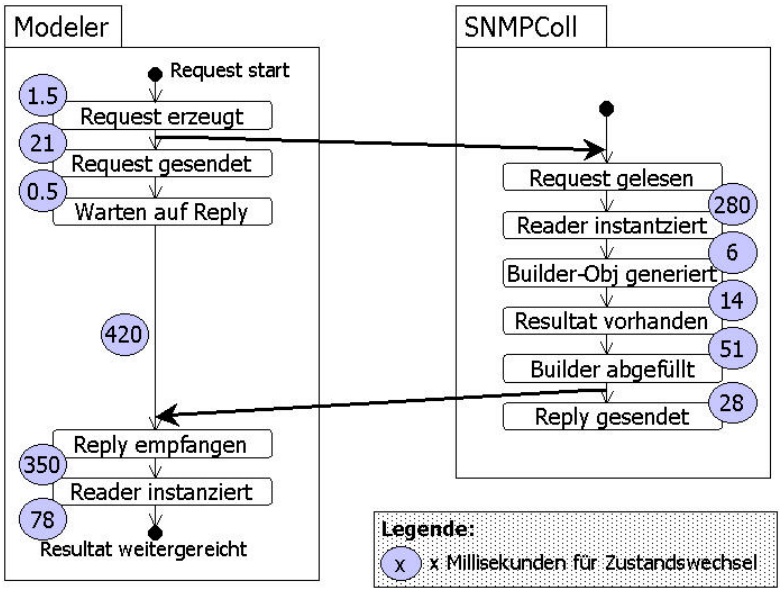

Abbildung 7.3: Zustandsübergänge im XML-Protokoll

stimmt werden, wie lange ein Zustandsübergang dauert. Gemessen wurden 50 *Topology*-Requests mit 16 Knoten.

Abbildung 7.3 zeigt die Mittelwerte für die Zustandsübergänge. Das Abfüllen der Builder dauert im Falle des Requests-Builder 1.5ms und im Falle des Reply-Builders 51ms. Diese starke Diskrepanz lässt sich damit erklären, dass die Strukturen im Kollektor komplizierter sind und es lange dauert, bis die Informationen dem Kollektor durch seine Hilfsklassen zur Verfügung gestellt werden können.

Es fällt auf, dass das Instanzieren der Reader 350ms resp. 280ms benötigt. Da der Reply umfangreicher ist, braucht das Generieren des Reply-Readers etwas länger. Ein weiterer Flaschenhals ist das Warten auf die Antwort des Kollektors, welches mit 420ms zu Buche schlägt. Untersucht man aber, wie das Protokoll skaliert, entdeckt man andere Flaschen-Hälse!

Die Abbildung 7.4 zeigt, wann ein Zustand aus Abbildung 7.3 zeitlich erreicht worden ist. Die oberste Linie beim Modeler ist die Zeit, die benötigt wird, bis der Modeler das Resultat weitergereicht hat. Die zweitoberste Linie des Modelers zeigt die Zeit, die benötigt wird, bis der Modeler den Reply-Reader instanziert hat.

Beim Modeler bilden sich zwei Lücken zwischen den Kurven. Die erste ist das Warten auf den Kollektor, die andere ist der Aufbau der Datenstruk-

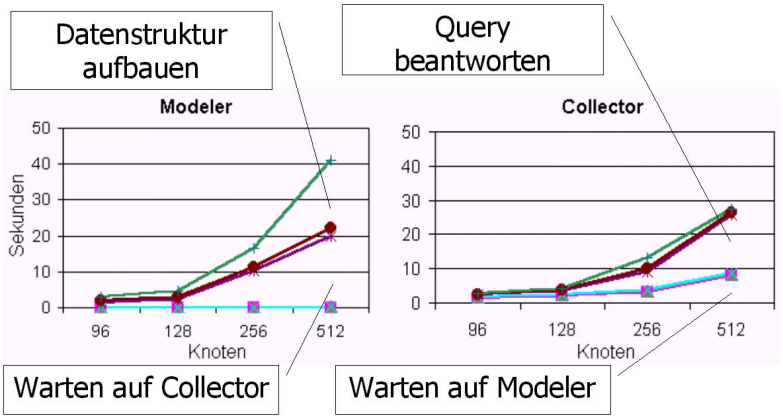

Abbildung 7.4: Skalierung im XML-Protokoll

turen. Beim SNMP-Kollektor gibt es ebenfalls zwei Lücken. Das Warten auf den Modeler und das Beantworten der Anfrage. Sowohl beim Modeler und beim Kollektor spielt das Aufbauen und Lesen der XML-Strukturen im Comm-Paket im Vergleich zur Gesamtantwortzeiten keine wesentliche Rolle. Das gegenseitige Warten ist aber durchs Comm-Paket verursacht, da das stückweise Senden und Empfangen nicht möglich ist. Der Empfänger kann mit der Arbeit erst beginnen, wenn die ganze Information übertragen worden ist. Die restlichen zwei Flaschenhälse haben nichts mit dem Comm-Paket zu tun.

Kapitel 8

Diskussion

8.1 Erfüllung der Aufgabenstellung

Während der Semesterarbeit ist die Aufgabenstellung konkretisiert und modifiziert worden.

Wir entschieden uns, in einem ersten Schritt, die Socket-Verbindungen unberührt zu lassen und lediglich das Protokoll zu auf XML umzustellen. Als feststand, dass auch die kollektorinterne Kommunikation auf XML umgestellt werden muss, entschieden wir uns, die Socket-Lösung für alle Kollektoren zu implementieren, was zu einem Mehraufwand führte.
Dies war eine gute Entscheidung, denn so gibt es eine vollständig funktionstüchtige Version, die mit der alten verglichen werden kann.

Die Umstellung auf HTTP-Verbindungen wurde gleichzeitig mit der Umstellung auf Servlets umgesetzt. Die Änderungen auf Kollektoren-Seite erwiesen sich als arbeitsintensiver als bei der Ausgabe der Semesterarbeit angenommen. Deshalb wurde entschieden, diese Umstellung nur für den SNMP-Kollektor durchzuführen. Die anderen Kollektoren können analog umgestellt werden
Die Hauptmotivation – anderen Applikationen leichten Zugriff auf die Kollektoren zu bieten – ist auch mit der Socket-Lösung einfach umsetzbar, wie es die Test-Applikation gezeigt hat.

Die Test-Applikation wurde gemäss Aufgabenstellung realisiert.

8.2 Vollständigkeit des Protokolls

Das ASCII-basierten Protokoll musste anhand des Java-Kodes analysiert werden. Für die Analyse der Kommunikation und zum Entwurf des XML-Protokolls ist die Kommunikation auf Seiten des Modelers näher angeschaut worden. Das neue Protokoll erfüllt alle Anforderungen von Seiten des Modelers, die das alte Protokoll erfüllte. Auf Seiten der Kollektoren können Anfrage-Arten (*Set*- und *Edge*-Requests) angenommen werden, die der Modeler nie sendet. Da der Kollektoren-Kode relativ undurchsichtig ist, ist der Sinn dieser Anfragearten nicht klar. Hinzu kommen Methoden, die nie aufgerufen werden.
Da der Modeler als einzige Schnittstelle nach aussen dient und alle Modeler-Anfragen mit Hilfe des XML-Protokolls beantwortet werden können, kann das neue Protokoll als vollständig betrachtet werden. Anfragen von Typ set und edge sind möglich, werden aber auf Kollektorenseite neu ignoriert, da die Semantik und Syntax zuerst geklärt und das Protokoll evtl. erweitert werden muss.

Das XML-Protokoll ist ebenfalls für die kollektorinterne Kommunikation geeignet. Die Aufgabenstellung der Semesterarbeit ging davon aus, dass nur die Modeler-Kollektor-Kommunikation auf XML umgestellt werden soll. Nachdem sich aber gezeigt hat, dass diese nicht von der kollektorinterne Kommunikation getrennt werden kann, wurde das XML-Protokoll erweitert.

8.3 Beurteilung des Comm-Pakets

Der Vorteil des Comm-Pakets liegt nun darin, dass sich Modeler und Kollektor nicht mehr um das Protokoll selber kümmern müssen. D.h. Ändert sich die XML-Struktur, tangiert dies lediglich das Comm-Paket. Man könnte sogar XML wieder ganz weglassen und über das Comm- Paket, das alte Protokoll fahren! (Dies ist aber nicht zu empfehlen, da viele Vorteile, wie automatische Validierung oder das Verwenden von XSL-Transformationen, die mit XML gewonnen worden sind, wieder verloren gehen würden.)

Heute verwendet jeder Hersteller eines XML-Parsers ein eigenes API und eigene Datenstrukturen, da alle Pioniere im Gebiet XML sind oder sein wollen. Möchte man in einem späteren Projekt den Parser (z.B: aus Performance-Gründen) auswechseln, tangiert dies wiederum nur das Comm-Paket.

Im Comm-Paket hat es noch allgemeine Objekte, wie Nodes, Links und Networks. Vorher gab es keine gemeinsamen Datentypen. Die Objekte, die es

gegeben hat, sammelten lediglich Kommunikations-Zeilen für die einzelnen Kollektoren.

Neue Applikationen, die selbst Kommunikation mit den Kollektoren betreiben wollen, können das Comm-Paket benutzen und haben die Sicherheit, eine gültige Anfrage zu stellen, ohne sich lange mit dem Aufbau eines XML-Strings zu beschäftigen.

Durch das Comm-Paket ist es aber nicht mehr möglich, die Informationen stückweise zu senden. Dadurch wird eine gewisse Wartezeit in Kauf genommen, was die Gesamt-Antwortzeit einer Anfrage verlängert. Der Modeler könnte aber während der Wartezeit anderweitig beschäftigt werden – was aber noch implementiert werden müsste.

8.4 Wahl des XML-Parsers

Für das Manipulieren und Verwalten des XML-Strings wurde der DOM-Parser von XERCES verwendet. Ebenfalls möglich wäre, JDOM zu benutzen. JDOM ist ein für Java optimierter und abgespeckter DOM-Parser, mit welchem leichter auf Informationen zugegriffen werden kann. JDOM hat dadurch eine andere Zugriffs-Semantik. Beispiel: In JDOM hat ein Element (=Tag) einen Wert. In DOM ist der Wert in einem Text-Node gespeichert, der ein Kind vom Element ist.

8.5 Kommunikations-String versenden

Die Builder-Klassen des Comm-Pakets serialisieren einen DOM-Baum zu einem String. Die Reader-Klassen erzeugen aus dem String einen DOM-Baum. Es stellt sich die Frage, ob das Versenden eines DOM-Baums effizienter wäre. Zwei wichtige Gründe sprechen aber gegen diese Idee. DOM-Bäume sind bei verschiedenen Parserhersteller mit unterschiedlichen Datenstrukturen aufgebaut. Es würden also Parser-Abhängigkeiten beim Modeler und bei den Kollektoren ins Spiel kommen. Zum anderen ist im verwendeten XERCES-Parser die Einbindung eines DTD, zur Validierung des Dokuments, erst bei der Serialisierung möglich. Das Einbinden von DTDs in DOM-Bäume wurde für eine spätere Version von XERCES versprochen.

8.6 XML-Validierung

Zur Validierung werden DTDs verwendet. Das DTD gibt zum Beispiel die Reihenfolge der Tags vor. Bei einer Anfrage muss z.B. das erste Tag ein Timeframe-Tag sein, was fürs Protokoll egal ist.

XML-Schema wäre flexibler ist aber bei vielen Parser noch nicht implementiert.

8.7 HTTP-Servlets

Durch die Servlet-Lösung müssen die Kollektoren nicht mehr von einem Administrator gestartet werden. Auch das Setzen von Umgebungsvariablen und das Bearbeiten von Konfigurationsdateien entfällt. Allerdings muss eine Java-Servlet-Umgebung installiert und gestartet werden. Für diese Arbeit wurde Tomcat-Jakarta als Servlet-Engine gewählt. Installation und Konfiguration ist dem Anhang C zu entnehmen.

8.8 Test-Applikation

Durch die Wahl, ein Servlet zu implementieren, werden Layout-Aufgaben dem Browser überlassen. Es musste lediglich für eine HTML-Ausgabe gesorgt werden.

Die Technik mit der XSL-Transformation erlaubt, den HTML-Output flexibel zu halten. Wird eine andere Ausgabe gewünscht, muss der Java-Kode nicht mehr verändert werden. Lediglich das XSL-Dokument muss den Wünschen angepasst werden.

8.9 Performance-Analyse

Um eine fundierte Analyse zu erhalten, müssen weitere Messungen durchgeführt werden. Dabei müssen typische Anfragen verwendet und alle Kollektoren betrachtet werden.

Die Performance-Analyse hat gezeigt, dass das XML-Protokoll bei wenig Knoten um Faktoren langsamer ist. Bei 16 Knoten ist die Gesamtantwortzeit aber immer noch unter einer Sekunde. Mit zunehmender Knotenanzahl skaliert das XML-Protokoll ähnlich wie das ASCII-Protokoll. Das Sammeln

und Bearbeiten der Informationen benötigt bei vielen Knoten einen grossen Anteil der Gesamtantwortzeit. Das Generieren und Lesen der XML-Strings wird vernachlässigbar.

8.10 Vorschläge für weitere Arbeiten

Diese Aussagen der Performance-Analyse gilt es mit weiteren Messreihen zu bestätigen oder zu wiederlegen. Es gilt auch abzuklären, was typische Anfragefolgen sind. In diesem Zusammenhang könnte die Parallelität von Remos verbessert werden.

Ebenfalls sollten die Kollektoren neu Entworfen werden. Sollte auf Servlets gesetzt werden, ist ein Redesign, aufgrund des Konzeptwechsels und der neuen Schnittstellen, sowieso notwendig.

Anhang A

Glossar

DOM **D**ocument **O**bject **M**odel ist eine plattformneutrale und sprachenunabhängige Schnittstelle, um dynamisch auf (XML-)Dokumente zuzugreifen und deren Struktur und Inhalt zu ändern. Es ist eine offizielle Empfehlung des World Wide Web Consortiums (W3C). Ein DOM-Parser wandelt ein Dokument in eine Baumstruktur um und traversiert resp. manipuliert sie.

DTD **D**ocument **T**ype **D**efinition beschreibt, wie (XML-)Dokumente aussehen müssen um gültig zu sein. Es beschreibt die Reihenfolge, Art, Kardinalität und Wertebereich von (XML-)Elementen des Dokuments.

HTML **H**yper**T**ext **M**arkup **L**anguage ist eine auszeichnede Beschreibungssprache für Dokumente. Dokumente im HTML-Format werden in Browsern im vom Autor gewünschten Erscheinungsbild angezeigt.

SAX steht für **S**imple **A**PI for **X**ML. Beim Parsen eines XML-Dokuments mittels SAX treten Events auf, die von einer Applikation verarbeitet werden können. SAX ist ein De-facto-Standard und eine alternative Möglichkeit, mit Inhalte eines XML-Dokuments zu arbeiten.

Xalan basiet auf die XSLT-Empfehlung des World Wide Web Consortiums (W3C) um XML-Dokumente in HTML- oder andere XML-Dokumente zu transformieren. Xalan benutzt normalerweise für seine Arbeit die Xerces-Parser.

Xerces ist eine Sammlung von XML-Parsern und umfasst Parser für DOM- und SAX-Standards.

XML E**X**tensible **M**arkup **L**anguage ist eine Strukturbeschreibungssprache auf logischer nicht auszeichnender Ebene und eignet sich deshalb sehr gut für Datenaustausch und -Abgleich. XML ist eine Empfehlung des World Wide Web Consortiums (W3C).

XML Schema gilt als Nachfolger von DTD. Es definiert ebenfalls Struktur und Semantik von XML-Dokumenten.

XSLT E**X**tensible **S**tylesheet **L**anguage **T**ransformations ist eine standardisierte Art, wie ein XML-Dokument in ein anderes XML- oder HTML-Dokument mit neuer Struktur umgewandelt werden kann. XSLT ist eine Empfehlung des World Wide Web Consortiums (W3C).

Anhang B

DTD

B.1 request.dtd

```
<!ELEMENT request (timeframe, (nodeid*| collector))>
<!ATTLIST request
    type (route | topology | close | suicide | set | edge |
          register | unregister) #REQUIRED
    version (3.0) #REQUIRED>

<!ELEMENT timeframe (#PCDATA)>

<!ELEMENT nodeid (#PCDATA)>
<!ATTLIST nodeid type (src | dst) #IMPLIED>

<!ELEMENT collector (network)+>
<!ATTLIST collector type (WANCollector | SNMPCollector) #REQUIRED>

<!ELEMENT network (ip, mask)>
<!ELEMENT ip (#PCDATA)>
<!ELEMENT mask (#PCDATA)>
```

B.2 reply.dtd

```
<!ELEMENT reply (error|node|link)*>
<!ATTLIST reply
    type (route | topology) #REQUIRED
```

```
    version (3.0) #REQUIRED>

<!ELEMENT node (name, ip, speed)>
<!ATTLIST node type (compute | switch | vswitch) #REQUIRED>

<!ELEMENT link (ip, ip, latency, bw, maxbw)>
<!ATTLIST link direction (directed | undirected) #REQUIRED>

<!ELEMENT error (errorno, node)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT ip (#PCDATA)>
<!ATTLIST ip type (src | dst) #IMPLIED>

<!ELEMENT speed (#PCDATA)>
<!ELEMENT latency (#PCDATA)>
<!ELEMENT bw (#PCDATA)>
<!ELEMENT maxbw (#PCDATA)>

<!ELEMENT errorno  (#PCDATA)>
<!ELEMENT errormsg (#PCDATA)>
```

Anhang C

Konfiguration

C.1 Eingesetzte Tools

C.1.1 Download

Jakarta-Tomcat (`http://jakarta.apache.org/tomcat/index.html`) ist eigenen Angaben zufolge *the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.*
In dieser Semesterarbeit wurde das Release 3.2.1 verwendet.

Als XML-Parser wurde Xerces 1.3.1 für Java verwendet. Dieser kann unter der URL `http://xml.apache.org/xerces-j/` bezogen werden.

Xalan wurde von `http://xml.apache.org/xalan-j/index.html` runtergeladen.

C.1.2 Installation und Konfiguration

Zuerst empfiehlt sich die Installation von Tomcat, da seine Parser durch die Parser von Xerces ersetzt werden müssen. Die Datei jakarta-tomcat-3.2.1.tar.gz muss zuerst entpackt werden. Die Installation ist abgeschlossen, wenn ein Ordner jakarta-tomcat-3.2.1 vorhanden ist. Zum Betrieb von Tomcat muss noch die Umgebungs-Variable TOMCAT_HOME mit dem Pfad des Ordners von Tomcat gesetzt werden. Doch dazu mehr im Abschnitt C.2.2

Im Ordner `lib` von TOMCAT_HOME befinden sich die mitgelieferten Parser. Da Remos neuere Versionen braucht, sind die Bibliotheken jaxp.jar und parser.jar zu entfernen. An deren Stelle muss xerces.jar eingefügt werden.

Der Vollständigkeit halber gehört auch die XSLT-Bibliothek xalan.jar in den Ordner `lib`. (Xerces.jar und Xalan.jar sind zuvor runtergeladen worden)

Im Ordner `conf` von TOMCAT_HOME muss die Datei `wrapper.properties` um folgende Zeilen erweitert werden.

```
wrapper.class_path=$(wrapper.tomcat_home)\lib\xalan.jar
wrapper.class_path=$(wrapper.tomcat_home)\lib\xerces.jar
```

Somit ist Tomcat einsatzbereit und alle benötigten Werkzeuge sind vorhanden.

C.2 Umgebungsvariablen setzen

C.2.1 Socketbasiertes Remos

Im Ordner "~/Work/xml/poseidon/commands" befindet sich die Datei ENV, welche zeilenweise alle Befehle auflistet, die aufgerufen werden müssen, bevor mit Remos gearbeitet oder weiterentwickelt werden kann. Im Vergleich zur alten Datei ist der Einbezug von xerces.jar in den Java-Pfad hinzugekommen. Unter Unix können mit dem Befehl `source` alle Befehle in einer Datei ausgeführt werden.

C.2.2 Remos auf Java-Servlet-Basis

Im Ordner "~/Work/servlets/commands" befindet sich ebenfalls eine Datei namens ENV mit der gleichen Funktion wie die obige. Zusätzlich gibt es Dateien, die den Umgang mit Tomcat erleichtern.

- install_remos kopiert alle benötigten Dateien vom in die Verzeichnisstruktur von Tomcat.
- start_jakarta setzt alle benötigten Umgebungsvariablen für Tomcat und startet Tomcat.
- stop_jakarta hält Tomcat an
- testapp startet die Test-Applikation
- uninstall_remos löscht alle Remos-Dateien von der Verzeichnisstruktur von Tomcat.

Literaturverzeichnis

[1] Doug Tidwell: *Introduction to XML*, Tutorial, Juli 1999
`http://www.ibm.com/developerworks/education/xmlintro/`

[2] Doug Tidwell: *XML programming in Java*, Tutorial, September 1999
`http://www.ibm.com/developerworks/education/xmljava/`

[3] Bob DuCharme: *Writing a data type checking XML parser with Xerces*, Artikel, Dezember 1999
`http://www.ibm.com/developerworks/library/xerces.html`

[4] Doug Tidwell: *Servlets and XML: made for each other*, Artikel, April 2000
`http://www.ibm.com/developerworks/xml/library/servlets-and-xml/`

[5] Lutz Emmerich: *Einführung in XML*, Kurs April 2001
`http://www.uni-magdeburg.de/service/xml/DTD.shtml`

[6] Roger Karrer, Nathalie Kocher: *An XML-based Remos Communicator*, Lab for Software Technology, ETH Zürich, August 2000

[7] The Remulac Group: *The Architecture of the Remos System (Extended Abstract)*, School of Computer Science, Carnegie Mellon University, März 2001

[8] The Remulac Group: *A Resource Query Interface for Network-Aware Applications*, School of Computer Science, Carnegie Mellon University

[9] The Remulac Group: *ReMoS: A Resource Monitoring System for Network-Aware Applications*, School of Computer Science, Carnegie Mellon University, Dezember 1997