Institut für Architektur von Anwendungssystemen (IAAS)

Diplomarbeit Nr. 2460

Entwicklung eines Routing-Verfahrens für SOAP-Nachrichten

Frederik Juchart

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Thorsten Scheibler
begonnen am:	13. Februar 2006
beendet am:	15. August 2006
CR-Klassifikation:	C.2.2, C.2.4, C.2.6

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Die zunehmende Vernetzung und der damit einhergehende verstärkte Datenaustausch zwischen weltweit tätigen Unternehmen in den letzten Jahren brachte auf technischer Seite viele Lösungsansätze für die system- und sprachunabhängige Kommunikation in verteilten Systemen hervor. Angefangen mit einfachen Remote-Procedure-Calls in den 70er Jahren – die noch nicht system- oder sprachunabhängig waren – über objektbasierte Verteilungsinfrastrukturen wie CORBA und DCOM in den 90ern bis hin zur aktuellen Entwicklung von Web Services, die ein lose gekoppeltes System von unabhängigen Diensten ermöglichen, entwickelten sich mehr und mehr offene Standards für den Austausch von Daten zwischen Systemen der unterschiedlichsten Hersteller.

Während frühere Technologien über keine bis maximal eine sehr stark eingeschränkte Interoperabilität (beispielsweise durch Dateiaustausch und nächtliche Stapelverarbeitung) verfügten, ermöglicht das Nachrichtenprotokoll SOAP bei den heutigen Web Services die Zusammenarbeit unterschiedlichster Systeme von verschiedensten Herstellern. SOAP ist ein vom World Wide Web Consortium (W3C) verabschiedeter auf XML basierender offener Standard zum systemunabhängigen Austausch von Nachrichten zwischen lose gekoppelten Systemen. Eine SOAP-Nachricht kann im Gegensatz zu beispielsweise den RPCs der 70er über mehrere sogenannte Intermediaries laufen, die zusätzliche Dienste für die Nachricht zur Verfügung stellen.

Im Rahmen dieser Diplomarbeit soll ein Routingverfahren für SOAP-Nachrichten entworfen und implementiert werden, um den Nachrichtenpfad festzulegen sowie die von den Intermediaries zur Verfügung gestellten Zusatzdienste in einer definierten Reihenfolge abzuarbeiten. Es existieren zwar bereits verschiedene Ansätze zum Thema Routing von

SOAP-Nachrichten, aber diese sind aus unterschiedlichen Gründen mehr oder weniger eingeschränkt, wie in Abschnitt 3.3 gezeigt wird. Ziel dieser Arbeit ist es zu zeigen, dass ein Routingverfahren mit einem BPEL-Prozess als Routinginformation möglich ist.

1.1. Motivation

Web Services machen häufig Gebrauch von zusätzlichen Diensten, um beispielsweise Loggingmechanismen oder erweiterte Sicherheitsmerkmale (z. B. WS-Security) nutzen zu können. Hierzu ist es erforderlich, dass die benötigten Dienste vom SOAP-Initial-Sender explizit im Header genannt und gegebenenfalls durch geeignete Mechanismen wie WS-Policy zwischen Sender und Empfänger vereinbart werden. Diese Zusatzdienste werden dann an den Intermediaries im SOAP-Messagepath abgearbeitet.

Leider stellt jedoch SOAP keine bestimmte Abarbeitungsreihenfolge innerhalb des Headers sicher, so dass es vorkommen kann, dass zuerst die Verschlüsselung ausgeführt wird und im Anschluß daran die verschlüsselten Daten bei einem Loggingdienst aufgezeichnet werden. Aus offensichtlichen Gründen können mit verschlüsselten Daten keinerlei Auswertungen gefahren werden, auch wenn das ursprünglich ein Ziel des Loggings gewesen sein sollte.

Es ist also nötig, die Abarbeitungsreihenfolge der Header in einer SOAP-Nachricht fest zu legen. Hierzu existiert derzeit noch kein Standard - lediglich eine sehr eingeschränkte Möglichkeit in Form der Erweiterung WS-Routing, indem dort in jeder Nachricht explizit die genaue Reihenfolge der SOAP-Intermedieries angegeben wird. Dies allein hat jedoch noch keine Auswirkung auf die Abarbeitungsreihenfolge der Header. Um diese sicher zu stellen, muss gewährleistet sein, dass auf den Intermediaries jeweils nur genau ein Zusatzdienst ausgeführt wird. Dies ist zum einen sehr aufwändig und zwingt dazu, unter Umständen unnötig viele Intermediaries einzuführen und ist zum anderen auch noch sehr unflexibel, da bei einer Änderung des Routings jeder Initial-Sender und Intermediary umkonfiguriert und im schlimmsten Falle sogar der Code verändert werden muss.

Besser wäre eine Möglichkeit, einen Prozeß zu definieren, nach dem bestimmte Nachrichten abgearbeitet werden können. Auf diesen Prozeß könnte dann im Header einer SOAP-Nachricht geeignet verwiesen werden, so daß der ursprüngliche Absender der Nachricht nur noch bestimmen muss, nach welchem Prozeß mit der Nachricht verfahren werden soll. Eine detaillierte Angabe der einzelnen SOAP-Knoten in der Nachricht ist somit nicht mehr erforderlich, da diese Informationen aus dem Prozeß gewonnen werden können. Zudem könnte hierdurch die Abarbeitungsreihenfolge der Header beeinflusst werden - je nachdem, wie der Prozeß gestaltet bzw. die SOAP-Erweiterung implementiert ist.

Mit bisherigen Routingverfahren ist es nicht möglich, eine SOAP-Nachricht über parallel ablaufende Nachrichtenpfade zu versenden und diese anschliessend wieder zu aggregieren.

Der in dieser Arbeit vorgestellte Mechanismus kann den Nachrichtenpfad festlegen, die Rei-

henfolge der Abarbeitung der Header an jedem Intermediary bestimmen und unterstützt zudem das parallele Abarbeiten einer Nachricht auf mehreren Knoten mit einer anschliessenden Aggregation.

Die Idee zur Betrachtung des Routings von SOAP-Nachrichten als Prozeß entstand bei der Beschäftigung mit SOA-Architekturen und dort speziell bei der Betrachtung der Funktionalität eines Orchestration-Service. Die Frage ist nun, ob es möglich ist, das Konzept der Orchestrierung von Diensten auch auf Zusatzdienste zu übertragen.

1.2. Aufbau der Arbeit

Zunächst wird im folgenden Kapitel das zum Veständnis benötigte Grundwissen über serviceorientierte Architekturen und Web Services gelegt sowie Routing im Allgemeinen kurz erläutert, bevor dann in Kapitel 3 der W3C-Standard SOAP mit seinem Nachrichtenformat und dem zugehörigen Verarbeitungsmodell beschrieben wird. Ein wesentlicher Teil des Verarbeitungsmodells ist die spezielle Behandlung von Headern der SOAP-Nachricht, wodurch SOAP einfach um Funktionalität erweitert werden kann. Die Möglichkeit, sogenannte Intermediaries zu nutzen, um anwendungsunabhängige Zusatzdienste anbieten zu können, ist ebenfalls Bestandteil dieses Verarbeitungsmodells. Darauf aufbauend werden bisherige Ansätze im Bereich des Routings von SOAP-Nachrichten (WS-Routing, WS-Referral) beschrieben und deren jeweilige Vor- und Nachteile aufgezeigt.

Beginnend von diesem allgemeinen Überblick und einer Beschreibung der verwendeten Standards wie WSDL und BPEL in Kapitel 4 wird im Kapitel 5 „SOAP-Routing mit BPEL“ schließlich das im Rahmen dieser Arbeit entwickelte Routingverfahren im Detail erläutert.

KAPITEL 2

Serviceorientierte Architekturen (SOA)

Bei einer serviceorientierten oder auch diensteorientierten Architektur handelt es sich nicht um ein fertig nutzbares Produkt, sondern um ein Konzept, das auf viele unterschiedliche Arten realisiert - sprich implementiert - werden kann. Eine Definition für SOA findet sich beispielsweise in [ibm06]:

> Service-Oriented Architecture (SOA) is a component model that inter-relates an application's different functional units, called services, through well-defined interfaces and contracts between these services. The interface is defined in a neutral manner that should be independent of the hardware platform, the operating system, and the programming language in which the service is implemented. This allows services, built on a variety of such systems, to interact with each other in a uniform and universal manner.

Im Sinne einer SOA ist ein Dienst also eine Einheit, die eine bestimmte Arbeit im Auftrag eines Anderen (beispielsweise eines Benutzers oder einer Software) verrichtet. In einer SOA könne solche Dienste ihrerseits wiederum Dienste nutzen, um die Arbeit zu erledigen. In einer serviceorientierten Architektur sind die Kommunikationsprotokolle, Nachrichtenformate, Schnittstellen und Dienstmerkmale festgelegt, über die die unterschiedlichen Dienste untereinander kommunizieren - hierdurch wird eine SOA erst ermöglicht.

Der wesentliche Aspekt einer serviceorientierten Architektur ist die lose Kopplung der verschiedenen Dienste: der Sender ist nicht mehr „fest“ mit dem Empfänger einer Nachricht „verdrahtet“, sondern kommuniziert mit diesem über eine Middlewareschicht, ohne den konkreten Endpunkt zu kennen. Die Middleware trifft die Routingentscheidungen, nach denen eine Nachricht zum Empfänger geleitet wird.

2.1. Routing und Routingpatterns

Unter Routing in serviceorientierten Architekturen versteht man die Weiterleitung von Nachrichten über eine Middlewareschicht zum letztlichen Empfänger. Ein Router in einer dienstorientierten Architektur leitet eine ankommende Nachricht an Hand unterschiedlicher Kriterien zu einem anderen Router weiter bis der Empfänger der Nachricht erreicht wird. Erst hierdurch wird die lose Kopplung einer solchen Architektur ermöglicht.

Integrationslösungen implementieren häufig eine sogenannte hub-and-spoke Architektur, bei der eine zentrale Instanz die vollständige Kontrolle über den Nachrichtenfluß hat. Diese Instanz – der Message-Broker – kann Nachrichten aus mehreren Quellen empfangen und eine Entscheidung treffen, wohin die Nachricht als nächstes weitergeleitet werden muss.

In aktuellen Integrationslösungen steht der Gedanke der serviceorientierten Architektur im Vordergrund. Deshalb basieren derzeitige Lösungen auf dem Enterprise Service Bus. Im Gegensatz zur zentralen Kommunikation eines Message Broker findet die Kommunikation in einem Service Bus verteilt statt. Die Router des Service Bus entscheiden dynamisch über das Routing der Nachrichten ohne eine zentrale Steuerungsinstanz.

Beide Ansätze – sowohl der Service Bus als auch der Message Broker – verwenden für die Routingentscheidung sogenannte Routing-Patterns, wobei der Service Bus dezentral mit mehreren Routern im Bus arbeitet, während der Message Broker alle Nachrichten selbst bearbeitet.

Nach [HW03] lassen sich Router in unterschiedliche sogenannte Routing-Patterns einteilen. Ein Muster (Pattern) ist eine häufig angewendete Lösungsstrategie für eine bestimmte Klasse von Problemen. Die bekanntesten Muster im Bereich des Routings sind unter anderem die Folgenden:

Content-based Router Leitet eine Nachricht an Hand des Inhaltes der Nachricht weiter. Ein CBR untersucht den Inhalt der Nachricht und entscheidet an Hand von im Router gegebenen Kriterien, an welchen Router die Nachricht weitergeleitet werden soll. Ein CBR benötigt also Anwendungswissen, um Nachrichten korrekt weiterleiten zu können.

Routing Slip Hier wird eine Nachricht entlang eines definierten Pfades geroutet. Dieser Pfad wird zur Laufzeit bestimmt und in Form eines „Routing Slip“ an die Nachricht gehängt. Jeder Knoten leitet nach erfolgreicher Bearbeitung der Nachricht diese an den nächsten im Routingslip angegebenen Knoten weiter.

Process Manager Hierbei handelt es sich um eine zentrale Einrichtung, die Nachrichten an Hand der Ergebnisse der einzelnen Zwischenschritte routet. Jeder Zwischenschritt verändert hierbei die Nachricht und leitet sie wieder an den Process Manager zurück, der darauf aufbauend eine erneute Routingentscheidung trifft.

Recipient List Leitet eine Nachricht an eine Reihe von Empfängern weiter. Die Liste der

Empfänger kann hierbei auf mehrere Arten zustande kommen. Beispielsweise gibt der Absender bereits eine Liste der Empfänger an (analog zu email) oder die Liste wird errechnet (beispielsweise aus Benutzereinstellungen).

Aggregator Fasst mehrere ankommende Nachrichten zu einer ausgehenden Nachricht zusammen. Ein Aggregator muss also erkennen können, wenn alle zusammengehörigen Nachrichten eingetroffen sind. Um mehrere Nachrichten korrekt zusammen zu fassen wird zusätzlich noch Anwendungswissen benötigt.

Die oben angesprochenen Routerarten lassen sich zu erweiterten Routing Patterns kombinieren, wodurch neue Interaktionsmuster geschaffen werden. Im Rahmen dieser Arbeit wird später insbesondere das „Scatter-Gather“ Pattern (siehe Abb. 2.1) eine wichtige Rolle spielen. Bei diesem Pattern wird eine Nachricht in mehrere Nachrichten aufgeteilt oder

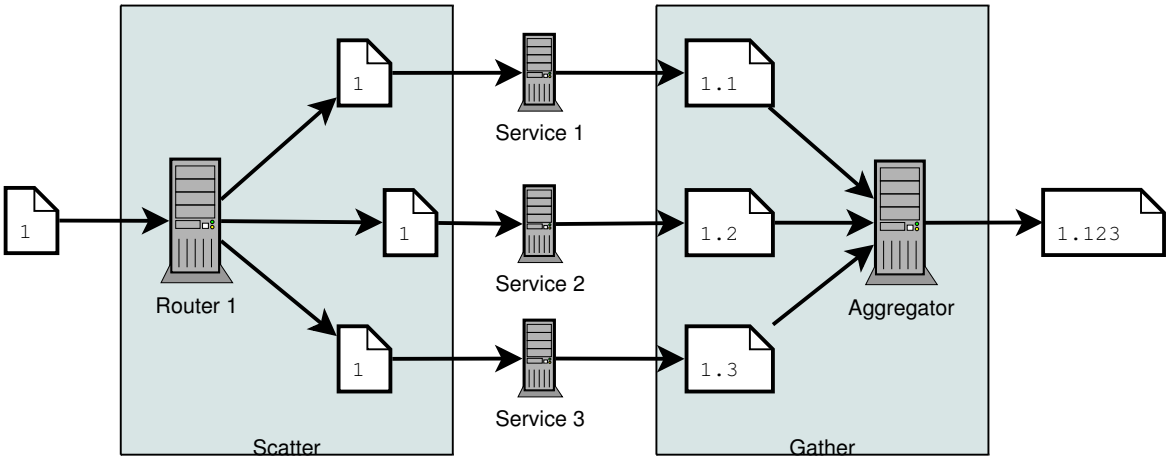

Abbildung 2.1.: Scatter Gather Routing Pattern

kopiert, um anschliessend die Antworten in einem Aggregator zu einer Antwortnachricht zusammen zu fassen. Beispielsweise kann eine Nachricht in mehrere Nachrichten kopiert und an unterschiedliche Ziele weitergeleitet werden, welche die Nachrichten bearbeiten und an einen Router weiterleiten, der als Aggregator dient und die Nachrichten wieder zu einer Nachricht zusammenfasst. Erst durch die Anwendung dieses Patterns wird eine parallele Bearbeitung von Nachrichten an unterschiedlichen Routern mit anschliessender Zusammenfassung zu einer Nachricht ermöglicht.

2.2. Web Services

Häufig wird im Zusammenhang mit serviceorientierten Architekturen der Begriff „Web Service“ genannt. Bei Web Services handelt es sich um Dienste, die eine wohl definierte sprachunabhängige Schnittstelle (WSDL) besitzen und über ein offenes Nachrichtenprotokoll

(SOAP) angesprochen werden können. Um allerdings eine vollwertige serviceorientierte Architektur im Sinne der oben zitierten Definition zu werden, müssen Web Services bzw. muss die Infrastruktur der Web Services um weitere Funktionalität wie z. B. Sicherheit und Zuverlässigkeit erweitert werden. Das Nachrichtenprotokoll SOAP (vgl. Kapitel 3) ermöglicht es, solche Erweiterungen in Form von zusätzlichen Headerblöcken zu implementieren.

Mehrere Web Services können mit Hilfe eines BPEL[1]-Prozesses zu einem neuen Web Service kombiniert werden. Dabei werden die Zusatzdienste jedoch nicht berücksichtigt, so dass hierfür eine gesonderte Methode gefunden werden muss - beispielsweise ein geeignetes Routing der Nachrichten durch Intermediaries, auf denen entsprechende Zusatzdienste installiert sind.

[1]Business Process Execution Language

KAPITEL 3

SOAP

Ursprünglich aus dem Protokoll zum entfernten Aufruf von Methoden XML-RPC (Remote Procedure Call = RPC) hervorgegangen, wuchs SOAP seit der ersten Veröffentlichung im Jahre 1999 dank der Unterstützung vieler namhafter internationaler Firmen zu dem universellen Nachrichtenprotokoll, das in der Welt der heterogenen XML-basierten verteilten Systeme und insbesondere der Web Services zum Standard geworden ist. Der Erfolg von SOAP beruht insbesondere auf seiner Einfachheit, die rasch zu einer weiten Akzeptanz und somit zu einer schnellen Verbreitung führte.

Die SOAP-Spezifikation (derzeit Version 1.2 – vgl. [GHM+03]) konzentriert sich ausschließlich darauf, ein Rahmenwerk für den Austausch von Nachrichten zu definieren und schreibt somit die grundlegende Funktionalität eines solchen „Messaging-Framework“ fest. Darunter fallen die Definition eines Nachrichtenformates, das Festlegen eines Verarbeitungsmodells für diese Nachrichten, Fehlerbehandlung, Erweiterungsmöglichkeiten, Datenrepräsentationsformate sowie eine Abbildung von RPCs auf SOAP-Nachrichten und letztlich eine Bindung des SOAP-Protokolls an darunter liegende Kommunikationsprotokolle wie z. B. HTTP[1] oder SMTP[2]. Insbesondere fehlen bei SOAP die für eine verteilte Infrastruktur üblichen Dienste wie beispielsweise Sicherheit, Routing oder Zuverlässigkeit.

In den folgenden Abschnitten wird das Nachrichtenformat sowie das SOAP zugrunde liegende Verarbeitungsmodell kurz erläutert.

[1] HyperText Transport Protocol
[2] Simple Mail Transport Protocol

3.1. Nachrichtenformat

Eine SOAP-Nachricht besteht aus einem XML-Dokument – dem sogenannten „SOAP-Envelope“ [3] [w3c03]. Jede Nachricht (vgl. Listing 3.1) besteht aus dem Wurzelelement `envelope`, das die Elemente `header` und `body` enthält, wobei das Vorhandensein eines Headerelements optional ist. Innerhalb des Headers einer Nachricht ist es möglich, weitere Headerblöcke frei zu definieren, die an unterschiedliche SOAP-Rollen gerichtet sein können und die Einfluß auf die Bearbeitung der Nachricht haben und eine Möglichkeit zur Einbindung eigener Erweiterungen (Logging, Security, Addressing) bieten.

Ein Headerblock einer SOAP-Nachricht kann beliebige Daten im XML Format enthalten, wobei Headerblöcke meist dazu genutzt werden entweder das SOAP-Rahmenwerk zu erweitern (beispielsweise durch Sicherheitsaspekte, das Hinzufügen von Transaktionen oder das Einbinden von Informationen für erweiterte Nachrichteninteraktionsmuster) oder um zusätzliche Anwendungsdaten einzubinden, die von älteren Anwendungsversionen als Teil des Bodys nicht verstanden würden (z. B. ein zusätzliches Datenfeld) [GDS+04]. Somit ist ein Headerblock die geeignete Stelle für eine Routing-Erweiterung von SOAP.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
</soapenv:Header>
<soapenv:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
</soapenv:Body>
</soapenv:Envelope>
```

Listing 3.1: Eine einfache SOAP-Nachricht (aus [GHM+03])

Der Body einer Nachricht enthält die Anwendungsdaten beziehungsweise den entfernten Prozeduraufruf und spielt für die Entwicklung eines Routingverfahrens im Rahmen dieser Arbeit keine weitere Rolle.

3.2. Verarbeitungsmodell

Zusätzlich zum reinen Format der übertragenen Nachrichten wird in der SOAP-Spezifikation genau festgelegt, wie eine Nachricht zu verarbeiten ist. Jede SOAP-Nachricht wird entlang des Nachrichtenpfades (Abb. 3.1) vom *SOAP Initial Sender* optional über eine Reihe von *SOAP Intermediaries* hin zum *SOAP Ultimate Receiver* transportiert. Im ein-

[3] Namensraum des SOAP-Envelope: `xmlns:soapenv=‘‘http://www.w3.org/2003/05/soap-envelope‘‘`

fachsten Fall wird die Nachricht ohne Intermediary direkt vom Sender zum Empfänger übertragen.

Jeder SOAP-Knoten entlang des Nachrichtenpfades muss bei der Bearbeitung einer Nachricht mehrere Schritte in vorgegebener Reihenfolge durchlaufen - zumindest muss das von aussen zu beobachtende Verhalten dementsprechend sein:

1. Die Rollen (`role`) bestimmen, als die der Knoten die Nachricht bearbeiten soll. Für diese Entscheidung kann sowohl der SOAP-Envelope als auch der Header ebenso wie der Body heran gezogen werden.

2. Feststellen der Headerblöcke, die als `mustUnderstand` zwingend bearbeitet werden müssen und die an diesen Knoten gerichtet sind (`role`).

   Kann einer dieser Headerblöcke nicht bearbeitet werden, so muß der bearbeitende Knoten eine SOAP Fehlermeldung mit dem Fehlercode `mustUnderstand` generieren und an den Absender der Nachricht schicken. In diesem Fall darf keine weitere Bearbeitung der Nachricht erfolgen.

3. Alle an diesen Knoten addressierten in Schritt 2 identifizierten zwingenden Headerblöcke abarbeiten. Ein SOAP-Knoten kann zusätzlich optionale Headerblöcke bearbeiten – muss dies jedoch nicht tun.

4. Sollte der Knoten der endgültige Empfänger der Nachricht sein, wird der Body der Nachricht ebenfalls bearbeitet. Im Falle eines *Intermediary* wird die SOAP-Nachricht entlang des Nachrichtenpfades (siehe Abb. 3.1) weitergeleitet.

Wie unschwer zu erkennen ist, wurde das SOAP-Verarbeitungsmodell so entworfen, dass mit Hilfe des `mustUnderstand`-Attributes das Vorhandensein einer bestimmten Erweiterung zwingend vorgeschrieben werden kann. Somit kann mit Hilfe dieses Mechanismus eine Routing-Erweiterung implementiert werden.

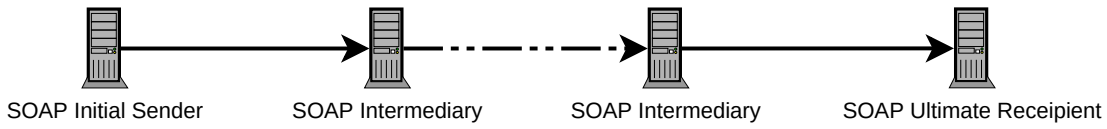

Abbildung 3.1.: SOAP Messagepath

Im Gegensatz zu Routern auf IP-Basis, die Pakete nur unverändert weiterleiten, können SOAP-Intermediaries die Nachrichten, die sie weiterleiten bearbeiten und verändern. Aus Anwendungssicht bleibt der Inhalt der Nachricht hierbei jedoch unverändert - d. h. es werden maximal sogenannte Zusatzdienste oder Merkmale hinzugefügt, die am Inhalt der Nachricht aus Anwendungssicht nichts verändern (z. B. Security, Logging, QoS. . .). Hierbei kann es allerdings sein, dass eine Erweiterung den Inhalt der Nachricht verschlüsselt und diese somit vor der endgültigen Bearbeitung im Ultimate Receiver auf geeignete Weise entschlüsselt werden muss.

3.3. Bisherige Routing-Ansätze

Das Routing von SOAP-Nachrichten wurde erst in wenigen Spezifikationen behandelt, die im Folgenden kurz vorgestellt werden sollen.

3.3.1. WS-Routing mit WS-Referral

Die WS-Routing-Spezifikation [NT01] wurde im Oktober 2001 von Microsoft freigegeben und beschreibt ein „SOAP-basierendes, zustandsloses Protokoll zum Austausch von Nachrichten zwischen dem Initial Sender und dem Ultimate Receiver“. Bei WS-Routing wird der Nachrichtenpfad bereits zu Beginn festgelegt, wobei WS-Routing Intermediaries zusätzliche Knoten in den Pfad einfügen können. Auf diese Weise kann der Nachrichtenpfad einfach und effizient dargestellt werden.

```
<rp:path xmlns:m="http://schemas.xmlsoap.org/rp/">
  <rp:action>http://www.im.org/chat</rp:action>
  <rp:to>soap://D.com/some/endpoint</rp:to>
  <rp:fwd>
    <rp:via>soap://B.com</rp:via>
    <rp:via>soap://C.com</rp:via>
  </rp:fwd>
  <rp:rev>
    <rp:via/>
  </rp:rev>
  <rp:from>mailto:henrikn@microsoft.com</rp:from>
  <rp:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</rp:id>
</rp:path>
```

Listing 3.2: WS-Routing Headerformat (aus [NT01])

Der in Listing 3.2 abgebildete WS-Routing-Header zeigt die wesentlichen Elemente der WS-Routing-Spezifikation:

from die Adresse des Absenders der Nachricht.

to die Zieladresse der Nachricht (= Ultimate Receiver).

action wird vom ursprünglichen Sender der Nachricht festgelegt und darf im Verlauf des Routings nicht verändert werden. `Action` erfüllt denselben Zweck wie der SOAPaction Header bei der Bindung von SOAP an das HTTP-Protokoll: die Angabe des Zwecks der Nachricht in Form einer URI.

fwd enthält den Pfad, den eine Nachricht nehmen soll.

rev gibt den Rückwärtspfad an, den Antwortnachrichten nehmen sollen. Ein leeres `via`-Element zeigt an, dass der umgekehrte Pfad durch das darunterliegende Transportprotokoll zur Verfügung gestellt wird.

via gibt die Adresse des nächsten Knoten im Pfad an. Nach der Weiterleitung wird das jeweils erste via-Element des aktiven Pfades gelöscht.

id enthält das eindeutige Kennzeichen der Nachricht. Wird beispielsweise beim Request-Response Nachrichtenpattern als Referenz genutzt.

Es gibt jedoch ein paar Nachteile: Zum ersten erfordert eine grundlegende Änderung des Routings vielfältige Änderungen an mehreren Intermediaries oder gar an allen Nachrichtensendern. Zweitens wird in der WS-Routing-Spezifikation zwar auf „additional services“ eingegangen, nicht aber auf die Reihenfolge dieser Zusatzdienste, so dass mittels WS-Routing lediglich die Reihenfolge der Intermediaries bestimmt werden kann – nicht aber die Reihenfolge der Abarbeitung der einzelnen Header der SOAP-Nachricht.

Für Änderungen am Routing wurde zur selben Zeit ebenfalls von Microsoft in Form von WS-Referral [NCLL01] ein zusätzliches Protokoll entworfen, das es ermöglicht, Routingeinträge zu verändern, neue Einträge zu erstellen oder einfach nur die Routingtabelle auszulesen. Hierdurch sollte eine Art dynamisches Routing ermöglicht werden, bei dem die Intermediaries durch den Austausch von Routinginformation und die Delegation von URL-Bereichen an andere Router in die Lage versetzt werden sollten, automatisch den richtigen Pfad für eine Nachricht zu finden. WS-Referral stellt also eine Art Konfigurationsmöglichkeit für SOAP-Router - unabhängig von der verwendeten Routing-Erweiterung - dar. Leider sind zur Änderung einer kompletten Route sehr viele Nachrichten nötig, da WS-Routing lediglich das Einfügen, Löschen und Verschieben von Knoten ermöglicht, nicht jedoch die Übertragung einer kompletten Route.

Weder WS-Routing noch WS-Referral haben sich letztlich durchsetzen können. WS-Routing wurde 2004 durch die Veröffentlichung der WS-Addressing-Spezifikation schließlich abgelöst.

3.3.2. WS-Adressing

WS-Addressing [GHR06a] kann jedoch im Gegensatz zu WS-Routing den eigentlichen Nachrichtenpfad nicht beschreiben. Somit ist WS-Addressing kein Routingprotokoll, ist jedoch im Umfeld der Web Services längst nicht mehr weg zu denken. Grund hierfür ist die in WS-Addressing enthaltene Definition von Endpunktreferenzen für Web Services (Listing 3.3) und deren Bindung an SOAP [GHR06b], so dass hierdurch eine transportunabhängige Beschreibung von Webserviceendpunkten in SOAP ermöglicht wird.

Ein Intermediary kann jedoch an Hand von WS-Addressing Informationen und einer lokalen Konfiguration bestimmen, welchen Weg eine bestimmte Nachricht nehmen soll. Bei dieser Art des Routings sind die Routinginformationen nicht direkt in der Nachricht enthalten.

```
<wsa:EndpointReference>
    <wsa:Address>xs:anyURI</wsa:Address>
    <wsa:ReferenceParameters>xs:any*</wsa:ReferenceParameters> ?
    <wsa:Metadata>xs:any*</wsa:Metadata>?
</wsa:EndpointReference>
```

Listing 3.3: WS-Addressing: Endpoint-Reference (aus [GHR06a])

Die einzelnen Elemente einer WSA-Endpointreferenz sind `Address`, `ReferenceParameters` und `Metadata`. `Address` enthält einen URI, der die Netzwerkadresse des Endpunktes angibt.

Das Element `ReferenceParameters` enthält Informationen, um eine bestimmte Interaktion mit dem Endpunkt zu ermöglichen. Die Parameter werden üblicherweise vom Herausgeber der Endpunktreferenz bereitgestellt und die Bedeutung ist normalerweise dem Nutzer einer Endpunktreferenz bekannt. Wie die `ReferenceParameters` an eine Nachricht gebunden werden, hängt von der verwendeten Protokollbindung (z. B. SOAP) ab.

Im `Metadata`-Element können das Verhalten, die Rahmenbedingungen sowie die Fähigkeiten des zu einer Endpunktreferenz gehörenden Dienstes beschrieben werden. Sowohl `ReferenceParameters` als auch `Metadata` sind optional.

Zusätzlich zu der definierten Endpoint-Reference Struktur werden in WS-Addressing Message Addressing Properties beschrieben, die es ermöglichen, die an einer Interaktion beteiligten Endpunkte an Hand ihrer Endpunktreferenz zu identifizieren sowie die Beziehung zwischen den verschiedenen Nachrichten zu beschreiben. Listing 3.4 zeigt die definierten Elemente, die im Folgenden erläutert werden.

```
<wsa:To>xs:anyURI</wsa:To> ?
<wsa:From>wsa:EndpointReferenceType</wsa:From> ?
<wsa:ReplyTo>wsa:EndpointReferenceType</wsa:ReplyTo> ?
<wsa:FaultTo>wsa:EndpointReferenceType</wsa:FaultTo> ?
<wsa:Action>xs:anyURI</wsa:Action>
<wsa:MessageID>xs:anyURI</wsa:MessageID> ?
<wsa:RelatesTo RelationshipType="xs:anyURI"?>xs:anyURI</wsa:RelatesTo> *
<wsa:ReferenceParameters>xs:any*</wsa:ReferenceParameters> ?
```

Listing 3.4: WS-Addressing: Message Addressing Properties (aus [GHR06a])

`wsa:To` enthält den absoluten URI des Empfängers der Nachricht.

`wsa:From` enthält den absoluten URI des Absenders der Nachricht.

`wsa:ReplyTo` enthält den absoluten URI des Empfängers der Antwort zu dieser Nachricht.

`wsa:FaultTo` enthält den absoluten URI des Empfängers von Fehlermeldungen zu dieser Nachricht.

`wsa:Action` enthält einen absoluten URI, welcher die Semantik der Nachricht beschreibt. Es wird empfohlen, dies mittels eines Verweises auf ein Element eines WSDL-Dokument zu tun. `wsa:Action` sollte auf eine in einem WSDL-Dokument definierte Input, Output, Fault-Message oder einen dort definierten PortType verweisen.

`wsa:MessageId` enthält einen absoluten URI, der die Nachricht eindeutig kennzeichnet. Es liegt in der Verantwortung des Absenders einer Nachricht, eindeutige `MessageId`s zu vergeben.

`wsa:RelatesTo` beschreibt die Art der Beziehung und die Beziehung zwischen unterschiedlichen Nachrichten, die hierbei durch ihre `MessageId` identifiziert werden.

`wsa:ReferenceParameters` enthält beliebige XML-Elemente, die so wie sie sind übertragen werden und somit anwendungsspezifische Erweiterungen zum Nachrichtenaustausch enthalten können.

Alle diese Elemente sind bis auf `wsa:Action` optional und ihr Inhalt wird bei Abwesenheit des Elementes auf einen definierten Standardwert gesetzt.

Bei einem auf WS-Addressing aufsetzenden Routing kann also der Pfad der Nachricht nicht mit WS-Addressing Mitteln vorgegeben werden. Das Routing kann in diesem Fall also nur durch Konfiguration der einzelnen Intermediaries in Abhängigkeit von im WS-Addressing-Header enthaltenen Informationen (z. B. `wsa:to`) beeinflusst werden. D. h. es werden die Routingentscheidungen an Hand von in den Intermediaries hinterlegten Informationen getroffen – womit ein erhöhter Konfigurationsaufwand für die Routingregeln der einzelnen Intermediaries nötig ist.

Eine solche Möglichkeit bietet beispielsweise das Apache Projekt „Synapse“[4], in dem an einem Intermediary an Hand von vorgegebenen Regeln Entscheidungen bezüglich des Routings und der abzuarbeitenden Header getroffen werden.

3.3.3. Folgerung

In den vorangegangenen beiden Abschnitten wurden bisherige Ansätze zum Thema Routing im weitesten Sinne beschrieben. Jeder dieser Ansätze hat individuelle Vor- und Nachteile – allen gemeinsam ist jedoch, dass es keine Möglichkeit gibt, die Abarbeitungsreihenfolge von Headerblöcken in SOAP auf standardisierte Art und Weise zu beschreiben. Insbesondere ist bei keinem der beiden Ansätze eine Möglichkeit für ein parallel ablaufendes Routing vorgesehen.

Auf Grund dieser Einschränkungen sollte im Rahmen dieser Arbeit ein Verfahren entwickelt werden, das es einerseits ermöglicht, die Abarbeitungsreihenfolge der Header an einem SOAP-Knoten festzulegen und das darüber hinaus auch das parallele Verarbeiten von Nachrichten auf unterschiedlichen Knoten mit anschliessender Zusammenführung (Aggregation) der einzelnen Nachrichtenpfade bietet, so dass beim Ultimate Receipient letztlich nur eine Nachricht ankommt.

[4] http://incubator.apache.org/synapse/

KAPITEL 4

WSDL und BPEL

Um die Definition der einzelnen Nachrichtenformate und die Darstellung eines Routings als BPEL-Prozess verstehen zu können, müssen zunächst die grundlegenden Standards WSDL und BPEL erläutert werden.

4.1. WSDL

Die Web Service Description Language [CCMW01] dient im Umfeld der Web Services aus Implementierungssicht denselben Zwecken wie eine klassische Interface Definition Language (IDL) z. B. bei CORBA. In WSDL werden also die verwendeten Datentypen sowie die genaue Syntax der beim Aufruf von Operationen und der Rückgabe von Ergebnissen verwendeten Nachrichten im Detail beschrieben.

4.1.1. Struktur eines WSDL-Dokuments

Die Struktur eines WSDL-Dokumentes besteht aus mehreren Definitionselementen, die ihrerseits weitere Elemente zur näheren Definition enthalten. Wie die einzelnen Elemente untereinander verknüpft sind, ist in Abb. 4.1 dargestellt.

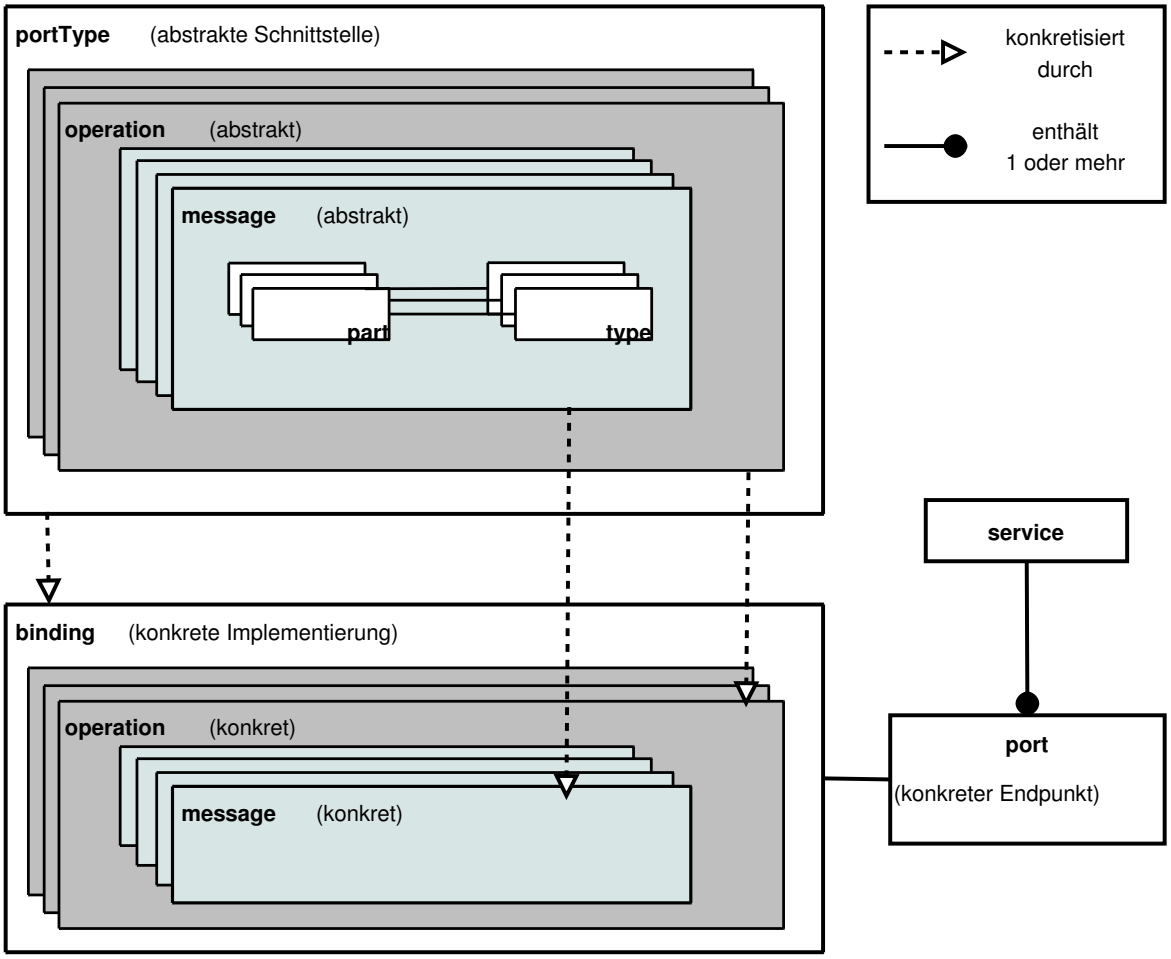

Abbildung 4.1.: Zusammenhang der Elemente von WSDL (fehlerbereinigt aus [GDS+04])

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
  <import namespace="uri" location="uri"/>*

  <wsdl:types> ?
  </wsdl:types>

  <wsdl:message name="nmtoken"> *
  </wsdl:message>

  <wsdl:portType name="nmtoken">*
    <wsdl:operation name="nmtoken">*
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="nmtoken" type="qname">*
    <wsdl:operation name="nmtoken">*
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="nmtoken"> *
    <wsdl:port name="nmtoken" binding="qname"> *
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing 4.1: Struktur eines WSDL-Dokuments

Ein WSDL-Dokument beschreibt also Datentypen, Nachrichten, PortTypen, Ports, Bindungen und Dienste. Die Struktur eines solchen Dokumentes sieht – beschränkt auf die Hauptelemente – wie in Listing 4.1 aus. Die vollständige Struktur kann in der Spezifikation nachgelesen werden.

4.1.2. Elemente eines WSDL-Dokuments

Die Elemente eines WSDL-Dokuments reichen aus, um die Schnittstelle eines Webdienstes exakt zu definieren. Die Bedeutung der wichtigsten Elemente:

`types` definiert die Datentypen, die von den `part`-Elementen der `message`-Elemente referenziert werden können.

```
<types>
  <xsd:schema .... />*
</types>
```

`message` enthält die Definitionen der Nachrichten, die von den Methoden genutzt werden. Ein `message`-Element kann aus mehreren `part`-Elementen bestehen, die den Parametern der aufgerufenen Methode entsprechen und ein Element aus der `types`-Definition referenzieren.

```
<message name="nmtoken"> *
  <part name="nmtoken" element="qname"? type="qname"?/> *
</message>
```

`portType` enthält die Schnittstellendefinition des Web Services. Im `portType`-Element werden die einzelnen Methoden des Web Services in `operation`-Elementen aufgezählt. Die `input`, `output`, `fault`-Elemente bestimmen durch eine Referenz auf ein `message`-Element den Typ der von dieser Operation verwendeten Nachricht. Je nachdem, welche Art des Nachrichtenaustausches durch eine `operation` unterstützt werden soll (Request/Response, Solicit/Response, One-Way, Notification), können die `input`, `output` und `fault` Elemente entfallen.

```
<wsdl:portType name="nmtoken">
  <wsdl:operation name="nmtoken" parameterOrder="nmtokens"?>*
    <wsdl:input name="nmtoken"? message="qname"/>
    <wsdl:output name="nmtoken"? message="qname"/>
    <wsdl:fault name="nmtoken" message="qname"/>*
  </wsdl:operation>
</wsdl:portType>
```

`binding` enthält Informationen über die konkrete Darstellung und die Art der Übertragung der Daten eines `portType`s auf Protokollebene (z. B. SOAP über HTTP/SMTP,

HTTP POST). Hierzu sind innerhalb der einzelnen `input`, `output` und `fault` Elemente Erweiterungselemente erlaubt, die Protokollabhängig sind und in der Spezifikation des jeweiligen Bindings definiert werden.

```
<wsdl:binding name="nmtoken" type="qname"> *
  <-- extensibility element (1) --> *
  <wsdl:operation name="nmtoken"> *
    <-- extensibility element (2) --> *
    <wsdl:input name="nmtoken"? > ?
      <-- extensibility element (3) --> *
    </wsdl:input>
    <wsdl:output name="nmtoken"? > ?
      <-- extensibility element (4) --> *
    </wsdl:output>
    <wsdl:fault name="nmtoken"> *
      <-- extensibility element (5) --> *
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

`port` beschreibt innerhalb eines `service` die zur Verfügung stehenden Operationen mit ihren Protokollbindungen.

```
<wsdl:port name="nmtoken" binding="qname"> *
  <-- extensibility element (1) -->
</wsdl:port>
```

`service` enthält eine Menge von `port`-Elementen, die jeweils konkrete Bindungen (z. B. an SOAP 1.2 über HTTP) für einen definierten `portType` angeben.

```
<wsdl:service name="nmtoken"> *
  <wsdl:port .... />*
</wsdl:service>
```

Kurz zusammengefasst bietet ein WSDL-Dokument die exakte Beschreibung eines Web Services inklusive eventueller Protokollbindungen und deren speziellen Anforderungen. Über Erweiterungsmechanismen lassen sich zusätzliche WSDL-Elemente definieren, um beispielsweise neue Protokolle zu unterstützen. Genauere Informationen zu WSDL finden sich unter anderem bei [GDS+04] in Kapitel 4 „Describing Web Services“ oder direkt in der WSDL Spezifikation des W3C [CCMW01].

4.2. BPEL

Die in dieser Arbeit genutzte Version 1.1 der „Business Process Execution Language for Web Services“ (BPEL4WS) wurde im Mai 2003 von mehreren großen Unternehmen[1] gemeinsam veröffentlicht. Ziel von BPEL ist es, auf Basis von Web Services und damit verbundenen Standards eine Integrationsplattform zu schaffen, um Geschäftsprozesse mit Hilfe von Web Services in einem einheitlichen Prozessintegrationsmodell zu verknüpfen. Während das Interaktionsmodell von WSDL lediglich eine Reihe von zustandslos übertragenen Nachrichten unterstützt, wird von Geschäftsprozessen typischerweise das Halten eines Zustandes und somit die Möglichkeit, Entscheidungen basierend auf den Geschäftsdaten zu treffen, gefordert. BPEL-Prozesse kommunizieren ausschließlich über Web Service Schnittstellen.

In BPEL können sowohl abstrakte als auch ausführbare Prozesse beschrieben werden. Im Falle eines abstrakten Prozesses wird kein direkt ausführbarer Prozess beschrieben, sondern die Struktur eines Prozesses. Dies ermöglicht die Beschreibung von Protokollen zum Nachrichtenaustausch zwischen Geschäftspartnern, ohne jedoch die genauen Kriterien zu liefern, nach denen einzelne Partner Entscheidungen treffen. Ausführbare Prozesse hingegen beziehen die Entscheidungskriterien mit ein, so dass der Prozess ankommende Nachrichten entsprechend ihres Inhaltes behandeln und an die zuständigen Partner weiterleiten kann.

Die komplexen und zustandsbehafteten Interaktionen mehrerer an einem Geschäftsprozess beteiligten Parteien lassen sich mit Hilfe von BPEL formal beschreiben, ohne die jeweiligen Implementierungsdetails zu veröffentlichen. Die Geheimhaltung der Implementierungsdetails ist aus zwei Gründen wichtig: zum einen ermöglicht dies den beteiligten Parteien auch zu einem späteren Zeitpunkt die Implementierung zu ändern, ohne dass deshalb gleich alle Partner ebenfalls Änderungen durchführen müssen. Der zweite Grund ist, dass Unternehmen Aussenstehenden üblicherweise keinen Einblick in interne Entscheidungskriterien bieten wollen.

Kurz: BPEL ermöglicht die Orchstrierung von Web Services zu einem Prozess. Hierzu definiert BPEL eine ganze Reihe von Sprachelementen, die im Folgenden erläutert werden sollen.

4.2.1. Struktur eines BPEL-Dokuments

Ein BPEL-Dokument besitzt immer eine fest vorgegebene Struktur (die inneren Elemente wurden zur besseren Lesbarkeit entfernt):

```
<process name="ncname" targetNamespace="uri"
         queryLanguage="anyURI"?
         expressionLanguage="anyURI"?
         suppressJoinFailure="yes|no"?
```

[1] BEA, IBM, Microsoft, SAP, Siebel

```
        enableInstanceCompensation="yes|no"?
        abstractProcess="yes|no"?
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  <partnerLinks>?
  </partnerLinks>
  <partners>?
  </partners>
  <variables>?
  </variables>
  <correlationSets>?
  </correlationSets>
  <faultHandlers>?
  </faultHandlers>
  <compensationHandler>?
  </compensationaHandler>
  <eventHandlers>?
  </eventHandlers>
  activity
</process>
```

Zunächst werden für den Prozess globale Parameter wie die zu verwendende Abfragesprache (`queryLanguage`) und die Sprache für Ausdrücke (`expressionLanguage`) festgelegt. Weiterhin wird bestimmt, ob es sich bei dem vorliegenden Prozess um einen ausführbaren oder abstrakten Prozess handelt (`abstractProcess`).

Der Parameter `suppressJoinFailure` bestimmt das Verhalten des Prozesses für den Fall, dass die `joinCondition` einer Aktivität zu `false` evaluiert wird. Wird dieser Parameter auf `yes` gesetzt, so führen nicht zutreffende `joinCondition`s nicht zu einem Fehler im Prozess sondern lediglich dazu, dass die vorliegende Aktivität nicht ausgeführt und alle ausgehenden `link`s auf negativ gesetzt werden.

Mit `enableInstanceCompensation` kann festgelegt werden, ob die Prozessinstanz durch Einwirkung von Aussen mit Hilfe eines `compensationHandler` als Ganzes rückgängig gemacht bzw. im Ausgang nachträglich (also nach normaler Beendigung des Prozesses) geändert werden kann.

Im folgenden Abschnitt werden die einzelnen Elemente des BPEL-Dokumentes und deren Bedeutung beschrieben.

4.2.2. Elemente eines BPEL-Dokuments

Die Hauptelemente, die in einem BPEL-Dokument genutzt werden können, sind Partner-Links, Partner, Variablen, CorrelationSets, Handler und Aktivitäten.

Mittels `partnerLinks` wird die Verknüpfung zwischen einer Aktivität in BPEL und einem verwendeten Web Service über das in dem zum Web Service gehörenden WSDL-Dokument beschriebene Nachrichtenformat bestimmt. Die `partnerLinkTypen`, auf die sich ein `partnerLink` immer beziehen muss, werden in einem WSDL-Dokument spezifiziert.

```
<partnerLinks>?
  <partnerLink name="ncname" partnerLinkType="qname"
               myRole="ncname"? partnerRole="ncname"?/>+
</partnerLinks>
```

Durch die `partnerLink`-Definitionen wird das Nachrichtenformat bestimmt, das zu einer bestimmten BPEL-Aktivität gehört.

`Variablen` dienen in BPEL dazu, empfangene Nachrichten zwischen zu speichern oder zu sendende Nachrichten vorzubereiten. In einer Variablen können nicht nur Nachrichten gespeichert werden, sondern beliebige durch ein Schema definierte XML-Dokumente.

```
<variables>?
  <variable name="ncname" messageType="qname"?
            type="qname"? element="qname"?/>+
</variables>
```

`CorrelationSets` sind für das Auffinden einer bestimmten Instanz eines Prozesses nötig. Hier werden diejenigen Elemente einer Nachricht genannt, die eine Prozessinstanz eindeutig identifizieren können (z. B. Kundennummer, Auftragsnummer). `correlationSets` werden später bei der Erstellung von Routingprozessen eine zentrale Rolle spielen.

```
<correlationSets>?
  <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>
```

Handler fangen Ereignisse ab, die parallel zur Ausführung anderer Aktivitäten auftreten können. Eventhandler reagieren auf das Eintreffen von Nachrichten (`onMessage`), den Ablauf einer Zeitspanne oder aber das Erreichen eines Zeitpunktes (`onAlarm`), FaultHandler enthalten eine eventuelle Fehlerbehandlung und der CompensationHandler dient zur Änderung des Prozessstatus von ausserhalb – beispielsweise durch eine Abbruchnachricht für einen Bestellprozess. Stellvertretend für die unterschiedlichen Handler wird hier nur der `eventHandler` gezeigt. Die Definitionen sämtlicher Handler-Typen befindet sich in [ACD+03].

```
<eventHandlers>?
  <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>
    <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
    <correlations>
    activity
```

```
  </onMessage>
  <onAlarm for="duration-expr"? until="deadline-expr"?>*
    activity
  </onAlarm>
</eventHandlers>
```

Aktivitäten in BPEL

In BPEL sind verschiedene Aktivitäten wie die Kontrollstrukturen `scope`, `sequence`, `flow`, `pick`, `switch`, `while`, `terminate`, `throw` und `compensate` sowie die Nachrichtenaktivitäten `receive`, `reply` und `invoke` definiert. Daneben gibt es noch `assign`, `wait` und `empty`.

`scope` bestimmt einen Gültigkeitsbereich. Anstatt für alle Daten globale Variablen nutzen zu müssen, ermöglicht ein `scope` eine bessere Strukturierung des Prozesses durch begrenzte und getrennte Gültigkeitsbereiche für Variablen, CorrelationSets und Handler. Bei verschachtelten `scope`s sind die Variablen, CorrelationSets und Handler des übergeordneten `scope` im inneren `scope` sichtbar.

```
<scope variableAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <variables>?
  ...
  </variables>
  <correlationSets>?
  ...
  </correlationSets>
  <faultHandlers>?
  ...
  </faultHandlers>
  <compensationHandler>?
  ...
  </compensationHandler>
  <eventHandlers>?
  ...
  </eventHandlers>
  activity
</scope>
```

`sequence` Alle innerhalb einer `sequence` stehenden Aktivitäten werden in genau der Reihenfolge ausgeführt, in der sie im BPEL-Dokument stehen.

```
<sequence standard-attributes>
  standard-elements
  activity+
```

```
</sequence>
```

`flow` ermöglicht die Graph-basierte Modellierung von Abläufen in BPEL. Jede Aktivität innerhalb eines `flow` ist ein Knoten in diesem Graph. Durch `link`s werden die Aktivitäten miteinander verknüpft. Jeder Aktivität verfügt hierbei über eingehende und ausgehende Kanten. Jede Aktivität bestimmt sich über `source` bzw. `target` als Quelle oder Ziel einer Kante. Aktivitäten ohne eingehende Kanten werden zu Beginn des `flow` parallel ausgeführt. Die Reihenfolge der übrigen Aktivitäten wird durch die `link`s bestimmt: jede Aktivität verfügt über eine `joinCondition`, die angibt, welche Vorbedingungen erfüllt sein müssen, damit die Aktivität ausgeführt werden kann. Die `joinCondition` wird ausgewertet, sobald der Zustand aller eingehenden Kanten der Aktivität bestimmt wurde. Die Kanten selbst müssen getrennt vom `flow`-Element definiert werden. Mit Hilfe von `flow`-Elementen kann die Parallelität in einem Routingprozess ausgedrückt werden.

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="ncname">+
  </links>
  activity+
</flow>
```

`pick` enthält mehrere Ereignisse. Bei Auftreten eines der spezifizierten Ereignisse wird die entsprechende Aktivität ausgeführt. Anschliessend ist `pick` nicht mehr aktiv, d.h. es wird auf das Eintreffen genau eines der spezifizierten Ereignisse gewartet.

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>+
    <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>
```

`switch` bietet die aus vielen Programmiersprachen gewohnte Funktionalität der Verzweigung an Hand eines gegebenen Kriteriums. Wird der `otherwise`-Zweig weggelassen, so wird als Standardaktivität `empty` ausgeführt, falls keiner der definierten Zweige des `switch`-Konstruktes zutrifft.

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

`while` Das `while`-Konstrukt ist in BPEL die einzige Art, Schleifen darzustellen. Andere Schleifenarten wie z. B. for-Schleifen können hiermit nachgebildet werden.

```
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

`terminate` beendet den laufenden Prozess vorzeitig ohne Fehlerbearbeitung durch Faulthandler und ohne Ausführung eines Compensationhandlers.

```
<terminate standard-attributes>
  standard-elements
</terminate>
```

`throw` ermöglicht die explizite Signalisierung eines Fehlers. Anschliessend können mit Hilfe eines zugehörigen Faulthandlers dem Fehler entsprechende Aktivitäten ausgeführt werden.

```
<throw faultName="qname" faultVariable="ncname"? standard-attributes>
  standard-elements
</throw>
```

`compensate` ermöglicht die explizite Rücknahme von Änderungen im Prozess. Beispielsweise können mit Hilfe von `compensate` innerhalb eines `faultHandler` beim Auftreten eines Fehlers verschiedene Aktivitäten rückgängig gemacht werden. `compensate` ruft gezielt den `compensateHandler` des angegebenen `scope` auf.

```
<compensate scope="ncname"? standard-attributes>
  standard-elements
</compensate>
```

`receive` ermöglicht den Empfang von Nachrichten durch den Prozess. Im Rahmen jeder `receive`-Aktivität werden `partnerLink`, `portType` und `operation` zwingend angegeben. Ein `receive` ist die einzige Möglichkeit, eine neue Instanz eines Prozesses zu starten: hierzu muss lediglich der Parameter `createInstance` auf „yes“ gesetzt werden. Um Nachrichten mit vorangegangenen Nachrichten in Beziehung zu setzen,

kann auf `correlations` zurück gegriffen werden.

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
         variable="ncname"? createInstance="yes|no"?
         standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</receive>
```

`reply` bildet das Gegenstück zu `receive` und ermöglicht es dem Prozess, auf eine durch `receive` empfangene Nachricht zu antworten. Diese Aktivität wird nur für synchronen Nachrichtenaustausch verwendet. Eine Antwort auf eine asynchron empfangene Nachricht erfolgt mittels `invoke`.

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
       variable="ncname"? faultName="qname"? standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</reply>
```

`invoke` ermöglicht es dem Prozess Web Services aufzurufen oder auf zu einem früheren Zeitpunkt empfangene Nachrichten asynchron zu antworten. Letztlich wird in beiden Fällen mittels `invoke` ein entfernter Web Service angesprochen.

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
        inputVariable="ncname"? outputVariable="ncname"?
        standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?
                 pattern="in|out|out-in"/>+
  </correlations>
  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
```

```
</invoke>
```

`assign` weist Variablen oder Properties Werte zu. Dies können entweder Werte aus anderen Variablen, empfangenen Nachrichten oder Ergebnisse von Berechnungen sein – oder schlicht die Zuweisung eines fest vorgegebenen Wertes.

```
<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

`wait` ermöglicht es dem Prozess, für eine bestimmte Zeit oder bis zum in `deadline-expr` angegebenen Zeitpunkt zu warten.

```
<wait (for="duration-expr" | until="deadline-expr")
      standard-attributes>
  standard-elements
</wait>
```

`empty` führt eine Aktivität ohne Wirkung aus. Die `empty`-Aktivität wird beispielsweise dann genutzt, wenn ein `fault` abgefangen und unterdrückt werden muss. Da innerhalb des `faultHandler`s eine Aktivität erforderlich ist, wird hier `empty` verwendet.

```
<empty standard-attributes>
  standard-elements
</empty>
```

BPEL bietet mit den hier vorgestellten Elementen die Möglichkeit, Geschäftsprozesse mit parallelen Abläufen zu modellieren. Eine detailliertere Beschreibung der Konzepte von BPEL findet sich unter anderem in [WCL+05].

KAPITEL 5

SOAP-BPEL-Routing (SBR)

Wie bereits in Kapitel 3 „SOAP“ beschrieben, existiert derzeit keine geeignete Möglichkeit, die Abarbeitungsreihenfolge der Header sowie den Nachrichtenpfad einer SOAP-Nachricht festzulegen. In diesem Kapitel wird eine Routingerweiterung für SOAP beschrieben, die zusätzlich zur rein sequentiellen Anordnung der SOAP-Knoten noch die parallele Abarbeitung von Headern auf unterschiedlichen Nachrichtenpfaden ermöglicht. Darüber hinaus wird noch die Reihenfolge der Headerabarbeitung (Zusatzdienste) berücksichtigt. An einem SBR-Router können somit mehrere Zusatzdienste in vorgegebener Reihenfolge abgearbeitet werden.

Eine Nachricht kann mit Hilfe dieser Erweiterung nicht nur eine streng lineare Anordnung von SOAP-Knoten durchlaufen, sondern kann an einem SOAP-Knoten kopiert werden und zusätzlich weitere Nachrichtenpfade nehmen. Sollten diese parallelen Pfade wieder zusammengeführt werden müssen, so erfordert dies zusätzlich zum „normalen“ Routing die Möglichkeit für einen SOAP-Knoten zu erkennen, wann auf zusätzliche Nachrichten zur Aggregation gewartet werden muss. Ein Mechanismus hierfür wird ebenfalls in diesem Kapitel vorgestellt.

Im Rahmen dieser Arbeit soll ein Routingverfahren entwickelt werden, dass sowohl die Abarbeitungsreihenfolge der Header festlegen kann als auch eine parallele Abarbeitung auf mehreren Intermediaries erlaubt. Zur Beschreibung eines solchen Routings wurde auf BPEL zurückgegriffen, da diese Prozessbeschreibungssprache alle Mittel besitzt, um Parallelität abbilden zu können und es zudem für BPEL bereits zahlreiche grafische Bearbeitungssoftware gibt, die ein einfaches Erstellen von Routingprozessen ermöglichen.

Die Verwendung von BPEL als Routingbeschreibung soll zum einen der zunehmenden Ver-

breitung von BPEL bei der Beschreibung von Geschäftsprozessen in Unternehmen Rechnung tragen und zum anderen das dadurch in Unternehmen bereits vorhandene Fachwissen auch für die Erstellung von Routinginformation nutzbar machen. Dadurch muss sich ein Administrator nicht in eine zusätzliche Beschreibungssprache einarbeiten, um ein Routing zu definieren.

Zur Erläuterung des entwickelten Verfahrens wird zunächst ein grober Überblick gegeben. Im Anschluss daran wird das Headerformat für die SOAP-BPEL-Routing-Erweiterung (SBR-Erweiterung) festgelegt. Nach der Headerbeschreibung folgt die Beschreibung der WSDL-Schnittstelle zwischen SBR-Router und Routingprozess sowie eine Erläuterung der von aussen beobachtbaren Eigenschaften eines Routingprozesses. Im Anschluss hieran erfolgt eine Beschreibung der Abläufe des Routings an den einzelnen SBR-Routern sowie der Kommunikation mit dem Routingprozess. Der Routingprozess selbst wird im Rahmen des Beispiels in Abschnitt 5.6 aufgezeigt.

5.1. Überblick

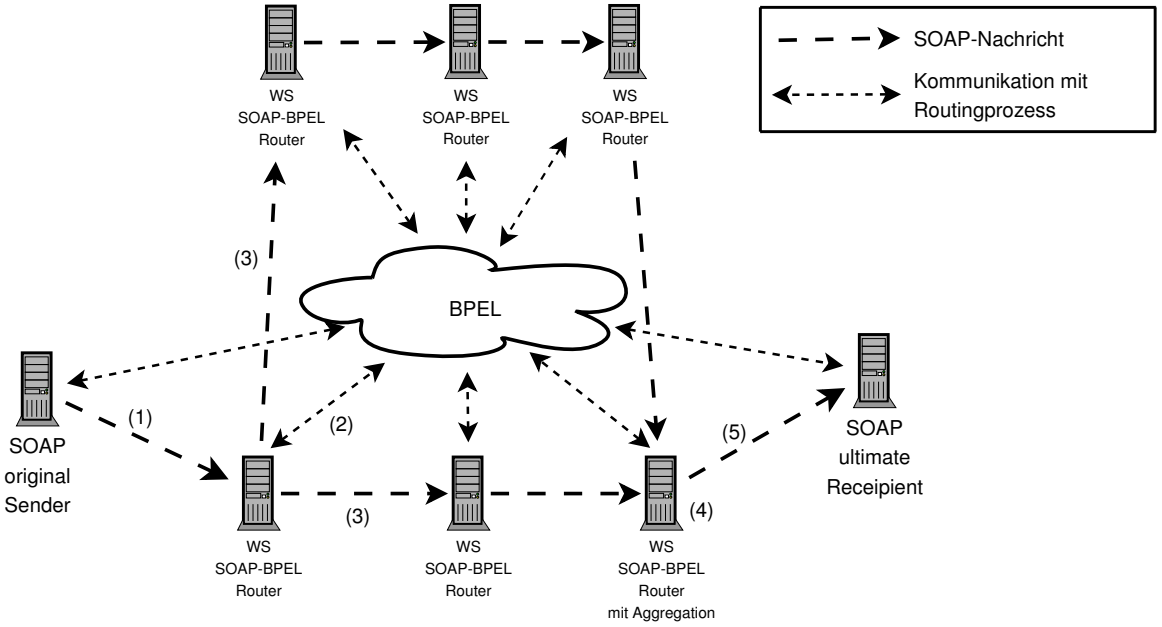

Abbildung 5.1.: Nachrichtenpfade beim SOAP-BPEL Routing

In Abb. 5.1 wird beispielhaft das Routing einer Nachricht über zwei parallele Routen dargestellt. Der Initial Sender erfragt zunächst vom Routingprozess den URI des ersten Intermediary und schickt anschliessend seine Nachricht an diesen Intermediary (1), der vom Routingprozess (2) erfragt, was mit der Nachricht zu tun ist. Anschliessend routet er die Nachricht entsprechend der Beschreibung im Prozess an zwei weitere Intermediaries (3). Hierbei wird die Nachricht dupliziert.

Jeder weitere Intermediary befragt seinerseits den Routingprozess und bearbeitet bzw. routet die Nachricht entlang der Nachrichtenpfade weiter. Der letzte Intermediary (4) vor dem Ultimate Receipient verfügt in diesem Beispiel über einen zusätzlichen anwendungsabhängigen Aggregationsdienst, der durch einen im Routing definierten Standardmechanismus (siehe Abschnitt 5.5.3) dafür sorgt, dass die über die unterschiedlichen Nachrichtenpfade ankommenden Nachrichten wieder korrekt zu einer Nachricht zusammengefasst werden. Anschliessend wird die Nachricht an den Ultimate Receipient weitergeleitet (5), der die Nachricht bearbeitet.

Die Kommunikation zwischen den einzelnen SBR-Routern und dem Routingprozess wird in Abschnitt 5.5 an Hand eines Beispiels detailliert beschrieben.

5.2. Routinginformationen

Um ein erfolgreiches Routing zu ermöglichen, ist es nötig, im Routingheader einer Nachricht bestimmte Informationen zu übermitteln. Welche Informationen dies im einzelnen sind, hängt von der Art des Routings ab.

Bei einem simplen Routing mit Next-Hop Semantik bestimmt jeweils der aktuelle Router an Hand von dem in der Nachricht vorgegebenen Ziel den nächsten anzusprechenden Hop mit Hilfe einer Routingtabelle selbst. Dies ist z. B. bei Routern für das Internet Protocol (IP) der Fall: hier wird an Hand der Zieladdresse und der gegebenen Subnetzmaske von jedem Router auf der Strecke der nächste Hop ausgewählt. In diesem Fall reicht es für das Routing aus, die Zieladresse im Header der Nachricht zu vermerken. Die Information, wohin die Nachricht als nächstes geroutet werden muss, ist in jedem Router in Form einer Routingtabelle enthalten.

Etwas anders sieht es bei dem in Abschnitt 3.3.1 beschriebenen WS-Routing aus: hier wird der Pfad, den die Nachricht nehmen soll, direkt in den Header der Nachricht geschrieben und die Router entlang des Pfades müssen die entsprechenden Informationen aus dem Header der Nachricht entnehmen.

Im Falle des in dieser Arbeit zu entwickelnden Routingverfahrens wird die Routinginformation in einem BPEL-Prozess festgehalten. Ein Router muss in diesem Fall also die für das Routing nötige Information aus dem Routingprozess entnehmen können. Da die zu einer Nachricht gehörende Prozessinstanz immer einen Zustand besitzt, muss dieser ebenfalls im Routingheader der Nachricht geeignet vermerkt sein. Die zum Routing einer Nachricht benötigten Informationen, die in der Nachricht enthalten sein müssen, sind der Routingprozess selbst bzw. ein Verweis auf eine Prozessinstanz, eine Nachrichtennummer, um die zur Nachricht gehörende Instanz auffinden zu können sowie – auf Grund der möglichen Parallelität im Routing – eine Pfadidentifikationsnummer, um den Pfad erkennen zu können, auf dem sich die vorliegende Nachricht befindet. Die Pfadidentifikationsnummer wird hierbei zu Beginn des Routings auf *1* gesetzt und später vom Routingprozess verwaltet.

Da das vorliegende Routing zusätzlich die Reihenfolge der Abarbeitung der Header der SOAP-Nachricht bestimmen können muss, ist es zusätzlich erforderlich, die am Router auszuführenden zu den Headerblöcken gehörenden Zusatzdienste zu nennen. Da jeder Headerblock eindeutig durch seinen Namespace und den Namen des Headerblockes gekennzeichnet ist, lassen sich diese Informationen zur Bestimmung des Zusatzdienstes nutzen.

Um die parallelen Pfade des Routings durch eine Aggregation wieder zusammenführen zu können, müssen die an einem Router zusammenzuführenden Pfade in jeder der zusammenzuführenden Nachrichten genannt werden. Dies ist erforderlich, damit der Aggregationsdienst die Nachrichten in der richtigen Reihenfolge bekommt. Da eine Aggregation von Nachrichten immer anwendungsabhängig ist, so ist es nötig, in den Nachrichten einen Aggregationsdienst zu definieren, der für die Zusammenfassung der Nachrichten genutzt werden soll.

5.3. Referenzierung einer Route im SOAP Header

Um das Routing einer SOAP-Nachricht beeinflussen zu können, ist es zunächst nötig, eine geeignete Headerstruktur zu definieren, welche die für ein erfolgreiches Routing benötigten Informationen enthält. Hierzu gibt es zwei Möglichkeiten, die in Erwägung gezogen wurden:

1. Serialisierung des Prozessstatus im Header
2. Verweis auf einen zentralen BPEL-Prozess

Diese beiden Möglichkeiten werden im Folgenden mit ihren Vor- und Nachteilen beschrieben. Im Anschluss daran wird das Headerformat basierend auf der Entscheidung für eine der beiden Varianten definiert.

5.3.1. Möglichkeit 1: Serialisierung des Prozessstatus im Header

Zunächst existiert die Möglichkeit der Serialisierung des Zustandes des Routingprozesses mit einer Übertragung dieses Status als XML innerhalb des Routing-Headers der SOAP-Nachricht.

Ein großer Vorteil der Serialisierung des Prozessstatus in der Nachricht ist, dass somit jede SOAP-Nachricht sämtliche Informationen mitbringt, um den Prozess auf einer beliebigen[1] BPEL-Maschine zu instanziieren.

Problematisch ist jedoch, dass BPEL-Maschinen keine Möglichkeit bieten, auf den internen Prozessstatus von ausserhalb zugreifen oder diesen gar von aussen ändern zu können. Für einen solchen Zugriff ist eine Implementierung nötig, die direkt die internen Klassen der

[1] Voraussetzung ist die Möglichkeit der Instanziierung eines Prozesses aus dem serialisiertem Prozessstatus

jeweiligen BPEL-Maschine nutzt und den dort laufenden Prozess kontrolliert. Hierdurch ergibt sich aber eine Abhängigkeit der Implementierung des Routings von der API des Herstellers einer bestimmten BPEL-Maschine. Da sich interne Programmierschnittstellen zudem von Version zu Version ändern können, ist dies keine allzu stabile Alternative. Ausserdem ist es ja gerade der Vorteil der Plattformunabhängigkeit von SOAP und BPEL, der wesentlich für deren Erfolg war. Aus diesem Grund sollten auch Erweiterungen so flexibel und unabhängig wie möglich sein, um überhaupt Chancen auf eine nennenswerte Verbreitung zu haben.

Zudem ergibt sich durch die Verwendung der BPEL-Maschine über API-Aufrufe die Notwendigkeit, auf jedem SOAP-Knoten, der das Routing unterstützen soll, eine BPEL-Maschine zu installieren oder alternativ eine zentrale BPEL-Maschine über einen zusätzlichen Wrapperdienst zu verwenden – wobei letzteres die Vorteile des serialisierten Status aufhebt und wie bereits erwähnt eine Abhängigkeit von einem bestimmten Hersteller einführt.

Nachteilig wirkt sich auch die Größe der SOAP-Nachricht mit Statusinformation bei geringem Nutzdatenvolumen auf die Übertragung und letztlich den Nachrichtendurchsatz aus. Das Verhältnis von Nutzdaten zu Routinginformation ist hier insbesondere bei kleinen Nachrichten – in Abhängigkeit der Komplexität des definierten Routings – sehr schlecht. Man stelle sich ein kompliziertes Routing vor, dessen Beschreibung und Statusinformationen mehrere Kilobytes an Daten benötigen, um einen einfachen RPC mit wenigen hundert Bytes Größe zu routen.

Eine Variante der Serialisierung des Prozessstatus besteht darin, nicht den Status, sondern nur den noch auszuführenden Teilbaum des Routingprozesses im Header mitzuführen. Dies hat den Vorteil, dass jeder Knoten genau weiss, wie das gesamte Routing nach dem Knoten aussieht. Dabei treten jedoch dieselben Unzulänglichkeiten auf wie beim Routing mit WS-Routing: der Absender der Nachricht muss bereits den gesamten Ablauf des Routings kennen und nachträgliche Änderungen am Routing werden sehr aufwändig und erfordern unter Umständen die Neukonfiguration sämtlicher Initial Sender. Zudem besteht auch in dieser Variante weiterhin die Abhängigkeit von der Programmierschnittstelle einer bestimmten BPEL-Maschine, da keine standardisierte Möglichkeit zur Erzeugung von Prozessinstanzen aus einer Prozessbeschreibung heraus existiert. Insbesondere lassen sich die Prozessinstanzen nicht auf standardisierte Weise serialisieren.

5.3.2. Möglichkeit 2: Verweis auf zentrale BPEL-Maschine

Die zweite Möglichkeit besteht darin, innerhalb einer SOAP-Nachricht nur einen Verweis auf einen Routingprozess in Form eines URI mit zu führen. Hier entstehen zwei Probleme: wie können zwei Unternehmen auf einen gemeinsamen zentralen Dienst (in diesem Fall den Routingprozess) zugreifen? Wer ist für die Administration dieses Dienstes zuständig? In den meisten Fällen wird dies aus Sicherheitsgründen von den beteiligten Unternehmen abgelehnt: keine Firma wird es einer anderen ermöglichen, Zugriff auf Netzwerkstruktur-

informationen zu bekommen.

Das zweite Problem besteht darin, dass der gesamte Routingvorgang von einer zentralen Stelle abhängig wird und somit ein Single Point of Failure eingeführt wird. Sollte die zentrale BPEL-Maschine ausfallen, können Nachrichten nicht mehr geroutet werden, da die benötigten Informationen nicht mehr abgefragt werden können.

Beide Probleme lassen sich lösen, indem der Routingprozess auf mehrere Unterprozesse verteilt wird, die jeweils auf die nächste BPEL-Maschine verweisen. In Abbildung 5.2 liefert z. B. der Routingprozess 1 bei der Anfrage von SBR-Router (n) die Adresse von Routingprozess 2 als bei Anfragen von SBR-Router (m) zuständigen Routingprozess.

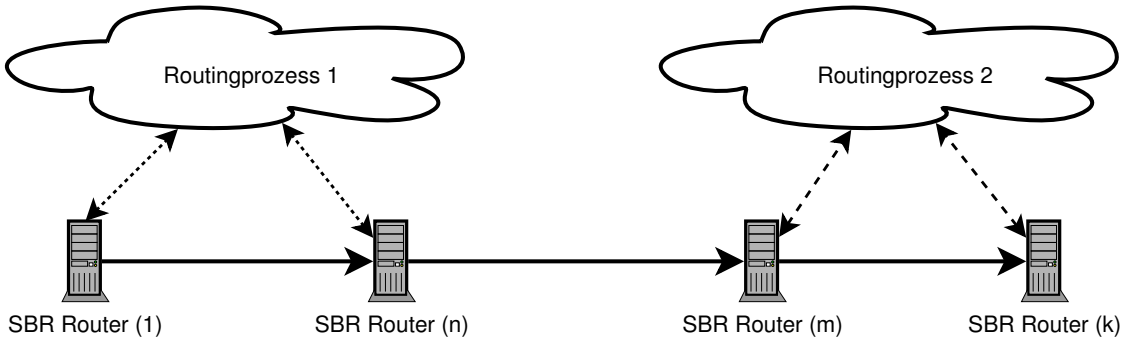

Abbildung 5.2.: Routing mit zwei Routingprozessen

Ein einzelner Routingprozess wird folglich aufgeteilt und liefert bei einer Anfrage eine Referenz auf den nachfolgenden Prozess. Dies ist zwar mit erhöhtem Modellierungs- und Administrationsaufwand verbunden, ermöglicht aber andererseits relativ einfach die Überwindung von Unternehmensgrenzen, da immer nur ein Teil des Routings bekannt sein muss. Zudem wird die Abhängigkeit von einem zentralen Routingprozess stark reduziert – je nachdem, wie sehr der Prozess aufgeteilt wird. Im Extremfall könnte nach jedem Routingschritt der Routingprozess gewechselt werden. Die einzelnen Routingprozesse können hierdurch unter Umständen wesentlich einfacher in ihrer Struktur werden, da komplexe parallele Vorgänge auf mehrere Prozesse verteilt werden können.

5.3.3. Entscheidung

Im Vergleich zur Option des in der Nachricht eingebetteten Prozessstatus ist die Nachrichtengröße bei der zweiten Möglichkeit deutlich verringert und somit das Verhältnis von Nutzdaten zu Routinginformationen wesentlich verbessert. Auch ist eine Abhängigkeit von einer verwendeten nicht-standardisierten Programmierschnittstelle nicht mehr gegeben. Prinzipiell kann somit jede BPEL-Maschine verwendet werden, die den verwendeten BPEL-Standard unterstützt.

Da es sich bei BPEL um einen offenen Standard handelt, ergibt es keinen Sinn, sich im Zusammenhang mit der Verwendung einer BPEL-Engine zur Ausführung von BPEL-Prozessen auf die API eines bestimmten Herstellers fest zu legen. Dies wäre gegeben, wenn der Status des Routingprozesses im Header der Nachricht serialisiert enthalten wäre. Weil es in diesem Fall zudem nötig wäre, auf jedem Intermediary eine BPEL-Maschine zu installieren und per API zu nutzen, wurde für diese Arbeit die zweite Möglichkeit gewählt.

5.3.4. Das Headerformat für Möglichkeit 2

Das somit zu entwerfende Headerformat muss für ein erfolgreiches Routing mehrere Daten enthalten. In Abbildung 5.3 ist das Format eines solchen `RoutingInfo`-Headers grafisch dargestellt. Die einzelnen Elemente werden im Folgenden erläutert. Das komplette XML-Schema kann in Anhang B nachgelesen werden.

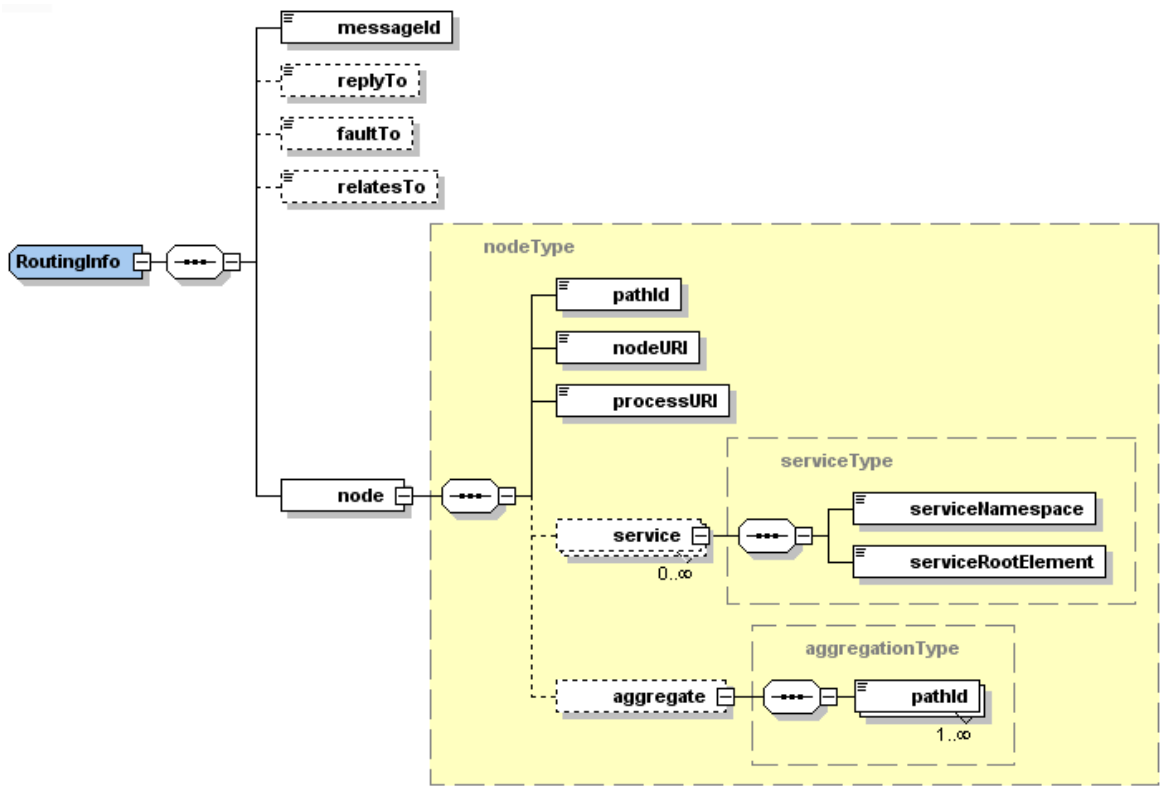

Abbildung 5.3.: SBR-Header Überblick

Jede Nachricht muss eindeutig identifiziert werden, um bei mehreren Nachrichten, die nach demselben Routingprozess bearbeitet werden sollen, die Nachricht der korrekten Prozessinstanz zuordnen zu können. Hierzu ist es erforderlich, jede Nachricht mit einer eindeutigen `messageId` zu versehen. Um den für das Routing der Nachricht zuständigen Prozess an-

sprechen zu können, muss eine Referenz (`processURI`) auf den Prozess im Header der Nachricht vorhanden sein.

Sollte im Verlauf des Routings ein Fehler auftreten, so müssen die hierdurch erzeugten SOAPFaults an ein vom Sender der Nachricht spezifiziertes Ziel geschickt werden. Hierzu wird das `faultTo`-Element genutzt. Um Nachrichteninteraktionsmuster wie Request-Response unterstützen zu können, muss ein URI im `replyTo`-Element angegeben werden, zu dem Antworten geschickt werden können. Über das `relatesTo`-Element im SBR-Header wird bei einer Antwort die `messageId` der beantworteten Nachricht übermittelt.

Darüber hinaus sind Informationen darüber nötig, auf welchem Pfad (`pathId`) innerhalb eines Routings sich die den Header enthaltende Nachricht befindet. Dies ist zum einen nötig, um bei Anfragen an den Routingprozess mit Hilfe von CorrelationSets die korrekte `receive`-Aktivität ansprechen zu können. Der zweite Zweck des `pathId`-Elementes ist es, bei einer Aggregation (`aggregate`) auf dem betreffenden Knoten die zusammengehörigen Nachrichten sammeln und in korrekter Reihenfolge an den Aggregationsdienst weiterleiten zu können.

Die `pathId` ist zu Beginn des Routings vom Initial Sender auf *1* zu setzen und wird im Verlauf des Routings vom Routingprozess verwaltet. Diese Informationen befinden sich gesammelt im `node`-Element, wo auch die am Knoten abzuarbeitenden Zusatzdienste in der Reihenfolge ihrer Abarbeitung in `service`-Elementen festgelegt sind.

Das Element `nodeURI` enthält den URI des Knoten, an den die Nachricht direkt gesendet wird. Dadurch kann vom Empfänger geprüft werden, ob die Nachricht wirklich an ihn gerichtet ist.

Die `service`-Elemente enthalten die eindeutigen Namen der am SBR-Router auszuführenden Zusatzdienste. Jeder Zusatzdienst ist für die Abarbeitung eines Headerblockes in SOAP zuständig. Headerblöcke werden an Hand ihres QName (vgl. [BM04]) unterschieden. Der QName besteht immer aus einem Namespace und einem lokalen Teil – dem Namen des referenzierten Elementes. Um einen Zusatzdienst eindeutig zu kennzeichnen, reicht es also aus, den Namespace und den Namen des Headerblockes anzugeben, den der Zusatzdienst bearbeiten soll.

Das `aggregate`-Element im Routingheader enthält eine Liste von `pathId`-Elementen, die Nachrichten bestimmen, die an diesem Knoten aggregiert werden sollen. Die Reihenfolge der `pathId`-Elemente bestimmt die Reihenfolge, in der die Nachrichten an den anwendungsspezifischen Aggregationsdienst übergeben werden. Der Aggregationsdienst selbst wird über das `service`-Attribut (nicht in der Grafik) des `aggregate`-Elementes in Form eines qualified Name (QName, vgl. [BM04]) bestimmt.

5.4. Beschreibung einer Route mit BPEL

Um die Besonderheit des SOAP-BPEL-Routings – die Parallelität – nutzen zu können, wird auf die Eigenschaften der Prozessbeschreibungssprache BPEL zurückgegriffen. Zum einen müssen die parallelen Zweige des Routings in BPEL geeignet abgebildet werden und zum anderen muss es eine Möglichkeit geben, jede einzelne Nachricht eindeutig einem Pfad zuweisen zu können. Dies ist nötig, damit die BPEL-Maschine bei einer Anfrage eines SOAP-Knoten auch die korrekte Antwort liefern kann.

Die eindeutige Zuordnung von Anfragen zu einer Prozessinstanz wird bei BPEL über die `correlationSets` ermöglicht. Die `correlationSets` lassen sich zudem noch dazu nutzen, Anfragen mit derselben Operation (`portType`, `partnerLink`) zu unterscheiden und entsprechend an die korrekten Aktivitäten innerhalb einer Prozessinstanz weiter zu leiten.

In diesem Abschnitt wird zunächst die Schnittstelle zum Routingprozess beschrieben, bevor im Anschluss daran die allgemeinen Eigenschaften eines Routingprozesses aufgezählt werden. Über diese Eigenschaften wird das Verhalten des Routingprozesses allgemeingültig festgelegt. Ein Beispielroutingprozess findet sich in Abschnitt 5.6.1.

5.4.1. Die WSDL-Schnittstelle zum Prozess

Um Anfragen an den Routingprozess zu ermöglichen, muss zunächst eine WSDL-Schnittstelle zum Prozess definiert werden. Darin werden sowohl die ein- als auch die ausgehenden Nachrichtenformate beschrieben. Das zugehörige WSDL-Dokument zeigt Listing A.1 im Anhang. Am Anfang des Dokuments werden Datentypen definiert, die von den Routingnachrichten verwendet werden.

Anfragen an den Prozess

Die Anfragen durch einen SOAP-Knoten werden mittels `RoutingRequest` Nachrichten (siehe Abb. 5.4) gestellt. Jede zu routende Nachricht besitzt eine eindeutige Identifikationsnummer `messageId` über die die zugehörige Prozessinstanz mit Hilfe eines CorrelationSets gefunden wird.

In flachen Routingszenarien, also bei rein sequentiellem Routing, würde die `messageId` zur Identifikation der Prozessinstanz und – da in einem solch einfachen Routingprozess immer nur ein `receive` aktiv ist – zur Bestimmung der Aktivität innerhalb der Instanz völlig ausreichen.

Anders verhält es sich jedoch, wenn im Routing Verzweigungen auftauchen – also eine Nachricht in mehrere Pfade kopiert wird. Jede der duplizierten Nachrichten besitzt dieselbe `messageId`, um das Auffinden des zugehörigen Routingprozesses zu ermöglichen. Dies reicht nun allerdings nicht mehr aus, um die Stelle innerhalb des Prozesses eindeutig zu finden,

für die eine ankommende Routinganfrage bestimmt ist, da es ja nun möglich ist, dass im Prozess an zwei Stellen zeitgleich ein `receive` auf dieselbe Operation aktiv ist.

Also muss ein weiteres Unterscheidungsmerkmal eingeführt werden, das auch bei mehrfacher Verzweigung noch eine eindeutige Zuordnung ermöglicht. Hierzu wurde das `pathId`-Element eingeführt, dem für jeden unterschiedlichen Pfad innerhalb eines Routingprozesses eine eindeutige Nummer zugewiesen werden muss. Dadurch kann innerhalb der Prozessinstanz mit Hilfe eines CorrelationSets über die `pathId` die korrekte Aktivität gefunden werden.

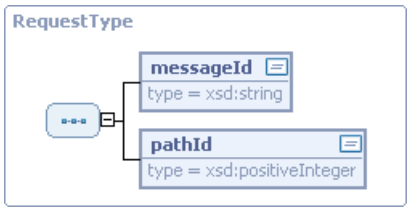

Abbildung 5.4.: Anfrage an den BPEL Prozess

Die Kombination aus `messageId` und `pathId` identifiziert somit zusammen mit dem Zustand des Prozesses den Status einer Nachricht eindeutig. Listing 5.1 zeigt eine Beispielanfrage an einen Routingprozess.

```
<RoutingRequest>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <pathId>1</pathId>
</RoutingRequest>
```

Listing 5.1: Beispielanfrage an den BPEL-Prozesses

Antwort vom Prozess

Die Antwort des Routingprozesses erfolgt in einer `RoutingResponse`-Nachricht (Struktur siehe Abb. 5.5) als synchrone Antwort auf die Anfrage eines SBR-Routers. In der Antwort des Routingprozesses müssen die Knoten aufgelistet sein, zu denen die Nachricht weitergeleitet werden soll. Für jeden Knoten sind hierfür der URI sowie optional eine Liste der an diesem Knoten zu bearbeitenden Zusatzdienste erforderlich.

Zusätzlich muss für jeden einzelnen Knoten vermerkt werden, welchen Prozess (`processURI`) der Knoten im Verlauf des Routings nach dem weiteren Vorgehen fragen soll. Die `pathId` ermöglicht eine Unterscheidung der verschiedenen Pfade innerhalb eines Routings.

Ein späteres Zusammenführen einzelner Pfade an Aggregationsknoten wird über die `aggregate` Liste in den Antworten des BPEL-Prozesses ermöglicht. Diese enthält die Nummern

(`pathId`) der Zweige, die an einem bestimmten Knoten wieder zu einer Nachricht vereint werden sollen. Hierbei gibt die Reihenfolge der Pfadnummern an, in welcher Reihenfolge die Nachrichten an den Aggregationsdienst des Knoten geliefert werden. Im Attribut `service` wird der Name des zu nutzenden Aggregationsdienstes angegeben.

Es ist die Aufgabe des Routing-Administrators, dafür zu sorgen, dass die Auflistung der Pfadnummern vollständig ist und mit der Struktur des Prozesses übereinstimmt – also alle für die Zusammenfassung nötigen Nachrichten wirklich zum Aggregationsknoten geroutet werden.

```
<RoutingResponse>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <routeTo>
    <node>
      <pathId>2</pathId>
      <nodeURI>http://node1.example.org:8081</nodeURI>
      <processURI>http://processes.example.org/proc1</processURI>
      <service>
        <serviceNamespace>http://www.example.org/Other</serviceNamespace>
        <rootElement>someservice</rootElement>
      </service>
      <service>
        <serviceNamespace>http://www.example.org/Secure</serviceNamespace>
        <rootElement>encryption</rootElement>
      </service>
    </node>
    <node>
      <pathId>3</pathId>
      <nodeURI>http://node2.example.org:8078</nodeURI>
      <processURI>http://processes.example.org/proc1</processURI>
      <service>
        <serviceNamespace>http://www.example.org/Log</serviceNamespace>
        <rootElement>logging</rootElement>
      </service>
    </node>
  </routeTo>
</RoutingResponse>
```

Listing 5.2: Beispielantwort des Routingprozesses

In Listing 5.2 ist eine einfache Antwort eines Routing-Prozesses zu finden. Diese weist den anfragenden Knoten an, die Nachricht mit der `messageId` *33ea4f-d5eg41-ab4ca5-5efa3b-7cd901* an zwei SBR-Router weiterzuleiten:

- Am Router *node1* wird die Nachricht unter der `pathId` *2* geführt und soll dort von zwei Zusatzdiensten *Other* und *Secure* bearbeitet werden, bevor der Prozess *proc1* nach dem weiteren Vorgehen befragt werden soll.

- Am Router *node2* erhält die Nachricht die `pathId` *3* und wird dort von einem Zusatzdienst *Log* bearbeitet.

An keinem der beiden SBR-Router soll in diesem Fall eine Aggregation stattfinden.

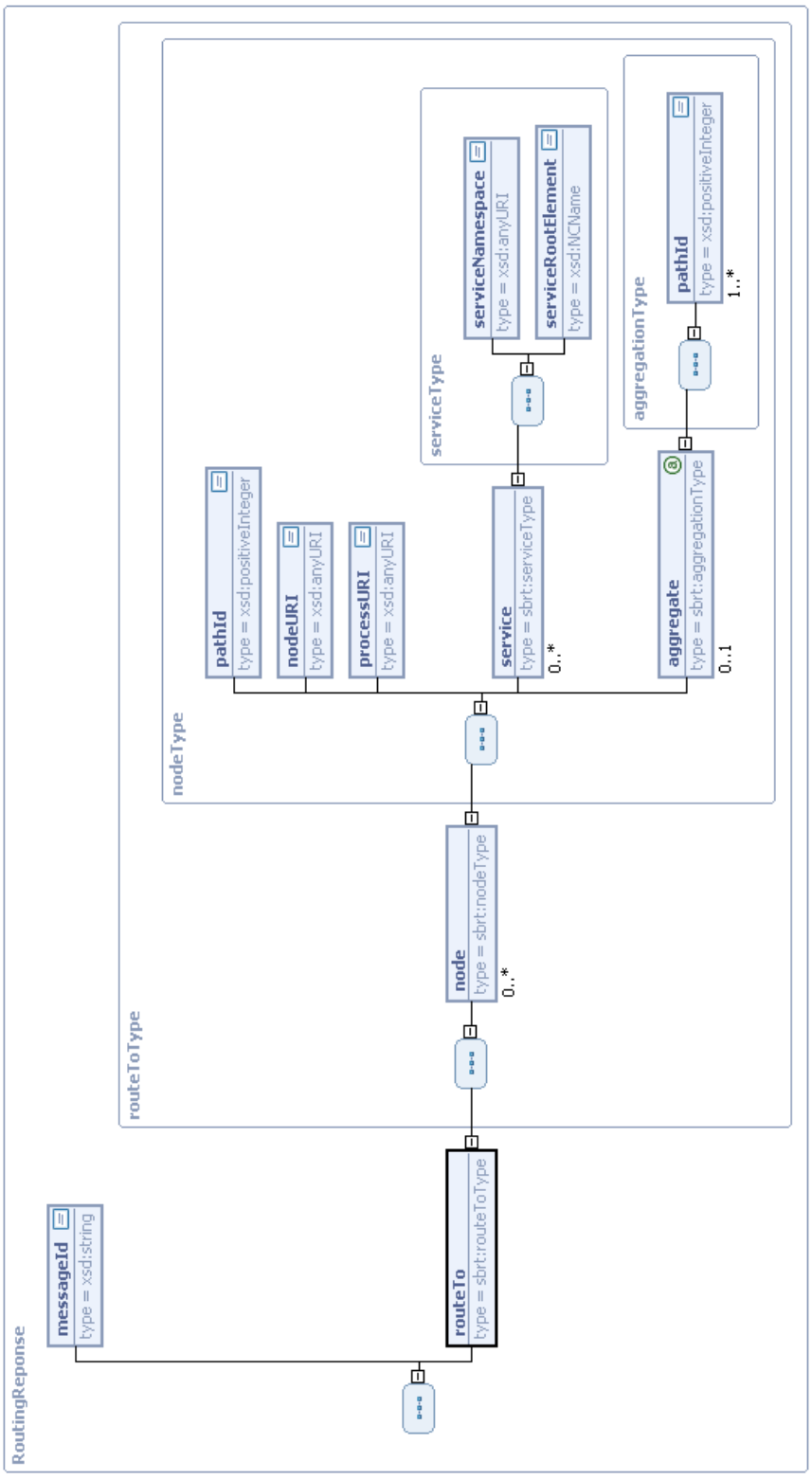

Abbildung 5.5.: Antwort vom BPEL Prozess

5.4.2. Eigenschaften des Routingprozesses

Eine Festlegung, wie der Routingprozess in BPEL genau auszusehen hat, wird im Rahmen dieser Arbeit bewusst nicht getroffen, um keine unnötigen Einschränkungen einzuführen. Es werden lediglich die von aussen beobachtbaren Eigenschaften – d. h. die erwarteten Rückgabewerte auf eine Anfrage – eines solchen Prozesses festgelegt. Im Anschluss an die generelle Beschreibung der Vorgänge beim Routing wird in Abschnitt 5.6 ein Beispielrouting mit zugehörigem Routingprozess vorgestellt.

Ein Routingprozess besitzt die in den vorangehenden Seiten beschriebene Schnittstelle. In einer Antwort auf eine Anfrage (Struktur der Antwort siehe Abb. 5.5) hat der BPEL-Prozess die Teile der Antwortnachricht folgendermaßen zu bestimmen:

1. `messageId` muss die Nummer der Nachricht so wie in der Anfrage enthalten. Dieses Feld MUSS ausgefüllt sein.

2. `routeTo` enthält eine Liste mit SBR-Routern, an die die Nachricht weitergeleitet werden soll. Enthält die Liste keine `node`-Elemente – ist `routeTo` also leer – so ist der anfragende SBR-Router der Ultimate Receipient der Nachricht.

3. `node` enthält Informationen über einen SBR-Router zu dem die Nachricht als nächstes weitergeleitet werden soll. Die Informationen des `node`-Elementes sind im Folgenden aufgezählt.

   a) `pathId` gibt die Nummer des Nachrichtenpfades an, zu dem dieser Router gehört. Findet keine Aufteilung statt – geht die Nachricht also nur an diesen einen SBR-Router weiter – so enthält die `pathId` denselben Wert wie das entsprechende Feld der Anfrage.

      Andernfalls muss hier für jeden `node` in der Liste eine eindeutige (diese darf auch nicht in anderen parallel laufenden Nachrichtenpfaden auftauchen) neue Nummer vergeben werden. Dieses Feld MUSS ausgefüllt sein.

   b) `nodeURI` enthält die Adresse des SBR-Routers. Dieses Feld MUSS ausgefüllt sein und einen SBR-Router eindeutig referenzieren.

   c) `processURI` enthält die Adresse des für das weitere Routing der Nachricht zuständigen Routingprozesses. Hiermit kann nach einem Routingschritt der Prozess für die weitere Verarbeitung der Nachricht durch den Routingprozess selbst bestimmt werden.

      Dieses Feld MUSS ausgefüllt sein und enthält im Normalfall die Adresse des aktuellen Prozesses. Soll jedoch ein anderer Routingprozess für den betreffenden SBR-Router zuständig sein, so wird dessen URI hier eingetragen. Es muss jedoch der URI eines gültigen Routingprozesses sein.

   d) `service` enthält eine oder mehrere Zusatzdienstbeschreibungen. Hierüber wird

festgelegt, welche Header in welcher Reihenfolge an dem betreffenden SBR-Router zu bearbeiten sind. Ein Zusatzdienst ist der für einen bestimmten Headerblock einer SOAP-Nachricht zuständige Dienst. Die in `service` enthaltenen Werte beziehen sich also auf die anderen in der SOAP-Nachricht enthaltenen Headerblöcke. Jeder Zusatzdienst ist eindeutig gekennzeichnet durch die Kombination aus `serviceNamespace` und `rootElement`. Diese geben zum einen den XML-Namensraum des Headerblockes und zum anderen den Namen des Hauptelementes wieder.

Die spätere Abarbeitungsreihenfolge im SBR-Router entspricht der Reihenfolge der `service`-Elemente der Antwort des Routingprozesses. Sollen keine Zusatzdienste ausgeführt werden, so entfallen die `service`-Elemente.

e) `aggregate` enthält eine Liste von `pathId`-Elementen, welche die Pfadnummern der am zugehörigen SBR-Router zusammenzufassenden Nachrichten enthält. Hier ist sicher zu stellen, dass alle aufgezählten Nachrichten im Verlauf des Routings zu diesem Router geleitet werden, da hier beim Fehlen einer Nachricht ein SOAPFault erzeugt wird.

   Zudem muss jede der zu aggregierenden Nachrichten den exakt selben Inhalt im `aggregate`-Element aufweisen, da die Reihenfolge der `pathId`-Elemente die Reihenfolge bestimmt, in der die ankommenden Nachrichten an den Aggregationsdienst geliefert werden.

   Mit Hilfe des zu `aggregate` gehörenden Attributes `service` wird der zu verwendende Aggregationsdienst bestimmt. Die `processURI` aller zu aggregierenden Nachrichten muss identisch sein. Soll an dem betreffenden SBR-Router keine Aggregation stattfinden, so taucht dieses Element nicht auf.

Beim Entwurf der Route ist darauf zu achten, dass es immer nur genau einen Ultimate Receipient geben darf. Das Verhalten bei mehreren Ultimate Receipients ist nicht definiert – es wird aber auch nicht unterbunden. Auswirkungen hiervon sind möglicherweise unangeforderte Nachrichten, die an einem Web Service ankommen, doppelte Nachrichten und Inkonsistenzen im Geschäftsablauf. Bei Erreichen des Ultimate Receipient darf kein weiterer Pfad des Routings aktiv sein.

Ein Routingprozess muss also sicherstellen, dass die unterschiedlichen Pfade im Routing zusammengeführt werden und anschliessend nur ein Pfad zum Ultimate Receipient geleitet wird. D. h. es ist beim Entwurf des Prozesses darauf zu achten, dass nur genau eine Nachricht zum Ultimate Receipient weitergeleitet wird und es keine Sackgassen (=Pfade, die nicht zum Ultimate Receipient führen) im Prozess gibt. Dies wird durch die Routingerweiterung auf den SBR-Routern jedoch nicht geprüft und kann dort auch nicht geprüft werden, da das weitere Routing lediglich im Routingprozess bekannt ist.

5.5. Die Abläufe des Routings im Detail

Wichtig für die korrekte Abarbeitung der Routinginformationen ist zunächst, dass die Routingerweiterung auf den jeweiligen Intermediaries als erstes ausgeführt wird und dort dann die Kontrolle über eventuell vorhandene WS-Erweiterungen übernimmt – sprich deren Abarbeitungsreihenfolge bestimmt. Es wird empfohlen, andere möglicherweise installierten Erweiterungen nicht automatisch abarbeiten zu lassen, sondern diese explizit in das SOAP-BPEL-Routing-Umfeld zu integrieren. Hierdurch werden eventuell denkbare Probleme wie zum Beispiel eine Verschlüsselung der Nachricht ohne Wissen des Routings mit der Folge eines kryptischen Logfiles bereits im Vorfeld ausgeschlossen.

Um jedoch die Konformität zum in Kapitel 3 vorgestellten Verarbeitungsmodell von SOAP zu gewährleisten, muss auch ein SBR-Intermediary nach Ausführung der SBR-Erweiterung zunächst die an ihn gerichteten (`role`) und als `mustUnderstand` gekennzeichneten Headerblöcke bearbeiten, bevor die Nachricht weitergeleitet werden darf. Um bereits im Vorfeld unerwünschte Nebeneffekte zu unterbinden, wird empfohlen, keine Header als `mustUnderstand` zu kennzeichnen, sondern solche Header explizit in das Routing zu integrieren. Hierdurch wird derselbe Effekt erzielt mit dem Unterschied, dass die Reihenfolge der Abarbeitung fest vorgegeben wird und somit keine unerwünschten Nebeneffekte mehr auftreten können.

Die Kommunikation zwischen den SOAP-BPEL-Routern und dem Routingprozess erfolgt direkt über Webserviceaufrufe, da jeder Routingprozess von aussen als einfacher Web Service über die in Abschnitt 5.4.1 definierte Schnittstelle ansprechbar ist.

Im Folgenden werden die zur Nutzung von SOAP-BPEL-Routing nötigen Schritte im Detail erläutert. Ein Überblick der bei der Abarbeitung notwendigen Schritte findet sich in Abbildung 5.6. Die im Verlauf des Routings unter Umständen auftretenden Fehlerfälle werden separat in Abschnitt 5.5.7 „Fehlerbehandlung“ aufgeführt.

Zunächst werden die für das Erzeugen eines gültigen SBR-Headerblocks nötigen Daten beschrieben, bevor im Anschluss daran die Verarbeitung eines Headerblocks an einem SBR-Router geschildert wird. Im Verlauf dieser Verarbeitung kommuniziert der SBR-Router mit dem BPEL-Prozess wie in Abschnitt 5.5.4 beschrieben. Die Weiterleitung von Nachrichten und dort insbesondere die Formulierung eines neuen SBR-Headerblocks wird in Abschnitt 5.5.5 abgehandelt. Der Sonderfall der Aggregaton mehrerer Nachrichtenpfade an einem SBR-Router findet sich im Anschluss daran in Abschnitt 5.5.3.

Ein ausführliches Beispiel zu den einzelnen Vorgängen folgt am Ende dieses Kapitels in 5.6.

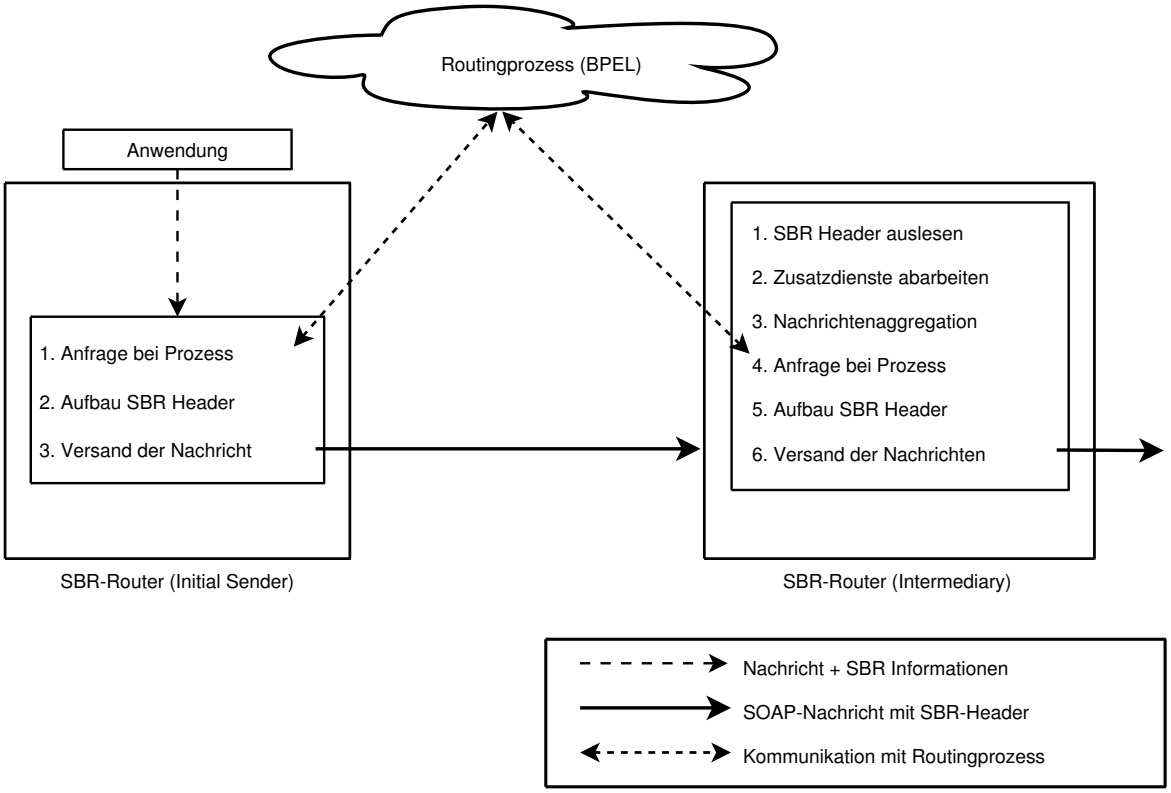

Abbildung 5.6.: Details des SOAP-BPEL-Routings

5.5.1. Erzeugen eines SBR-Headers

Wie bereits in Abschnitt 5.3.4 beschrieben, besteht ein SBR-Header aus mehreren Elementen, die von einem potentiellen Nutzer zunächst ausgefüllt werden müssen. Wichtig ist hierbei, dass die Elemente für die `messageId` sowie die in `node` enthaltenen Elemente `pathId`, `nodeURI` und `processURI` mit gültigen Werten versehen sind. Sollte ein Request-Response Nachrichteninteraktionsmuster vorliegen, so muss zusätzlich das Element `replyTo` ausgefüllt sein. Wenn Fehlermeldungen empfangen werden sollen, so ist das Element `faultTo` zu setzen.

Vom Absender der mit einem SBR-Header versehenen Nachricht ist sicherzustellen, dass das Element `messageId` einen global eindeutigen Wert enthält. Wie solche eindeutigen Werte erzeugt werden ist nicht Bestandteil der SBR-Spezifikation. Zur Erzeugung von UUIDs siehe zum Beispiel [wik06]. UUIDs sind im RFC 4122 der Internet Engineering Taskforce spezifiziert [LMS05].

Die `pathId` muss bei Initialisierung des Routings – sprich in der ersten gesendeten Nachricht – auf *1* gesetzt sein.

Mit den beiden Werten `messageId` und `pathId` erfragt der Initial Sender beim Routing-

prozess die für den ersten Routingschritt erforderlichen Werte. Aus der Antwort erhält er die Daten der Router, zu denen er die Nachricht schicken muss.

Der im Element `nodeURI` enthaltene URI bestimmt den nächsten SOAP-Knoten. Es ist darauf zu achten, dass die Nachricht im Anschluss auch tatsächlich an diesen Knoten gesendet wird, da beim Empfänger das Element `nodeURI` geprüft wird.

Da über den `processURI` die Kommunikation mit dem Routingprozess abläuft, muss dieses Element auf einen gültigen URI eines solchen Prozesses verweisen.

Der empfangende SBR-Router benötigt diese Informationen, um sinnvoll mit dem Prozess interagieren zu können. Zum einen kann der Knoten mit den Informationen prüfen, ob er die Nachricht korrekterweise erhalten hat (`nodeURI`) und zum anderen sind die `messageId` und `pathId` sowie der `processURI` für eine erfolgreiche Kommunikation mit dem Routingprozess zwingend erforderlich.

Mit Hilfe der optional enthaltenen `service`-Elemente werden die Zusatzdienste bestimmt, die am jeweiligen Router zu bearbeiten sind.

5.5.2. Verarbeitung an einem SBR-Router

Wenn eine Nachricht mit SBR-Header an einem Router ankommt, so muss dieser mehrere Aktionen in vorgegebener Reihenfolge durchführen:

1. Identifikation des Routing-Headers und Auslesen der Informationen.
2. Abarbeitung der im Routing-Header genannten Zusatzdienste in der im Header gegebenen Reihenfolge.
3. eventuelle Aggregation von mehreren Nachrichten (siehe Abschnitt 5.5.3).
4. Anfrage nach neuen Zielknoten bei dem in der Nachricht angegebenen Prozess.
5. Auswertung der Antwort und Erstellung von SBR-Headern für an weitere Router ausgehende Nachrichten.
6. Senden der Nachricht(en) an den bzw. die weiteren SOAP-Knoten.

An jedem SBR-Router wird zunächst an Hand des Namespace[2] nach vorhandener Routing-Erweiterung gesucht. Wird ein solcher Headerblock mit dem Namen `RoutingInfo` gefunden, so werden die darin enthaltenen Informationen ausgewertet. Dabei wird eine Liste der zu bearbeitenden Headerblöcke erstellt und die zu den in den `service`-Elementen genannten Zusatzdiensten passenden Module in angegebener Reihenfolge aufgerufen. Wie dieser Aufruf genau aussieht – ob es sich um einen Webserviceaufruf oder lokale Methodenaufrufe handelt – ist implementierungsabhängig. Wichtig ist hierbei, dass die im Routing genannten Zusatzdienste ausgeführt werden müssen – d. h. eine Nennung des Zusatzdienstes im

[2] `urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08`

Routing entspricht diesbezüglich dem Setzen des `mustUnderstand` im dem Zusatzdienst zugehörigen Headerblock bei SOAP-Nachrichten ohne SBR-Erweiterung.

Im Anschluss an die Abarbeitung der Header sammelt der SBR-Router gegebenenfalls alle Nachrichten, die in `aggregate` angegeben sind und erfragt im Anschluss beim BPEL-Prozess die nächsten SOAP-Knoten, zu denen die Nachricht weitergeleitet werden muss. Die Antwort des Prozesses wird umgehend ausgewertet, passende Headerblöcke generiert und die Nachricht an die angegebenen SOAP-Knoten gesendet.

5.5.3. Aggregation von Nachrichten

Die SBR-Erweiterung bietet die Möglichkeit, eine Nachricht in mehrere Nachrichten zu kopieren und über unterschiedliche Pfade zu leiten. Damit wird die parallele Abarbeitung von Headerblöcken an unterschiedlichen Intermediaries ermöglicht. Um die so vervielfachten Nachrichten wieder zusammenführen zu können – schließlich darf am Ultimate Receipient nur eine Nachricht ankommen – wurde das Aggregation-Pattern [HW03] angewendet. Jeder SBR-Router kann als Aggregator dienen. Voraussetzung hierfür ist die zusätzliche Installation eines Aggregationsdienstes an dem betreffenden SBR-Router, der die anwendungsspezifische Logik für das Zusammenführen mehrerer Nachrichten in eine Nachricht enthält. Ein SBR-Router muss hierfür eine passende Schnittstelle bieten.

Um feststellen zu können, welche Nachrichten zusammen gehören, wird diese Information am letzten Knoten jedes Pfades vor dem Aggregationsknoten in den Header jeder der zusammenzufassenden Nachrichten eingefügt. Das so entstandene `aggregate`-Element enthält eine Liste von `pathId`-Elementen, welche die Nachrichten der einzelnen Pfade eindeutig kennzeichnen. Es ist wichtig, dass die Reihenfolge der `pathId`-Elemente an jeder Nachricht identisch ist, da sonst nicht mehr klar geregelt ist, in welcher Reihenfolge die Nachrichten an den Aggregationsdienst weitergereicht werden müssen. Diese Informationen werden vom Routingprozess vorgegeben und von den Routern in den Header der Nachricht kopiert. Es ist also keine komplizierte Logik in den SBR-Routern erforderlich, um die `aggregate` Liste korrekt auszufüllen.

Der Aggregationsknoten muss die zusammengehörenden Nachrichten sammeln und mit den Nachrichten in der richtigen Reihenfolge den Aggregationsdienst aufrufen, von dem er dann die zusammengesetzte Nachricht erhält. Die für diese Nachricht vergebene `pathId` muss die erste `pathId` des in den ursprünglichen Nachrichten enthaltenen `aggregate`-Elements sein.

Erst nachdem diese Nachricht erhalten wurde, kontaktiert der SBR-Router mit den darin enthaltenen Informationen (`messageId` und `pathId`) den Routingprozess, um bestimmen zu können, wohin die aggregierte Nachricht geroutet werden soll.

Die Möglichkeit der einfachen Zusammenführung von mehreren Pfaden ohne Aggregation ist nicht direkt vorgesehen, lässt sich aber durch Implementierung eines geeigneten

Aggregationsdienstes leicht nachbilden. Ein solcher Aggregationsdienst liefert einfach die Nachricht eines der Pfade als aggregierte Nachricht zurück, ohne tatsächlich etwas zu berechnen.

5.5.4. Kommunikation mit dem BPEL-Prozess

Nach Abarbeitung der Zusatzdienste am SBR-Router sowie einer eventuellen Aggregation fragt der Router mittels `RoutingRequest`-Nachricht den Routingprozess nach dem weiteren Vorgehen. Der Request enthält die Nachrichtennummer der momentan in Bearbeitung befindlichen Nachricht sowie die zugehörige Pfadidentifikationsnummer.

Daraufhin liefert der Routingprozess in einer `RoutingResponse`-Nachricht die Daten des bzw. der nächsten SBR-Router. Darin ist vermerkt, welcher Pfadnummer der einzelne Router angehört, welche Zusatzdienste an dem Router auf der Nachricht ausgeführt werden sollen und gegebenenfalls welche Nachrichten dort zusammengefasst werden sollen.

Bei der Antwort des Routingprozesses kann man drei Fälle unterscheiden:

1. Es soll zu mehreren SBR-Routern weitergeleitet werden. Dann gelten für jedes `node`-Element dieselben Regeln wie beim Fall des einzelnen Zielknoten. Zusätzlich muss jedoch sichergestellt sein, dass das `pathId`-Element eines jeden `node` eindeutig ist. Keiner der angegebenen SBR-Router darf der Ultimate Receipient sein, da am Ende des Routings immer nur genau ein Pfad aktiv sein darf.

2. Es gibt genau einen SBR-Router zu dem weiter geroutet werden soll. In diesem Fall werden die Elemente `pathId`, `processURI` und `nodeURI` gesetzt. `Service` gibt optional die am nächsten SBR-Router zu bearbeitenden Zusatzdienste an. Über `aggregate` wird das Zusammenführen mehrerer Pfade am nächsten SBR-Router ermöglicht.

   Sollte der in `nodeURI` angegebene Router der Ultimate Receipient einer Nachricht sein, so ist darauf zu achten, dass entweder genau ein Pfad zum Ultimate Receipient führt oder dass die zum Ultimate Receipient führenden Pfade dort über `aggregate` zusammengeführt werden und es keine weiteren Pfade im Routingprozess gibt.

3. Der anfragende SBR-Router ist bereits der Ultimate Receipient. In diesem Fall enthält die Antwort des Routingprozesses lediglich die Nachrichtennummer sowie ein leeres `routeTo`-Element.

5.5.5. Weiterleitung der Nachrichten

Im Anschluss an die Kommunikation mit dem Routingprozess wird die Antwort im SBR-Router ausgewertet und daraus neue SBR-Headerblöcke für die ausgehenden Nachrichten generiert.

Um einen gültigen SBR-Headerblock für eine ausgehende Nachricht zu erstellen, werden die Werte aus dem entsprechenden `node`-Element der Antwort des Routingprozesses in den neuen Headerblock kopiert. Das `messageId`-Element sowie die gesamte `routeTo`/`node`-Struktur wird nach `RoutingInfo`/ kopiert. Die Werte für `replyTo`, `faultTo` und `relatesTo` werden aus der ursprünglich am Router eingetroffenen Nachricht übernommen. Damit ist schon ein gültiger und vollständiger SBR-Headerblock entstanden und die Nachricht kann an den in `node`/`nodeURI` angegebenen SBR-Router weitergeleitet werden.

5.5.6. Bearbeitung am Ultimate Receipient

Wenn die Nachricht am Ultimate Receipient eintrifft, werden zunächst wie an jedem anderen SBR-Router die im Header angegebenen Zusatzdienste ausgeführt. Anschliessend wird der Routingprozess kontaktiert und antwortet mit einem leeren `routeTo`-Element. Daraufhin wird der zuständige Web Service aufgerufen und die Nachricht bearbeitet. Eine eventuelle Antwort wird an den im `replyTo` genannten URI gesendet.

5.5.7. Fehlerbehandlung

Beim Entwurf des Routingverfahrens wurde von einer fehlerfreien Umgebung ausgegangen und ein in einer solchen Umgebung funktionstüchtiges Routing entwickelt. In der Realität kommt es durchaus vor, dass Nachrichten verloren gehen, Web Services nicht erreichbar oder Intermediaries nicht verfügbar sind. In jedem der im Vorangegangenen beschriebenen Schritte des Routings können daher Fehler auftreten. Wie auf einige dieser Fehler reagiert wird, ob ein Fehler keine Reaktion erfordert oder ob eine Reaktion mit Hilfe der in dieser Arbeit vorgestellten Mechanismen nicht möglich ist, soll in diesem Abschnitt beschrieben werden.

Bedingt durch die mögliche Parallelität des Routings kann es vorkommen, dass an unterschiedlichen Stellen des Routings ein SOAPFault erzeugt und an den Absender der Nachricht geschickt wird. Dieser muss nur eine der Fehlermeldungen interpretieren und kann weitere SOAPFaults als Duplikate verwerfen – selbst wenn der Grund des Fehlschlags unterschiedlich war. Wichtig ist lediglich, dass ein Fehler im Routing auch gemeldet wird. Eine Meldung eines Fehlers ist jedoch nur möglich, wenn in der vom Initial Sender versandten Nachricht das Element `faultTo` enthalten ist. In jedem Fall wird die weitere Verarbeitung des Pfades des Routings, in dem der Fehler auftrat, abgebrochen.

Fehler im Routingprozess

Bei Auftreten eines Fehlers innerhalb des Routingprozesses – also bei fehlerhafter Definition des Routings durch den Administrator – werden im Allgemeinen keine Fehlermeldungen ausgelöst, da derartige logische Routingfehler nicht automatisiert erkannt werden können.

Durch fehlerhafte Definition des Routingprozesses können unter Umständen Prozessinstanzen terminiert werden, ohne dass das Routing der Nachricht abgeschlossen ist und somit erzeugen erneute Anfragen eine neue Instanz, in der dann das Routing von vorne beginnt. Solche Fehler sind seitens der SBR-Router nicht zu erkennen.

Dies lässt sich beispielsweise durch Versetzen des Prozesses in einen speziellen Zustand verhindern, so dass ankommende Nachrichten mit derselben `messageId` immer eine SOAP-Fehlermeldung erhalten. Wie eine solche Lösung im einzelnen aussieht hängt vom konkreten Routingprozess ab. Wenn ein Routingprozess korrekt definiert wurde, treten solche Fehler nicht auf und eine Fehlerbehandlung im Prozess ist deshalb im Normalfall nicht nötig.

Von den SBR-Routern automatisch erkannt werden inkonsistente Werte innerhalb einer Antwort wie z. B. die mehrfache Vergabe einer `pathId` an mehrere Zielknoten oder derselbe URI in mehreren Zielknoten in der Antwort. In diesen Fällen wird ein SOAPFault erzeugt und der Administrator sollte den fehlerhaften Routingprozess überarbeiten.

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
                  xmlns:xml="http://www.w3.org/XML/1998/namespace"
                  xmlns:m="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08">
  <soapenv:Body>
    <soapenv:Fault>
      <soapenv:Code>
        <soapenv:Value>env:Receiver</env:Value>
        <soapenv:Subcode>
          <soapenv:Value>m:ProcessFailure</soapenv:Value>
        </soapenv:Subcode>
      </soapenv:Code>
      <soapenv:Reason>
        <soapenv:Text xml:lang="en">
          Routingprocess failed! Please inform administrator!
        </soapenv:Text>
      </soapenv:Reason>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

In den weiteren Abschnitten werden die SOAPFaults nicht mehr vollständig dargestellt, sondern lediglich `Code`, `Subcode` und `Reason` aufgelistet.

Fehler bei der Verarbeitung am SBR-Router

Bei der Suche nach dem zu einem bestimmten Headerblock gehörenden Zusatzdienst kann es vorkommen, dass kein passender Zusatzdienst gefunden wird. In diesem Fall wird ein

SOAPFault generiert, da jeder im Routing angegebene Headerblock abgearbeitet werden muss.

```
<soapenv:Code>
  <soapenv:Value>env:MustUnderstand</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:MissingService</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    specified service is missing
  </soapenv:Text>
</soapenv:Reason>
```

Zusätzlich sollten in diesem Fall im `soapenv:Header` diejenigen Headerblöcke angegeben werden, für die kein Zusatzdienst gefunden werden konnte.

Fehler bei der Aggregation

Im Falle der Aggregation können mehrere Fehlerfälle auftreten: zum einen kann der für die Aggregation zuständige Dienst auf dem SBR-Router möglicherweise nicht aufgefunden werden. Zum anderen ist es möglich und denkbar, dass Nachrichten unterwegs verloren gehen und somit nicht alle für eine erfolgreiche Aggregation notwendigen Nachrichten am Aggregationsknoten eintreffen. In beiden Fällen muss ein SOAPFault den Absender über den Fehlschlag informieren.

```
<soapenv:Code>
  <soapenv:Value>env:MustUnderstand</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:AggregationFailure</soapenv:Value>
  </soapenv:Subcode>
  <soapenv:Subcode>
    <soapenv:Value>m:AggregationServiceNotFound</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    aggregation of messages failed: aggregation service not found
  </soapenv:Text>
</soapenv:Reason>
```

Da bei einem Routing einer Nachricht im Gegensatz zu Geschäftsprozessen, die mitunter mehrere Tage andauern können, im Regelfall nur relativ kurze Verzögerungen auf Grund

der Verarbeitung eines Headerblocks vorkommen, kann zur Erkennung von fehlenden Nachrichten ein Timeout-Mechanismus eingeführt werden. Wenn nach dem Eintreffen der ersten zu aggregierenden Nachricht mehr als der als Timeout festgelegte Wert vergeht und immer noch nicht alle zugehörigen Nachrichten angekommen sind, so wird ein SOAPFault generiert. Der Timeout wird pro SBR-Router durch den Administrator festgelegt. Wie der Timeout-Mechanismus konkret aussieht und welche Timeout-Werte gesetzt werden können, hängt von der jeweiligen Implementierung ab.

```
<soapenv:Code>
  <soapenv:Value>env:MustUnderstand</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:AggregationFailure</soapenv:Value>
  </soapenv:Subcode>
  <soapenv:Subcode>
    <soapenv:Value>m:AggregationMessagesMissing</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    aggregation of messages failed: messages missing
  </soapenv:Text>
</soapenv:Reason>
```

Später eintreffende Nachrichten mit derselben `messageId` werden in diesem Fall ohne weitere Rückmeldung verworfen.

Fehler bei der Kommunikation mit dem Routingprozess

Sollte ein SBR-Router nicht mit dem zu einer Nachricht gehörenden Prozess kommunizieren können, so werden mehrere Wiederholungsversuche unternommen, bevor ein SOAPFault das Fehlschlagen der Verbindung zum Routingprozess an den Absender der Nachricht mitteilt:

```
<soapenv:Code>
  <soapenv:Value>env:Receiver</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:ProcessTimeout</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    contacting routingprocess failed
  </soapenv:Text>
```

```
</soapenv:Reason>
```

Bei fehlerhaften Antwortnachrichten des Prozesses (beispielsweise mehrfache Vergabe derselben `pathId`) wird ebenfalls ein SOAPFault generiert:

```
<soapenv:Code>
  <soapenv:Value>env:Receiver</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:ProcessFailure</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    Routingprocess failed! Please inform administrator!
  </soapenv:Text>
</soapenv:Reason>
```

Fehler bei der Weiterleitung der Nachrichten

Sollte ein SBR-Router keine Verbindung zu einem nachfolgenden Knoten aufnehmen können, so wird dies wie im Falle der Kommunikation mit dem Routingprozess mehrfach wiederholt. Sollte auch dann keine Kommunikation erfolgen können, so wird ein SOAPFault erzeugt.

```
<soapenv:Code>
  <soapenv:Value>env:Receiver</env:Value>
  <soapenv:Subcode>
    <soapenv:Value>m:RoutingFailure</soapenv:Value>
  </soapenv:Subcode>
</soapenv:Code>
<soapenv:Reason>
  <soapenv:Text xml:lang="en">
    contacting next node failed
  </soapenv:Text>
</soapenv:Reason>
```

Weitere mögliche Fehler

Eine ausführliche Besprechung von Problemen und möglichen Umgehungslösungen findet sich in Kapitel 7. Potentielle Erweiterungen des in dieser Arbeit vorgestellten Verfahrens sind im letzten Kapitel der Ausarbeitung 8 unter „Ausblick“ zu finden.

5.6. Ein Beispiel

Im Folgenden wird ein Routing vorgestellt das eine Nachricht über zwei Pfade leitet wie es in Abb. 5.7 dargestellt ist. Dieses Beispiel kann als Modellierungshilfe bei der Erstellung von eigenen Routingprozessen dienen.

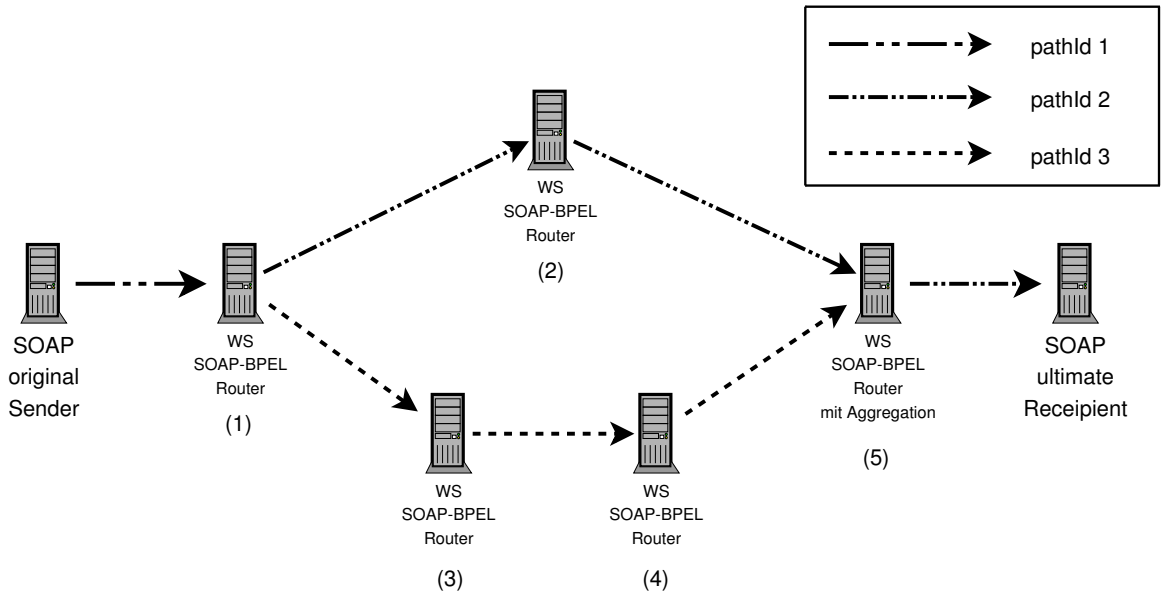

Abbildung 5.7.: Überblick Beispielrouting

5.6.1. Der Routingprozess

Beim Entwurf des Routingprozesses müssen zunächst die Antworten festgelegt werden, die bei Anfragen von den einzelnen Knoten gegeben werden sollen:

- Die Antwort an Router (1) muss die Adressen der Router (2) und (3) mit den jeweiligen Pfadnummern *2* und *3* enthalten.
- Die Antwort an Router (2) muss die Adresse von Router (5) enthalten sowie die Aggregationsliste mit den Pfadnummern *2* und *3*. Um später den richtigen Aggregationsdienst aufzurufen, muss zusätzlich das Attribut `service` auf den QName gesetzt sein, der den für diese Aggregation zuständigen Aggregationsdienst eindeutig bestimmt.
- Die Antwort an Router (3) muss die Adresse von Router (4) enthalten. Die `pathId` des `node`-Elementes muss *3* sein.

- Die Antwort an Router (4) muss – mit Ausnahme der Zusatzdienstdefinitionen sowie der im `node`-Element enthaltenen `pathId` (hier *3*)– dieselbe sein wie die Antwort an Router (2). Insbesondere muss das `aggregate`-Element identisch sein.
- Die Antwort an Router (5) muss die Adresse des Ultimate Receipients enthalten und die `pathId` ist *2*.

Bei allen Antworten ist die Nachrichtennummer `messageId` gleich der ursprünglichen Nummer. Die in den einzelnen `node`-Elementen enthaltenen `service`-Elemente sind in Abhängigkeit von den auszuführenden Zusatzdiensten zu bestimmen.

Die Angabe der `processURI` hängt davon ab, wie der Prozess entworfen wird: wird das gesamte Routing in einem Prozess abgebildet, so wird hier immer die Adresse dieses Prozesses zurückgeliefert. Soll das Routing auf mehrere Prozesse verteilt werden, so muss an den entsprechenden Stellen immer die URI des nächsten Prozesses geliefert werden. In diesem Beispiel wird das Routing mit einem einzelnen Prozess beschrieben.

Eine Beispielimplementierung

In diesem Abschnitt wird das oben skizzierte Routing konkret in einen BPEL-Prozess abgebildet. Es wird an dieser Stelle nochmals darauf hingewiesen, dass diese Realisierung lediglich eine von vielen Möglichkeiten darstellt, ein Routing in einem BPEL-Prozess zu beschreiben und es keineswegs verpflichtend ist, sich nach den in diesem Abschnitt dargelegten Prozess zu richten. Einzig die oben definierten Antworten auf die Anfragen sind verpflichtend, um das oben skizzierte Routing korrekt zu implementieren.

In der hier gewählten Variante repräsentieren die einzelnen `sequence`s im globalen `flow` des Prozesses (siehe Abb. 5.8) jeweils die Bearbeitung einer Anfrage des zugehörigen SBR-Routers. Durch Einblick in den Zustand des Routingprozesses kann hiermit der Status der einzelnen Nachrichtenpfade überprüft werden: wenn ein `receive` aktiv ist, so wurde die Nachricht am entsprechenden SBR-Router noch nicht vollständig bearbeitet. Ist eine `sequence` komplett bearbeitet, so kann davon ausgegangen werden, dass der zugehörige Router die Nachricht komplett bearbeitet hat und sie nun an den nächsten Router weiterleitet.

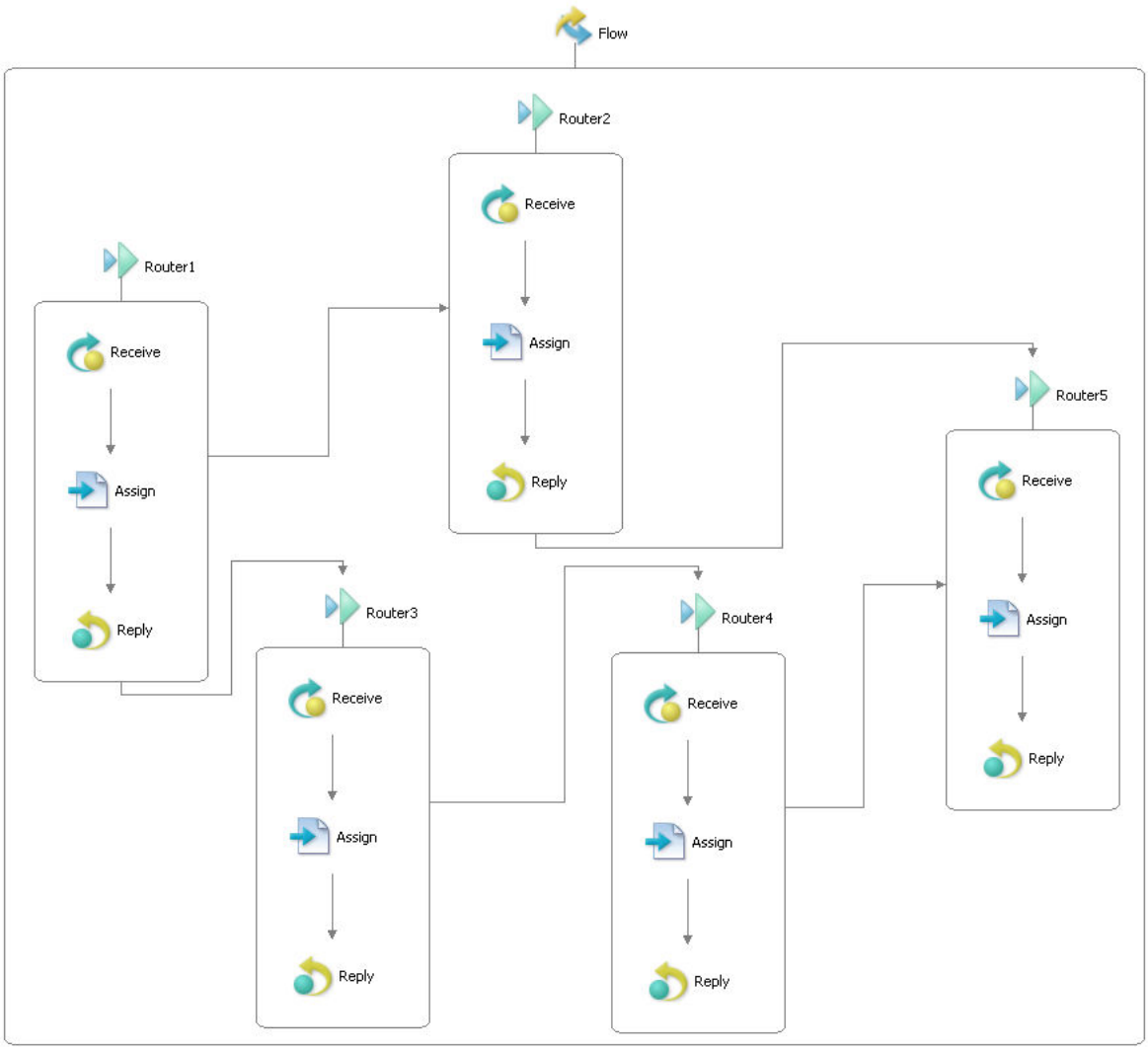

Abbildung 5.8.: Überblick über den Routingprozess

PartnerlinkTypes und PartnerLinks existieren in diesem Prozess nur jeweils einer, da der Prozess unabhängig vom anfragenden SBR-Router immer dieselbe Rolle einnimmt.

```
<plnk:partnerLinkType name="SOAPtoBPEL_PL">
   <plnk:role name="routingservice">
      <plnk:portType name="ns1:RoutingServicePortType"/>
   </plnk:role>
</plnk:partnerLinkType>
```

```
<partnerLinks>
  <partnerLink myRole="routingservice" name="RouterToBPEL-PL"
               partnerLinkType="route1:SOAPtoBPEL_PL"/>
</partnerLinks>
```

CorrelationSets und Properties sind für das Zuordnen von Anfragen an eine Prozessinstanz sowie zum Unterscheiden von ankommenden Anfragen nach der Aufteilung in zwei oder mehr Routen erforderlich. Hierbei ist für jeden Pfad ein eigenes CorrelationSet erforderlich. Zum Auffinden der richtigen Prozessinstanz wird die `messageId` herangezogen.

```
<correlationSets>
  <!-- correlationSet messageIdCS zum Auffinden
       der richtigen Prozessinstanz anhand der messageId -->
  <correlationSet name="messageIdCS" properies="ns1:msgId"/>

  <!-- correlationSet Path1ToPath2CS für Pfad mit pathId=2 -->
  <correlationSet name="Path1ToPath2CS" properties="ns1:pathId"/>

  <!-- correlationSet Path1ToPath3CS für Pfad mit pathId=3 -->
  <correlationSet name="Path1ToPath3CS" properties="ns1:pathId2"/>
</correlationSets>
```

Die zugehörigen `property` bzw. `propertyAlias`-Elemente sind wie folgt definiert:

```
<bpws:property name="msgId" type="xsd:string"/>
<bpws:property name="pathId" type="xsd:positiveInteger"/>
<bpws:property name="pathId2" type="xsd:positiveInteger"/>

<bpws:propertyAlias messageType="sbr:RoutingRequest"
                    part="messageId" propertyName="tns:msgId" />
<bpws:propertyAlias messageType="sbr:RoutingResponse"
                    part="messageId" propertyName="tns:msgId" />

<bpws:propertyAlias messageType="sbr:RoutingRequest" part="pathId"
                    propertyName="tns:pathId"/>
<bpws:propertyAlias messageType="sbr:RoutingResponse"
                    part="routeTo" propertyName="tns:pathId"
                    query="/routeTo/node[1]/pathId"/>

<bpws:propertyAlias messageType="sbr:RoutingRequest" part="pathId"
                    propertyName="tns:pathId2"/>
<bpws:propertyAlias messageType="sbr:RoutingResponse"
                    part="routeTo" propertyName="tns:pathId2"
                    query="/routeTo/node[2]/pathId"/>
```

Wesentlich hierbei ist der Unterschied in den Abfragen der `propertyAlias`-Definitionen für die RoutingResponse-Nachrichten. Hierdurch wird es möglich, beide Correlationsets beim Absenden der Antwort an Knoten (1) zu initialisieren und spätere `receive`-Aktivitäten damit zu verknüpfen.

```
<reply operation="getNextHop" partnerLink="RouterToBPEL-PL"
       portType="sbr:RoutingServicePortType" variable="RoutingResponse">
  <correlations>
    <correlation initiate="yes" set="Path1ToPath2CS" />
    <correlation initiate="yes" set="Path1ToPath3CS" />
  </correlations>
</reply>
```

Somit enthält `Path1ToPath2CS` die `pathId` 2 und `Path1ToPath3CS` die `pathId` 3. Dies ermöglicht das parallele Warten zweier `receive` auf denselben Nachrichtentypen in unterschiedlichen Zweigen des Routingprozesses.

```
<flow>
  <sequence>
    <!-- Receive für Anfragen von SBR-Router (2) -->
    <receive operation="getNextHop" partnerLink="RouterToBPEL-PL"
             portType="sbr:RoutingServicePortType"
             variable="RoutingRequest-2">
      <correlations>
        <correlation set="messageIdCS" />
        <correlation set="Path1ToPath2CS" />
      </correlations>
    </receive>
    <assign>
      ...
    </assign>
    <reply>
      ...
    </reply>
  </sequence>

  <sequence>
    <!-- Receive für Anfragen von SBR-Router (3) -->
    <receive operation="getNextHop" partnerLink="RouterToBPEL-PL"
             portType="sbr:RoutingServicePortType"
             variable="RoutingRequest-3">
      <correlations>
        <correlation set="messageIdCS" />
        <correlation set="Path1ToPath3CS" />
      </correlations>
    </receive>
    <assign>
      ...
    </assign>
```

```
    <reply>
      ...
    </reply>
  </sequence>
  ...
</flow>
```

Die beiden parallelen `receive` unterscheiden sich in den verwendeten Variablen und im Correlationset sowie in den darauf folgenden `assign` Operationen. Die einzelnen Aktivitäten bzw. Sequences sind über Links miteinander verknüpft, wie in Abb. 5.8 dargestellt.

Ein Gesamtüberblick des Prozesses sowie der vollständige Routingprozess als BPEL-Dokument ist in Anhang C abgedruckt.

5.6.2. Die Abarbeitung an den SBR-Routern

Original Sender : Hier wird der SBR-Header erzeugt. Er enthält alle Daten, die nötig sind, um zum einen die Nachricht und zum zweiten den nächsten Router eindeutig zu identifizieren. Zusätzlich gibt der Original Sender den Routingprozess vor, nachdem die Nachricht zu bearbeiten ist. Hinzu kommen lediglich noch optionale Angaben von Zusatzdiensten in den `service`-Elementen des `node`-Elementes:

```
<ns1:RoutingInfo soapenv:mustUnderstand="1"
       xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08"
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <replyTo>http://originalsender.example.org:8079/response</replyTo>
  <faultTo>http://originalsender.example.org:8079/response</faultTo>
  <node>
    <pathId>1</pathId>
    <nodeURI>
      http://router1.example.org:8081/SBR-Router/RoutingService
    </nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
    <service>
     <serviceNamespace>
       http://www.example.org/services/Service1
     </serviceNamespace>
     <serviceRootElement>service1</serviceRootElement>
    </service>
  </node>
```

```
</ns1:RoutingInfo>
```

Anschliessend wird der Header an die Nachricht gehängt und diese dann an Router (1) verschickt.

Router (1) : Zunächst wird der Header auf korrekte Struktur geprüft. Nachdem diese Prüfung erfolgt ist, werden die enthaltenen Daten ausgewertet. Als erstes wird geprüft, ob der in `nodeURI` angegebene URI der URI des derzeitigen Routers entspricht. Da dies im Beispiel der Fall ist, werden nun die in der Nachricht enthaltenen `service`-Elemente abgearbeitet und die zugehörigen Zusatzdienste ausgeführt. Erst wenn dies fehlerfrei geschehen ist, wird der unter `processURI` angegebene Routingprozess kontaktiert und eine RoutingRequest-Nachricht abgeschickt. Die Nachricht enthält die `messageId` und die im `node`-Element des Headers enthaltene `pathId`. Aus der Antwort des Prozesses

```
<RoutingResponse>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <routeTo>
   <node>
     <pathId>2</pathId>
     <nodeURI>
       http://router2.example.org:8081/SBR-Service/router2
     </nodeURI>
     <processURI>
       http://proc.example.org:8080/active-bpel/services/route1
     </processURI>
   </node>
   <node>
     <pathId>3</pathId>
     <nodeURI>
       http://router3.example.org:8081/SBR-Service/router3
     </nodeURI>
     <processURI>
       http://proc.example.org:8080/active-bpel/services/route1
     </processURI>
   </node>
  </routeTo>
</RoutingResponse>
```

wird nun für jeden Zielknoten der Antwort ein neuer SBR-Header generiert. Für die Nachricht an Router (2):

```
<ns1:RoutingInfo soapenv:mustUnderstand="1"
       xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08"
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <replyTo>http://originalsender.example.org:8079/response</replyTo>
  <faultTo>http://originalsender.example.org:8079/response</faultTo>
<node>
  <pathId>2</pathId>
    <nodeURI>
      http://router2.example.org:8081/SBR-Service/router2
    </nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
</ns1:RoutingInfo>
```

und für die Nachricht an Router (3):

```
<ns1:RoutingInfo soapenv:mustUnderstand="1"
       xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08"
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <replyTo>http://originalsender.example.org:8079/response</replyTo>
  <faultTo>http://originalsender.example.org:8079/response</faultTo>
<node>
  <pathId>3</pathId>
    <nodeURI>
      http://router3.example.org:8081/SBR-Service/router3
    </nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
</ns1:RoutingInfo>
```

Anschliessend wird die bearbeitete Nachricht dupliziert und mit dem jeweiligen Header versehen an Router (2) und Router (3) gesendet.

Router (2) : An diesem Router werden dieselben grundlegenden Aktionen durchgeführt, wie an Router (1). Der einzige Unterschied liegt in der Antwort des Prozesses, der diesmal nur einen Knoten angibt, zu dem weitergeleitet werden soll. Zudem ändert sich diesmal die `pathId` nicht und bleibt bei *2*. Im Anschluss an die Anfrage an den Prozess wird ein neuer SBR-Header aus den Informationen des Routingprozesses analog zu den von Router (1) erzeugten Headern erstellt. Anschließend wird die Nachricht an Router (4) gesendet.

Router (3) : Die Bearbeitung der Nachricht erfolgt zunächst wie bei Router (2). Bei einer

Anfrage an den Routingprozess liefert dieser jedoch eine abweichende Antwort:

```
<RoutingResponse>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <routeTo>
   <node>
     <pathId>3</pathId>
     <nodeURI>
       http://router5.example.org:8081/SBR-Service/router5
     </nodeURI>
     <processURI>
       http://proc.example.org:8080/active-bpel/services/route1
     </processURI>
     <aggregate
         service="http://www.example.org/services/aggregation/a1">
       <pathId>2</pathId>
       <pathId>3</pathId>
     </aggregate>
   </node>
  </routeTo>
</RoutingResponse>
```

Der Router muss nun in die ausgehende Nachricht zusätzlich das `aggregate`-Element einfügen, bevor die Nachricht an Router (5) gesendet wird.

Router (4) : Auch an Router (4) wird die Nachricht wie an jedem anderen Router behandelt. Auch hier ist in der Antwort des Prozesses – wie bei Router (3) – ein `aggregate`-Element enthalten. Die im `node`-Element angegebene `pathId` ist jedoch unterschiedlich:

```
<RoutingResponse>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <routeTo>
   <node>
     <pathId>2</pathId>
     <nodeURI>
       http://router5.example.org:8081/SBR-Service/router5
     </nodeURI>
     <processURI>
       http://proc.example.org:8080/active-bpel/services/route1
     </processURI>
     <aggregate
         service="http://www.example.org/services/aggregation/a1">
       <pathId>2</pathId>
       <pathId>3</pathId>
```

```
      </aggregate>
    </node>
  </routeTo>
</RoutingResponse>
```

Die Nachrichten, die von den Routern (3) und (4) ausgehen, enthalten also alle für eine erfolgreiche Aggregation an Router (5) notwendigen Informationen.

Router (5) : Im Gegensatz zu den anderen Routern, die in diesem Beispiel vorkommen, ist an Router (5) noch zusätzlich einen Aggregationsdienst installiert. Dieser wird jedoch erst ganz am Ende der Abarbeitung aufgerufen – und auch erst dann, wenn alle zugehörigen Nachrichten am Router eingetroffen sind. D. h. es werden zunächst alle Zusatzdienste je eintreffender Nachricht ausgeführt, bevor der Aggregationsdienst aufgerufen wird. Im Anschluss an die erfolgte Aggregation wird der Routingprozess kontaktiert. Der Wert des `pathId`-Elementes der aggregierten Nachricht und somit der Anfrage an den Prozess ist gleich dem ersten im `aggregate`-Element aufgeführten `pathId` – in diesem Beispiel also *2* Die Antwort des Routingprozesses sieht wie folgt aus:

```
<RoutingResponse>
  <messageId>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</messageId>
  <routeTo>
    <node>
      <pathId>2</pathId>
      <nodeURI>
        http://router5.example.org:8081/SBR-Service/router5
      </nodeURI>
      <processURI>
        http://proc.example.org:8080/active-bpel/services/route1
      </processURI>
    </node>
  </routeTo>
</RoutingResponse>
```

Ultimate Receipient : Hier kommt die Nachricht mit den gesetzten Werten für `replyTo` und `faultTo` an, so dass eine Antwort auf die ankommende Nachricht gegeben werden kann. Da in diesem Beispiel keine weiteren Zusatzdienste am Ultimate Receipient auszuführen sind, wird direkt der Routingprozess kontaktiert. Da dieser ein leeres `routeTo`-Element liefert, wird der durch den Prozess addressierte Web Service aufgerufen und eine Antwort an den in `replyTo` genannten URI mit folgendem Header gesendet:

```
<ns1:RoutingInfo soapenv:mustUnderstand="1"
       xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08"
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <messageId>43ff44-ea4f5d-1276d3-12fea8-7ad3ef</messageId>
  <relatesTo>33ea4f-d5eg41-ab4ca5-5efa3b-7cd901</relatesTo>
</ns1:RoutingInfo>
```

Dies ermöglicht es dem Original Sender, die eintreffende Antwort mit seiner Anfrage zu korrelieren und somit den Webserviceaufruf erfolgreich zu beenden.

KAPITEL 6

Der Prototyp

In diesem Kapitel wird die Architektur des im Rahmen der Arbeit zu entwickelnden Prototypen vorgestellt. Die Implementierung des SBR-Routers erfolgt als HttpServlet, das über die AxisClient-Schnittstelle mit dem Routingprozess kommuniziert. Da am Router selbst lediglich der SBR-Header von Bedeutung ist, konnte auf die Implementierung eines vollständigen SOAP-Stacks verzichtet werden. Lediglich die zur Bearbeitung des SBR-Headers nötige Funktionalität wurde implementiert.

Über eine definierte Schnittstelle lassen sich Zusatzdienste einbinden, die für andere Headerblöcke der SOAP-Nachricht zuständig sind und im Rahmen des Routings durch das Servlet bei Bedarf aufgerufen werden. Auch anwendungsabhängige Aggregationsdienste lassen sich über eine solche Schnittstelle einbinden.

Als Web Service für den Demonstrator wurde ein fest kodierter „Echo“-Service implementiert, der den Inhalt – in diesem Fall eine einfache Zeichenkette – einer Nachricht unverändert an den Absender zurücksendet. Ebenfalls für den Demonstrator wurden mehrere Zusatzdienste implementiert, die den Inhalt der Nachricht auf dem Weg zum Ultimate Receipient verändern und dadurch das Routing nach aussen sichtbar machen. Im Rahmen der Ausarbeitung wird jedoch lediglich die Architektur des SBR-Router-Prototypen gezeigt.

6.1. Überblick der wichtigsten Klassen

In Abbildung 6.1 werden die zentralen Klassen des Prototypen mit den dazwischen bestehenden Abhängigkeiten dargestellt. Im HttpServlet `SbrRouter` ist die Anwendungslo-

gik enthalten. Hier wird die Entscheidung getroffen, was mit ankommenden Nachrichten geschehen soll. Zur Bearbeitung der Nachrichten bedient sich das Servlet den Unterstützungsklassen `ServiceManager` und `AggregationManager`, um Zusatzdienste bzw. Aggregationsdienste an Hand ihres QName auffinden zu können. Im `MessageStorage` werden Nachrichten für eine Aggregation zwischengespeichert.

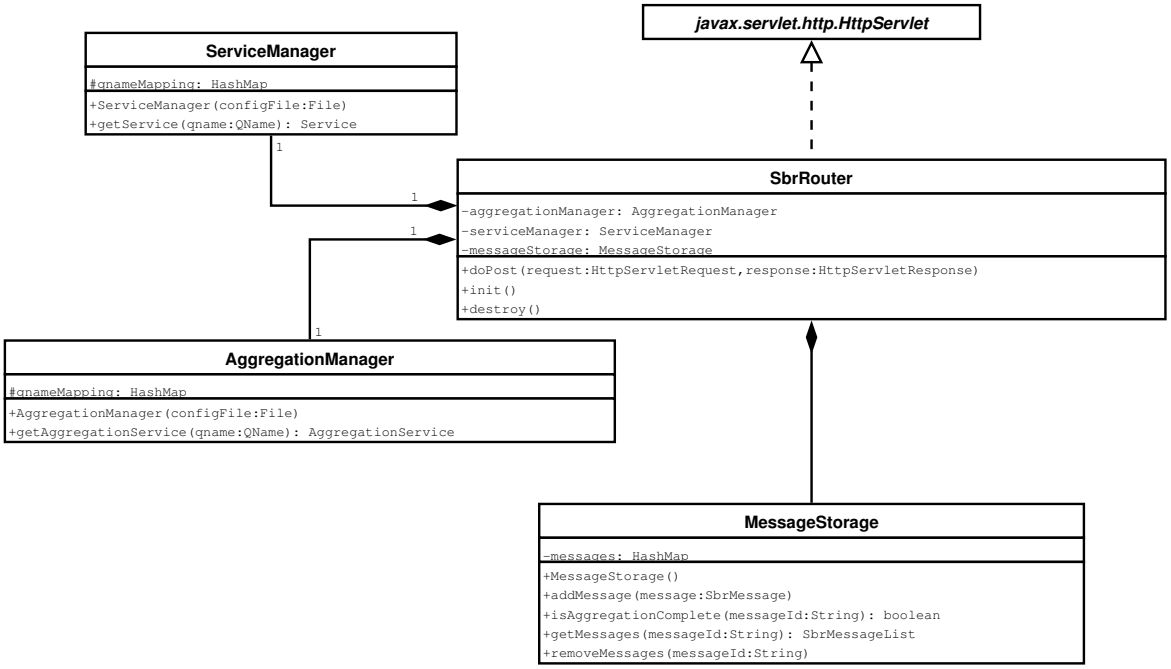

Abbildung 6.1.: Überblick der Hauptklassen

Das HttpServlet `SbrRouter` nimmt eine ankommende SOAP-Nachricht entgegen und extrahiert den SBR-Header und die darin enthaltenen Informationen. Anschliessend werden die im Header angegebenen Zusatzdienste in ihrer definierten Reihenfolge aufgerufen. Zunächst muss zum Auffinden der Zusatzdienste die Methode `getService` des `serviceManager` aufgerufen werden. Anschliessend wird auf dem zurückgelieferten Objekt die Methode `invoke` ausgeführt. Mit dieser Methode wird die SOAP-Nachricht entsprechend der Funktion des Zusatzdienstes bearbeitet und wieder an das aufrufende Servlet zurückgeliefert.

Nach der Abarbeitung der Zusatzdienste wird geprüft, ob mit der erhaltenen Nachricht eine Aggregation nötig ist. Falls ja, so wird die Nachricht zunächst im `MessageStorage` über `addMessage` gespeichert. Anschliessend wird mittels `isAggregationComplete` geprüft, ob bereits alle zu aggregierenden Nachrichten eingetroffen sind. Wenn alle zusammengehörenden Nachrichten eingetroffen sind, so wird der `AggregationService` mit einem Array von Nachrichten aufgerufen. Zum Auffinden des `AggregationService` wird die Methode `getAggregationService` der Unterstützungsklasse `AggregationManager` genutzt. Die aufzurufende Methode des zurückgelieferten `AggregationService`-Objektes heisst `aggregate`. In der dieser Methode zu übergebenden Liste sind alle zu aggregierenden Nachrichten

in der richtigen (wie im `aggregate`-Element der Nachricht angegebenen) Reihenfolge enthalten.

Anschliessend fragt das HttpServlet über einen Webserviceaufruf `getNextHops` den Routingprozess nach dem nächsten Routingschritt. Nach Auswertung der erhaltenen Antwort werden neue SBR-Header für die ausgehenden Nachrichten generiert und die Nachricht wird direkt an die in der Antwort angegebenen Router weitergeleitet. Sollten in der Antwort des Routingprozesses keine weiteren Router enthalten sein, so ist der aktuelle Router der Ultimate Receipient. In diesem Fall wird der Demonstrationswebservice lokal aufgerufen.

6.2. Zusatzdienste

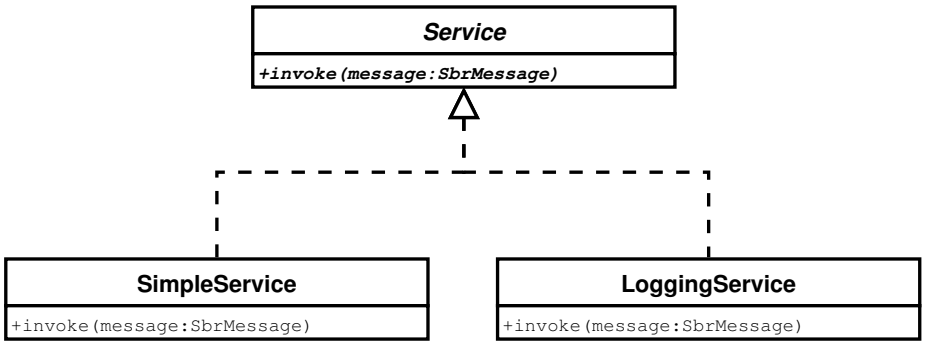

Abbildung 6.2.: Überblick Zusatzdienstschnittstelle

Jeder Zusatzdienst muss die Schnittstelle `Service` implementieren. In der implementierten Methode `invoke` ist die Bearbeitungslogik des Zusatzdienstes enthalten, die über vollen Zugriff auf die gesamte Nachricht verfügt.

6.3. Aggregationsdienste

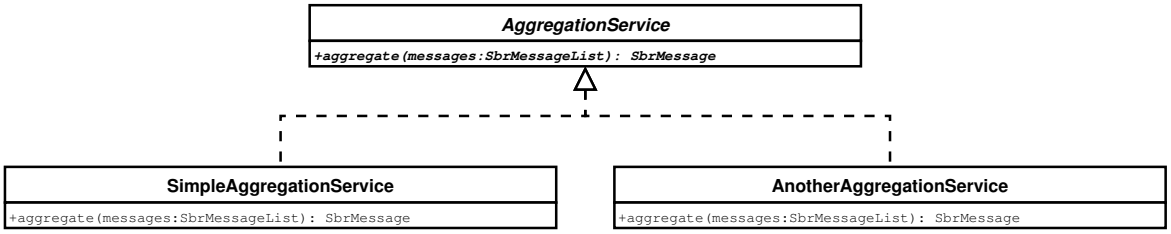

Abbildung 6.3.: Überblick Aggregationsdienstschnittstelle

Je nachdem, welche Nachrichtenpfade und welche Nachrichten zusammengeführt werden sollen, ist ein anderer anwendungsabhängiger Aggregationsdienst nötig. Daher wurde analog zu den Zusatzdiensten eine Schnittstelle für Aggregationsdienste geschaffen. Jeder Aggregationsdienst muss die Schnittstelle `AggregationService` implementieren und die Methode `aggregate` zur Aggregation mehrerer Nachrichten zur Verfügung stellen.

6.4. Datentypen

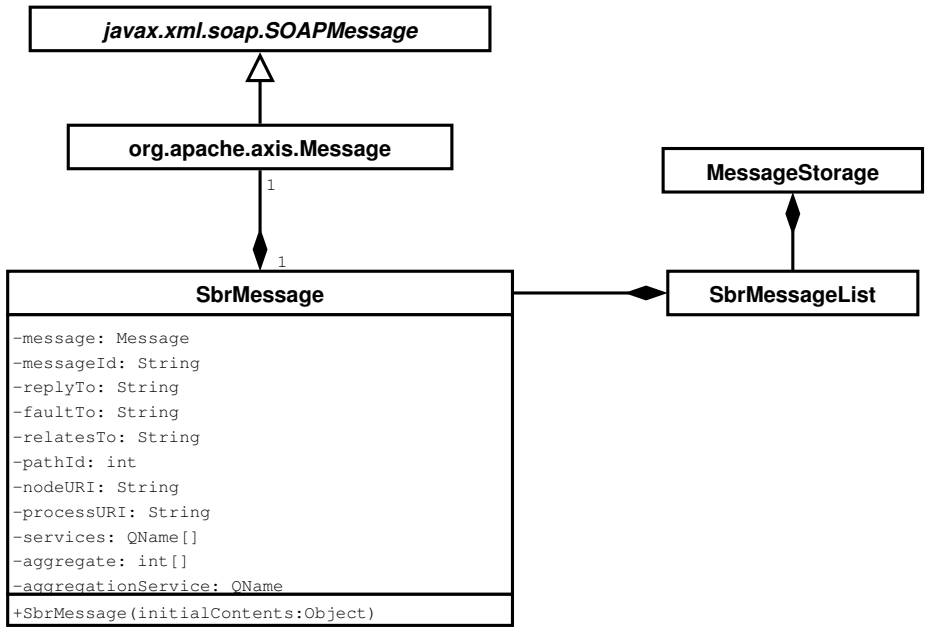

Abbildung 6.4.: Die SbrMessage Klasse

In der Klasse `SbrMessage` sind zusätzlich zur gesamten Nachricht in Form eines `Message`-Objektes die Werte des SBR-Headers zur Vereinfachung des Zugriffes enthalten. Die `SbrMessage` Objekte können im Falle einer Aggregation in einer `SbrMessageList` im `MessageStorage` gespeichert werden. Der Zugriff auf diese Liste erfolgt über die entsprechenden Methoden im `MessageStorage`.

Weitere im Rahmen des Prototypen genutzte Datentypen wurden automatisch durch das Tool `wsdl2java` des Axis-Projektes generiert und dienen dem Aufruf der `getNextHops` Operation des Routingprozesses.

KAPITEL 7

Diskussion

In diesem Kapitel werden die Vor- und Nachteile des in Kapitel 5 vorgestellten Routingverfahrens besprochen.

7.1. Kompatibilität zu anderen WS-Standards

Was die Kompatibilität von SBR zu anderen Web Service Standards wie beispielsweise WS-Security oder WS-Policy angeht, so ist vom Entwurf her mit keiner Einschränkung zu rechnen. Es dürfen allerdings keine Erweiterungen verwendet werden, die ebenfalls dem Routing dienen – oder Erweiterungen, die auf solchen Erweiterungen basieren oder diese verwenden.

Die Anwendung von WS-Addressing und anderen das Ziel einer Nachricht vorgebenden Erweiterungen ist nicht empfohlen, da das Ziel der Nachricht vom Routingprozess vorgegeben wird. Sollte die vom Routingprozess gegebene Adresse des Ultimate Receipients von der im WSA-Header angegebenen Adresse abweichen, so ist nicht definiert, was zu tun ist.

Es muss in jedem Falle jedoch darauf geachtet werden, dass SBR als erste Erweiterung ausgeführt wird und keine andere Erweiterung automatisch aufgerufen wird – andernfalls können Seiteneffekte auftreten, die vom Routingadministrator nicht vorhergesehen wurden und deshalb im Routingprozess keine Berücksichtigung finden. Da es sich bei SBR um eine Routingerweiterung handelt, die zusätzlich noch die Reihenfolge der Headerabarbeitung bestimmt, werden andere Erweiterungen von SBR aus aufgerufen. Bei einer Implementierung ist also sicher zu stellen, dass eine solche Aufrufmöglichkeit existiert.

7.2. Vor- und Nachteile

Das in diesem Kapitel vorgestellte Routingverfahren mit Hilfe von BPEL als Konfigurationssprache für ein Routing ist auf den ersten Blick im Vergleich mit WS-Routing unnötig kompliziert: jeder SBR-Router muss mit einem Prozess Kontakt aufnehmen, um zu erfahren, wie mit der gerade angekommenen Nachricht weiter verfahren werden soll.

Der Grund hierfür ist einfach: man erhält eine sehr große Flexibilität des Routings. Auf einfache Art und Weise kann ein Administrator ein vorhandenes Routing abändern, ohne an mehreren Stellen eingreifen zu müssen. Auch ein die Unternehmensgrenzen überschreitendes Routing ist hiermit möglich, ohne dass der Administrator des einen Unternehmens über Detailwissen des Netzwerkes des zweiten Unternehmens verfügen muss. Er muss lediglich die Adresse eines Routingprozesses im anderen Unternehmen kennen und diese auf dem letzten Schritt im eigenen Unternehmen dem nächsten SBR-Router als `processURI` mitgeben. Dabei muss der Routingprozess nicht von ausserhalb erreichbar sein, sondern kann hinter der Firewall im geschützten Bereich des Firmennetzes liegen – sofern alle SBR-Router, die Informationen von diesem Routingprozess benötigen, den Prozess erreichen können.

Ein weiterer wesentlicher Punkt ist die Fähigkeit von SBR im Gegensatz zu anderen Routingverfahren, die Abarbeitungsreihenfolge der Headerblöcke festzulegen. Hierdurch ist es nicht mehr von der jeweiligen Implementierung oder der individuellen Konfiguration eines SOAP-Intermediaries abhängig, in welcher Reihenfolge die installierten Zusatzdienste abgearbeitet werden, sondern es wurde eine zentrale Konfigurationsmöglichkeit geschaffen, die eine bereits standardisierte Sprache einsetzt für die zahlreiche Tools zur Bearbeitung existieren.

Der größte Vorteil besteht darin, dass es mit dem hier vorgestellten Routingverfahren möglich ist, auch auf Zusatzdienstebene eine parallele Abarbeitung zu definieren. Beispielsweise kann ein Logging parallel zur übrigen Verarbeitung stattfinden.

Nachteilig im Vergleich mit anderen Routingverfahren wirkt sich die Kommunikation mit dem Routingprozess aus. Hierdurch ist es nötig, zusätzlich zu den eigentlichen Nachrichten weitere Routinginformationsnachrichten zwischen den einzelnen Routern und dem Routingprozess auszutauschen. Dies führt zu einer zusätzlichen Verzögerung bei der Verarbeitung von Nachrichten an den einzelnen SBR-Routern. Hierdurch wird die Implementierung von zeitkritischer Kommunikation sehr stark eingeschränkt.

7.3. Probleme und mögliche Lösungen

Wie bereits in Abschnitt 5.5.7 „Fehlerbehandlung“ vorgestellt, können bei Verwendung eines verteilten Systems immer Fehler auftreten, die teilweise nur schwer oder gar nicht zu erkennen sind. An jeder Stelle des Routings kann es zum Verlust von Nachrichten auf

Grund von z. B. Netzwerkfehlern kommen. In den meisten Fällen treten hierdurch kaum oder keine Probleme auf, da solche Fehler an den Original Sender in Form von SOAPFaults gemeldet werden.

Was passiert aber, wenn im Routing ein Pfad definiert wurde, der nicht wieder aggregiert wird, sondern ohne Weiterleitung der Nachricht endet und wenn in einer solchen Sackgasse eine Nachricht verloren geht? Auch dieser Fehler wird an den Absender der Nachricht gemeldet, bleibt aber von den übrigen Pfaden des Routings unbemerkt, so dass die Nachricht dennoch am Ultimate Receipient ausgeliefert wird. Zur Verhinderung dieses Fehlers gibt es zwei Möglichkeiten:

Zum einen das Verbieten von Sackgassen: Hierdurch wird der Fehler bereits im Vorfeld verhindert und kann überhaupt nicht mehr auftreten. Allerdings muss nun jeder getrennte Pfad eines Routings immer zu einem Aggregationsknoten geleitet werden, bevor die Nachricht an den Ultimate Receipient ausgeliefert werden darf. D. h. es wird erzwungen, dass auf Nachrichten gewartet wird, die keine neuen Informationen bringen.

Die andere Möglichkeit besteht darin, im Prozess eine weitere Operation zu definieren, über die einzelne SBR-Router Fehlerfälle melden können. Hierdurch wird der Prozess in die Lage versetzt, auf Fehler im Routing angemessen reagieren zu können. Hier taucht nun aber ein weiteres Problem auf: sollte die Abarbeitung in einer Sackgasse länger dauern als die Abarbeitung auf dem Pfad zum Ultimate Receipient, so kann es immer noch passieren, dass in der Sackgasse ein Fehler auftritt, nachdem die Nachricht bereits zum Ultimate Receipient ausgeliefert wurde.

Der Vorteil den Sackgassen vermeintlich bieten – parallele Abarbeitung von Headern, die den Inhalt der Nachricht nicht ändern – ist im Vergleich zu den Nachteilen kaum spürbar. Aus diesem Grund wurde die ursprünglich angedachte Möglichkeit von Sackgassen im Routing schnell wieder verworfen und die Definition von Sackgassen im Routingprozess verboten.

Was passiert aber, wenn eine Nachricht auf dem Pfad zum Ultimate Receipient verloren geht? Dann wird ein SOAPFault an den Original Sender geschickt und dieser ist somit über das Problem informiert. Was ist mit den auf dem Weg bisher ausgeführten Zusatzdiensten an den übrigen SBR-Routern? Beipielsweise könnte bereits in einem Loggingdienst die Bestellung eines Artikels vermerkt worden sein. Im Falle eines Fehlers im Routing müsste diese Operation rückgängig gemacht werden, da die Bestellung ja nie erfolgt ist. Hierzu ist im vorliegenden Verfahren keine Möglichkeit definiert. Eine mögliche Lösung wird jedoch im Ausblick im folgenden Kapitel kurz beschrieben.

KAPITEL 8

Zusammenfassung und Ausblick

In diesem Kapitel wird die vorliegende Arbeit zunächst kurz zusammengefasst, bevor im zweiten Abschnitt „Ausblick“ Erweiterungsmöglichkeiten beschrieben werden.

8.1. Zusammenfassung

Das in dieser Arbeit vorgestellte Routingverfahren ermöglicht es auf standardisierte Art und Weise eine Route für SOAP-Nachrichten vorzugeben. Zusätzlich zur einfachen Angabe von Intermediaries über die eine Nachricht geroutet werden soll, bietet das vorgestellte SBR-Verfahren die Möglichkeit, zu jedem Intermediary die dort abzuarbeitenden Headerblöcke der Nachricht inklusive deren Reihenfolge festzulegen.

Möglich wird das Routing erst durch einen Mechanismus auf jedem SBR-Intermediary, welcher die vollständige Kontrolle der SBR-Erweiterung über etwaige zusätzlich installierte Erweiterungen gewährleistet:

1. muss die SBR-Erweiterung immer als einzige Erweiterung automatisch beim Eintreffen einer SOAP-Nachricht aufgerufen werden.
2. muss sich jede zusätzliche Erweiterung über eine implementierungsabhängige aber dort festgelegte Schnittstelle bzw. einen standardisierten Mechanismus aufrufen lassen und die bearbeitete Nachricht wieder an den Aufrufer zurückgeben.
3. muss jede Erweiterung eine Möglichkeit bieten, den Namespace und das Wurzelelement des zugehörigen Headerblockes an die Routingerweiterung mitzuteilen. Nur

so kann die zu einem Header gehörige Erweiterung automatisch durch die SBR-Erweiterung aufgerufen werden.

Im Gegensatz zu bereits existierenden Routingverfahren für SOAP-Nachrichten wie z. B. WS-Routing sind in einer Nachricht mit SBR-Header nicht die weiteren Intermediaries aufgelistet, sondern die Zusatzdienste, die am bearbeitenden Intermediary abzuarbeiten sind. Jeder Zusatzdienst ist hierbei durch den QName des zugehörigen Headerblockes eindeutig zu identifizieren.

Zusätzlich ist die Adresse eines Routingprozesses (BPEL) angegeben, über die sich der bearbeitende SOAP-Knoten eine Liste der im nächsten Schritt anzusprechenden Intermediaries mitsamt der Liste der dort zu bearbeitenden Headerblöcke besorgen kann.

SBR ermöglicht es also, eine sehr feingranulare Routingstruktur unter Berücksichtigung der Headerabarbeitungsreihenfolge in einem BPEL-Prozess zu definieren. Durch die Definition des Routings als BPEL-Prozess wird zum einen eine weitere Anwendungsmöglichkeit von BPEL aufgezeigt und zum anderen eine wohldefinierte Schnittstelle zur Abfrage der Routinginformationen durch die SBR-Router genutzt.

Zudem ermöglichen es die in BPEL vorhandenen Aktivitäten, ein Routing mit parallel ablaufenden Zweigen zu definieren. Die parallele Abarbeitung von Kopien einer Nachricht auf unterschiedlichen Intermediaries mit anschliessender Aggregation ist mit dem in dieser Arbeit beschriebenen Verfahren möglich und im Bereich des Routings von SOAP-Nachrichten bislang einzigartig.

8.2. Ausblick

In jedem neu entwickelten Verfahren gibt es Verbesserungsmöglichkeiten und denkbare Erweiterungen. Ein paar Vorschläge hierzu werden in diesem Abschnitt vorgestellt.

8.2.1. Routingprozess

Im Rahmen dieser Arbeit wurden lediglich die grundlegenden Prinzipien von SBR erläutert und implementiert. So holt sich der enthaltene Beispielprozess die Routinginformationen aus fest in den Prozess eingebundenen Werten. Man könnte sich jedoch auch einen Prozess vorstellen, der z. B. um die Zusatzdienste zu bestimmen einen weiteren Web Service kontaktiert, um dort Informationen über die möglicherweise vom Ziel der Nachricht geforderten oder angebotenen Dienstmerkmale wie Sicherheit oder Transaktionen zu erhalten. Basierend auf diesen Informationen könnte der Prozess dann über das weitere Routing sowie die zu verwendenden Zusatzdienste dynamisch entscheiden.

Der in dieser Arbeit enthaltene Beispielprozess macht allerdings auch deutlich, dass die Definition eines Routings mittels BPEL nicht ganz trivial ist, sondern hierbei auf eine

Vielzahl von Schwierigkeiten eingegangen werden muss. Zum einen müssen immer die korrekten `pathId`-Werte übermittelt werden und zum anderen müssen die zugehörigen `correlationSets` innerhalb des Prozesses für jeden Zweig manuell erstellt werden. Auch ist bei der Zusammenfassung (Aggregation) zweier oder gar mehrerer Pfade die Liste der `pathId`s manuell zu erstellen und für jeden Pfad konsistent zu halten. Hierbei können sich sehr leicht Fehler einschleichen, so dass eine Unterstützungssoftware speziell zur Erstellung von Routingprozessen wünschenswert wäre. Dies gilt umso mehr, wenn der Routingprozess Entscheidungen dynamisch unter Einbeziehung von weiteren Prozessen oder Web Services treffen muss.

8.2.2. Splitting von Nachrichten statt Kopieren in mehrere Pfade

An den Abläufen beim Aufteilen eines Nachrichtenpfades in mehrere parallele Pfade gibt es Optimierungsmöglichkeiten. Im vorgestellten Verfahren werden die Nachrichten lediglich auf die ausgehenden Pfade kopiert. Es ist jedoch denkbar, dass auf einem Pfad nicht alle Informationen der Ausgangsnachricht zur Bearbeitung benötigt werden. SOAP-Nachrichten sind mitunter sehr groß, insbesondere wenn SOAP-Attachments genutzt werden. Daher kann es sinnvoll sein, auf die unterschiedlichen Pfade nur den jeweils benötigten Teil einer Nachricht zu schicken. Hierzu wäre ein Verfahren zur anwendungsabhängigen Aufteilung der Nachrichten ähnlich dem Verfahren bei der Aggregation von Nachrichten nötig, so dass in der Antwort des Routingprozesses zusätzliche Informationen zum „Splittingdienst“ enthalten sein müssten.

Hiermit könnten die zu übertragenden Datenmengen der einzelnen Pfade unter Umständen deutlich gesenkt werden und somit ein schlankeres und ressourcenschonenderes Routing erzielt werden.

8.2.3. Transaktionelles Verhalten des Routings

Der Verlust von Nachrichten im Routing nachdem bereits Zusatzdienste auf der Nachricht ausgeführt wurden ist nicht unproblematisch, da hierdurch beispielsweise in einem Logging fälschlicherweise die Auslieferung der Nachricht vermerkt worden sein könnte, ohne dass dies tatsächlich geschehen ist.

Daher wird im Folgenden eine mögliche Erweiterung zum in Kapitel 5 vorgestellten Routingverfahren kurz umrissen, die derartige Fehler vermeiden kann. Um die vorangegangenen Bearbeitungsschritte im Fall eines Fehlers wieder rückgängig zu machen, muss eine solche Erweiterung in der Lage sein, die einzelnen am Routing bisher beteiligten SBR-Router über Erfolg oder Mißerfolg des Routingprozesses zu benachrichtigen.

Hierzu ist es denkbar, dass die SBR-Router über eine zusätzliche Schnittstelle mit dem Routingprozess Verbindung aufnehmen können und den aktuellen Status des Routings

erfahren können. Im Falle eines Fehlers wird dies dem Routingprozess über eine weitere Schnittstelle mitgeteilt und bei Anfragen der SBR-Router kann daraufhin gemeldet werden, ob der Routingprozess fehlerfrei terminiert wurde, dieser sich noch in der Ablaufphase befindet oder ob er mit einem Fehler abgebrochen wurde. Dies erfordert allerdings ein periodisches Abfragen durch die SBR-Router (Polling).

Eine verbesserte Variante hiervon ist, dass die SBR-Router, die eine Nachricht erfolgreich weitergeleitet haben, eine asynchrone Anfrage an den Prozess stellen, die dieser dann erst im Fehlerfall oder bei ordnungsgemäßer Beendigung des Prozesses beantwortet. Auf diese Art wäre sichergestellt, dass jeder zum Routing gehörende SBR-Router über den Erfolg oder Mißerfolg des Gesamtroutings informiert wird. Somit kann beispielsweise ein einmal getätigtes Logging einer Nachricht mit dem Zusatz „failed“ oder „success“ erweitert werden.

Eine solche Erweiterung geht weit über die Anforderungen an ein normales Routing hinaus. Durch diese Erweiterung würde ein zuverlässiges und (für die Zusatzdienste) transaktionelles Routing entstehen, das verlorengegangene Nachrichten erkennen und eventuell bislang getätigte Änderungen rückgängig machen kann.

ANHANG A

WSDL-Schnittstellendefinition

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:sbr="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/
    routingservice"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:sbrt="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
                  targetNamespace="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/
                      routingservice" name="routingservice">
        <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:sbrt="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
                targetNamespace="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
                    elementFormDefault="qualified">
        <xsd:complexType name="serviceType">
                <xsd:sequence>
                        <xsd:element name="serviceNamespace" type="xsd:anyURI"/>
                        <xsd:element name="serviceRootElement" type="xsd:NCName"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="aggregationType">
                <xsd:sequence>
                        <xsd:element name="pathId" type="xsd:positiveInteger" maxOccurs="
                            unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="service" type="xsd:QName" use="required"/>
        </xsd:complexType>
        <xsd:complexType name="nodeType">
                <xsd:sequence>
                        <xsd:element name="pathId" type="xsd:positiveInteger"/>
                        <xsd:element name="nodeURI" type="xsd:anyURI"/>
                        <xsd:element name="processURI" type="xsd:anyURI"/>
                        <xsd:element name="service" type="sbrt:serviceType" minOccurs="0"
                            maxOccurs="unbounded"/>
                        <xsd:element name="aggregate" type="sbrt:aggregationType"
                            minOccurs="0"/>
```

```
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="routeToType">
            <xsd:sequence>
                <xsd:element name="node" minOccurs="0" maxOccurs="unbounded" type
                    ="sbrt:nodeType">
                        </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:schema>
        </wsdl:types>
        <wsdl:message name="getNextHopsResponse">
            <wsdl:part name="messageId" type="xsd:string"/>
            <wsdl:part name="routeTo" type="sbrt:routeToType"/>
        </wsdl:message>
        <wsdl:message name="getNextHopsRequest">
            <wsdl:part name="messageId" type="xsd:string" />
            <wsdl:part name="pathId" type="xsd:positiveInteger" />
        </wsdl:message>
        <wsdl:portType name="routingservicePortType">
            <wsdl:operation name="getNextHops">
                <wsdl:input message="sbr:getNextHopsRequest"/>
                <wsdl:output message="sbr:getNextHopsResponse"/>
            </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="routingserviceSOAP" type="sbr:routingservicePortType">
            <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http
                "/>
            <wsdl:operation name="getNextHops">
                <soap:operation soapAction="urn:iaas.uni-stuttgart.de/proposals/
                    sbr/2006/08/routingservice/getNextHops"/>
                <wsdl:input>
                    <soap:body use="literal"
                        namespace="urn:iaas.uni-stuttgart.de/proposals/
                            sbr/2006/08/routingservice" />
                </wsdl:input>
                <wsdl:output>
                    <soap:body use="literal"
                        namespace="urn:iaas.uni-stuttgart.de/proposals/
                            sbr/2006/08/routingservice" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="routingservice">
            <wsdl:port name="routingserviceSOAP" binding="sbr:routingserviceSOAP">
                <soap:address location="http://localhost:8080/active-bpel/services
                    /RoutingService"/>
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Listing A.1: WSDL-Schnittstellendefinition

ANHANG B

XML-Schema des Routing-Headers

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sbrt="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
            targetNamespace="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
            elementFormDefault="qualified">
        <xsd:complexType name="serviceType">
                <xsd:sequence>
                        <xsd:element name="serviceNamespace" type="xsd:anyURI"/>
                        <xsd:element name="serviceRootElement" type="xsd:NCName"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="aggregationType">
                <xsd:sequence>
                        <xsd:element name="pathId" type="xsd:positiveInteger" maxOccurs="
                            unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="service" type="xsd:QName" use="required"/>
        </xsd:complexType>
        <xsd:complexType name="nodeType">
                <xsd:sequence>
                        <xsd:element name="pathId" type="xsd:positiveInteger"/>
                        <xsd:element name="nodeURI" type="xsd:anyURI"/>
                        <xsd:element name="processURI" type="xsd:anyURI"/>
                        <xsd:element name="service" type="sbrt:serviceType" minOccurs="0"
                            maxOccurs="unbounded"/>
                        <xsd:element name="aggregate" type="sbrt:aggregationType"
                            minOccurs="0"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="RoutingInfo">
                <xsd:sequence>
                        <xsd:element name="messageId" type="xsd:string"/>
                        <xsd:element name="replyTo" type="xsd:anyURI" minOccurs="0"/>
                        <xsd:element name="faultTo" type="xsd:anyURI" minOccurs="0"/>
                        <xsd:element name="relatesTo" type="xsd:string" minOccurs="0"/>
                        <xsd:element name="node" type="sbrt:nodeType"/>
```

```
                </xsd:sequence>
            </xsd:complexType>
</xsd:schema>
```

Listing B.1: XML-Schema für den Routingheader

ANHANG C

Der Beispielprozess

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="route1" targetNamespace="urn:iaas.uni-stuttgart.de/proposals/sbr
    /2006/08/route1"
                  xmlns:tns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/route1"
                  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
                  xmlns:ns3="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
                  xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/
                      routingservice"
                  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:import namespace="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/routingservice"
             location="project:/route1/WSDL/routingservice.wsdl"/>
<plnk:partnerLinkType name="SOAPtoBPEL_PLT"
                      xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  <plnk:role name="routingservice">
    <plnk:portType name="ns1:routingservicePortType"/>
  </plnk:role>
</plnk:partnerLinkType>
<bpws:property name="msgId" type="xsd:string"
               xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsRequest"
                    part="messageId" propertyName="tns:msgId"
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsResponse"
                    part="messageId" propertyName="tns:msgId"
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>

<bpws:property name="pthId" type="xsd:positiveInteger"
               xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsRequest"
                    part="pathId" propertyName="tns:pthId"
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsResponse"
                    part="routeTo" propertyName="tns:pthId"
                    query="/ns3:routeTo/ns3:node[1]/ns3:pathId"
```

```
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>

<bpws:property name="pthId3" type="xsd:positiveInteger"
               xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsRequest"
                    part="pathId" propertyName="tns:pthId3"
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
<bpws:propertyAlias messageType="ns1:getNextHopsResponse"
                    part="routeTo" propertyName="tns:pthId3"
                    query="/ns3:routeTo/ns3:node[2]/ns3:pathId"
                    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
</wsdl:definitions>
```

Listing C.1: Zusätze zur WSDL-Schnittstelle für den Prozess

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:ns1="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/routingservice"
         xmlns:ns2="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/route1"
         xmlns:ns3="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         name="route1" suppressJoinFailure="yes" targetNamespace="http://route1">
   <partnerLinks>
      <partnerLink myRole="routingservice" name="RouterToBPEL-PL" partnerLinkType="ns2:
          SOAPtoBPEL_PLT"/>
   </partnerLinks>
   <variables>
      <variable messageType="ns1:getNextHopsRequest" name="getNextHopsRequest"/>
      <variable messageType="ns1:getNextHopsResponse" name="getNextHopsResponse"/>
      <variable messageType="ns1:getNextHopsRequest" name="RequestP2"/>
      <variable messageType="ns1:getNextHopsRequest" name="RequestP3"/>
      <variable messageType="ns1:getNextHopsResponse" name="ResponseP2"/>
      <variable messageType="ns1:getNextHopsResponse" name="ResponseP3"/>
   </variables>
   <correlationSets>
      <correlationSet name="messageIdCS" properties="ns2:msgId"/>
      <correlationSet name="Path1ToPath2CS" properties="ns2:pthId"/>
      <correlationSet name="Path1ToPath3CS" properties="ns2:pthId3"/>
   </correlationSets>
   <flow>
      <links>
         <link name="L1"/>
         <link name="L2"/>
         <link name="L3"/>
         <link name="L4"/>
         <link name="L7"/>
      </links>
      <sequence name="Router1">
         <source linkName="L1"/>
         <source linkName="L2"/>
         <receive createInstance="yes" operation="getNextHops" partnerLink="RouterToBPEL-
             PL" portType="ns1:routingservicePortType" variable="getNextHopsRequest">
            <correlations>
               <correlation initiate="yes" set="messageIdCS"/>
            </correlations>
         </receive>
         <assign>
            <copy>
               <from part="messageId" variable="getNextHopsRequest"/>
               <to part="messageId" variable="getNextHopsResponse"/>
            </copy>
            <copy>
               <from>
<routeTo xmlns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types">
```

```
  <node>
    <pathId>2</pathId>
    <nodeURI>http://router2.example.org:8081/SBR-Service/router2</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
  <node>
    <pathId>3</pathId>
    <nodeURI>http://router3.example.org:8081/SBR-Service/router3</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
</routeTo>
               </from>
               <to part="routeTo" variable="getNextHopsResponse"/>
            </copy>
         </assign>
         <reply operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="getNextHopsResponse">
            <correlations>
               <correlation set="messageIdCS"/>
               <correlation initiate="yes" set="Path1ToPath2CS"/>
               <correlation initiate="yes" set="Path1ToPath3CS"/>
            </correlations>
         </reply>
      </sequence>
      <sequence name="Router3">
         <target linkName="L1"/>
         <source linkName="L3"/>
         <receive operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="RequestP3">
            <correlations>
               <correlation set="messageIdCS"/>
               <correlation set="Path1ToPath3CS"/>
            </correlations>
         </receive>
         <assign>
            <copy>
               <from part="messageId" variable="RequestP3"/>
               <to part="messageId" variable="ResponseP3"/>
            </copy>
            <copy>
               <from>
<routeTo xmlns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types">
  <node>
    <pathId>3</pathId>
    <nodeURI>http://router4.example.org:8081/SBR-Service/router4</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
</routeTo>
               </from>
               <to part="routeTo" variable="ResponseP3"/>
            </copy>
         </assign>
         <reply operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="ResponseP3">
            <correlations>
               <correlation set="messageIdCS"/>
            </correlations>
         </reply>
      </sequence>
```

```
      <sequence name="Router2">
         <target linkName="L2"/>
         <source linkName="L4"/>
         <receive operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="RequestP2">
            <correlations>
               <correlation set="messageIdCS"/>
               <correlation set="Path1ToPath2CS"/>
            </correlations>
         </receive>
         <assign>
            <copy>
               <from part="messageId" variable="RequestP2"/>
               <to part="messageId" variable="ResponseP2"/>
            </copy>
            <copy>
               <from>
<routeTo xmlns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types">
  <node>
    <pathId>2</pathId>
    <nodeURI>http://router5.example.org:8081/SBR-Service/router5</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
    <aggregate service="http://www.example.org/services/aggregation/a1">
      <pathId>2</pathId>
      <pathId>3</pathId>
    </aggregate>
  </node>
</routeTo>
               </from>
               <to part="routeTo" variable="ResponseP2"/>
            </copy>
         </assign>
         <reply operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="ResponseP2">
            <correlations>
               <correlation set="messageIdCS"/>
            </correlations>
         </reply>
      </sequence>
      <sequence name="Router4">
         <target linkName="L3"/>
         <source linkName="L7"/>
         <receive operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
             routingservicePortType" variable="RequestP3">
            <correlations>
               <correlation set="messageIdCS"/>
               <correlation set="Path1ToPath3CS"/>
            </correlations>
         </receive>
         <assign>
            <copy>
               <from part="messageId" variable="RequestP3"/>
               <to part="messageId" variable="ResponseP3"/>
            </copy>
            <copy>
               <from>
<routeTo xmlns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types">
  <node>
    <pathId>3</pathId>
    <nodeURI>http://router5.example.org:8081/SBR-Service/router5</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
```

```
    <aggregate service="http://www.example.org/services/aggregation/a1">
      <pathId>2</pathId>
      <pathId>3</pathId>
    </aggregate>
  </node>
</routeTo>
                </from>
                <to part="routeTo" variable="ResponseP2"/>
             </copy>
          </assign>
          <reply operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
              routingservicePortType" variable="ResponseP3">
             <correlations>
                <correlation set="messageIdCS"/>
             </correlations>
          </reply>
       </sequence>
       <sequence name="Router5">
          <target linkName="L4"/>
          <target linkName="L7"/>
          <receive operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
              routingservicePortType" variable="RequestP2">
             <correlations>
                <correlation set="messageIdCS"/>
                <correlation set="Path1ToPath2CS"/>
             </correlations>
          </receive>
          <assign>
             <copy>
                <from part="messageId" variable="RequestP2"/>
                <to part="messageId" variable="ResponseP2"/>
             </copy>
             <copy>
                <from>
<routeTo xmlns="urn:iaas.uni-stuttgart.de/proposals/sbr/2006/08/types">
  <node>
    <pathId>2</pathId>
    <nodeURI>http://receipient.example.org:8081/SBR-Service/ultimate_receipient</nodeURI>
    <processURI>
      http://proc.example.org:8080/active-bpel/services/route1
    </processURI>
  </node>
</routeTo>
                </from>
                <to part="routeTo" variable="ResponseP2"/>
             </copy>
          </assign>
          <reply operation="getNextHops" partnerLink="RouterToBPEL-PL" portType="ns1:
              routingservicePortType" variable="ResponseP2">
             <correlations>
                <correlation set="messageIdCS"/>
             </correlations>
          </reply>
       </sequence>
    </flow>
</process>
```

Listing C.2: Ein Beispielroutingprozess

Literaturverzeichnis

[ACD+03] Andrews, Tony ; Curbera, Francisco ; Dholakia, Hitesh ; Goland, Yaron ; Klein, Johannes ; Leymann, Frank ; Liu, Kevin ; Roller, Dieter ; Smith, Doug ; Thatte, Satish ; Trickovic, Ivana ; Weerawarana, Sanjiva: *Business Process Execution Language for Web Services 1.1.* Version: May 2003. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`, Abruf: 13.08.2006

[BM04] Biron, Paul V. (Hrsg.) ; Malhotra, Ashok (Hrsg.): *XML Schema Part 2: Datatypes Second Edition.* Version: Oct 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`, Abruf: 13.08.2006

[CCMW01] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: *Web Service Description Language (WSDL).* Version: Mar 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, Abruf: 13.08.2006

[GDS+04] Graham, Steve ; Davis, Doug ; Simeonov, Simeon ; Daniels, Glen ; Brittenham, Peter ; Nakamura, Yuichi ; Fremantle, Paul ; Koenig, Dieter ; Zentner, Claudia: *Building Web Services with Java : Making Sense of XML, SOAP, WSDL, and UDDI (2nd Edition).* Sams, 2004. – ISBN 0672326418

[GHM+03] Gudgin, Martin (Hrsg.) ; Hadley, Marc (Hrsg.) ; Mendelsonhn, Noah (Hrsg.) ; Moreau, Jean-Jacques (Hrsg.) ; Nielsen, Hernrik F. (Hrsg.): *SOAP Version 1.2.* Version: 24.06.2003. `http://www.w3.org/TR/2003/REC-soap12-part1-20030624/`, Abruf: 13.08.2006

[GHR06a] Gudgin, Martin (Hrsg.) ; Hadley, Marc (Hrsg.) ; Rogers, Tony (Hrsg.):

Web Services Addressing 1.0 - Core. Version: May 2006. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/, Abruf: 13.08.2006

[GHR06b] Gudgin, Martin (Hrsg.) ; Hadley, Marc (Hrsg.) ; Rogers, Tony (Hrsg.): *Web Services Addressing 1.0 - SOAP Binding*. Version: May 2006. http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/, Abruf: 13.08.2006

[HW03] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003 http://www.eaipatterns.com. – ISBN 0321200683

[ibm06] *SOA and Web services: New to SOA and Web Services*. Version: Jun 2006. http://www.ibm.com/developerworks/webservices/newto/#1, Abruf: 13.08.2006

[LMS05] Leach, P. ; Mealling, M. ; Salz, R.: *A Universally Unique IDentifier (UUID) URN Namespace (RFC4122)*. Version: July 2005. http://www.ietf.org/rfc/rfc4122.txt, Abruf: 13.08.2006

[NCLL01] Nielsen, Hendrik F. ; Christensen, Erik ; Lucco, Steve ; Levin, David: *Web Services Referral Protocol (WS-Referral)*. Version: Oct 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp, Abruf: 13.08.2006

[NT01] Nielsen, Henrik F. ; Thatte, Satish: *Web Services Routing Protocol (WS-Routing)*. Version: Oct 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-routing.asp, Abruf: 13.08.2006

[w3c03] *SOAP Envelope Schema*. Version: May 2003. http://www.w3.org/2003/05/soap-envelope/, Abruf: 13.08.2006

[WCL+05] Weerawarana, Sanjiva ; Curbera, Francisco ; Leymann, Frank ; Storey, Tony ; Ferguson, Donald F.: *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. – ISBN 0131488740

[wik06] *Universally Unique Identifier - Wikipedia*. Version: 13.08.2006. http://de.wikipedia.org/wiki/UUID, Abruf: 13.08.2006

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

(Frederik Juchart)