Flexible Vermittlung von skalierbaren Dienstobjekten in verteilten Systemen*

Arnd Grosse, Stefan Dolk und Rainer Ruggaber

Universität Karlsruhe, Institut für Telematik,
D-76128 Karlsruhe, Germany,
{grosse, dolk, ruggaber}@telematik.informatik.uni-karlsruhe.de,
Tel.: +49(0)721-608 6407, Fax: +49(0)721-388097

Zusammenfassung Die zunehmende arbeitsteilige Integration zwischen weiträumig verteilten Anwendungen bei gleichzeitiger Partitionierung zuvor monolithischer Anwendungen in feingranularere (Dienst-)-Objekte erfordert eine systemtechnische Unterstützung, um zur Laufzeit flexibel das jeweils benötigte Dienstobjekt zu ermitteln und darauf zuzugreifen. Diese Flexibilität wird in verteilten Systemen durch eine Vermittlungsschicht auf Basis verteilter Trading-Dienste bereitgestellt. Der Nachteil dieses Ansatzes liegt in seiner Beschränkung auf die Vermittlung von Objektreferenzen. In dem folgenden Artikel wird ein in Java realisierter erweiterter Trading-Ansatz vorgestellt, der neben der Referenz zudem den Objektzustand und die Implementierung vermittelt und damit eine flexible Erstellung verteilter Anwendungen unterstützt.

1 Einleitung

Zukünftige Informationssysteme werden vermehrt modular gestaltet und setzen sich verteilt aus über Rechnergrenzen hinweg *kooperierenden Komponenten* zusammen. Dahinter steht der Anspruch, sowohl eine stärkere Verknüpfung ehemals monolithisch aufgebauter und isoliert ablaufender Anwendungen zu erzielen, als auch einzelne beteiligte Komponenten flexibel wiederverwenden zu können [1]. Die Komponenten unterliegen dabei kontinuierlichen Änderungen und Anpassungen, wodurch hohe Anforderungen an die jeweilige Konfiguration und an eine schnelle Aktualisierung gestellt werden. Dieses führt soweit, daß eine Aktualisierung möglich sein muß, ohne den gesamten Anwendungsverbund oder eine einzelne Teilapplikation herunterzufahren.

Zur Realisierung solcher Anwendungen bedarf es fortgeschrittener Verfahren, um die Interoperabilität der einzelnen Komponenten über Rechnergrenzen hinweg sicherzustellen. Mit Middleware-Diensten im allgemeinen und der

* Dieser Beitrag entstand im Rahmen des Sonderforschungsbereichs 346 „Rechnerintegrierte Konstruktion und Fertigung von Bauteilen" der Deutschen Forschungsgemeinschaft DFG

CORBA-Architektur im besonderen wird hier im Bereich verteilter Systeme eine Schlüsseltechnologie zur Realisierung verteilter Anwendungen geliefert, die eine Kooperation zwischen den Komponenten auch über große räumliche Distanzen sicherstellt und die sich damit z.B. für eine unternehmensübergreifende Verwendung anbietet [9].

Mit der Modularisierung von Anwendungsfunktionen und der damit verbundenen gestiegenen Anzahl an Komponenten ergibt sich jedoch eine in verteilten Systemen inhärente Komplexität bezüglich deren Konfiguration. Mit dem *Trading-Dienst* steht in der CORBA-Architektur ein sog. Common Service zur Verfügung [8], der die Vermittlung von Diensten zwischen verteilten Anwendungen und damit die Komplexität der Zuordnung zwischen Objekten übernimmt. Grundlage dieses Dienstes bildet dabei ein Standardisierungsvorschlag der ISO [5].

Die hier als Dienst betrachteten Objekte bilden dabei in Verbindung mit dem Trader einen Dienstemarkt. Auf diesem Markt machen Server-Anwendungen ihr Dienstangebot in Form von Objektreferenzen bekannt. Client-Anwendungen können diese Angebote zur Laufzeit unter Angabe gewünschter Diensteigenschaften vom Trader erfragen. Der Trader selbst wählt hierzu aus der Menge der bei ihm registrierten Dienstangebote — eventuell unter Einbeziehung mit ihm kooperierender Trader — geeignete Objektreferenzen aus und übergibt sie der Client-Anwendung, die daraufhin das Objekt mittels eines entfernten Aufruf verwenden kann [6].

Somit wird durch den Trader die Zuordnung zwischen Objekten im verteilten System übernommen. Jedoch unterliegt bei einem solchen System die reine Vermittlung von Objektreferenzen und die damit verbundene zumeist RPC-basierte Kommunikation einer ineffizienten und zugleich unflexiblen Beschränkung. Diese ergibt sich insbesondere bei Verwendung der Dienstfunktionen aufgrund der hohen Latenzzeiten zwischen Objektaufrufen weiträumig verteilter Anwendungen. Auf den folgenden Seiten soll nun eine Erweiterung des Trading-Dienstes in verteilten Systemen vorgestellt werden, die diese Problematik beseitigt, indem an Stelle von Objektreferenzen sog. *skalierbare Dienstobjekte* vermittelt werden. Das zugrunde liegende neue Konzept wird dabei im nächsten Kapitel vorgestellt. Im Anschluß daran wird beispielhaft die Realisierung unter Java skizziert. Den Abschluß bildet eine kurze Zusammenfassung.

2 Skalierbare Dienstobjekte

Das Konzept der *skalierbaren Dienstobjekte* beschreibt die Möglichkeit, neben der reinen Dienstreferenz weitere zu einem Dienstobjekt gehörende Bestandteile zu erfassen und beim Trading-Dienst zur Vermittlung zu registrieren. Eine Anwendung kann auf Basis dieser weiteren Bestandteile bei Bedarf das Dienstobjekt lokal bei sich instantiieren und verwenden. Dahinter steht der Gedanke, daß die reine Vermittlung von Objektreferenzen je nach Anwendungskontext sowohl sehr einschränkend als auch sehr ineffizient sein kann. Dieses gilt z.B. für schmalbandige oder unzuverlässige Kommunikationsverbindungen, wie sie im Bereich der

Mobilkommunikation oder des Teleservice anzutreffen sind. So kann insbesondere im Bereich weiträumig verteilter Anwendungen oder im Mobilrechnerumfeld eine effiziente Kommunikation zwischen den Objekten in Folge von Verbindungsabbrüchen, mangelnder Kommunikationskapazität oder Funkschatten nicht immer sichergestellt werden.

Ein weiterer Nachteil der ausschließlichen Vermittlung von Objektreferenzen liegt in den langen Laufzeiten, die mit entfernten Objektaufrufen verbunden sind und damit einer feingranulareren Realisierung verteilter Anwendungen entgegenstehen. So kann diesem Problem zwar durch die Installation lokaler Dienstobjekte begegnet werden, erfordert dann aber vor Anwendungsstart das Vorhandensein des Server-Codes. Auch ist damit ein zusätzlicher administrativer Aufwand für die Konfigurationssteuerung verbunden.

Die lokale Objektinstantiierung unter Verwendung des Traders ermöglicht in diesen Fällen ein (wenn auch mitunter eingeschränktes) Weiterarbeiten der Anwendung auf dann lokalen Instanzen des Dienstes. Es lassen sich zudem durch diese Erweiterung sehr einfach mobile Code-Paradigmen [2] wie Remote Evaluation oder Code on Demand flexibel unterstützen, indem Code-Fragmente mit Hilfe der Trading-Schicht zwischen den Anwendungen vermittelt werden, und von den Client-Anwendungen lokal bei sich zur Ausführung installiert werden.

Eine erweiterte Vermittlung besitzt weiterhin den Vorteil, daß trotz entfernter Objektanbietern eine hohe Code-Aktualität sichergestellt wird, indem vor der Verwendung eines bestimmten (per Trading vermittelbaren) Objekts innerhalb einer Anwendung dessen Code-Konsistenz geprüft werden und bei Bedarf der aktuelle Code vom Trader übertragen werden kann. Die Bereitstellung von Code durch den Trader anstelle der jeweiligen Installation in Form von z.B. Klassenbibliotheken ermöglicht es Anbietern von Objekten, den zugehörigen Code für einzelne Kunden entweder mit spezifischen Voreinstellungen zu versehen oder parallel unterschiedliche Implementierungen eines Objektes vorzuhalten. Hierdurch wird sowohl die Versionierung des Objektcodes als auch die Durchführung von Updates einfach und effizient möglich.

Des weiteren erlaubt der so erweiterte Trading-Ansatz, daß Änderungen einzelner Bestandteile unabhängig dem Trading-Dienst bekanntgemacht werden und sich so im Fall über große Distanzen kooperierender Trader effizientere Verteilungsprotokolle realisieren lassen, als dieses bei monolithisch aufgebauten Dienstobjekten der Fall ist. So ist es sinnvoll, einen sich selten ändernden Bestandteil eines entfernten Objekts bei einem lokalen Trader vorzuhalten, während ein sich häufig ändernder Bestandteil von dem entfernten Trader erst bei Anfrage durch einen Client erfragt wird. Eine detaillierte Betrachtung bezüglich des dabei resultierenden Kommunikationsaufwands und dessen Optimierung, insbesondere unter dem Aspekt des Weitverkehrs, findet sich in [3].

Die einzelnen Bestandteile eines Dienstobjekts werden im folgenden als *Part* bezeichnet, wobei ein Part eine Ausprägung zu einer *Facette* des zugehörigen Dienstobjekttyps darstellt. Eine Unterteilung des Dienstes in Parts und Facetten zeigt die Abbildung 1. In der OMG-Spezifikation des Trading-Dienstes [8] lassen sich die folgenden drei Facetten unterscheiden:

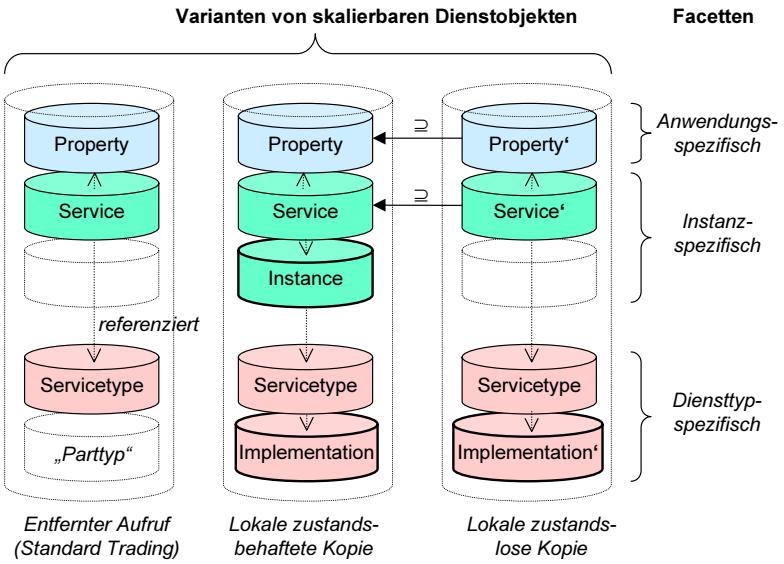

Abbildung1. Skalierbare Dienstobjekte bestehend aus Parts

1. *Diensttypspezifische Facette*: Jedem Dienstobjekt ist eindeutig ein Diensttyp zugeordnet, der über einen Diensttypbezeichner (Servicetype Name) referenziert wird. Der Bezeichner dient als Verweis auf die Schnittstellenbeschreibung des Diensttyps in einem zugrunde liegenden Diensttypverzeichnis (Servicetype Repository). Zudem werden damit die Bezeichner der Basisdiensttypklassen sowie die Bezeichner von Eigenschaftstypen des Diensttyps festlegt und müssen demgemäß bei einem Export eines Dienstobjektes existieren. Da Diensttypen im Trader mittels des Diensttyp-Verzeichnisses verwaltet werden, ist somit über den Diensttypbezeichner jedem Dienstobjekt dieser (diensttypspezifische) Part durch den Trading-Standard zugeordnet. Durch Ein- und Austragen dieses Parts im Diensttypverzeichnis kann dieser dynamisch durch Anwendungen geändert werden, solange von diesem Diensttyp keine weiteren Diensttypen abgeleitet wurden. Der Part, der den Diensttyp und damit die Schnittstelle beschreibt, wird im SOFT-Konzept als *Servicetype-Part* bezeichnet und ist somit eine Ausprägung der diensttypspezifischen Facette.
2. *Anwendungsspezifische Facette*: Jeder Dienstanbieter kann sein Dienstangebot mit beschreibenden Eigenschaften versehen. Welche dieser Eigenschaften von dem Anbieter unbedingt vorzusehen sind, welche eingeschränkt als nur lesbar angegeben oder gar optional sind, wird innerhalb des obigen Servicetype-Part vorgegeben. Der Part der anwendungsspezifischen Facette, der die beschreibenden Eigenschaften zu einem Dienstobjekt enthält, wird als *Property-Part* bezeichnet und kann durch die anbietende Anwendung beim Trader dynamisch geändert werden.

3. *Instanzspezifische Facette*: Eine *Service-Part* genannte Ausprägung dieser Facette enthält die instanzspezifischen Informationen zu einem Dienstobjekt. Hierbei handelt es sich beim Trading-Service [8] um die eindeutige (CORBA-)Objektreferenz, unter der das Dienstobjekt aufgerufen wird, sowie den Diensttypbezeichner, der als Referenz auf den zugehörigen Servicetype-Part dient. Eine Änderung dieses Parts während der Lebenszeit des Dienstobjekts ist hier nicht möglich, da bei erneuter Instantiierung eines Dienstobjektes auch eine neue Objektreferenz durch das Laufzeitsystem erzeugt und zudem vom Trader eine neuer Angebotsidentifikator vergeben wird.

Das Konzept der skalierbaren Dienstobjekte erweitert diese Ausprägungen nun um zwei zusätzliche Parts, die mit Hilfe eines geeignet erweiterten Traders vermittelt werden können. Die damit möglichen Verwendungsvarianten gegenüber dem konventionellen Trading sind zusätzlich in Abbildung 1 dargestellt:

1. Der erste Part wird hier *Implementation-Part* genannt und enthält eine Implementierung zu einem Diensttyp. Er erweitert die diensttypspezifische Facette. Eine Anwendung kann so eine Instanz des Dienstobjekts über eine erweiterte Trader-Anfrage bei sich anlegen, indem sie die Implementierung bei sich installiert und dann Operationen auf einer lokalen Instanz der Implementierung aufruft. Implementation-Parts werden über den Diensttypbezeichner referenziert und sind damit unabhängig von instantiierten Dienstobjekten. Verschiedene Dienstobjekte zu einem Diensttyp können durchaus unterschiedliche Implementierungen beim erweiterten Trader registrieren, indem unterschiedliche Implementation-Parts unter demselben Servicetype-Part registriert werden. Hierdurch ist, wie schon eingangs erwähnt wurde, eine effiziente Form der Versionierung möglich.
2. Im Gegensatz hierzu stellt der zweite Part, der sog. *Instance-Part*, eine Momentaufnahme, d.h. eine Zustandskopie, des beim Trader registrierten Dienstobjekts dar. Der Instance-Part erweitert somit die instanzspezifische Facette. Eine Anwendung kann hierdurch eine Zustandskopie des entfernten Dienstobjekts bei sich anlegen und auf dieser Kopie lokal arbeiten, anstelle entfernte Aufrufe darauf zu tätigen. Dies dient insbesondere zur Einsparung des Kommunikationsaufwands bei schlechten Kommunikationsverbindungen, zur Optimierung der Leistung des verteilten Systems oder zur Unterstützung für abgekoppelte Operationen. Eine zwingende Voraussetzung hierfür ist das Vorhandensein eines geeigneten Implementation-Parts auf Client-Seite.

Durch Erfassung des Implementation- sowie des Instance-Parts können somit Dienstobjekte einer bestehenden entfernten Anwendung lokal vermittelt werden. Hierbei existiert im Fall kontextbehafteter Dienstobjekte das Problem der Definition von geeigneten Synchronisationspunkten innerhalb des Ausführungspfads des Dienstobjektes zum Abspeichern seines Zustands oder dem späteren konsistenten Zusammenführen von Berechnungsergebnissen replizierter Objekte. Dieses Problematik wird in [7] eingehend beschrieben und liegt nicht im Aufgabenbereich der durch den Trader gebildeten Vermittlungsschicht, sondern muß durch die Implementierung des Server-Objekts unterstützt werden.

Während der Implementation-Part der diensttypspezifischen Facette zuzuordnen ist, überdeckt der Instance-Part sowohl die diensttypspezifische, anwendungsspezifische als auch instanzspezifische Facette. Die Besonderheit ist, daß sich die dem Instance-Part zugrunde liegende Implementierung von dem zuvor beschriebenen Implementation-Part unterscheiden kann, wenn z.B. nach der Instantiierung des Objekts die Implementierung geändert wird. Die nachfolgende Realisierung der skalierbaren Dienstobjekte sieht darüber hinaus die Möglichkeit vor, lediglich den Instance-Part zu exportieren. Hierbei muß jedoch auf Client-Seite bereits ein (zustands-)kompatibler Implementation-Part vorliegen.

3 Realisierung

Das zuvor beschriebene Konzept wurde unter Verwendung des Java-Development-Kit 1.1.6 und der Java-CORBA-Implementierung OrbixWeb 3.0 unter Solaris V2.6 realisiert. Als Basis diente ein OMG-konformer Trader, der am Institut für Telematik der Universität Karlsruhe in Java implementiert wurde und als Grundlage für den erweiterten *Part-Trader* dient. Die Motivation zur Verwendung von Java als Implementierungssprache lag in ihrer Portabilität auf beliebige Rechnerplattformen durch den inhärenten Byte-Code-Mechanismus. Der Code des erweiterten Traders konnte damit ohne Änderungen sowohl auf einem DEC-Alpha Rechner als auch einem Pentium II PC problemlos getestet werden, was den Anspruch der Plattformunabhängigkeit untermauert. Zweifelslos hätte das folgende Konzept auch auf Basis von z.B. C++ realisiert werden können, jedoch wäre dann eine problemlose Code-Vermittlung zwischen unterschiedlichen Rechnerplattformen nicht mehr möglich gewesen bzw. hätte durch umständliche Erweiterung der Attribute gesondert berücksichtigt werden müssen.

Die durch die erweiterte Vermittlung erzielte Flexibilität soll beispielhaft mittels eines skizzierten Dienstexports und Dienstimports unter Verwendung der erweiterten Parts verdeutlicht werden. Dem Ablauf liegt dabei die folgende verkürzte IDL-Datei des erweiterten Part-Traders zugrunde, die abwärtskompatibel zur bestehenden OMG-Spezifikation [8] ist:

```
// package sfb346.SOFT, softTrading.idl
module OMG { #include<CosTrading.idl>; }; // OMG-konformes Trading
// Erweitertes Trading-Modul
module SOFTtrading { ...
  struct Parts { OctetSeq implementation; OctetSeq state;};
  struct Offer { OMG::CosTrading::Offer offer; Parts parts; };
  typedef sequence<Offer> OfferSeq;
  ...
  interface SOFTRegister : OMG::CosTrading::Register { ...
    OMG::CosTrading::OfferId SOFTexport ( in Object reference,
            in OMG::CosTrading::ServiceTypeName type,
            in OMG::CosTrading::PropertySeq properties,
            in Parts parts // Erweiterung zur Aufnahme der Parts
            ) raises ( ... ); };
  ...
```

```
interface SOFTLookup : OMG::CosTrading::Lookup { ...
  void SOFTquery ( ..., out OfferSeq offers, ... ) raises ( ...,
        NoImplementationAvailable, NoSerializationAvailable);
  }; ...
```

Das obige Programmfragment skizziert die minimal notwendigen Ergänzungen im Modul `SOFTtrading` zur Vermittlung von Parts. Im wesentlichen wurde hierfür der Dienstangebotstyp `Offer` zur Aufnahme von Parts erweitert. Deutlich wird die Part-Vermittlung anhand der Schnittstellen der Kernfunktionen des Trading, dem Dienstimport (`SOFTquery`) und dem Dienstexport (`SOFTexport`). Die standardisierten Methoden des OMG-Trading sind dabei weiterhin mittels Vererbung verfügbar.

Die Erweiterung um die Part-Funktionalität wird im folgenden anhand eines Vermittlungsvorgangs auf Basis dieser Schnittstellen gezeigt. Es sei dabei angenommen, daß der Server sowohl den Implementation- als auch den Instance-Part seines Dienstangebots exportiert und dabei keinerlei Einschränkung hinsichtlich der Verwendung des Angebots durch Client-Anwendungen macht. Da es auch möglich ist, nur die Implementierung oder den Zustand zu exportieren, ohne aber einen entfernten Aufruf durch Clients zuzulassen, wurde hierfür eine eindeutige (Dummy-)Dienstreferenz definiert. Der schematische Ablauf einer erweiterten Vermittlung ist in Abbildung 2 dargestellt.

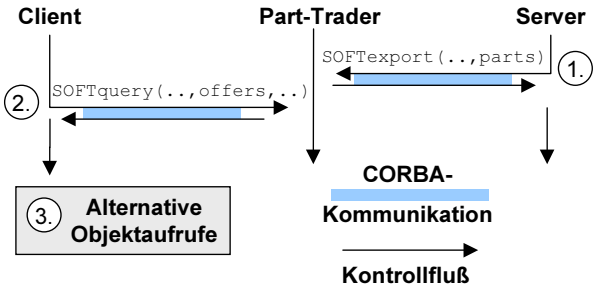

Abbildung2. Registrierung und Anfrage nach Parts beim erweiterten Trader

Im ersten Schritt (Abbildung 2, 1.) exportiert die Server-Anwendung mittels `SOFTexport` ihr Dienstangebot inklusive der Parts an den erweiterten Trader:

```
// Instantiiere ein Register-Proxyobjekt zur Kommunikation mit dem Trader
// OrbixWeb spezifisches bind anstelle string_to_object + ..narrow
sfb346.SOFT.softTrading.SOFTRegister regiRef =
  sfb346.SOFT.softTrading.SOFTRegisterHelper.bind(
    ":"+rep_trader,trader); ...
// Das parts-Object der Klasse Parts enthält den
// den Implementation- und den Instance-Part
id = regiRef.SOFTexport( servicereference,
  servicetype, serviceproperties, parts); ...
```

Das sfb346.SOFT-Java-Package zur Realisierung des erweiterten Traders sieht eine Reihe von Hilfsklassen vor, die für den Anwendungsprogrammierer des Dienstes die Bestimmung des Implementation-Part sowie des Instance-Part erleichtern. Sie sind jedoch aus Platzgründen hier nicht weiter detailliert. Hierzu zählt eine Hilfsklasse zur Bestimmung des Instance-Part, die sich des Object-Serialization-APIs von Java bedient [10]. D.h. sämtliche Server, die die Funktionalität des Traders für den Instance-Parts nutzen wollen, müssen hierzu die Serialization-Funktionalität implementieren, damit der Trader hierüber den aktuellen Zustand des zu exportierenden Objekts auslesen kann.

Zur Festlegung des Implementation-Parts kann auf Server-Seite entweder ein JAR-File (Java Archiv) der Implementierung angegeben werden, das sämtliche von dem Server benötigten `*.class`-Dateien zur Instantiierung enthält, oder die Implementierung des Dienstobjekts wird durch eine Hilfsklasse analysiert und das zu vermittelnde JAR-File hierüber automatisch generiert. Für eine genaue Analyse wird die Java-Reflection-Funktionalität verwendet [11], indem der Klassenname zu einem Dienstobjekt ermittelt und basierend auf der Laufzeitumgebung rekursiv nach zugehörigen `*.class`-Implementierungsdateien gesucht wird. Diese werden daraufhin in das generierte Archiv eingetragen.

Eine Client-Anwendung stellt nun in einem zweitem Schritt (Abbildung 2, 2.) eine Dienstanfrage an den erweiterten Trader. Sie hat z.B. das folgende Aussehen:

```
// Instantiiere ein Lookup-Proxyobjekt zur Kommunikation mit dem Trader
sfb346.SOFT.softTrading.SOFTLookup looRef =
        sfb346.SOFT.softTrading.SOFTLookupHelper.bind(
            ":"+this.rep_trader, this.trader );
// Erweiterte Suchanfrage
looRef.SOFTquery( // Standardparameter
        servicetype, serviceconstraints, servicepreference,
        servicepolicies, servicedesired_props, how_many,
        offerSeq, // Sequenz aus erweiterten Dienstangeboten
        off_itr, limits_aplied); ...
```

Der Trader sucht in seiner Datenbasis nach geeigneten Dienstangeboten auf diese Anfrage und meldet im Parameter `offerSeq` sämtliche (erweiterten) Dienstangebote bis zu der vom Client begrenzten maximalen Anzahl `how_many` zurück. Eine Beschränkung der vom Client gewünschten Parts wird durch Verwendung des `serviceconstraints`-Parameter gesteuert, indem beim Vermittlungsprozeß innerhalb des Part-Traders eine gültige Erweiterung der OMG-Constraint-Language verwendet wird ([8], Trading Object Service, Appendix B).

Gekennzeichnet wird dieses durch den «`SOFT 1.0`»-*Constraint-Identifikator* am Anfang eines `serviceconstraints`-Strings. Fehlt eine solche Angabe, so interpretiert der Part-Trader den Aufruf als eine OMG-konforme Trading-Anfrage und es werden keine erweiterten Parts bei der Vermittlung berücksichtigt, sondern nur Referenzen für einen entfernten Aufruf in der Rückgabesequenz `offerSeq` zurückgegeben. Eine Client-Anwendung kann dieses Verhalten durch die Angabe des Constraint-Identifikators und der zusätzlichen Verwendung der Schlüsselworte `SOFTimpl`, `SOFTstate` sowie `SOFTlocal` steuern. Der Part-Trader

berücksichtigt diese Schlüsselworte bei seiner Dienstauswahl. Ein Beispiel eines solchen Constraint-String ist z.B.:

```
"<<SOFT 1.0>> SOFTimpl and SOFTstate"
```

Hiermit werden vom Part-Trader nur Dienstangebote geliefert, die neben entfernten Referenzen zusätzlich einen Implementation-Part und einen Instance-Part anbieten. Das Schlüsselwort `SOFTlocal` schließt die Rückgabe entfernter Referenzen auf Dienste für die Vermittlung aus, so daß hierdurch nur lokale Instantiierungen möglich sind, während die alleinige Angabe des Constraint-Identifikators sämtliche Kombinationen als Rückgabewerte zuläßt.

Ein Client hat nun im 3. Schritt der Abbildung 2 die Auswahl, welche Variante des Objektaufrufs er auf einem importierten Dienstobjekt ausführen möchte. Prinzipiell hat er dazu die im vorherigen Kapitel genannten drei Möglichkeiten aus Abbildung 1. Die Instantiierung von Dienstobjekten für den Client übernimmt dabei eine weitere Java-Hilfsklasse namens `sfb346.SOFT.softTrading.util.ObjectCreation`, die hier etwas näher dargestellt werden soll:

```
 // Generierung von Dienstobjekten aus Trader-Angeboten
package sfb346.SOFT.softTrading.util; ...
class ObjectCreation { ...
  // Generierbare Objektfacetten
  public static final int Proxy = 0; // CORBA Proxy
  public static final int Local = 1; // Lokale Implementierung
  public static final int State = 2; // Lokale Zustandskopie
  ...
  public ObjectCreation () {};
  public ObjectCreation (
    int    which2initialize, // gewünschter Objekttyp
    String dir2save_impl,    // Abspeichern der Implementierung
    sfb346.SOFT.softTrading.Offer offer // ermitteltes Dienstangebot
    ) { ... }; ...
  // Allgemeine Methode zum Erzeugen eines lokalen oder eines Proxy-Objekts
  // Erzeuge Objekte, falls schon über Konstruktor initialisiert
  public java.lang.Object getObject() throws ... { ... };
  // Statische Methoden, um Initialisierung durchzuführen
  // Bei der 2. Methode ist Verzeichnis schon gespeichert oder es wird ein
  // hier nicht näher erläutertes Default-Verzeichnis verwendet
  public static java.lang.Object getObject( int which2initialize,
    String dir2save_impl, sfb346.SOFT.softTrading.Offer offer
    ) throws ... { ... }
  // Implementierungsverzeichnis schon gespeichert
  public static java.lang.Object getObject( int which2initialize,
     sfb346.SOFT.softTrading.Offer offer) throws ... { ... } ...
  // spezifische Methoden wie getProxyObject, getLocalObject & getStateObject
}; // end ObjectCreation
```

Beispiel:
`Calculator` definiert eine (IDL-)Schnittstelle zu einem Kalkulationsdienst. Es wird angenommen, daß die in Abbildung 2 durchgeführte `SOFTquery`-Anfrage erfolgreich war und im ersten Element des Rückgabearrays `offerSeq` ein Dienstangebot steht, welches sämtliche möglichen Parts enthält. Durch Benutzung der Hilfsklasse `SOFT.util.ObjectCreation` hat eine Client-Anwendung die genannten drei Möglichkeiten, ein Dienstobjekt bei sich zu instantiieren und auf diesem eine gewünschte Methode aufzurufen:

1. Der erste Fall zeigt exemplarisch den konventionellen Methodenaufruf über CORBA, indem die Methode `getProxyObject` der Hilfsklasse `SOFT.util.ObjectCreation` verwendet wird. Da die Methode ein Objekt vom Typ `org.omg.CORBA.Object` zurückliefert, ist im Gegensatz zu `getObject` kein expliziter Typecast auf ein CORBA-Objekt notwendig.

```
// Erzeuge Proxy-Dienstobjektinstanz, Typecast von Object auf Calculator
Calculator mycalc =
  CalculatorHelper.narrow(
    SOFT.util.ObjectCreation.getProxyObject( offerSeq.value[0] );
// Beispielaufruf zum entfernten Objekt
int result = mycalc.add(1,1);
```

   Hierbei ist transparent, daß durch die Hilfsklasse nur eine Objektreferenz zur Kommunikation über CORBA mit dem eigentlichen Dienstobjekt zurückgeliefert wird. Dieser Aufruf ist identisch mit einem Bindeaufruf auf der vom IDL-Compiler erzeugten Helper-Klasse zu Calculator, wie z.B.:
   `mycalc = CalculatorHelper.narrow( offerSeq.value[0].reference)`
   Der Aufruf ermittelt zuerst die OMG-Objektreferenz des Dienstangebots und generiert im Anschluß mittels der Helper-Klasse zum Calculator-Interface ein Proxy-Objekt zur Kalkulationsschnittstelle.

2. Die zweite Möglichkeit ergibt sich durch Verwendung des folgenden Aufrufs:

```
// Erzeuge Hilfsobjekt zur Objektgenerierung anstelle der statischen Methode
SOFT.util.ObjectCreation o = new
  SOFT.util.ObjectCreation( SOFT.util.ObjectCreation.Local,
                            ".", offerSeq.value[0]);
// Erzeuge zustandslose Kopie des Dienstes
Calculator mycalc = (Calculator) o.getObject();
int result = mycalc.add(1,1);
// Erzeuge eine zweite zustandslose Kopie, etc.
Calculator mycalc2 = (Calculator) o.getObject();
```

   Anstelle eines Proxy-Objekts wird durch die Hilfsklasse eine lokale Instanz des Dienstobjekts erzeugt und auf das gesuchte Interface per Typecast abgebildet. Die Implementierung des Dienstes wird in diesem Beispiel im lokalen Verzeichnis abgelegt. Hierbei ist zu beachten, daß die von `getObject`

erzeugte Instanz von der Klasse `org.omg.CORBA.Object` sein muß, da es ansonsten zu einem Typecast-Fehler während des Ablaufs kommt[1].

3. Die dritte Fall ergibt sich durch Verwendung des `SOFT.util.ObjectCreation.state`-Arguments. Ein Aufruf der Methode `getObject` erzeugt wie im 2. Fall eine lokale Instanz von `Calculator` und belegt zusätzlich unter Verwendung der Serialization-Funktionalität die erzeugte Instanz mit dem zuvor von der Server-Anwendung exportierten Dienstobjektzustand.

Diese Erweiterung der Trading-Funktion erlaubt somit eine leistungsfähigere Verwendung des Vermittlungsdienstes. Insbesondere für zeitweilig abgekoppelte Operationen, weit entfernte Aufrufe oder auch für kommunikationsintensive und damit zeitkritische Objektverwendungen bieten sich die hier gezeigten Alternativen an. Weiterhin lassen sich hierüber eine einfache Form der Objektmigration (Export der Dienstobjektimplementierung und des Zustands sowie anschließendes Beenden des Dienstes) und eine effiziente Form der Software-Verteilung über eine Vermittlungsschicht basierend auf kooperierenden Tradern realisieren [3]. Die Realisierung dieser Erweiterung verwendet dabei insbesondere spezifische Java-Funktionalitäten, die in dieser Einfachheit in anderen Programmiersprachen wie z.B. C++ nicht zur Verfügung stehen bzw. insbesondere im Hinblick auf Plattformunabhängigkeit nur unter großem Aufwand zu realisieren ist. Die weitere Verarbeitung der Anwendung ist von den möglichen alternativen Realisierungen der Dienstobjekte unbeeinflußt.

4 Verwandte Arbeiten

Die beiden neu eingeführten Parts ermöglichen somit eine lokale Instantiierung von Dienstobjekten bei einer Anwendung und unterstützen damit sowohl eine ungeplante Wiederverwendbarkeit als auch verschiedenste lokale und entfernte Verwendungsmuster für Server-Objekte.

Ein ähnlicher Ansatz hinsichtlich der Zerlegung von Objekten ist in [4] beschrieben, wobei hier jedoch eine wesentlich feingranularere Separation von Objekten bis auf Methodenebene stattfindet. Jedem Bestandteil eines Objekts wird hier ein *Chassis*- bzw. ein *Engine*-Objekt zugeordnet, das als eine Art "Wrapper" dient und für die Koordination der einzelnen Objektbestandteile sorgt. Dieser Ansatz erlaubt somit zur Laufzeit die Verteilung und die Rekonfiguration von Objekten auf unterschiedlichen Rechnern innerhalb einer CORBA-Domäne. Eine allzu feingranulare Verteilung der Objektbestandteile unterliegt dabei jedoch der genannten Laufzeitproblematik. Dahingegen sorgt das Konzept der skalierbaren Dienstobjekte für die Vermittlung von Implementierungen und Zuständen auch über CORBA-Domänen hinweg. Es ermöglicht damit die flexible Verwendung von Diensten in einem weit verteilten Kommunikationsumfeld, wenn hierzu Trader unterschiedlicher Domänen auf Part-Ebene miteinander kooperieren.

[1] Dieses Problem tritt bei Verwendung von OrbixWeb-spezifischen Server-Realisierungen auf, da hier die Server-Implementierung entgegen dem CORBA-Standard nicht von einem CORBA-Objekt erbt, sondern nur das Server-Interface implementiert.

5 Zusammenfassung

In der vorliegende Arbeit wurde auf Basis des von der OMG definierten Trading-Dienstes ein Konzept für skalierbare Dienstobjekte basierend auf Parts vorgestellt, das neben der reinen Vermittlung von CORBA-Objektreferenzen zusätzlich in der Lage ist, auch die Dienstobjektimplementierung (Implementation-Part) sowie den Dienstobjektzustand (Instance-Part) zwischen Dienstanbietern und Dienstbenutzern zu vermitteln. Wie gezeigt wurde, ist die dabei erzielte Lösung mit geringstem Aufwand in bestehende Anwendungen zu integrieren und flexibilisiert damit die Anwendungsmöglichkeiten von CORBA-basierten Anwendungen erheblich, was anhand einiger ausgewählter Java-Code-Fragmente zu einem auf Parts basierenden Vermittlungsvorgang motiviert wurde. Die entfernte oder lokale Verwendung eines Dienstobjektes ist dabei für den weiteren Anwendungsablauf transparent.

Die innerhalb der Realisierung generierten JAR-Files enthalten derzeit zum Teil redundante Klassen, falls diese Klassen schon bei der Client-Anwendung und ihrem zugehörigen Trader bekannt sind. Eine Lösung dieses Problems liegt in die Definition eines geeigneten Aushandlungsprotokolls, um die wirklich fehlenden oder modifizierten Implementierungen zu ermitteln. Dieses ist Gegenstand hierauf aufbauender Arbeiten.

Literatur

1. Bronsard, F., Bryan, D. , Kozacynski W.V., Liongosari, E.S, Ning, J.Q., Olafsson, A. ,Wetterstrand, J.W.: Toward Software Plug-and-Play. ACM SIGSOFT – Symposium on Software Reusability (SSR'97) (1997)
2. Carzaniga, A., Picco, G.P., Vigna, G.: Designing Distributed Applications with Mobile Code Paradigmas. Int. Conf. on Software Engineering (1997)
3. Grosse, A., Kottmann, D., Hartroth, J.: Disseminating Object-Oriented Applications in Large Scale Environments. Int. Conf. on Object Oriented Information Systems (1997)
4. Gründer, H., Geihs, K., Knape, T., Baumert, F.: Dynamische verteilte Objekte. Informatik Forschung und Entwicklung, Vol. 13 (1998) 26–37
5. International Organization for Standardization: Draft Rec. X.950-1|ISO/IEC DIS 13235-1 – ODP Trading Function. SC21 N10770 (1997)
6. Keller, L.: AGORA: Ein elektronischer Markt zur Vermittlung von Mehrwertdiensten. Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 20, Nr. 1 (1997) 3–10
7. Kottmann, D.: Replikation in vernetzten Systemen mit mobilen Teilnehmern. Dissertation, Fakultät für Informatik, Universität Karlsruhe (1996)
8. Object Management Group: CORBAservices: Common Object Services Specification. OMG-Document-No. formal/97-12-02.ps (1997)
9. Object Management Group: CORBA/IIOP 2.2 Specification, OMG-Document-No. formal/98-07-01.pdf (1998)
10. Sun Microsystems Inc.: Java Object Serialization Specification. ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.pdf (1997)
11. Sun Microsystems Inc.: Java Core Reflection – API and Specification. ftp://ftp.javasoft.com/docs/jdk1.1/java-reflection.pdf (1997)