DotQoS - Dienstgüte in .NET

Andreas Ulbrich, Torben Weis
TU Berlin, iVS, Einsteinufer 17, 10587 Berlin, Germany
{ulbi,weis}@ivs.tu-berlin.de

Abstract

Das Erbringen von Dienstgüte (Quality of Service) ist heute eine wichtige Forderung an verteilte Infrastrukturen. In den letzten Jahren wurden verschiedene Ansätze untersucht, Dienstgüteerbringung in Verteilungsplattformen – insbesondere in objekt-orientierte Middleware – zu integrieren. Dabei erwies sich die Integration als problematisch. Einerseits setzt Dienstgüteerbringung Wissen über die Verteilung und die Manipulation der Nachrichtenübertragung voraus, andererseits ist Verteilungstransparenz das zentrale Ziel beim Einsatz von Middleware. Außerdem greifen Mechanismen zur Dienstgüteerbringung an vielen verschiedenen Stellen einer verteilten Anwendung und sind deshalb nur schwer vom Anwendungscode selbst zu trennen.

In diesem Beitrag stellen wir unseren Ansatz zur Integration von Dienstgüte in .NET Remoting – der Verteilungsplattform von .NET – vor. Wir zeigen, wie verschiedene Eigenschaften und Fähigkeiten des .NET Rahmenwerkes, zur Dienstgüteintegration genutzt werden können und diese im Vergleich zu existierenden Lösungen stark vereinfachen. Beim Entwurf von DotQoS haben wir auf Erkenntnisse unserer Arbeit an MAQS, einem Dienstgüterahmenwerk für CORBA, aufgebaut.

1 Einleitung

Middleware-Plattformen werden eingesetzt, um die Entwicklung und Wartung verteilter Anwendungen zu vereinfachen. Durch Abstraktion, die denen nicht verteilter Systeme entsprechen, z.B. entfernter Methodenaufruf, kann mittels eines Middleware weitgehende Verteilungstransparenz erreicht werden [8].

Die spezifischen Eigenschaften verteilter Systeme wie teilweiser Ausfall oder dynamische Lastsituation auf Knoten bzw. Übertragungswegen verlangen nach speziellen Vorkehrungen beim Einsatz verteilter Anwendungen. Die Vorkehrungen können unter dem Begriff der Dienstgüte (QoS – Quality of Service) in Bezug auf die nichtfunktionalen Eigenschaften eines Systems zusammengefasst werden.

Aufbauend auf unserem CORBA-basierten [12] Dienstgüterahmenwerk MAQS (Management Architecture for Quality of Service) [3, 2, 1] haben wir untersucht, wie Dienstgüteerbringung in das .NET Rahmenwerk [6], insbesondere in dessen Verteilungsplattform .NET Remoting, integriert werden kann. Dabei betrachten wir sowohl die Spezifikation von Dienstgüte als auch die Integration von Mechanismen zur Dienstgüteerbringungen.

Der Rest dieses Beitrags ist wie folgt strukturiert. Im zweiten Teil dieser Einleitung diskutieren und motivieren wir einige Anforderungen an DotQoS. Im darauffolgenden Abschnitt wird die Spezifikation von Dienstgüte in DotQoS kurz dargestellt. Anschließend erläutern wir, wie Dienstgütemechanismen in das .NET Remoting-Rahmenwerk integriert werden können.

Ein Einführung in .NET und .NET Remoting würde den Rahmen dieses Beitrags sprengen. Wir verweisen daher auf die entsprechenden Standards [5, 6] und Literatur [13]. Wichtige Konzepte werden kurz vorgestellt, wenn sie benutzt werden. Im Übrigen gehen wir davon aus, dass der Leser mit der Terminologie im Dienstgüteumfeld weitgehend vertraut ist.

1.1 Anforderungen und Entwurfsziele

DotQoS soll einfach in .NET Remoting zu integrieren sein. Ein Rahmenwerk zur Dienstgüteerbringung ist idealerweise ein *add-on* zur existierenden Verteilungsplattform, ohne diese modifizieren zu müssen. Dadurch wird die Kompatibilität mit zukünftigen Version der Middleware vereinfacht und die Middleware kann von existierenden Andwendungen ohne Änderung weiter genutzt werden. Existierende Lösungen wie MAQS, TAO [15] oder QuO [11] nehmen umfassende Modifikation an der zugrundeliegenden Plattform vor.

Daraus folgt, dass Dienstgüteintegration ohne die Einführung von speziellen Aspektsprachen wie in QuO oder

*Die hier vorgestellte Arbeit wird unterstützt durch das europäische QCCS-Projekt www.qccs.org, IST-1999-20122, das DFG-Projekt GE 776/4 und das Discourse-Projekt www.discourse.de

der Erweiterung von Schnittstellenbeschreibungssprachen wie in MAQS [2] möglich sein muss. Das gilt insbesondere dann, wenn die zu Grunde liegende Plattform von Hause aus keine zusätzliche Sprache für verteilte Objekte benötigt. Dadurch wird sichergestellt, dass sich das Programmiermodell durch die Einführung von Dienstgüte nicht ändert und existierende Werkzeuge ohne Modifikation weitergenutzt werden können. Besonders in der Industrie ist das eine der wichtigsten Voraussetzung zur Akzeptanz von Dienstgüterahmenwerken.

Außerdem fordern wir die Unterstützung von Mehrkategoriedienstgüte, also zum Beispiel die Erbringung eines bestimmten Sicherheitsniveaus bei gleichzeitiger Sicherstellung von Ausfallsicherheit. Solche Szenarien treten in der Praxis recht häufig auf und werden bisher nur unzureichend unterstützt. Voraussetzung dafür ist eine reflexive Laufzeitumgebung, so dass Dienstgütemechanismen zur beliebig geladen und kombiniert werden können.

Reflexion ist auch Voraussetzung für Adaptivität [4]. DotQoS soll so entworfen werden, das Mechanismen zur Laufzeit an eine sich verändernte Umgebung angepasst werden können. Dazu ist es notwendig, das Dienstgütemechanismen, dazu gehören z.B. auch Nachrichtenformate und Transportprotokolle, eingekapselt werden können [10].

Wegen der besonderen Anforderungen, die Echtzeit-Dienstgüte an ein System stellt [14], betrachten wir diese nicht.

2 Dienstgütespezifikation in DotQoS

Für das Erbringen von Dienstgüte ist es notwendig, Dienstgüte spezifizieren zu können. Im Allgemeinen ist eine Dienstgütespezifikation die Definition eines Vetrages für eine bestimmte Dienstgütekategorie. Ein solche Vetragdefinition enthält Dimension, die die einzelnen Parameter eines Vetrages beschreiben.

Existierende Ansätze wie MAQS erweitern die CORBA IDL zur Vertragsdefinition oder führen dafür spezielle Sprachen wie QML [7] ein. Außerdem kann mit Hilfe solcher Sprachen deklariert werden, welche Dienstgüte eine bestimmte Schnittstelle unterstützt. Das führt dazu, dass zum alleinigen Zwecke der Dienstgütespezifikation IDL- bzw. Precompiler angepasst, bzw. ganz neu implementiert werden müssen.

Mit .NET, und dessen Unterstützung von Reflexion und erweiterbaren Metadaten, sind solche Werkzeuge überflüssig und die Spezifikation von Dienstgüte kann direkt in der Implementierungssprache der Anwendung erfolgen. Wegen der Mehrsprachigkeit von .NET, die auf einem gemeinsamen Typsystem (Common Type System) aufbaut, bleibt gleichzeitig die Sprachunabhängkeit erhalten.

2.1 Vetragsdefinition

Ein Vertrag wird definiert, durch ein Klasse, die von `DotQoS.Contracts.QoSCategorySchemeBase` abgeleitet ist. Diese Basisklasse für Veträge implementiert unter Anderem Vergleichsoperation von Vertragsinstanzen bzw. die Vertragsinstanziierung aus einer XML Konfigurationsdatei.

Eine Vertragsdimension ist ein *public property* eines einfachen Datentypes (`int`, `string`, `enum` usw.). Damit das property als Dimension erkennbar ist, wird es mit einem sogenannten *custom attribute* dekoriert. Dabei handelt es sich um erweiterbare Metadaten, die an Klassen-, Schnittstellen- und Methodendefinition angehängt und zur Laufzeit per Reflexion abgefragt werden können. Die verschiedenen Sprachen für .NET, z.B. C#, bzw. die .NET Basisbibliothek bieten viefältige Möglichkeiten mit erweiterbaren Metadaten zu arbeiten. Custom attributes können durch das Implementieren einer Klasse, die von `System.Attribute` ableitet, einfach selbst realisiert werden. Für Vetragsdimensionen haben wir das `QosDimension`-Attribut implementiert. Damit lassen sich sowohl die Maßeinheit der Dimension als auch die Richtung zunehmender Dienstgüte konfigurieren.

Folgendes Beispiel zeigt eine einfache Vetragsdefinition in C#, die nur eine Dimension enthält. Der Vertrag `Performance` enthält die Dimension `Throughput`, die mit Hilfe des `QoSDimension`-Attributes als solche markiert ist. Die Maßeinheit `QoSUnit` der Dimension ist s^{-1} (je Sekunde), wobei höhere Werte wegen `QoSDirection` einem besseren Dienstgüteniveau entsprechen.

```
public class Performance :
    QoSCategorySchemeBase
{
    [QoSDimension(
        QoSUnit.PerSec,
        QoSDirection.Ascending)]
    public int Througput {
        get { ... } set { ... }
    }
}
```

2.2 Deklaration von Dienstgüteunterstützung

Neben der Definition von Veträgen ist es auch möglich, zu deklarieren, welche Dienstgütekategorien eine Klasse oder Schnittstelle unterstützt. Zu diesem Zweck wird das `QoSContractClass`-Attribut genutzt. Auch hierbei handelt es sich um zuvor beschriebene erweiterbare Metadaten. Dieses Attribut kann jeder Klassen- bzw. Schnittstellendefinition zugewiesen werden. Die unterstützte Dienstgütekategorie wird dabei durch die entsprechende

Vetragsdefinition spezifiziert. Das folgende Beispiel zeigt eine Klasse `Query`, die `Performance` aus dem vorangegangen Beispiel unterstützt.

```
[QoSContractClass(typeof(Performance))]
public class Query :
    DotQoS.RemoteObject
{
    // query methods ...
}
```

Es sei angemerkt, dass es sich hierbei lediglich um eine Deklaration handelt, die Auskunft darüber gibt, welche Dienstgütekategorien eine Klasse unterstützt. Es wird überhaupt keine Aussage über den dazu verwendeten Dienstgütemechanismus gemacht. Es ist die Aufgabe des Dienstgüteentwicklers, das Rahmenwerk so zu konfigurieren, dass es zur Laufzeit die richtigen Mechanismen aktivieren kann. Durch diese Trennung ist es möglich Mechanismen beliebig auszutauschen, was letztendlich eine elementare Voraussetzung für Adaptivität darstellt.

Durch die Angabe mehrerer `QoSContractClass`-Attribute kann die Unterstützung von mehreren Dienstgütekategorien einfach deklariert werden. Auch hier wird keine Aussage über deren Umsetzung gemacht.

3 Dienstgüteerbringung in DotQoS

Zur Laufzeit werden Dienstgütemechanismen zur Dienstgüteerbringung eingesetzt. Diese Mechanismen müssen an den passenden Stellen in das .NET Remoting-Rahmenwerk integriert werden. Dazu ist es erforderlich, die grundlegende Arbeitsweise von .NET Remoting kurz zu verdeutlichen.

3.1 .NET Remoting

Remoting ist die Verteilungsplattform von .NET. Seine Funktionalität entspricht weitestgehend der von CORBA oder Java RMI. Remoting kommt – anders als CORBA – ohne Schnittstellenbeschreibungsprache aus, da clientseitige Proxies und server-seitige Dispatcher direkt per Reflexion erzeugt werden können.

Eine Besonderheit von Remoting ist dessen flexibler Nachrichtenpfad. Er besteht aus einer Kette von Sinks. Sinks sind in etwa mit den Interceptoren in CORBA vergleichbar. Eine Sink erhält eine Nachricht, analysiert und modifiziert die Nachricht und reicht sie schlussendlich an die nächste Sink in der Kette weiter.

Abbildung 1 zeigt den Nachrichtenpfad von .NET Remoting, der aus verschiedenen Sinks aufgebaut ist. Bei den Sinks kann zwischen Anwendungsschicht (über der gestrichelten Linie) und Transportschicht (unter der gestrichelten Linie) unterschieden werden. In DotQoS werden Sinks

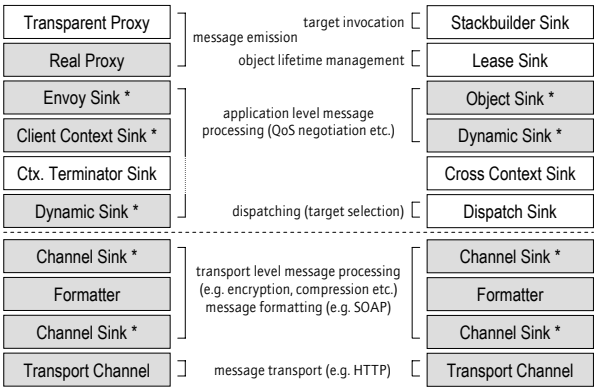

Abbildung 1. .NET Nachrichtenpfad (Client links, Server rechts), Elemente mit $*$ können mehrfach auftreten

genutzt, um Dienstgütemechanisms in die Verteilungsplattform zu integrieren. In Remoting ist dies durch Hinzufügen oder Austauschen von Sinks einfach möglich. Alle grau hervorgehobenen Elemente können ausgetauscht oder erweitert werden, um Remoting an die speziellen Bedürfnisse der Dienstgüteerbringung anzupassen.

3.2 Dienstgütemechanismen in DotQoS

Ein Dienstgütemechanismus in DotQoS kann als Sink implementiert werden. Dabei können auf der Anwendungsebene Mechanismen für Kategorien wie Authorisierung und Authentifikation oder Fehlertoleranz implementiert werden. Auf Transportebene ist es möglich Mechanismen zu realisieren, die mit der serialisierten Nachricht arbeiten, zum Beispiel Verschlüsselung oder Kompression. Da Sinks auf Anwendungsebene und Transportebene unabhängig voneinander arbeiten, kann ein hoher Grad an Flexibilität erreicht werden.

Anhand von Authorisierung und Authentifikation soll die Implementierung von Dienstgütemechanisme mit Hilfe von Sinks kurz dargestellt werden. Ziel ist, das bestimmte Methoden eines Serverobjektes nur nach von authorisierten Nutzern gerufen werden kann. Dies kann einfach mit je einer client- und einer server-seitigen Sink auf Anwendungsebene erreicht werden.

Die client-seitige Sink fügt Nutzerinformation, z.B. Nutzername und Passwort oder dessen Zertifikat, zur Nachricht hinzu. In .NET gibt es für solche Zwecke einen Logical Call Context, der einer Nachricht anhängt und mit dieser serialisiert wird. Die server-seitige Sink kann daher auf diesen Call Context zugreifen und die Nutzerinformation lesen. Mit diesen Informationen kann dann festgestellt werden, ob der Nutzer berechtigt ist, die in der Nachricht spezifizierte Methode aufzurufen. Ist dies nicht der Fall, erzeugt die Sink

die Antwortnachricht mit einer entsprechenden Fehlermeldung, die beim Client zu einer Exception führt. Anderenfalls reicht die Sink die Nachricht einfach zur Bearbeitung weiter.

Neben den o.g. Sinks gibt es auch noch die Möglichkeit, das Nachrichtenformat (z.B. SOAP) an eigene Bedürfnisse anzupassen. Das geschieht mittels eines Formatters, der letzten Endes wieder als Sink implementiert ist.

Ein weitere Möglichkeit Dienstgütemechanismen zu integrieren sind Transportkanäle. Diese implementieren in erster Linie das Nachrichtenprotokol (z.B. HTTP) können aber auch um zusätzliche Dienstgütemerkmale, wie zum Beispiel Bandbreitensteuerung erweitert werden.

Auerdem erlaubt .NET Remoting als eine der ersten Middlewareplattformen, den Standardproxy durch eine angepasste Implementierung zu ersetzen. Damit kann direkt in die Nachrichtenemmision eingegriffen werden. Eine Aufgabe des Proxys ist zum Beispiel die Auswahl des Übertragungskanals. Unter Anderem können Proxies deshalb zur Lastverteilung eingesetzt werden.

Mit Sinks auf Anwendungs- und Transportebene, flexiblen Transportkanälen und Proxies bietet .NET zahlreiche Möglichkeiten Dienstgütemechanismen zu implementieren, ohne dabei die Middleware selbst modifizieren zu müssen. Durch die flexiblen Kombinationsmöglichkeiten von Sinks, Kanälen und Proxies ist es relativ einfach Mehrkategoriedienstgüten umzusetzen. In [9] wurde gezeigt, dass Mechanismen auf Anwendungs- und Transportebene ausreichend sind, um ein Vielzahl von Dienstgütekategorien zu unterstützen.

Dienstgütemechanismen können als solche eingekapselt werden und flexibel in den Nachrichtenpfad integriert werden. Damit sind die notwendigen Voraussetzungen für eine adaptive Middleware erfüllt.

3.3 Integration von Dienstgütemechanismen

Objekte können in DotQoS eine Vielzahl an Kombination von Dienstgütekategorien unterstützen. Eine Integration aller dazu notwendiger Mechanismen in den Nachrichten Pfad bei der Erzeugung der Objekte bzw. Proxies ist deshalb nicht praktikabel. Auerdem ist es notwendig, Mechanismen erst zur Laufzeit entsprechend den Anforderungen an die Dienstgüte auszuwählen bzw. zur Adaption auszutauschen. In .NET wird eine Kette von Sinks aber schon bei der Initialisierung einer Client/Server-Interaktion erzeugt. Diese Kette bleibt während der gesamten Interaktion bestehen. Um dennoch die Auswahl bzw. Anpassung von Mechanismen zur Laufzeit zu ermöglichen, haben wir generische Sinks implementiert, die immer im Nachrichtenpfad vorhanden sind, und damit auch bei der Initialisierung einer Interaktion erzeugt werden können. Die Sinks, die Mechanismen realisieren, werden dann von diesen generischen Sinks entsprechend den Dienstgüteanforderung zur Laufzeit erzeugt und ausgewählt. Dadurch entstehen Sinks-Loops die dynamisch adaptiert werden können. In Abbildung 2 sind alle Elemente des DotQoS-Nachrichtenpfades erkennbar.

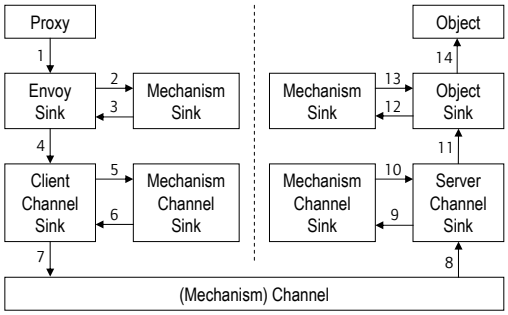

Abbildung 2. DotQoS Nachrichtenpfad mit Dienstgüte Mechanismen

Eine vom Proxy erzeugte Nachricht durchläuft zuerst die generische Sink (1) in der entsprechend der geforderten Dienstgüte die notwendigen Mechanismen ausgewählt werden. Die Nachricht wird durch diese Mechanismen geschickt (2, 3), bevor sie der Transportschicht übereignet wird (4). Hier werden analog zur Anwendungsschicht die notwendigen Mechanismen auf die Nachricht angewandt (5, 6). Danach wird die Nachricht vom Transportkanal an den Server übertragen (7, 8). Im Server kommen zuerst die Diesntgütemechanismen auf der Transportebene zur Anwendung (9, 10) danach (11) die der Anwendungsebene (12, 13) bevor die Nachricht ihr Zielobjekt erreicht. Die Antwortnachricht durchläuft diesen Nachrichtenpfad entsprechend rückwärts.

4 Zusammenfassung und Ausblick

Wir haben mit DotQoS ein Werkzeug geschaffen, das auf einfache Art und Weise die Erweiterung des .NET Remoting-Rahmenwerkes um Dienstgütemanagement ermöglicht. Dabei war es nicht notwendig, neue Sprachen und Übersetzer einzuführen. Anwendungsprogrammierer benutzen auch für Dienstgüte ihr bekanntnes Programmiermodel. DotQOS sollte zur besseren Akzeptanz von Dienstgüte-Rahmenwerken beitragen, da es in Form von dynamisch bindbaren Bibliotheken (DLL) mit jeder existierenden .NET Installation arbeitet. Ausserdem müssen Entwickler ihr Werkzeugketten nicht für die Berücksichtigung von Dienstgüte bei Entwurf und Implementierung anpassen. DotQoS bildet mit seiner Flexibilität die notwendige Voraussetzung [10] für die Umsetzung adaptiver Anwendungen.

Ohne die zahlreichen vorteilhaften Merkmale von .NET, wie erweiterbare Metadaten oder anpassbare Nach-

richtenpfade, wäre die Entwicklung eines Dienstgüte-Rahmenwerkes mit ähnlichen Fähigkeiten ungleich aufwendiger.

Gegenwärtig arbeiten wir an der Integration weiterer Dienstgütemechanismen und einer verteilten Betriebsmittelsteuerung. Damit wird es in Zukunft möglich sein, die Dienstgütemechanismen in Abhängigkeit von den zur Verfügung stehenden Betriebsmitteln transparent für die Anwendung zu adaptieren.

Literatur

[1] C. Becker and K. Geihs. QoS as a Competitive Advantage for Distributed Object Systems. In *Proceedings of EDOC'98*, La Jolla, USA, Nov. 1998.

[2] C. Becker and K. Geihs. Generic QoS Specification for CORBA. In *Proceedings of KiVS'99*, Mar. 1999.

[3] C. Becker and K. Geihs. Generic QoS-Support for CORBA. In *Proceedings of ISCC'00*, Antibes, France, July 2000.

[4] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98*, The Lake District, England, Sept. 1998.

[5] ECMA. C# Language Specification. ECMA Standard 334, European Computer Manufacturers Association, Geneva, Switzerland, 2001.

[6] ECMA. Common Language Infrastructure. ECMA Standard 335, European Computer Manufacturers Association, Geneva, Switzerland, 2001.

[7] S. Frølund and J. Koistinen. Quality of Service Specification in Distributed Object System Design. In *Proceedings of the COOTS'98*, Santa Fee, USA, Mar. 1998.

[8] International Organization for Standardization. ITU-T X.901. ODP Reference Model Part 1. Technical Report SC21 N8926rev, ISO/IEC JTC1/SC21/N, 1995.

[9] K. Geihs and C. Becker. A Framework for Re-use and Maintenance of Quality of Service Mechanisms in Distributed Object Systems. In *Proceedings of ICSM'01*, Florence, Italy, 2001.

[10] F. Kon, F. Costa, G. S. Blair, and R. H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, Vol. 45, No. 6:33–38, June 2002.

[11] J. P. Loyall, D. D. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS Aspect Languages and their Runtime Integration. In *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*, volume 1511, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[12] OMG. The Common Object Request Broker: Architecture and Specification. Specification 02-06-33, Object Management Group, Inc., Needham, USA, 2002.

[13] I. Rammer. *Advanced .NET Remoting*. Apress, 2002.

[14] D. C. Schmidt. Middleware for Real-Time and Embedded Systems. *Communications of the ACM*, Vol. 45, No. 6:43–48, June 2002.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real–Time Object Request Broker. *Computer Communications Journal*, 21, 1998.