Migration einer Middlewaretechnologie
Ein Vergleich von Corba und der Web Service-Technologie am Beispiel eines verteilten Service Schedulers

Studienarbeit
Carsten Hastedt
Februar 2004

bei Prof. Dr. Christiane Floyd
weitere Betreuung: Dr. Wolf-Gideon Bleek

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Inhaltsverzeichnis

1 Einleitung

Diese Studienarbeit ist im Rahmen einer Kooperation des Fachbereichs Informatik der Universität Hamburg (Softwaretechnik Center des Arbeitsbereichs Softwaretechnik) mit der Firma Tenovis entstanden. Am Standort Bargteheide entwickelt die Firma Tenovis Software im Telefoniebereich zur Unterstützung von Callcentern sowie Unified Messaging Lösungen. Im Folgenden wird der Themenbereich und die Zielsetzung dieser Arbeit motiviert, sowie die als Grundlage dieser Arbeit dienende Software-Komponente und dessen Anwendungsbereich vorgestellt. Abschließend folgt eine Aufstellung über den Aufbau der Arbeit bzw. der einzelnen Kapitel.

1.1 Motivation und Zielsetzung

Bei der Entwicklung von verteilten Anwendungen treten typische Anforderungen bzw. Probleme auf. Im Rahmen dieser Studienarbeit soll untersucht werden, wie diese Anforderungen mittels einer bestimmten Middleware umgesetzt werden können. Dazu werden zum Vergleich die beiden Middleware Technologien Corba und Web Services herangezogen. Insbesonders soll untersucht werden, inwieweit sich Web Services als Alternative bzw. Ergänzung zu etablierten Technologien wie Corba eignen. Im Zuge dessen soll zusätzlich die flexible Verwendbarkeit von XML untersucht werden.

Die Idee, speziell diese beiden Technologien zu vergleichen, ist aus zweierlei Gründen entstanden. Zum einen stand eine bei Tenovis entwickelte, auf Corba basierende Software als Grundlage für diese Studienarbeit zur Verfügung (der Service-Scheduler). Es sollte nun versucht werden, diesen Service-Scheduler zu erweitern bzw. zu verbessern. Da zum anderen in den letzten Jahren die vielversprechende Web Service Technologie auftauchte, bestand das Interesse, einen auf der dieser Technologie basierenden Service-Scheduler zu entwickeln und diesen mit der Corba-Variante zu vergleichen.

Der Fokus bei diesem Vergleich liegt auf den verwendeten Middlewaretechniken, d.h. es wird vorwiegend die Kommunikation zwischen verteilten Software-Komponenten betrachtet. Andere Fragestellungen, die bei der Entwicklung verteilter Anwendungen auftreten, werden nur am Rande erwähnt (Benutzungsschnittstelle, SW-Deployment). Der Vergleich zwischen Corba und der Web Service Technologie erfolgt nach verschiedenen Gesichtspunkten (Implementierungsaspekte, Kommunikationsmodelle, Interoperabilität, lose Kopplung, Firewalls). Zusätzlich wird untersucht, welche der beiden Technologien geeigneter ist, um die für den Service-Scheduler aufgestellten Anforderungen umzusetzen.

Zur Verdeutlichung der Software-Architektur des Schedulers bzw. der dynamischen Aspekte bei der verteilten Kommunikation werden Diagramme der UML verwendet. Eine Einführung in UML ist zu finden bei Oestereich (vgl. [Oestereich 1999]).

1.2 Anwendungsgebiet

Der Bedarf an einen Service-Scheduler ist im Bereich von Software für Callcenter entstanden. Als Callcenter bezeichnet man im allgemeinen eine Abteilung einer Firma, die für den telefonischen Kontakt mit den Kunden zuständig ist. Dazu sind eigens für diese telefonische Kundenberatung ausgebildete Fachkräfte angestellt (z.B. für eine telefonische Kreditberatung in einer Bank). Die von der Firma Tenovis angebotene Software unterstützt die Abläufe in einem Callcenter mit dem Ziel eine möglichst gute Betreuung der Kunden zu erreichen. So sollte ein Kunde beim Anrufen nicht zu lange in einer Warteschlange verbleiben und möglichst schnell an einen Berater vermittelt werden. Dieser Berater sollte möglichst die zum Kunden passenden Qualifikationen haben und schnell auf die Daten zu diesem Kunden zugreifen können. Dieses Vermitteln an die passenden Berater und das zur

Verfügung stellen von Kundeninformationen wird durch entsprechende Software unterstützt. Da der in dieser Arbeit vorgestellte Service-Scheduler eine sehr allgemeine Software-Komponente ist, die mehr im „Hintergrund“ abläuft und nicht unmittelbar die Interaktion und Handhabung der gesamten Callcenter-Software betrifft, erfolgt an dieser Stelle keine detailliertere Beschreibung des Anwendungsgebietes Callcenter. Es wird nur der den Service-Scheduler betreffenden Ausschnitt beschrieben. Ausführlichere Beschreibungen des Anwendungsgebietes Callcenter sind zu finden in den Diplomarbeiten von Bleek und Nilius (vgl. [Bleek 1997], [Nilius 2002]).
Die Software für das Callcenter legt Informationen über Telefongespräche in einer Datenbank ab (z.B. welcher Kunde wie lange mit welchem Callcenter Mitarbeiter gesprochen hat oder wie häufig ein Kunde in einer Warteschleife aufgelegt hat). Um diese Informationen geeignet aufzubereiten, sind verschiedene Komponenten entwickelt worden. Es existiert beispielsweise eine Komponente, die die Daten im Html-Format aufbereitet, und eine weitere, die eine Tabelle im Excel-Format generiert. Es bestand die Anforderung, dieses Aufbereiten von Statistikdaten automatisch und zeitgesteuert durchführen zu können. Die Funktionalität der automatischen, zeitgesteuerten Ausführung sollte nicht für jede einzelne Komponente neu entwickelt werden, statt dessen sollte eine einzige möglichst wiederverwendbare Lösung geschaffen werden. Diese Lösung soll nicht angepasst werden müssen, falls weitere Komponenten in die zeitliche Steuerung einbezogen werden. Aus diesen Erwägungen heraus ist die Idee des Service-Schedulers entstanden. Der Scheduler ist eine zentraler Prozess, der andere Software-Komponenten zeitgesteuert beauftragen kann, eine bestimmte Aufgabe durchzuführen.

Im Folgenden und in der gesamten Studienarbeit wird der Begriff des „Service“ verwendet. Dieser Begriff soll in dem Sinne verstanden werden, dass eine Software-Komponente anderen Komponenten eine bestimmte Funktionalität (Dienstleistung) zur Verfügung stellt. Dieses zur Verfügung Stellen eines Service erfolgt über eine bestimmte Schnittstelle, die dann von dem Service-Scheduler angesprochen werden kann. Der Begriff „Service“ bezeichnet also im Rahmen dieser Studienarbeit eine Server-seitige Software-Komponente und ist nicht mit dem Begriff des „Web Service“ zu verwechseln. Ein „Service“ kann mit verschiedenen Technologien implementiert werden, z.B. mit Corba oder als Web Service.

Der Scheduler soll also verschiedene Services zeitgesteuert beauftragen eine bestimmte Aufgabe zu erledigen. Dazu sollen die Informationen, wann der Scheduler welchen Service beauftragen soll, über ein komfortables GUI eingegeben werden. Der Benutzer muss neben den Einstellungen für den Scheduler (wann welchen Service aufrufen) auch entsprechende Parameter für die Services eingeben. Der oben angesprochenen Service zum Erstellen von HTML-Statistiken benötigt beispielsweise einen Parameter, der den Zeitraum angibt über den die Daten aufbereitet werden sollen. Es existiert ebenfalls ein Service, der die erstellten Statistiken per E-Mail verschicken kann. Dieser „Mailing-Service“ benötigt als Parameter natürlich die E-Mail Adresse des Empfängers (gewöhnlich ein leitender Angestellter des Callcenters, der sich für die Statistiken interessiert). Dieses Eingeben der Parameter soll nicht zwingend auf dem Rechner stattfinden, auf dem der Scheduler läuft, weshalb ein eigenständiges, entferntes Werkzeug als Benutzungsschnittstelle erforderlich ist. Dieses Werkzeug (im Folgenden GUI-Client genannt) ist als eigenständige Applikation konzipiert, es entsteht also eine Client-Server Architektur zwischen GUI-Client und Scheduler (zum Begriff Client-Server vgl. [Rechenberg & Pomberger 1999, S.666])

Mit der Kommunikation zwischen GUI-Client und Scheduler bzw. zwischen Scheduler und den Services ergibt sich also eine zweifache Client-Server Architektur. Folgende Abbildung stellt die entstehende Architektur mit Scheduler, Services und GUI-Client dar.

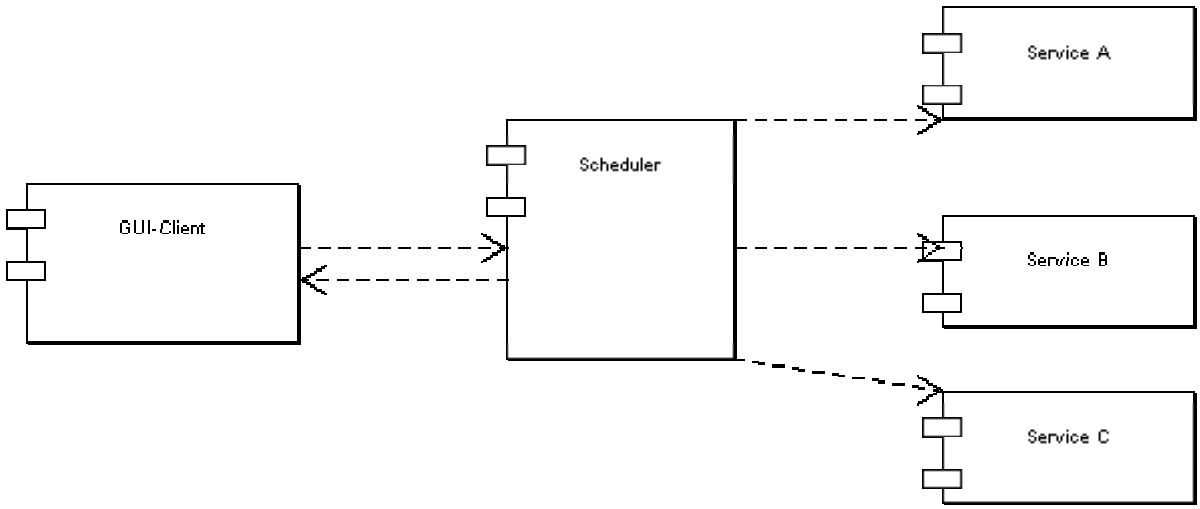

Abb 1.1: Architektur mit Scheduler, Services und GUI-Client

Es ist zu erkennen, dass die Kommunikation zwischen GUI-Client und Scheduler in beide Richtungen verläuft. Dies soll verdeutlichen, dass beim Scheduler auftretende Zustandsänderungen dem GUI-Client mitgeteilt werden sollen, damit er diese an am GUI anzeigen und den Benutzer darüber benachrichtigen kann. Dies können z.B. Ereignisse sein wie „Service A ist gerade beauftragt worden“ oder „Service C ist mit seinem Auftrag fertig“. (zu den detaillierten Anforderungen siehe Kapitel 3.1 und 4.1)

Der Schwerpunkt bei dieser Studienarbeit liegt in der Betrachtung der beiden Client-Server Architekturen, also der Kommunikation zwischen GUI-Client und Scheduler bzw. zwischen Scheduler und den Services. Die Umsetzung der GUI wird ausgeklammert und nur am Rande erwähnt.

1.3 Gliederung der Arbeit

In Kapitel 2 erfolgt eine Einführung und ein Vergleich der beiden Middlewaretechnologien Corba und Web Services. In den folgenden Kapiteln werden die eingeführten Technologien auf den Service-Scheduler bezogen. In Kapitel 3 wird der schon existierende Corba-Scheduler beschrieben, wonach in Kapitel 4 eine auf der Web Service Technologie basierende Variante mit ihren Vor- und Nachteilen gegenüber der Corba-Variante vorgestellt wird. Ein Fazit, abschließende Bemerkungen und Anknüpfungspunkte für weitere Forschungsfragen stehen in Kapitel 5 und 6.

2 Einführung in die Middlewaretechnologien Corba und Web Services

Dieses Kapitel gibt eine Einführung in Corba und den Web Service Technologie. Nach einer sehr allgemeinen Charakterisierung von Middleware erfolgt eine detaillierte Beschreibung über die Funktionsweise der beiden Technologien, wobei die Einführung in Web Services etwas ausführlicher ausfällt, da sie die neuere Technologie ist und über Corba schon sehr viel geschrieben wurde. Abschließend erfolgt ein Vergleich der beiden Technologien nach verschiedenen Gesichtspunkten.

2.1 Middleware für die Entwicklung verteilter Anwendungen

Middlewterstützt und vereinfacht die Realisierung von verteilten Anwendungen. Sie ist eine Softwareare un -Schicht, die zwischen den miteinander kommunizierenden Prozessen und deren jeweiliger Systemplattform (Hardware und Betriebssystem) angesiedelt ist.
(vgl. [Rechenberg & Pomberger 1999, S. 786])

Um Merkmale für eine Klassifikation einer Middleware zu haben, werden an dieser Stelle nur kurz die Bereiche erwähnt, in denen eine Unterstützung durch eine Middleware erfolgen kann. Eine ausführliche Einführung in komponentenbasierte Softwareentwicklung ist z.B. bei Griffel nachzulesen (vgl. [Griffel 98]). Unterstützung für die Entwicklung verteilter Systeme durch Middleware findet üblicherweise in folgenden Bereichen statt:

- Abstraktion von Details der Systemplattform und der Übertragungsprotokolle. Middleware setzt auf System-Schnittstellen und -Protokollen (z.B. TCP/IP) auf und definiert ein eigenes Protokoll.
- Es wird ein Mechanismus bereitgestellt, über den die zu übertragenden Daten aus einer Darstellung in einer Programmiersprache in das Middleware-eigene Protokoll überführt werden und beim Empfänger umgekehrt wieder in eine Programmier-sprachendarstellung. Dadurch können verteilte Anwendungen unabhängig von den meist heterogenen Systemplattformen entwickelt werden.
- Middleware unterstützt ein oder mehrere Programmiermodelle. Dies können z.B. der entfernte Prozeduraufruf (remote procedure call, RPC), ein verteilter objektorientierter Ansatz (distributed object-oriented, DOO) oder ein auf Versenden von Nachrichten basierender Ansatz sein (message oriented middleware, MOM). (vgl. [Rechenberg & Pomberger 1999, S. 697-702])
- Middleware unterstützt bestimmte Kommunikationsmodelle (Client-Server, Peer-to-Peer). In Peer-to-Peer Systemen fungieren die Kommunikationspartner gleichzeitig als Server und als Client.
- Middleware unterstützt bestimmte Kommunikationsarten (synchrone bzw. asynchrone Kommunikation). Die Begriffe „synchron“ bzw. „asynchron“ beziehen sich auf den Kontrollfluss der verteilten Anwendung. Ein synchroner entfernter Aufruf bezeichnet einen Aufruf, bei dem der Client-Kontrollfluss solange anhält bis die Antwort vom Server erfolgt ist. Der Client blockiert also nach Aufruf des Servers. Im Gegensatz dazu blockiert der Client bei asynchronen Server-Aufrufen nicht und fährt in seinem Programmfluss weiter fort (vgl. [Rechenberg & Pomberger 1999, S. 696]).
- Dienste, die das Auffinden bzw. die Identifikation entfernt aufzurufender Prozesse ermöglichen (Namens- bzw. Maklerdienste)
- Dienste, die Unterstützung für Multithreading, Transaktionen, Sicherheit etc. bieten.

2.2 Corba

Corba (Common Object Request Broker Architecture) ist eine Spezifikation des Standardisierungsgremiums OMG (Object Management Group). Die erste Corba-Spezifikation entstand 1991 mit dem Ziel, einen Standard für eine objektorientierte, programmiersprachen- und betriebssystemunabhängige Middleware zu schaffen. Im Laufe der Jahre wurden einige Lücken der Spezifikation geschlossen (u.a. standardisiertes Kommunikationsprotokoll), so dass eine anfangs nicht vorhandene Interoperabilität zwischen verschiedenen Corba-Produkten erreicht werden konnte. Mittlerweile existieren zahlreiche kommerzielle und frei erhältliche Implementierungen der Spezifikation und Corba ist als etablierter Standard zu bezeichnen. Grundlage für die Betrachtungen in dieser Arbeit ist die 2002 verabschiedete Spezifikation in der Version 3.0.

Gegen Ende der 90er Jahre entstand die Hoffnung (Erwartung), dass mit Corba eine verbreitete Integration von Softwarekomponenten nicht nur im firmeneigenen Intranet, sondern auch übers Internet stattfinden würde (Object Web). Die Verwendung von Corba als Middleware in der Software-Industrie beschränkt sich jedoch auf verteilte Anwendungen im Intranet, eine Integration von Software-Komponenten übers Internet mittels Corba ist eher selten anzutreffen. Als Gründe dafür sind folgende Punkte zu nennen:

- Die eingangs erwähnte fehlende Interoperabilität zwischen Corba-Produkten verschiedener Hersteller setzte für ein Verwenden von Corba eine Beschränkung auf ein Produkt voraus. In einem Firmeneigenen Intranet ist eine Einigung auf ein Corba-Produkt wesentlich einfacher zu erreichen als über Firmengrenzen hinweg.
- Als Standardeinstellung von Intranet-schützenden Firewalls hat sich durchgesetzt, dass das von Corba verwendete Übertragungsprotokoll IIOP (Internet Inter-ORB Protokoll) im Gegensatz zu Http nicht durchgelassen wird. Des Weiteren beeinträchtigen Portfilterung und NAT (Network Address Translation) von Firewalls den Einsatz von Corba. Es gibt Bemühungen ein Firewall-Tunneling über Http zu ermöglichen oder die NAT-Problematik zu umgehen, was aber zum einen zu herstellerabhängigen Lösungen führt, zum anderen mit Einschränkungen im Funktionsumfang von Corba einhergeht.
- Werden auf Client-Seite eigenständige, über Middleware mit einem Server kommunizierende Applikationen eingesetzt, müssen diese erst installiert werden und bei Änderungen an der Software aktualisiert werden. Auf Grund dieses Deployment-Problems werden im Bereich der GUI-Applikationen vorwiegend auf HTML basierende, im Browser ablaufende Client-Anwendungen eingesetzt.

Im Folgenden wird die Corba-Technologie vorgestellt. Dazu wird zunächst die Rolle der Schnittstellenbeschreibungssprache IDL und des Object Request Brokers ORB beschrieben. Nach kurzen Anmerkungen zu den von Corba unterstützten Datentypen und des Übertragungsprotokolls IIOP folgt abschließend eine Beschreibung der von einer Corba-Middleware angebotenen Dienste. Eine detaillierte Einführung in Corba sowie der Programmierung mit Java ist bei Brose nachzulesen (vgl. [Brose, Vogel, Duddy 2001]). Eine Gegenüberstellung von Corba, Soap und der Http-Kommunikation über die Servlet-Technologie ist bei Eberhart und Fischer zu finden (vgl. [Eberhart, Fischer 2001]).

2.2.1 IDL

Corba ist eine Middleware zur Realisierung von verteilten, objektorientierten Anwendungen. Unter Verwendung von Corba wird es einer Applikation ermöglicht auf einem anderen Rechner liegende Objekte zu referenzieren und mit diesen über Methodenaufrufe zu kommunizieren. Dabei können die entfernt angesprochenen Objekte in einer anderen Programmiersprache implementiert sein, es muss nur ein auf die jeweilige Programmiersprache zugeschnittenes Corba-Produkt existieren.

Diese Programmiersprachenunabhängigkeit wird erreicht durch eine für Corba spezifizierte, deklarative Schnittstellenbeschreibungssprache, die IDL (Interface Definition Language). Über diese IDL werden die auf Server-Seite anzusprechenden Objekte mit ihren Methoden und den dazugehörenden Datenstrukturen definiert. Ein zu dem auf Server-Seite eingesetzten Corba-Produkt gehörender IDL-Compiler generiert aus der IDL-Definition Objekte in der Server-Programmiersprache (die Skeletons). Die Skeletons Entpacken einen über das Corba-Protokoll IIOP hereinkommenden Aufruf (Demarshalling, Deserialisierung) und leiten diesen an die zugehörigen Objekt-Implementierungen (Servants) weiter. Auf Client-Seite werden mittels des IDL-Compilers des dort eingesetzten Corba-Produktes entsprechend Objekte in der Client-Programmiersprache erzeugt (die Stubs), welche das Aufbereiten entfernter Methoden-Aufrufe für die IIOP-Kommunikation vornehmen (Marshalling, Serialisierung). Der durch den Client getätigte entfernte Methodenaufruf muss nicht zwingend über statische Stubs erfolgen, er kann auch über ein vom Corba-Produkt angebotenen Mechanismus dynamisch zu Laufzeit zusammengestellt werden (Dynamic Invokation Interface).

Bei der Definition von Methoden kann angegeben werden, ob sie synchron oder asynchron aufzurufen sind. Eine Methodendefinition ohne weitere Angabe bedeutet einen synchronen Aufruf, d.h. der Kontrollfluss kehrt erst wieder zum Client zurück, wenn die Methode Server-seitig abgearbeitet und durchlaufen ist. Als One-way deklarierte Methoden ohne Antwort vom Server sind asynchron und blockieren den Kontrollfluss des Clients nicht. In Verbindung mit der Möglichkeit Callbacks zu nutzen (siehe Kapitel 2.2.2) stehen einem Entwickler alle denkbaren Kommunikationsarten zur Verfügung.

Durch diese einheitliche Schnittstellenbeschreibung auf Basis der objektorientierten IDL bzw. deren Datentypen sowie der Verwendung des spezifizierten Übertragungsprotokolls IIOP wird die betriebssystem- und programmiersprachenunabhängige Kommunikation zwischen verschiedenen Corba-Produkten ermöglicht.

2.2.2 ORB und entfernte Objektreferenzen

Um die oben beschriebenen Methodenaufrufe durchführen zu können, muss ein Client eine Referenz auf das aufzurufende entfernte Objekt besitzen. Referenzen auf entfernte Objekte können in Corba über den Namingservice geholt werden oder vom Server als Resultat eines Methodenaufrufs geliefert werden (z.B. eine Factory-Methode). Der Namingservice ist ein Dienst, an dem Corba-Serverobjekte zu registrieren sind und damit für Clients referenzierbar werden.

Um dieses programmiersprachenunabhängige Übergeben von Referenzen auf entfernte Objekte zu ermöglichen, wird jedem Skeleton bzw. Stub-Objekt inklusive der dazugehörenden Servants eine Corba-interne, interoperable Objekt Referenz (IOR) zugeordnet. Für die korrekte Zuordnung zwischen Stubs und Skeletons für entfernte Methoden-Aufrufe, d.h. der Lokalisierung des aufzurufenden Objektes im Netz und die Vermittlung der zu übertragenden Daten an dieses, ist der Objekt Request Broker (ORB) zuständig. Der ORB nimmt die serialisierten Daten der Stubs entgegen und überträgt (vermittelt) diese dann über IIOP an den ORB auf dem Server-Rechner, auf dem der Aufruf an das entsprechende Server-Objekt vermittelt wird. Dem Entwickler einer verteilten Anwendung bleibt dieser Kommunikationsmechanismus über den ORB inklusive der Details des Übertragungsprotokolls verborgen, er arbeitet nur auf Basis der definierten Schnittstelle in Gestalt der generierten Stubs bzw. Skeletons.

Als Bindeglied zwischen dem Server-Programm und dem ORB ist ein Objekt Adapter vorhanden, der für die Erzeugung und Verwaltung der internen Objekt Referenzen auf die einzelnen Servants zuständig ist. Um ein Server-Objekt beim Namingservice anzumelden, muss für dieses eine interoperable Objekt Referenz vom Objekt Adapter geholt werden, welche dann zusammen mit dem Server-Objekt beim Namingservice registriert wird. Es sind

zwei Objekt Adapter für Corba spezifiziert, der ältere Basic Object Adapter (BOA) und der später spezifizierte Portable Object Adapter (POA), welcher eine Migration von Server-Programmen auf ein anderes Corba-Produkt erleichtert.

Durch diesen Kommunikations-Mechanismus über den ORB mit der Möglichkeit entfernte Referenzen zu Erzeugen und zu übertragen ist ein wichtiges Merkmal von verteilter objektorientierter Middleware erfüllt. So ist es z.B. möglich Objektreferenzen über eine Factory-Methode zu liefern oder das Observer-Muster über einen Callback-Mechanismus zu realisieren, wodurch die starre Unterscheidung in Client und Server aufgehoben werden kann. Für diesen Callback-Mechanismus erzeugt der Beobachter auf Basis einer entsprechend definierten IDL ein Beobachter-Objekt und übergibt dem zu beobachtenden Server eine Referenz auf dieses Objekt über eine entsprechende Methode. Der beobachtete Server kann nun, immer wenn ein entsprechendes Ereignis auftritt, über die Referenz auf das Beobachter-Objekt den Client benachrichtigen (zu den Entwurfsmustern „Beobachter“ und „Fabrik“ siehe [Gamma et al. 1996]).

2.2.3 Datentypen

Für die Definition der Schnittstelle über die IDL sind die für die Methodenparameter und Objektattribute benutzbaren Datentypen spezifiziert. Dazu gehören neben einfachen Datentypen wie Integer, Strings oder Enumerations auch zusammensetzbare Datentypen wie Structs, Unions oder Arrays. Zur Ausnahmebehandlung von Methodenaufrufen sind eigene Ausnahmetypen definierbar.
Neben diesen Datentypen, die eine statische Typisierung zur Kompilierzeit erzwingen, gibt es in Corba auch die Möglichkeit einer dynamischen Typisierung über den Datentyp „Any“. Ist ein Parameter mit dem „Any-Type“ deklariert worden, kann dieser zur Laufzeit Werte jeden Datentyps annehmen.
Das zur Speicherung und Austausch von Daten immer häufiger verwendete XML-Format wird von Corba nicht direkt unterstützt. Ein komplettes XML-Dokument kann aber natürlich als String-Parameter übertragen werden.

2.2.4 IIOP

Um eine interoperable Kommunikation zwischen verschiedenen ORB-Herstellern zu gewährleisten, wurde für Corba das abstrakte General Inter-Orb Protokoll (GIOP) spezifiziert. Für dieses Protokoll ist definiert, wie die entfernten Methoden-Aufrufe inklusive der zugehörigen Datentypen auf ein binäres Format abzubilden sind. GIOP ist unidirektional verbindungsorientiert angelegt, d.h. Anfragen gehen immer vom Client aus, der Server antwortet nur. Zur Ermöglichung der oben erwähnten Callbacks muss auf Client-Seite also ebenfalls ein GIOP-Server laufen.
Für die tatsächliche Übertragung wird das abstrakte GIOP auf ein vom Betriebssystem angebotenes Protokoll abgebildet. Die Abbildung von GIOP auf TCP/IP wird Internet Inter-Orb Protokoll (IIOP) genannt. IIOP ist der spezifizierte Standard für die Kommunikation zwischen den einzelnen ORBs.
Der Entwickler einer verteilten Anwendung muss keinerlei Kenntnis über das Protokoll haben, er entwickelt ausschließlich auf Basis der mittels der IDL definierten Schnittstelle, weswegen an dieser Stelle keine Details des Protokolls besprochen werden.

2.2.5 Corba-Dienste

Zusätzlich zu der IIOP-Kommunikation über den ORB bzw. der Schnittstellenbeschreibung durch die IDL sind für Corba noch zusätzliche Dienste spezifiziert. Diese Dienste unterstützen die Entwicklung von verteilten Anwendungen, stellen Lösungen für immer wiederkehrende Probleme bereit und erhöhen bei Benutzung dieser Services die Portabilität bzw. die Wiederverwendbarkeit einer Komponente. Hier sollen kurz die wichtigsten Services

erwähnt werden, welche direkt die Kommunikation auf der Ebene der verteilten Objekte unterstützen (Objekt-Services):

- Naming Service: Über den oben schon erwähnten Naming-Service können Server-Objekte über einen registrierten Namen aufgefunden werden. Der Client wendet sich an diesen zentralen Dienst, bekommt eine Referenz auf das entfernte Server-Objekt und kommuniziert mit diesem ohne zu wissen, auf welchem Rechner sich dieses Server-Objekt befindet (Ortstransparenz).
- Trading Service: Der Trading Service dient ebenfalls dem Auffinden von Server-Objekten und bietet das Suchen nach Objekten an, die eine bestimmte Schnittstelle implementieren.
- Event Service: Der Event Service bietet eine spezifizierte Lösung für einen Ereignis-Mechanismus. Es können sich mehrere Clients bei einem Server für die Benachrichtigung über ein Ereignis registrieren. Die Benachrichtigung erfolgt über einen einzigen Event-Kanal, d.h. es werden immer alle Clients benachrichtigt.
- Notification Service: Da dem Event Service einige wichtige Merkmale fehlen, wurde später der Notification Service spezifiziert. Beim Notification Service können die Benachrichtigungen gefiltert werden, d.h. Nachrichten werden nur an bestimmte Clients verschickt. Zusätzlich kann auch die Güte der Benachrichtigung eingestellt werden (Quality of Service). Güte-Eigenschaften können z.B. eine unterschiedlich priorisierte, garantierte oder eine zeitgesteuerte Benachrichtigung sein.
- Transaction Service: Der Transaction Service bietet Unterstützung für verteilte, objektorientierte Transaktionsverwaltung, d.h. mehrere entfernte Methoden-Aufrufe sollen als Einheit entweder alle ausgeführt werden oder gar nicht, falls ein Aufruf fehlerhaft ist.
- Concurrency Control Service: Zur Steuerung des Verhaltens bei mehreren gleichzeitig durchgeführten Methoden-Aufrufen an einem Server-Objekt ist der Concurrency Control Service spezifiziert worden.
- Security Service: Zur Gewährleistung der Sicherheit einer verteilten Anwendung bietet der Security Service u. a. Mechanismen zur abgesicherten Kommunikation, Authentifikation und Autorisation.

Diese und eine Reihe von weiteren Services werden in Kombination mit den grundlegenden Corba-Spezifikationen (ORB, Objekt Adapter, IDL) als Object Management Architecture (OMA) bezeichnet.

2.3 Die Web Service-Technologie

Ende des Jahres 2000 tauchte ein neuer Modebegriff in der Softwareindustrie und der entsprechenden Presse auf, der (wie bei Corba oder anderen Technologien) die Lösung aller Integrationsprobleme versprach, der Begriff der „Web Services“. Da für diesen Begriff keine einheitliche Definition existiert und dieser eher durch Marketing der Softwareindustrie ins Leben gerufen wurde, ist er als etwas unscharf zu bezeichnen. Im Allgemeinen spricht man von einem Web Service, wenn folgende Punkte zutreffen:

- Es handelt sich um eine Softwarekomponente, die im Internet/Intranet zur Verfügung gestellt wird und sich über eine Schnittstelle in eigene Anwendungen einbinden lässt.
- Diese Schnittstelle wird über ein XML-basiertes Transportprotokoll angesprochen. Meist ist mit dem Protokoll das Simple Object Access Protocol (SOAP) gemeint. Es können aber auch andere Mechanismen wie XML-RPC oder der Representational State Transfer (REST) eingesetzt werden, die im Rahmen dieser Arbeit nicht betrachtet werden
- Dieses XML-basierte Protokoll setzt auf Internet-Standardprotokollen auf, so dass eine herstellerübergreifende Integration möglich ist (z.B. http).

Diese Interpretation des Begriffes „Web Service“ ist als allgemeingültig anzusehen und findet sich in der entsprechenden Literatur wieder (z.B. in [Cerami 2002] und [Chappell, Jewell 2002]).

Die Schnittstelle eines Web Service kann mittels der „Web Service Description Language“ (WSDL) beschrieben werden (siehe Kapitel 2.3.3). Zum Registrieren und zum Auffinden eines Web Services kann die standardisierte „Universal Description, Discovery and Integration“ (UDDI)-Methode eingesetzt werden. Häufig werden diese beiden unterstützenden Technologien als fester Bestandteil von Web Services betrachtet, meiner Ansicht nach sollte aber für eine Middleware, die SOAP, WSDL und UDDI unterstützt, ein eigener präziserer Begriff definiert werden. Der Begriff „Web Service“ beschreibt eher nur eine Server-seitige Komponente im Sinne der oben genannten Punkte.

Es folgt eine kurze Beschreibung der Möglichkeiten, die XML und XML Schema bieten, mit anschließender Einführung der darauf aufbauenden Web Service Technologien. Dabei werden die Schwerpunkte im Kontext der im Rahmen dieser Arbeit betrachteten Anwendung (der Service Scheduler) gesetzt.

2.3.1 XML & XML-Schema

Mit der 1998 vom W3C spezifizierten Auszeichnungssprache XML (Extensible Markup Language) ist es möglich neben den eigentlichen Nutzdaten zusätzlich eine semantische Beschreibung der Nutzdaten in Dokumenten abzulegen. Das XML-Format ist textbasiert, somit plattform- und programmiersprachenunabhängig und eignet sich deswegen sehr gut zum Austausch von Daten in heterogenen Umgebungen, insbesondere über das Internet. Die semantische Strukturierung wird mit frei definierbaren, die Nutzdaten umschließenden Tags (Auszeichnungen) erreicht. Eine Einführung in XML geben u.a. Harold und Means (vgl. [Harold, Means 2002]).

Ist die semantische Strukturierung eines XML-Dokumentes bekannt, d.h. sind die in einem Dokument vorkommenden Tags vordefiniert, kann dieses Dokument durch entsprechende Parser maschinell ausgewertet werden (Extraktion der Nutzdaten, Validierung). Die standardisierten Parser (SAX, DOM) sind dabei so benutzbar, dass eine Software nur die für sie interessanten Nutzdaten verarbeiten kann, während eine zweite Software völlig andere Daten des gleichen XML-Dokumentes verwendet. Es ist also möglich, dass ein XML-Format ohne Anpassungen von verschiedenen Systemen verwendet wird.

Eine weitere Möglichkeit der maschinellen Verarbeitung ist die regelbasierte Transformation von XML-Dokumenten eines Formats in ein anderes. So können z.B. rein informative XML-Dokumente, die keine Darstellungsinformationen beinhalten, in das XHMTL-Format transformiert werden. Eine Einführung in die Benutzung der Parser und entsprechender APIs für die Programmiersprache Java gibt Brett McLaughlin (vgl. [McLaughlin 2001]).

Die Definition der semantische Strukturierung eines XML-Dokumentes erfolgt üblicherweise mittels der DTD (Document Type Definition) oder des XML-Schema Standards. Die DTD ist eine einfache Definitionssprache, mit der die in einem XML-Dokument erlaubten Elemente mit deren Unterelementen und Attributen definiert werden können (vgl. [Harold, Means 2002]). Der XML-Schema Standard bietet im Gegensatz zur DTD eine umfangreiche Unterstützung von Datentypen. So sind zahlreiche Typen für Zeichenketten, Zahlenwerte, zeitbezogene Angaben oder Listen vordefiniert, es können aber auch eigene komplexe Typen definiert werden. Des Weiteren kann die Kardinalität von Elementen sehr flexibel eingeschränkt werden. Mit dieser Unterstützung kann eine Typ-Überprüfung der Elemente bzw. Attribute eines XML-Dokumentes erreicht werden. XML-Schema bietet außerdem explizit Unter-stützung für Namensräume, Schema-Komposition und objektorientierter Konzepte wie Vererbung (zur Einführung in XML Schema vgl. [van der Vlist 2002]).

Das Verhältnis einer Definition der semantische Strukturierung mittels einer DTD oder XML-Schema zu einem dieser Definition genügenden XML-Dokument ist vergleichbar mit dem Verhältnis von Klassen zu Objekten in objektorientierten Programmiersprachen. Diese Analogie kann für ein XML-Databinding benutzt werden, indem durch entsprechende Werkzeuge aus einer XML-Schema- oder DTD-Definition eine Klasse in einer bestimmten Programmiersprache generiert wird. Objekte dieser Klasse können dann automatisch in XML-Dokumente transformiert werden und umgekehrt, so dass für Programmierer ein Datenaustausch über XML einfach zu implementieren ist (vgl. [McLaughlin 2001]).

2.3.2 SOAP

SOAP (Simple Object Access Protocol) ist ein Protokoll zum Austausch von strukturierten und typisierten Daten in verteilten Anwendungen. Die Protokollstruktur basiert auf XML, wodurch eine programmiersprachen- und plattformunabhängige Kommunikation ermöglicht wird. Zusätzlich zu den mittels XML strukturierten Daten können mit SOAP auch beliebige Binärdaten übertragen werden (SOAP-Attachments). Die Übertragung der SOAP-Dokumente und Binärdaten kann über ein beliebiges Transportprotokoll erfolgen, die SOAP-Spezifikation schreibt keine Bindung an ein bestimmtes Protokoll vor. In den bis jetzt existierenden SOAP-Implementierungen wird standardmäßig HTTP verwendet, es sind aber z.B. auch Implementierungen über SMTP oder die eine Protokollschicht niedriger liegenden TCP oder UDP denkbar.
Die erste Spezifikation von SOAP entstand im Jahre 1999. Die vom World Wide Web Consortium im Moment aktuellste veröffentlichte Version ist SOAP 1.2 und dient als Grundlage für diese Arbeit (zur Einführung in SOAP vgl. [Chappell, Jewell 2002] und etwas ausführlicher [Englander 2002]).

Aufbau eines SOAP-Dokuments

Grundsätzlich ist SOAP ein Ein-Weg-Protokoll. Die Spezifikation definiert lediglich, wie ein SOAP-Client einen Aufruf bei einem SOAP-Server tätigt, d.h. wie die Aufruf-Daten zu einem SOAP-Dokument verpackt werden müssen.
Ein SOAP-Dokument ist ein XML-Dokument (der SOAP-Envelope) und besteht aus einem optionalen Header-Element und dem Body-Element (siehe Abb.2.1).

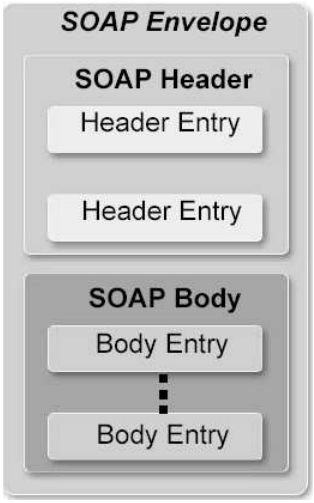

Abb. 2.1: Aufbau eines SOAP-Dokuments

Die zu übertragenden Nutzdaten stecken im SOAP-Body. Der SOAP-Header ist als Container für zusätzliche Informationen vorgesehen, die Spezifikation sagt nichts darüber aus, was dort platziert werden soll. Beispielsweise könnten im SOAP-Header Ids zur Realisierung von Sessions mitgeschickt werden. Die Information, wie der Server zu erreichen ist, steckt nicht im SOAP-Dokument sondern ist im darunterliegenden Transport-protokoll verpackt.

Dokumenten-Austausch vs RPC

Es gibt zwei Arten, wie ein SOAP-Client einen Aufruf beim SOAP-Server tätigen kann, das einfache Verschicken eines XML-Dokuments (Document style) oder die Durchführung eines entfernten Prozedur-Aufrufs (RPC style).
Beim Dokumenten-Austausch wird einfach ein komplettes XML-Dokument in den SOAP-Body gepackt und zum Server gesendet. Ein Programmierer auf Server-Seite muss sich darum kümmern, das dieses Dokument eigenständig verarbeitet und verwertet wird. Dieses Aufruf-Modell bedeutet eine lose Kopplung zwischen Client und Server, da keine Schnittstellen explizit definiert werden. Ein Aufruf für eine Bestellung könnte beim Dokumenten-Austausch beispielsweise folgendermaßen aussehen:

```
<env:Body>
        <m:Bestellung xmlns:m="einURI">
                <Produkt>Web Service Bücher</Produkt>
                <Menge>10</Menge>
        </m:Bestellung>
</env:Body>
```

Beim entfernten Prozedur-Aufruf werden die Nutzdaten im SOAP-Body auf eine Server-seitig definierte Prozedur abgebildet. Dazu muss im SOAP-Body als erstes Element der Name der Prozedur stehen, dessen Kindelemente sind dann die Parameter des Aufrufs. Damit diese Abbildung funktioniert muss der Client wissen, wie der SOAP-Body zusammengesetzt werden soll, d.h. er braucht eine Beschreibung der Server-Prozedur(en). Diese Beschreibung wird üblicherweise ebenfalls auf XML-Basis durchgeführt, als Standard hat sich WSDL durchgesetzt (siehe Kapitel 2.3.3). Der Client kann sich anhand der WSDL-Beschreibung den korrekten (d.h. der Prozedur-Signatur genügenden) SOAP-Body erzeugen. Dies kann statisch zur Kompilierzeit aber natürlich auch dynamisch zur Laufzeit erfolgen. Um die statische Variante zu vereinfachen existieren Werkzeuge, die aus einer WSDL-Beschreibung Implementierungen in einer bestimmten Programmiersprache generieren (die stubs), die das Zusammensetzen des korrekten SOAP-Bodys (Marshalling) hinter einem Aufruf in dieser Programmiersprache kapseln. Der Programmierer auf Client-Seite benötigt in diesem Falle keinerlei Kenntnisse über XML oder SOAP. Analog dazu

geschieht beim Empfang der Daten auf Server-Seite das Umgekehrte, der Inhalt des SOAP-Bodys wird durch entsprechende Implementierungen (die skeletons) ausgepackt (Demarshalling) und an die aufzurufende Prozedur delegiert.

Im Vergleich zum Dokumenten-Austausch geht bei SOAP-RPC die lose Kopplung zwischen Client und Server verloren, da die Aufrufe explizit auf eine fest definierte Schnittstelle abgestimmt sein müssen. Die oben erwähnte Beispiel-Bestellung könnte mit SOAP-RPC (Prozedur „bestelleProdukt“) folgende Struktur haben:

```
<env:Body>
        <m:bestelleProdukt xmlns:m="einURI">
                <Produkt>Web Service Bücher</Produkt>
                <Menge>10</Menge
        </m: bestelleProdukt >
 </env:Body>
```

Bei beiden Aufruf-Modellen stellt sich die Frage, wie der aufgerufene Server eine Antwort an den Client zurückschickt. Dieser Frage-Antwort Mechanismus ist zwar im SOAP-RPC Kapitel der Spezifikation beschrieben, ist aber grundsätzlich nicht Bestandteil von SOAP, das wie schon erwähnt lediglich ein Ein-Weg-Protokoll ist. Ein Zusammenhang zu einer evtl. Antwort ist in einem SOAP-Aufruf nicht vorhanden. Dieser Zusammenhang, d.h. bei SOAP-RPC die Definition eines Rückgabewertes in der Prozedur-Signatur und dessen Rücksendung an den Client oder beim Dokumenten-Austausch die Definition und Versendung eines Rückgabe-dokumentes, muss von der verwendeten Middleware hergestellt werden (z.B. mit WSDL, siehe Kapitel 2.3.3). Die Antwort ist dabei wie die Anfrage ein ganz normales Versenden eines SOAP-Envelopes. Ob die Anfrage und die darauf folgende Antwort über einen einmaligen Aufruf oder durch zwei getrennte Aufrufe abgewickelt wird, ist in SOAP nicht spezifiziert und hängt auch vom verwendeten, darunterliegenden Transportprotokoll ab. Bei der Verwendung von HTTP kann ein Anfrage-Antwort Mechanismus beispielsweise durch einen Aufruf erledigt werden (HTTP-Request und -Response).

Datentypen und Codierung

Bei SOAP-RPC müssen die Datentypen der mitgeschickten Parameter konform mit der Prozedur-Signatur auf Server-Seite sein. In der SOAP-Spezifikation ist eine Menge von Datentypen definiert, für die ein Anbieter für SOAP-unterstützende Middleware eine Abbildung auf die Typen der jeweiligen Programmiersprache implementiert, d.h. das Marshalling bzw. Demarshalling von bzw. nach XML ist damit spezifiziert. Die Codierung über diese spezifizierte Menge von Typen wird SOAP-Codierung oder auch Abschnitt-5-Codierung genannt (da sie in Abschnitt 5 der Spezifikation steht). Die SOAP-Codierung beinhaltet die einfachen Datentypen aus XML-Schema wie Integer, Gleitkommawert, Zeichenkette, Aufzählung und die nicht in XML-Schema vorkommenden Datentypen Array und Struktur.
Für die Übertragung mittels SOAP muss nicht unbedingt die SOAP-Codierung verwendet werden. Statt dessen können auch eigene Datentypen mit XML Schema definiert und für die Übertragung per SOAP verwendet werden, weswegen man hierbei von literaler Codierung spricht (literal encoding). Dies kann zu einem Interoperabilitätsproblem werden, da sich die Client- und Server-seitige Middleware auf eine Codierung der Daten einigen müssen. Diese Festlegung der Codierung kann über die Schnittstellenbeschreibung per WSDL erfolgen (siehe Kapitel 2.3.3).
Für die XML-Elemente, die einen Parameter eines RPC-Aufrufes beschreiben, kann der zugehörige Datentyp über ein Attribut (xsi:type) explizit angegeben und mit übertragen werden. Geht der Datentyp eines Elementes aus einer externen Beschreibung der Schnittstelle hervor (Identifizierung über den Elementnamen und eines Namensraumes), kann eine Angabe auch weggelassen werden.

SOAP mit Attachments

Zusätzlich zum Austausch der im XML-Format strukturierten Daten kann SOAP auch zum Versenden von beliebigen Binärdaten (z.B. Bilder oder Musik) verwendet werden. Dazu kommt das gleiche Modell wie beim Versenden von E-Mail-Anhängen zum Einsatz, das MIME-Format. Der SOAP-Envelope muss dabei als erster MIME-Part eingefügt werden, die zu versendenden Binärdaten werden einfach dahintergehängt (siehe Abb.2.2)

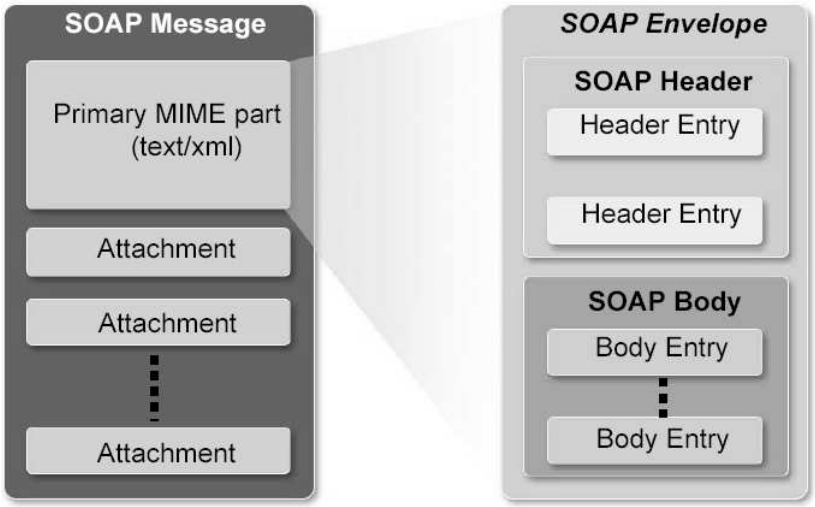

Abb. 2.2: Aufbau eines SOAP-Dokuments mit Attachments

SOAP-Faults

Ein SOAP-Aufruf kann aus verschiedenen Gründen misslingen. Neben der Nichterreichbarkeit des Servers aufgrund von Netzwerkproblemen können auch syntaktische bzw. semantische Fehler auftreten. So kann z.B. die Struktur des SOAP-Bodys oder die Datentypen der Parameter eines RPC-Aufrufs nicht der vom Server geforderten Signatur entsprechen.

Um auf solche Fälle standardisiert zu reagieren, ist in der SOAP-Spezifikation ein SOAP-Fault Element definiert, das als Reaktion auf Fehlerfälle als Antwort zurückgeschickt wird. Der SOAP-Body besteht dann nur aus diesem Fault-Element. Neben Standard-Fehlermeldungen (z.B. ungültige Nachricht, ungültige Codierung, Fehler beim Empfänger), die von der verwendeten Middleware implementiert werden, können auch eigene Fehlermeldungen im Sinne einer Ausnahmebehandlung definiert werden.

2.3.3 WSDL

Zur Beschreibung der Schnittstelle eines Web Services wurde vom World Wide Web Consortium die auf XML basierende WSDL (Web Services Description Language) spezifiziert (zur Einführung in WSDL vgl. [Chappell, Jewell 2002]).

Ein WSDL-Dokument besteht aus folgenden Elementen:

- Types Element: Zu jedem Element einer per RPC oder im Document-Style zu übertragenen XML-Nachricht gehört ein bestimmter Datentyp. Beschreibt das WSDL-Dokument einen auf SOAP basierenden Web Service, ist es üblich bei der Verwendung von einfachen Datentypen auf die in XML-Schema vordefinierten Typen zurückzugreifen (Angabe durch den Schema Namensraum und den Typ, z.B.: XSD:string), da diese in die SOAP-Spezifikation übernommen wurden. Sollen SOAP-spezifische, nicht in XML-Schema vorkommende Typen verwendet werden, müssen diese über den Namensraum der SOAP-Codierung angegeben werden (z.B. SOAP-ENC:array). Über das Types Element können nun aufbauend auf dieser Menge von Grundtypen eigene komplexe Typen definiert werden.

- Message Element: Die Deklaration der auszutauschenden XML-Nachrichten erfolgt über das Message Element. Eine hier deklarierte XML-Nachricht kann dann bei der Kommunikation mit dem Web Service in Parameter eines RPC-Aufrufes umgesetzt oder im Dokumenten-Stil verschickt werden. Eine Nachricht kann aus mehreren Teilen bestehen, welche bei Verwendung von RPC den einzelnen Parametern der aufzurufenden Methode entsprechen. Zur Deklaration der Teile einer Nachricht wird ein eindeutiger Bezeichner angeben und der zu verwendende Datentyp. Die Angabe der Datentypen erfolgt durch einen Verweis auf XML-Schema Typen, in SOAP definierte Typen wie Array oder auf die im Types Element definierten komplexen Typen.

- PortType Element: Über das PortType Element wird die eigentliche Schnittstelle des Web Service definiert. Es wird ein Bezeichner für diesen Typ von Web Service vergeben und die Menge der unter diesem Service aufrufbaren Operationen definiert. Dies ist vergleichbar mit der Definition einer Funktionsbibliothek oder einer Klasse inklusive ihrer Methoden in Programmiersprachen. Für die Definition einer Operation kann eine eingehende XML-Nachricht deklariert werden (Unter-Element Input) und/oder eine ausgehende XML-Nachricht (Unter-Element Output). Die hier anzugebenen XML-Nachrichten sind die im Message Element deklarierten. Für eine Operation können vier Aufruf-Verhaltensmuster deklariert werden, zwei vom Client des Web Service initiierte Aufrufarten (Anfrage-Antwort oder einen „one-way“ Aufruf ohne Antwort) und zwei vom Web Service initiierte (Aufforderung-Antwort oder eine Benachrichtigung ohne Antwort), bei denen der Client ebenfalls als Server fungiert. Mit WSDL wird nur die abstrakte Schnittstelle eines Web Service beschrieben und keine Annahmen über die Implementierung des Service gemacht. Deshalb ist eine WSDL-Beschreibung unabhängig davon, wie die aufgeführten vier Aufrufarten von einer Web Service Middleware umgesetzt werden und ob alle vier überhaupt angeboten werden. Unter Verwendung von Http als Übertragungsprotokoll müsste bei den vom Web Service initiierten Aufrufen auf Client-Seite ein Http Server laufen.

- Binding Element: Die Implementierung eines Web Service muss auf ein bestimmtes Transportprotokoll abgestimmt sein, d.h. der Web Service wird an ein bestimmtes Protokoll gebunden. Diese Bindung der im PortType Element definierten Operationen an ein Protokoll erfolgt über das Binding Element. Neben der Bindung an SOAP sind Bindungen für Http (Übertragung von XML ohne SOAP) und das MIME-Format spezifiziert. Wird der Web Service an SOAP gebunden, muss das tatsächlich zu verwendende Übertragungsprotokoll sowie die Aufrufart der Operationen (RPC oder Dokumenten-Austausch) angegeben werden. Die Angabe der Aufrufart ist wichtig, da

die Web Services Middleware einen RPC-Aufruf auf eine im Soap-Body angegebene Methode mit den entsprechenden Parametern abbilden muss, während dies beim Verschicken eines Dokumentes nicht der Fall ist (siehe Kapitel 2.3.2). Die Spezifikation der SOAP-Bindung verlangt neben der Angabe der Aufrufart zusätzlich die Angabe darüber, ob die SOAP-Codierung oder eine literale Codierung des SOAP-Aufrufes vorgenommen werden soll (siehe Kapitel 2.3.2). Es ergeben sich damit vier mögliche Kombinationen der SOAP-Übertragung: RPC/Soap-Encoded, RPC/Literal, Document/Encoded und Document/Literal.

- Import Element: Über das Import Element können Deklarationen aus anderen Dokumenten eingebunden werden. Diese Modularisierbarkeit von WSDL-Dokumenten fördert die Übersichtlichkeit und Wiederverwendbarkeit einzelner Deklarationen. So könnte z.B. das Types Element mit den eigenen Typ-Definitionen ausgelagert werden. Um Namenskonflikte beim Importieren zu vermeiden, wird jedem Import ein Namensraum zugeordnet.

- Service Element: Der Hauptzweck eines WSDL-Dokuments ist die Beschreibung einer abstrakten Schnittstelle ohne Angabe eines tatsächlich existierenden Web Service, der diese Schnittstelle implementiert. Über das Service Element kann zusätzlich ein tatsächlich vorhandener Web Service mit seiner URL angegeben werden, welcher die im PortType definierten Operationen implementiert.

Bei der Definition der Operationen im PortType Element sieht die WSDL-Spezifikation keine Möglichkeit vor, eine Operation als asynchron zu kennzeichnen, d.h. das Operationen in allen vier Aufrufarten synchron und für den Aufrufer blockierend sind (der Kontrollfluss kehrt erst nach der Server-seitigen Abarbeitung der Operation zum Aufrufer zurück). Asynchronität muss zusätzlich von einer Web Service Middleware durch entsprechende APIs angeboten werden, bei Java-basierten Systemen beispielsweise über die JAXM-API.

Üblicherweise wird eine WSDL-Beschreibung eines Web Service nicht per Hand eingegeben, sondern aus einer existierenden Implementierung eines Web Service in einer bestimmten Programmiersprache automatisch generiert. Für einen Client wiederum kann eine entsprechende programmiersprachliche Implementierung aus der WSDL generiert werden, so dass ein Entwickler im Idealfall nicht mit einem doch recht umfangreichen WSDL-Dokument in Berührung kommt.

2.3.4 Objekte und entfernte Objektreferenzen

Ein Client kann einen Web Service über die Angabe einer URL aufrufen. Diese URL kann entweder über das Service Element aus einer WSDL-Beschreibung oder per „Absprache“ unter den Entwicklern ermittelt werden. Es ist in Bezug auf SOAP oder WSDL kein eigenes Objektmodell mit integrierten entfernten Objektreferenzen spezifiziert. Da es keine entfernten Objekt-Referenzen gibt, ist es nicht möglich, ein Objekt, welches eine bestimmte Web Service-Schnittstelle implementiert, dynamisch zur Laufzeit zu erzeugen und dem Kommunikationspartner zwecks entfernter Aufrufe zur Verfügung zu stellen. Damit entfällt die Möglichkeit Callbacks oder einen Factory-Mechanismus auf spezifizierte und damit interoperable Weise durchzuführen. Eine Callback-Funktionalität wird nicht nur durch das Fehlen von entfernten Objektreferenzen, sondern auch durch die Verwendung von Http als Übertragungsprotokoll verhindert (siehe Kapitel 2.3.3).

Diese fehlenden, für eine verteilte objektorientierte Middleware typischen Merkmale müssten zusätzlich von einer Web Service Middleware oder vom Entwickler eines Web Service erbracht werden. Ein Factory-Mechanismus könnte relativ einfach realisiert werden, z.B. durch Übergabe einer URL, die auf einen weiteren Web Service verweist, oder eines WSDL-Dokuments, welches ein Service Element beinhaltet. Sollen verschiedene Instanzen einer

Web Service Schnittstelle ermöglicht werden (z.B. verschiedene Konto-Instanzen einer Konto-WSDL), könnte dies durch Übergabe einer Instanz-ID an einen „Konto-Verteiler“ geschehen, der den Aufruf dann an die eigentlichen Konto-Instanzen delegiert.

2.3.5 UDDI

Middleware für verteilte Systeme beinhaltet üblicherweise einen Naming- oder Trading-Service, über den die Server aufgefunden werden können. Nachdem die Server bei diesen Naming- bzw. Tradingservices angemeldet wurden, können Clients sich die Information holen, auf welchem Rechner ein bestimmter Server läuft und diesen dann aufrufen.

Um für Web Services Ähnliches zu erreichen, wurde durch eine Initiative von vielen großen Firmen UDDI (Universal Description, Discovery and Integration) spezifiziert. Allerdings ist das Ziel von UDDI ein etwas anderes und die Möglichkeiten dieser Spezifikation gehen weit über denen von Naming- und Tradingservices hinaus. In der Softwareindustrie besteht die Hoffnung, dass sich mit der Web Service-Technologie eine firmenübergreifende Integration von Geschäftsprozessen übers Internet entwickeln wird, einzelne Firmen also eine gewisse Funktionalität als Web Service anbieten, die dann in die Software-gesteuerten Abläufe einer anderen Firma eingebunden wird (Schlagwort B2B). UDDI bietet die Möglichkeit umfangreiche Informationen über den Serviceanbieter und den angebotenen Services abzulegen, wie z.B. Firmenadresse, Kontaktpersonen, Beschreibung der Firma (White Pages), eine Kategorisierung des Angebotes (Yellow Pages) und schließlich die technischen Details wie Service-URLs bzw. syntaktische und semantische Schnittstellenbeschreibungen (Green Pages).
UDDI basiert wie auch die anderen Web Service-Technologien auf XML. Die Informationen über Services werden in XML-Dokumenten abgelegt, deren Struktur mit Hilfe von XML-Schema definiert ist. Zum Registrieren und Auffinden von Informationen ist eine SOAP-basierte Schnittstelle spezifiziert.
Aufgrund des Umfangs von UDDI und der Tatsache, dass das Registrieren eines Web Services weit mehr als das einfache Veröffentlichen einer URL beinhaltet, wird diese Registrierungsmöglichkeit nicht für den in dieser Arbeit betrachteten Scheduler genutzt und von einer Beschreibung weiterer technischer Details von UDDI abgesehen (Die Verwendung von UDDI ist beschrieben in [Chappell, Jewell 2002] und [Cerami 2002]).

2.4 Vergleich der Web Service-Technologie mit Corba

Nach der Einführung in Corba und der Web Service-Technologie in Kapitel 2.2 und 2.3 werden in diesem Kapitel beide Middleware-Techniken nach verschiedenen Gesichtspunkten verglichen.

2.4.1 Implementierungsaspekte

Middleware soll die Entwicklung verteilter Anwendungen unterstützen, d.h. zum einen viele Möglichkeiten bieten und zum anderen das Programmieren durch Abstraktion von Systemplattformen und Protokollen vereinfachen.

Das Definieren der Server-Schnittstelle erfolgt bei Corba über die C++ ähnelnde IDL, welche vom Entwickler erlernt werden muss. Middleware für Web Service bietet üblicherweise den Mechanismus an, eine Schnittstelle direkt in der Web Service implementierenden Programmiersprache zu definieren. Zum Entwickeln eines Services ist eine Beschreibung über eine WSDL also nicht zwingend nötig. Diese kann aus der Web Service Implementierung generiert werden, so dass ein Entwickler im Idealfall nicht mit WSDL in Berührung kommt, was gegenüber Corba einen geringeren Entwicklungsaufwand bedeutet. Allerdings ist momentan (Herbst 2003) die Interoperabilität zwischen Web Service Middleware verschiedener Hersteller nicht immer gewährleistet (siehe Kapitel 2.4.2), so dass ein Entwickler auf eine interoperable Schnittstelle achten muss bzw. zum Schreiben eines Clients ein Interoperabilitätsproblem verstehen und sich deshalb mit SOAP und den sehr umfangreichen WSDL-Beschreibungen auskennen muss. Solange dies der Fall ist, ist das Abstraktionsniveau bei Web Services wesentlich niedriger als bei Corba. Bei Corba sind keinerlei Protokollkenntnisse nötig und die IDL wesentlich einfacher zu verstehen als WSDL.

Ein weiterer Unterschied zwischen einer IDL- und einer WSDL-Beschreibung besteht darin, dass IDL eine reine abstrakte Schnittstellenbeschreibung darstellt, während bei WSDL zusätzlich ein tatsächlich existierender Web Service inklusive der URL eingetragen werden kann und somit ein Auffinden über einen Namingservice umgangen werden kann.

Als Vorteil von SOAP wird häufig angebracht, das die über die Leitung geschickten Daten für Menschen lesbar sind und somit das Debugging erleichtert wird. Meiner Meinung nach entsteht der Bedarf für das Suchen von Fehlern auf Protokollebene doch erst dann, wenn die entsprechende Middleware keine vollständig abstrahierenden APIs und Interoperabilität mit anderen Produkten anbieten kann. In Corba wird die Frage nach Protokoll-Debugging gar nicht gestellt, weil der Bedarf gar nicht existiert.

2.4.2 Interoperabilität

Die Spezifikationen von Corba und SOAP bzw. WSDL haben eine Plattform- und Programmiersprachenunabhängigkeit zum Ziel, deren Umsetzung nur durch Interoperabilität zwischen Middleware verschiedener Hersteller zu erreichen ist.

Bei Corba gab es anfangs Interoperabilitätsprobleme, die mittlerweile behoben sind. Diese entstanden durch fehlende Spezifikationen u. a. für das Transportprotokoll. Bei der Web Service-Technologie liegt das Augenmerk hauptsächlich auf Protokollebene (SOAP) und der Schnittstellenbeschreibung über WSDL. Die SOAP-Spezifikation weist einige Ungenauigkeiten auf, die den Middleware Herstellern einen gewissen Interpretations-Spielraum bei der Implementierung ließ (Stichworte: optionale XSI:type-Angabe, SoapAction-Header, SOAP-Actor, vgl. [Chapell, Jewell]). Anfänglich dadurch entstandene Interoperabilitätsprobleme scheinen mittlerweile behoben. Ein Problem für die Interoperabilität stellt die Flexibilität von SOAP in Bezug auf die Kommunikationsart und die Datentypen dar. So ist die Voreinstellung mancher Web Service Middleware die Benutzung des Document-styles in Verbindung mit literaler Codierung (z.B. Microsofts .Net), während andere Werkzeuge die RPC-Verwendung mit SOAP-Codierung voreingestellt haben (z.B. Apache Axis), was zu unterschiedlicher

Serialisierung der Daten in ein SOAP-Dokument führt. Weiterhin führt die Verwendung von nicht in der SOAP-Codierung vorhandenen Datentypen in Verbindung mit RPC zu Problemen.
Verteilte Anwendungen erfordern häufig Mechanismen um Transaktionen oder eine Sessionverwaltung zu ermöglichen. Für diese Mechanismen existieren keine eindeutigen Spezifikationen, weswegen Lösungen dafür Herstellerabhängig und damit nicht interoperabel sind. Ein Objektmodell mit der Möglichkeit, entfernte Referenzen auf verschiedene Instanzen zu verwenden, ist ebenfalls nicht spezifiziert und müsste eigens nachgebildet werden. Diese zusätzlichen Dienste sind in Corba spezifiziert, wodurch Interoperabilität auch in diesem Bereich ermöglicht werden kann.

2.4.3 Lose Kopplung

Ein Aufruf entfernter Methoden über einen RPC-Mechanismus bedeutet zum einen Typsicherheit in Bezug auf die Parameter, zum anderen eine feste Kopplung zwischen Client und Server, da auf eine feste Schnittstelle hin programmiert wird. Die feste Kopplung kann durch Verwendung eines dynamisch typisierbaren Parameter (Any-Type) aufgehoben werden. Diese Möglichkeiten sind sowohl in Corba, als auch bei Web Services vorhanden. Die Wahl der geeigneteren Middleware für die Verwendung von RPC ist im Kontext weiterer Vergleiche zu treffen (Performanz, Interoperabilität, Kommunikationsmodelle, Implementierungsaufwand).
Eine noch losere Kopplung kann durch das Verschicken eines XML-Dokumentes über SOAP ohne Abbildung auf statisch deklarierte Parameter auf Server-Seite erreich werden. Hierbei muss kein Typ-Cast eines Any-Typs erfolgen, die exakte Datentyp also gar nicht bekannt sein. Mittels geeigneter XML-Abfragesprachen (XPath, Xquery) können die benötigten Daten aus dem XML-Dokument extrahiert werden. Dadurch kann ein Server beispielsweise verschieden strukturierte Dokumente verarbeiten und muss nicht so häufig angepasst werden. Bei Corba kann natürlich Ähnliches durch Übertragen von XML-Dokumenten als String erreicht werden.
Der Begriff lose Kopplung wird ebenfalls im Bereich von MoM-Produkten (message oriented middleware) verwendet, welche Nachrichtenfilterung und Quality of Service bieten. Insbesondere mit der asynchronen Nutzung dieser Dienste kann eine lose Kopplung von verteilten Systemen erreicht werden. Bei Corba ist diese Möglichkeit über den Notification Service spezifiziert worden. Im Bereich von Web Services gibt es keine derartige Spezifikation, weswegen nur herstellerabhängige Lösungen existieren.

2.4.4 Kommunikationsmodelle

In verteilten Anwendungen auf Basis von Corba können aufgrund der Möglichkeit entfernte Objektreferenzen inklusive Callbacks zu nutzen alle denkbaren Kommunikationsmodelle implementiert werden. Eine Strikte Einteilung in Client und Server muss nicht vorgenommen werden. Allerdings werden diese Möglichkeiten durch Firewalls eingeschränkt (siehe Kapitel 2.4.5).
Die Möglichkeit vom Server initiierte Aufrufe beim Client vorzunehmen, ist zwar in der WSDL-Spezifikation vorgesehen, bei Web Services steht aber die Verwendung von Http als SOAP-Übertragungsprotokoll im Mittelpunkt, weshalb Callbacks ohne Http-Server auf Client-Seite nicht möglich sind.

2.4.5 Firewalls

Für die Web Service-Technologie ist Http als Übertragungsprotokoll favorisiert, da dieses Protokoll nicht von Firewalls blockiert wird. So sollen Softwarekomponenten problemlos weltweit kommunizieren können und ähnliches Potential hervorrufen, das HTML in Verbindung mit Http für menschlich lesbare Informationen erbracht hat. Diese Möglichkeit

wurde mit Corba nicht erreicht, da hier die Probleme mit Firewalls relativ groß sind (siehe Kapitel 2.2).

An dieser Stelle muss man natürlich nach dem Sinn von Firewalls fragen, die doch unerwünschte Kommunikation verhindern soll. Ein Einbetten von Protokollen in Http verschleiert den Unterschied zwischen sicherheitskritischen Kommunikationsdaten und ungefährlichen Informationen. Firewalls könnten diese eingebetteten Protokolle erkennen und herausfiltern. Diese Aufgabe erscheint aber ähnlich schwierig als ein standardisiertes Verhalten von Firewalls in Bezug auf das Corba Protokoll IIOP zu erreichen. Vielleicht sollte für die Kommunikation über SOAP/Http ein anderer Standardport als der Port 80 festgelegt werden um eine strikte Trennung herbeizuführen.

2.4.6 Performanz

Die Performanz bei der Kommunikation in verteilten Systemen wird durch die zu übertragende Datenmenge und der Geschwindigkeit der Serialisierung/Deserialisierung der Daten beeinflusst. Die Kommunikation über SOAP ist wesentlich unperformanter als die über Corba, da die zu übertragende Datenmenge aufgrund der vielen XML-Tags relativ groß ist. Zusätzlich ist das Parsen der SOAP-Dokumente beim Deserialisieren und das Zusammenstellen derselben beim Serialisieren relativ aufwendig.

Ein anderer Performanz-Aspekt ergibt sich in Bezug auf die Anforderungen einer verteilten Anwendung und das verwendete Kommunikationsmodell. Soll z.B. eine Client-Komponente ständig über Zustandänderungen des Servers benachrichtigt werden, muss diese bei der Verwendung von Http als Übertragungsprotokoll ständig beim Server nachfragen (pollen). Bei Corba kann ein Beobachtermuster über Callbacks implementiert werden, so dass der Client bei Zustandsänderungen sofort benachrichtigt werden kann. Das ständige Pollen bei der Verwendung von Http hat zur Folge, dass zum einen die Netzwerklast erhöht wird und zum anderen der Client Zustandsänderungen evtl. nicht sofort bemerkt (je nach „Poll-Frequenz“).

3 Der Corba Service Scheduler

Im Folgenden wird der zum Vergleich der beiden Middleware Technologien herangezogene Scheduler auf Corba-Basis vorgestellt. Die Beschreibung geht dabei nicht zu sehr ins Detail, sondern nur so weit, wie es für das Verständnis der Art und Weise der Corba-Verwendung nötig ist. Die GUI-Komponente wird nur erwähnt, wenn die verteilte Kommunikation über Corba diese unmittelbar betrifft bzw. beeinflusst. Dabei ist zu berücksichtigen, dass einige Nachteile dieser Scheduler Version nicht unmittelbar auf die Verwendung von Corba zurückzuführen ist.

3.1 Anforderungen

Die in Kapitel 1.2 angedeuteten Anforderungen an den Service Scheduler werden nun etwas detaillierter dargestellt:

- Es sollen Services zeitgesteuert beauftragt werden können, ihre über eine Schnittstelle zur Verfügung gestellte Funktionalität durchzuführen. Die Services können dabei auf verschiedene Rechner verteilt sein.
- Die Parameter für diese Aufträge können sich in Syntax und Semantik stark unterscheiden (eine Generierung einer Statistik über Telefongespräche benötigt beispielsweise Angaben über den zu betrachtenden Zeitraum. Ein Service, der automatisch die generierte Statistik per E-Mail verschickt, benötigt eine E-Mail Adresse)
- Ein zentraler Scheduler soll die zeitgesteuerte Beauftragung der Services über deren Service-Schnittstelle durchführen. So sind die Service-Aufrufe zentral steuerbar und beobachtbar. Hinzu kommt, dass dadurch die Funktionalität der Zeitsteuerung nicht für jeden einzelnen Service implementiert bzw. angepasst werden muss.
- Damit der Scheduler flexibel einsetzbar ist, soll er nicht auf bestimmte Services zugeschnitten sein. Es soll möglich sein, weitere Services in die zeitgesteuerte Beauftragung einzubeziehen, ohne den Scheduler anpassen zu müssen.
- Es soll eine GUI-Komponente entstehen, über die der Benutzer neben den zeitbezogenen Einstellungen zu einem Auftrag auch die unterschiedlich strukturierten Service-Parameter eingeben kann. Die Benutzung dieser GUI-Komponente soll von einem beliebigen Rechner aus ermöglicht werden, Internet-weite Benutzbarkeit wäre optimal.
- Diese GUI-Komponente soll den aktuellen Zustand der Aufträge anzeigen, d.h. ob gerade ein Auftrag von einem Service bearbeitet wird, schon beendet ist (erfolgreich oder fehlerhaft) oder wann er das nächste Mal ausgeführt wird.
- Es soll nicht nur ein Service zu einem bestimmten Zeitpunkt aufgerufen werden können, sondern auch „Service-Sequenzen“, d.h. ein zweiter Service soll genau dann aufgerufen werden, wenn der erste beendet ist (z.B. zuerst eine Statistik erstellen, diese danach per Mail verschicken).
- Die Durchführung eines Auftrages durch einen Service kann unterschiedlich viel Zeit beanspruchen. So kann z.B. eine Generierung einer Statistik durchaus mehrere Stunden dauern, während das automatische Verschicken der Statistik per E-Mail durch einen weiteren Service natürlich in Sekunden erledigt ist. Diese Anforderung sollte bei der Implementierung des Schedulers beachtet werden, insbesondere bei der Realisierung von Service-Sequenzen. Evtl. Timeouts bei der Netzwerkkommunikation dürfen das Verhalten natürlich nicht beeinflussen.
- Ein durch einen Service produziertes Ergebnis (z.B. eine im Html-Format generierte Statistik) kann für einen nachfolgend aufgerufenen Service von Interesse sein. So muss der Mailing-Service auf die generierte Statistik zugreifen können, um sie als Anhang einer E-Mail zu verschicken. Ein Ergebnis soll also von einem Service zum einem weiteren übergeben werden können und kann ein beliebiges Dokument sein.
- Die Services sollen in unterschiedlichen Programmiersprachen entwickelt werden können.

3.2 Die Corba-Schnittstelle der Services

Es bestand die Anforderung, dass Services in die zeitgesteuerte Beauftragung des Schedulers einbezogen werden können, ohne den Scheduler anpassen zu müssen. Die Forderung nach dieser Flexibilität betrifft zwei Bereiche des Schedulers, zum einen die GUI-Komponente, über die die servicespezifischen Parameter eingetragen werden, zum anderen die Schnittstelle zu den Services.

Es wurde eine sehr einfach zu implementierende Lösung gewählt: Die Schnittstelle zu den Services besteht aus einem einfachen Methodenaufruf, dem eine einziger String übergeben wird. In diesen String sind dann alle Parameter enthalten, die der Service zur Ausführung seines Auftrages benötigt. Wie die Parameter in diesem String einzutragen sind, d.h. in welcher Reihenfolge sie zu stehen haben oder wie sie voneinander separiert werden, wird dem Scheduler nicht bekannt gemacht. Dadurch kann natürlich jeder Service ohne Anpassung des Schedulers beauftragt werden, sobald er diese eine Methode anbietet. Dementsprechend wurde an der GUI-Komponente nur ein einziges Textfeld zur Eingabe dieses Parameter-Strings vorgesehen. Die Verantwortung für das Zusammenstellen eines korrekten Parameter-Strings liegt somit beim Benutzer, der sich diese Information aus der Dokumentation des betreffenden Service holen muss. Um dem Scheduler signalisieren zu können, dass die übergebenen Parameter fehlerhaft sind, wird beim Aufruf der Service-Methode eine Exception mit entsprechendem Fehler-String ausgelöst.

Die Anforderungen, dass der aktuelle Zustand eines Service-Auftrags („ist gerade in Bearbeitung“ oder „Bearbeitung abgeschlossen“) an der GUI-Komponente angezeigt werden soll und dass der Aufruf von „Service-Sequenzen“ möglich sein soll, machen es erforderlich, dass der Scheduler genau mitbekommt, wann ein Auftrag beendet ist. Grundsätzlich existieren zwei Möglichkeiten, dies zu erreichen. Die erste Variante geht davon aus, dass ein Auftrag beendet ist, sobald der Kontrollfluss nach dem Aufruf der oben erwähnten Methode zum Scheduler Prozess zurückkehrt. Die aufgerufene Methode blockiert also bis zur Beendigung des Auftrages (synchroner Aufruf). Die zweite Variante wäre ein asynchroner Mechanismus, d.h. nach Aufruf der Methode kehrt der Kontrollfluss sofort zum Scheduler zurück, während der Service seiner Auftrag durchführt (siehe auch Kapitel 2.1 und 2.2.1). Nach Beendigung des Auftrages ruft der Service eine Methode am Scheduler zur Signalisierung der fertigen Bearbeitung auf. Gewählt wurde hier die erste Variante, trotz der Tatsache, dass Aufträge auch mehrere Stunden andauern können. Dazu wird im Scheduler zum Aufruf einer „Service-Sequenz“ ein Thread gestartet, der hintereinander die synchronen, blockierenden Methoden der Services aufruft. Folgendes Sequenzdiagramm in Abb. 3.1 soll diesen Ablauf verdeutlichen.

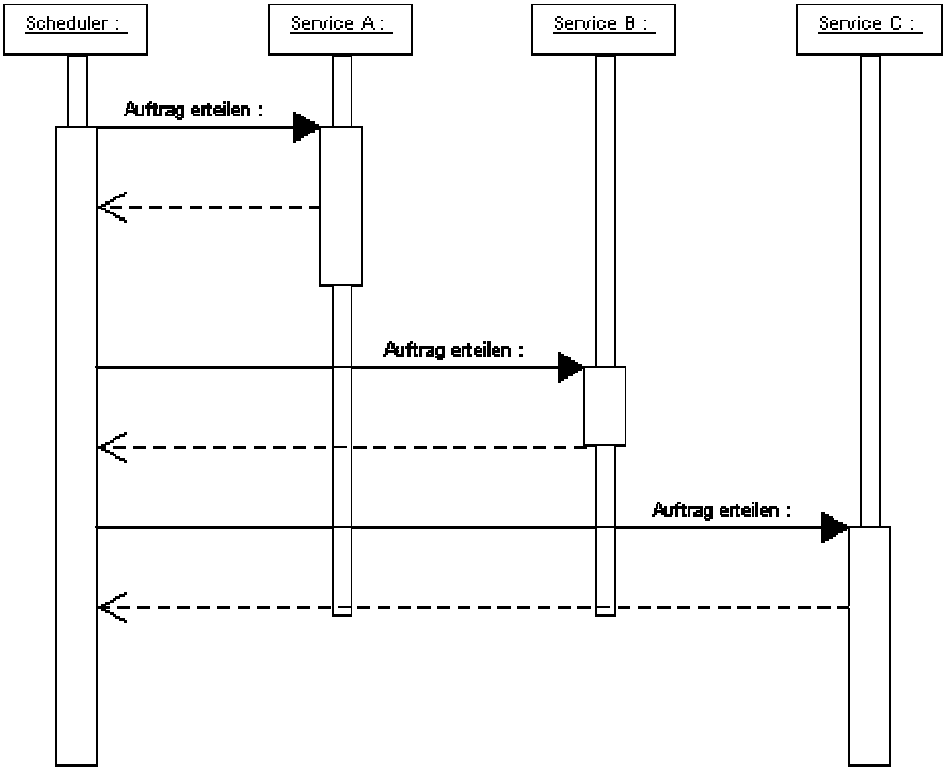

Abb. 3.1: Ablauf einer Service-Sequenz:

Für „Service-Sequenzen“ bietet die Service-Schnittstelle keine Möglichkeit ein Ausführungsergebnis eines Services dem nächstfolgenden zur Verfügung zu stellen. Die Möglichkeit eine Datei zu übertragen, bestünde unter Corba indem einfach die einzelnen Bytes übertragen werden (IDL-Typ sequence<octet>). Diese Möglichkeit wurde aus Zeitgründen nicht in Betracht gezogen, statt dessen wurde für das oben genannte Beispiel einer Statistikerstellung und nachfolgender Versendung per E-Mail folgende Lösung gewählt: Die beiden beteiligten Services haben Zugriff auf einen gemeinsamen Datenträger (laufen also auf demselben Rechner oder können auf ein gemeinsames Laufwerk im Netzwerk zugreifen), auf dem das Ausführungsergebnis (z.B. eine Statistik im Html Format) vom ersten Service abgelegt und vom zweiten Service wieder abgeholt wird. Neben der Tatsache, dass dieser Datenträger beiden Services bekannt sein muss, ist es erforderlich den Dateinamen der Statistik über den Parameter-String an der GUI-Komponente anzugeben.

3.3 Auffinden der Services

Da bei Corba Produkten ein Naming Service zur Verfügung steht (siehe Kapitel 2.2.5), sollte dieser auch zum Auffinden der Services verwendet werden. Der Naming Service ist ein eigenständiger Prozess, bei dem sich die Services unter einem eindeutigen Namen anmelden. Unter diesem Namen muss ein Service dem Scheduler bekannt gemacht werden, damit er beauftragt werden kann. Zum Zeitpunkt der Beauftragung eines Services holt der Scheduler sich anhand des Service-Namens eine Corba-Referenz auf den registrierten Service und versucht ihn aufzurufen. Der Scheduler stellt also erst zum Zeitpunkt der Ausführung eines Auftrages fest, ob dieser Name korrekt ist.
Die Angabe des Service-Namens erfolgt durch den Benutzer, der diesen neben den eigentlichen Parametern am GUI eingeben muss. Die Möglichkeit, dass der Scheduler alle Services beim Naming Service erfragt und diese dann als zur Verfügung stehende Services an der GUI-Komponente anzeigt, wurde nicht geschaffen.

3.4 Die Corba-Schnittstelle des Schedulers

Der Scheduler besteht aus zwei Komponenten, zum einen aus dem Scheduler Prozess, der die Aufträge verwaltet und schließlich die Services beauftragt, sowie zum anderen ein GUI-Werkzeug zum Einstellen und Überwachen von Aufträgen. Aufgrund der Anforderung die Aufträge zu überwachen und deren aktuellen Zustand anzuzeigen wurde von vornherein eine Web-basierte Html-Version ausgeschlossen. Das GUI-Werkzeug ist also eine eigenständige Applikation, die erst auf den Client-Rechnern installiert werden muss (GUI-Werkzeug als Client des Scheduler Prozesses). Im Folgenden werden die Funktionalitäten beschrieben, die die Scheduler-Schnittstelle dem GUI-Client zur Verfügung stellt.

3.4.1 Erstellen, Ändern, Löschen von Aufträgen

Den Methoden zum Erstellen und Ändern wird ein Auftragsobjekt übergeben, welches die zeitbezogenen Angaben enthält (wann und wie oft wird der Auftrag ausgeführt) und ein oder mehrere Services die als Sequenz hintereinander ausgeführt werden sollen. Die Auftragsobjekte werden dabei als Werteparameter übergeben, d.h. es wird nicht mit in Corba möglichen entfernten Objektreferenzen gearbeitet (denkbar wäre eine Schnittstelle, die für jeden Auftrag eine entfernte Referenz auf ein Auftragsobjekt bereitstellt, an dem dann Operationen durchgeführt werden könnte). Da die Aufträge als Werteparameter übergeben werden, ist jeder Auftrag zur eindeutigen Identifizierung mit einer ID versehen worden. Die Verwaltung und Erzeugung von Ids übernimmt der Scheduler-Prozess. Zum Erstellen eines neuen Auftrages muss sich der GUI-Client eine neue Auftrags-ID vom Scheduler besorgen, den Auftrag mit den vom Benutzer eingegebenen Werten zusammenstellen und dem Scheduler schicken. Folgender Ausschnitt aus der IDL-Definition der Scheduler-Schnittstelle zeigt die entsprechenden Datenstrukturen und Methoden:

Ausschnitt der IDL-Definition des Schedulers zum Erstellen von Aufträgen

```
//Schnittstelle des Schedulers für den GUI-Client
interface SchedulerIF
{
      long createNewTaskID();       // Erzeugen einer neuen Task-ID
      void addTask(in Task task)    // Hinzufügen eines neuen Auftrags
      ...
};

//benötigte Informationen zu einem Service-Aufruf
struct ServiceCall
{
      string serviceName;    // Name zur Anmeldung beim Naming Service
      string parameter;      // Parameter des Services als String
};

//Sequenz von Service-Aufrufen definieren
typedef sequence<ServiceCall> SequenceOfServiceCalls;

//benötigte Informationen zu einem Auftrag
struct Task
{
      long  id;
      SequenceOfServiceCalls sequenceOfServiceCalls;
      long startTime;    // Startzeit des Auftrags als Zeitstempel seit 1970
};
```

Definiert ist die Scheduler-Schnittstelle über das Interface „SchedulerIF“ mit den Methoden zum Erzeugen einer neuen ID bzw. zum Hinzufügen eines Auftrages am Scheduler. Ein Auftrag ist über die Struktur „Task“ definiert und wird mit der vorher erzeugten ID bzw. den Service-Aufrufen („SequenceOfServiceCalls“) zum Scheduler gesendet. Hat der Scheduler den Auftrag erhalten, ruft er zum angegebenen Startzeitpunkt die Service-Sequenz hintereinander auf. Der Startzeitpunkt ist als Zeitstempel in Sekunden seit dem 1.1.1970 angegeben.

3.4.2 Beobachten von Aufträgen

Um die Zustandänderungen von Aufträgen beobachten zu können, bietet der Scheduler ein Callback Mechanismus nach dem Beobachter-Muster an (siehe Kapitel 2.2.2). Dazu registriert sich ein GUI-Client als Beobachter für ein oder mehrere Aufträge und wird über Zustands-änderungen benachrichtigt. Eine Zustandänderung kann z.B. das Starten eines Service-Aufrufs sein. Der Folgende Ausschnitt aus der IDL-Definition der Scheduler-Schnittstelle zeigt die entsprechenden Methoden.

Ausschnitt der IDL-Definition des Schedulers zum Beobachter-Mechanismus

```
interface SchedulerIF
{
      ....
      void addObserver(in Observer observer);  // Beobachter anmelden
};

// Schnittstelle des Beobachters
interface Observer
{
      void notifyTaskStarted(long taskID);
      void notifyTaskFinishedSuccessful(long taskID);
      void notifyTaskFinishedWithError(long taskID, string errorText);
};
```

Der GUI-Client implementiert die Observer-Schnittstelle und meldet sich beim Scheduler über die „addObserver“-Methode als Beobachter an. Der Scheduler hat dadurch die entfernte Corba-Referenz auf das Observer-Objekt (den GUI-Client) und kann diesen über Ereignisse informieren. Zur Benachrichtigung über das Ende eines Auftrages gibt es eine Methode für die erfolgreiche und eine für die fehlerhafte Durchführung. Damit kann auf das Problem reagiert werden, dass der Benutzer den Parameter-String an dem GUI fehlerhaft eingeben kann. Fehlerhafte Parameter werden erst zum Zeitpunkt der Auftragsausführung vom Service erkannt und dem Scheduler über eine Exception mitgeteilt. Dieser kann schließlich den GUI-Client über die „notifyTaskFinishedWithError“-Methode benachrichtigen. Das folgende Sequenzdiagramm in Abb. 3.2 verdeutlicht einen typischen Ablauf eines Auftrags mit zwei Service-Aufrufen und Benachrichtigung des GUI-Clients.

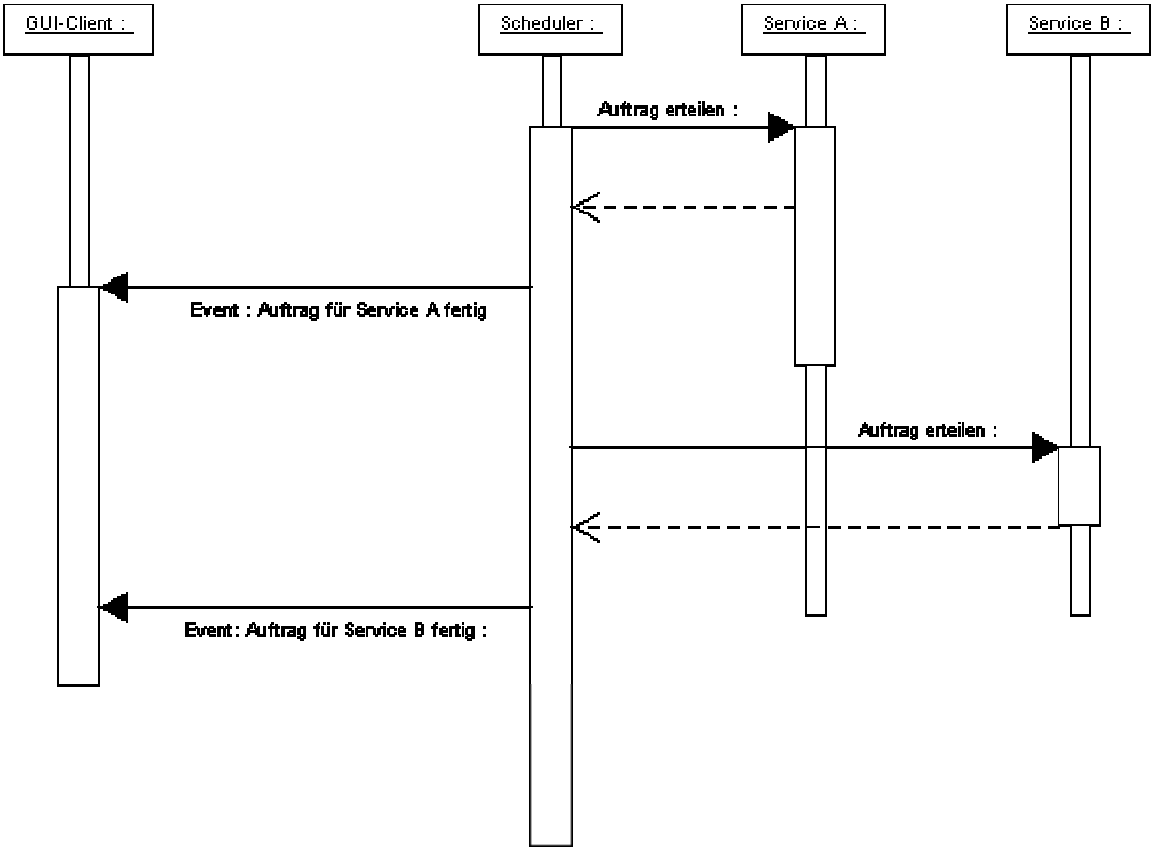

Abb. 3.2: Auftrag mit zwei Service-Aufrufen und Benachrichtigung des GUI-Clients

3.5 Kritische Betrachtung des Corba Schedulers

Die in den vorangegangenen Kapiteln beschriebene Version des Scheduler weist einige Schwächen auf. So führt die Verwendung des einfachen, synchronen Methodenaufrufs zwecks Beauftragung der Services zwar zu einer einfachen Implementierung des Schedulers und ermöglicht ein einfaches Hinzufügen von weiteren Services, bewirkt aber für den Benutzer erhebliche Nachteile.

So ist das Eintragen des Parameter-Strings an der GUI-Komponente in ein einziges Textfeld ist sehr unkomfortabel und fehleranfällig, die Korrektheit der Parameter wird erst zum Ausführungszeitpunkt festgestellt. Es wäre wünschenswert, dass entweder komfortablere Eingabemöglichkeiten geschaffen werden oder eine Methode von den Services angeboten wird, die die Parameter validiert. Mit einer solchen Methode könnte der Scheduler die Parameter-Korrektheit schon zum Zeitpunkt der Eingabe überprüfen. Des Weiteren muss der Name, unter dem sich die Services beim Corba Naming Service registrieren, dem Benutzer bekannt sein und von ihm an der Oberfläche eingetragen werden. Hier wäre natürlich eine Auswahlliste mit den aufrufbaren Services sinnvoll.

Unkomfortabel ist auch die gewählte Lösung ein Auftrags-Ergebnis von einem zum nächsten Service über ein gemeinsames Netzwerk-Laufwerk zu übergeben. Hier wäre eine Ergebnisübergabe über die Schnittstellen der Services eleganter.

Gut an dieser Corba-Variante des Schedulers ist die Benachrichtigung des GUI-Client bei Zustandsänderungen über Callbacks. So kann der GUI-Client diese sofort am GUI anzeigen und muss dafür nicht ständig beim Scheduler nachfragen, ob sich eine Zustandänderung ergeben hat. Natürlich gibt es bei dieser Art der Kommunikation Schwierigkeiten, falls Firewalls mit Port- bzw. Protokollfilterung vorhanden sind.

4 Der Web Service Scheduler

Um zu erforschen, inwieweit sich die Web Service Technologie in Verbindung mit dem flexiblen Einsatz von XML für die Entwicklung verteilter Anwendungen eignet, sollte die bestehende Corba-Variante des Schedulers auf diese Technologie migriert werden. Die Umsetzung dieses Web Service Schedulers wird in diesem Kapitel beschrieben. Entwickelt wurde der Scheduler bzw. die GUI-Komponente in Java (zur Verwendeten Software siehe Kapitel 4.5)

4.1 Zusätzliche Anforderungen

Da die Corba Variante des Schedulers einige Schwächen aufweist (siehe Kapitel 3.5), sollte im Rahmen dieser Studienarbeit versucht werden, diese Schwächen für den Web Service Scheduler zu beseitigen. Dabei gelten aber weiterhin die Anforderungen, wie sie in Kapitel 3.1 für die Corba Variante aufgestellt wurden. Als zusätzliche Anforderungen sind aufzuzählen:

- Der Benutzer soll bei der Eingabe der Service-Parameter entlastet werden, d.h. für jeden einzelnen Parameter soll ein sinnvolles Eingabefeld an der GUI-Komponente erscheinen. Der Umstand, dass dem Scheduler relativ einfach weitere Services zur Beauftragung hinzugefügt werden können, soll dabei weiterhin bestehen bleiben.
- Hat ein Auftrag mehrere Service-Aufrufe (Service-Sequenz), so soll es möglich sein, beliebige Dokumente von einem Service an den folgenden weiterzureichen.
- Die Gui-Komponente soll internetweit benutzbar sein, d.h. die Kommunikation zwischen Scheduler Prozess und dessen GUI-Clients sollte wenig Probleme mit Firewalls haben. Als Standardeinstellung für Firewalls kann angenommen werden, dass das Http-Protokoll über Port 80 durchgelassen wird und die Scheduler-Kommunikation hierüber abgewickelt werden sollte.
- Weiterhin soll dem Benutzer die aufrufbaren Services in einer Auswahlliste zur Verfügung gestellt werden.

4.2 Der gewählte Lösungsansatz

Da bei der Web Service Technologie mit WSDL eine Sprache zur Beschreibung von Schnittstellen zur Verfügung steht, sollte diese für die Services verwendet werden. Dies hat den positiven Nebeneffekt, dass das Implementieren eines für den Scheduler ansprechbaren Service relativ einfach ist. Der Service kann in einer dem Entwickler bekannten Programmiersprache entwickelt werden, woraus ein WSDL-Compiler automatisch die WSDL Schnittstellen-Beschreibung generiert. Die WSDL-Beschreibung des Service muss dann noch dem Scheduler zugänglich gemacht werden. Die meisten Web Service Werkzeuge bieten die Möglichkeit an, den Service inklusive Anhängen des Strings „?wsdl“ an dessen URL aufzurufen und so die WSDL geliefert zu bekommen. So kann der Scheduler automatisch zur Laufzeit die Schnittstellen Beschreibungen der Services ermitteln, sobald ihm die URL bekannt ist. Für die erste Version soll der Einfachheit halber die Verwaltung der dem Scheduler bekannten Services zunächst nicht durch den Benutzer an dem GUI erfolgen, sondern durch ein einfaches Eintragen in eine dafür vorgesehene Datei.

Für die einzelnen Parameter eines Service-Aufrufs sollen entsprechende GUI-Elemente zur Eingabe angezeigt werden. Da der Scheduler nicht auf bestimmte Services zugeschnitten sein soll, wird die Auswahl und Anordnung dieser GUI-Elemente automatisch aus der Schnittstellenbeschreibung generiert. Es wird also eine Abbildung der Parameter aus der WSDL auf entsprechende GUI-Elemente benötigt. Da WSDL auf XML basiert, kann bei Parametern komplexeren Typs eine baumartige Anzeige erfolgen, entsprechend der Verschachtelung in der Definition. Da der Schwerpunkt in dieser Studienarbeit auf

Middleware-Techniken gelegt wird, soll diese flexible Möglichkeit eine Benutzungsoberfläche XML-gesteuert aufzubauen nur erwähnt werden und nicht im Detail beschrieben werden.

Nachdem die Parameter vom Benutzer am GUI eingegeben wurden, müssen diese Eingaben dem Scheduler wiederum bekannt gemacht werden, d.h. die Eingaben müssen in einem geeigneten, zu der WSDL-Beschreibung „passenden“ Format von der GUI-Komponente zum Scheduler übertragen werden. Dieser führt dann die Beauftragung der Services über SOAP anhand dieser Informationen durch. Dieses zu der WSDL-Beschreibung „passende“ Format wird aus der WSDL-Beschreibung generiert und anhand der Benutzereingaben sinnvoll gefüllt zum Scheduler geschickt (siehe Kapitel 4.3.1).

4.3 Die Kommunikation zwischen Scheduler und der Gui-Komponente

Die Schnittstelle zwischen Scheduler und der GUI-Komponente soll wie bei der Corba-Variante die Funktionalitäten zum Erstellen, Ändern, Löschen von Aufträgen sowie zum Beobachten der Auftragszustände nebst Fehlerbehandlung bieten.

Im vorigen Abschnitt 4.2 wurde schon angedeutet, dass komplexe Objekte zwischen Scheduler und seinen GUI-Clients ausgetauscht werden sollen. So benötigen die Gui-Clients die Definitionen zu den einzelnen Operationen der Services, der Scheduler wiederum die durch Benutzereingaben gefüllten Objekte zum Aufrufen der Services. Zusätzlich sollen die Clients über Zustandsänderungen bzw. Fehler benachrichtigt werden. Es wurde entschieden, die Kommunikation nicht mittels SOAP durchzuführen, sondern durch ein einfaches XML-Messaging, d.h. es werden XML-Dokumente per Http verschickt ohne in ein SOAP-Envelope eingebettet zu werden. Der Hauptgrund dafür ist Folgender:

Um ein Beobachten von Zustandsänderungen der Aufträge durch den GUI-Client zu ermöglichen, muss dieser ein Polling durchführen und ständig beim Server nach Zustandsänderungen fragen. Ein Callback-Mechanismus, wie sie die Corba-Variante durchführt, ist nicht möglich, da Http als Übertragungsprotokoll verwendet wird und die Clients nicht als Http-Server fungieren sollen. Dazu soll ein kleines Framework geschrieben werden, dass dieses Polling kapselt. Dieses Framework (Kapitel 4.3.2) soll nicht auf den Scheduler zugeschnitten sein, sondern nach beliebigen Ereignissen in Form von XML-Dokumenten pollen können. Dies gewährleistet zum einen eine gewisse Wiederverwendbarkeit und führt zum anderen dazu, dass das Pollen wesentlich performanter implementiert werden kann (existieren z.B. viele Ereignisse, nach denen gepollt werden soll, so kann das Framework intern nur eine einzige Anfrage übers Netz schicken, anstatt für jedes Ereignis eine). Durch die Unterstützung von beliebigen Ereignissen kann für dieses Poll-Framework kein Mapping von XML-Konstrukten auf Methoden bzw. Parametern in eine Programmiersprache erfolgen, wie es bei der Verwendung von Soap üblich ist (im „rpc“ oder „document-style“). Natürlich könnten die XML-Dokumente im „document-style“ ohne ein solches Mapping mit Soap auf Server-Seite übertragen werden, so dass dort das gesamte XML-Dokument weiterverarbeitet werden kann. Für ein XML-Messaging über Soap entsteht aber nahezu kein Vorteil, außer dass gegenüber einem Messaging ohne Soap ein Exception-Handling hinzukommt. Die gewählte Lösung ohne Soap ist in Verbindung mit XML-Databinding performanter, da die zu verschickenden XML-Dokumente nicht erst in einen Soap-Envelope verpackt und server-seitig wieder entpackt werden müssen.

Da nun das Nachfragen nach Ereignissen über das Poll-Framework per XML-Messaging inklusive XML-Databinding erfolgt, wurden die Funktionalitäten zum Erstellen, Ändern, Löschen von Aufträgen ebenfalls mit dieser Technik umgesetzt. Dies wird im Folgenden detailliert beschrieben.

4.3.1 Erstellen, Ändern und Löschen von Aufträgen

Würde man die Scheduler Schnittstelle zum Erstellen, Ändern und Löschen von Aufträgen mit Soap implementieren, so würde man dafür explizit Methoden mit entsprechenden Parametern definieren. Statt dessen werden XML-Dokumente definiert, die dieselben benötigten Informationen beinhalten und von den GUI-Clients per Http an den Scheduler verschickt werden. Der Scheduler nimmt diese Dokumente entgegen, reagiert entsprechend und antwortet gegebenenfalls mit einem Antwortdokument (in der Http Response). Da der Scheduler in Java entwickelt wird, ist die Http Kommunikation mit der Servlet-Technologie sehr leicht zu implementieren (vgl. [Hunter 2001] oder [Eberhart, Fischer 2001]).

Für das Erstellen eines neuen Auftrages und der Sicherstellung dessen Identität muss sich ein GUI-Client erst eine vom Scheduler neu generierte Auftrags-ID abholen. Nach diesem Vorgang kann an dem GUI des Clients ein neuer Auftrag zusammengestellt werden und ein entsprechend gefülltes XML-Dokument zum Scheduler geschickt werden. Folgendes Beispieldokument soll dies verdeutlichen:

XML-Dokument für das Erstellen eines neuen Auftrags

```
<?xml version="1.0" encoding="UTF-8"?>

<task xmlns="http://hastedt/SchedulerSchemas">
        <id>5</id>      // ID des Auftrags
        <start>712345667</start> //auszuführende Startzeit des Auftrags in Sek. seit 1970
        <description>Test Task</description>
        <servicecall>
                <serviceDefinitionID>7</serviceDefinitionID>
                <state>READY</state>
                <forwardAttachmentToNextService>false</forwardAttachmentToNextService>
                <parameter>
                        <parameterName>Art der Statistik</parameterName>
                        <value>Excel</value>
                </parameter>
                <parameter>
                        <parameterName>Wochentag</parameterName>
                        <value>Montag</value>
                </parameter>
        </servicecall>
</task>
```

Wird dieses XML-Dokument von der GUI-Komponente zum Scheduler geschickt, so legt dieser einen neuen Auftrag mit der angegebenen ID an. Diese ID kann dann vom Scheduler benutzt werden, um die GUI-Komponente über Zustandsänderungen zu diesem Auftrag zu benachrichtigen. So benachrichtigt der Scheduler die GUI-Komponente beispielsweise bei einer erfolgreichen Durchführung eines Auftrags, indem er dem entsprechenden Ereignis-objekt die ID mitgibt. Zusätzlich sind die Angaben zu den Parametern des Auftrags zu erkennen. Die Angaben in den „value-tags“ sind die vom Benutzer an der Oberfläche eingegeben Parameter, die dem Service bei Beauftragung übergeben werden sollen. In dem obigen XML-Dokument wird beispielsweise ein Auftrag erstellt, der einen einzigen Service-Aufruf beinhaltet (da nur ein „servicecall“-Element vorhanden ist, bei Service-Sequenzen sind mehrere dieser Elemente vorhanden). Der Service-Aufruf gehört zu einem Service der Statistiken erstellt und hat die beiden Parameter „Art der Statistik“ und „Wochentag“. Der Benutzer hat an dem GUI angegeben wann (über das „start“-Element) und mit welchen Argumenten der Service vom Scheduler aufgerufen werden soll (eine Excel-Statistik vom Montag). Die Angabe, welchen Service der Scheduler aufrufen soll, erfolgt über die „serviceDefinitionID“, welche eindeutig einem WSDL-Dokument zugeordnet ist (die Verwaltung dieser serviceDefinitionIDs erfolgt auf Seiten des Schedulers, d.h. die GUI-Komponente muss auch diese ID vorher erfragen).

Wie oben schon erwähnt, wird das Problem, wie der Client ein solches XML-Dokument aus einer Java-Datenstruktur generiert, mit XML-Databinding gelöst (siehe Kapitel 2.3.1). Dazu wurde eine Definition für alle zwischen GUI-Komponente und Scheduler zu verschickenden XML-Dokumente in XML-Schema verfasst. Dazu beispielhaft die zu dem obigen XML-Dokument gehörende XML-Schema Definition (die oben im XML-Dokument gefetteten Stellen korrespondieren dabei mit den in der XML-Schema-Defintion gefetteten):

XML-Schema Definition des auszutauschenden Dokuments zum Erstellen von Aufträgen

```
<xsd:element name="task">  //Defintion eines Auftrags
<xsd:complexType>
        <xsd:sequence>
                <xsd:element name="id" type="xsd:int"/>
                <xsd:element name="description" minOccurs="0" type="xsd:string"/>
                <xsd:element name="starttime" type="xsd:long"/>
                <xsd:element ref="myNS:servicecall" maxOccurs="unbounded" />
        </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="servicecall"> //Defintion eines Aufrufs an einen Service
<xsd:complexType>
        <xsd:sequence>
                <xsd:element name="serviceDefinitionID" type="xsd:int"/>
                <xsd:element name="state" type="myNS:servicecallState"/>
                <xsd:element name="forwardAttachmentToNextService" type="xsd:boolean"/>
                <xsd:element ref="myNS:parameter" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="parameter"> //Definition eines Parameters zu einem Service Aufruf
<xsd:complexType>
        <xsd:sequence>
                <xsd:element name="parameterName" type="xsd:string"/>
                <xsd:element name="value" type="xsd:string"/>
                <xsd:element ref="myNS:parameter" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Es ist zu erkennen, wie die Struktur des XML-Dokuments zum Erstellen eines neuen Auftrags inklusive der verwendeten Datentypen vordefiniert werden kann.

Aus dieser Beschreibung in XML-Schema kann ein XML-Databinding Werkzeug Java-Klassen generieren, die client- und server-seitig zum automatischen Serialisieren bzw. Deserialisieren der XML-Dokumente verwendet werden können (z.B. Castor, vgl. [McLaughlin 2001]). Damit muss zum Zusammenstellen des obigen Beispiel-Dokuments zum Erstellen eines neuen Auftrags nicht mit XML-Konstrukten hantiert werden, sondern ausschließlich mit den generierten Java-Objekten, die mit den entsprechenden Werten gefüllt werden. Sind die Java-Objekte mit Werten gefüllt, muss nur noch die „Marshalling-Methode“ aufgerufen werden, die hieraus das gewünschte XML-Dokument erzeugt. Entsprechend dem Erstellen funktioniert das Ändern und Löschen von Aufträgen, wobei beim Löschen nur die Auftrags-ID mitgegeben werden muss und nicht ein komplettes Auftragsobjekt.

Dieses Vorgehen ist ein mächtiges und flexibles Mittel zur Übertragung von komplexen Daten, da mit XML-Schema eine sehr umfangreiche Datenmodellierung möglich ist. Hinzu

kommt, dass die Implementierung in Java mit der Servlet-Technologie bzw. den Bibliotheken zur Programmierung eines Http-Clients sehr einfach ist und einem die Arbeit zum Serialisieren bzw. Deserialisieren der XML-Dokumente durch XML-Databinding vollständig abgenommen wird. Der Entwickler muss sich nur auf die XML-Schema-Definition der zu übertragenen Daten konzentrieren.

Zu erwähnen ist noch, dass natürlich Fehler in der Kommunikation zwischen GUI-Komponente und Scheduler verhindert bzw. abgefangen werden müssen. Für die erste Version ist dieses Problem so gelöst, dass der Scheduler die Korrektheit der mitgeschickten XML-Dokumente als Vorbedingung voraussetzt und nur bei internen Fehlern auf Scheduler-Seite einen entsprechenden Fehlertext als Http-Response zurückgibt. Die GUI-Komponente ist also dafür zuständig, dass in den zusammengestellten XML-Dokumenten keine fehlerhaften Angaben vorhanden sind. Dies wird dadurch erreicht, dass schon an der Benutzungsoberfläche fehlerhafte Angaben/Aktionen verhindert werden. So kann ein Löschen eines Auftrages, der gerade aktiv ist und ausgeführt wird, dadurch verhindert werden, dass der Löschen-Button ausgegraut ist. Fehlerhafte Eingaben in Textfeldern, bei denen z.B. nur Zahlen eingegeben werden dürfen, können entsprechend verhindert werden. Dies ist dadurch möglich, dass das GUI aus der WSDL-Beschreibung erzeugt wird, in denen die Angabe der Datentypen zu den einzelnen Parametern vorhanden ist.

4.3.2 Http-Polling

Damit die GUI-Clients Zustandsänderungen von Aufträgen mitbekommen und diese an der Benutzungsoberfläche anzeigen können, müssen sie ständig beim Scheduler nachfragen.

Dazu soll ein kleines Poll-Framework entstehen, welches das ständige Nachfragen kapselt und das Beobachter-Muster auf dieses Polling aufsetzt. Für den Framework-Benutzer auf Client-Seite soll es also so wirken, als würde er den Server (Scheduler) beobachten. Dazu sind entsprechend dem Beobachter-Muster Methoden zum Anmelden, Abmelden und Benachrichtigen vorgesehen. Auf Server-Seite kann diese Framework benutzt werden, indem ein Objekt als ein zu Beobachtendes angemeldet wird. Die Events werden dann in Form von XML-Dokumenten verschickt. Wenn also z.B. beim Scheduler ein Auftrag beendet worden ist, erzeugt er ein XML-Dokument, indem diese Zustandsänderung beschrieben ist (Auftrags-ID, Dauer des Auftrags, evtl. Fehlertext usw.). Dieses XML-Dokument feuert er als Event am Framework, worauf dieses bei der nächsten Anfrage der Client-Seite des Frameworks als Event vorliegt. Die Verantwortlichkeit des Serialisierens bzw. Deserialisierens der XML-Dokumente liegt außerhalb des Frameworks und muss von den Nutzern auf Client- und Server-Seite durchgeführt werden (siehe Kapitel 4.3.3). Die Unterscheidung von verschiedenen Eventtypen geschieht über einen einfachen Namen, der auf Server-Seite beim Feuern des Events angegeben wird. Anhand dieses Namens kann der Client die Events differenzieren. Das folgende Klassendiagramm in Abb. 4.1 soll die Kapselung des Pollens auf Client-Seite verdeutlichen

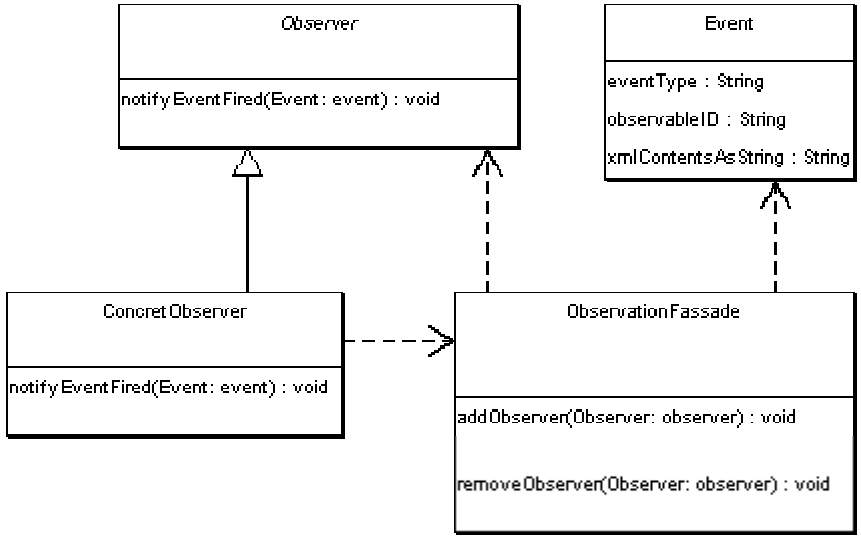

Abb. 4.1: Klassendiagramm Kapselung des Pollens auf Client-Seite

Die eigentliche Arbeit wird von der ObservationFassade erledigt. Diese Klasse kapselt das gesamte Polling inklusive Fehlerbehandlung. Möchte ein Client (in diesem Falle die GUI-Komponente) als Beobachter fungieren, muss er eine konkretes Exemplar der abstrakten Klasse Observer erzeugen (hier als ConcretObserver angedeutet) und dieses an der ObservationFassade anmelden. Die ObservationFassade fragt in einem Hintergrund-Thread ständig beim Scheduler nach, ob Ereignisse aufgetreten sind. Ist dies der Fall, so bekommt die ObservationFassade vom Scheduler das Ereignis als XML-Dokument mit Angabe des Ereignis-Typs und erzeugt daraus ein Event-Objekt. Daraufhin ruft die ObservationFassade alle registrierten konkreten Observer auf (notifyEvent-Methode) und übergibt ihnen das Event-Objekt. Der Name ObservationFassade wurde gewählt, da diese Klasse im Sinne des Fassaden-Musters (vgl. [Gamma et al. 1996]) die Komplexität eines Subsystems kapselt (die http-Kommunikation, das Pollen inklusive Fehlerbehandlung bei Netzwerkproblemen, Verwaltung verschiedener Server bzw. Eventtypen).

Zu erwähnen ist noch, dass das Polling jeweils nur einmal durchgeführt wird, auch wenn mehrere, verschiedenartige Events existieren. Diese werden dann gemeinsam als Http-Antwort geschickt und auf Client-Seite wieder als getrennte Events behandelt. So ist dieses ständige Pollen inklusive einer nicht zu hohen Pollrate in Bezug auf die Netzlast überhaupt zu vertreten.

4.3.3 Beobachten von Aufträgen

Die Benachrichtigung der GUI-Clients über Zustandsänderungen von Aufträgen erfolgt wie auch das Erstellen, Ändern und Löschen über das Verschicken von XML-Dokumenten inklusive XML-Databinding auf Client- und Server-Seite. Um den Beobachter-Mechanismus zu realisieren, wird dazu das in Kapitel 4.3.2 beschriebene Poll-Framework verwendet. Die Clients werden damit wie bei der Corba-Variante darüber informiert, ob ein Service-Aufruf gerade durch den Scheduler durchgeführt wird oder ein Aufruf gerade beendet worden ist. Folgendes Sequenzdiagramm in Abb 4.2 soll den dynamischen Ablauf dabei verdeutlichen.

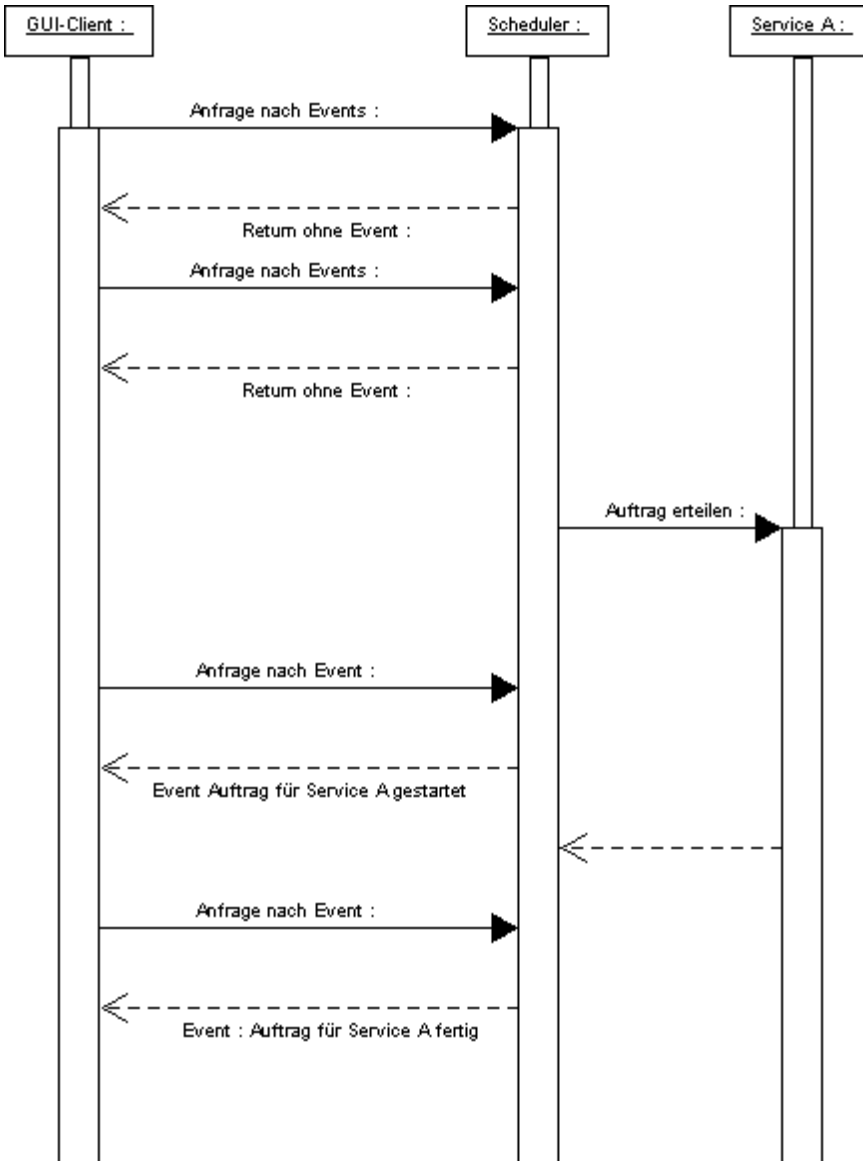

Abb. 4.2: Benachrichtigung des Clients über den Start bzw. die Beendigung eines Auftrags:

In diesem Beispiel fragt der GUI-Client zweimal vergeblich nach Ereignissen beim Scheduler an. Zum Zeitpunkt der dritten Anfrage wurde vorher vom Scheduler ein Auftrag an Service A erteilt, worüber der GUI-Client benachrichtigt wird. Eine entsprechende Benachrichtigung erfolgt dann auch bei der vierten Anfrage, zu deren Zeitpunkt der vorher an Service A erteilte Auftrag schon fertiggestellt wurde. Ereignen sich zwischen zwei Anfragen mehrere Ereignisse auf Scheduler-Seite, so werden diese bei der nächstfolgenden Anfrage des GUI-Clients zusammen als Antwort geschickt. Das Polling-Framework gewährleistet, dass die Ereignisse auf Client-Seite wieder in der korrekten zeitlichen Reihenfolge auftreten.

Beispielhaft für alle denkbaren Ereignisse soll die XML-Schema Definition für ein Dokument angegeben werden, welches dem GUI-Client eine Beendigung eines Auftrags signalisiert:

XML-Schema Definition eines Ereignis-Dokuments für die Beendigung eines Service-Aufrufs

```
<xsd:element name="serviceExecutionResult">  //Definition eines Ergebnis-Ereignisses
<xsd:complexType>
        <xsd:sequence>
                <xsd:element name="taskID" type="xsd:int"/>   //Auftrags-ID
                <xsd:element name="executionStartTime" type="xsd:long"/>
                <xsd:element name="executionStopTime" type="xsd:long"/>
                <xsd:element name="executionResultValue" type="myNS:executionResultValue"/>
                <xsd:element name="errorText" minOccurs="0" type="xsd:string"/>
        </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="executionResultValue">  //mögliche Ergebniss-Typen definieren
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="OK"/>
                <xsd:enumeration value="ERROR_SYSTEM"/>
                <xsd:enumeration value="ERROR_PARAMETER"/>
        </xsd:restriction>
</xsd:simpleType>
```

In diesem Beispiel findet sich die Auftrags-ID wieder, anhand der der GUI-Client den entsprechenden Auftrag identifizieren kann und das Ereignis „Auftrag beendet“ an der Oberfläche entsprechend darstellen kann. Diese Definition des Ereignisses für die Beendigung eines Service-Aufrufs beinhaltet eine Fehlerbehandlung, wodurch die GUI-Clients über fehlerhaft eingegebene Parameter (executionResultValue ERROR_PARAMETER) und intern aufgetretene Fehler wie Netzwerkprobleme (ERROR_SYSTEM) inklusive optionalen Fehlertext benachrichtigt werden.

Die aus der Schema-Definition resultierenden XML-Dokumente bzw. die Benutzung des XML Data-Binding von Java aus soll hier nicht weiter beschrieben werden, da dies entsprechend dem in Kapitel 4.3.1 beschriebenen Vorgehen ist.

4.4 Die Soap-Kommunikation zwischen Scheduler und den Services

Die Kommunikation zwischen Scheduler und den Services soll mit SOAP über Http erfolgen, da dies als Standard für die Implementierung von Web Services gilt und somit eine einfache Entwicklung von Services bzw. deren Integration in den Scheduler-Ablauf zu erwarten ist. Im Idealfall käme der Entwickler eines Services gar nicht mit Soap und WSDL in Berührung.

4.4.1 Kontrollfluss

Anhand der WSDL-Definition der Services und den von dem GUI-Client gelieferten Informationen über die Parameter ist der Scheduler in der Lage, die Services aufzurufen. Haben die Services ihre Aufgabe durchgeführt, muss der Scheduler darüber informiert werden, damit er die GUI-Clients per Event benachrichtigen kann (siehe Kapitel 4.3.3) bzw. evtl. einen in der „Service-Sequenz“ folgenden Sevice-Aufruf durchführen kann. Der Scheduler kann die Fertigstellung eines Service-Auftrags auf zwei Arten ermitteln. Beim ersten Fall kommt der Kontrollfluss nach Aufruf der Service-Methode sofort zurück, ohne dass der Auftrag durchgeführt ist, der Service ist also ein asynchroner. Hat dieser asynchrone Service seine Aufgabe beendet, ruft er seinerseits beim Scheduler eine dafür vorgesehene Methode auf („taskFinished“) und benachrichtigt ihn über die Fertigstellung (der

Scheduler bietet diese ebenfalls als Soap Web Service an. Im zweiten Fall ist der aufgerufene Service ein synchroner Service, d.h. der Kontrollfluss der aufgerufenen Methode kommt erst wieder zum Scheduler zurück, wenn die Aufgabe durchgeführt und beendet ist. Diesen synchronen Soap-Aufruf kann der Scheduler intern für sich in einen asynchron wirkenden umgestalten, indem er eigens für diesen Auftrag einen Thread startet, welcher den synchronen Soap-Aufruf tätigt und nach Wiedererlangung des Kontrollflusses die eigentliche Scheduler-Logik darüber benachrichtigt. Beide Möglichkeiten sollen vom Scheduler unterstützt werden. Das folgendes Sequenzdiagramm in Abb 4.3 soll die asynchrone Kommunikation zwischen Scheduler und Services verdeutlichen.

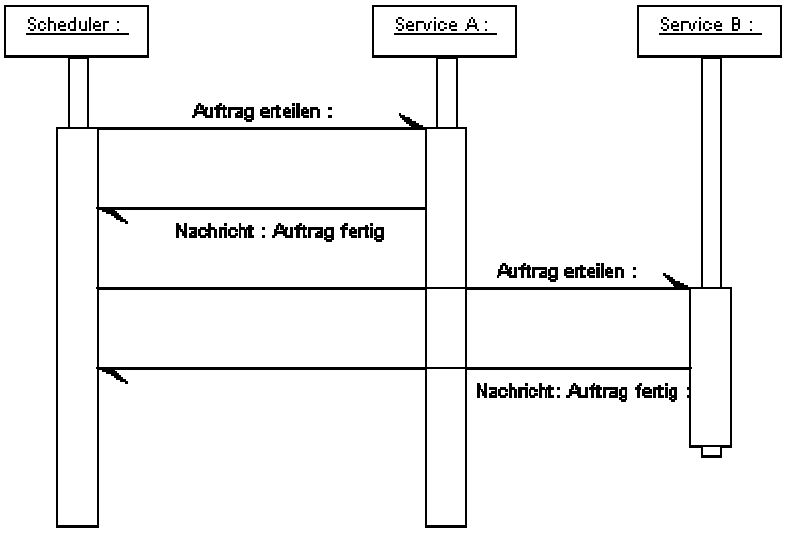

Abb. 4.3: Kontrollfluss mit zwei asynchron arbeitenden Services

Bei dieser asynchronen Variante der Soap-Kommunikation muss beim dem Rückruf der Services in Richtung Scheduler erkannt werden, welcher Service den nun mit seinem Auftrag fertig geworden ist. Dazu wird beim Beauftragen des Services eine Session-ID im Soap-Header mitgeschickt, die der Service dann der „taskFinished“-Methode übergibt. Um diese asynchrone Variante der Kommunikation inklusive der Session-ID zu nutzen, müssen entsprechende Services natürlich darauf abgestimmt sein.
Bei der synchronen Variante kann quasi jeder Web Service in den Scheduler eingebunden werden, es muss nur eine WSDL-Beschreibung vorliegen (wobei der Scheduler mit seinem WSDL-Parser im Rahmen dieser Studienarbeit noch nicht die komplette WSDL- bzw. XML-Schema Spezifikation unterstützt)

4.4.2 Zusammenstellen des Soap-Envelopes

Der Scheduler fungiert beim Aufruf der Services als Soap-Client. Für die Programmiersprache Java gibt es die von Sun spezifizierten APIs JAX-RPC, JAXM und SAAJ zur Implementierung eines Soap-Clients (vgl. [Chappell, Jewell 2002] und [Englander 2002]), die auch bei der hier verwendeten Soap-Middleware AXIS zur Verfügung stehen.
JAX-RPC ist darauf zugeschnitten, den Entwickler eines Soap-Clients von den Details des Protokolls bzw. der WSDL-Beschreibung fernzuhalten. Diese API ist quasi das Soap-Gegenstück zur Programmierung eines Corba-Clients. Es werden zum Aufruf eines Services Stellvertreter-Objekte (stubs) verwendet, die aus einer WSDL-Beschreibung generiert werden und sich um die Serialisierung des Aufrufs in einen SOAP-Envelope kümmern. Ein Aufruf über JAX-RPC kann auch ohne stub dynamisch zusammengestellt werden, hier müssen aber alle Parameter als Java-Klasse zur Verfügung stehen, d.h. eigene, komplexe Datenstrukturen können nur verwendet werden, wenn sie in kompilierter Form vorliegen. Da der Scheduler für die Service-Aufrufe ohne generierte Klassen auskommen soll, ist diese API

für diese Anforderung nicht geeignet. Statt dessen werden die APIs JAXM und SAAJ verwendet.

JAXM hat eine etwas geringere Abstraktion von den SOAP-Details, hier werden SOAP-Header und SOAP-Body direkt zusammengestellt. Die Informationen über die Struktur des SOAP-Bodys bekommt der Scheduler aus der WSDL-Definition und der von den GUI-Clients mit sinnvollen Werten gefüllten Parameter-Objekte. So ist z.B. anhand des in der WSDL angegebenen SOAP-Styles (RPC oder Document) zu entscheiden, ob der Name der aufgerufenen Service-Methode als Wrapper-Element im SAOP-Body erscheinen muss.

Da der Scheduler die Möglichkeit anbietet, beliebige Ergebnis-Dokumente (z.B. im Html-Format) von einen Service an den nächsten weiterzureichen, muss der SOAP-Aufruf für einen solchen Fall zusammen mit dem Ergebnis-Dokument im MIME-Format übertragen werden. Diese Möglichkeit wird von der SAAJ-API (SOAP with Attachments API for Java) unterstützt.

4.4.3 Nebenläufigkeit

Das Fungieren des Schedulers als Soap-Client für die Services hat zwei Aspekte in Bezug auf Nebenläufigkeit.
Zum Einen muss der Scheduler natürlich in der Lage sein mehrere Service-Aufrufe parallel auszuführen. Bei asynchronen Service-Aufrufen stellt dies kein Problem dar, da der Kontrollfluss unmittelbar zum Scheduler zurückkehrt. Bei synchronen Service-Aufrufen wird das Problem gelöst, indem für jeden Aufruf ein eigener Thread gestartet wird (wie schon in Kapitel 4.4.1 erwähnt). In diesem Thread wird der synchrone Service-Aufruf durchgeführt, so dass weitere Aufrufe parallel dazu erfolgen können.
Der zweite Aspekt in Bezug auf Nebenläufigkeit betrifft die Services. Ein Benutzer kann den Scheduler über die GUI-Komponente so eingestellt haben, dass er ein und denselben Service zur gleichen Zeit aufruft. Dieser Aspekt muss bei der Implementierung der Services beachtet werden. Ist ein Service in einer objektorientierten Programmiersprache implementiert, besteht die Möglichkeit, jeden Aufruf an ein anderes Exemplar der den Service implementierenden Klasse weiterzureichen. Dies kann explizit von der benutzten Web Service Middleware unterstützt werden. Ohne eine solche Unterstützung müssen geeignete Maßnahmen zur Thread-Synchronisation ergriffen werden.

4.5 Verwendete Software

Für die Entwicklung des Web Service Schedulers wurde folgende Software verwendet:

- Java zur Entwicklung von Scheduler und der GUI-Komponente
- Apache Axis als „Web Service Middleware“ für die Kommunikation über Soap (basiert auf der Servlet Technologie)
- Die Servlet Engine Jetty zur Integration von Axis und der Realisierung des „Poll-Frameworks“
- Jakarta http-Client zum Pollen von der GUI-Komponente aus
- Castor für das XML-Databinding
- JDOM Zum Parsen der WSDL-Beschreibungen durch den Scheduler
- Getestet wurde der Scheduler auf den Betriebssystemen Suse Linux 8.0 und Windows 98

5 Fazit und Vergleich der Lösungen

Nachdem der Scheduler mit den beiden Middleware Technologien Corba und Web Services umgesetzt wurde, soll in diesem Kapitel ein kurzer Vergleich beider Varianten erfolgen.

Der gravierendste Unterschied liegt in der Kommunikation zwischen Scheduler und seinen GUI-Clients bei der Benachrichtigung dieser über aufgetretene Ereignisse. Grundsätzlich ist natürlich die Nutzung von Callbacks dem Polling vorzuziehen, um das Netzwerk nicht zu belasten. Die Nutzung von Callbacks ist in Corba einfach, da dieser Mechanismus in Verbindung mit entfernten Objektreferenzen vollständig in die Middleware integriert ist. Im Rahmen der Web Service Technologien ist die Nutzung von Callbacks allein für die Schnittstellenbeschreibung mit WSDL spezifiziert (siehe Kapitel 2.3.3). Wie der Callback-Mechanismus und ob er überhaupt von der Web Service Middleware umgesetzt werden soll, ist nicht spezifiziert. Hierfür müsste der Entwickler selbst einen (Http-)Server auf Client-Seite integrieren und eine eigene Lösung entwickeln, was im Vergleich zu Corba einen erheblichen Mehraufwand darstellt. Da die GUI-Clients des Schedulers keinen eigenen Http-Server integrieren sollten, wurde die Lösung übers Polling gewählt. Für die Entscheidung, welche Middleware-Technologie zur Realisierung von Client-Benachrichtigungen gewählt wird, ist abzuwägen, ob auf Client-Seite eine Server-Komponente für das jeweilige Protokoll laufen darf (Corba hat für Callbacks intern natürlich einen IIOP-Server auf Client-Seite), ob diese vom Server erreichbar ist (Stichwort Firewalls) oder ob Polling mit geringer Frequenz eine Alternative darstellt. Die hier gewählt Polling-Lösung startet alle 5 Sekunden beim Server eine Anfrage nach Events, was für den Scheduler bzw. der Netzwerklast durchaus akzeptabel scheint. Vielleicht wird sich ja auch das Hauptargument gegen Polling, nämlich die hohe Belastung des Netzwerks, in Zukunft etwas abschwächen. Dies könnte der Fall sein, wenn eine anzunehmende Steigerung der Übertragungskapazitäten von Netzwerken eintritt, so dass der Performanzaspekt bei der Kommunikation mit geringen Datenmengen in den Hintergrund rückt.

Die größere Benutzerfreundlichkeit wurde beim Web Service Scheduler durch die Verwendung von XML erreicht, indem eine Abbildung der WSDL-Definition auf GUI-Elemente geschaffen wurde. Dies ermöglichte die Auswahl geeigneter GUI-Elemente sowie eine Validierung der Parameter zum Zeitpunkt der Eingabe (z.B. nur ganze Zahlen erlaubt beim Parameter des Typs Integer). Diese Flexibilität hätte aber durchaus auch mit Corba erreicht werden können, indem komplette XML-Dokumente als String-Parameter eines entfernten Methodenaufrufs mitgeschickt werden. Dazu wäre zur Implementierung der Services eine XML-Schema Definition der Parameter notwendig, die dem Scheduler zur Verfügung gestellt wird. Aus dieser Definition könnte ebenfalls eine Abbildung auf Gui-Elemente stattfinden. Die Services müssten das XML-Dokument bei einem eingehenden Auftrag dann selbst verarbeiten, was beispielsweise mit XML-Databinding zu realisieren wäre. Es ist also festzuhalten, dass die Verwendung von XML eine hohe Flexibilität und Änderbarkeit bezüglich der Implementierung bringen kann. Eine Kommunikation durch Verschicken von XML-Dokumenten (XML-Messaging) ohne anschließende Deserialisierung in Methoden-Aufrufe und Parameter der Programmiersprache ist daher eine gute Möglichkeit. Vor diesem Hintergrund erscheint die Verwendung von SOAP für diese Art der Kommunikation nicht unbedingt notwendig. Diese kann ebenso direkt mit dem darunterliegenden Übertragungsprotokoll (z.B. Http) realisiert werden. Durch das Einpacken der XML-Nachricht in einen SOAP-Envelope und das Auspacken auf Server-Seite durch die Middleware entsteht größerer Aufwand und bringt kaum einen Vorteil (außer evtl. ein integriertes Exception-Handling).

Allgemein ist zu sagen, dass die Implementierung eines RPC-Services (also eines Services bei dem die eingehenden Aufrufe auf Methoden bzw. Parameter in der Programmiersprache deserialisiert werden) mit SOAP im Idealfall einfacher als bei Corba ist, da die Schnittstellen-definition ausschließlich in der verwendeten Programmiersprache erfolgt. Dies ist aber aufgrund von Interoperabilitätsproblemen momentan nicht immer der Fall (siehe Kapitel

2.4.1). Die Verwendung der generierten Stubs zur Implementierung der Clients ist bei beiden ähnlich. Bei der Entscheidung, welche Middleware eingesetzt werden soll, spielen die oben angesprochene Callback-Problematik bzw. die Firewall-Problematik die größere Rolle.

Ein Plus des Web Service Schedulers ist die Möglichkeit, beliebige Dokumente als Anhang zu Verschicken, was mit SOAP relativ einfach zu realisieren ist. Dies wäre in Corba mit ein wenig Mehraufwand ebenfalls umzusetzen (siehe Kapitel 3.2).

Zur Performanz der beiden Scheduler-Varianten ist zu sagen, dass der Corba Scheduler hier natürlich im Vorteil ist, da dieser ohne die Verwendung von XML implementiert ist und das Serialisieren bzw. Deserialisieren von XML aufgrund des größeren Datenvolumens zeitaufwendig ist.

6 Ausblick und weitere Forschungsfragen

Im Gegensatz zu der etablierten Middleware Technologie Corba sind im Bereich der Web Services noch viele Veränderungen zu verzeichnen. Dies betrifft zum einen zusätzliche oder überarbeitete Spezifikationen, zum anderen die Umsetzung dieser Spezifikationen bzw. zusätzlicher Unterstützungen im Bereich der Web Service-Middleware. So gibt es beispielsweise Bemühungen, Standards in den Bereichen verteilter Transaktionen, Workflows oder Sicherheit zu schaffen. Es bleibt abzuwarten, inwieweit hier Standards etabliert werden können und ein Wildwuchs von Spezifikationen vermieden werden kann. Anzumerken ist, dass viele der im Web Service Bereich gerade entstehenden Spezifikationen bei Corba schon vorhanden sind.

Aufgrund der großen Unterstützung in der Industrie ist abzusehen, dass sich die Vision im Bereich der Web Servie Technologie verwirklichen wird, nämlich eine Integration von SW-Komponenten übers Internet (Stichwort B2B-Markt). Welche Standards sich in diesem Bereich etablieren, ob z.B. Soap-RPC eine große Rolle spielen oder eher ein XML-Messaging ganz ohne Soap die Regel sein wird, scheint offen zu sein.

Ansatz für weitere Forschungen im Bereich Web Services geben die oben genannten Bereiche Sicherheit, Transaktionen oder Workflows. In Bezug auf die Verwendung von XML ergeben sich Ansatzpunkte für weitere Forschungen im Bereich von GUIs sowie Persistenz. So könnte die in dieser Arbeit nur angedeutete Vorgehensweise, eine grafische Benutzungs-oberfläche aus XML zu generieren, ausgebaut werden. Im Bereich Persistenz wäre bei dem hier besprochenen Scheduler denkbar, dass dieser die Aufträge, die er in XML-Form von der GUI-Komponente bekommt, persistent in einer Datenbank ablegt (bei Absturz des Schedulers gingen die Daten dann nicht verloren).

Interessant wäre auch eine Forschung mit gesellschaftlichem Aspekt. Es könnten der Frage nachgegangen werden, wann und warum sich eine Technologie durchsetzt und ob technisch bessere Technologien dabei auf der Strecke bleiben. Dabei könnte untersucht werden, welche Rolle dabei Marketing der Herstellerfirmen, die Presse, Entscheider in Unternehmen, Ausbildungseinrichtungen wie UNIs, sowie Entwickler, die die Technologie benutzen, spielen.

Literaturverzeichnis

[Bleek 1997] Bleek, W.-G. „Techniken zur Konstruktion verteilter und technisch eingebetteter Anwendungssysteme“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik

[Brose, Vogel, Duddy 2001] Brose, G.; Vogel, A.; Duddy, K. “Java Programming with Corba”, 3. edition, Wiley, New York

[Cerami 2002] Cerami, E. “Web Services Essentials”, O’Reilly

[Chappell, Jewell 2002] Chappell, D. A.; Jewell, Tyler: “Java Web Services”, O’Reilly

[Eberhart, Fischer 2001] Eberhart, A.; Fischer S. „Java-Bausteine für E-Commerce-Anwendungen“. 2.Auflage: Hanser Verlag, München

[Englander 2002] Englander, R. „Java and Soap“, O’Reilly

[Gamma et al. 1996] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. „Entwurfsmuster: Elemente wiederverwendbarer objektoriertierter Software“ Addison-Wesley, Bonn

[Griffel 1998] Griffel, F. „Componentware – Konzepte und Techniken eines Softwareparadigmas“, dpunkt Verlag, Heidelberg

[Harold, Means 2002] Harold, E.R.; Means, W.S. “XML in a Nutshell” 2. edition, O’Reilly

[Hunter 2001] Hunter, J. „Java Servlet Programming“ 2. edition, O’Reilly

[McLaughlin 2001] McLaughlin, B. “Java & XML” 2. edition, O’Reilly

[Nilius 2002] Nilius, F. „Anbindung und Kapselung existierender Anwendungssysteme“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik

[Oestereich 1999] Oestereich, B. „Objektorientierte Softwareentwicklung – Analyse und Design mit Unified Modeling Language“ 4. Auflage, 1. korr. Nachdr., Oldenbourg, München

[Rechenberg & Pomberger 1999] Rechenberg, P., Pomberger, G. (Hrsg.) „Informatik-Handbuch“. 2., aktualisierte und erweiterte Auflage. München, Wien: Hanser Verlag

[van der Vlist 2002] van der Vlist, E. “XML Schema”, O’Reilly