Mit Netz und doppeltem Boden

Sichere Integration von C++-Komponenten in eine Java-Enterprise-Anwendung

Andreas Senft

Bei der Ablösung größerer C++-Bestandsanwendungen durch eine Java-Enterprise-Anwendung ist eine „Big Bang"-Lösung oft nicht möglich. Eine stufenweise Migration erfordert jedoch eine sichere Anbindung von C++-Komponenten an die Java-Welt. Dazu gibt es eine verhältnismäßig einfache, aber dennoch sichere Lösung.

Problemstellung

Beim Ablösen eines Altsystems steht man oft vor dem Problem, dass nicht das gesamte System auf einmal ersetzt werden kann („Big Bang"-Lösung), sondern dass die Migration schrittweise durchzuführen ist. Hierbei ist zu beachten, dass Teile der neuen Anwendung und Teile der Altanwendung parallel produktiv sind und auch miteinander kommunizieren müssen. Es gibt prinzipiell verschiedene Möglichkeiten, einen solchen Fall anzupacken. Ein paar grundsätzliche Lösungen werden im weiteren Verlauf näher beleuchtet.

Kontext

In einem zurückliegenden Projekt ging es um den Fall, eine alte, aber umfangreiche C++-Anwendung abzulösen. Die bestehende Anwendung war bereits in die Jahre gekommen und es war nicht mehr möglich, notwendige Anpassungen mit vertretbarem Aufwand durchzuführen. Daher war eine Neuimplementierung auf Basis von Java vorgesehen, welche flexibler an neue Anforderungen angepasst werden konnte.

Der Umfang der Bestandsapplikation ließ es nicht zu, eine Komplettablösung in einem Schritt zu realisieren. Neue Anforderungen sollten möglichst bald umgesetzt werden, und dies möglichst basierend auf der neuen Anwendung. Die Neuimplementierung aller C++-Komponenten in Java hätte dafür allerdings zu lange gedauert. Daher sollten in einem ersten Schritt nur die Benutzungsoberfläche und einige ausgewählte Service-Komponenten in Java neu implementiert werden. Die restlichen Funktionalitäten sollten demgegenüber auf die bestehenden C++-Komponenten zurückgreifen. Spätere Versionen der Anwendung sollten diese Komponenten dann sukzessive durch Java-basierte Neuentwicklungen ersetzen.

Da die Anbindung der C++-Komponenten in die neue Java-Architektur als kritischer Punkt identifiziert wurde, wurden Lösungen evaluiert und ein Prototyp entwickelt, welcher die problematischen Aspekte behandelt und so die Machbarkeit der Unternehmung belegt. Im Folgenden werden betrachtete Lösungsansätze näher beleuchtet, insbesondere die durch den Prototyp umgesetzte Lösung sowie die zugrunde liegenden Ideen.

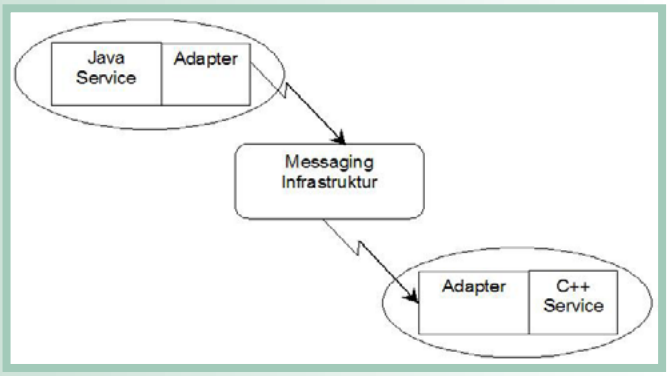

Abb. 1: Struktur einer Messaging-Lösung

Randbedingungen

Folgende Randbedingung waren bei der Ablösung der alten Anwendung zu beachten:

- Die als Ablösung zu entwickelnde Java-Anwendung soll auf Tomcat [Tomcat] lauffähig sein.
- Die C++-Komponenten werden nicht als stabil eingeschätzt, weswegen Sicherungsmaßnahmen ergriffen werden müssen, um die Java-Anwendungen vor Fehlern abzuschirmen.
- Die C++-Services sind zustandslos bzw. können durch geringfügige Änderungen zustandslos gemacht werden.
- Größere Anpassungen der C++-Komponenten sind zu vermeiden.
- Die C++-Services sind relativ grobgranular. Die Performanz der Anbindung von Java nach C++ ist daher nicht kritisch.
- Nach Möglichkeit sollten keine zusätzlichen Technologien eingeführt werden, welche ausschließlich in der Migrationsphase Verwendung finden.

Variante 1: Messaging

Die Verwendung von Messaging-Lösungen ist ein probates Mittel im Umfeld von EAI(Enterprise Application Integration)-Projekten. Hierbei werden Adapter für die beteiligten Systeme entwickelt, welche Service-Aufrufe und ihre Antworten in ein definiertes Nachrichtenformat überführen. Die Übermittlung dieser Nachrichten erfolgt dann über eine geeignete Middleware. Java-seitig kann hierzu auf den *Java Message Service* [JMS] zurückgegriffen werden, welcher unterschiedliche nachrichtenbasierte Systeme über eine standardisierte Anwendungsschnittstelle zugreifbar macht.

Eine Integration der Java-Anwendung mit bestehenden C++-Komponenten über eine Messaging-Lösung ist prinzipiell machbar. Die Umsetzung hätte entsprechende Adapter-Implementierungen sowie die Definition eines geeigneten Nachrichtenformats notwendig gemacht. Darüber hinaus wäre zusätzlicher Aufwand für Einrichtung und Wartung der Messaging-Infrastruktur angefallen.

Zwei Gründe sprachen nun gegen eine Message-basierte Lösung:

- Im Prinzip sind nur synchrone Aufrufe nötig. Die asynchrone Verarbeitung durch Messaging bringt also keine Vorteile.
- Tomcat biet von Haus aus keine JMS-Unterstützung. Obwohl diese natürlich integriert werden kann, war das Aufsetzen einer Messaging-Infrastruktur nur für den Zeitraum der Ablösung nicht gewünscht.

Daher wurde die Messaging-Variante verworfen und nach einer alternativen Lösung des Problems gesucht.

Variante 2: CORBA

Aus ähnlichen Gründen wie beim Messaging wurde die Verwendung von CORBA [CORBA] verworfen. Konkrete Negativpunkte waren:

- In Hinblick auf mögliche Instabilitäten der Komponenten erschien eine ausreichende Abschirmung vor Fehlern nicht gegeben.
- Die Kontrolle über die verfügbaren C++-Komponenten (Stichwort: Initialisierung bei Bedarf) ist eingeschränkt.
- Das Abbilden der vorhandenen C++-Services über IDLs ist unter Umständen aufwändig.
- Das Aufsetzen und die Wartung eines CORBA-Brokers nur für die Übergangszeit der Ablösung wurden als inadäquat betrachtet, insbesondere da im Projekt kaum Erfahrung mit CORBA vorhanden war.

Variante 3: JNI-Anbindung an autonomen C++-Prozess

Eine weitere Integrationsmöglichkeit stellt die Java-seitige Anbindung an einen autonomen C++-Prozess dar, welcher die aufzurufenden Service-Schnittstellen zur Verfügung stellt. Ein Vorteil dieser Lösung ist, dass sich die C++-Komponenten in einem eigenen Adressraum befinden und ein Absturz des C++-Prozesses nicht unmittelbar auch den Absturz des Java-Prozesses nach sich zieht.

Leider sind auch einige Nachteil dieser Lösung auszumachen. Um zwei Prozesse über das *Java Native Interface* [JNI] zu koordinieren, ist es nötig, die virtuelle Java-Maschine aus dem C++-Code heraus zu starten. Da die Java-Anwendung jedoch in einem Container laufen soll, ist dies nicht realisierbar. Und selbst bei gegebener Umsetzbarkeit wären Unwägbarkeiten wie Nebenläufigkeiten und Reaktion auf Abstürze eines Prozesses zu berücksichtigen.

Variante 4: Direkte Einbindung via JNI

Die direkte Anbindung der C++-Komponenten an die Java-Services über JNI stellt eine verhältnismäßig einfach zu realisierende Lösung dar. Die Java-Implementierung kann direkt auf die C++-Komponenten zugreifen und muss keine Probleme mit Nebenläufigkeiten berücksichtigen.

Der große Nachteil ist jedoch, dass die Verwendung von JNI im Kontext eines Applikationsservers nicht zu empfehlen ist. Einerseits kollidiert der Mechanismus zum Laden der nativen Bibliotheken mit den Classloader-Voraussetzungen der Applikationsserver. Andererseits würde ein Fehler im nativen Code unter Umständen den ganzen Applikationsserver zum Absturz bringen, was im Betrieb ein ernstes Problem darstellen würde.

Variante 5: Entkopplung durch einen Proxy-Prozess

Als notwendige Eigenschaft einer Lösung wurde das Auslagern des nativen Codes in einen separaten Prozess identifiziert. Ein anderer wichtiger Aspekt bezieht sich auf die Möglichkeit, dass der native Prozess unerwartet terminieren kann (z. B. durch einen Absturz). Dies kann dadurch behoben werden, dass ein zusätzlicher Proxy-Prozess eingeführt wird, welcher den aufrufenden Prozess vom aufgerufenen nativen Prozess abschirmt. Der Proxy-Prozess übernimmt die Verantwortung für das Starten des nativen Prozesses und delegiert alle Aufrufe an diesen weiter. Im Falle eines Absturzes kann der Prozess neu gestartet werden, ohne dass der Aufrufer davon betroffen ist.

Abbildung 2 verdeutlicht die Funktionsweise: Ein Service-Aufruf aus dem Client-Prozess heraus wird an den Service-Proxy im Proxy-Prozess gerichtet. Dieser Service-Proxy sorgt dann dafür, dass der Zielprozess, welcher die eigentliche native Implementierung bereitstellt, gestartet wird, falls dies noch nicht geschehen ist. Anschließend wird der Aufruf an den Service im Zielprozess weitergeleitet. Für den Aufrufer ist diese Delegation transparent.

Ausgehend von einem zustandslosen Service kann auch der Absturz des Zielprozesses durch einen Neustart ohne Wissen des Client-Prozesses korrigiert werden; lediglich die Antwortzeit ist durch den Prozessstart in diesem Fall etwas höher. Ein weiterer Vorteil dieser Proxy-Prozess-Lösung ist die vollständige Kontrolle über die Zielprozesse. So können Zielprozesse je nach Bedarf sofort oder erst bei tatsächlichem Bedarf instanziert werden und gegebenenfalls bei längerer Nichtverwendung auch wieder terminiert werden, ohne dass der Aufrufer davon etwas bemerkt. Dies erlaubt, die Anzahl der gleichzeitig aktiven Prozesse zu begrenzen und so mit den vorhandenen Ressourcen hauszuhalten.

Auch wenn die Etablierung eines zusätzlichen Prozesses gewisse Performanzeinbußen mit sich bringt, gewinnt man den Vorteil der totalen Abschirmung vor Fehlern. Insbesondere können Fehlerbehandlungsroutinen im Proxy-Prozess generisch umgesetzt werden, während etwa bei einer CORBA-Lösung spezifische Business-Delegates auf Client-Seite implementiert werden müssten. In Hinblick auf das Ansprechen der Services bietet die Verwendung eines zusätzlichen Prozesses auch den Nutzen, das Management der Zielprozesse vom Deployment/Redeployment in Tomcat zu entkoppeln. Auch können mehrere Tomcat-Instanzen auf denselben Proxy-Prozess zugreifen. Variante 5 diente als Grundlage für den Prototyp, welcher im Folgenden näher beschrieben wird.

Die Welt aus Sicht des Clients

Aus Sicht des Clients geht es prinzipiell darum, eine Service-Methode aufzurufen. Die Tatsache, dass der Aufruf an einen anderen Prozess delegiert wird, kann im Zeitalter von Dependency-Injection [DI] verborgen werden. Ein Objekt, welches das Service-Interface implementiert, wird der Client-Klasse über eine `set`-Methode oder in Form eines Konstruktorarguments injiziert. Bei Wahl eines geeigneten Remoting-Protokolls kann hierbei sogar ein ganz „normales“ Interface verwendet werden, welches keine protokollspezifischen Eigenschaften (wie es etwa ein RMI Remote Interface tut) aufweist. Als Beispiele seien das Http-Invoker-Protokoll oder das RMI-Invoker-Protokoll genannt, welche beide vom Spring-Framework [SPR] zur Verfügung gestellt werden.

Das andere Ende

Der Zielprozess beinhaltet im Wesentlichen den nativen C++-Code sowie dessen Kapselung über JNI. Da die direkte Benutzung von JNI jedoch mühsam und fehlerträchtig ist, wurde zur Unterstützung das frei verfügbare Werkzeug SWIG [SWIG] herangezogen, welches zu den C++-Klassen Java-Wrapperklassen generiert, welche intern wiederum JNI nutzen.

SWIG ist ein mächtiges Werkzeug. Es ermöglicht über eine Beschreibungssprache die gezielte Spezifikation der zu erzeu-

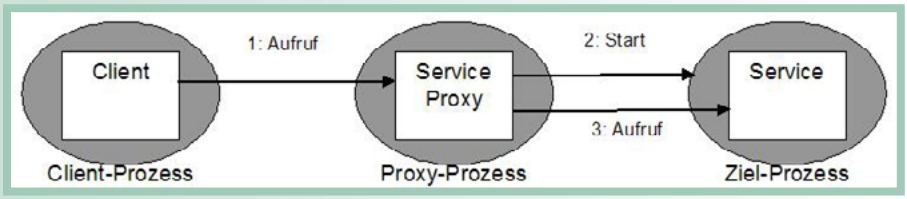

Abb. 2: Aufruf-Delegation durch den Proxy-Prozess

genden Wrapperklassen. Für einfache C++-Klassen kann SWIG jedoch auch direkt C++-Header-Dateien verarbeiten, was die Arbeit weiter beschleunigen kann. Die von SWIG erzeugten Klassen werden über einfache Java-Services aufgerufen, welche ein entsprechendes Service-Interface implementieren. Um diese Service-Prozesse nun zugreifbar zu machen, werden sie über das RMI-Invoker-Protokoll mit Hilfe des Spring-Frameworks exportiert. Der Start des Zielprozesses gestaltet sich denn auch einfach. Es wird ein Spring `ApplicationContext` erzeugt, welcher die Services instanziert und deren Export auslöst. Damit nicht im Zielprozess und im Proxy-Prozess dieselben Informationen mehrfach konfiguriert werden müssen, werden diese als Programmargument der `main`-Methode übergeben, wenn der Proxy-Prozess den Start des Zielprozesses initiiert.

Der Weg zum Ziel

Wie bereits angedeutet, wird der Aufruf des Clients nicht direkt an den Service im Zielprozess delegiert. Stattdessen kommuniziert der Client mit dem Service-Proxy im Proxy-Prozess. Da der Service-Proxy dasselbe Interface implementiert wie der eigentliche Service, ist dies völlig transparent. Der Proxy-Prozess ist als Spring-basierte Applikation implementiert. In einem `ApplicationContext` werden die Interfaces spezifiziert, welche zur Verfügung gestellt werden sollen. Für jedes Interface wird ein dynamischer Proxy (`java.lang.reflect.Proxy`) generiert und über das gewählte Remoting-Protokoll exportiert, sodass es vom Client aufgerufen werden kann. Der Export wird wiederum durch Remoting-Klassen des Spring-Frameworks unterstützt.

Die Implementierung des dynamischen Proxy tut nun zwei Dinge. Falls noch kein Ziel existiert, wird über den Aufruf von `java` ein Zielprozess instanziert. Dieser sorgt, wie oben beschrieben, selbsttätig dafür, dass der eigentliche Service über einen Remote-Aufruf verfügbar ist. Ist nun ein Ziel vorhanden, wird über Springs Remoting-Klassen der Aufruf einfach weiterdelegiert. Zur Veranschaulichung ist das Gesamtszenario in Abbildung 3 aufgeführt.

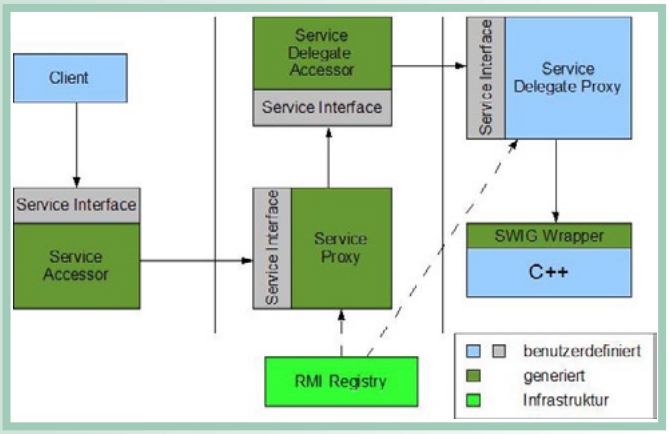

Abb. 3: Details zum Proxy-Mechanismus

Die farblichen Kennzeichnungen in Abb. 3 weisen darauf hin, welche Bestandteile der Applikation mit SWIG (über Tools) oder über das Spring-Framework (dynamisch zur Laufzeit) generiert werden. Ein wichtiger Aspekt ist, dass die Prozessteuerungs- und Fehlerbehandlungslogik innerhalb des Service-Proxy generisch umgesetzt ist und gegebenenfalls über Konfigurationseinstellungen gesteuert werden kann. Neben der Kapselung des nativen Service (der Service Delegate Proxy in Abb. 3) ist keine Implementierung erforderlich. Lediglich Konfigurationseinstellungen zum Proxying sind geeignet zu definieren.

Das große Ganze

Eine Web-Anwendung, welche diesen Mechanismus zur Anbindung nativen Codes nutzt, hätte demnach die in Abbildung 4 dargestellte Struktur. Im Sinne des Proxy-Prozesses stellt der Applikationsserver den Client dar, weshalb in dieser Abbildung der „Service“ dem „Client“ in vorherigen Abbildungen entspricht.

Weitere Herausforderungen

Während der Implementierung des Prototyps wurden weitere Aspekte identifiziert, welche für eine generellere Lösung relevant sind. Einerseits sind dies zustandsbehaftete Services: Falls der native Service einen Zustand hält, können nicht alle Anfragen an denselben Service durchgeleitet werden. Stattdessen muss für jeden Anwender explizit eine Session verwaltet werden, anhand derer ein dedizierter Prozess für jede Session ausgewählt wird. Sinnvoll anwendbar ist ein solcher Ansatz allerdings nur, falls die Anzahl der gleichzeitig aktiven Sessions überschaubar bleibt.

Außerdem ist die Kontrolle der erzeugten Prozesse zu berücksichtigen. Eine kontrollierte Terminierung eines Zielprozesses kann zum Beispiel mittels einer JMX-basierten Schnittstelle [JMX] ausgelöst werden. Problematisch ist jedoch der Fall, wenn der Proxy-Prozess selbst unerwartet terminiert. In dem Fall muss es nach einem Neustart ermöglicht werden, die Prozesse zu identifizieren, welche mit dem vorherigen Proxy-Prozess gestartet wurden. Möglichkeiten hierzu bietet etwa das Einbinden plattformabhängigen Codes zur Prozesssteuerung oder eine Prozesssuche, welche z. B. mit JMX realisiert werden könnte. Für die Überwachung und den automatischen Neustart des Proxy-Prozesses können Hilfsmittel wie zum Beispiel „Java Service Wrapper“ [JSW] zum Einsatz kommen. Dieses Tool ist für viele Plattformen (u. a. Windows, Linux, Solaris) verfügbar und erlaubt es, eine Java-Anwendung als Hintergrund-Prozess zu starten. Ein zusätzlicher Überwachungsprozess kommuniziert mit der Java-Anwendung und startet sie gegebenenfalls automatisch neu, falls keine Reaktion mehr erfolgt.

Fazit

Ein Proxy-basierter Ansatz zur Delegation an native Prozesse kann mit verhältnismäßig einfachen Mitteln umgesetzt werden. Durch den Einsatz von Open-Source-Software kann eine generische Infrastruktur geschaffen werden, welche das Rück-

grat des vorgestellten Ansatzes bildet. Die konkrete Anbindung der Services erfordert dann nur noch einen verhältnismäßig geringen Implementierungs- und Konfigurationsaufwand. Der vorgestellte Ansatz ist nicht als genereller Ersatz für Messaging oder CORBA gedacht, sondern stellt eine Alternative für Anwendungsfälle dar, in denen Fehlersicherheit und große Kontrollmöglichkeiten von Bedeutung sind.

Zum Abschluss hier noch eine Zusammenfassung der im Prototyp genutzten Features:

- Transparentes Remoting (Spring-Framework),
- Fernsteuerung der erzeugten Prozesse (JMX, Spring-Framework),
- Kapselung von Low-Level JNI-Code (SWIG),
- Prozesskontrolle und automatischer Neustart (JSW).

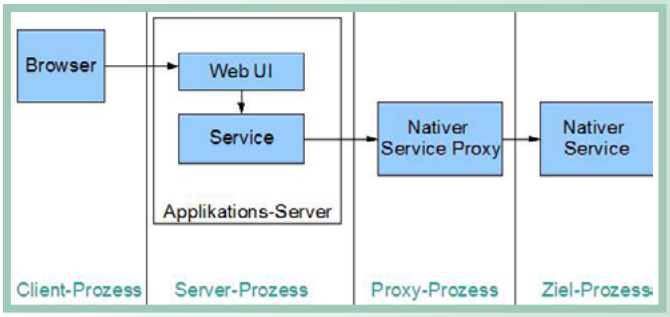

Abb. 4: Entkopplung durch einen Proxy-Prozess

Links

[CORBA] Common Object Request Broker Architecture, `http://www.corba.org`
[DI] Dependency Injection, `http://de.wikipedia.org/wiki/Dependency_Injection` (deutsch), `http://www.martinfowler.com/articles/injection.html` (englisch)
[JMS] Java Message Service, `http://java.sun.com/products/jms/`
[JNI] Java Native Interface, `http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html`
[JSW] Java Service Wrapper, `http://wrapper.tanukisoftware.org`
[SPR] Spring-Framework, `http://www.springframework.org/` und `http://static.springframework.org/spring/docs/2.0.x/reference/remoting.html` (Remoting)
[SWIG] Simplified Wrapper and Interface Generator, `http://www.swig.org`
[Tomcat] Apache Tomcat Servlet-Container, `http://tomcat.apache.org/`

Dipl.-Inform. (FH) Andreas Senft ist Berater und Entwickler im Java-Enterprise-Bereich bei der eMundo GmbH. Er verfügt über langjährige Erfahrung in objektorientierter Softwareentwicklung mit Schwerpunkten in Architektur und Design. Andreas Senft ist aktives Mitglied der Spring-Framework-Community. E-Mail: Andreas.Senft@e-mundo.de.