FAKULTÄT FÜR !NFORMATIK

Evaluierung der Kommunikation und Koordination von Middleware-Technologien in der .NET Umgebung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Yevhen Lozhkin
Matrikelnummer 0327349

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: A.o. Univ. Prof. Dipl.-Ing. Dr. eva Kühn
Mitwirkung: Univ.-Ass. Dipl.-Ing. Richard Mordinyi

Wien, 01.09.2009

(Unterschrift Verfasser) (Unterschrift Betreuerin)

Erklärung zur Verfassung der Arbeit

Yevhen Lozhkin, Donaustraße 105/2, 2344 Maria Enzersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 01.09.2009_____

Inhaltsverzeichnis

Kurzfassung

In dieser Diplomarbeit werden vier gängige Technologien für die Entwicklung von Verteilten Systemen untersucht. Es handelt sich dabei um die Technologien XVSM, .NET Remoting, WCF (Windows Communication Foundation) und ASP.NET.
Das Beispielsystem eines Verteilten Systems – ein System zur Parkplatzreservierung in Parkhäusern – erlaubt es, Einblick in die Arbeit der vier Middlewares zu erhalten und mit den Prinzipien des Systemaufbaus mit Hilfe jeder einzelnen dieser Technologien vertraut zu werden. Des Weiteren ermöglicht das entwickelte Beispielsystem die Analyse der einzelnen Middlewares nach Schlüsselkriterien. Als Grundlage bei der Ausarbeitung der Schlüsselkriterien dienten die Praktikumsarbeit von Ayse Cicek „*The use of middleware technologies*“, die zum Zeitpunkt der Erstellung dieser Arbeit noch nicht abgeschlossen war, und der Artikel „How do J2EE and Microsoft's .Net compare in enterprise environments?“. Es werden die Schlüsselkriterien Notifikation, Synchronisation, Plattformunabhängigkeit, Verbindungsaufbau, Skalierbarkeit Wartung und Evolution, Komplexität und Transparenz der Technologie, Ausfallssicherheit, Installation und Konfiguration der Plattform sowie die Anzahl der Lines of Code betrachtet.
Am Ende der Arbeit wird jede der vier Technologien bewertet. Die Bewertung findet auf Basis der gesammelten Erfahrungen bei der Entwicklung des Parkplatzreservierungssystems sowie einer Code-Analyse statt. Das Ergebnis dieser Diplomarbeit stellt einen Vergleich der Technologien untereinander dar.

Abstract

In this diploma thesis four established technologies used for developing distributed systems are investigated. These technologies are XVSM, .NET Remoting, WCF (Windows Communication Foundation) and ASP.NET.
The given sample of a distributed system – a reservation system for parking facilities – grants insight in the working behaviour of those middleware systems as well as the principles of their system architecture.
Further this developed sample system allows an analysis of the individual middleware systems by defined key criteria.
As a basis for the elaboration of the key criteria served the practical work of Ayse Cicek "The use of middleware technologies" (which was not yet completed when this work was written), and the article "How do J2EE and Microsoft's.Net compare in enterprise environments?". Ten key criteria are analyzed: notification, synchronisation, platform independence, connection linking, scalability, maintenance and evolution, complexity and transparency of the technology, reliability, platform installation and configuration and the lines of code (LOC).
The work concludes with an appraisal of all four technologies. This assessment is based on the experience gained through the process of developing the parking space reservation system and on a supplemental code analysis. The result of this diploma thesis then finally presents a comparison of the investigated technologies.

Danksagung

Ich möchte meiner Frau Maria Weissenböck für ihre Unterstützung und große Hilfe beim Verfassen dieser Diplomarbeit danken.
Des Weiteren gilt mein Dank eva Kühn für das Vorschlagen des interessanten Themas und die kompetente Betreuung sowie meiner Kollegin Ayse Cicek für die konstruktive Zusammenarbeit und den regelmäßigen und fruchtbaren Informationsaustausch.
Außerdem danke ich Richard Mordinyi für das genaue Korrekturlesen und seine sehr konstruktiven Anregungen zur Verbesserung dieser Arbeit.
Ich danke auch meinem Vater Oleksandr Lozhkin, der mich mein ganzes Studium hindurch unterstützt hat.

1 Motivation

Middleware-Technologien für Verteilte Systeme [1] haben in letzter Zeit ein sehr hohes Entwicklungsniveau erreicht. Mit Hilfe des OMG-Konsortiums und Firmen wie Microsoft oder Sun Microsystems wurden verschiedene Middleware-Architekturen, die den Aufbau von Verteilten Systemen unterstützen, erzeugt und weiterentwickelt. Middleware bezeichnet in der Informatik ein anwendungsneutrales Programm, das so zwischen Anwendungen vermitteln, dass die Komplexität dieser Applikationen und ihrer Infrastruktur verborgen wird.[1]

Es gibt viele verschiedene Quellen ([2], [3], [4], [5]) zu den einzelnen Middleware-Technologien. Betrachtet man die gängigen Handbücher (z. B. [6], [7]), in denen je eine Middleware genau beschrieben wird, stellt man fest, dass zwar zahlreiche spezifische Details der jeweiligen Architektur aufgegriffen werden, jedoch selten Vergleiche zwischen den Technologien und/oder Sprachen angestellt werden (z. B. [8], [9], [10], [11]).

Wie entscheidet man, ob man die eine oder die andere Architektur für seine Anwendung verwenden soll? Wie kann man schnell, ohne in die Tiefe einer Middleware einzudringen, sagen, welche der gängigen Technologien für eine Aufgabe besser geeignet ist? Welche Kriterien sind bei der Auswahl der Architektur zu beachten?

In Rahmen diese Diplomarbeit wurde versucht, diese Fragen zu beantworten. Mit Hilfe des hier durchgeführten Technologievergleichs kann jeder für sich entscheiden, welche Architektur in seinem / ihrem speziellen Fall geeigneter ist.

Da in dieser Arbeit das gesamte Angebot an Middleware nicht erfasst werden konnte, werden die meistverwendeten Middlewares aus der „.NET-Welt“ (.NET Remoting, ASP.NET, WCF) mit der neuen XVSM Architektur verglichen.

XVSM (engl. eXtensible Virtual Shared Memory) ist eine Space-basierte Technologie[2], die von der Space Based Computing Group am Institut für Computersprachen der Technischen Universität Wien entwickelt wurde.

Der Technologienvergleich für die „Java-Welt“ wird in der Diplomarbeit von Ayse Cicek [11] separat behandelt.

[1] siehe http://de.wikipedia.org/wiki/Middleware. (letzter Zugriff am 20.08.09)
[2] siehe http://www.xvsm.org (letzter Zugriff am 20.08.09)

Kapitel 2 der Diplomarbeit befasst sich mit der Aufgabenstellung und der gewählten Lösung. Es wird aufgezeigt, auf welche Weise und mit welchen Mechanismen der Technologienvergleich durchgeführt wird. In diesem Kapitel wird eine für die Verteilten Systeme typische Aufgabe definiert, anhand derer die Middelware-Architekturen verglichen werden.
Kapitel 3 umfasst eine Technologieübersicht. Die Middleware-Technologien .NET Remoting, ASP.NET, WCF und XVSM.NET werden aus Sicht der aktuellen Entwicklungen im Bereich Enterprise-Lösungen beleuchtet.
In Kapitel 4 wird die Programmierung des Beispiels in jeder der Technologien beschrieben: verwendete Algorithmen, Datenstrukturen. Des Weiteren werden Besonderheiten und Probleme der Entwicklung in den vier Technologien aufgegriffen.
In Kapitel 5 werden die gewonnenen Erkenntnisse systematisiert, bewertet und einander gegenüber gestellt.
Im letzten Kapitel werden die Ergebnisse des Technologievergleichs präsentiert und Schlussfolgerungen gezogen.

2 Technologienübersicht

.NET Remoting, ASP.NET und WCF erlauben keinen direkten Zugriff auf gemeinsame Daten s.g. „Shared Data“. Um diesen Zugriff zu erhalten, wird in dieser Diplomarbeit die Event Basierte Architektur (siehe [12]) als Umweg benutzt.
Es gibt sehr viele Computersysteme, die ursprünglich auf bestimmte Events reagieren sollten. Ein paar Beispiele dafür sind: eine Firma bekommt eine neue Bestellung; ein Web-Server bekommt die Anfrage eines Users, der auf eine bestimmte Website zugreifen will; das vordere linke Rad eines Autos bremst. In keinem der oben beschriebenen Fälle fordert das System (Bestellungssystem, Web-Server, Bremsanlage) eine Aktion. Stattdessen verursacht das von Außen initiierte Event eine Veränderungen der realen Welt oder in der Computerarbeit. Eine Architektur, die keine Kontrolle von Außen benötigt, und die in Echtzeit auf Events reagiert, heißt ereignisgesteuerte Architektur (von engl. event-driven architecture, EDA) [WR 1]. Um mittels EDA das „Shared Data“-Konzept zu erreichen, muss das System so eingerichtet werden, dass alle beteiligten Anwendungsdomänen nach einen bestimmten Event Daten bekommen und durch die Manipulationen der Daten (ändern, löschen usw.) Events auslösen.
Was unterscheidet EDA von anderen Software-Architekturen, die heutzutage sehr aktiv verwendet werden? Die ereignisgesteuerte Architektur hat folgende Eigenschaften:

- *„Erzeuge-Komponente (Publishers)“* – Komponente des Systems, die ein Event auslösen kann.
- *„Konsument-Komponente (Subscribers)“* – Komponente des Systems, die notifiziert werden und auf ein bestimmtes Ereignis reagieren kann.
- *Breitband Nachrichten* – Das System notifiziert alle „interessierten“ Komponenten. Mehr als eine Komponente des Systems kann eine Nachricht über ein Event gleichzeitig empfangen und bearbeiten.
- *rechtzeitig* – Die Komponente publiziert Events in quasi Echtzeit, anstatt sie lokal zu speichern und darauf zu warten, dass sie abgefragt werden.
- *asynchron* – Komponente, die Eventnachrichten versendet; wartet nicht auf Empfang und Bearbeitung der Nachrichten durch andere Komponenten.

- *Ontologie* – Das System klassifiziert und beschreibt den Namen eines Events normalerweise in Hierarchiedarstellung. Die „Empfänger-Komponenten“ können sowohl auf ein konkretes Event als auch auf Events einer bestimmten Sorte warten.

Die ereignisgesteuerte Architektur ist eine sehr elegante und einfache Architektur, weil sie auf Ähnlichkeiten mit der realen Welt basiert. Der Observer ist ein Entwurf-Pattern der Kategorie Verhaltens-Patterns.
Heutzutage ist der Begriff „ereignisgesteuerte Architektur“ untrennbar mit dem der Verteilten Systeme verbunden. Oft sind verschiedene Komponenten von ereignisgesteuerten Systemen im Netz verteilt. Zur Erleichterung der Entwicklung solcher Systeme wurden viele verschiedene Technologien und Arten von Middlewares erzeugt.
Im Weiteren untersuchen wir vier verschiedene Technologien, bewerten sie anhand eines realen Beispiels und vergleichen die gewonnen Ergebnisse am Ende.

2.1 XVSM

XVSM (XcoSpaces.NET siehe [13]) ist eine Middleware, die auf den Grundprinzipien des Space based computing (anders: Shared data space)[3] basiert. Es ist eine relativ neue Architektur, die am Institut für Computersprachen der TU-Wien, von der Space Based Computing Group unter der Leitung von eva Kühn entwickelt wurde. Wie alle anderen Space basierten Technologien (Tuple Spaces (Linda) [WR 2], Java Spaces [WR 3], T-Spaces [WR 4], Corso (Coordinated Shared Objects) [14],XML Spaces [15]) ist XVSM.NET auf dem Begriff des abstrakten Netzwerkraums, der aus verteilten Objekten besteht, begründet.

Die Space based computing Architektur ist eigentlich eine natürliche Weiterentwicklung der Architekturen für verteilte Netzwerktechnologien, da alle Peers im Space gemeinsame Daten haben. Das vereinfacht die Kommunikation und Kollaboration zwischen den Peers, was ein Ziel der Erzeugung eines Peer-2-Peer Netzwerks darstellt.

[3] mehr zu dem Thema siehe http://www.spacebasedcomputing.org/ (letzter Zugriff am 20.08.09)

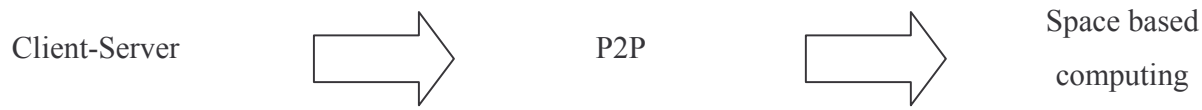

Abbildung 1: Evolution der Architekturen für verteilte Netzwerktechnologien

Die Grundprinzipien der XVSM - Kommunikation sind Container[4], die im Space erzeugt werden. Nachdem man Zugriff auf diese Container bekommen hat, kann man Lese- und Schreibe-Operationen durchführen. Um Zugriff zu bekommen, genügt es, über den Namen und die Adresse des Containers eine Referenz zu diesem zu bekommen. Jeder Prozess, der auf den Container Zugriff bekommt, kann mit dem Container folgende Operationen durchführen (mehr dazu siehe [16]):

- *Take* – Lesen mit Löschen (nach dem Lesen werden die gelesene Daten aus dem Container gelöscht);
- *Read* – Standardleseoperation (nach dem Lesen werden die gelesene Daten weiter im Container existieren und für weiter Operationen erreichbar sein);
- *Shift* – Ersetzen (bereits im Container existierende Daten werden durch andere Daten ersetzt);
- *Write* – Hinzufügen neuer Daten in den Container;
- *Destroy* – einen Container aus dem Space löschen;

Für all diese Operationen wurden Transaktionsmechanismen vorgesehen [17], um Inkonsistenz zu vermeiden. Bei erfolglosem Commit der Transaktion, wird ein automatisches Rollback durchgeführt. Transaktionen sind notwendig, wenn man eine Anwendung für Unternehmensniveaus (Enterprice level) entwickelt. Denn dabei spielen die Transaktionen eine große Rolle.

[4] Ein Container ist eine strukturierte Einheit zur Datenaufbewahrung, die Einträge enthalten kann, die Einzelinformationen entsprechen.

Am interessantesten für XVSM ist die Einführung eines Benachrichtigungsmechanismus (Notifikation). Der Mechanismus erlaubt es, die Middleware schnell und bequem an die Anforderungen einer ereignisorientierten Anwendung anzupassen. Im Vergleich zu anderen Middlewares, die Notifikationen benutzen, funktioniert der Benachrichtigungsmechanismus, wie alle anderen Features dieser Middleware, sehr einfach. Man kann Benachrichtigungen für eine Aktion mit einem konkreten Container verbinden, zum Beispiel kann das Hinzufügen neuer Daten in einen Container der Auslöser für eine Notifikation sein.

Während der Entwurfsphase der Middleware wurden von den Entwicklern mehrere Standarttypen von Containern, die ein Programmierer für seine eigenen Anwendungen brauchen könnte, festgelegt, z. B.:

- *Fifo Coordinated Container* (das Objekt, das als erstes im Container gespeichert wurde, wird als erstes gelesen; ähnlich wie bei einer Warteschlange);
- *Lifo Coordinated Container* (das Objekt, das als letztes im Container gespeichert wurde, wird als erstes gelesen; ähnlich wie bei einem Stack);
- *Key Coordinated Container* (Container, in dem das Lesen und Schreiben eines Objektes in einer bestimmten Position möglich ist);
- *Vector Coordinated Container* (Vektor-Container, Aufruf des Containerinhalts läuft ähnlich wie in einer Vektor-Klasse in Java bzw. einer ArrayList in .NET ab)

(mehr dazu siehe [13])

Außerdem hat XVSM, wie alle anderen Space based computing Middelwares, ein großes Abstraktionsniveau. Dies ist sehr hilfreich für Entwickler mit wenig Erfahrung im Bereich der Verteilten Systeme und kann ihnen helfen, Verteilte Anwendungen mit geringem Aufwand zu schreiben.

2.2 .NET Remoting

Verteilte Systeme sind heutzutage ein wichtiger Bestandteil vieler Applikationen. Eine Verteilte Anwendung ist eine Software, die Daten auf zwei oder mehreren Computern bearbeiten kann. Das bedeutet, dass auch die Daten, mit denen diese Anwendung arbeitet, verteilt sind. Mit dem Fortschreiten der objektorientierten Programmierung kann der Begriff Daten von dem Begriff Klassenobjekt ersetzt werden, das diese Daten inkapsuliert. Bei einer solchen Herangehensweise erweisen sich die so genannten Remote Objects als die meist genutzten in den Verteilten Systemen. Das Remote Object muss sich nicht unmittelbar innerhalb der Anwendungsdomäne befinden, die es benutzt. Für diesen Fall existiert das mächtige Tool .NET Remoting [6], das die Herstellung von und den Zugriff auf Remote Objects regelt.

.NET Remoting ist eine objekt-orientierte Technologie für die Entwicklung von Verteilten Technologien in Microsoft .NET Umgebung; sie fungiert auch als Basis für .NET Web Services[5]. .NET Remoting ist eine Technologie, die sehr breit im Einsatz ist. Sie ermöglicht leichtes Administrieren und Konfigurieren, verfügt über eine hohe Ausfallsicherheit und Produktivität und hat ein eigenes Sicherheitsservice, z. B. Authentifikation, Kryptographie oder Zugriffsteuerung. Einer der unbestrittenen Vorteile von .NET Remoting ist die Möglichkeit der Interaktionen mit anderen Verteilten Systemen. Diese wird mittels der Nutzung von offenen Standards, wie HTTP, SOAP, WSDL und XML, erreicht.

.NET Remoting gibt dem/der Entwickler/in die Möglichkeit, ein Remote Object mit Hilfe eines Links bzw. eines Wertes zu übermitteln. Bei der Übermittlung eines Objektes über einen Link wird das Objekt auf einem entfernten Rechner aufgerufen und nicht seine Kopie auf dem lokalen Rechner. Dies könnte zum Beispiel notwendig sein, wenn die Ressourcen (CPU, Speicher usw.) auf einen lokalen Rechner beschränkt sind.
Bei der Übermittlung eines Remote Objects mit Hilfe eines Wertes, überschreiten die Objekte die Domänengrenze der Anwendungen durch Serialisierung, d.h. der aktuelle Zustand des Objektes wird bestimmten Regeln zufolge in Bitreihenfolge dargestellt. Das serialisierte Objekt geht ins Netzwerk und erreicht die empfangende Anwendung in diesem Zustand. Die empfangende

[5] siehe http://de.wikipedia.org/wiki/Web_Service (letzter Zugriff am 20.08.09)

Anwendung, die dieselben Regeln wie bei der Serialisierung nur in umgekehrter Reihenfolge verwendet, deserialisiert das Objekt und wandelt es in eine identische, lokale Kopie des Remote Objects um.
Ein weiterer Vorteil der Technologie ist die Möglichkeit, das Objekt mittels «Server-Aktivierung» bzw. «Client-Aktivierung» zu aktivieren. Bei der Server-Aktivierung wird das Remote Object auf dem Server zugänglich gemacht, der für die Konfiguration des vorliegenden Objektes als allgemein bekannt verantwortlich ist; außerdem wird das Remote Object nur wenn notwendig aktiviert. In diesem Fall arbeiten alle Clients mit dem gegebenen Objekt. Bei der Client Aktivierung arbeitet jeder Client mit seinem eigenen Exemplar des Remote Objects. Man sollte wissen, dass in .NET Remoting das Aktivierungsregime unabhängig vom Typ des Objektes ist (d.h. ein und derselbe Typ kann je nach Einstellung in einer Anwendung vom Client und in einer anderen vom Server aktiviert werden).

Für eine bessere Verwaltung des Life-Cycles des Remote Objects in .NET Remoting wurde die Automatische Speicherbereinigung über Leasing[6] eingeführt, die vom LeaseManager verwaltet werden. Falls notwendig, kann man Sponsor-Objects für die Verlängerung des Leasings verwenden. Für die Übermittlung von Objekten/Nachrichten verwendet .NET Remoting Channel-Objects, die einen sehr flexiblen Transportmechanismus zu den beiden Seiten des Netzwerkes darstellen. Dieser Mechanismus hat das Potential, verschiedene Protokolle und Datenformate zu unterstützen. Als Standard wird TCP und HTTP verwendet, doch der/die Entwickler/in kann seinen/ihren eigenen Channel erstellen und verwenden.
Außerdem bietet .NET Remoting die Möglichkeit der Erweiterung auf und Verbindung mit spezialisierten Komponenten zu einem beliebigen Zeitpunkt eines entfernten Aufrufs, was diesem System größere Flexibilität gibt. Einer der größten Nachteile dieser Technologie ist zum gegenwärtigen Zeitpunkt die Plattformabhängigkeit.

[6] Leasing und Sponsorship sind Lösungen in .NET für die Steuerung der Existenzdauer von Remote Objects, mehr dazu siehe [23] oder http://msdn.microsoft.com/en-us/magazine/cc300474.aspx. (letzter Zugriff am 20.08.09)

2.3 WCF

Windows Communication Foundation (WCF) ist eine von vier neuen Komponenten des .NET Framework 3.0 (Windows Communication Foundation, Windows Workflow Foundation, Windows CardSpace und Windows Presentation Foundation) [18], siehe Abbildung 2. Sie gehört zu den Technologien der neuesten Generation für die Entwicklung Verteilter Systeme. Eigentlich ist .NET Framework 3.0 eine Erweiterung und Verbindung aller Technologien für Verteilte Systeme, die in den früheren Versionen von .NET Framework benutzt wurden (wie ASP.NET, .NET Remoting, Enterprise Services (COM+) usw.). [WR 5]

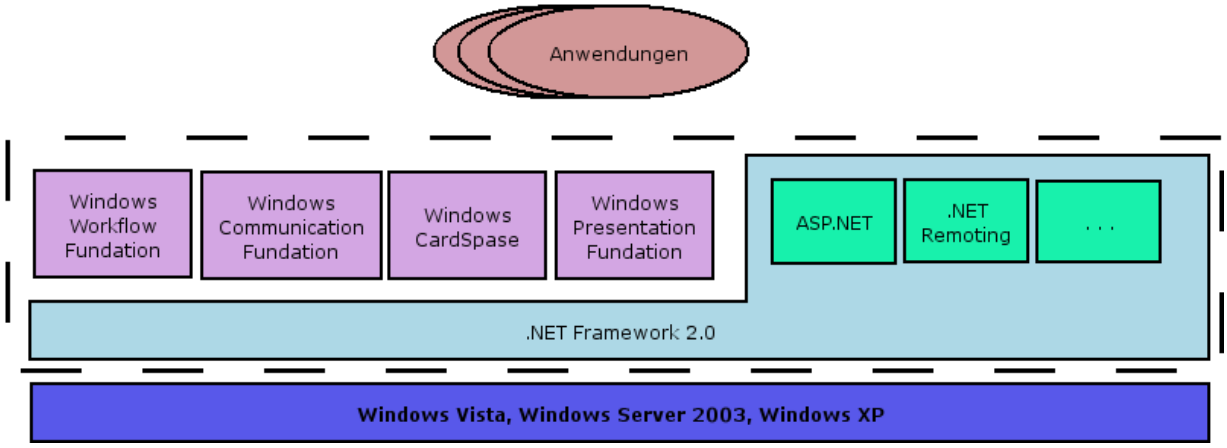

Abbildung 2: Architektur des .NET Framework 3.0

Grundlage für die WCF-Technologie ist eine SOA (service oriented architecture), bei der eine Anwendung ihre Services zur Verfügung stellt und ein Client diese benutzen kann. Für die richtige Kommunikation zwischen dem Client und dem Server muss man so genannte „Endpoints“ definieren. Ein Endpoint beinhaltet eine **A**dresse, eine **B**indung und einen s.g. „**C**ontract“ (**ABC**), siehe Abbildung 3:

- Die **A**dresse ist die URI eines Services.
- Die **B**indung definiert die Regeln während der Kommunikation zwischen Client und Server.
- Der **C**ontract definiert alle Methoden, die das Service zur Verfügung stellt, und ist eine Referenz auf die Service-Klasse.

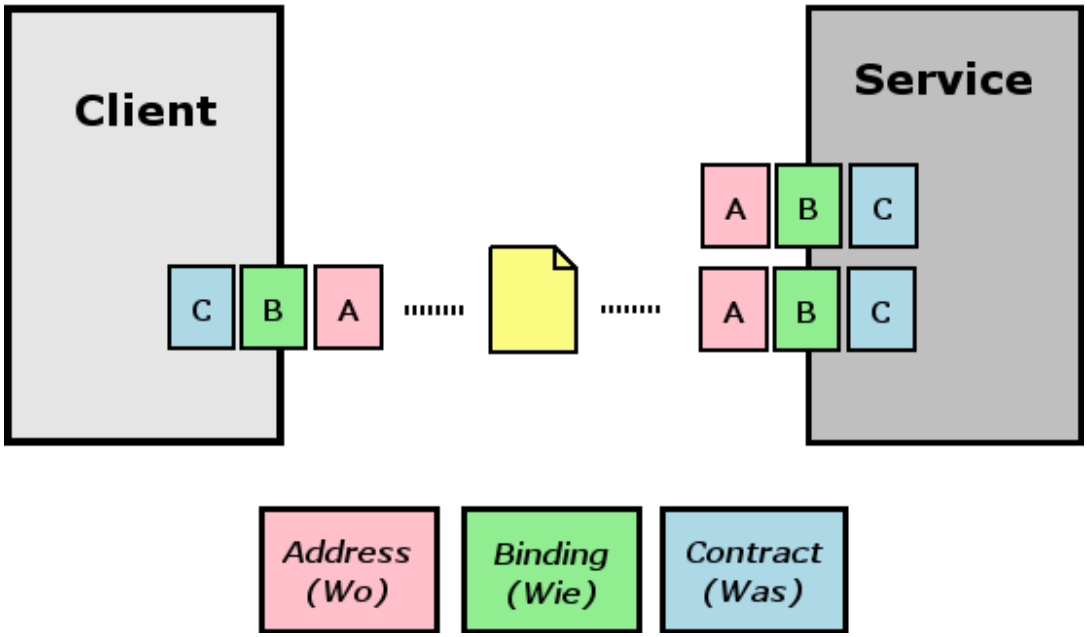

Abbildung 3: Kommunikationsaufbau in WCF

Alle Bindungen außer Namen und Namensräumen (engl. namespace) haben noch einige obligatorische Elemente, z.B. TcpTransportBindingElement, ReliableSessionBindingElement, SecurityBindingElement, alle diese Elemente beschreiben, wie eine Verbindung zustande kommt. Diese sind in der Bindung in Form eines Stacks gespeichert. Das erste Element entspricht der obersten Stackkomponente, das letzte Element der untersten Stackkomponente. Dabei sollen alle eingehenden Meldungen diesen Stack von unten nach oben durchgehen, und alle ausgehenden Meldungen umgekehrt – von oben nach unten. Wie man sieht, hat die Reihenfolge der Elemente unmittelbaren Einfluss auf die Aufarbeitung der Meldungen im Kommunikationsstack. WCF erleichtert dem/der Entwickler/in die Erstellung einer Bindung und bietet ihm/ihr ein Set bereits vorgefertigter Bindungen, die man in den meisten Fällen benutzen kann.

Außer einem normalen Contract, der bestimmt, was genau der Endpoint dem Client-Programm übergibt, stellt WCF einen Duplex-Contract zu Verfügung. Dieser Contract beschreibt nicht nur die Methoden, die das Service dem Client zu Verfügung stellt, sondern auch die Methoden, die der Client für den Server freigibt. Wie eine Bindung hat der Contract auch einen eigenen Namen

und Namensraum, deshalb können sowohl Contract als auch Bindung eindeutig in den Meta-Daten identifiziert werden.

Für die Beschreibung eines Dienstes, seiner Endpoints, Bindungen und Contracts (Klasse) existiert in WCF eine ServiceDescription – Struktur (Abbildung 4), die den Dienst großteils realisiert. Diese Struktur wird auch bei der Erzeugung von Meta-Daten, Dienst-Konfigurationen und Dienst-Kanälen verwendet.

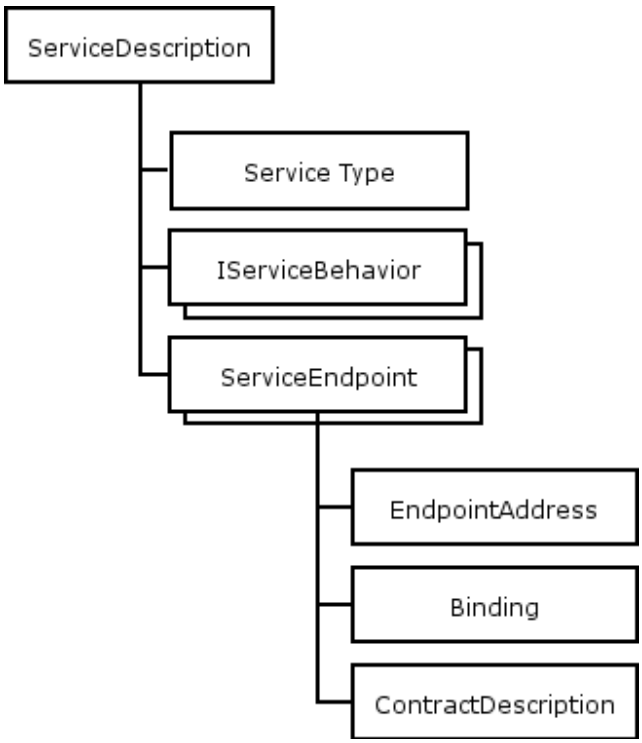

Abbildung 4: Struktur der ServiceDescription-Klasse

Normalerweise werden diese Objekte aus den Komponenten, die vom Framework angeboten werden, erzeugt. WCF sieht aber für Sonderfälle eine händische Erzeugung dieser Objekte vor. Obwohl man die ServiceDescription-Objekte direkt erzeugen und anfüllen kann, werden sie normalerweise automatisch als Teil der Systemfunktionalität erzeugt.

Für die vollwertige Arbeit einer Client-Anwendung braucht man einen Kanal, den man mit Hilfe eines Objektes der ChannelDescription-Klasse erstellen kann, siehe Abbildung 5.

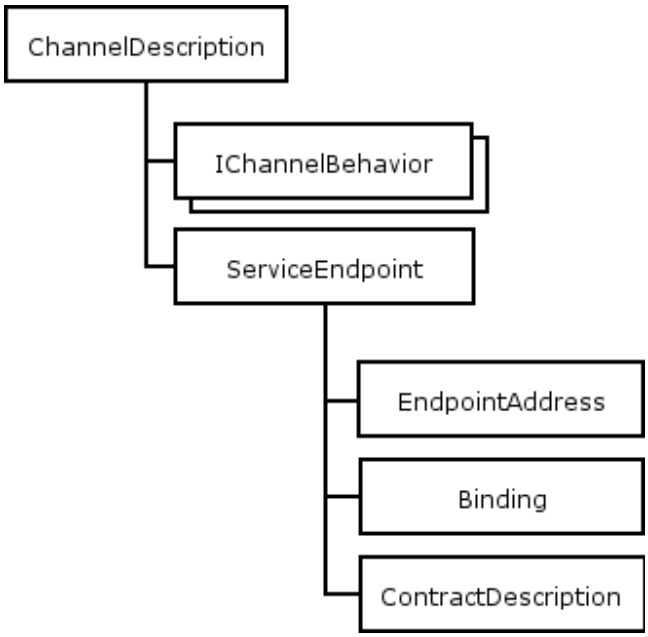

Abbildung 5: Struktur der ChannelDescription-Klasse

Der Kanal dient zur Verbindung des Client-Programms mit einem bestimmten Endpoint. Um das Kanalverhalten zu verwalten und einzurichten, hat die ChannelDescription-Klasse ein IChannelBehaviors-Inteface. Die ChannelDescription-Klasse hat zur Herstellung der Verbindung im Unterschied zur ServiceDescription-Klasse nur einen Endpoint (ServiceEndpoint).

Die Runtime eines WCF-Services wird normalerweise im Hintergrund erzeugt, und zwar mit Hilfe von `ServiceHost.Open` [19]. Das ServiceHost-Objekt steuert die Erzeugung der ServiceDescription für jedes Service und befüllt die ServiceEndpoint-Objekte mit Endpoints, die im Code oder in der Konfiguration definiert sind. Danach benutzt der ServiceHost ein ServiceDescription-Objekt für die Erzeugung eines Stacks von Kanälen in Form eines EndpointListener.

Die WCF Runtime ist ein Set von verschiedenen Objekten, die für die Zuweisung und den Empfang von Meldungen verantwortlich sind. Der EndpointListener stellt ein Äquivalent der ServiceEndpoint-Ausführung dar und beinhaltet einen Stack von Kanälen, der für das Senden und den Empfang einer Meldung zuständig ist. Meldungen sind eine Einheit des

Informationsaustauschs zwischen Client und Server. Je nach Kodierung kann man eine Meldung in ein binäres, SOAP-, XML- oder ein anderes beliebiges Benutzerformat serialisieren.
Die WCF-Kanäle kann man theoretisch in zwei Arten unterscheiden: Transport-Kanäle und Protokoll-Kanäle. Transport-Kanäle beschäftigen sich mit dem Senden und dem Empfang von Meldungen auf Basis von TCP, UDP oder MSMQ Protokollen. Die Protokoll-Kanäle beschäftigen sich mit der Bearbeitung und Veränderung von Meldungen (mit Hilfe des SOAP-Protokolls).

2.4 ASP.NET

ASP.NET (engl. Active Server Pages) ist eine Technologie für die Entwicklung von Web-Anwendungen und Web-Services [7]. Es ist auf die von Microsoft entwickelte .NET Framework 2.0 Plattform aufgebaut. Die Technologie wird in der Praxis sehr breit genutzt, insbesondere für die Entwicklung von Verteilten Systemen. Sie stellt eine typische SOA (service oriented architecture) dar, bei der ein zentraler Server seine Dienstleistungen – d.h. Services – anbietet, die von einem entfernten Computer – dem Client – in Anspruch genommen werden. ASP.NET ist eine objekt-orientierte Technologie, die es beim Schreiben von Web-Anwendungen ermöglicht, alle Klassen-Bibliotheken der Laufzeitumgebung (runtime environment) zu verwenden. Das ist relativ neu. In den Script-Sprachen sind objekt-orientierte Technologien nur begrenzt vorhanden und befriedigen nicht alle Anforderungen des modernen Programmierens.

Die .NET Umgebung enthält eine große Anzahl bereits entwickelter Bibliotheken mit verschiedenen Klassen, Interfaces, Strukturen usw. Da die Verwendung der Klassen in ASP.NET identisch mit ihrer Verwendung in allen anderen Arten von .NET-Anwendungen (Windows-Anwendung, Windows-Dienst oder Konsole-Anwendung) ist, stellt .die NET-Umgebung den Web-Entwickler/innen auch dieselben Werkzeuge zur Verfügung, wie für die Entwicklung von nicht-Web-Anwendungen. Noch ein Vorteil von ASP.NET gegenüber anderen Technologien zur Entwicklung von Web-Anwendungen ist, dass im Unterschied zu anderen Script-Programmiersprachen der ASP.NET-Code kompiliert anstatt interpretiert wird. Es ist laut [7] bekannt, dass man bei der Interpretation eines Programmcodes den Programmcode auf dem

Script-Host zeilenweise in maschinenorientierten Code umwandeln muss, was viel Zeit beansprucht. Die vollziehenden Dateien von ASP.NET hingegen sind schon kompiliert und erfordern keinen zusätzlichen Zeitaufwand. Die Vorkompilation geschieht im Moment des ersten Aufrufs der Website und cacht sich im Systemkatalog. Eine neue Kompilation ist nur im Falle der Veränderung des Ausgangscodes des Programms notwendig.

Die Kompilation der Programme in .NET geschieht in zwei Etappen. Die s.g. Vorkompilation verwandelt Code, der in C# (Java #, VB.Net) geschrieben ist, in einen Code der „Microsoft Intermediate Language“ (MSIL oder IL) [20], welche auch die Sprache der .NET Umgebung ist. Sie wird von CLR (Common Language Runtime) erkannt. Genau diese Vorkompilation bedingt die Unterstützung verschiedener Programmiersprachen in einem System, so kann zum Beispiel die Server-Komponente in C#.NET geschrieben sein und die Client-Komponente in VB.NET. Aber der herausragendste Vorteil von ASP.NET ist seine Funktionsweise innerhalb der CLR-Umgebung. Dies gibt dem Entwickler die Möglichkeit, beim Schreiben eines Programms folgende CLR-Funktionen zu verwenden:

- Automatische Speichersteuerung und Speicherbereinigung.
- Typensicherheit: Bei der Kompilation von Code fügt der Kompilator Informationen über alle verwendeten Klassen in das Assembly (kompilierte Datei) hinzu, was zur vollkommenen Selbstständigkeit des Assembly führt. Ein/e andere/r Entwickler/in kann es ohne zusätzliche Bibliotheken verwenden.
- Strukturierte Fehlerbearbeitung.
- CLR schafft die Möglichkeit, gleichzeitig mit mehreren Threads zu arbeiten. Dieses Feature stellt ein Threadpool zur Verfügung, das von verschiedenen Klassen verwendet werden kann.

Bei der gegenwärtigen Entwicklung des WWW gibt es viele verschiedene Browser auf verschiedenen Systemen oder von verschiedenen Herstellern, und dementsprechend viele verschiedene Standards. Deshalb muss der/die Entwickler/in bei der Erarbeitung von Web-Anwendungen diese Unterschiede berücksichtigen, und die von ihm/ihr entwickelten Anwendungen an den Browser des Clients anpassen. ASP.NET löst dieses Problem, indem es

dem/der Entwickler/in einen Satz von Elementen [7] der Web-Serververwaltung bietet. Dieser Satz generiert HTML-Seiten, die bereits an den Client-Browser angepasst sind. Ein Beispiel dafür sind die Elemente für die Steuerung der Verifikation von ASP.NET

Noch ein Plus von ASP.NET ist die einfache Installation und Konfiguration der fertigen Anwendungen auf dem Server, im Vergleich zu anderen Frameworks zur Entwicklung von Web-Services z.B. COM-Komponente, Perl, Java Service Pages, PHP usw. Bei der Installation eines .NET Frameworks werden alle notwendigen Klassen und Komponenten sowohl auf dem Server als auch auf dem Client-Rechner installiert und registriert. Deshalb reicht es in den meisten Fällen aus, die fertigen Dateien in den virtuellen Katalog zu kopieren. Bei der Konfiguration der Web-Anwendung ist auch die spezielle Datei web.config vorgesehen, die die Abhängigkeit der Anwendung von den Einstellungen des IIS (Internet Information Server)[7] minimiert.
Alle oben angeführten Punkte und die große Popularität der Windows-Plattform führen dazu, dass ASP.NET zum Standard bei der Entwicklung von Web-Anwendungen auf der Basis von Microsoft-Technologien geworden ist. Demnach stellt es für andere Technologien zur Entwicklung von Web-Anwendungen eine große Konkurrenz dar.

[7] siehe http://www.dotnetframework.de/%7B632A91E5-52D1-4191-881E-D18BD40936A5%7D.aspx (letzter Zugriff am 20.08.09)

3 Problemstellung und Lösung

3.1 Problemstellung

Die Entwickler/innen oder andere Personen, die darüber entscheiden, welche Technologie verwendet wird, haben regelmäßig die Qual der Wahl: „Welche Middleware setze ich ein, um diese konkrete Aufgabe schnell, zuverlässig und fehlerfrei lösen zu können? Soll ich die Middleware einsetzen, die ich bereits kenne? Und wenn ich eine für mich neue Middleware entscheiden will, wie groß wird der Aufwand sein?“ Wird die falsche Entscheidung getroffen, vergrößert sich der Aufwand der Entwicklungsarbeit stark.
Um solche Situationen zu vermeiden, sollte es Bewertungen von Technologien geben. Zu diesem Zweck wird im Rahmen dieser Diplomarbeit ein konkretes Programmierbeispiel genommen und mit den oben erwähnten Technologien gelöst. Auf Grundlage der auftauchenden Fragen, Probleme und Erfahrungen wird am Ende der Arbeit eine Tabelle zusammengestellt, die einen übersichtlichen Vergleich der Middlewares erlauben soll.
Als Beispiel wurde eine Anwendung mit Schichtenarchitektur gewählt. Es handelt sich dabei um ein Parkplatzreservierungssystem. Dieses System hat viele Besonderheiten, die bei der Entwicklung eines Verteilten Systems beachtet werden müssen, z.B. Kollaboration und gemeinsamer Zugriff auf Objekte, Verbindungsaufbau, Ausfalltoleranz usw. An diesen Punkten wird ersichtlich, wie die einzelnen Middelwares sie unterschiedlich implementieren können.

Beispielbeschreibung:
Im Rahmen dieses Beispiels soll ein Parkplatzreservierungssystem entwickelt werden. Das System dient dazu, das Finden eines Parkplatzes zu erleichtern. Besonders wichtig ist das System für Gäste, die sich zum ersten Mal in einer Stadt befinden.

In modernen Autos gibt es immer mehr digitale Geräte, wie z.B. Handys, Smartphons, Laptops, GPS-Geräte. Der Serviceanbieter, der den Kunden das Parkplatzreservierungssystem anbietet, stellt einen Dienst zur Verfügung, der die Reservierung eines Parkplatzes in einem Parkhaus ermöglicht.

Die Servicebenutzerin sendet ihre Position, die Parameter ihres Autos und den gewünschten Zielort an das System. Alle Parkhäuser, die am System teilnehmen wollen, müssen sich im Vorfeld im System registrieren lassen. Wenn das System eine Anfrage von einem Autofahrer bekommt, werden alle Teilnehmer davon benachrichtigt. Falls ein Parkhaus einen freien Platz hat, gibt es dem System Bescheid und das System leitet die Informationen an den Broker weiter. Der Broker entscheidet je nach seiner Businesslogik, welcher der zur Verfügung stehenden Parkplätze an die Autofahrerin zurückgeschickt werden soll.
Jedes Parkhaus verwaltet die Daten über seine aktuelle Auslastung auf einem eigenen Rechner. Dabei ist jeder Parkplatz mit einem Sensor und einer Signallampe ausgestattet. Der Sensor ermittelt, ob der Parkplatz besetzt ist, er steuert außerdem die Signallampe: rot steht für besetzt oder reserviert, grün für frei.
Am Ende des Reservierungsprozesses erhält die Autofahrerin eine Parkplatznummer, mit der der Platz problemlos zu finden ist. Sobald die Reservierung stattgefunden hat, wird die Signallampe des Parkplatzes auf rot gestellt, um anderen Autofahrern, die vermutlich früher an diesem Parkplatz vorbeikommen, zu signalisieren, dass er bereits reserviert ist. Wenn in einer bestimmten Zeitspanne keine Bestätigung des Sensors kommt, dass der Parkplatz tatsächlich besetzt wurde, wird die Signallampe wieder auf grün gesetzt und der Parkplatz im System freigegeben. Die Situation im Parkhaus soll immer auf dem aktuellen Stand sein und verändert sich im „Real Time“-Modus. D.h. wenn ein Auto weggefahren oder in einen Parkplatz eingefahren ist, soll die Information über den neuen freien/besetzten Parkplatz sofort in den Reservierungsprozess einbezogen werden. Das Parksystem kann beliebig kompliziert sein und mehrere Parkhäuser umfassen. Das System muss vermeiden, denselben Parkplatz gleichzeitig an mehr als ein Auto zu vergeben.
Ein typischer Use Case ist in Abbildung 6 abgebildet.

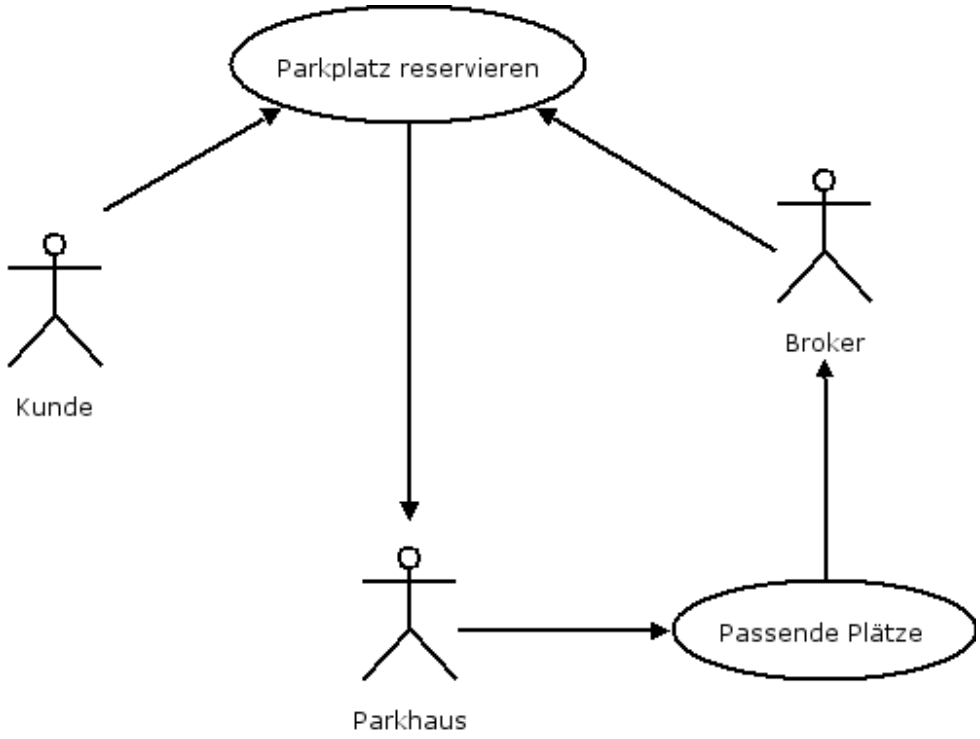

Abbildung 6: Use Case Parkplatzreservierung

Use Case:	Parkplatzreservierung
Primary Actor:	Kunde (Benutzer)
Scope:	Parkplatzreservierungssystem
Level:	Benutzerziel
Preconditions:	der Kunde hat Zugriff auf das Service, das System läuft, mind. ein Broker ist im System angemeldet
Minimal guarantees:	der Benutzer wird informiert, wenn für ihn ein oder kein Parkplatz reserviert wurde
Success guarantees:	der Benutzer bekommt einen reservierten Parkplatz
Trigger:	der Benutzer sendet eine Anfrage mit den Parametern seines Autos, seinen Standort und den gewünschten Zielort ans Netz

Main scenarios:

1. der Benutzer gibt seinen Parameter ein
2. der Benutzer sendet die Parameter ans System
3. der Broker bekommt die Anfrage und startet ein Timeout

4. der Broker bekommt Zugriff auf die Daten der Parkhäuser und reserviert für den Benutzer einen Platz, falls ein passender vorhanden ist
5. nach Ablauf des Timeouts, wird aus allen Plätzen, die für den Benutzer reserviert wurden, auf Basis der entsprechenden Entscheidungslogik der beste Platz ausgewählt
6. der Broker sendet den besten Platz an den Benutzer, alle anderen Plätze werden wieder als „frei“ markiert
7. der Benutzer bestätigt, dass er den Parkplatz in Anspruch nimmt

Extensions:

2a. es gibt keinen Broker im System
- eine Fehlermeldung wird angezeigt
- der Use Case wird beendet

4a. der Broker hat keinen Zugriff auf die Daten der Parkhäuser oder gibt es keine Parkhäuser im System
- nach Ablauf des Timeouts, wird der Benutzer informiert, dass das System keine verfügbaren Parkplätze hat
- der Use Case wird beendet

7a. der Broker bekommt keine Rückbestätigung des Benutzers
- nach einem festgelegten Zeitintervall wird der Parkplatz als „frei“ markiert
- der Use Case wird beendet

Der Use Case zeigt, welche Features das Parkplatzreservierungssystem zu besitzen hat.
Das Parkplatzreservierungssystem muss:

- verteilt sein (d.h. die verschiedenen Systemdomänen müssen auf voneinander unabhängigen Rechnern laufen können)
- Zugriff auf gemeinsame Daten bieten (d.h. die Daten über das Vorhandensein freier Parkplätze müssen allgemein verfügbar sein, für Lesen und Schreiben)
- eine sichere und zuverlässige Verbindungen garantieren (da es im Beispiel um eine echte Anwendung gehen soll, sollte es nicht zu Systemausfällen kommen)
- kollaborativ sein (d.h. das System muss die Kollaboration der Broker - falls es im System mehr gibt - unterstützen)

- gut skalierbar sein (d.h. die Erweiterung des System um zusätzliche Parkhäuser, Benutzer und Broker sollte die allgemeine Performanz nicht beeinflussen)
- einfach zu warten sein (d.h. Änderungen und Ergänzungen im Programmcode dürfen keinen großen Aufwand darstellen)

3.2 Lösung

Da der Vergleich der oben erwähnten Technologien objektiv und umfassend erfolgen soll, bestand das beste Vergleichsverfahren in der Erzeugung von ein und derselben Applikation mit Hilfe jeder der in Kapitel 2 genannten Middlewares. Natürlich tauchen beim Implementierungsprozess je nach Technologie spezifische Besonderheiten bzw. Schwierigkeiten auf, die aufgenommen und in einer Tabelle zusammengefasst wurden. Um einen objektiven Vergleich zu bewerkstelligen, wurden während der Entwicklung einige aussagekräftige Kriterien definiert. Die Kriterien sollen nicht zu eng fokussiert sein, sondern eher allgemein und auch für weniger erfahrene Entwickler verständlich und nachvollziehbar.

Im Laufe der Arbeit wurden folgende Kriterien zur Bestimmung der Komplexität und des Aufwands beim Programmieren festgelegt:

- Notifikationen: Eignung einer Technologie für die Realisierung eines Benachrichtigungsmechanismus. Dieses Kriterium gibt außerdem an, wie erfolgreich eine Technologie für die Entwicklung von EDA Systemen genutzt werden kann.
- Synchronisation: Zeigt, wie eine Middleware den gemeinsamen Zugriff von Prozessen auf Daten-Objekte regelt und die Daten dabei in konsistentem Zustand bleiben.
- Plattformunabhängigkeit: Indikator für die Möglichkeit, in einem Verteilten System Teile bzw. Domänen zu verbinden, die für verschiedene Plattformen und in verschiedenen Programmiersprachen geschrieben wurden.
- Verbindungsaufbau: Verhalten der Technologie und Aufwand beim Aufbau einer neuen Verbindung.
- Skalierbarkeit: Verhalten von Systemen bezüglich des Ressourcenbedarfs bei wachsenden Eingabemengen, d.h. beim Anstieg der Anzahl von Clients.

- Evolution und Wartbarkeit: Gibt an, wie leicht der Code einer Technologie verändert und bearbeitet werden kann, und welchen Einfluss dies auf alle anderen Systemteile hat. Außerdem wie sehr das System den Agilen Prinzipien entspricht.
- Komplexität und Transparenz der Technologie: Kriterium, das die Verständlichkeit und das Abstraktionsniveau der Technologie bestimmt.
- Ausfallssicherheit: Gibt das Sicherheitsniveau des Systems bezüglich Datenverlust bei unerwarteten Verbindungsunterbrechungen und anderen Störungen an.
- Installation und Konfiguration der Plattform: Zeigt die Komplexität bei der Installation der Middleware und die Transparenz der Standard-Konfigurationen der Middelware an.
- Lines of Code: Anzahl der Codezeilen, die benötigt wurden, um das Beispielsystem mit Hilfe einer Technologie aufzubauen. Hier werden nur funktionale Methoden und Klassen berücksichtigt, also Klassen die im Pattern MVC – **M**odel **V**iew **C**ontroller [12] als Controller bezeichnet werden.

Die Kriterienauswahl wurde beeinflusst von meiner Mitarbeit bei der Erstellung des Fragebogens „Einsatz von Middleware-Technologien“[8], der von meiner Kollegin Ayse Cicek im Rahmen ihrer Diplomarbeit verwendet wird [11]. Der Fragebogen (siehe Anhang 3, Kapitel 7.3) wendet sich an Entwickler/innen, die im Bereich Verteilte Systeme tätig sind und Erfahrungen mit verschiedenen Middelware-Technologien haben. Er versucht zu ermitteln, wie zufrieden die Entwickler/innen mit den verwendeten Middelware-Produkten sind und wieso.

Wie bereits erwähnt, wurde das in 3.1 beschriebene Beispiel mit Hilfe von vier Middleware-Technologien gelöst. Es handelt sich dabei um .NET Remoting, ASP.NET, WCF und XVSM. Da ein und dieselbe Aufgabe mehrmals implementiert werden musste, fand beim Programmieren der Komponenten eine Orientierung an den Regeln der Wiederverwendbarkeit [22] statt. Das heißt, dass die allgemeinen Komponenten, wie GUI, Sortierungsalgorithmen etc., für alle Anwendungen gleich sind. Die Methoden für Kommunikation, Informationsaustausch und Zugriff auf gemeinsame Daten jedoch unterscheiden sich je nach verwendeter Technologie. (Die

[8] Die Umfrage zum Zeitpunkt des Verfassens dieser Arbeit ist noch am laufen auf http://www.spacebasedcomputing.org (letzter Zugriff am 20.08.09) die Analyse und Auswertung werden später dort auch präsentiert. Mehr davon siehe [11] und [21]

allgemeinen Komponenten mussten nur in einzelnen Fällen technologiespezifisch angepasst werden.)

4 Entwicklung des Beispielsystems

Im 2. Kapitel wurde eine kurze Einleitung und allgemeine Beschreibung der verwendeten Technologien gegeben. Der beste Weg, um die Besonderheiten jeder Technologie zu verstehen, ist die Entwicklung einer typischen Verteilten Anwendung mit Hilfe jeder der beschriebenen Architekturen. Als Anwendung habe ich das in Kapitel 3.1 beschriebene „Parkplatzreservierungssystem“ gewählt.
Die Lösung dieses Beispiels sieht die Benutzung der gesamten Palette an Basisfunktionen (Verbindungsaufbau, Kommunikation, Ausfalltoleranz etc.) einer Middelware vor und stellt ein umfassendes Bild jeder der von uns verwendeten Middlewares zur Verfügung.

Zu Beginn werden die Anwendungsdomänen definiert, wie sie zusammenhängen und wie die Kommunikation zwischen ihnen abläuft:

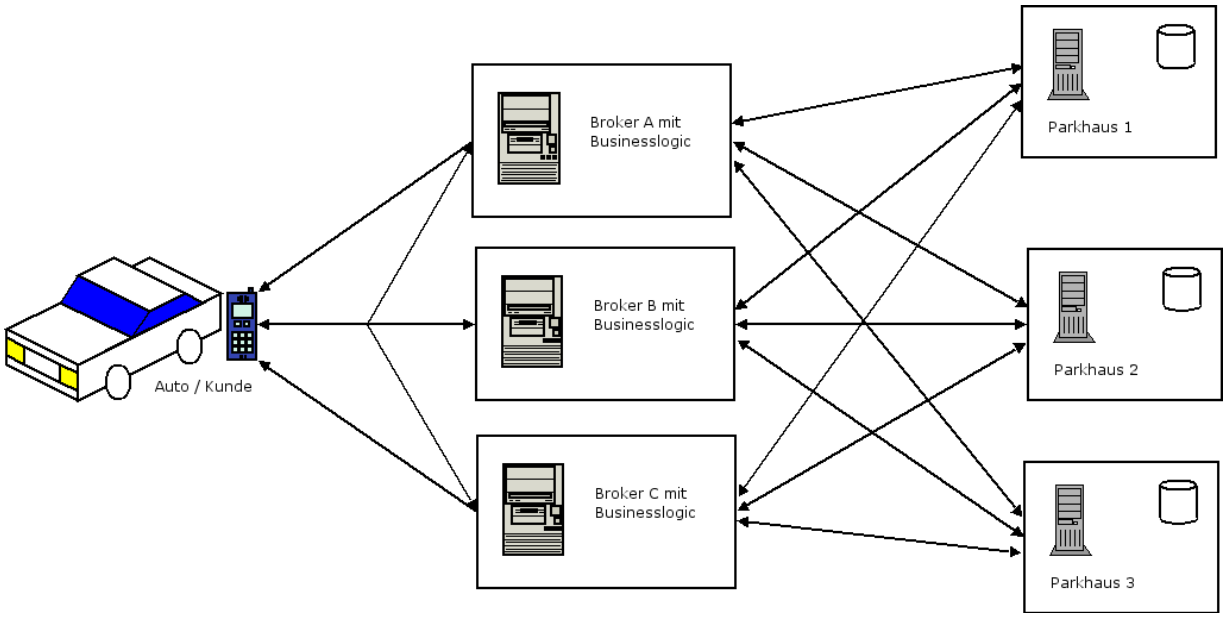

Abbildung 7: Architektur des Parkplatzreservierungssystems

Wie man in Abbildung 7 sieht, gibt es in der Anwendung drei verschiedene Domänen:

- die Kundendomäne, die eine Anfrage sendet;

- die Parkhausdomäne, die diese Anfragen entsprechend bearbeiten;
- die Brokerdomäne, die entscheidet, welcher der von den Parkhäusern angebotenen Parkplätze an den Kunden geschickt werden soll.

Die Anwendung ist so aufgebaut, dass die Entscheidungslogik, die auf dem Niveau der Brokerdomäne für die Entscheidungsfindung zuständig ist, sowohl allen mit dem System befassten Personen bekannt, als auch für alle verpflichtend ist. Die Parkhäuser sind autonom, d.h. die Logik für die Anfragebearbeitung kann individuell sein und ist für jedes Parkhaus separat. Jedes Parkhaus kann also indirekt über die eigene Preispolitik auf den Entscheidungsprozess Einfluss nehmen und so die Chancen erhöhen, dass dem Kunden ein Parkplatz in diesem Parkhaus vorgeschlagen wird. Zu diesem Zweck muss aber die Komplexität der Parkhausdomäne erhöht werden, zum Beispiel durch Hinzufügen einer Datenbank mit VIP-Kundendaten (im Normalfall bekommen die Parkhäuser nur die Kunden-ID, mit Zustimmung des Kunden werden aber auch die Stammdaten übermittelt), durch ein Rabattsystem, durch die Verwendung eines „Overbooking“-Szenarios etc.
Das System bietet den beteiligten Parkhäusern viele Möglichkeiten für die Konkurrenzpolitik, in der Hinsicht, dass die Parkhausdomäne dahingehend entwickelt und deployed werden kann. Darüber hinaus hat der Broker keinen Zugriff auf die Parkhaus- bzw. Kundendaten, sowie auf die Parkhauslogik zur Parkplatzaufteilung, was dem Parkhaus volle Datenhoheit und -sicherheit garantiert.
Außerdem können mehr als ein Broker am System teilnehmen. Dies bedeutet, dass der Benutzer die Möglichkeit hat, einen Parkplatz über mehr als einen Broker, falls mehrere im System angemeldet sind, zu suchen. Die Broker können sich zum Beispiel dadurch unterscheiden, dass sie verschiedene Entscheidungslogiken besitzen. Die Broker werden vom Benutzer gleichzeitig angesprochen. Dadurch entsteht ein konkurrierendes System, und es tauchen damit verbundene Synchronisationsprobleme auf: Die Parkhäuser stellen den Brokern ihre Parkplätze als „Shared Data“ zur Verfügung (jeder Broker hat also Zugriff auf diese Daten), und mehr als eine beteiligte Domäne braucht Zugriff auf diese Daten. Die Lösung des Problems kann je nach Middleware unterschiedlich sein.
Es sei erwähnt, dass das Schema in Abbildung 7 bei den einzelnen Technologien geringfügige Veränderungen erfahren kann, z. B. kann die Server-Domäne zum System hinzugefügt werden.

In den Unterpunkten dieses Kapitels wird detaillierter auf die verwendeten Algorithmen und Entwicklungsverfahren eingegangen, und zwar:

- auf den Entscheidungsalgorithmus der Brokerdomäne,
- auf eine der vielen verschiedenen Parkplatzverteilungslogiken für die Parkhäuser (gewählt wurde eine einfache Logik, die keinen zusätzlichen Programmieraufwand verlangt)
- und auf die Datenstruktur, mit deren Hilfe der gesamte Informationsaustausch im System stattfindet.

4.1 Verwendete Algorithmen

Aus Abbildung 7 (siehe S 28) wird ersichtlich, dass in diesem System drei verschiedene Domänen existieren, die miteinander kommunizieren sollen, z.B. der Kunde mit dem Parkhaus (den Parkhäusern), das Parkhaus mit dem Broker. In diesem System hat die Kundin selbst keine Möglichkeit zu entscheiden, welchen der verfügbaren Plätze sie bekommt. Um Entscheidungen zu treffen, würde sie eine Fülle an Informationen benötigen, wie z.B. eine Liste aller verfügbaren Parkplätze. Sie könnte dann einen Parkplatz auswählen und eine Reservierungsanfrage an das betreffende Parkhaus stellen. Nur in dem Fall, dass der Parkplatz immer noch frei ist, könnte die Kundin diesen dann reservieren. Ein solcher Ablauf würde dem Autofahrer viel Konzentration abverlangen und ihn vom Steuern des Autos ablenken. Deshalb gibt es in unserem System eine Brokerdomäne, die die Entscheidungsfindung übernimmt; außerdem wird so ein höheres Decoupling erreicht. Dafür eignet sich eine ereignisbasierte Architektur (anstatt einer gewöhnlichen servicebasierten Architektur). Das gemeinsame Sequenzdiagramm (Abbildung 8) sieht also folgendermaßen aus:

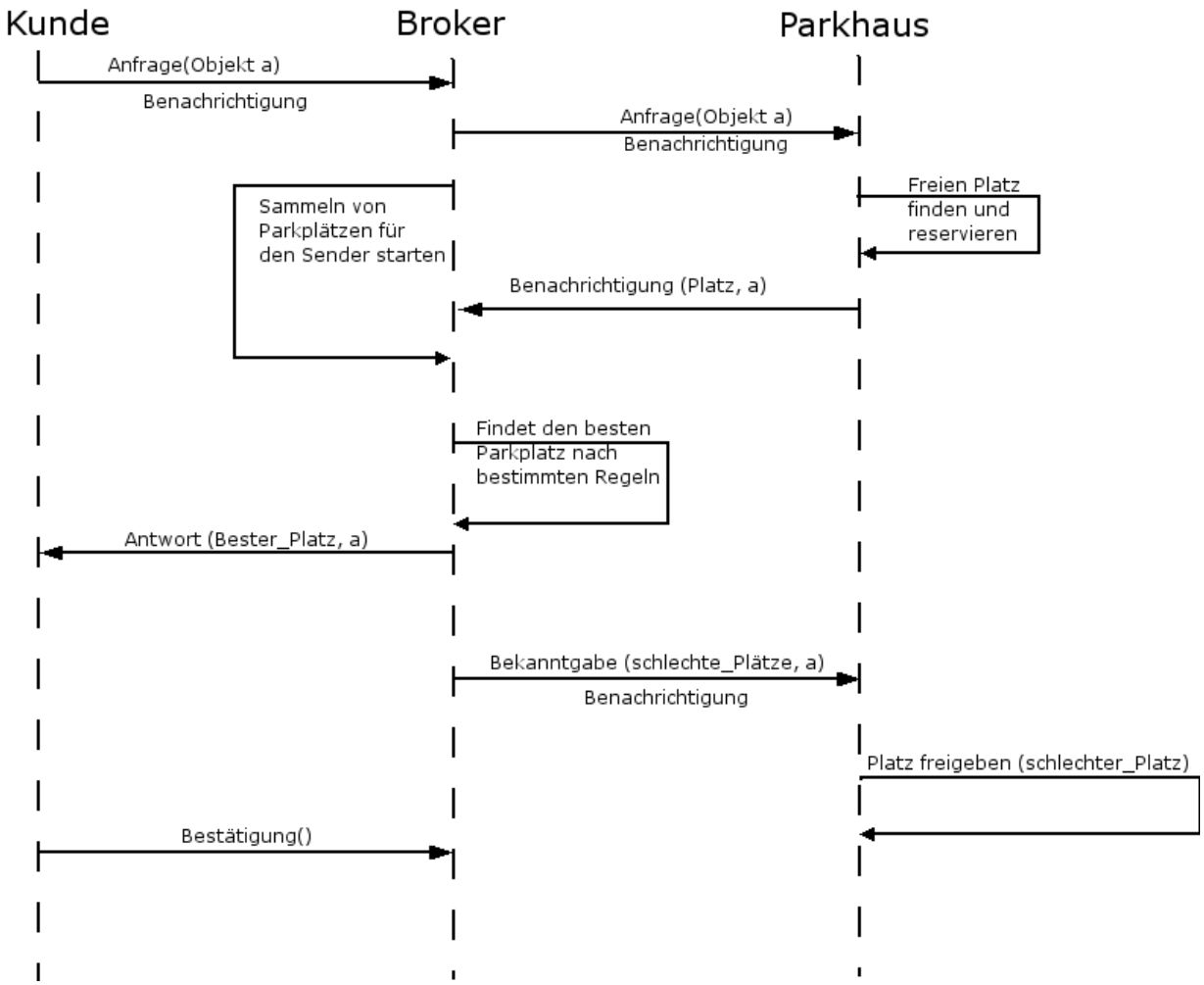

Abbildung 8: Allgemeines Sequenzdiagramm des Parkplatzreservierungssystems

Den allgemeinen Systemablauf kann man, wie folgt, beschreiben:
Will ein Kunde einen Parkplatz in einem teilnehmenden Parkhaus reservieren, muss er eine Anfrage stellen, in der alle notwendigen Parameter definiert sind (Parkplatzkategorie, Standort und Zielort). Alle Parkhäuser im System werden vom Broker davon benachrichtigt. In Folge wird jedes Parkhaus versuchen, einen Platz für diesen Kunden zu finden. Falls ein entsprechender Parkplatz vorhanden ist, wird der Platz für den Kunden reserviert und das Signallicht von Grün auf Rot gesetzt. Die Daten zu allen reservierten Parkplätzen werden an den Broker geschickt. Der Broker wird vor den Parkhäusern benachrichtigt und beginnt mit dem Sammeln von Parkplätzen für den Kunden. Der Broker trifft, nach Ablauf eines gesetzten Zeitintervalls (= Timeout), auf

Basis eines bestimmten Regelsets eine Entscheidung. Er wählt aus allen Parkplätzen nur einen aus und übermittelt ihn an den Kunden. Alle anderen, für den Kunden reservierten Parkplätze werden wieder freigegeben. Die entsprechenden Parkhäuser werden davon benachrichtigt, das Signallicht für den Parkplatz wird wieder auf Grün gesetzt. Falls der Kunde den angebotenen Parkplatz annimmt, schickt er eine Bestätigung an das System, falls nicht wird auch dieser Platz wieder freigegeben.
In einem System mit derartigem Aufbau gibt es Synchronisationsprobleme, z. B. wenn mehrere Brokerdomänen beteiligt sind und diese auf die Daten der Parkhäuser zugreifen müssen. Das Problem wird gelöst, indem die Parkhausdomäne den Brokern sofort nach der Kundenanfrage einen passenden Platz als Broadcast übermittelt und den Brokern nicht erlaubt, selbst in die Daten des Parkhauses zu schreiben. Dadurch wird auch vermieden, dass in einem Parkhaus vorübergehend mehrere Parkplätze für einen Kunden reserviert werden, nur weil der Kunde mehrere Broker benutzt.
Der eben beschriebene Systemaufbau ist sehr praktikabel und verfügt über eine gute Verwendbarkeit, jedoch führt er zu einer zusätzlichen Domäne im System. Zu der so genannten Koordinationsdomäne – einer Domäne, die sich um die Verbindung mit allen anderen Domänen kümmert. Da sich die Anzahl der Kunden und Parkhäuser (manche Parkhäuer schließen z.B. über Nacht andere nicht) während der Systemarbeit ständig ändert, übernimmt die Koordinationsdomäne die Kommunikation zwischen den anderen Domänen im System.

Noch ein wichtiger Algorithmus ist die Logik auf deren Grundlage der Broker seine Entscheidungen trifft und einen der angebotenen Parkplätze auswählt. In der Brokerdomäne gibt es so genannte „Entscheidungsfaktoren“ – Schlüsselkriterien im Entscheidungsalgorithmus. Man kann z.B. festlegen, dass der Preis das wichtigste Kriterium ist. Dann wählt der Broker jenen Platz aus, der am wenigsten kostet. Genauso kann man die Entfernung vom Parkplatz bis zum Zielort als Schlüsselkriterien setzen etc. Es besteht die Möglichkeit, mehrere Kriterien anzulegen, sodass Parkplätze, die im Bezug auf das erste Kriterium gleich gereiht wurden, nach einem zweiten (und gegebenenfalls dritten oder vierten) Kriterium gereiht werden können.
In unserem Beispiel gibt es vier Entscheidungsfaktoren:

- Preis – Auswahl des billigsten Parkplatzes;
- Entfernung zum Zielort – Auswahl, sodass der Weg vom Parkhaus zum Zielort möglichst kurz ist;
- Entfernung zum Parkhaus – Auswahl, sodass der Weg vom Standort zum Parkhaus möglichst kurz ist;
- Kategorie - Auswahl nach der passenden Parkplatzkategorie.

Darüber hinaus können alle vier Faktoren einen Einfluss auf das Ergebnis haben. In diesem Fall bekommt jeder Faktor einen Wichtigkeitsgrad, die Faktoren werden in Folge sortiert gespeichert. Die Reihenfolge **Preis, Kategorie, Entfernung zum Zielort, Entfernung zum Parkhaus** beispielsweise bedeutet, dass - falls zwei/mehrere preisgleiche Parkplätze existieren - die Parkplatzkategorie verglichen wird, wenn auch die gewünschte Kategorien in zwei/mehreren Parkhäusern vorhanden ist, wird die Entfernung zum Zielort verglichen usw.

Für die Realisierung dieses Algorithmus ist die Klasse Broker verantwortlich. Den Algorithmus für den Prozess ebenso wie die Beschreibung der gemeinsamen Klassen und des GUI befindet sich im Anhang dieser Arbeit (Kapitel 7.1 bzw. Kapitel 7.2).

Wie bereits erwähnt, wurden im Entwicklungsprozess alle Aspekte der Wartung und Wiederverwendbarkeit beachtet, um die Entwicklungsarbeit möglichst gering zu halten und eine gewisse Flexibilität in der Anwendung zu gewinnen.
Daraus ergibt sich folgende Anwendungsstruktur:

Das Assembly *Shared* beinhaltet Klassen, die für alle vier Systemvarianten gleich sind. Die Klasse *MyMessage* stellt das Informationsaustauschobjekt dar. Die statische Klasse *Util* beinhaltet statische Daten des Systems.

Das Assembly *ClientShared* beinhaltet die allgemeinen Klassen *Broker, KundenList, Parkhaus, ParkPlatz* und Klassen für Dialogfenster des Clients-GUI wie *InputDialogSender, InputDialogParkhaus, InputDialogBroker* und *PleaseWaitDialog*.

Das Assembly *Sender* hat Senderklassen: *Program* (startet den Sender) und *MsgForm* (GUI und Ablauflogik).

Das Assembly *Parking* beinhaltet die Parkhausklassen; *Program* (startet das Parkhaus) und *MsgForm* (GUI und Ablauflogik).

Das Assembly *Broker* beinhaltet die Brokerklassen: *Program* (startet den Broker) und *MsgForm* (GUI und Ablauflogik).

Das Assembly *Server* beinhaltet die Serverklassen. Diese können je nach Technologie verschieden sein (bzw. unter Umständen fehlen).

4.1.1 XVSM

Die XVSM-Technologie wurde in ihren Grundzügen bereits in Kapitel 2.1. beschrieben. Nun kann mit Hilfe der bereits gewonnen Kenntnisse ein Sequenzdiagramm erzeugt werden, das in Abbildung 9 zu sehen ist.
Man sollte erwähnen, dass die XVSM-Technologie für das „Shared Data“ Konzept durch die Unterstützung von Transaktionen[9] sehr gut geeignet ist. Aus Abbildung 3 wird ersichtlich, dass die Parkhäuser ihren Parkplatz allen beteiligten Brokern zur Verfügung stellen. Die Broker bekommen die Daten zu einem bestimmten Parkplatz als *Parking*-Objekt aus dem entsprechenden Container (siehe Kapitel 2.1) und können sie verändern. Diese Eigenschaft macht die Parkhausdomäne sehr „einfach“.

[9] siehe http://de.wikipedia.org/wiki/Transaktion_(Informatik) (letzter Zugriff am 20.08.09)

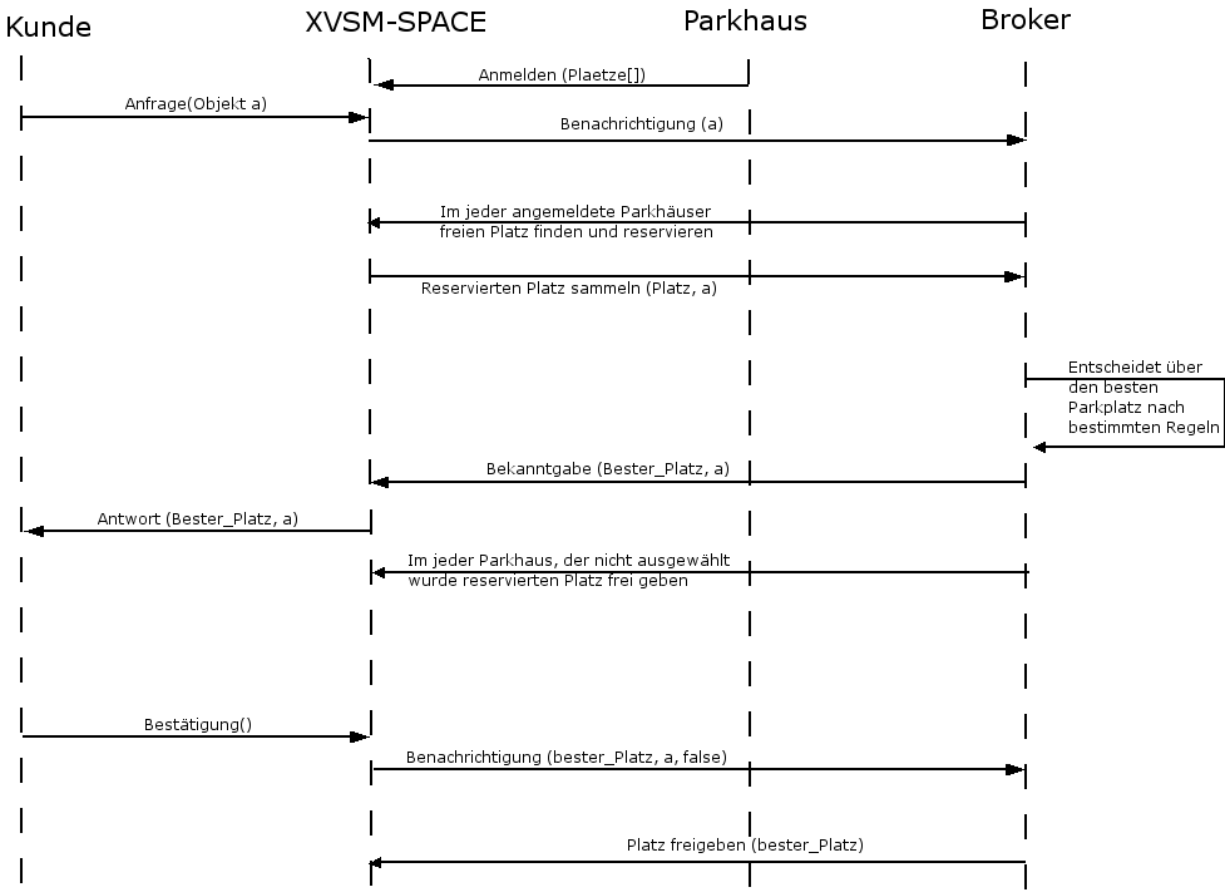

Abbildung 9: Sequenzdiagramm des „XVSM-Parkplatzreservierungssystems“

Es sei erwähnt, dass die Koordinationsdomäne *XVSM-SPACE*, die man in der Abbildung sehen kann, abstrakt ist und nicht vom Entwickler erzeugt wird. Die Middlware übernimmt diese Aufgabe. Sie erzeugt die Domäne automatisch, wenn eine andere Domäne versucht, eine Verbindung mit dem Space aufzubauen:

```
kernel = new XcoKernel();
kernel.Start(0);
```

Daraus folgt, dass diese Variante unseres Beispiels ohne Server-Domäne auskommt, da jede am System teilnehmende Domäne sowohl Client als auch Server ist.
Im Programm läuft die Kommunikation zwischen allen Domänen über zwei Container ab:

```
private ContainerReference senderCRef – Kunden-Container;
private ContainerReference parkingCRef – Parkhaus-Container;
```

Die Container tragen die entsprechenden Namen:

```
private const String senderContainerName = "sender";
private const String parkingContainerName = "parkhaus";
```

All diese Container haben den Containertyp *KeySelector* [13], weil es in unserem Fall nicht um die Reihenfolge der eintreffenden Daten sondern um deren Adressierung geht. Die Kommunikation über die Container ist in Abbildung 10 dargestellt.

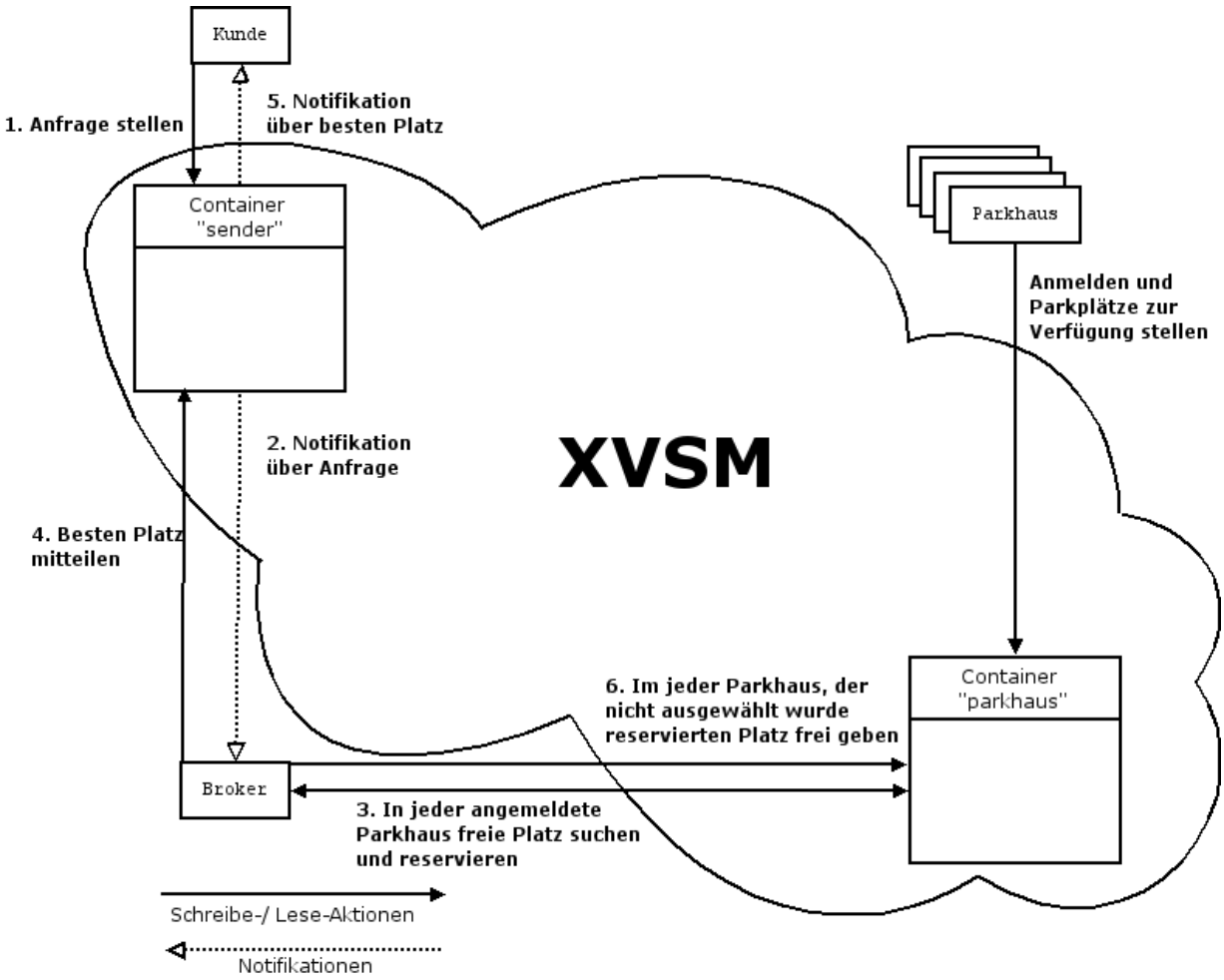

Abbildung 10: Kommunikationsdiagramm

Abbildung 10 zeigt, dass die Anmeldung des Parkhauses beim Reservierungssystem jederzeit passieren kann. Der Broker bekommt aber nur die Daten, die sich zum Zeitpunkt der Abfrage bereits im Container „parkhaus“ befinden.
Die Broker-Domäne ist die Hauptdomäne. Sie beinhaltet einerseits die Entscheidungslogik und wickelt andererseits die Container-Initialisierung ab (siehe Listing 1).

```
try
{
    // Referenz auf den Sender-Container mit diesem
    // Namen und dieser Adresse
    senderCRef = kernel.GetNamedContainer(remoteAddress, senderContainerName);
}
catch(XcoException e)
{
    // falls der Sender-Container nicht existiert: Container
    // erzeugen, dann Referenz bekommen
    kernel.CreateNamedContainer(null, senderContainerName, -1, false, new
KeySelector<string>("key"));
    senderCRef = kernel.GetNamedContainer(remoteAddress, senderContainerName);
}

try
{
    // Referenz auf den Parkhaus-Container mit diesem
    // Name und dieser Adresse
    parkingCRef = kernel.GetNamedContainer(remoteAddress,
parkingContainerName);
}
catch (XcoException e)
{
    // falls der Parkhaus-Container nicht existiert: Container
    // erzeugen, dann Referenz bekommen
    kernel.CreateNamedContainer(null, parkingContainerName, -1, false, new
KeySelector<string>("key"));
    parkingCRef = kernel.GetNamedContainer(remoteAddress,
parkingContainerName);
}
```

Listing 1: Container-Initialisierung

Als Folge bekommen alle übrigen Domänen (d.h. Kunde/Sender bzw. Parkhaus) Zugriff zu den nun bereits existierenden Containern nur mit der Space-Adresse `remoteAddress` und dem Namen `senderContainerName`:

```
senderCRef = kernel.GetNamedContainer(remoteAddress, senderContainerName);
```

Falls es im System keinen Broker-Container gibt, funktioniert das gesamte Parkplatzreservierungssystem nicht, da die Kommunikation zwischen den Parkhäusern und den Kunden über diesen Container erfolgt. Tritt dieser Fall ein, wird eine Exception generiert und die Ausführung des Programms abgebrochen.

Die Grundbausteine dieser Technologie sind einerseits Notifikationen bzw. Benachrichtigungen (Methoden, die bei einem bestimmten Ereignis im Container aufgerufen werden) und andererseits der Zugang zu den gemeinsamen Daten, die sich in einem Container befinden. Ersteres entspricht dem Grundprinzip der eventbasierten Architektur. Zweiteres entspricht dem Shared Data Konzept. In XVSM sieht die Realisierung von Benachrichtigungen folgendermaßen aus:

```
senderNotifier = kernel.CreateWriteNotification(senderCRef, null,
WriteOperation.Shift);
```

Listing 2: Deklaration der Benachrichtigung

In Listing 2 wird eine Notifikation für den Container „Sender“ ohne lokale Transaktionen generiert. (Für unser Beispiel genügt es, einen bereits in der Middleware eingebauten Transaktionsmechanismus zu benutzen.) Die Notifikation wird ausgelöst, wenn im Container eine Shift-Operation (=Event) durchgeführt wird.

```
senderNotifier.SetWriteCallback(Receive);
```

Die Code-Zeile oben definiert die Methode, die jedes Mal nach dieser Notifikation aufgerufen wird. Folgende Codezeile bringt alles zum Laufen:

```
senderNotifier.Start();
```

Die Benachrichtigungsbearbeitungsmethode `public void Receive` nimmt folgende Parameter an:

`Notification source` – Notifikation-Objekt, das die Source des Aufrufs ist

`WriteOperation op` – Operation, die durchgeführt wurde (write, shift, read, take, destroy)

`List<IEntry> entries` – Daten, die bei einer der oben genannten Operationen beteiligt sind

Der wichtigste dieser Parameter ist `List<IEntry> entries`, also der Datensatz, der die Notifikation verursacht hat. In unserem Fall ist das ein Objekt der *MyMessage*-Klasse. Auf diese Weise kann eine Domäne die Information direkt aus dem Container bekommen, sodass es nicht notwendig ist, den Container noch einmal aufzurufen, um den empfangen Datensatz zu lesen. Da es im Beispiel nur zwei Container gibt, über die aber mehrere verschiedene Domänen kommunizieren sollen (viele Kunden, viele Parkhäuser und ein Broker), wurde noch eine zusätzliche Eigenschaft definiert. Dabei handelt es sich um einen Schlüssel für den *KeySelector*-Container, der – je nach Container – die ID eines Kunden, Parkhauses oder Brokers ist. So muss nicht mehr auf jede Benachrichtigung reagiert werden, sondern nur auf jene, die mit der konkreten Domäne zu tun hat. Das reduziert die Rechnerbelastung, da das Programm keine Operationen durchführen muss, die nicht unbedingt notwendig sind.
Nach der Bearbeitung der Informationen, soll das System den Datensatz löschen, sofern er nicht mehr benötigt wird (siehe Listing 3).
Nun sind alle Details bekannt und der gesamte Systemablauf kann beschrieben werden:

Der Broker stellt eine Verbindung zum Space her und erzeugt dort zwei Container vom Typ *KeySelector* (mehr zu KeySelector und anderen Container-Typen siehe [23]), und zwar: „Sender“ und „Parkhaus“. Die Parkhausdomäne muss sich im Container „Parkhaus“ anmelden, wenn sie am Parkplatzreservierungssystem teilnehmen will, und ihr Parkhaus-Objekt für die gemeinsame Verwendung zur Verfügung stellen. Sie muss also beim Starten ein Parkhaus-Objekt erzeugen und dieses dann in den Container „Parkhaus“ schreiben. Der Kunde muss eine Verbindung zum Space aufbauen und seine Anfrage als *MyMessage*-Objekt in den Container „Sender“ schreiben. Darüber wird ein Broker bzw. werden mehrere Broker benachrichtigt (je nachdem wie viele am System teilnehmen). In dieser Implementierung muss kein Timer für den Kunden gestartet werden, weil der Broker direkt auf die Parkhaus-Objekte zugreifen kann, die im Container „Parkhaus“ gespeichert sind. Bekommt der Broker also eine Notifikation, d.h. eine Kundenanfrage, geht er den ganzer Container „Parkhaus“ durch und sucht in jedem dort gespeicherten Parkhaus-Objekt einen passenden Platz. (Falls es im System mehr als einen Broker gibt, bekommen alle Broker dieselben Parkplätze.) Der Broker sammelt die Daten aller gefundenen Plätze und speichert sie in folgender Datenstruktur:

```
private Dictionary<int, CustomerList> senders;
```

Wenn alle Parkhäuser durchgegangen worden sind, wird aus der Gesamtmenge an Parkplätzen nur ein Platz, der beste, ausgesucht und wieder als *MyMessage*-Objekt in den Container „Sender“ gestellt, und zwar an die Stelle im Container, an der sich bis jetzt die Kundenanfrage befunden hat. Im Parkhaus-Objekt werden Änderungen durchgeführt (d.h. der ausgewählte Parkplatz wird als „besetzt“ markiert) und wieder an seinen ursprünglichen Platz im „Parkhaus“-Container gestellt. Der Kunde wird über die Antwort auf seine Anfrage mittels Notifikations benachrichtigt, und der Kunde bekommt Informationen zu „seinem“ Parkplatz. Ist der Kunde damit zufrieden, muss er die Reservierung bestätigen. Daraufhin wird der Datensatz gelöscht (siehe Listing 3). Das Löschen löst eine Notfikation für das Parkhaus aus, welches entsprechend reagieren kann (z.B. Signallämpchen für diesen Parkplatz wird auf „rot“ gesetzt).

```
// Entry von Container löschen
kernel.Destroy(senderCRef, null, System.Threading.Timeout.Infinite, new
KeySelector<string>("key", "S" + this.myID));
```

Listing 3: Entfernung eines Datensatzes aus dem Kunden-Container

Die XVSM Technologie eignet sich sehr gut für die Arbeit mit Shared Data, weil jede Domäne auf die Datensätze eines Containers, der sich im Space befindet, zugreifen kann. Die Transaktionen dienen dazu, Fehler (wie z.B. einen Verbindungsausfall) beim Zugriff auf gemeinsame Daten zu verhindern; XVSM blockiert automatisch die Daten für die anderen Systemdomäne, wenn eine Domäne gerade Änderungen in den Daten vornimmt.
Leider sind nicht alle anderen Technologien für die Arbeit mit Shared Data geeignet, weil nicht in jeder Middleware vorgesehen ist, dass gleichzeitige auf ein und dasselbe Objekt zugegriffen wird, d.h. man muss ein so genanntes „workaround“ finden. Ist eine Middleware nicht für die Arbeit mit Shared Data geeignet, wird die Lösung auf Basis der eventbasierten Architektur aufgebaut.
Der Vorteil beim Programmieren mittels XVSM-Technologie liegt in den zwei Mechanismen – Notifikationen und Transaktionen. Der Benachrichtigungsmechanismus erlaubt es, der Kommunikation zwischen Conainern und Domänen auf einem sehr hohen Abstraktionsniveau ablaufen zu lassen. Das entwickelte System arbeitet stabil und ist für den Entwickler leicht erfassbar. XVSM erfordert vom Entwickler keine tiefgehenden Kenntnisse in diesem Bereich.
In den unten aufgelisteten Fehlerszenarien verhält sich das System wie folgt:

- Ausfall eines Parkhauses
  Bei einem kurzen Ausfall der Verbindung zu einem Parkhaus, gehen die Daten dieses Parkhauses nicht verloren, sondern verbleiben im Container „Parkhaus“, d.h. es wird bei der Auswahl des besten Parkplatzes berücksichtigt. Auch gehtd dank des eingebauten Transaktionsmechanismus die Benachrichtigung über die Zuteilung eines Parkplatzes zu einem Kunden nicht verloren.
  Bei einem längeren Ausfall werden die Daten aus dem Container entfernt und das Parkhaus wird bei der Platzauswahl nicht berücksichtigt.
- Ausfall eines Brokers
  Wie auch im Fall der Parkhäuser hat ein kurzzeitiger Verbindungsausfall dank des Transaktionsmechanismus und des Shared-Memory-Konzepts keinen Einfluss auf die Systemarbeit.
  Bei einem längeren Ausfall findet keine Parkplatzauswahl statt.

4.1.2 .NET Remoting

In diesem Kapitel gebe ich einen Überblick über dasselbe System, das nun aber mit Hilfe der .Net Remoting-Technologie entwickelt wurde. Wie bereits in Kapitel 3.2 erwähnt, ist .NET Remoting eine praktikable und weit verbreitete Technologie für die Entwicklung von Unternehmensprogrammen in der .NET Umgebung. Die Technologie erlaubt es, eine ganze Reihe von Kernkomponenten auf relativ niedrigem Niveau zu definieren und somit einen maximal geeigneten Netzkommunikationsmechanismus für jeden konkreten Fall zu erzeugen. Zum Beispiel kann .NET Remoting sowohl „Client-Server“-Patterns als auch „P2P“-Patterns sehr gut realisieren.
Für unser Beispiel eignet sich diese Middleware sehr gut, da sie großen Freiraum in der Programmentwicklung zulässt. So kann etwa das Parkplatzreservierungssystem mit Hilfe beider Patterns („P2P“ und „Client-Server“) ausgearbeitet werden. In beiden Fällen muss im System eine Server-Domäne vorhanden sein. Die Server-Domäne im „Client-Server“-Pattern stellt die Kommunikation zwischen allen Clients sicher. Im „P2P“-Pattern werden die Referenzen auf allen teilnehmenden Systemdomäne gespeichert, damit alle „Peers“ untereinander kommunizieren können.

Das Allgemeine Sequenzdiagramm (S. 31 Abbildung 8) lässt erkennen, dass das Parkplatzreservierungssystem bestimmte Ereignisse hat und die Domänen auf diese Ereignisse entsprechend reagieren sollen. Darüber hinaus eignet sich das „Client-Server“-Pattern mittels Remote Events[10] gut für die Entwicklung des Beispielsystems, siehe Abbildung 11.
Man spricht von einem Remote Event, wenn ein Client-Programm auf ein am Server stattfindendes Ereignis reagiert.

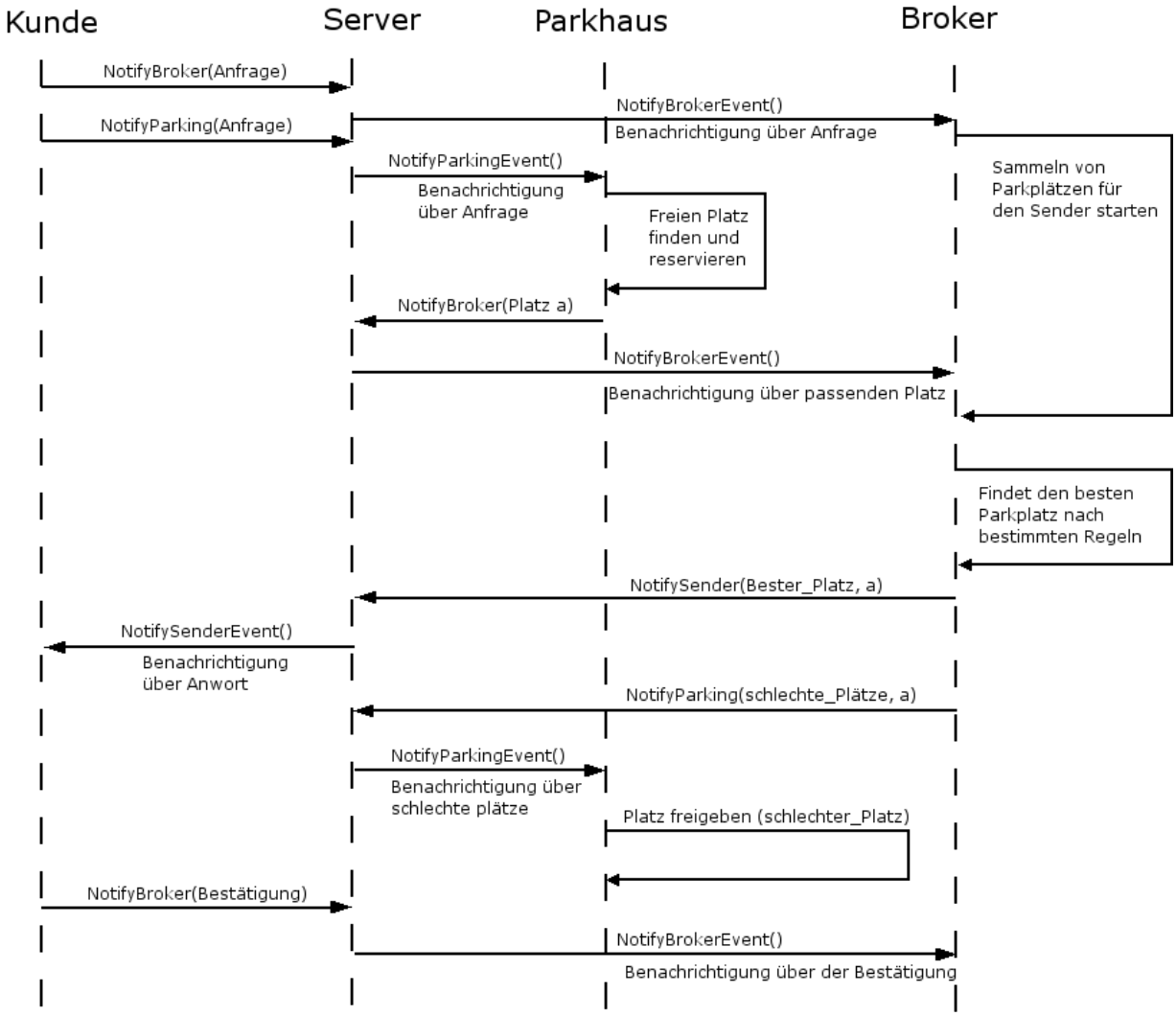

Abbildung 11: Sequenzdiagramm des „.NET Remoting-Parkplatzreservierungssystems“

[10] siehe http://www.gigaspaces.com/wiki/display/XAP66/About+Jini (letzter Zugriff am 20.08.09)

Dies ist möglich, da sowohl die Client- als auch die Server-Klasse eine Erweiterung der `MarshalByRefObject`-Klasse sind. Beim Erzeugen eines Client-Objekts wird somit ein Listener für ein bestimmtes Remote Event geschaffen.
Leider bringt dies ein zusätzliches Assembly, den Server, mit sich, was die Komplexität des Systems erhöht. Zu den bereits existierenden Assemblys (Shared, ClientShared, Sender, Parking, Broker) kommt das *Server*-Assembly. Dieses hat eine Klasse *Server*, die das Remote Event erzeugt und im Netz registriert:

```
// Erstellung von Clientformatierungssenken, das den BinaryFormatter zum
// Serialisieren von Meldungen für den Clientchannel verwendet, über
// den Remotemeldungen übertragen werden
BinaryClientFormatterSinkProvider clientProvider = new
            BinaryClientFormatterSinkProvider();
// Ezeugung von Formatierungschannelsenken des Servers,
// das den BinaryFormatter verwendet.
BinaryServerFormatterSinkProvider serverProvider = new
            BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
// Vorbereitung der Parameter für den Channel
IDictionary p = new Hashtable();
p["port"] = 12345;
p["name"] = "RemoteServer";
// Registrierung des Channels
ChannelServices.RegisterChannel(new HttpChannel(p, clientProvider,
            serverProvider));
// Registrierung des Remote Object = ServerObject mit Name „Server", erstellt
// mittels Server-Aktivierung im Singelton-Modus [6]
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerObject),
"Server", WellKnownObjectMode.Singleton);
```

Listing 4: Eröffnung einer Verbindung sowie Registrierung des Remote Objects auf dem Server

Das *Server*-Assembly enthält die *ServerObjekt*-Klasse, die das Interface des Remote Objects (*IServer*) realisiert. Das Server Programm wird als Konsolenanwendung geschaffen und beinhaltet keine GUI-Elemente.

Außerdem kommt das *Shared*-Assembly hinzu, in dem sich gemeinsame Klassen für alle Domäne befinden. Es handelt sich dabei um das Remote Interface *IServer*, die Interfaces *IBroker*, *ISender*, *IParkhaus*, *IForm* und die Klassen *NotifyEventArgs*, *SenderObjekt*, *BrokerObjekt* und *ParkhausObjekt*.

Das Remote Interface *IServer* bestimmt die Kommunikation mit der *ServerObjekt*-Klasse (siehe Listing 5):

```
namespace Shared
{
public delegate void NotifySenderEventHandler(object sender, NotifyEventArgs e);
public delegate void NotifyParkingEventHandler(object sender, NotifyEventArgs e);
public delegate void NotifyBrokerEventHandler(object sender, NotifyEventArgs e);

    public interface IServer
    {
        // Remote Event für Notifikation der Kundendomäne
        event NotifySenderEventHandler NotifySenderEvent;
        // Remote Event für Notifikation der Parkhausdomäne
        event NotifyParkingEventHandler NotifyParkingEvent;
        // Remote Event für Notifikation der Brokerdomäne
        event NotifyBrokerEventHandler NotifyBrokerEvent;
        // Notifikation der Kundendomäne
        void NotifySender(MyMessage message);
        // Notifikation der Parkhausdomäne
        void NotifyParking(MyMessage message);
        // Notifikation der Parkhausdomäne
        void NotifyBroker(MyMessage message);
    }
}
```

Listing 5: Interface *IServer*

Wie man sieht, werden hier die Delegate für die Bearbeitung der Remote Events, die Remote Events selbst und die Methoden, die auf einem Remote Object Clients aufrufen können, definiert. Die Interfaces *IBroker*, *ISender*, *IParking* beschreiben die Remote Objects für die jeweilige Domäne. Sie bestimmen welche Methode auf der Client-Seite aufgerufen wird, falls das Remote Event stattfindet, siehe Listing 6.

```
public interface IBroker
{
    void NotifyBrokerEventMethod(object sender, NotifyEventArgs e);
}
```

Listing 6: Interface *IBoker*

Das Interface *Iform*, siehe Listing 7, bestimmt die Methode, die ein Objekt vom Client-Programm aufruft, und wird nur für das GUI des Client-Programms verwendet:

```
public interface Iform
{
    void setValue (MyMessage message);
}
```

Listing 7: Interface *Iform*

Die Klasse *NotifyEventArgs* beschreibt die Event-Argumente und ist eine Erweiterung der *EventArgs* Klasse. Sie beinhaltet ein Objekt der *MyMessage*-Klasse. (siehe Listing 8)

```
[Serializable]
public class NotifyEventArgs : EventArgs
{
    private MyMessage _message ;
    public MyMessage Message
    {
        get { return _message; }
    }
    public NotifyEventArgs(MyMessage message)
    {
        _message = message;
    }
}
```

Listing 8: Implementierung der Klasse *NotifyEventArgs*

Die Klassen *SenderObject*, *BrokerObject* und *ParkingObject* beschreiben das Verhalten des jeweiligen Client-Programms. (siehe Listing 9)

```
public class SenderObject : MarshalByRefObject, ISender
{
    private IServer _server;
    private IForm _form;
    //Konstruktor
    public SenderObject(IServer server, IForm form)
    {
        _form = form;
        _server = server;
```

```
        // Anmeldung beim NotifySenderEvent
        _server.NotifySenderEvent += new
                  NotifySenderEventHandler(NotifySenderEventMethod);
    }

    ~SenderObject()
    {
        // Abmeldung beim NotifySenderEvent
        _server.NotifySenderEvent -= new
                  NotifySenderEventHandler(NotifySenderEventMethod);
    }
    // Definition des Listeners für NotifySenderEvent
    public void NotifySenderEventMethod(object sender, NotifyEventArgs e)
    {
        _form.setValue (e.Message);
    }
}
```

Listing 9: Implementierung der Klasse *SenderObject*

Der Konstruktor der „Client“-Objektklasse (in diesem Fall der Sender) schließt sich mit Hilfe des Interfaces des Remote Objects an das Remote Event *NotifySenderEvent* an und definiert den *EventHandler* (Listener). Die anderen Konstruktorparameter sind notwendig, um Zugriff auf das Client-GUI zu bekommen, d.h. dass bei der Ausführung von `NotifySenderEventMethod` die Methode *setValue* aus der Klasse *MsgForm* aufgerufen wird.
Der Ablauf des Client-Programms sieht folgendermaßen aus:
Beim Start des Client-Programms wird ein „Client“-Objekt (z.B. ein Sender) erzeugt, indem ein Verbindungskanal zum Remote Object geöffnet und eine Verbindung hergestellt wird (siehe Listing 10). Danach werden die Interfaces vom GUI-Formular und dem Remote Object als Konstrukorparameter an das *SenderObject* übergeben.

```
//Erzeugung des Clientformatierungssenken
BinaryClientFormatterSinkProvider clientProvider = new
BinaryClientFormatterSinkProvider();
//Erzeugung des Serverformatierungssenken
BinaryServerFormatterSinkProvider serverProvider = new
BinaryServerFormatterSinkProvider();
// Vorbereitung der Parameter für den Channel
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
IDictionary p = new Hashtable();
p[“port”] = 0;
p[“name”] = “Sender”;
// Registrierung des Channels
ChannelServices.RegisterChannel(new HttpChannel(p, clientProvider,
serverProvider));
// Verbindung mit Server
```

```
server = (Iserver)Activator.GetObject(typeof(Iserver),
"http://localhost:12345/Server");
// Erzeugung das Sender-Objects
MySender = new SenderObject(server, form);
```

Listing 10: Eröffnung eines Channels sowie Verbindung zum Remote Object am Client

Die Methode *sendMessage* erlaubt in diesem Fall das Aufrufen der Methode *NotifySender* auf dem Remote Object. (siehe Listing 11)

```
public void sendMessage (MyMessage message)
{
    server.NotifySender(message);
}
```

Listing 11: Implementierung der Methode *sendMessage*

Nachdem nun alle Informationen zur Kundenanfrage empfangen wurden, wird die Methode *sendMessage* aufgerufen. Diese wiederum ruft die Methode *NotifySender* auf dem Remote Object auf (siehe Listing 12).

```
public void NotifySender(MyMessage message)
{
    if (NotifyParkingEvent != null)
        // Initiieren des Remote Events zur Benachrichtigung der
        // Parkhäuser
        NotifyParkingEvent(this, new NotifyEventArgs(message));
    if (NotifyBrokerEvent != null)
        // Initiieren des Remote Events zur Benachrichtigung des
        // Brokers
        NotifyBrokerEvent(this, new NotifyEventArgs(message));
}
```

Listing 12: Implementierung der Methode *NotifySender*

Mittels der Methode, die in Listing 12 abgebildet ist, werden die Events *NotifyParkingEvent* und *NotifyBrokerEvent* initiiert, an die die Parkhausdomäne bzw. die Brokerdomäne bereits angeschlossen sind. In jeder Parkhausdomäne bzw. in der Brokerdomäne gibt es die Methode *setValue*, die das jeweils entsprechende Ereignis (*NotifyParkingEvent* oder *NotifyBrokerEvent*) bearbeitet. Diese Methode bestimmt aufgrund des Zustands des *MyMessage*-Objekts, welches konkrete Event passiert ist und wie das Programm darauf reagieren soll. Falls das Ereignis *NotifyBrokerEvent* passiert, wenn das Timeout für die betreffende Anfrage bereits abgelaufen ist, wird dieser Parkplatz sofort an das Parkhaus zurückgegeben, um den Platz freizugeben. Falls das

Ereignis vom Kunden initiiert wurde (also eine neue Kundenanfrage), legt der Broker in seiner Datensammlungsstruktur einen neuen Datensatz für die Anfrage an. Alle Parkplätze, die bis zum Ende des Timouts von den Parkhäusern kommen und sich auf diese Anfrage beziehen, werden in die Liste hinzugefügt. Am Ende des Timeouts sucht der Broker den besten Platz aus und initiiert mittels der Methode *sendMessage* auf der Server-Seite ein neues Ereignis *NotifySenderEvent*. Auf diese Weise bekommt der Kunde (= Sender) den Parkplatz zugeschickt. Die Parkhausdomäne ist immer im Wartezustand und reagiert bei neuen Kundenanfragen oder falls ein reservierter Parkplatz wieder freigeben werden muss auf das Ereignis *NotifyParkingEvent*.
Die .NET Remoting-Technologie hat sich für die Entwicklung von Verteilten Technologien als gut geeignet erwiesen, birgt aber gewisse Nachteile im Bereich der Abstraktion. Das Abstraktionsniveau ist niedrig und verlangt, dass sich der/die Entwickler/in gut mit Verteilten Systemen auskennt. Auch die Ausfallssicherheit zählt zu den Schwächen dieser Middelware; das System ist stark von der Serverdomäne abhängig. Ist der Server nicht erreichbar, ist das System unbrauchbar. Ist einer der Clients offline, kann das System zwar weiterarbeiten, die Kundenanfragen aber nicht mehr vollständig bearbeiten. Im Fall, dass ein Parkhaus offline ist, wird es bei der Parkplatzauswahl nicht berücksichtigt. Falls ein Broker offline ist, werden die Parkplätze nicht nach dessen Logik gereiht. Existiert nur ein einziger Broker im System können bei dessen Ausfall keine Reservierungen stattfinden.
In den unten aufgelisteten Fehlerszenarien verhält sich das System wie folgt:

- Ausfall eines Parkhauses
  Sowohl bei kurzen als auch bei längeren Ausfällen kann die Parkhaus-Domäne nicht auf Remote Events reagieren und folglich dem Broker keine Parkplätze zur Verfügung stellen. Das Parkhaus wird bei der Platzauswahl nicht berücksichtigt.
- Ausfall eines Brokers
  Beim Ausfall eines Brokers werden die Parkplätze nicht nach dessen Logik gereiht, d.h. in Systemen mit nur einem Broker können keine Reservierungen stattfinden.
- Ausfall des Servers
  Bei einem Serverausfall bricht das gesamte System zusammen.

4.1.3 WCF

Wie die vorhergehende Technologie braucht auch WCF eine zusätzliche Domäne: den Server. Die Serverdomäne ist für die Kommunikation und Synchronisation aller Domänen im System und für den Datenaustausch zwischen ihnen verantwortlich. Damit eine Domäne Teil des Systems sein kann, muss sie sich beim Server registrieren. Sobald die Registrierung erfolgreich stattgefunden hat, ist diese Domäne für alle Systemteilnehmer erreichbar. Dementsprechend kann sie auch auf alle anderen Domäne zugreifen.
Diese Variante des Parkplatzreservierungssystems wurde so entwickelt, dass sich alle Clients beim Server registrieren müssen. Bei Bedarf kann der Server, wenn einer der Clients es verlangt, die Methode *receive* von einem anderen Client aufrufen. Als Parameter hat diese Methode ein Objekt der *MyMessage*-Klasse, das alle notwendigen Informationen beinhaltet. Abbildung 11 erklärt den gesamten Ablauf des mit Hilfe von WCF aufgebauten Systems:

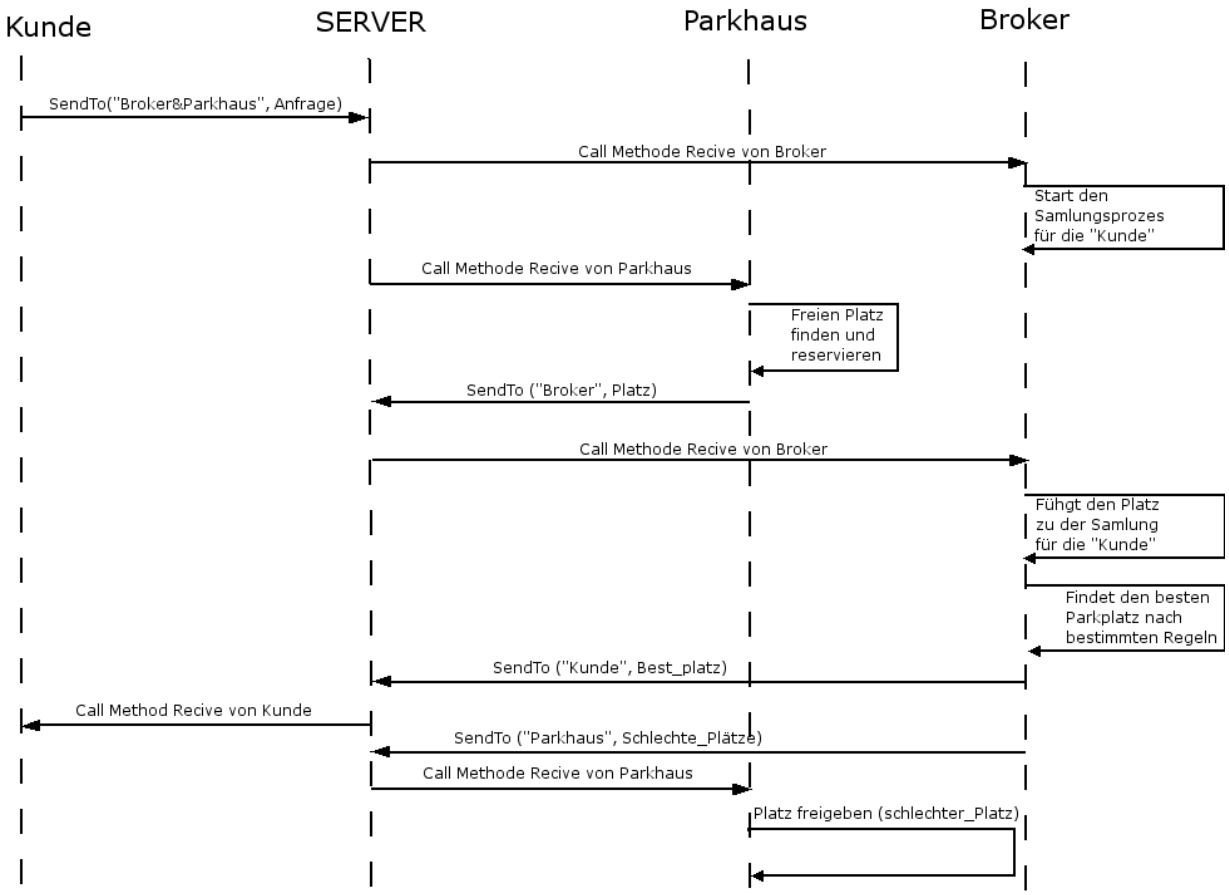

Abbildung 12: Sequenzdiagramm des „WCF-Parkplatzreservierungssystems“

Der Aufbau des Parkplatzreservierungssystems auf der Basis von WCF beinhaltet eine relativ einfache Logik: Der Kunde schickt eine Anfrage an den Server, der diese an den Broker und an alle Parkhäuser weiterleitet. Der Broker registriert die Anfrage in seiner inneren Datenstruktur und speichert alle Antworten der Parkhäuser bei der betreffenden Anfrage. Nachdem die Zeit abgelaufen ist, bestimmt der Broker den besten Platz und schickt ihn über den Server an den Kunden zurück. Für alle anderen, nicht ausgewählten Parkplätze stellt der Broker den Flag *isFree* auf „true“ und sendet die Information über den Server an das entsprechende Parkhaus. Die Parkhäuser stellen bei Erhalt einer solchen Meldung den Parkplatz wieder zur Verfügung. Bei weiteren Anfragen suchen die Parkhäuser wieder einen passenden Parkplatz aus und schicken ihn über den Server an den Broker.

Und so wird die Logik in der Praxis realisiert:

Zum Assembly *Server* gehört die Klasse *Programm*. Sie hat nur eine Mainmethode (siehe Listing 13) und dient ausschließlich zum Starten unseres Services auf dem Server-Rechner.

```
static void Main(string[] args)
{
    // Erzeugung des Services
    Uri uri = new Uri(ConfigurationManager.AppSettings["addr"]);
    ServiceHost host = new ServiceHost(typeof(Service.MyService), uri);
    // Starten des Services
    host.Open();
    Console.WriteLine("Spiel service listen on endpoint {0}", uri.ToString());
    Console.WriteLine("Press ENTER to stop Service service...");
    // Warten
    Console.ReadLine();
    // Schließen des Services
    host.Abort();
    host.Close();
}
```

Listing 13: Erzeugung und Starten der WCF-Services

Die Konfigurationsdatei *App.config* (siehe Listing 14), die sich in diesem Assembly befindet, benötigt man, um die Adresse des Services zu bestimmen.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="addr" value="net.tcp://localhost:22222/myservice" />
  </appSettings>
  <system.serviceModel>
    <services>
      <service name="Service.MyService" behaviorConfiguration="MyBehavior">
        <endpoint address=""
                  binding="netTcpBinding"
                  bindingConfiguration="DuplexBinding"
                  contract="Service.IEngine" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior  name="MyBehavior">
          <serviceThrottling maxConcurrentSessions="10000" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netTcpBinding>
        <binding name="DuplexBinding" sendTimeout="00:00:01">
          <reliableSession enabled="true" />
          <security mode="None" />
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Listing 14: Konfigurationsdatei *App.config*

Außerdem befinden sich in diesem Assembly die Hilfsinterfaces *IEngine* und *IEngineCallback*. Sie bestimmen die Endpoints auf der Server-Seite. Man muss dazusagen, dass das Interface *IEngine* die Endpoints mit normalem Contract bestimmt, d.h. es beschreibt die Methoden, die das Service zur Verfügung stellt. Das Interface *IEngineCallback* hingegen beschreibt die Endpoints mit Duplex-Contract. Es zeigt dem Service also, welche Methoden auf der Client-Seite zur Verfügung gestellt werden.

```
interface IEngine
{
    // Beschreibung der Methode für Anmeldung des Clients beim Server
    [OperationContract(IsOneWay = false, IsInitiating = true, IsTerminating =
false)]
    bool Join(string name);
    // Beschreibung der Methode für Senden der Nachricht an den Client
    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating =
false)]
```

```
    void SendMsg(string to, MyMessage msg);
    // Beschreibung der Methode für Abmeldung des Clients vom Server
    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating =
true)]
    void Leave();
}
```

Listing 15: Hilfsinterface *IEngine*

In Listing 15 ist zu sehen, dass die Methoden *Join*, *SendMsg*, *Leave* für die Clients erreichbar sind und auf unserem Service implementiert werden sollen. Das Attribut *OperationContract* beschreibt den Contract für jede Methode.

Der Endpoint mit Duplex-Contract wird in Listing 16 folgendermaßen beschrieben:

```
interface IEngineCallback
{
    // Beschreibung der Methode zum Empfangen der Nachricht vom Client
    [OperationContract(IsOneWay = true)]
    void Receive(string senderName, MyMessage message);
}
```

Listing 16: Hilfsinterface *IEngineCallback*

Hier wird die Methode *Receive* definiert, die auf Client-Seite implementiert wird und für das Service erreichbar sein soll. Außerdem wurde im *Server* Assembly die Hilfsklasse *EventArgs* zu *EngineEventArgs* erweitert. Sie hat folgende Instanzvariablen:

`public string name` – Name des Empfängers;

`public MyMessage message` – Nachricht für den Empfänger.

Die Hauptklasse des *Server* Assemblys ist die Klasse *MyService*, in der das Service realisiert ist: Das Attribut `[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession, ConcurrencyMode = ConcurrencyMode.Multiple)]` setzt den gesamten Verhältnismodus für das Service. In der Klasse gibt es folgende Instanzvariable:

`IEngineCallback callback = null` - Interface für den Endpoint mit Duplex-Contract;

`static Dictionary<string, EngineEventHandler> client = new Dictionary<string, EngineEventHandler>()` – Tabelle mit allen registrierten Clients;

`private EngineEventHandler myEventHandler` – "Listener" für ein Event;

`public delegate void EngineEventHandler(object sender, EngineEventArgs e)` – Delegate für die Methode, die das Event bearbeiten wird;

`public static event EngineEventHandler EngineEvent` – das Event;

Außerdem soll die Klasse *MyService* alle Methoden des Interfaces *IEngine* realisieren.

Die Methode *Join* wird wie folgt in Listing 17 realisiert:

```
public bool Join(string name)
{
    bool userAdded = false;
    // Erzeugung eines neuen Listeners für den Client
    myEventHandler = new EngineEventHandler(MyEventHandler);
    lock (syncObj)
    {
        // Wenn der Client in der Tabelle „client" nicht existiert,
        // wird er dort zusammen mit seinem Listener gespeichert
        if (!client.ContainsKey(name) && name != "" && name != null)
        {
            client.Add(name, MyEventHandler);
            userAdded = true;
        }
    }
    if (userAdded)
    {
        callback =
OperationContext.Current.GetCallbackChannel<IEngineCallback>();
        EngineEvent += myEventHandler;
    }
    return userAdded;
}
```

Listing 17: Implementierung der Methode *Join*

In Listing 17 registriert die Methode `Join` den Client beim Service. Sie bekommt als Parameter einen eindeutigen Client-Namen und erzeugt einen Event-Listener für den Client. Falls der Client unter diesem Namen noch nicht registriert ist, speichert die Methode den Listener in der Tabelle der registrierten Clients unter dem Client-Namen. So hat man auch in Zukunft wieder auf den Client Zugriff. Danach wird das *callback*-Interface für den Client definiert; es wird zu dem Event

EngineEvent subskribiert. Bei erfolgreicher Registrierung gibt die Methode „true“ zurück, andernfalls „false“.

```
public void SendMsg(string to, MyMessage msg)
{
    // Falls die Nachricht an Broker und Parkhäuser adressiert ist,
    // zuerst an Broker dann an Parkhäuser schicken
    if (to.Equals("BPH"))
    {
        this.SendMsg("B", msg);
        this.SendMsg("PH", msg);
    }
    // Erzeugung des Eventarguments
    EngineEventArgs e = new EngineEventArgs();
    e.name = this.name;
    e.message = msg;
    try
    {
        // Suchen in allen angemeldeten Clients
        foreach (KeyValuePair<string, EngineEventHandler> kvp in client)
        {
            EngineEventHandler sendTo;
            if (kvp.Key.StartsWith(to))
            {
// falls Client gefunden wird, aufrufen der Call-Back Methode vom Client
                string resiver = kvp.Key;
                lock (syncObj)
                {
                    sendTo = client[resiver];
                }
                sendTo.BeginInvoke(this, e, new AsyncCallback(EndAsync),
null);
            }
        }
    }
    catch (KeyNotFoundException)
    {
    }
}
```

Listing 18: Implementierung der Methode *SendMsg*

Wenn die Methode *SendMsg* (Listing 18) aufgerufen wird, wird mit Hilfe der Eingangsparameter *to* und *msg* ein Argument für das Event *EngineEvent* erzeugt. Danach wird der Empfänger in der Tabelle der Clients gesucht. Falls er gefunden wird, bekommt man über den Listener Zugriff auf den Client. Über die Methode *MyEventHandler* wird also eine Methode auf Client-Seite aufgerufen. (siehe Listing 19)

```
private void MyEventHandler(object sender, EngineEventArgs e)
{
    try
    {
        callback.Receive(e.name, e.message);
    }
    catch
    {
        Leave();
    }
}
```

Listing 19: Implementierung der Methode *MyEventHandler*

Man sollte erwähnen, dass diese Methode asynchron aufgerufen wird. Das Service kann also weiter arbeiten und muss nicht auf die Antwort des Clients warten. Wenn die Ausführung der Methode beendet wird, wird die Methode *EndAsync* aufgerufen. (siehe Listing 20)

```
private void EndAsync(IAsyncResult ar)
{
    EngineEventHandler d = null;
    try
    {
        System.Runtime.Remoting.Messaging.AsyncResult asres =
(System.Runtime.Remoting.Messaging.AsyncResult)ar;
        d = ((EngineEventHandler)asres.AsyncDelegate);
        d.EndInvoke(ar);
    }
    catch
    {
        EngineEvent -= d;
    }
}
```

Listing 20: Implementierung der Methode *EndAsync*

Der asynchrone Aufruf erlaubt es, die Performance des Systems zu steigern.

Die Methode *Leave* (Listing 21) wird bei Beendigung der Arbeit eines Clients aufgerufen. Der Client wird aus der Tabelle gelöscht und vom Event *EngineEvent* unsubscribed.

```
public void Leave()
{
    if (this.name == null)
        return;
        lock (syncObj)
        {
            // Löschen des Clients aus der Clients-Tabelle
            client.Remove(this.name);
        }
        // Abmelden des Listeners vom Event
```

```
            EngineEvent -= myEventHandler;
            this.name = null;
        }
```

Listing 21: Implementierung der Methode *Leave*

Um die Arbeit des Clients zu unterstützen, wurde im Assembly *SharedClients* die Definition von Remote Interfaces hinzugefügt. (siehe Listing 22)

```
public interface IEngine
{
    [System.ServiceModel.OperationContractAttribute(AsyncPattern = true,
Action = "http://tempuri.org/IEngine/Join", ReplyAction =
"http://tempuri.org/IEngine/JoinResponse")]
    System.IAsyncResult BeginJoin(string name, System.AsyncCallback callback,
object asyncState);
    bool EndJoin(System.IAsyncResult result);

    [System.ServiceModel.OperationContractAttribute(IsOneWay = true,
IsInitiating = false, Action = "http://tempuri.org/IEngine/Leave")]
    void Leave();

    [System.ServiceModel.OperationContractAttribute(IsOneWay = true,
IsInitiating = false, Action = "http://tempuri.org/IEngine/SendMsg")]
    void SendMsg(string to, MyMessage msg);
}
```

Listing 22: Interface *IEngine*

Die Methoden *Leave* und *SendMsg* werden synchron aufgerufen, die Methode *Join* hingegen asynchron. Deshalb hat letztere Methode eine andere Aufrufstruktur. Die Initialisierung passiert mit Hilfe der *BeginJoin*-Methode, bei Beendigung wird die Methode *EndJoin* aufgerufen.

Nach folgendem Prinzip wird das Interface *IEngineCallback* beschrieben. (siehe Listing 23)

```
public interface IEngineCallback
{
    [System.ServiceModel.OperationContractAttribute(IsOneWay = true, Action =
"http://tempuri.org/IEngine/Receive")]
    void Receive(string senderName, MyMessage message);
}
```

Listing 23: Interface *IEngineCallback*

Mit Hilfe des Interfaces *IEngineChannel,* siehe Listing 24, stellt man den Verbindungskanal zum Service ein:

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
"3.0.0.0")]
public interface IEngineChannel : IEngine, System.ServiceModel.IClientChannel
{
}
```

Listing 24: Interfaces *IEngineChannel*

Dieses Assembly enthält außerdem die Proxy-Klasse für das Service. (siehe Listing 25)

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
"3.0.0.0")]
public partial class EngineProxy :
System.ServiceModel.DuplexClientBase<IEngine>, IEngine
{
    public EngineProxy(System.ServiceModel.InstanceContext callbackInstance)
        : base(callbackInstance)
    {
    }
    public EngineProxy(System.ServiceModel.InstanceContext callbackInstance,
string endpointConfigurationName)
        : base(callbackInstance, endpointConfigurationName)
    {
    }
    public EngineProxy(System.ServiceModel.InstanceContext callbackInstance,
string endpointConfigurationName, string remoteAddress)
        : base(callbackInstance, endpointConfigurationName, remoteAddress)
    {
    }
    public EngineProxy(System.ServiceModel.InstanceContext callbackInstance,
string endpointConfigurationName, System.ServiceModel.EndpointAddress
remoteAddress)
        : base(callbackInstance, endpointConfigurationName, remoteAddress)
    {
    }
    public EngineProxy(System.ServiceModel.InstanceContext callbackInstance,
System.ServiceModel.Channels.Binding binding,
System.ServiceModel.EndpointAddress remoteAddress)
        : base(callbackInstance, binding, remoteAddress)
    {
    }
    public System.IAsyncResult BeginJoin(string name, System.AsyncCallback
callback, object asyncState)
    {
        return base.Channel.BeginJoin(name, callback, asyncState);
    }

    public bool EndJoin(System.IAsyncResult result)
    {
        return base.Channel.EndJoin(result);
    }

    public void Leave()
    {
        base.Channel.Leave();
    }

    public void SendMsg(string to, MyMessage msg)
    {
        base.Channel.SendMsg(to, msg);
    }
}
```

Listing 25: Service Proxy-Klasse *EngineProxy*

Am Anfang von Listing 25 stehen die verschiedenen Konstruktoren des Proxy-Objekts und die Implementierung der Methoden des Interfaces *IEngine*.

Auch in den Assemblys *Sender*, *Parkhaus* und *Broker* wurden einige Änderungen durchgeführt. Und zwar gibt es in jedem dieser Assemblys eine Konfigurationsdatei *App.config*, in der alle Attribute der Endpoints (Adresse, Bindung, Contract) beschrieben sind. (siehe Listing 26)

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name=""
                address="net.tcp://localhost:22222/myservice"
                binding="netTcpBinding"
                bindingConfiguration="DuplexBinding"
                contract="IEngine" />
    </client>
    <bindings>
      <netTcpBinding>
        <binding name="DuplexBinding" sendTimeout="00:00:05" >
          <reliableSession enabled="true" />
          <security mode="None" />
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Listing 26: Konfigurationsdatei *App.config*

Wie zu erwarten war, mussten auch in der Klasse *MsgForm* all dieser Assemblys Änderungen vorgenommen werden. Erstens musste das Interface *IEngineCallback* in die Klasse implementiert werden. Und zweitens gab es Änderungen im Programmablauf.
Nachdem die Domäne alle notwendigen Informationen vom Benutzer (ID, Name etc.) bekommt, wird das Proxy-Objekt *EngineProxy proxy* erzeugt:

```
InstanceContext site = new InstanceContext(this);
proxy = new EngineProxy(site);
```

Dann muss sich der Client beim Service registrieren. Dafür ist ein eindeutiger Name notwendig. Der Name besteht aus der Abkürzung für die Domäne (S – Sender, PH – Parkhaus, B – Broker) und einer ID-Nummer.

```
IAsyncResult iar = proxy.BeginJoin("S"+myID, new AsyncCallback(OnEndJoin),
null);
```

Die Registrierung findet asynchron statt, am Ende des Registriervorgangs wird die Methode *OnEndJoin* (siehe Listing 27) aufgerufen.

```
private void OnEndJoin(IAsyncResult iar)
{
    try
    {
        bool result = proxy.EndJoin(iar);
        HandleEndJoin(result);
    }
    catch (Exception e)
    {
        HandleEndJoinError();
    }
}
```

Listing 27: Implementierung der Methode *OnEndJoin*

Folgende Methoden werden je nach Ergebnis der obigen Methoden aufgerufen:

HandleEndJoinError – falls beim Registrieren ein Fehler aufgetreten ist;

HandleEndJoin – falls die Registrierung erfolgreich war.

Das Interface *IEngineCallback* enthält die Methode *Receive*. (siehe Listing 28)

```
public void Receive(string senderName, MyMessage message)
{
    // Prüfen ob die Nachricht vom Broker kommt
    if (senderName.StartsWith("B") && message.KundeID == myID)
    {
        // Zugriff auf GUI-Element herstellen
        if (textBox1.InvokeRequired)
        {
            textBox1.Invoke(new HandleFormElrmrntDelegate(setWert), new
object[] { message });
        }
        else
        {
            //Ausgabe der empfangenen Nachricht im GUI
            setValue(message);
        }
    }
}
```

Listing 28: Implementierung der Methode *Receive*

Diese Methode wird beim Duplex-Contract verwendet und bei Anfragen von anderen Clients über das Service aufgerufen. Bei uns ist dies dann der Fall, wenn ein Client eine Nachricht bekommt. Die beim Client eintreffende Nachricht wird mit Hilfe der Methode *setValue* im GUI ausgegeben und entsprechend bearbeitet.

Um eine Nachricht von einer Domäne an die andere zu schicken, wird die Methode *sendMsg* verwendet. (siehe Listing 29)

```
private void sendMsg(MyMessage message)
{
    try
    {
        // ID setzen
        message.CusomerID = this.myID;
        // Nachricht an Broker und Parkhäuser senden
        proxy.SendMsg("B"+"PH", message);
    }
    catch
    {
        AbortProxyAndUpdateUI();
        Error("Error: Connection to Engine server lost!");
    }
}
```

Listing 29: Implementierung der Methode *sendMsg*

Listing 29 zum Beispiel ist aus der Kunden-Domäne. Nachdem die Anfrage erzeugt wurde, sendet die Kunden-Domäne das *MyMessage*-Objekt an den Broker und dann an alle Parkhäuser. Wenn der Client aufhört zu arbeiten, werden folgende Methoden in nachstehender Reihenfolge aufgerufen:
proxy.Leave(),proxy.Abort() und *proxy.Close()*.
In den unten aufgelisteten Fehlerszenarien verhält sich das System wie folgt:

- Ausfall eines Parkhauses
  Sowohl bei kurzen als auch bei längeren Ausfällen wird das Parkhaus aus der Tabelle aller registrierter Clients entfernt. Das Parkhaus wird bei der Platzauswahl nicht berücksichtigt.
- Ausfall eines Brokers
  Wie auch bei Parkhausausfällen wird beim Ausfall eines Brokers dieser aus der Tabelle aller registrierter Clients entfernt, d.h. die Parkplätze werden nicht nach seiner Logik gereiht. In Systemen mit nur einem Broker können keine Reservierungen stattfinden.

- Ausfall des Servers
  Bei einem Serverausfall bricht das gesamte System zusammen.

4.1.4 ASP.NET

Grundlage für den Aufbau unseres Beispielsystems ist die Entwicklung eines Web-Services mittels ASP.NET und der Zugriff auf das Service durch die Domäne (Remote Clients). Die Server-Domäne ist in diesem Fall der Web-Server, auf dem die Broker-Domäne als Web-Dienst eingerichtet ist und auf Basis des SOAP-Protokolls läuft.
Als Web-Server, der ASP.NET 2.0 unterstützt, ist ein Microsoft Internet Information Server (IIS) Version 5 oder höher notwendig. Bei der Entwicklung des Web-Dienstes wurde Microsoft VisualStudio verwendet. Dieses erlaubt es, den entwickelten Dienst automatisch auf demWeb-Server zu installieren und an der richtigen Stelle des Serververzeichnisses zu platzieren. Diese Eigenschaft von VisualStudio ist sehr praktisch, da man die Einstellungen nicht mehr händisch zu machen braucht. Um einen Web-Dienst zu erzeugen genügt es, in Visual Studio 2005 aus den angebotenen Projekten den Punkt „new Web-Service“ auszuwählen und in der erzeugten Klasse alle notwendigen Web-Methoden zu definieren. Alle anderen Einstellungen werden von VisualStudio und IIS selbst vorgenommen. Nachdem der Web-Dienst definiert wurde, kann man VisualStudio bei Bedarf mit einem Dienst verbinden und automatisch ein Proxy-Objekt generieren. So können Client-Teile entwickelt werden, ohne den vollständigen Code des Web-Dienstes zu kennen (siehe Abbildung 13), sondern nur dessen Beschreibung (Dokumentation).

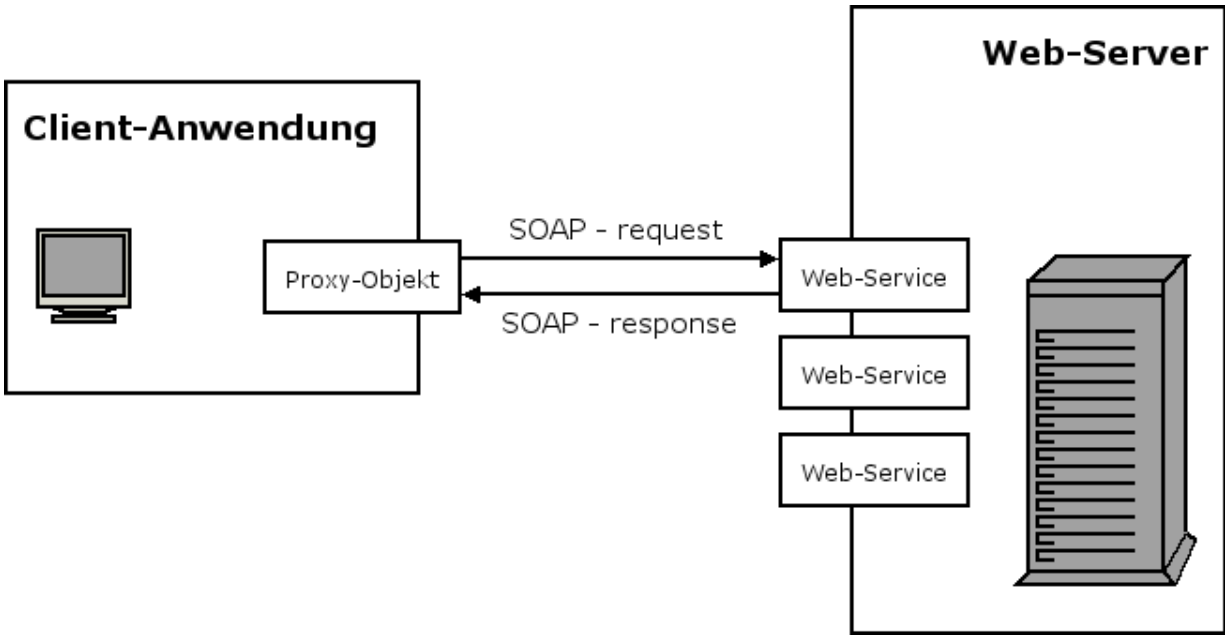

Abbildung 13: Architektur von ASP.NET

Trotz der Simplizität der Arbeit mit und der leichten Handhabbarkeit von Web-Diensten mit ASP.NET, entspricht die Technologie leider nicht genau einer eventbasierten Architektur. Am häufigsten wird ASP.NET zur Entwicklung von serviceorientierten Architekturen benutzt. Die größte Effektivität erlangt man bei der Projektierung einer Web-Anwendung mittels ASP.NET. Darüber hinaus wird ASP.NET sehr häufig zusammen mit ADO.NET[11] benutzt. So erhält man eine zuverlässige Verbindung zu einer Datenbank.
In diesem Beispiel kommt keine Datenbank zum Einsatz, da dies Komplexität im System erzeugt hätte (Projektierung einer Datenbank, Verbindung, Synchronisierung usw.). Ein Ziel der Entwicklung des Parkplatzreservierungssystems mit Hilfe von ASP.NET ist der Aufbau eines Verteilten Systems und nicht die Benutzung einer Datenquelle. Deshalb werden statt einer klassischen Datenbank einfache Statische Klassen benutzt, die bei Bedarf gegen eine echte Datenbank ausgetauscht werden können.
Da in ASP.NET Remote Events, Notifikationen etc. wegfallen, wurde der gesamte Programmablauf auf eine Servicebasierte Architektur umgestellt.
Mit der ASP.NET Technologie und basierend auf SOA sieht unser Parkplatzreservierungssystem folgendermaßen aus:

[11] http://msdn.microsoft.com/de-de/library/bb979090.aspx (letzter Zugriff am 20.08.09)

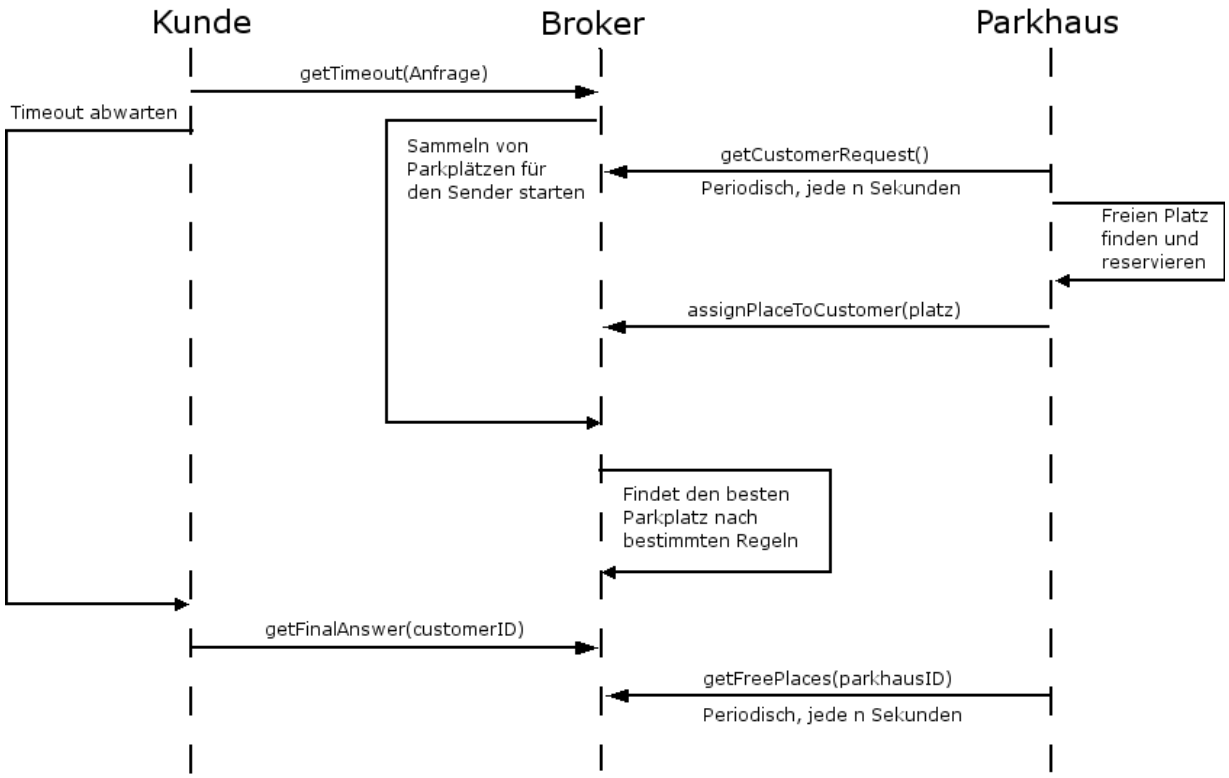

Abbildung 14: Sequenzdiagramm des „ASP.NET-Parkplatzreservierungssystems“

Der Kunde stellt eine Verbindung mit dem Web-Dienst (Broker) her und bekommt einen Timeout-Wert. Dann startet er seinen Timer und sendet eine Anfrage an den Broker. Sobald der Broker die Anfrage bekommt, speichert er sie in der Liste der aktiven Anfragen (in der Liste befinden sich alle Anfrage, die zu diesem Zeitpunkt noch keine Antwort, also noch keinen besten Platz, bekommen haben). Dann sammelt der Broker zu dieser Anfrage alle passenden Parkplätze, die er von den Parkhäusern bekommt. Wenn der Timer des Kunden (=Senders) ausläuft, fordert er vom Broker den besten Parkplatz (der Broker wählt diesen aus seiner Liste aus). Außerdem muss der Broker alle anderen Plätze, die dieser Anfrage zugeordnet sind, als „frei“ markieren und in einer Tabelle mit ausschließlich freien Plätzen speichern. Das Parkhaus muss den Broker in einem bestimmten Zeitintervall abfragen (z.B. jede Sekunde), erstens ob eine neue Anfrage eingetroffen ist, und zweitens ob es im Parkhaus wieder freie Plätze gibt. Falls eine neue Anfrage vorhanden ist, wird das Parkhaus einen passenden Platz suchen, für den Kunden reservieren und an den Broker schicken. Wenn der Broker den Platz rechtzeitig bekommt, d.h. das Timeout noch nicht abgelaufen ist, wird der Platz der entsprechenden Anfrage zugeordnet. Andernfalls wird der

Platz sofort wieder als frei markiert und in der Tabelle mit den freien Plätzen gespeichert. Falls ein Parkhaus in dieser Tabelle einen Platz findet, der ihm „gehört“, markiert es diesen Platz bei sich als frei.
Die Besonderheit dieser Entwicklungsvariante besteht darin, dass die Domänen keine Möglichkeit haben, auf Events zu reagieren. Sie müssen den Web-Dienst selber in einem bestimmten Zeitintervall abfragen, ob ein Event stattgefunden hat. Erst dann können sie entsprechend darauf reagieren.
Eine wichtige Rolle in dieser Implementierung spielt der Web-Dienst *PRService*. Er beinhaltet in seinem Assembly die Klasse *Service*, die alle Web-Methoden dieses Dienstes beschreibt.

```
[WebMethod]
public int getTimout(MyMessage message)
{
  //Nachricht in der Liste der aktiven Anfragen speichern
    Broker.CustomerRequestList.Add(message);
  //Erzeugen eines Datenhalters für alle diese Anfrage betreffenden Parkplätze
    Broker.CustomerPlaces.Add(message.CustomerID, new List<MyMessage>());
  //Gibt das Timeout an den Kunden zurück
    return Broker.Timout;
}
```

Listing 30: Web-Methode *getTimout*

Mittels der Methode in Listing 30 bekommt der Web-Dienst eine Anfrage vom Kunden und speichert diese in der Liste der aktiven Anfragen. Dann ordnet er dem Kunden eine neue leere Liste für die von den Parkhäusern eintreffenden Parkplätze zu. Der Kunde bekommt das Timeout des Brokers zurück.

```
[WebMethod]
public MyMessage getFinalAnswer(int aCustomerID)
{
    //Auswahl des besten Parkpatzes für den Kunden
    MyMessage best = Broker.getBestPlace(aCustomerID);
    MyMessage msg= null;
    //Holt Datenhalter mit allen passenden Parkplätzen für diese Anfrage
    foreach (MyMessage mm in Broker.CustomerRequestList)
        if (mm.CustomerID==aCustomerID)
            msg = mm;
    //Entfernt alle passenden Parkplätze aus der Liste
```

```
    Broker.CustomerRequestList.Remove(msg);
    //Löscht Nachricht aus der Liste der aktiven Anfragen
    Broker.CustomerPlaces.Remove(aCustomerID);
  //Gibt den besten Parkpatz an den Kunden zurück
    return best;
}
```

Listing 31: Web-Methode *getFinalAnswer*

Die in Listing 31 dargestellte Web-Methode wird vom Kunden nach Ablauf der Timeout-Zeit aufgerufen. Dann bestimmt der Broker den besten Platz, löscht die Anfrage aus der Liste der aktiven Anfragen und entfernt die Liste der Parkplätze, die für diesen Kunden „gesammelt“ wurden.

```
[WebMethod]
public List<MyMessage> getCustomerRequest ()
{
    //Gibt die Liste der aktiven Anfragen an das Parkhaus zurück
    return Broker.CustomerRequestList;
}
```

Listing 32: Web-Methode *getCustomerRequest*

Die Methode *getCustomerRequest* (siehe Listing 32) übermittelt dem Parkhaus die Liste der aktiven Anfragen. (Das Parkhaus braucht diese Liste, um zu wissen, welche Anfragen neu sind.)

```
[WebMethod]
public List<MyMessage> getFreePlaces(int aParkingID)
{
    //Erzeugt eine Liste für Parkplätze, die das Parkhaus mit ID = aParkingID
    //wieder freigeben muss
    List<MyMessage> freePlaces = new List<MyMessage>();
    //Sucht diese Parkplätze in der gemeinsamen Liste und sammelt
    //sie in neuer Liste
    foreach (MyMessage mm in Broker.FreePlaces)
        if (mm. ParkingID == aParkingID)
        {
            freePlaces.Add(mm);
            //Löscht die Plätze, die vom Parkhaus wieder freigegeben werden
            //sollen, aus der gemeinsamen Liste
            Broker.FreePlaces.Remove(mm);
        }
    //Gibt Liste mit Parkplätzen, die das Parkhaus wieder freigeben muss,
    //zurück
    return freePlaces;
}
```

Listing 33: Web-Methode *getFreePlaces*

Die Methode *getFreePlaces* in Listing 33 übermittelt dem Parkhaus die Liste mit Plätzen, die das Parkhaus wieder freigeben muss.

```
[WebMethod]
public void assignPlaceToCustomer(MyMessage message)
{
    //Falls die Liste mit den passenden Parkplätze für diesen Kunden noch
    //existiert
    if (Broker.CustomerPlaces.ContainsKey(message.CustomerID))
    {
        //Wird dieser Parkplatz hinzugefügt
        Broker.CustomerPlaces[message.CustomerID].Add(message);
    }
    else
    {
        //sonst muss er wieder freigegeben werden
        message.IsFree = true;
        Broker.FreePlaces.Add(message);
    }
}
```

Listing 34: Web-Methode *assignPlaceToCustomer*

Mit Hilfe der in Listing 34 abgebildeten Methode ordnet das Parkhaus einen reservierten Parkplatz zu einer Kundenanfrage zu. Dies geschieht in der Tabelle *CustomerPlaces*, in der der Broker alle passenden Parkplätze für die betreffende Anfrage sammelt. Falls die Anfrage jedoch nicht mehr aktuell ist und der Kunde bereits einen Platz bekommen hat, wird der Platz sofort in die Tabelle mit den freien Plätzen hinzufügt.

Wie bereits erwähnt, ist die Klasse Broker jetzt eine statische Klasse, die außerdem zwei Hilfslisten bekommt:

`private static List<MyMessage> customerRequestList` –Liste der aktiven Anfragen;

`private static List<MyMessage> freePlaces` – Liste mit freien Parkplätzen;

Des Weiteren wurde die Methode *getFreePlaces* verändert. Nun werden die Parkplätze, die nicht ausgewählt und als „frei“ markiert wurden, in die Liste mit den freien Parkplätzen gestellt. Die Parametereingabe läuft somit mit Hilfe der Methode *getParam* über eine XML-Konfigurationsdatei (siehe Listing 35).

```
private static int id=getParam("ID");

private static int getParam(string aParamName)
    {
        XmlDocument document = new XmlDocument();
        try{
            document.Load("Property.xml");
            string xpathExpression = "/broker_prop/"+aParamName;
            XmlNode node = document.SelectSingleNode(xpathExpression);
            int value = int.Parse(node.InnerText);
            return value;
        }catch(Exception ee){
            Console.WriteLine("XML Error!!! " + ee.ToString() );
            return -1;
        }
    }
```

Listing 35: Methode *getParam*

Alle anderen Klassen haben kaum Veränderungen erfahren:

- Neue Liste in der Klasse *MsgForm* der Parkhaus-Domäne, in der alle KundenIDs gespeichert werden, für die das Parkhaus bereits einen Platz reserviert hat:

  ```
  private List<int> sendCustomer.
  ```

- Timer-Objekt in der Kunden-Domäne als auch in der Parkhaus-Domäne:

  ```
  private System.Windows.Forms.Timer timer
  ```

  Weiters wurde folgendes Ereignis hinzugefügt:

  ```
  timer.Tick += new EventHandler(timer_Tick);
  ```

- Eigenes Timerintervall für jede Domäne (außer für die Kunden-Domäne); in der Parkhaus-Domäne wurde es z.B. auf 1 Sekunde festgelegt:

  ```
  timer.Interval = 1000;
  timer.Enabled = true;
  ```

- Für die Kunden-Domäne muss dieses Intervall vom Broker geholt werden:

  ```
  timer.Interval = server.getBearbeitungsZeit(message);
  timer.Enabled = true;
  timer.Tick += new EventHandler(timer_Tick);
  ```

Wenn das Timeout im Objekt *server* (= Proxy-Objekt des Web-Dienstes) abläuft, wird die Methode *timer_Tick* (siehe Listing 36) aufgerufen:

```
void timer_Tick(object sender, EventArgs e)
        {
            message = server.getFinalAntwort(myID);
            setWert();
            timer.Enabled = false;
            timer.Tick -= new EventHandler(timer_Tick);
        }
```

Listing 36: Methode Sender.*timer_Tick*

Der Kunde ruft die Methode *getFinalAnswer* vom Web-Dienst auf und bekommt den besten Platz zugesandt. Danach wird der Platz im GUI ausgegeben.
Der gleiche Event wird in der Parkhaus-Domäne auf andere Weise bearbeiten. (siehe Listing 37)

```
void timer_Tick(object sender, EventArgs e)
{
    //Holt vom Broker die Liste mit den aktiven Anfragen
    MyMessage[] customerList = server.getCustomerRequest();
    if (customerList != null)
    {
        //Die Liste wird durchgegangen
        for (int i = 0; i < customerList.Length; i++)
        {
            //Falls für eine Anfrage noch kein Parkplatz reserviert wurde,
            //wird dies gemacht
            if (!sendCustomer.Contains(customerList [i].CustomerID))
            {
                setValue(customerList[i]);
            }
        }
    }
    //Holt vom Broker die Liste mit den Parkplätzen, die wieder freigegeben
    //werden müssen
    MyMessage[] freePlacesList = server.getFreePlaces(myID);
    if (freePlacesList!= null)
    {
        for (int i = 0; i < freePlacesList.Length; i++)
        {
            //Löscht die Anfrage, für die in einem anderen Parkhaus ein
            //besserer Parkplatz gefunden wurde, aus der Liste der
            //Kundenanfragen
            sendCustomer.Remove(freePlacesList[i].CustomerID);
            //Markiert Parkplatz als "frei"
            parking.Places[freePlacesList[i].ParkingPlaceID].IsFree = true;
        }
    }
}
```

Listing 37: Methode *Parking.timer_Tick*

Die Parkhaus-Domäne fragt den Web-Dienst jede Sekunde nach den aktuellen Kundenanfragen ab, was Ineffizienz bei der Entwicklung und beim Deployment der Domäne verursacht. Außerdem kann es zu einer Netztrafik-Überlastung führen. Falls sich unter den Kundenanfragen eine neue Anfrage befindet, wird diese mit der Methode *setValue* bearbeitet. Danach wird der Web-Dienst nach Plätzen gefragt, die das Parkhaus wieder auf „frei“ setzen muss.
Die Methode *setValue* reserviert einen passenden Platz für den Kunden und gibt diesen Platz an den Dienst weiter. Dann wird die ID des Kunden in einer Liste gespeichert, um bei der nächsten Abfrage des Web-Dienstes sicherzustellen, dass diese Anfrage bereits bearbeitet wurde (und für das Parkhaus demnach nicht neu ist):

```
server.assignPlaceToCustomer(message);
sendCustomer.Add(message.KundeID);
```

Ein solcher Programmablauf macht das in der ASP.NET-Technologie entwickelte Parkplatzreservierungssystem funktionsfähig. Dennoch hat diese Variante viele Nachteile. Dazu zählt etwa das Entstehen einer großen Datenübertragungsrate zwischen den Domäne, was die Performance des Systems negativ beeinflusst.
In den unten aufgelisteten Fehlerszenarien verhält sich das System wie folgt:

- Ausfall eines Parkhauses
  In dieser Version des Beispielsystems kann der Broker nicht feststellen, ob ein Parkhaus online ist. Beim Ausfall eines Parkhauses kann er einen Parkplatz reservieren, die Information über die Reservierung wird das Parkhaus aber nicht erreichen.
- Ausfall eines Brokers
  Da die gesamte Kommunikation über einen Broker abläuft, der im Beispiel als Web-Dienst realisiert ist, bricht bei dessen Ausfall das ganze System zusammen.
- Ausfall des Servers
  Bei einem Serverausfall bricht das gesamte System zusammen.

5 Evaluierung und Analyse der erstellten Software

Auf Basis der vorhergehenden Kapitel können nun die Besonderheiten der Technologien besprochen und die einzelnen Technologien verglichen werden. Ziel dieses Kapitels ist es, den Vor- und Nachteilen jeder Technologie auf den Grund zu gehen und die Erfahrungen bei der Systementwicklung aufzuzeigen.

Als Grundlage für die Ausarbeitung der Vergleichskriterien [24] und für den Vergleich der Technologien diente der Fragebogen „Einsatz von Middleware-Technologien“, den Ayse Cicek mit meiner Hilfe im Rahmen ihre Praktikumsarbeit „*The use of middleware technologies*“ [21] entwickelt hat. (siehe Anhang 3) Dieser Fragebogen ist ein Versuch, Middleware-Produkte für Verteilte Systeme aus Sicht der Programmierer her bewerten zu lassen. Die Evaluierung der Technologien basiert außerdem auf dem Artikel „How do J2EE and Microsoft's .Net compare in enterprise environments?“ [8] und einigen anderen Quellen [25], [26], [27], [28]. Mit Hilfe der Programmierungsbeschreibungen (siehe Kapitel 4) wurden die Technologien anhand folgenden Kriterienkatalogs verglichen:

- Notifikationen;
- Synchronisation;
- Plattformunabhängigkeit;
- Verbindungsaufbau;
- Skalierbarkeit;
- Evolution und Wartbarkeit;
- Komplexität und Transparenz der Technologie;
- Ausfallssicherheit;
- Installation und Konfiguration der Plattform;
- Lines of Code;

Diese Kriterien stellen eine Möglichkeit dar, die vorliegenden Technologien zu bewerten. Darüber hinaus können die Kriterien als Grundlage für den Vergleich der Technologien dienen. Zur besseren Differenzierung der Bewertung der Technologien verwende ich folgende Skala[12]:

- 1 (++) – sehr gut: Diese Note wird vergeben, wenn das betreffende Kriterium von der Middleware erfüllt wird und **kein zusätzlicher Aufwand** benötigt wird, um den

[12] Diese Skala ist nicht dazu geeignet, ein Kriterium wie „Lines of Code“ zu bewerten.

erwünschten Systemzustand bzw. die erwünschte Konfiguration zu erreichen. D.h. nur mit Hilfe der von der Middleware zur Verfügung gestellten Klassen und Methoden.

- 2 (+) – gut: Wird vergeben, wenn das betreffende Kriterium in der Middleware zwar erfüllt wird, aber mit **einem kleinen zusätzlichen Aufwand** zu rechnen ist. Z. B. müssen zusätzliche Klassen oder Interfaces implementiert werden.
- 3(~) – mittelmäßig: Wird vergeben, wenn das betreffende Kriterium in den Technologien nicht vorgesehen ist und keine Standardmechanismen oder -werkzeuge existieren, um dieses Kriterium zu erfüllen. Die Technologie bietet aber die Möglichkeit, das Problem mit Hilfe einer nicht standardisierten Lösung zu lösen. Um dies zu erreichen ist **ein großer zusätzlicher Aufwand** zu erwarten.
- 4 (-) – schlecht: Wird vergeben, wenn in einer Technologie nicht nur keine Standardmethoden zum Erreichen des erwünschten Zustands vorgesehen sind, sondern alle Versuche, diesen Zustand herzustellen, zu instabiler Systemarbeit oder einem kompletten Umbau der Systemarchitektur führen. Dies bedeutet einen **sehr großen zusätzlichen Aufwand**.
- 5(--) – sehr schlecht: Wird vergeben, wenn ein Kriterium von der Middleware **unter keinen Umständen erfüllt werden kann.**

Da nun alle Kriterien und die verwendete Bewertungsskala beschrieben sind, kann mit Hilfe der Programmierungsbeschreibungen nun die Analyse der einzelnen Technologien durchführt werden und jede Technologie einer Bewertung unterzogen werden.

5.1 XVSM

Obwohl diese Middleware relativ neu ist und bei den Entwicklern noch wenig Popularität besitzt, ist sie gut aufgebaut und sehr leicht in der Verwendung. Dies und das hohe Abstraktionsniveau der Middleware positionieren diese Technologie im Bereich der eventbasierten Architekturen. Die Architektur des mit Hilfe von XVSM entwickelten Systems zeichnet sich durch seine Einfachheit und hohe Verständlichkeit aus. Das erlaubt Entwicklern ohne große Erfahrungen im Bereich der Verteilten Systeme, in kürzester Zeit die Grundlagen der Technologie zu beherrschen und funktionsfähige Systeme zu entwickeln. Die Einfachheit in der Verwendung und die Existenz eines guten Tutorials [23] kompensieren das Fehlen weit verbreiteter Fachliteratur.

Im Folgenden gehe ich auf jedes der oben erwähnten Kriterien in Bezug auf die XVSM-Technologie ein:

Notifikation – Benachrichtigungen [13] sind einer der Vorteile der XVSM-Technologie. Der Benachrichtigungsmechanismus ist detailliert aufgebaut und bietet dem/der Entwickler/in großes Potenzial beim Aufbau von Systemen mit eventbasierter Architektur.

Plattformunabhängigkeit – Die Technologie ist Plattformunabhängig und stellt die Möglichkeit zur Verfügung, Domänen auf unterschiedlichen Plattformen zu entwickeln. Es gibt Java und .NET Implementierungen von XVSM, die mittels XML miteinander kommunizieren können.

Verbindungsaufbau – Verlangt keine zusätzliche Eingabe von Parametern. Ports werden automatisch geöffnet. Außerdem unterstützt die Technologie das Öffnen von freien Ports im Bereich von 8000 bis 9000. Um die Verbindung zu einem Container aufzubauen, braucht man nur dessen Adresse.

Synchronisation - Ist auch ein Vorteil von XVSM. Die Technologie unterstützt das Shared Data Konzept voll. Jede mit dem Space verbundene Domäne hat Zugang zu den Daten in den Containern. Die Manipulationen der Daten werden mit Hilfe des Transaktionsmechanismus gesteuert, XVSM garantiert also den konsistenten Zustand des Containerinhalts.

Skalierbarkeit – Ist bei diesem System gut, besonders im Vergleich zu XVSM für Java [26]. Die Benachrichtigungen werden schnell übertragen, und mit dem Anwachsen der beteiligten Domänen steigen die benötigen Ressourcen nicht wesentlich an.

Evolution und Wartbarkeit – Aufgrund der Besonderheiten im Systemaufbau (Fehlen der Server-Domäne) können Veränderungen im Kommunikationsbereich des Systems zu Änderungen in allen Domänen des betroffenen Systembereichs führen. Wenn z. B. die Kommunikationsreihenfolge in der Parkhaus-Domäne verändert wird, muss man in jedem Parkhaus Änderungen durchführen, im Gegensatz zum in WCF programmierten Systembeispiel, wo es genügt, nur die Server-Domäne zu ändern. Andererseits vereint XVSM sehr viele Software

Architektur Stile (pub/sub, database...) in einem API (engl. *application programming interface*), wodurch sich Änderungen in der Middleware nicht bis in die Applikationsebene durchschlagen.

Komplexität und Transparenz der Technologie – Wie bereits erwähnt, hat die Technologie ein hohes Abstraktionsniveau. Da die meisten Mechanismen vor dem Entwickler „versteckt“ sind, ist die Arbeit der Middleware im Detail nicht so leicht nachvollziehbar.

Ausfallssicherheit – XVSM bietet eine hohe Ausfallsicherheit, da es eine P2P-Technologie ist [WR 6], und der Ausfall einer der Domänen keine Gefahr für die Existenz des ganzen Systems darstellen würde.

Installation und Konfiguration der Plattform – Diese Middleware ist keine Standardkomponente von Windows. Ihre Installation ist jedoch sehr einfach und verlangt lediglich das Kopieren der DLL-Datei auf den Rechner und das Einstellen des Pfads zu den Dateien im Workflow [23].

Lines of Code - 118

In Tabelle 1 sieht man die von mir vorgenommene Bewertung der Kriterien:

	++	+	~	-	--
Kriterien	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Notifikation	X				
Plattformunabhängigkeit	X				
Synchronisation	X				
Verbindungsaufbau	X				
Skalierbarkeit		X			
Evolution und Wartbarkeit			X		
Komplexität und Transparenz der Technologie		X			
Ausfallssicherheit	X				
Installation und Konfiguration der Plattform		X			

Tabelle 1: Bewertung der XVSM-Technologie

5.2 .NET Remoting

Diese Middleware verlangt es, das Verhalten des Systems auf relativ niedrigem Niveau zu definieren. Das erklärt, warum so viele verschiedene Mechanismen direkt vom Entwickler beschrieben werden müssen. Außerdem muss man, um die Eigenschaften von eventbasierter Architektur zu erreichen, komplizierte Objekte erzeugen. Für die .NET Remoting-Technologie können unsere Vergleichskriterien folgendermaßen beschrieben werden:

Notifikation - Benachrichtigungen sind keine Standardeigenschaft dieser Technologie, aber der Benachrichtigungsmechanismus kann vom Entwickler eingebaut werden, indem er die Objekte auf Remote Events reagieren lässt.

Plattformunabhängigkeit – Wie die meisten Technologien der .NET-Welt kann .NET Remoting auf einer Windows-Plattform installiert werden.

Verbindungsaufbau – Aufgrund des niedrigen Entwicklungsniveaus der Technologie liegt der Verbindungsaufbau in der Verantwortung des Entwicklers. Der Prozess der Erzeugung eines Verbindungsaufbaus mit dieser Technologie ist leider nicht ganz einfach. Um eine Verbindung einzurichten, muss man Name und Nummer des Ports eingeben und dann Verbindungskanal sowie -modus für das Remote Object definieren. Dies gibt die Möglichkeit, die optimale Verbindung für das System herzustellen.

Synchronisation - Die Technologie hat keinen eingebauten Synchronisationsmechanismus. Der Entwickler hat die Möglichkeit, ihn selber hinzuzufügen. Und zwar indem er einen Synchronisations-Algorithmus (z.B. einen Semaphor-Algorithmus[13]) für den Zugriff auf Daten einbaut, die für einen gemeinsamen Zweck vorgesehen sind.

Skalierbarkeit – Die Middeware verfügt über hohes Potential, eine gute Skalierbarkeit zu erreichen. Dies verlangt von dem/der Entwickler/in jedoch zusätzliche Programmierarbeit: die

[13] siehe http://de.wikipedia.org/wiki/Semaphor_(Informatik) (letzter Zugriff am 20.08.09)

Implementierung eines Suchservers (Discovery Server) und eines Proxy-Servers; außerdem werden mehrere Server Domänen benötigt [6].
Evolution und Wartbarkeit – Die Anwesenheit der Server-Domäne erlaubt es, die Änderungen im Notifikationsablauf nur im Serverteil vorzunehmen. Die Clientteile können unverändert bleiben.

Komplexität und Transparenz der Technologie – Die Verständlichkeit der Technologie ist gut, da man aber beim Entwickeln eine große Menge zusätzlicher Klassen und Interfaces braucht, steigt die Komplexität. Im Gegensatz zu XVSM benötigen die verteilten Komponenten der Applikation eine detaillierte Beschreibung und Implementierung (siehe Listing 6 und Listing 10).

Ausfallssicherheit – Leider muss der Entwickler diesen Teil des Systems selber ausarbeiten und mit Hilfe der Standardwerkzeuge von .NET implementieren.

Installation und Konfiguration der Plattform – .NET Remoting ist Teil der Standard-Windows-Komponenten, es muss keine zusätzliche Installation durchgeführt werden.

Lines of Code 157

Die Bewertung der .NET Remoting Technologie ist in Tabelle 2 zu sehen:

	++	+	~	-	--
Kriterien	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Notifikation			X		
Plattformunabhängigkeit:					X
Synchronisation			X		
Verbindungsaufbau			X		
Skalierbarkeit			X		
Evolution und Wartbarkeit		X			
Komplexität und Transparenz der Technologie			X		
Ausfallssicherheit			X		
Installation und Konfiguration der Plattform	X				

Tabelle 2: Bewertung der .NET Remoting-Technologie

5.3 WCF (Windows Communication Foundation)

Das Prinzip von WCF ist sehr ähnlich wie das der anderen Verteilten Technologien:
Eine Klasse, die ein Remote Object beinhaltet, muss ein Remote Interface implementieren. Dieses Interface steht für die Client-Klasse zur Verfügung und wird in die Proxy-Klasse implementiert. Der Client kommuniziert mit dem Server über das Proxy. WCF ist sehr abstrakt, der/die Entwickler/in muss sich folglich um viele Dinge nicht selber kümmern (z.B. das Öffnen und Beschreiben von Kanälen, das Öffnen von Ports usw.). Alles läuft über die „Endpoints". Diese registriert man am Server und bekommt dann über das HTTP-Protokoll Zugriff auf sie. Das Registrieren von Endpoints läuft über die Config-Datei, wodurch das kompilierte Programm nicht beeinflusst wird. Diese Eigenschaft macht die WCF-Technologie dynamisch und flexibel. Sowohl Client als auch Server sind leicht zu installieren und auf verschiedenen Rechnern einzustellen.

Notifikation – Ist keine Standardeigenschaft dieser Technologie. Da die Middleware aber End-Points mit Duplex-Contract unterstützt, kann der Benachrichtigungsmechanismus leicht nachgebaut werden. (Will der Server etwa einen Client benachrichtigen, kann er die im Duplex-Contract beschriebene Methode direkt aufrufen).

Plattformunabhängigkeit – Ist gleich wie bei der „.NET Remoting"-Technologie; es besteht also ebenfalls keine Möglichkeit, die Domäne für verschiedene Plattformen zu entwickeln.

Verbindungsaufbau – Findet unaufwendig und rasch statt; alle Adressen und Protokolle werden in der Konfigurationsdatei definiert.

Synchronisation - Wie in der vorhergehenden Technologie hat der Entwickler die Möglichkeit, den Synchronisationsmechanismus mit den Standard-.NET Werkzeugen zu implementieren.

Skalierbarkeit – Ist wesentlich besser als bei .NET Remoting. Die Services spielen dabei eine große Rolle. Wenn man sie überlastet, z.B. sitzungsbasiert, statusbehaftet und transaktional programmiert, verliert man an Skalierbarkeit [WR 7].
Evolution und Wartbarkeit – Ist gleich wie bei der „.NET Remoting"-Technologie.

Komplexität und Transparenz der Technologie – WCF hat ein hohes Abstraktionsniveau. Deshalb sind einige Mechanismen vor dem Entwickler „versteckt". Das Existieren verschiedener Informationsquellen im Internet und einer großen Auswahl an Fachliteratur macht das Verständnis der Technologie einfach.

Ausfallssicherheit – Ist gleich wie bei der „.NET Remoting"-Technologie.

Installation und Konfiguration der Plattform – WCF kommt als Standardkomponente des Betriebsystems Windows ab Windows-Vista; bei der Arbeit mir früheren Versionen von Windows muss man die Middelware auf dem Rechner installieren.

Lines of Code 234

So sieht die Bewertung der WCF-Technologie aus:

	++	+	~	-	--
Kriterien	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Notifikation		X			
Plattformunabhängigkeit:					X
Synchronisation			X		
Verbindungsaufbau	X				
Skalierbarkeit	X				
Evolution und Wartbarkeit		X			
Komplexität und Transparenz der Technologie			X		
Ausfallssicherheit			X		

Installation und Konfiguration der Plattform		X			

Tabelle 3: Bewertung der WCF-Technologie

5.4 ASP.NET

Obwohl die ASP.NET-Technologie für Systeme mit SOA sehr gut geeignet ist, ist sie im Fall von Systemen mit eventbasierter Architektur nicht die ideale Lösung. Die Implementierung bringt eine ganze Menge an lästiger Programmierarbeit mit sich.

Notifikation – Gibt es in dieser Technologie nicht. Deshalb müssen sich die Client-Domänen periodisch mit dem Server verbinden, um ihn abzufragen. Dies wird als Polling bezeichnet.[14] Das bedeutet zusätzlichen Aufwand für den Entwickler und eine zusätzliche Belastung der Netztrafik.

Plattformunabhängigkeit – Ist sehr hoch. Die Daten werden vom Server mittels SOAP-Protokoll geschickt. Jede Plattform, die dieses Protokoll versteht, kann die Daten bearbeiten. Aber es muss erwähnt werden, dass das Web-Service von ASP.NET nur auf IIS installiert werden kann.

Verbindungsaufbau – Findet bereits bei der Implementierung des Client-Programms statt. Die Adresse des Web-Dienstes wird beim Erzeugen des Proxy-Objekts im .NET Framework (z.B. Visual Studio) definiert.

Synchronisation - Die Technologie ist nicht geeignet für das Shared Data Konzept und dessen Synchronisation, weil die Middleware für SOA-Systeme geschaffen wurde. Am häufigsten wird ASP.NET in Kombination mit ADO.NET und Datenbanken benutzt, die diese Funktionen unterstützen.

[14] siehe http://de.wikipedia.org/wiki/Polling_(Informatik) (letzter Zugriff am 20.08.09)

Skalierbarkeit – Ist nicht besonders gut. Mit dem Ansteigen der beteiligten Clients wachsen auch die dafür benötigten Ressourcen stark an. Die Client-Domänen erzeugen eine große Anzahl an Verbindungen zum Web-Dienst. Dies kann zu einer großen Belastung der Netz-Trafik führen.

Evolution und Wartbarkeit – Änderungen im Web-Dienst oder das Verschieben des Web-Dienstes von einem Web-Server auf den anderen, kann die Notwendigkeit mit sich bringen, ein neues Proxy-Objekt zu generieren und alle Client-Programme neu zu kompilieren.

Komplexität und Transparenz der Technologie – ASP.NET ist sehr leicht zu verstehen. Die Entwicklung eines Verteilten Systems auf der Basis von ASP.NET ist unkompliziert und verlangt kein Erzeugen zusätzlicher Hilfsklassen.

Ausfallssicherheit – Ist von der Technologie nicht vorgesehen; leider bietet die Middleware auch kaum Werkzeuge, um die Ausfallssicherheit zu erhöhen.

Installation und Konfiguration der Plattform – ASP.NET ist Teil der Standardinstallation des Windows-Betriebssystems, um aber den Serverteil des Systems benutzen zu können, braucht man einen Web-Server mit einer eigenen speziellen Soft- und Hardware Infrastruktur. Außerdem müssen eine Menge an Einstellungen im Sicherheitssystem des Web-Servers vorgenommen werden.

Lines of Code – 67

Die Bewertung der ASP.NET-Technologie ist in Tabelle 4 sehen:

	++	+	~	-	--
Kriterien	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Notifikation				X	
Plattformunabhängigkeit:			X		
Synchronisation				X	
Verbindungsaufbau	X				
Skalierbarkeit				X	
Evolution und Wartbarkeit				X	

Komplexität und Transparenz der Technologie	X				
Ausfallssicherheit				X	
Installation und Konfiguration der Plattform				X	

Tabelle 4: Bewertung der ASP.NET-Technologie

5.5 Weitere Vergleichsdaten

Bevor ich zu den Schlussfolgerungen komme, werden Erfahrungen anderer Student/innen der TU Wien präsentiert.

In Rahmen des Übungsteils der Lehrveranstaltung „Verteiltes Programmieren mit Space Based Computing“ war ein Technologievergleich durchzuführen. (Mehr zur Aufgabestellung des Sommersemesters 2008 findet man im Anhang.)

Die Student/innen sollten die Übungsaufgabe einmal mittels XVSM (.NET oder Java) lösen und dann mittels einer anderen Technologie. In den Präsentationsfolien zu ihrer Arbeit sollten sie einen kurzen Vergleich der verwendeten Technologie anführen.

XVSM.NET betreffende Vergleiche gibt es nur zwei: die Arbeit von Gruppe 6 (Adnan Selimovic, Alen Suljagic) und von Gruppe 13 (Alexander Wagner, Alexander Marek).

Gruppe 6 hat XVSM(.NET) mit Sockets (Java) verglichen und ist zu folgendem Ergebnis gekommen (Abbildung 15 und Abbildung 16):

Intro
- Working Hours: 130 Hours
- Lines of Code(LOC)
  - File-Sharing
    - MozartSpaces: ~800
    - XCOSpaces: ~500
  - Scrabble
    - XCOSpaces: ~ 800
    - Sockets: ~700

Abbildung 15: Auszug aus der Arbeit von Gruppe 6

Die XCOSpaces und MozartSpaces sind XVSM-Technologien, für .NET bzw. für Java.

MozartSpaces
- Advantages
  - "embedded" core (no central server, except for list of clients)
  - Usage of "Tupels" and AtomicEntrys
- Disadvantages
  - Slow performace on write

LOC for reading from LindaContainer

XCOSpaces
- Same as MozartSpaces

Sockets
- Advantages
  - Full control over communication
  - Speed
- Disadvantages
  - Implementation of data controlling routines (read, write, etc.)

Time

Abbildung 15: Auszug aus der Arbeit von Gruppe 6

Gruppe 13 hat XVSM(XcoSpaces) mit WCF verglichen und folgende Ergebnisse präsentiert (Abbildungen 16 und 17):

Technologievergleich

	XVSM(XcoSpaces)	Corso	WCF
Vorteile	•EinfacheAnwendung (wegen gutem Tutorial, etc.) •Verschiedene Container/Selektoren •Callback sehr hilfreich •Aspekte einfach	•Sehr vieleMöglichkeiten •Läuft sehr stabil •Replikation wäre möglich	•Sehr stabil •Mehrere Protokolle wählbar/ konfigurierbar •Einfach zu implentieren
Nachteile			

	•Teilweisenoch buggy(Transaction-Timeout bei Shiftauf Container) •Teilweise nicht ausreichend dokumentiert (Aspekte)	•Sehr komplex •Keineverschiedenen Container und Selektoren =>Schwierig Datenstrukturen einzusetzen •Keine automatische Serialisierung(!!)	•Zentraler Server muss aktiv Koordinieren (bei Änderung der Datenmüssen alle Clients manuell benachrichtigt werden) •Keine Transaktionen

Abbildung 16: Auszug aus der Arbeit von Gruppe 13

Technologievergleich				
	FilesharingXcoSpaces	FileSharing Corso	Scrabble XcoSpaces	Scrabble WCF
Aufwand	mittel (erstmalige Verwendung)	hoch (komplexe API)	gering	mittel (lediglich Verwaltungsaufwand)
Performance	gut	gut	gut	mittel (keine so gute Concurrency wie bei „Konkurrenz“)
Komplexität	gering	hoch	mittel	mittel bis hoch
LinesofCode	1039	1267	2047	1853
Flexibilität	gut (Container leicht erweiterbar)	diffizil(komplexe Datenstrukturen)	gut (Container leicht erweiterbar)	sehr gut (Protokolle + Bindings konfigurierbar)

Abbildung 17: Auszug aus der Arbeit von Gruppe 13

Wie man sieht, ähneln die Ergebnisse dieser Arbeiten sehr stark den Ergebnissen der vorliegenden Diplomarbeit. Jedoch muss darauf hingewiesen werden, dass in den Arbeiten der Lehrveranstaltung „Verteiltes Programmieren mit Space Based Computing“ teilweise nicht nur verschiedene Technologien sondern auch verschiedene Sprachen verglichen werden.

6 Schlussfolgerungen

Im Rahmen dieser Diplomarbeit wurde ein verteiltes Beispielsystem mit eventbasierter Architektur projektiert und mit vier verschiedenen Middlewares aus der .NET-Welt entwickelt. Dieses System stellt eine typische verteilte Anwendung dar. Jede Variante des Systems ist funktionsfähig und erfüllt die Anforderungen, die in Kapitel 2.1 beschrieben wurden. Während der Entwicklung des Systems konnte ich Erfahrungen sammeln, die mir die Möglichkeit geben, die vier Technologien zu vergleichen.

Als Ergebnis meiner Diplomarbeit ist folgende Vergleichstabelle (Tabelle 5) und unten stehendes LOC-Diagramm (Abbildung 18) entstanden:

	XVSM	.NET Remoting	WCF	ASP.NET
Notifikation	1	3	2	4
Plattformunabhängigkeit	1	5	5	3
Synchronisation	1	3	3	4
Verbindungsaufbau	1	3	1	1
Skalierbarkeit	2	3	1	4
Evolution und Wartbarkeit	3	2	2	4
Komplexität und Transparenz der Technologie	2	3	3	1
Ausfallssicherheit	1	4	3	4
Installation und Konfiguration der Plattform	2	1	2	4
Gesambewertung	**1,56**	**3,00**	**2,44**	**3,22**

Tabelle 5: Technologievergleich

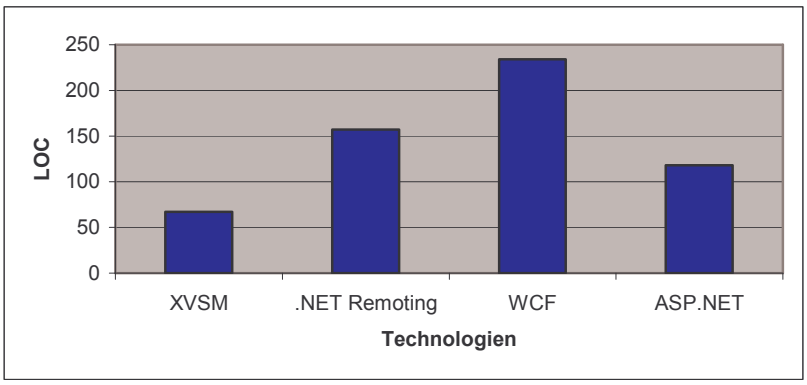

Abbildung 18: LOC-Diagramm

Aus dieser Tabelle wird ersichtlich, dass sich alle Technologien in etwa auf einem Niveau befinden. Es gibt jedoch zwei „Gewinner“, XVSM und WCF; die Technologien .NET Remoting und ASP.NET liegen etwas dahinter.
.NET Remoting und ASP.NET haben so wenige Punkte erhalten, da sie bei der Entwicklung eines Verteilten Systems mit eventbasierter Architektur meiner Meinung nach gewisse Nachteile besitzen. Zum Beispiel die komplexe Darstellung der Remote Objects in .NET Remoting. Oder das Fehlen eines Benachrichtigungsmechanismus in ASP.NET, was als Folge hat, dass ASP.NET trotz seiner Popularität und Einfachheit bei der Projektierung von Systemen mit serviceorientierter Architektur in unserem Fall am wenigsten geeignet ist. Es hat sehr gegensätzliche Bewertungen und kombiniert einerseits perfekte Plattformunabhängigkeit mit guter Verständlichkeit, unterstützt andererseits aber keine Notifikationen und hat eine niedrige Ausfallssicherheit.

.NET Remoting wurde ähnlich bewertet wie ASP.NET. Die Bewertung ist zwar nicht so gegensätzlich, man kann sie aber trotzdem nur als mittelmäßig bezeichnen. Die Technologie eignet sich demnach nicht besonders gut für die Entwicklung von Systemen mit eventbasierter Architektur. Sie bietet jedoch gute Möglichkeiten zur Entwicklung von Systemen mit einem niedrigen Abstraktionsniveau. Genau diese Eigenschaft verlangt dem Entwickler einen großen Programmieraufwand ab und impliziert, dass er bereits gute Kenntnisse im Bereich der Verteilten Systeme haben muss. Ich bin der Meinung, dass diese Technologie besser dazu geeignet ist, eine Middleware zu entwickeln, als direkt bei der Projektierung von Verteilten Systemen mit eventbasierte Architektur verwendet zu werden.

Eine der beiden „Gewinner“-Technologien ist WCF. Diese Middleware ist relativ neu und Teil des neuen Betriebssystems Windows Vista. Sie erfüllt alle Grundanforderungen für den Aufbau von Verteilten Systemen. Aufgrund einer gewissen Komplexität bei der Beschreibung und Definition von Contracts, wurde sie jedoch während der Entwicklung des Beispielsystems nicht als ausreichend praktisch empfunden. Auch die Existenz einer Server-Domäne kann als Nachteil gesehen werden, weil jede Firma, die ein solches Service anbietet, mit einer Installation auf ihrem Server und mit den dazugehörigen Sicherheitseinstellungen einverstanden sein muss. Im Großen

und Ganze ist WCF demnach eine gute Middleware, um ein eventbasiertes Verteiltes System aufzubauen, aber nicht hundertprozentig bequem in der Handhabung.

Die andere „Gewinner“-Technologie ist XVSM. Diese Middleware hat zwei mächtige Mechanismen – Notifikationen und Transaktionen –, die sie für die Entwicklung von Verteilten Systemen mit eventbasierter Architekturen sehr geeignet machen. Das entwickelte System arbeitet stabil und ist noch dazu für den Entwickler leicht erfassbar. XVSM fordert vom Entwickler keine tiefgehenden Kenntnisse in diesem Bereich. Dei Technologie wird nur mit einer Note, die schlechter als 2 ist, bewertet. Das letzte Argument, das für diese Technologie als „Gewinner“ spricht, sind die Lines of Code. Aus Abbildung 18 (S. 84) wird ersichtlich, dass von allen entwickelten Beispielsystemen das mit der XVSM-Technologie projektierte System die geringste Anzahl an funktionalen Codezeilen aufweist.

7 Anhang

7.1 Anhang 1: Entscheidungsalgorithmus

Eingabe: Folge Plätze, Folge Kriterien

Ausgabe: bester Platz aus Folge Plätze

Variables: Index i, j

```
1:    für i = 1 … n {
2:          sort(Plaetze, Kriterien[i]);
3:          für j = 1 … m {
4:                Plaetze[j].punkte = Plaetze[j].punkte + j;
5:          }
6:    }
7:    sort (Plaetze, punkte);
8:    falls (Plaetze[1].punkte != Plaetze[2].punkte) dann {
9:          BestPlatz = Plaetze[1];
10:   } sonst {
11:   für i = 1 … n {
12:         falls (Platze[1].Kriterien[i] != Platze[2].Kriterien[i]){
13:               falls (Platze[1].Kriterien[i]>
Platze[2].Kriterien[i]){
14:                     BestPlatz = Plaetze[2];
15:                     retourniere BestPlatz;
16:                     stop;
17:               } sonst {
18:                     BestPlatz = Plaetze[1];
19:                     retourniere BestPlatz;
20:                     stop;
21:               }
22:         }
23:   }
24:   BestPlatz = Plaetze[1];
25:   retourniere BestPlatz;
```

7.2 Anhang 2: Beschreibung der gemeinsamen Klassen und GUI

Die Klasse *Util* beinhaltet eine Menge statischer Objekte, die Auswirkungen auf die ganze Applikation haben können.

Die Entfernungsmatrix *distance_matrix* stellt die Strecke zwischen den verschiedenen Orten dar. Da mein Ziel keine fertige kommerzielle Anwendung mit Bewegungsoptimierung und Bestimmung der Autoposition ist, wird die Lage von Auto, Parkhaus oder Zielort nur abstrakt definiert und als „Zone“ bezeichnet (z. B. befindet sich das Auto in der 1. Zone, das Parkhaus in der 4. Zone und der Zielort ist in der 6. Zone) In meinem Beispiel gibt es 25 Zonen. Die statische Methode *getDistance(int zoneA, int zoneB)* gibt die Entfernung zwischen zwei Zonen zurück, die Entfernung ist abstrakt und hat keine Maßeinheit. Eine verkürzte Darstellung der Matrix kann man in Abbildung 19 sehen.

	1. zone	2. zone	3. zone	…	…	23. zone	24. zone	25. zone
1. zone	0	645	868	…	…	343	312	396
2. zone	645	0	252	…	…	324	891	672
3. zone	868	252	0	…	…	547	1141	867
…	…	…	…	0	…	…	…	…
…	…	…	…	…	0	…	…	…
23. zone	343	324	547	…	…	0	660	330
24. zone	312	891	1141	…	…	660	0	695
25. zone	396	672	867	…	…	330	695	0

Abbildung 19: Entfernungsmatrix

Das Array *private static string[] list = new string[distance_matrix.Length]* beinhaltet die Zonenbezeichnungen im folgenden Format: „[ZonenNr]. zone“; das Array wird mit der Methode *getList()* gefüllt.

Das Array *public static string[]decisionFactors* beinhaltet die möglichen Entscheidungskriterien (notwendig für die Erstellung einer Reihung nach der Wichtigkeit der Entscheidungskriterien, siehe Brokerbeschreibung). In meinem Beispiel hat das Array die Werte: `"keine", "Enfernung zum Ziel", "Enfernung zum Parkhaus", "Preis", "Kategorie"`

Das Array definiert die Parkplatzkategorien, die im Parkhaussystem vorhanden sind. In meinem Beispiel sind folgende fünf Parkplatzkategorien vorgesehen:

`PKW` – „normaler“ Parkplatz für einen PKW;
`behinderte` – behindertengerechter Parkplatz;
`frauen` – Frauenparkplatz;
`bus` – Parkplatz für Busse;
`LKW` – Parkplatz für LKWs;

Das Array *public static string[] timeout* dient zum Speichern von Timeoutwerten. Ein Timeout ist die Zeitspanne, die für die Zuteilung eines Parkplatzes vorhanden ist. In dieser Zeit muss der Broker eine Antwort an den Kunden schicken. Ich benutze einen Wert von 3, 5 oder 10 Sekunden.

Wie Sie sehen, kann das Parkplatzsystem sehr einfach an konkrete Anforderungen anpasst werden. Dafür müssen nur die entsprechenden Daten in der Klasse Util geändert werden.

Wie auch andere Verteilte Systeme muss das System den Informationsaustausch zwischen den Domains unterstützen. Zur Darstellung von Informationen dient die *MyMessage*-Klasse, die eine Datenstruktur darstellt, siehe Listing 38. Die Datenstruktur beinhaltet alle notwendigen Informationen für die Systemarbeit, also die Informationen zur Kundenanfrage und zum Parkplatz.

```
[Serializable]
    public class MyMessage
    {
        private int customerID;
        private int parkingPlaceCategory;
```

```
private int targetPlace;
private int startPlace;
private int parkingID;
private string parkingName;
private int parkingAddresse;
private int parkingPlaceID;
private double price;
private double distanceToParking;
private double distanceToTarget;
private bool isFree;
private int points;
private int customerParkingPlace;
```

Listing 38: Implementierung der Klasse *MyMessage*

Da das Klassenobjekt über das Netzwerk übertragen wird, muss die Klasse das Attribut *[Serializable]* haben. Die Klasseninstanzvariablen beinhalten folgende Daten:

`int customerID` – eindeutige Kundennummer;

`int customerParkingPlace` – Index des Parkplatzkategorie-Arrays, wird in der Kunden-Domain gesetzt und beinhaltet die vom Kunden gewünschte Parkplatzkategorie;

`int parkingPlaceCategory` – Index des Parkplatzkategorie-Arrays, diese Variable wird gefüllt, falls ein freier Parkplatz im Parkhaus existiert und der gewünschten Parkplatzkategorie des Kunden entspricht;

`int targetPlace` – Index des Arrays, der die Bezeichnung für den Zielort beinhaltet;

`int startPlace` – Index des Arrays, der die Bezeichnung für den aktuellen Standort beinhaltet;

`int parkingID` – eindeutige Parkhausnummer, wird gefüllt, falls das Parkhaus einen Platz für den Kunden reserviert;

`string parkingName` – Name des Parkhauses;

`int parkingAddresse` – Index des Arrays, der die Bezeichnung für die Parkhausadresse beinhaltet;

`int parkingPlaceID` – eindeutige Nummer des Parkplatzes, der im Parkhaus reserviert wurde;

`double price` – Preis pro Stunde in einem Parkhaus;

`double distanceToParking` – Entfernung vom Standort bis zum Parkhaus;

`double distanceToTarget` – Entfernung vom Parkhaus bis zum Zielort;

`bool isFree` – Variable, die anzeigt, ob der Parkplatz wieder freigeben werden soll (per default gleich „false“), wird vom Broker nur dann auf „true“ gesetzt, wenn es einen anderen, besseren Parkplatz gibt;
`int points` – Variable beinhaltet jene Punkte, die der Broker beim Treffen einer Entscheidung benötigt;

Des Weiteren wurden für die Variablen und Klassenkonstrukteure Get- und Set-Eigenschaften entwickelt.

Als nächstes betrachten wir die anderen wiederverwendbaren Klassen im Detail. Die Klasse *PleaseWaitDialog* ist eine Erweiterung von *System.Windows.Forms*, die dem Benutzer gegebenenfalls den Verbindungsprozess mit dem Server anzeigt.
Die Dialogfenstern *InputDialogSender*, *InputDialogParking*, *InputDialogBroker* sind notwendig, um die entsprechende Domain zu starten. Sie fordern den Benutzer auf, einige notwendige Informationen einzugeben.

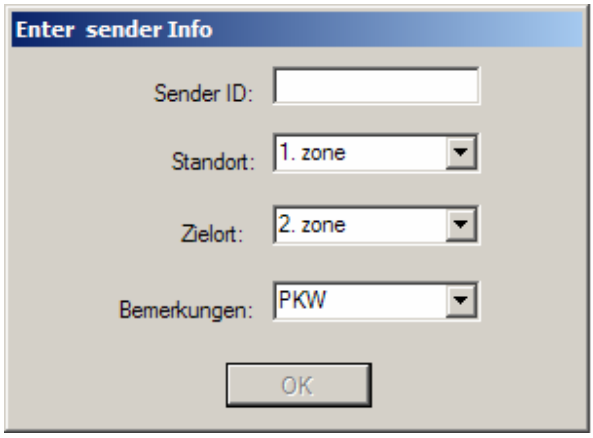

Abbildung 20: Startformular der Kunden-Domäne

Aus Abbildung 20 wird ersichtlich, welche Informationen einzugeben sind, um das den Kunden repräsentierende Programm zu starten. Folgende Daten sind erforderlich: eindeutige Kundennummer (Sender ID), Standort, Zielort und als Bemerkung die Parkplatzkategorie. Alle

diese Daten werden in einem Objekt der *MyMassage*-Klasse gespeichert und über das Netz ans System übermittelt.

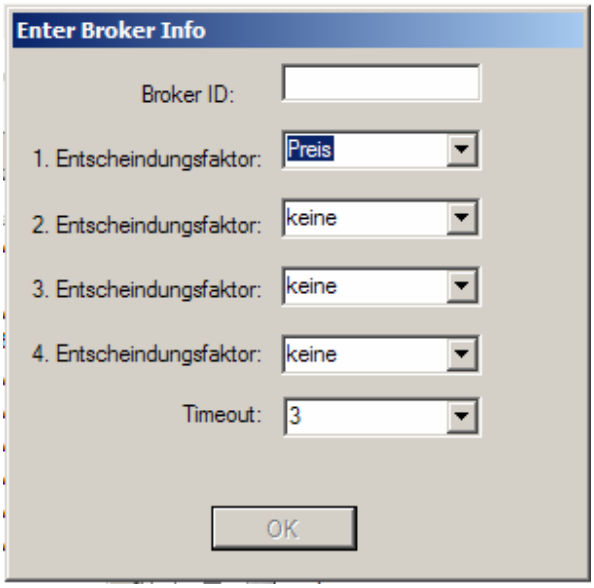

Abbildung 21: Startformular der Broker-Domäne

Abbildung 21 zeigt uns eine Ansicht des Dialogfensters *InputDialogBroker*. Es verlangt die Eingabe folgender Daten: die eindeutige Brokernummer (Broker ID), die vier Entscheidungskriterien (wobei wiederholte Kriterien ignoriert werden) und eine Timeout-Variante. Zur Brokerdomain gehört darüber hinaus die Klasse *CustomerList*, die eine Datenstruktur darstellt. Diese Datenstruktur speichert alle Meldungen, die während dem Warten auf und dem Sammeln von Kundenanfragen zum Broker gelangen.

`private List<MyMessage> list` – Liste, die alle passenden Parkplätze beinhaltet;

`private int timeCounter` – Zeitzähler; wird jede Sekunde um eins erhöht;

In die Klasse *Broker* wurde eine Datenstruktur implementiert, die die Arbeit des Brokers darstellt:

`private int id` – eindeutige Brokernummer;

`private int faktor1` – erster Entscheidungsfaktor, Index aus dem entsprechenden Array;

`private int faktor2` – zweiter Entscheidungsfaktor, Index aus dem entsprechenden Array;

`private int faktor3` – dritter Entscheidungsfaktor, Index aus dem entsprechenden Array;

`private int faktor4` – vierter Entscheidungsfaktor, Index aus dem entsprechenden Array;

`private int timout` – Timeout;

`private Dictionary<int, CustomerList> senders` – Dictionary-Struktur, in der jeder Kundenanfrage (identifiziert durch die Kundennummer) ein Objekt der *CustomerList*-Klasse entspricht;

`private List<MyMessage> list` – Hilfsliste;

Der auf Seite 26 beschriebene Algorithmus für die Auswahl des besten Parkplatzes wurde in die Methode `public MyMessage getBestPlace(int customerID)` implementiert. Dafür wurden einige Hilfsmethoden benötigt:

`private void collectPoints()` – Addiert nach jedem Sortierdurchgang die Punkte, die ein Parkplatz erhält. Für jeden Entscheidungsfaktor gibt es einen Sortierdurchgang. Die Punkte geben an, welche Position ein Parkplatz in der sortierten Liste einnimmt. Ein Parkplatz kann bei jedem Sortierdurchgang eine unterschiedliche Position einnehmen und dementsprechend unterschiedlich viele Punkte erhalten. Am Ende gibt die geringste Punkteanzahl an, welcher der günstigste Parkplatz ist.

`private MyMessage getBestByFaktor(int faktor)` – Vergleicht die Parkplätze dann, wenn sie am Ende des Sortierprozesses die gleiche Punkteanzahl haben. In diesem Fall wird der wichtigste Entscheidungsfaktor als entscheidendes Vergleichskriterium hergenommen. Wenn auch hier das Ergebnis gleich ist, wird der zweite Entscheidungsfaktor herangezogen (gegebenenfalls auch der dritte bzw. vierte).

`private void sortListByField(int faktor)` – Sortiert eine Liste mit MyMessage-Objekten mit Hilfe der gegebenen Entscheidungsfaktoren.

Die statischen Methoden

`private static int SortByTargetDistance (MyMessage x, MyMessage y),`
`private static int SortByParkingDistance(MyMessage x, MyMessage y),`
`private static int  SortByPrice (MyMessage x, MyMessage y),`
`private static int  SortByCategory (MyMessage x, MyMessage y)` und
`private static int SortByPoints(MyMessage x, MyMessage y)` sind notwendig, um die Liste entsprechend der Entscheidungsfaktoren zu sortieren.

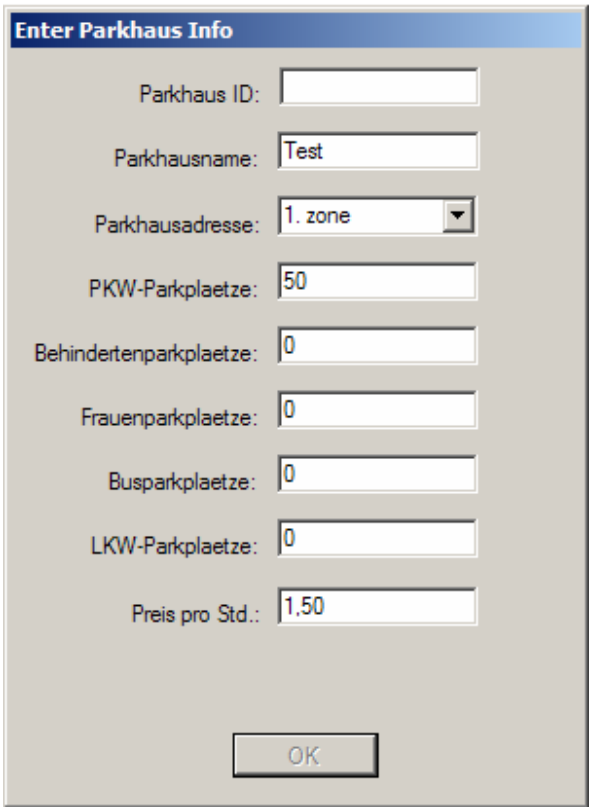

Abbildung 22: Startformular der Parkhaus-Domäne

Abbildung 22 zeigt das Fenster *InputDialogParking* und die Daten, die zum Starten der Parkhausdomain notwendig sind. In der Domain gibt es auch eine *Parking*- und eine *ParkingPlace*-Klasse. *ParkingPlace* ist die Struktur, die einen Parkplatz beschreibt:

`private int id` – eindeutige Nummer des Parkplatzes im Parkhaus;

`private string category` – Kategorie des Parkplatzes;

`private bool isFree` – Zustand des Parkplatzes (true – Platz ist frei, false - der Platz wurde reserviert);

In der Klasse *Parking* wurde eine Datenstruktur definiert, die das Parkhaus repräsentiert:

`int id` – eindeutige Nummer des Parkhauses;

`int address` – Parkhausadresse, und zwar Index aus dem Adressenarray;

`string name` – Parkhausname;

`ParkPatz[] places` – Array mit Parkplätzen für das Parkhaus;

`double price` – Preis pro Stunde;

Außerdem gibt es in dieser Klasse folgende Methoden:

`public void setParkingPlaces(int passenger_car, int disabled_person, int ladies, int bus, int lorry)` – Erzeugt ein Array und befüllt es mit der entsprechenden Anzahl von Parkplätzen der einzelnen Kategorien.

`public int getFreePlace(int customerPlaceCategory)` – Reserviert einen Parkplatz der gewünschten Kategorie und gibt seine ID als Ergebnis zurück. Falls es keine freien Plätze der Kategorien „Frau“ oder „Behinderte“ gibt, wird ein Platz aus der Kategorie PKW reserviert.

7.3 Anhang 3: Fragebogen „Einsatz von Middleware-Technologien“

Questionnaire
Use of Middleware Technologies

Technical University of Vienna
Institute of Computer Languages E185/1

Technology Questionnaire: Target group are IT developers in the field of “Distributed Systems”. The questionnaire is thought to be filled in on a per project basis.
The purpose of this questionnaire is to improve distributed system technologies. It will be used as a pre-work for a diploma thesis at the Technical University of Vienna that deals with the evaluation of middleware technologies. Goal of this project is to compare the features of these technologies and classify them.
To do that your feedback and experiences will be much appreciated. If you wish, your data will be treated anonymously!
If you are interested in the results of the questionnaire please fill in at least your email address. We will inform you on the results as soon as they are available.

I General Information

[optional]

First and last name:_____ **Address:**_____

Position:_____ **Email:**_____

Company / Institution:_____ **Date & Place:**_____

II Company Information

What is the number of the employees, working in the IT department in your company?

1 – 3 persons	
3 – 10 persons	
10 – 15 persons	
15 – 30 persons	
more than 30 persons	

For which industries or branches do you develop software solutions?

advertising, marketing and PR		metal processing	
aerospace		multimedia	
automotive industry		oil and gas industry	
banking, financial service		pharmacy	
chemicals		precision mechanics, optics, clock and watch industry	
civil engineering		public sector	
consumer products		real estate	
contractors and engineering companies		retail	
defense		robotics	
education and research		security service	
energy and water supply		service industries	
healthcare industry		synthetic materials	
heating		telecommunications	
insurance		textile industry	
logistics and transport		tourism and hospitality	
mechanical engineering		waste management and recycling	
media		wood and furniture industry	
medical systems		ISP/ASP	
other:			

III Use of middleware technologies in your company

Which enterprise products/technologies of which companies do you use as a basis/foundation in your software applications? E.g: Enterprise Service Bus, Enterprise Integration Patterns, Service Oriented Architectures etc.

Company	*Product*:
IBM	
Microsoft	
Oracle	
SAP	
Open Source	
other:	

Which communication technologies for distributed applications do you employ?

RPC (Remote Procedure Calls)	
WCF (Windows Communication Foundation)	
DCE (Distributed Computing Environment)	
MOM (Message Oriented Middleware)	
CORBA (Common Object Request Broker Architecture)	
DCOM (Distributed Common Object Model)	
EJB / RMI (Enterprise Java Beans /Remote Method Invocation)	
DTPM (Distributed Transaction Processing Middleware)	
Database Connectivity Middleware	
Space Technology (Virtual Shared Memory)	
.NET Remoting	
Sockets	
other:	

Which programming platforms/languages do you use?

Java		PHP	
.NET (C#, VB.NET)		Ada	
VB6		Abap	
C/C++		Delphi	
Ruby		other:	
Python			

How satisfied are you with the following middleware supported features that you use?

Please rank on a scale (1) to (5): (1)...very satisfied; (5)...unsatisfied.

	(1)	(2)	(3)	(4)	(5)
authorisation					
encryption					
notification					
transaction					
multi party interaction					
abstraction					
replication					
complexity of the implementation					
usage of the tools					

Are there missing features, which you have to implement by yourself?_____

If yes, how would you describe the needed effort for the implementation of these features?

Please rank on a scale (1) to (5): (1)...very large effort; (5)...no effort.

	(1)	(2)	(3)	(4)	(5)
authorisation					
encryption					
notification					
transaction					
multi party interaction					
abstraction					
replication					
complexity of the implementation					
usage of the tools					

Do you introduce by yourself features into middleware solutions used by you?

If yes, could you please let us know about these features and describe the reasons for such decision?

Which features of your application software are currently provided by your middleware?

Please rank the support of the features by your middleware on a scale (1) to (5): (1)...full support; (5)...no support.

	(1)	(2)	(3)	(4)	(5)
stability of the technology					
secuity mechanism					
ability to integrate the tools at the customer					
reliability of the application					
scalability of the technology					
extensibility and substitutability of the system					
simultaneous usage of the technology					
data protection (from changes and damage to the data)					
transparency of the systems					
adding of new requirements to the system					
performance of the tool					
reusability of the features					

How do you evaluate the role of your middleware in the integration of your applications for the following tasks?

Please rank the importance on a scale (1) to (5): (1)...very important; (5)...unimportant.

	(1)	(2)	(3)	(4)	(5)
Support for data exchange between the system components					
Support for integration of databases					
Support for integration of legacy systems					

IV Business Information

Please rank the following statements.

Legend: (1) fully applies; (2) applies to certain extent; (3) no opinion; (4) hardly ever applies; (5) not applicable.

	(1)	(2)	(3)	(4)	(5)
All middleware features are used in my software solutions					
Training needs for middleware are very high					
Are you interested in evaluating alternative middleware?					

Do you plan in the foreseeable future to implement an additional middleware technology into your solutions? _____

If yes, which reasons influence such decisions?

stability	
semantics support	
security mechanism	
system integration	
reliability	
scalability	
extensibility and substitutability	
confidentiality of data	
simultaneous usage of resources	
data protection (from changes and damage to the data)	
transparency of the systems	
performance	
reusability	
platform independence	
other:	

Who is the decision maker in your company regarding the use of additional technologies?

What in your opinion are the most three important preconditions to decide for a new middleware technology?

If you manufacture a product based on a middleware solution: how many installations of your product are currently in use in total? Please differentiate between installations at client and server side.

Which sources do you use to get more information regarding the topic of distributed systems and new solutions in that field?

direct contact with manufacturers	
workshops	
conferences	
trainings	
publications / magazines	
fairs	
internet	
peer developers / partners	
other:	

Do you want us to keep you informed regarding the result of this project and new developments in the field of middleware technologies?____

Many thanks for your cooperation!

8 Literaturverzeichnis

8.1 Bücher, Zeitschriften

[1] Andrew S. Tanenbaum: *„Verteilte Systeme: Grundlagen und Paradigmen“*, Pearson Studium, 2003.

[2] Martin Beinhart: *„Entwurf und Implementierung einer service-orientierten Software-Architektur durch Integration verschiedener Middleware-Systeme“*, Diplomarbeit, TU-Wien 2005.

[3] Alberto Coen-Porisini [Hrsg.]: *“Software engineering and middleware”*, Springer, 2003.

[4] Steffen Heinzl, Markus Mathes: “Middleware in Java”, Vieweg, 2005.

[5] Thomas Jell: „Middleware Technologien zur Entwicklung unternehmenskritischer Anwendungen“, LogOn Technology Transfer, 1998.

[6] Scott McLean, James Naftel, Kim Williams: *„Microsoft .NET Remoting“*, Microsoft Press, 2002.

[7] Matthew MacDonald, Mario Szpuszta: *„Pro ASP.NET 2.0 in C# 2005“*, APress, 2005.

[8] Humphrey Sheil and Michael Monteiro: *„How do J2EE and Microsoft's .Net compare in enterprise environments?“*, JavaWorld.com, 2002.

[9] Apostolos Zarras: *„A Comparison Framework for Middleware Infrastructures“*, JOURNAL OF OBJECT TECHNOLOGY, Vol. 3, No. 5, May-June 2004.

[10] Bernhard Voglmayr: *„Middleware-Konzepte für verteilte Automatisierungssysteme basierend auf IEC 61499“*, Diplomarbeit, TU-Wien 2007.

[11] Ayse Cicek: *„Communication and Collaboration in Java based Middleware Technologies“*, Diplomarbeit, TU-Wien (in Arbeit).

[12] Schahram Dustdar, Harald Gall, Manfred Hauswirth: *„Software-Architekturen für Verteilte Systeme“*, Springer, 2007.

[13] Tomas Scheller: *„Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM“*, Diplomarbeit, TU-Wien (in Arbeit).

[14] eva Kühn: *“Virtual Shared Memory for Distributed Architectures”*, Nova Science Pub Inc., 2002

[15] Robert Tolksdorf, Franziska Liebsch, and Duc Minh Nguyen: *„XMLSpaces.NET: An Extensible Tuplespace as XML Middleware.“*, Technical Report B 03-08, FU Berlin, Institut für Informatik, 2003.

[16] Christian Schreiber: *„Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM Custom Coordinators, Transactions and XML protocol“*, Diplomarbeit, TU-Wien 2008.

[17] Anton Michlmayr: *„Integrating transactions with content-based publish/subscribe middleware“*, Diplomarbeit, TU-Wien, 2005.

[18] Jürgen Kotz, Rouven Haban, Simon Steckermeier: *„.NET 3.0 WCF,WPF und WF – Ein Überblick“*, Addison Wesley, 2007.

[19] Dr. Holger Schwichtenberg: *„Microsoft .NET 3.0 Crashkurs“*, Microsoft Press Deutschland, 2007.

[20] Andrew W. Troelsen: *„C sharp and the .NET Platform“*, Computer Bookshops, 2003.

[21] Ayse Cicek: *„The use of middleware technologies“*, Praktikumsarbeit, TU-Wien, (in Arbeit).

[22] Krzysztof Cwalina, Brad Abrams: *„Richtlinien für das Framework-Design“*, Addison Wesley, 2007.

[23] Tomas Scheller: *„XcoSpaces .Net Kernel Tutorial“*, Diplomarbeit, TU-Wien 2008.

[24] Kurt Holm: *„Der Fragebogen - die Stichprobe“*, Francke, 1991.

[25] P. Jogalekar, M. Woodside: *„Evaluating the Scalability of Distributed Systems. IEEE Transactions on Parallel and Distributed Systems“*, Volume 11, Issue 6, Pages 589-603, June 2000.

[26] Bernhard Löwenstein: *„Benchmarking of Middleware Systems“*, Diplomarbeit TU Wien, 2008.

[27] Pascal Fenkam, Mehdi Jazayeri: *„Formally designing an event-based application for mobile collaboration: a case study“*, 4. Software Engineering and Middleware Workshop, Linz, 2004.

[28] Gustavo Alonso [Hrsg.]: *„Middleware 2005“*, Springer, 2005.

[29] Wolfgang Emmerich: *„Engineering Distributed Objects“*, John Wiley & Sons, 2000.

[30] eva Kühn: *„Space Based Computing – Leveraging the Coordination Paradigm“*.

[31] Robert C. Martin: *„Agile Software Development, Principles, Patterns, and Practices“*, Pearson Education, Inc., 2003.

[32] eva Kühn, Schmied, Fabian: *„XL-AOF - lightweight aspects for space-based computing“*, 1. Workshop on Aspect Oriented Middleware Development, Grenoble, 2005.

8.2 Web-Referenzen

[WR 1] Gregor Hohpe: *„Programming Without a Call Stack— Event-driven Architectures“*, www.eaipatterns.com, 2006 (letzter Zugriff am 20.08.09).
[WR 2] http://lindaspaces.com/ (letzter Zugriff am 20.08.09).
[WR 3] Qusay H. Mamoud: *„Getting Started With JavaSpaces Technology“*, July 12, 2005 http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/ (letzter Zugriff am 20.08.09).
[WR 4] http://www.almaden.ibm.com/cs/tspaces/ (letzter Zugriff am 20.08.09).
[WR 5] Microsoft: *„Einführung in .NET Framework 3.0“*, http://www.microsoft.com/germany/msdn/library/net/EinfuehrungInNETFramework30.mspx?mfr=true (letzter Zugriff am 20.08.09).
[WR 6] https://portal.ftw.at/projects/all/realsafe (letzter Zugriff am 20.08.09).
[WR 7] http://msdn.microsoft.com/de-de/magazine/2009.01.foundations.aspx (letzter Zugriff am 20.08.09).
[WR 8] http://de.wikipedia.org/wiki/Middleware (letzter Zugriff am 20.08.09).
[WR 9] http://de.wikipedia.org/wiki/Web_Service (letzter Zugriff am 20.08.09).
[WR 10] http://www.spacebasedcomputing.org/ (letzter Zugriff am 20.08.09).
[WR 11] http://msdn.microsoft.com/en-us/magazine/cc300474.aspx. (letzter Zugriff am 20.08.09).
[WR 12] http://www.dotnetframework.de/%7B632A91E5-52D1-4191-881E-D18BD40936A5%7D.aspx (letzter Zugriff am 20.08.09).
[WR 13] http://de.wikipedia.org/wiki/Transaktion_(Informatik) (letzter Zugriff am 20.08.09).
[WR 14] http://www.gigaspaces.com/wiki/display/XAP66/About+Jini (letzter Zugriff am 20.08.09).
[WR 15] http://msdn.microsoft.com/de-de/library/bb979090.aspx (letzter Zugriff am 20.08.09).
[WR 16] http://www.inf.unisi.ch/carzaniga/siena/software/index.html (letzter Zugriff am 20.08.09).
[WR 17] http://de.wikipedia.org/wiki/Semaphor_(Informatik) (letzter Zugriff am 20.08.09).

[WR 18] http://de.wikipedia.org/wiki/Polling_(Informatik) (letzter Zugriff am 20.08.09).

[WR 19] http://www.javaworld.com/javaworld/jw-06-2002/jw-0628-j2eevsnet.html (letzter Zugriff am 20.08.09).