Kapselung verschiedener Middleware-Technologien

Eine flexible und zugleich benutzerfreundliche, vernetzte Zusammenarbeit ist in der Software-Entwicklung unabdingbar geworden. Zu diesem Zweck haben wir eine schlanke, dienstanbietende Schnittstelle entwickelt, die die drei Middleware-Technologien Cajo, Java Message Service und JXTA kapselt. Als Ergebnis steht eine technologieunabhängige Schnittstelle zur Verfügung, welche den Datenaustausch für verschiedene Anwendungen über eine situationsgerechte Middleware ermöglicht. Drei solche Anwendungen, die einen Grossteil der erforderlichen Kommunikationsmöglichkeiten für gemeinsame Software-Entwicklung abdecken, haben wir ausgearbeitet: Reine, textbasierte Kommunikation (Chat), Austausch von Dateien (File Sharing) und Verteilung von Java-Objekten (Messaging).

Franco Ehrat, Peter Gysel, Joel Muller | peter.gysel@fhnw.ch

In einer Zeit, in der Mobilität und Kommunikation gross geschrieben werden, müssen auch in der Software-Erstellung Kommunikationsbedürfnisse beachtet werden. Entwickler müssen unabhängig vom aktuellen Standort kollaborativ neue Produkte erstellen und testen können. In Projektgruppen soll neben der reinen Entwicklung auch der Ergebnis- und Erfahrungsaustausch auf einem schnellen und einfachen Weg stattfinden können.

Die äusseren Bedingungen, unter welchen kommuniziert wird, können sich stark verändern. Wenn die Kommunikationspartner sich innerhalb des gleichen Netzes befinden, so haben sie quasi unbeschränkte Bandbreite zur Verfügung. Befinden sie sich jedoch in getrennten lokalen Netzen, dann stehen oft zwei Firewalls zwischen ihnen und es wird möglicherweise zweimal eine Network Address Translation (NAT) durchgeführt. In einem dritten Szenario befindet sich der eine Partner unterwegs und ist somit nur über mobile Kommunikation erreichbar. Benötigt wird ein Kommunikationswerkzeug, mit dem sowohl Entwickler untereinander, als auch Entwickler mit Kunden auf einfache Art und Weise über die situationsgerechte Middleware Daten austauschen können. Dieses Bedürfnis war die Motivation für die Software-Firma Giniality AG in Basel, um mit der Fachhochschule Nordwestschweiz ein solches Kommunikations-Framework mit besonderer Berücksichtigung der Peer-to-Peer (P2P) Kommunikation zu entwickeln.

Ziele

Mit einem Prototyp wollen wir demonstrieren, wie ganz verschiedene Kommunikationstechnologien gekapselt und über eine einzige, zentrale Schnittstelle, das Service Providing Interface (SPI), angesprochen werden können (Abb. 1). Das SPI soll die nötigen Funktionalitäten für die folgenden drei Anwendungen beinhalten: textbasierte Kommunikation (Chat), Austausch von Dateien (File Sharing, Server-Seite und Client-Seite) und Verteilen von Java-Objekten (Messaging). Unabhängig von der verwendeten Middleware-Technologie werden die Kommunikationsmöglichkeiten zentral im SPI definiert und erst in den darunter liegenden Technologie-Adaptern konkret implementiert. Die Technologie-Adapter verwenden einerseits ihre eigenen Technologie-Frameworks und erweitern bzw. implementieren anderseits das SPI.

Für die folgenden drei Middleware-Technologien stellen wir exemplarisch Technologie-Adapter zur Verfügung: Virtual-Virtual Machine (Cajo, beruhend auf Java Remote Method Invocation, RMI), Message Oriented Middleware (Java Message Service, JMS) und Peer-to-Peer (JXTA).

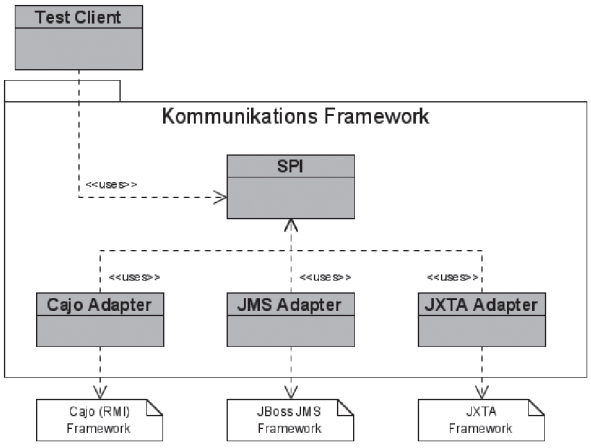

Abb. 1: Kommunikations-Framework: Verschiedene Middleware-Technologien werden über eine zentrale, technologieun-

Mit dem zuvor skizzierten Entwicklungsmuster soll dem Framework ein hoher Grad an Erweiterbarkeit gegeben werden. Weitere Adapter mit neuen Kommunikationstechnologien können separat entwickelt und hinzugefügt werden. Dabei muss das SPI nicht geändert werden, da es die Technologien und die dazugehörigen Adapter nicht kennt. Dies ist von grossem Nutzen für Applikationsentwickler, da sie lediglich wenige Klassen bzw. Interfaces kennen müssen. Zudem können über ein schlankes Application Programming Interface (API) auf die zur Verfügung gestellten Funktionalitäten zugreifen.

Die Tauglichkeit des Frameworks überprüfen wir mit einem Test-Client. Dieser Test-Client bietet die Möglichkeit, die verschiedenen Anwendungen durchzuführen und graphisch darzustellen. Auf die dabei verwendete Architektur des SPIs, die drei benutzten Middleware-Technologien und den Test-Client selber gehen wir weiter unten noch genauer ein.

Skype, MSN und ECF

Bestens bekannt sind Kommunikationsanwendungen wie Skype oder MSN. Sie bieten viele angenehme Funktionen, beruhen aber auf der klassischen Client-Server-Architektur des Internets. Das heisst, die Kommunikation nutzt vorwiegend das Hypertext Transfer Protocol (HTTP) und erfolgt über einen Server eines Dritten. Zudem müssen meistens beide Kommunikationspartner zur selben Zeit online sein.

Das sich in Entwicklung befindende Eclipse Communication Framework (ECF) [ECF06] kommt unseren Bedürfnissen näher. Dieses bietet ebenfalls eine Reihe von Kommunikationskanälen an, die direkt in Eclipse eingebunden werden können. So ermöglicht es beispielsweise der IRC-Provider (Internet Relay Chat) von ECF, Chat-Räume von Open-Source-Projekten direkt in der Eclipse- Umgebung zu betreten. Zudem können in ECF eigenständige Anwendungen auf der Rich-Client-Platform (RCP) entwickelt werden. Die Einschränkung für das ECF besteht darin, dass nur aus Eclipse heraus kommuniziert werden kann.

Cajo

Cajo [Caj06] ermöglicht eine Kommunikation zwischen mehreren virtuellen Maschinen und vereinfacht den Gebrauch von Java RMI. Es ist eine kleine (< 40 KByte), freie Java-Klassen-Bibliothek. Sie kann auf mobilen Geräten mittels der Java Micro Edition (JME) integriert werden.

Beim klassischen Java Remote Method Invocation (RMI) müssen die vom entfernten Objekt angebotenen Methoden zu Kompilationszeit in einem Interface definiert werden. Soll eine neue Methode angeboten werden, so muss das Interface neu kompiliert und verteilt werden. Interessant an Cajo ist, dass es keine zur Kompilationszeit explizit definierten Interfaces mehr benötigt. Objekte unbekannter Klassen können dynamisch zu Laufzeit mittels Java Reflection instanziert werden. Cajo stellt deshalb keine neuen Bedingungen an die Struktur von Anwendungen. Bestehende Software-Produkte können unverändert übernommen werden. Damit werden einfache P2P-Kommunikationsszenarien ermöglicht. Es sei jedoch darauf hingewiesen, dass das Auffinden der IP-Adresse von Peers vielleicht noch verbessert werden kann.

Java Message Service

Das API der Java Message Services (JMS) [JMS07] ermöglicht den Zugriff von Java-Programmen auf Nachrichtensysteme von Unternehmen. Es stellt eine Message Oriented Middleware (MOM) dar und bietet zwei verschiedene Kommunikationsarten: publish/subscribe oder Punkt-zu-Punkt-Messaging. Bei der ersten Variante wird die Kommunikation mit so genannten Topics gehandhabt. Man abonniert ein bestimmtes Topic und erhält alles, was von den andern an dieses Topic gesendet wird. Die zweite Variante arbeitet mit Queues. Eine Queue wird nur von zwei Punkten im System gleichzeitig benutzt. Ein User sendet eine Meldung an die Queue und der andere empfängt sie.

Die JMS-Schnittstelle benutzt die klassische Client-Server-Architektur und funktioniert asynchron. Dies ist zugleich Vorteil und Einschränkung dieser Technologie. Wenn eine Nachricht an ein Topic oder eine Queue gesendet wird, dann wird sie dort gespeichert und weitergesendet. Ist ein Benutzer online, so erhält er die Nachrichten sofort. Andernfalls wird sie ihm zugestellt, sobald er wieder online ist.

Das JMS-API wurde im Dokument [JMS02] standardisiert. Es gibt dazu sowohl offene als auch kommerzielle Implementierungen. In unserem Prototyp haben wir uns für das Open-Source-Produkt JBoss Application Server [JBoss06] entschieden, da wir die ganze J2EE-Umgebung aus einer Hand wollten.

JXTA

Der Name JXTA hat seinen Ursprung im englischen Wort juxtapose, was mit „nebeneinander stellen“ übersetzt werden kann. Die P2P-Architektur und die herkömmliche Server-Client-Architektur sollen nebeneinander existieren. JXTA dient der Standardisierung von P2P-Anwendungen durch offene Protokolle [JXTA07]. Dabei werden drei ehrgeizige Ziele verfolgt: Die Interoperabilität über verschiedene P2P-Systeme hinweg, die Plattformunabhängigkeit und die universelle Verbreitung, d.h. jedes Gerät kann auch Server sein.

Einige Möglichkeiten die JXTA anbietet, sind das Finden von Peers und Ressourcen, auch über Firewalls hinweg, der Austausch von Dateien mit jedermann, die Bildung eigener Gruppen von Peers

über verschiedene Netze hinweg und die sichere Kommunikation zwischen Peers über öffentliche Netze.

Service Providing Interface

Das Service Providing Interface (SPI) ist das zentrale Element unseres Kommunikations-Frameworks. Die Herausforderung besteht darin, das Interface einerseits so schlank wie möglich zu definieren und andererseits alle nötigen Anwendungsfälle („use cases“) für die drei vordefinierten Anwendungen abzudecken. Zudem soll es auch möglich sein, mit allen Technologie-Adaptern das Interface, so wie es definiert ist, zu implementieren.

In Abb. 2 erklären wir die Architektur unseres SPIs am Beispiel der Anwendung „File Sharing“ für den Technologie-Adapter Cajo. Das Interface FileServerModel, bietet folgende Anwendungsfälle an: Auflistung, welche Dateien auf dem Server freigegeben sind (getAllSharedFiles), Dateien freigeben (setSharedFiles) und den Server löschen (deleteServer).

Das Interface FileSharingModel erlaubt es, auf einem entfernten Server abzufragen, welche Dateien freigegeben sind (getRemoteFileList), eine Datei herunter zu laden (downloadFileFromServer) oder einen entfernten Benutzer für einen „Push“ anzufragen (requestFileTransfer). Auf diese Anfrage sind zwei Reaktionen möglich: Herunterladen der betreffenden Datei (downloadFile) oder eine Ablehnung (notAcceptFileTransfer).

Das SPI besteht aus dem Singleton ModelFactory, sowie für jede Anwendung aus einem Interface und einer abstrakten Klasse (z.B. FileServerModel und AbstractFileServerModel). Die Anwendung File-Sharing beinhaltet eigentlich zwei Fälle: Die Serverseite (FileServerModel) und die Clientseite (FileSharingModel). Deshalb sind in Abb. 2 zwei Interfaces und zwei abstrakte Klassen dargestellt.

Der Test-Client wählt über die Methoden der ModelFactory die Anwendung aus, z.B. „File Sharing“ über die Methode createFileSharingModel. Die gewünschte Middleware-Technologie wird beim Aufruf als String mitgegeben. In einem Property-File wird nachgeschaut, welche Implementierungsklasse (z.B. BCCajoFileSharinModel) entsprechend dem mitgegebenen Technologie-String über „ClassLoading“ geladen werden soll.

In der gleichen Abbildung ist ersichtlich, dass die technologiespezifischen Klassen (z.B. BCCajoFileSharinModel) der Anwendung (in unserem Fall der Test-Client) nicht bekannt sind. Dies wird durch Anwendung des Observer Patterns [Gam95] erreicht. Eine abstrakte Controller-Klasse im Test-Client (z.B. AbstractFileSharingController) implementiert das Interface Observer. Nachdem das Model eines Technologie-Adapters (z.B. BCCajoFileSharingModel) erzeugt worden ist, registriert sich der Controller beim abstrakten Model (z.B. AbstractFileSharingModel). Der Test-Client benützt also nur die Klassen des SPIs und nicht diejenigen der Technologie-Adapter.

Der Test-Client

Um die Tauglichkeit des SPIs zu verifizieren, haben wir einen einfachen Test-Client entwickelt. Am Beispiel „File-Sharing“ soll gezeigt werden, dass das SPI die für diese Anwendung benötigten use cases abdeckt.

Beim Starten hat der Benutzer die Möglichkeit, die Anwendung und die Middleware-Technologie zu wählen. Für die Anwendung „File Sharing“ erscheint das in Abb. 3 dargestellte Fenster.

Der obere Teil („File Server editing“) steht für die vom Benutzer selbst verwalteten Datei-Server. Die Anwendung ermöglicht es, einen neuen Server anzulegen, die freizugebenden Dateien zu verwalten oder einen bestehenden Datei-Server zu löschen. Die erstellten Server werden links in einer Liste angezeigt.

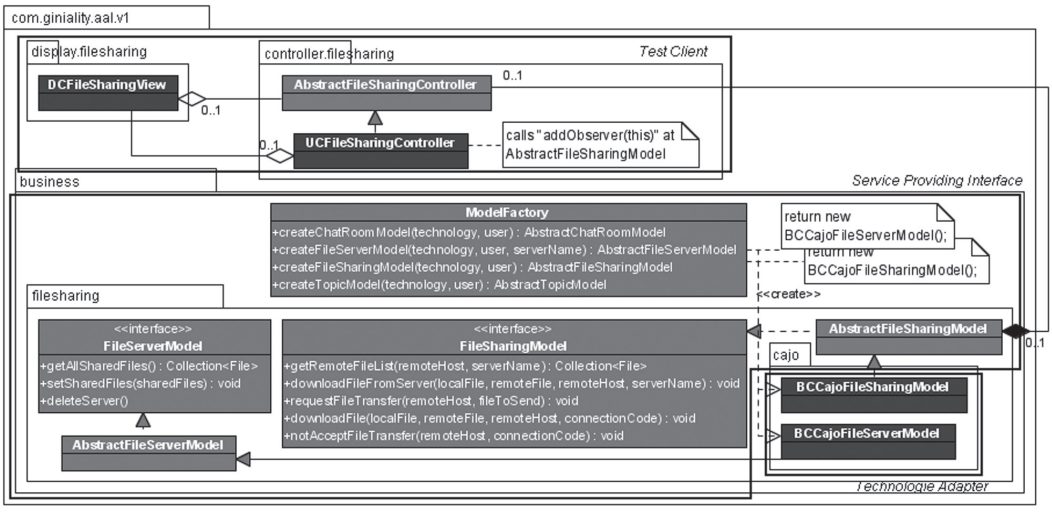

Abb. 2: Das Framework mit den drei Teilen Test-Client, Service Providing Interface (SPI) und den Technologie-Adaptern. Das SPI ist mit den Methoden für die Anwendung File-Sharing dargestellt.

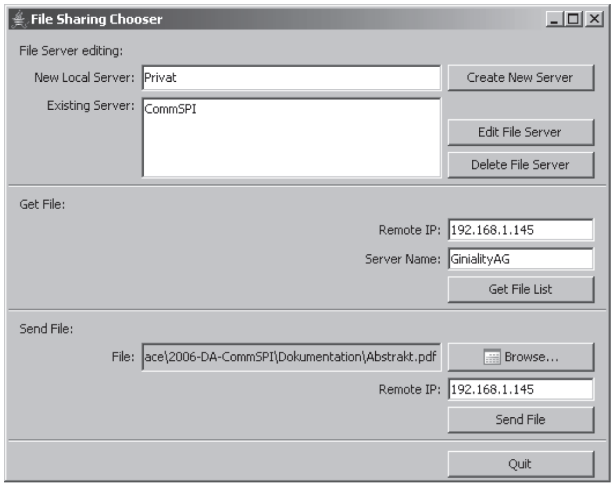

Abb. 3: Dialog-Fenster für die Anwendung „File Sharing“

Im mittleren Teil („Get File“) kommt die bewusst gewählte P2P-Architektur des Frameworks zum Ausdruck. Um von einem entfernten Benutzer freigegebene Dateien zu laden, muss man dessen IP-Adresse und den Namen des Servers kennen. Sind diese Informationen vorhanden, so kann man sich die Liste der freigegebenen Dateien anzeigen lassen und dann direkt einzelne Dateien herunter laden.

Im unteren Teil („Send File“) wird die Möglichkeit angeboten, einem entfernten Benutzer eine ausgewählte Datei zu senden („Push“). Dies geschieht jedoch nicht ohne Sicherheitsvorkehrungen: Die Datei wird nicht ohne Erlaubnis des Gegenübers gesendet und auf dessen Festplatte gelegt. Drückt der Sender den Knopf „Send File“, so wird zuerst eine Anfrage an den Empfänger gesendet, ob er die betreffende Datei von diesem Sender akzeptieren möchte oder nicht. Akzeptiert der Empfänger, so kann er zuerst den Speicherort festlegen und anschliessend wird die Datei übermittelt. Nach Abschluss der Datenübertragung wird der Empfänger informiert. Damit zwischen Anfrage und Datentransfer nichts manipuliert werden kann, erhält die Anfrage eine eindeutige Identität, die der Bestätigung und dem eigentlichen Transfer beigelegt wird. Stimmt diese Identität nicht überall überein, so wird die Übermittlung abgebrochen.

Alle hier beschriebenen use cases finden in den Interfaces FileServerModel und FileSharingModel des SPIs (siehe Abb. 2) eine entsprechende Methode.

Schlussfolgerungen und Ausblick

Mit unserem Test-Client zeigen wir, dass es möglich ist, ganz verschiedene Middleware-Technologien zu kapseln und vor dem Anwendungsentwickler zu verbergen.

Bei der Implementierung der Technologie-Adapter hat sich gezeigt, dass es schwierig ist, technologie-unabhängige Schnittstellen zu definieren. Cajo und JXTA sind synchrone Technologien, während JMS grundsätzlich asynchron funktioniert. Bei einem synchronen Aufruf kommt das Ergebnis der entfernten Aktion direkt als Rückgabewert der Methode zurück. Bei JMS muss auf das Ergebnis der entfernten Aktion gewartet werden. Dies stellt hohe Anforderungen an die Definition der Interfaces.

Als mögliche Weiterführung der Arbeiten soll geprüft werden, ob es sinnvoll und möglich ist, dass die Anwendung selber die unter gegebenen Umständen geeignetste Middleware-Technologie ermittelt. Ferner müssen für die P2P-Architektur Mechanismen entwickelt werden, die es dem Benutzer erlauben, auf einfache Art die IP-Adresse des gewünschten Kommunikationspartners zu ermitteln. Schliesslich wäre es wünschenswert, beim JMS-Adapter eine schlankere Lösung als JBoss, z.B. ActiveMQ [AMQ06], zu verwenden

Verdankung

Die Autoren bedanken sich bei den Herren Neudeck und Sauer der Firma Giniality für die ausgezeichnete fachliche Beratung während des Projektes.

Referenzen

[AMQ06] ActiveMQ, http://activemq.apache.org/

[Caj06] The cajo Project, https://cajo.dev.java.net

[ECF06] Eclipse Communication Framework, http://www.eclipse.org/ecf/

[Gam95] Erich Gamma et al., „Design Patterns: Elements of Reusable Object-Oriented Software“, Addison-Wesley, 1995.

[Gra02] J. Gradecki, „Mastering JXTA - Building Java Peer-to-Peer Applications“, Wiley Publishing, 2002.

[JBoss06] JBoss Application Server, http://labs.jboss.com/portal/jbossas

[JMS02] Java Message Service Specification, JMS API Version 1.1, April 2002, http://java.sun.com/products/jms/docs.html

[JMS07] Java message Service, http://java.sun.com/products/jms/index.jsp

[JXTA07] JXTA, http://www.jxta.org/