SOA für Multi-Messaging
Integrationskonzept am Beispiel Skype®

SOA for multi messaging -
an concept on integration with Skype ®

Fachhochschule Brandenburg
hiogi GmbH

1. Betreuer:	Prof. Dr. Jörg Berdux
2. Betreuer:	Prof. Dr. Thomas Preuß
Name:	Lars K.W. Gohlke
Abgabetermin:	August 2008

In Kooperation mit der hiogi GmbH.

Eingereicht an der Fachhochschule Brandenburg
Fachbereich Informatik und Medien.

Inhaltsverzeichnis

Inhaltsverzeichnis II

Danksagung III

1 Einleitung 1

2 Serviceorientierte Architektur 5

2.1 Was ist SOA? . . . 5
2.2 Service . . . 10
2.3 Servicevertrag . . . 14
2.4 Lose Kopplung . . . 18
2.5 Enterprise Service Bus . . . 26
2.6 Message-Exchange Pattern . . . 30
2.7 SOA-Entwurfsmuster . . . 35
2.7.1 Service Messaging . . . 36
2.7.2 Asynchronous Queuing . . . 37
2.7.3 Event-Driven Messaging . . . 39
2.7.4 Legacy Wrapper . . . 41

3 Enterprise Integration Pattern 45

3.1 Grundlegende Konzepte . . . 51
3.2 Nachrichtenendpunkte . . . 54
3.2.1 Ereignisgetriebener Konsument . . . 54
3.2.2 Konkurrierende Konsumenten . . . 56
3.3 Nachrichtenkanäle . . . 57
3.3.1 Nachrichtenbus . . . 57
3.3.2 Kanaladapter . . . 59
3.3.3 Punkt-zu-Punkt-Kanal . . . 60

3.3.4 Publish-Subscribe-Kanal 61
3.3.5 Garantierte Zustellung 63
3.4 Nachrichten 64

4 Referenzanwendung **67**
4.1 Architektur 72
4.2 Auflösung der Dienstabhängigkeiten 75
4.3 Robustheit und Skalierbarkeit 79

5 Zusammenfassung **87**

Tabellenverzeichnis **89**

Abbildungsverzeichnis **91**

Quellen- und Literaturverzeichnis **95**

Literaturverzeichnis **99**

Erklärung zur Diplomarbeit **101**

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, auf eine von der Form abweichende Art und Weise meinen persönlichen Dank folgenden Personen auszudrücken:

- Natürlich zuerst unseren Kooperationspartern der Hiogi GmbH, **Christoph Daecke** und **Martin Holbe**.
- Weiterhin meinen Betreuern, **Hrn. Prof. Dr. Thomas Preuß** und **Hrn. Prof. Dr. Jörg Berdux**, denen ich das gute Gelingen dieser Arbeit zu verdanken habe. Insbesondere Letzterem, welcher bis auf seinen dreiwöchigen Urlaub fast ganztags und vorrangig in den späten Abendstunden *ausnahmslos* über Skype zu erreichen war. Einfach klasse!

Dann möchte ich weiterhin bedanken bei ...

- ... **Hrn. Dr.-paed. Wolfgang Gohlke** – meinem aufopferungsvollen Großvater – als Wächter über den Kommateufel und Hüter des ordentlichen Ausdrucks.
- ... meiner liebevollen **Großmutter**, für das professionelle Catering und den besten Apfelkuchen.
- ... meinem **Vater**, ohne ihn wäre es nie zu dieser Diplomarbeit gekommen ;). Auch wenn du einmal *fast* über meiner Diplomarbeit eingeschlafen wärest.

- ... meinem Kommilitonen **Stefan Pratsch**, mit dem die praktische Zusammenarbeit während dieser Zeit sehr viel Spaß und Freude bereitet hat. Vor allem die gut gemeinten Fehlerhinweise nachts um 1 Uhr, kurz bevor *er* schlafen gegangen ist – danke, für die grauen Haare ;).
- ... allen, die hier unerwähnt bleiben.

Es hat Spaß gemacht ... und doch ist es schön es abgeschlossen zu haben.

1 Einleitung

Wir leben im Zeitalter der Informationen und Wissen ist Macht. Wissen ist nicht nur Macht, sondern bedeutet sehr oft auch einen Wettbewerbsvorteil gegenüber der Konkurrenz zu haben und damit die eigene wirtschaftliche Zukunft zu sichern. Dieser Vorteil ergibt sich nicht selten aus der Tatsache, einfach schneller als die Anderen reagiert zu haben. Die für diese Entscheidung notwendigen Informationen müssen deshalb ebenfalls schneller vorhanden sein. Wieso nutzen wir nicht dafür die Möglichkeit der verzögerungsfreien Sofortnachrichten? Diese etablierte einfache Technik kann Zeitschranken überwinden und Geschäftsabläufe beschleunigen. Es fehlt jedoch noch die Einbindung in eine zeitgemäße *serviceorientierte Architektur*, um so die Komplexität der Integration in bestehende Geschäftsabläufe auf ein Minimum zu reduzieren.

Das Thema dieser Arbeit zeigt die Verbindung moderner Instant-Messaging am Beispiel von Skype in eine serviceorientierten Architektur. Dabei wird der Autor darauf eingehen, worin die Motivation bestand und welche Aktualität dieses Thema besitzt.

Betriebliches Umfeld

Diese Diplomarbeit entstand im Rahmen einer Kooperation mit der hiogi GmbH. Die hiogi GmbH ist ein Startup-Unternehmen in Berlin. Das Ziel des Unternehmens besteht darin, eine Community aufzubauen und in diesem Zusammenhang personalisierte Werbung zu schalten. Hiogi – unter `www.hiogi.de` im Internet vertreten – bietet eine Wissensaustauschplattform an. Über diese können per SMS Fragen gestellt und von Mitgliedern der Community beantwortet werden. Der Fragesteller bewertet die Antwort. Wird diese vom Fragenden als gut bewertet, werden dem Beantworter Boni in Form von hiogi-Coins (virtuelle Währung) gutgeschrieben. Dabei

liegt das Hauptaugenmerk auf der Mobilität, gemäß dem Slogan *mobile question – mobile answer*.

Aufgabenstellung

Diese Arbeit beschäftigt sich mit der Erstellung einer Referenzimplementierung eines Multi-messaging-Dienstes mit Hilfe von Skype in einer serviceorientierten Architektur. Dabei sind im Vorfeld der Kooperation mit der hiogi GmbH folgende Anforderungen formuliert worden:

- Realisierung des IM Gateways zur Übermittlung von Nachrichten unter Zuhilfenahme der Skype-Kommunikationsplattform.
- Erstellung eines Skype-Buddy zum Empfangen von offenen Fragen aus der hiogi-Community.
- Erstellung eines personalisierten Skype-Buddy zum Einstellen von Fragen in die hiogi-Community zwecks Empfang einer Antwort und anschließender Ergänzungen.

Die Praxistauglichkeit der konzeptionellen Ansätze der Integration (unter Berücksichtigung der *Enterprise Integration Patterns*) werden anhand der Umsetzung aufgezeigt.

Dieses Beispiel zeigt, in welche Richtung die Entwicklung im Bereich des Instant-Messaging gehen wird. Dabei wird die zunehmende Verknüpfung und Einbettung einzelner Kommunikationsmittel in ein Gesamtsystem zur besseren Erreichbarkeit der Menschen angestrebt. Waren in den vergangen Jahren nur Emails und Webangebote die vorherrschenden elektronischen Kommunikationsmittel im Internet, so wird sich das in der Zukunft in Richtung Multi-Messaging verändern. Dazu gehören auch u.a. RSS-Feeds, SMS/MMS und eine Vielzahl von Instant-Messaging-Lösungen[1].

Die zunehmende Vernetzung vieler Unternehmensbereiche führt zu einem noch stärkeren Integrationsdrang, dem durch den Einsatz von serviceorientierten Architekturen Rechnung getragen werden wird und muss.

[1] Im anglo-europäischen Raum werden vornehmlich ICQ/YAHOO/MSN/IRC und Skype eingesetzt. Asien setzt in diesen Bereichen, bedingt durch seinen Entwicklungsstand, teilweise andere Akzente.

Dies wird n.u.M. nur gelingen, wenn die im Kapitel 2.1, *Was ist SOA?*, Seite 5 - 10, genannten Prinzipien und Anforderungen beachtet werden. Auf diese Weise wird es möglich sein, den Entwicklungstendenzen unserer Zeit zu entsprechen, einzelne Dienste zu entwickeln und anzubieten.

2 Serviceorientierte Architektur

Der Begriff *Serviceorientierte Architektur* (sehr oft mit *SOA* abgekürzt) wurde von dem Markforschungsunternehmen Gartner 1996 das erste Mal benutzt (Gar96a)(Gar96b). Seit 2004 ist die Welle der Begeisterung für das Thema SOA ungebrochen. Die Bedeutung dieses Begriffes wird allerdings selten präzise erklärt.

Wer sich mit diesem Thema beschäftigt, wird feststellen, dass SOA allzu oft mit *Webservices* und *SOAP* in einem Atemzug genannt wird. Bei den meisten Informationen über SOA wird sogar so weit gegangen, dass SOA über *SOAP* in Verbindung mit *Webservices* definiert wird[1] [2] [3] [4]. Was so nicht falsch ist, aber die Sicht auf die Dinge einschränkt und die Technologie auf eine Stufe mit der Philosophie hebt. Deswegen sei an dieser Stelle angemerkt, dass sich diese Technologien sehr gut für den Einsatz in einer SOA eignen, aber keinen alleinigen Anspruch erheben sollten, in diesem Architekturstil Verwendung zu finden. Sowohl *SOAP* als auch *Webservices* lassen sich ebenfalls **nicht**-serviceorientiert einsetzen.

2.1 Was ist SOA?

SOA könnte man (Rit06) zufolge als ein *Managementkonzept* für eine IT betrachten, die sehr schnell auf Veränderungen im Geschäftsumfeld reagieren kann und sich an Geschäftsprozessen orientiert.

Oder aber man betrachtet SOA als ein *Architekturkonzept*, das fachliche Dienste und Funktionen als Services bereitstellt.

[1] http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html
[2] http://www.dia-bonn.de/seminare/web_services.html
[3] http://www.jochen-vajda.de/download/soa_webservices_soap.pdf
[4] http://java.sun.com/developer/technicalArticles/WebServices/soa/

Weil die technische Sicht in dieser Arbeit Vorrang hat, soll SOA als ein Architekturstil gesehen werden, um Komponenten als grobkörnige autonome und lose gekoppelte Dienste verteilt nutzen und organisieren zu können.

Diese Dienste können aus mehreren Komponenten bestehen. Sie werden immer dynamisch zur Laufzeit gebunden und bezeichnen damit die Abkehr von enggekoppelten monolithischen Systemen[5].

Mit dem Ansatz einer serviceorientierten Umgebung soll flexibel auf sich dauernd verändernde Anforderungen im Geschäftsablauf reagiert werden können, damit die einzelnen Bestandteile des Geschäftsprozesses – die Services – immer wieder neu aufgestellt und angeordnet werden können. Die Neuaufstellung der Dienste setzt voraus, dass die Dienste spezifisch auf ihre Aufgaben zugeschnitten, jedoch so allgemein gehalten sind, dass sie sich einfach wiederverwenden lassen.

Der Begriff SOA hat einen überwiegend nicht-technischen Bezug. Aufgrund der Weitläufigkeit des Themas beschränken wir uns mit dem Hinweis auf diese Tatsache und stellen die architektonischen Aspekte ausdrücklich in der Vordergrund.

Die Ziele von serviceorientierten Architekturen sind *Flexibilität*, *Interoperabilität* und *Kostensenkung*. Die Flexibilität soll einerseits durch eine schnellere Anpassung der Geschäftsprozesse (Time-to-Market) und andererseits durch die Steigerung der Qualität durch Skalierbarkeit gemäß dem Leitspruch *„start small and grow fast"* erreicht werden. Die Senkung der Kosten soll durch die Wiederverwendbarkeit von Quellcode und Diensten erreicht werden. Dazu gehört auch die Anbindung bestehender Altanwendungen und externer Dienstleister.

[5] Diese Form der *gewachsenen* Architektur wird auch als *Stovepipe* (engl.: Ofenrohr) bezeichnet(RGO07, S.9),(Sti02) und ist eines von mehreren SOA Antimustern (ocg).
Stovepipe-Architekturen sind Systeme, die für ein ganz spezifisches Problem angeschafft und entwickelt wurden. Sie sind schwer erweiterbar und von begrenzter Funktionalität. Die dabei im System entstandenen und verwendeten Daten sind schwer zu exportieren (CtAtPPotDoEtD99).
Dieser Architekturtyp kann oft auch dadurch entstehen, weil Unternehmen fusionieren und verschiedene Systeme – u.a. Eigenentwicklungen, sogenannte *Inhouse-Lösungen* – miteinander verbinden wollen.

Für die Entwicklung der serviceorientierten Architektur haben sich Design-Prinzipen herausgebildet, die (Heu07) wie folgt benennt:

- *Schnittstellenorientierung*. Über die Schnittstellen sollen die Implementierungsdetails abstrahiert werden und analog zu einem Vertrag eine gewisse Stabilität besitzen. Die Eindeutigkeit wird durch die Spezifikation sowohl auf technischer als auch auf fachlicher Ebene abgesichert.

- *Interoperabilität*. Auf Grundlage der gesamten Architektur sollten Standardisierungsentscheidungen getroffen werden, die die fachliche und technische Heterogenität der Dienste überbrückbar machen. Dabei zu berücksichtigen sind sowohl gemeinsame Datenmodelle als auch eine einheitliche Fachsprache bzw. Terminologie. Die damit einhergehende Harmonisierung der Kommunikation ist unbedingt erforderlich für die Wiederverwendbarkeit der Dienste.

- *Autonomie und Modularität*. Im Gegensatz zu bestehenden Architekturen gruppiert SOA einzelne Aufgaben und Aufgabendomänen in klar abgegrenzte Dienste. Die Dienste können unabhängig voneinander, wenn sie nicht in einer Domäne mit anderen ihre Arbeit verrichten, benutzt werden. Die Aufteilung in Dienste folgt dem Prinzip der Softwareentwicklung „Teile und Herrsche“ bzw. der modularen Version des Prinzips „separation of concerns“ [6] [7] [8]. Dadurch entsteht auch u.a. die lose Kopplung.

- *Bedarfsorientierung*. Die Entwicklung hinsichtlich der Fähigkeiten der Dienste beschränkt sich auf die Anforderungen. Die Dienste sollen so performant wie möglich und so komplex wie nötig sein, um die Anforderungen an die Wartung so gering wie möglich zu halten und

[6] „*Separation of Concerns*: Dieses Konzept als ein Kernteil im Software-Engineering hat das Ziel, jene Teile eines Software-Systems zu identifizieren, zu kapseln und zu manipulieren, die relevant für eine bestimmte Angelegenheit oder Sache konkret zuständig sind. Solche Concerns sind eine primäre Motivation für die Zerlegung von Software in handhabbare und verständliche Teile (i.e. Dekomposition). Beispiele von Concerns sind Features, Aspekte, Rollen, Variation, Konfiguration etc.“ (DGH03, S.71).

[7] http://www.ibm.com/developerworks/webservices/library/ws-soa-practical/

[8] http://www.onjava.com/pub/a/onjava/2006/09/06/separation\discretionary{-}{}{}-of\discretionary{-}{}{}-concerns\discretionary{-}{}{}-in-web-services.html

die Auftretenswahrscheinlichkeit von Fehlern zu minimieren.

Der Einsatz einer serviceorientierten Architektur stellt einige Anforderungen an die Dienste und ihre Beschreibung. Folgende Aspekte sollten danach in Betracht gezogen werden (WH04):

- *Die Beschreibung des Dienstes.* Sie erfolgt unabhängig von seiner Implementierung, von seinem Ort oder von seiner Nutzung. Welche Aufgaben erfüllt er? Wie lange dauert eine Abarbeitung? Können Aufträge parallel oder nur sequentiell abgearbeitet werden?
- *Die Umsetzung und Hosting der Dienste als Anbieter.* Was wird angeboten? Wie wird es angeboten? Welche Sicherungsmaßnahmen hinsichtlich Zugangskontrolle und Erreichbarkeit sind umgesetzt?
- *Die Zusammensetzung von Diensten, bestehend aus anderen Diensten.* Sind es atomare Dienste, die unabhängig sind, oder bauen sie als Teil einer Verarbeitungskette auf anderen Diensten auf?
- *Die Unterstützung für synchrone, asynchrone und conversational Dienste.* Wie verläuft die Kommunikation - synchron oder asynchron? Muss darauf geachtet werden, in welchem Zustand sich der Dienst befindet?
- *Steuerung von Anwendungen, aufbauend auf Diensten und Regeln.* Wie sollen die Dienste verknüpft werden, entweder während der Laufzeit mittels einer *business rule engine*[9] oder statisch per Konfiguration vor dem Start der Anwendung?

[9] „Eine Business-Rule-Engine (BRE) ist eine technische Softwarekomponente als Bestandteil eines Business-Rule-Management-Systems (BRMS), die eine effiziente Ausführung von Geschäftsregeln bzw. Business-Rules ermöglicht. Das primäre Ziel der BRE ist es, die Geschäftslogik von der Programmlogik oder Prozesslogik zu trennen, was grundlegende Änderungen an der fachlichen Geschäftslogik ermöglicht(sic) ohne Änderungen am Programm-Code oder am Design des Geschäftsprozesses vornehmen zu müssen." (ocb)

- *Automatisierte Datenkonvertierung*. Welche Hilfsmittel werden gewählt, um die Heterogenität zwischen den verschiedenen Datenmodellen und -strukturen zu überwinden? Sollen die Transformatoren lokal in die Services integriert oder zentral an den *Enterprise Service Bus* (siehe Kapitel 2.5, S. 26) angeschlossen werden?

- *Erreichbarkeit lokaler und entfernter Dienste*. Welches Kommunikationsmedium stellt die problemlose Integration entfernter Dienste sicher? Welche technischen Mittel eignen sich, um eine geringe Latenz und eine hohe Skalierbarkeit und damit eine hohe Gesamtperformanz zu gewährleisten? Wird an genügend Risikomanagement im Rahmen der Erreichbarkeit der Services gedacht?

- *Unterstützung der Qualitätssicherung*. Mit SOA hält eine verteilte Umgebung Einzug. Damit verbunden sind zusätzliche Fragen, auf welche Weise z.B. die Qualität in einem verteilten System abgesichert werden kann. Wie müssen die Simulation, das Testen und Debuggen einzelner Dienste, ihrer Kompositionen und des Gesamtsystems erfolgen? Welche Integrationstests müssen neue Dienste bestehen?

Bestandteile einer SOA

Eine der Eigenschaften von SOA ist die Abbildung von Geschäftsprozessen durch die Komposition verfügbarer Dienste. Dabei lassen sich architektonische und strukturelle Komponenten definieren:

Die *architektonischen* Komponenten beschreiben die inhärente Funktionsweise einer SOA, dazu gehören:

- Ein lose gekoppeltes Komponentenmodell.

- Die Interaktion in Form von Diensten in vielen Client/Server-Modellen.

- Von jeder als Dienst gekapselten Funktionalität existiert nur eine Instanz.

- Die nachrichtenbasierte Kommunikation der Dienste untereinander.

- Die Dienste unterstützen sowohl die synchrone als auch die asynchrone Kommunikation.
- Die Kommunikation der Dienste ist in Schnittstellenvereinbarungen geregelt.

Die Bestandteile der *strukturellen* Komponenten sind hier im einzelnen erläutert (Abb. 2.1, S. 11):

- Der *Verzeichnisdienst*. Er gibt Auskunft über veröffentlichte Services und Anfragen von Dienstnutzern nach diesen mit einer Referenz auf die beim Dienstanbieter installierte Instanz.
- Der *Dienstanbieter*. Er bietet Dienstnutzern einen oder mehrere Services an und registriert diese bei einem oder mehreren Verzeichnisdiensten.
- Der *Dienstnutzer*. Er sucht bei einem Verzeichnisdienst nach einem Dienst. Dieser Verzeichnisdienst gibt dem Nutzer eine Referenz auf den Anbieter. Vor der Nutzung bindet sich der Dienstnutzer an den Dienstanbieter.
- Der *Servicevertrag*. Er legt die Rahmenbedingungen für die Nutzung des Dienstes fest (siehe Kapitel 2.3, S. 14).

Im Folgenden werden einige Begriffe eingeführt, die sich im Umgang mit SOA durchgesetzt haben. Sofern es dem Verständnis keinen Abbruch tut, wird auf die Übersetzung (aus dem Englischen, d.V.) bewusst verzichtet.

2.2 Service

Der Service ist ein Prozess, welcher eine Funktionalität anbietet. Er ist in der Regel *wiederverwendbar* ausgelegt, so dass er auch von anderen Diensten und Clients genutzt werden kann. Charakteristisch für einen Dienst ist weiterhin, dass er *grobkörnig* ist, selbst in kleinere Komponenten untergliedert sein kann und *nachrichtenorientiert* kommuniziert. Jeder Dienst besitzt nach außen eine feste technologieneutrale Schnittstelle und ist unabhängig von anderen Diensten nutzbar. Die Schnittstelle wird über eine Vereinbarung festgelegt. Im Gegensatz zu herkömmlichen Bibliotheken oder

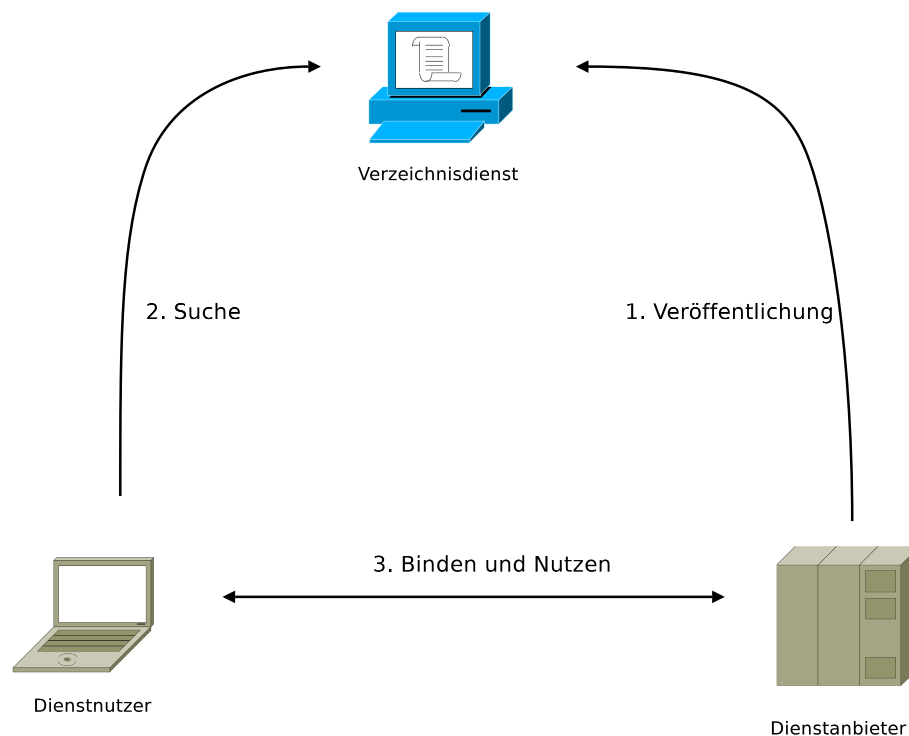


Abbildung 2.1: Die strukturellen Bestandteile von SOA (ohne Servicevertrag).

anderen Komponenten werden Services aufgerufen und nicht in die aufrufende Anwendung eingebettet.

Die Grobkörnigkeit gegenüber traditionellen Softwarekomponenten entsteht dadurch, dass in einem Service im Rahmen einer Aufgabe entsprechend mehrere Komponenten mit verschiedenen Spezialaufgaben integriert werden.

Die Dienste sind nur sehr lose gekoppelt. Diese lose Kopplung erfolgt einerseits durch die Kommunikation der Dienste untereinander mithilfe von Nachrichten, nicht jedoch über RMI bzw. RPC. Andererseits erfolgt sie durch den Einsatz von technologieneutralen Schnittstellen, die verschiedene Softwareplattformen und -implementierungen zulassen.

Arten von Services

In den SOA Blueprints (WH04) findet man eine Vielzahl von Benennungen für Dienste. Einige davon hat der Verfasser aufgegriffen und nach Struktur und Verhalten eingeteilt.

Strukturelle Dienste sind *Component Services* und *Composite Services*. *Component Services* sind atomare Dienste, die keine anderen Dienste zur Abarbeitung in Anspruch nehmen.

Als Beispiel könnte man sich einen Logging Service vorstellen, der jede entgegengenommene Zeile mit Zeitstempel in eine Datei schreibt. Typischerweise sind keine anderen Dienste an der Abarbeitung der Funktion beteiligt (Abb. 2.2).

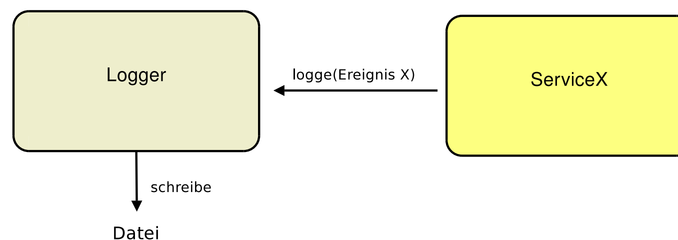


Abbildung 2.2: Ein einfacher Dienst, der Ereignisse in einer Datei mitloggt.

Auf einem höheren Abstraktionsniveau koordinieren die *Composite Services* das Zusammenspiel von *Component Services*. Der Verbund aus vielen *Component Services* ermöglicht die Abbildung komplexer Arbeitsabläufe.

So benötigt ein Anmeldevorgang meist eine Vielzahl kleinerer Teilschritte, die in einem Verbund gekapselt werden können. Nach außen wird nur ein Teilschritt sichtbar sein. Intern werden mehrere Dienste eingebunden. Ein Dienst z.B., der die Datenbank befragt – ein Dienst, der die Benachrichtigungsmail verschickt – ein Dienst, der den ganzen Vorgang begleitet und das Webportal anbietet (Abb. 2.3, S. 13).

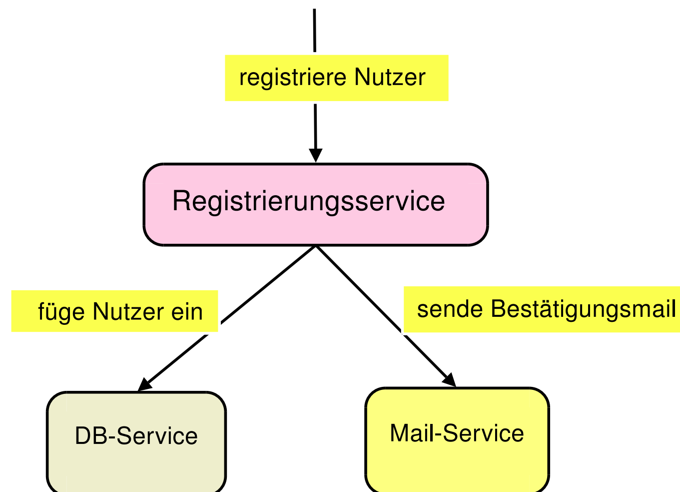


Abbildung 2.3: Ein Registrierungsdienst, aufgespaltet in Teildienste.

Granularität	SOA	Beschreibung
fein	Component Service	kleine spezifische Lösung
grob	Composite Service	komplexe umfassende Lösung, Verbund von Diensten

Tabelle 2.1: Übersicht der strukturellen Einteilung von Dienstarten in einer SOA.

Eine andere Einteilung der Dienste betont den Verhaltenstyp (WH04, vgl.):

- Data Service.
- Service Broker.
- Workflow Service.

Ein *Data Service* kapselt die Persistenzschicht. Auf einem nachrichtenbasierten Frage-Antwort-Mechanismus beruhend, werden Daten aus einer oder mehreren Quellen angeboten. Dabei bleibt dem Nutzer verborgen, wo sich die Daten befinden und wie darauf zugegriffen wird. Dem Dienst

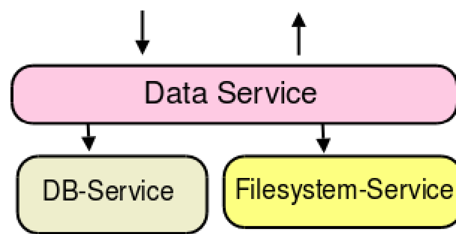


Abbildung 2.4: Kapselung der Persistenzschicht durch einen Data Service.

obliegt es, die Anfragen zu den Daten weiterzuleiten und in einem korrekten Nachrichtenformat zurückzugeben (Abb. 2.4).

Der *Service Broker* reicht auf der Grundlage von Regeln Nachrichten an unterschiedliche Dienste weiter. Falls notwendig, müssen die Nachrichten für diese Dienste aufbereitet werden (Abb. 2.5).

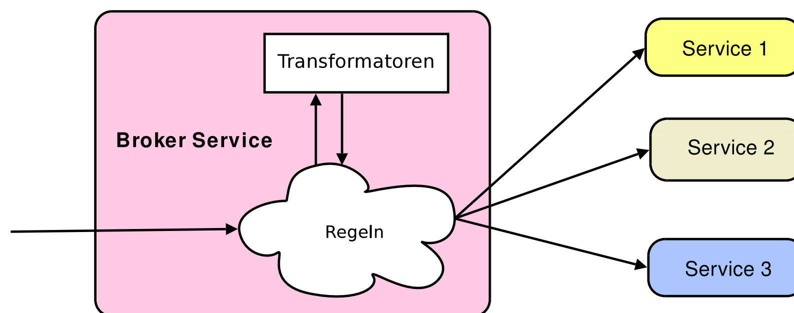


Abbildung 2.5: Der Service Broker verteilt die Nachrichten.

Der *Workflow Service* bzw. auch *Conversational Service* genannt, ist eine Zustandsmaschine, deren Status sich infolge von Anfragen verändert. Eine Zustandsmaschine definiert immer einen Anfangs- und Endzustand. Man beginnt mit einer vereinbarten Anfrage, die den Dienst in einen bestimmten Zustand versetzt. Der finale Zustand kann durch eine oder mehrere Anfragen erreicht werden.

2.3 Servicevertrag

Jede Schnittstelle braucht eindeutige Regeln dafür, wie sie zu benutzen ist. Klar muss sein, welche Informationen in welchem Format angeboten

werden sollen und wie ihr Austausch erfolgt, nämlich einseitig mit *mehreren* Diensten oder mit *einem* anderen Dienst.

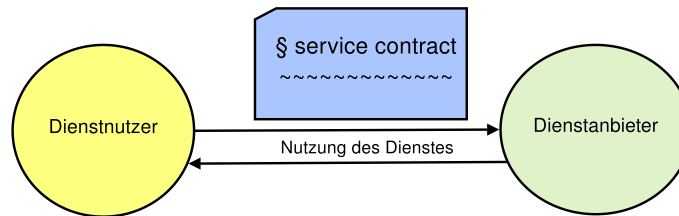


Abbildung 2.6: Beziehung zwischen dem Nutzer und Anbieter eines Dienstes.

(Erl07) beschreibt es folgendermaßen:

Die Vereinbarung für einen Dienst beinhaltet die technischen Voraussetzungen für den Einsatz, die angebotene Funktionalität und die öffentlich zugänglichen Informationen.

Ausgehend vom Prinzip *design-by-contract*[10] von Bertrand Meyer wird die Schnittstellenvereinbarung auch *service contract* genannt.

Design-by-contract „ist ein Konzept aus dem Bereich der Softwareentwicklung. Ziel ist das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler Verträge zur Verwendung von Schnittstellen, die über deren statische Festlegung hinausgehen.“[11]

Die Idee stammt aus der Geschäftswelt: In einer Kundenbeziehung sichern sich Anbieter und Kunde Obligationen zu. Der Kunde zahlt eine bestimmte Summe und der Anbieter erfüllt im Gegenzug sein Angebot. Dieses Konzept wird nun auf SOA übertragen.

Jeder Service erwartet vom Nutzer, dass dieser gewisse Vorbedingungen erfüllt. Bei einem Webshop z.B. soll der Warenkorb nicht mehr als 100 Artikel enthalten. Erfüllt der Nutzer diese Bedingungen, versucht der Service sein Angebot und die Nachbedingungen zu erfüllen. Die Nachbedingungen werden nach Ausführung der Arbeit geprüft und sollen dem Nutzer die Sicherheit geben, dass sein Auftrag erfolgreich erledigt wurde. Ein Beispiel für die Nachbedingung könnte der Vergleich der Lieferpläne im Lager sein. Falls eine der Vor- oder Nachbedingungen nicht erfüllt ist –

[10] http://www.gruntz.ch/courses/sem/ws06/DBC.pdf
[11] http://de.wikipedia.org/wiki/Design_by_contract

Bedingungen sind in beiden Fällen mit dem logischen UND verknüpft – schlägt die gesamte Prüfung fehl. Sollte die Prüfung der Vorbedingungen scheitern, liegt ein Fehler im Aufruf vor. Sollte jedoch die Prüfung der Nachbedingungen scheitern, handelt es sich um einen Fehler im Service. Das Ziel dieses Konzepts besteht darin, den Ablauf einer Dienstnutzung ständig eindeutig zu bestimmen. Übertragen auf SOA würde man jedem Dienstnutzer bei einem Fehler einen Hinweis über dessen Ursache geben und dabei anmerken, ob es sich um einen Nutzungs- oder einen Bearbeitungsfehler handelt (Abb. 2.7).

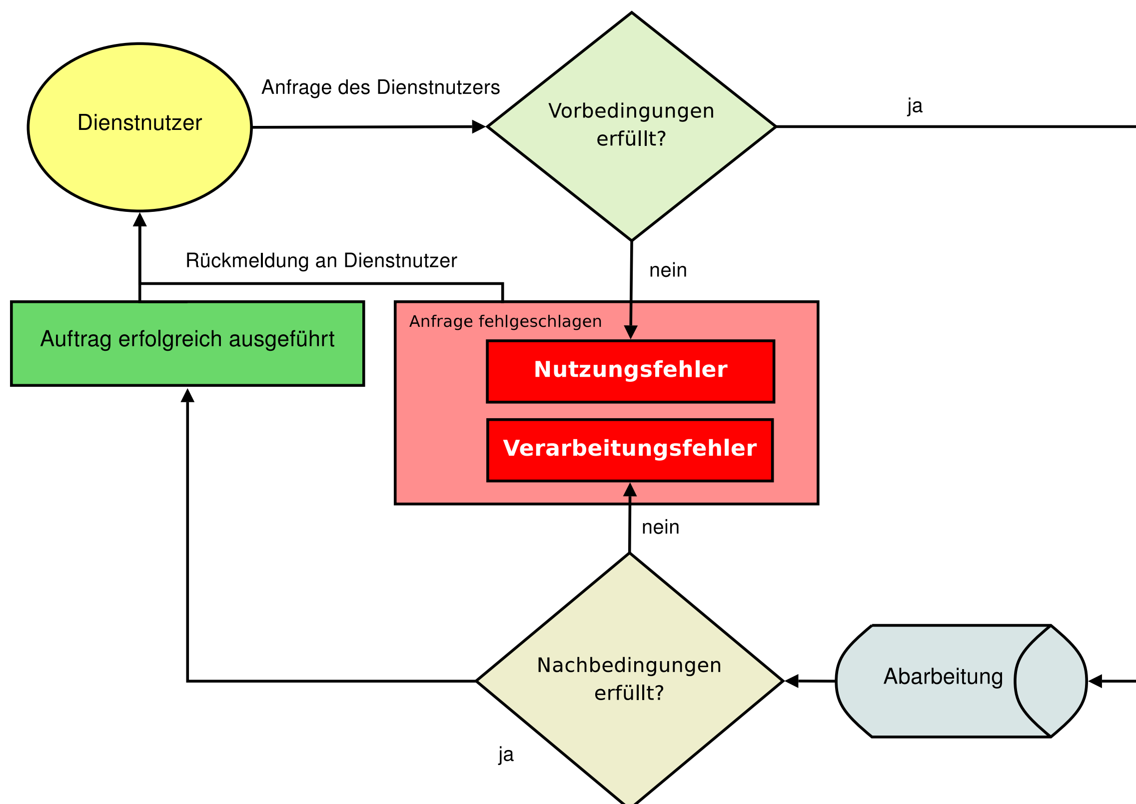


Abbildung 2.7: Die schematische Darstellung des *design-by-contract* Aufrufs eines Dienstes.

Die Schnittstellenvereinbarung stellt eine der zentralen Prinzipien bei SOA dar. In (ocf) werden folgende Angaben für einen Servicevertrag als sinnvoll erachtet:

Für alle Dienste sollten unabhängig von ihrem Zweck und ihrer Funktionalität Name, Version und Eigentümer beschrieben werden. Zusätzlich sollte geklärt werden, wer die Einhaltung des Vertrags überwacht und wer die Befugnis zu seiner Änderung hat. Bei Änderungen der Vereinbarung sollte festgeschrieben werden, wer unterrichtet wird, um entsprechende Hinweise zu geben und sie in die Entwicklung einfließen zu lassen. Dabei sollte immer darauf geachtet werden, auch eine Bestätigung zu erhalten, um Fehler durch Missverständnisse auszuschließen. Auch sollte nach einer Änderung klar sein, wer darüber informiert wird, dass diese Entscheidung getroffen wurde. Das sind dann meist die Nutzer des Dienstes bzw. die öffentliche Stelle für die Servicebeschreibung. Weiterhin sollten auch der Typ bzw. die Rolle des Service bestimmt und klar sein, auf welcher Schicht dieser Dienst agiert. In Frage kämen Business-, Präsentations-, Daten-, Integrationsschicht, Prozessebene u.a.. Nun folgen Ausführungen, die individuell verschiedene Ausprägungen haben können. Diese werden nach *funktionalen* und *nicht-funktionalen* Aspekten unterschieden.

Funktionale Aspekte beinhalten:

- *Die funktionalen Anforderungen.* Diese sind im Pflichtenheft festgeschrieben.

- *Die Serviceoperationen.* Das sind Aktionen, die ein Service ausführt, mit welchen Methoden er arbeitet und welche Parameter von ihm erwartet werden (die Vorbedingungen).

- *Den Serviceaufruf.* Über welche URL wird der Dienst erreicht, welche Schnittstelle, welche Protokolle (z.B. SOAP, HTTP, JMS u.v.m.) und Arten der Kommunikation (synchron, asynchron, ereignisgetrieben) werden vom Dienst genutzt.

Nicht-funktionale Aspekte beinhalten:

- *Die Sicherheit.* Wer ist berechtigt, den Dienst auszuführen, wer darf welche Operationen des Dienstes nutzen?

- *Die Qualität des Dienstes.* Sie legt die erlaubte Ausfallzeit des Dienstes fest.

- *Die Transaktionsfähigkeit.* Kann der Dienst größere Operationen als Transaktion zusammenfassen und wie wird das gesteuert?
- *Die Dienstgütevereinbarung*[12]. Diese beinhaltet sowohl Angaben über die Reaktionszeit des Dienstes als auch über den Umfang der Operationen, welche in der Vereinbarung enthalten sind.
- *Die Terminologie.* Wichtige Begriffe zur Harmonisierung der Sprache und des Verständnisses werden geklärt und festgelegt.
- *Die Beschreibung des Prozesses.* Welche Leistungen erbringt der Dienst?

2.4 Lose Kopplung

Eine der Kerneigenschaften von SOA ist die *lose Kopplung*. Die Autoren Stefan Tilkov, Marcel Tilly, und Hartmut Wilms beschreiben lose Kopplung als "(...) *einen Architekturansatz, bei dem die Abhängigkeiten zwischen Dienstanbietern und Dienstnutzern auf ein Minimum reduziert werden. Kopplung kann in mehreren Dimensionen erfolgen, zum Beispiel zeitlich, örtlich, in Bezug auf Syntax und Semantik, bestimmte Technologien*"(TTW05).

Kopplung als Eigenschaft von Software beschreibt die wechselseitige Abhängigkeit diverser Komponenten. Sie wird in der Regel nach eng bzw. stark und lose eingeteilt. Niedrige Kopplung wird auch als *lose Kopplung* bezeichnet.

Die Wikipedia bezeichnet lose Kopplung als eine Herangehensweise, bei der die Schnittstellen unter minimalen Annahmen über die sendenden bzw. empfangenden Teilnehmer entwickelt werden (ocd). Folglich wird das Risiko, wodurch eine Veränderung in einem Modul Änderungen in einem anderen Modul nach sich zieht, minimiert.

Lose Kopplung ist also eine wesentliche Eigenschaft von SOA. Allerdings erkauft man sich damit eine zusätzliche Abstraktion und naturbedingt auch eine höhere Komplexität, die wiederum zusätzlichen Aufwand für

[12] engl.: Service Level Agreement, mit SLA abgekürzt

die Entwicklung und Pflege bedeutet.

Die lose Kopplung ist gekennzeichnet durch:

- **Design-Unabhängigkeit** der beteiligten Softwarebestandteile sowohl *strukturell* als auch hinsichtlich des softwareentwicklungstechnischen Ziels von *separation of concerns*.
- **Schmale Schnittstellen.** Es gibt nur wenige Datenelemente, die über die zur Verfügung stehenden Schnittstellen ausgetauscht werden.
- **Geringen Schnittstellenverkehr.** Die Häufigkeit von Datenaustausch über die Schnittstellen zwischen Diensten zur Abarbeitung eines Vorgangs ist gering.

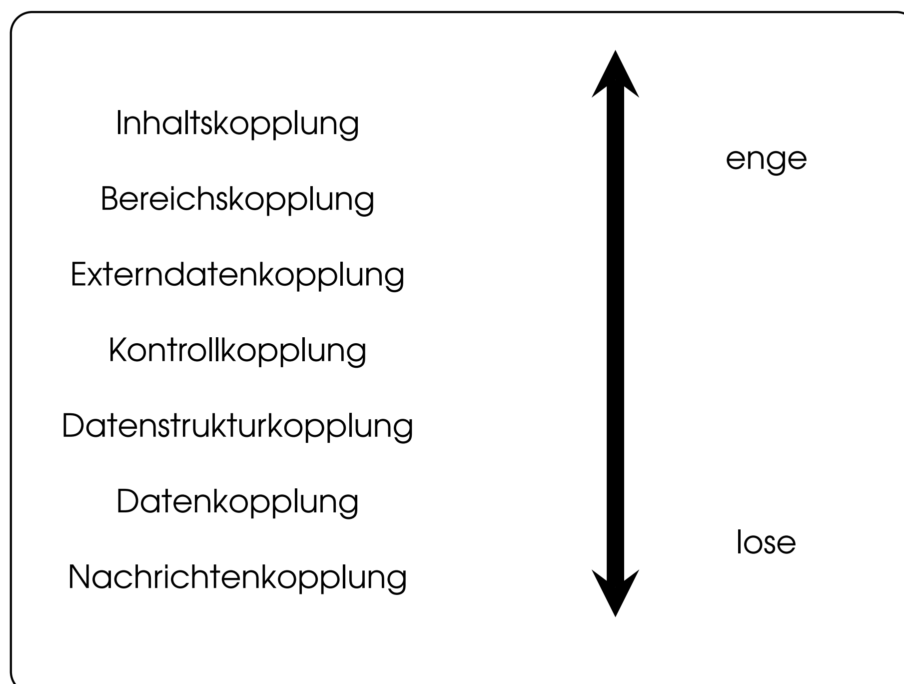


Abbildung 2.8: Arten und Stärke der Kopplung.

Der Grundgedanke loser Kopplung besteht darin, die Abhängigkeiten zwischen den Komponenten aus Effizienzgründen zu minimieren. Lose Kopplung ist das vorherrschende Prinzip moderner Integrationslösungen bei Geschäftsprozessen, um Fehlertoleranz und Skalierbarkeit zu erzielen (Abb. 2.9, S. 20).

Mit der Nutzung von loser Kopplung eröffnen sich nicht nur lang erwartete Möglichkeiten und Perspektiven, es treten auch eine Reihe von Problemen auf, die an dieser Stelle nicht verschwiegen werden sollen.

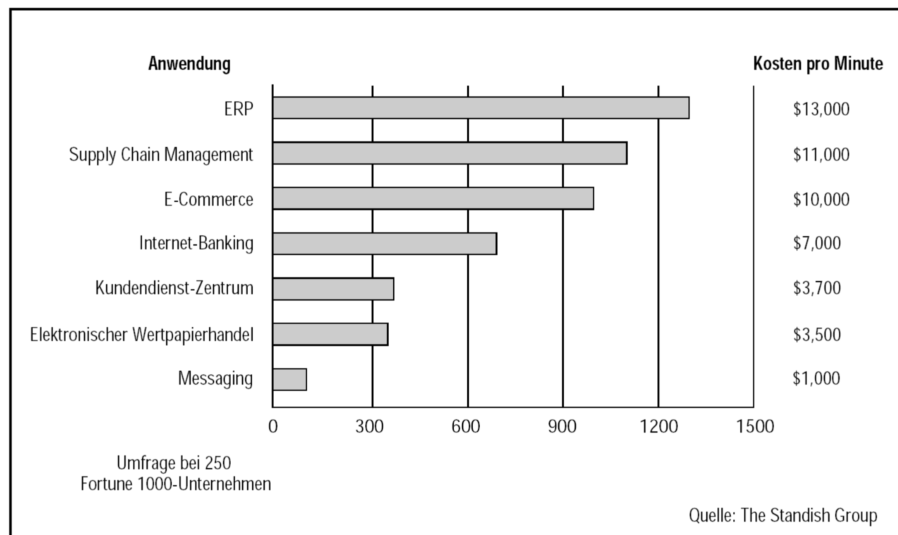


Abbildung 2.9: Ausfallkosten pro Minute aus einer Umfrage der Standish Group (vgl. (Sys00)).

Der größte Nachteil und zugleich der entscheidende Vorteil ist die höhere Abstraktion, welche dazu führt, dass die Komplexität steigt und somit das System unüberschaubarer wird. Beim Auftreten von Fehlern kann es sehr viel schwieriger sein, die Zusammenhänge zu erkennen und Kausalketten nachzuvollziehen, als dass in enggekoppelten Systemen der Fall ist. Aus diesem Grund wird die technische Testbarkeit aufwändiger, da einzelne Modultests nicht mehr ausreichen und Überprüfungen ganze Arbeitsabläufe in vielen Lastsituationen erfassen müssen. Bei schlechter Konzeption (z.B. hohem Schnittstellenverkehr und den damit nötigen vielen Indirektionsstufen[13]), kann die Performanz sehr leicht einbrechen.

Schauen wir uns ein Beispiel an, bei dem schlecht optimierte Kommunikation zu einem Flaschenhals und zu einem Leistungsengpass führen kann. Wir haben einen Dienst und einen Nutzer, die über eine unsichere Leitung miteinander Nachrichten austauschen. Deshalb muss für Sicherheit

[13] Dazu gehört u.a. der verkettete Aufruf von Kleinstmethoden, weil dabei das Umkopieren des Speichers ins Gewicht fällt.

gesorgt werden. Zum einen ist ein sicherer Zugang zum Nachrichtenkanal, dem Enterprise Service Bus (siehe Kapitel 2.5, S. 26), erforderlich. Dafür nutzen wir SSL auf der Transportschicht. Zum anderen müssen wir auch sicherstellen, dass niemand in diesem Nachrichtenkanal unsere Daten liest und verändert. Bisher ist nur der Weg *zum* Nachrichtenkanal sicher, im Kanal jedoch erscheinen die Daten im Klartext. Um auch dort die Nachrichten verschlüsselt übermitteln zu können, verwenden wir zusätzlich eine Sicherung auf der Nachrichtenschicht, indem wir die Nutzdaten der Nachrichten kryptographisch kodieren. Das sind die Rahmenbedingungen für den Datenaustausch des Dienstes und des Nutzers untereinander. Für den Fall, dass sehr viele kleine Nachrichten ausgetauscht werden, kann man von einer Verzögerung beim Absichern der Nachricht beim Nutzer und während des Verschickens zum Enterprise Service Bus ausgehen. Das gilt auch für den Dienst, der die Daten wiederum entschlüsseln muss. Sollte die Verschlüsselung zu stark und/oder die Häufigkeit des Nachrichtenaustauschs zu groß sein, können diese notwendigen Sicherungsmaßnahmen zu einer erheblichen Beeinträchtigung im Betriebsablauf führen.

Mithilfe dieser Art der Kopplung ist es möglich, Entwicklungsprozesse erheblich zu beschleunigen. Da nur wenige Voraussetzungen nötig sind und wenige Annahmen gemacht werden müssen, lässt sich nach Vereinbarung der Schnittstellen die Entwicklung sehr gut parallelisieren. Die entwickelten Komponenten lassen sich autonom betreiben und können leicht ausgetauscht werden. Dadurch werden Zeit und Kosten bei der Herstellung und Wartung von Software eingespart. Es wird eine schnellere Time-to-Market[14] erreicht. Daraus kann ein Wettbewerbsvorteil entstehen.

Bei korrektem Einsatz verspricht lose Kopplung eine sehr gute Skalierbarkeit. Damit ist die Fähigkeit gemeint, auf Schwankungen des Lastverhaltens und Veränderungen des Funktionsumfangs einzelner Teile mit vergleichsweise geringem Aufwand reagieren und umgehen zu können (Kapitel 4.3, S. 79f.).

[14]Beschreibt die Dauer von der Produktentwicklung bis zur Platzierung des Produkts am Markt.

	Enge Kopplung	Lose Kopplung
Physikalische Verbindung	Punkt-zu-Punkt	über Vermittler
Kommunikationsstil	synchron	asynchron
Datenmodell	komplexe gemeinsame Typen	nur einfache gemeinsame Typen
Typsystem	streng	schwach
Bindung	statisch	dynamisch
Plattformspezifika	stark / viel	schwach / wenig
Interaktionsmuster	über komplexe Objektbäume navigieren	datenzentrierte autonome Nachrichten
Kontrolle fachlicher Logik	zentrale Kontrolle	verteilte Kontrolle
Deployment	gleichzeitig	zu verschiedenen Zeitpunkten
Versionierung	explizite Upgrades	implizite Upgrades
Abhängigkeiten	viele enge	wenige explizite
Performanz	schlechter	besser
Ausfallsicherheit	schlechter	besser
Komplexität	geringer	höher

Tabelle 2.2: Nicolai Josuttis listet in (Jos07) diese Unterschiede zwischen enger und loser Kopplung auf.

Formen loser Kopplung

Asynchrone Kommunikation. Die bekannteste Ausprägung loser Kopplung ist die asynchrone Kommunikation. Dabei ist es möglich, Sender und Empfänger zeitlich zu entkoppeln, so dass nicht beide zeitgleich erreichbar sein müssen. Grundsätzlich können Dienste in zwei Modi miteinander kommunizieren: synchron und asynchron. Dabei kommt es darauf an, wie verlässlich der Dienst ist und wie lange der Aufruf dauert.

Ein **synchroner** Aufruf gibt direkt nach der Abarbeitung dem Anfragenden eine Antwort zurück und blockiert somit den Aufrufer für die Dauer der Abarbeitung (Abb. 2.10, S. 23). Sollte sich ein Aufruf über viele Dienste er-

strecken, kann es zu einer Blockierung kommen - im extremen Fall sogar zu einem *Deadlock*.

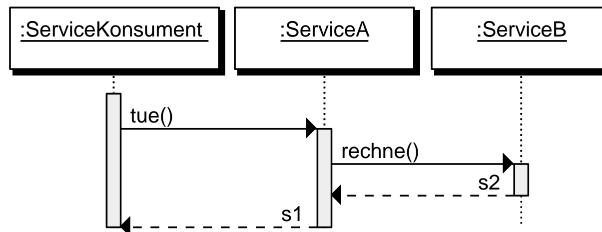


Abbildung 2.10: Synchron arbeitende Services.

Bei der **asynchronen** Kommunikation folgt dem Aufruf nicht sofort eine Antwort. In einigen Fällen erfolgt eine Bestätigung des Aufrufs. Die eigentliche Antwort wird zeitlich entkoppelt und später durch eine Callback[15] oder anderweitigen Mechanismus verschickt. Um die Nachrichten zuordnen zu können, müssen diese korreliert werden. Die Korrelation erfolgt im einfachsten Fall durch Sequenzzähler im Kopf der Nachricht.

Bei der asynchronen Kommunikation wird unterschieden zwischen der senderseitigen und der empfängerseitigen Asynchronität. Beim Sender verhindert die Asynchronität die Blockierung für die Zeit, in der die Antwort generiert wird. Der Empfänger versteht unter Asynchronität den Einsatz einer Nachrichtenwarteschlange, die die Nachrichten zwischenpuffert. Als Beispiel sei hier ein Enterprise Service Bus genannt, der die entkoppelte Kommunikation über Warteschlangen anbietet.

Der Vorteil besteht darin, dass die teilnehmenden Komponenten zeitlich entkoppelt sind. Es ist allerdings ein höherer Aufwand notwendig, um den korrekten Ablauf der Kommunikation zu überwachen. Durch den Zeitversatz kann die Reihenfolge der eintreffenden und der erwarteten Nachrichten unterschiedlich sein. Zur Lösung dieses Problems verwendet man Korrelationsidentifikatoren, um Anfragen und Antworten einander zuzu-

[15] Eine Rückruffunktion in Form einer asynchronen Operation. Sie verhindert, dass der Aufrufer auf die Abarbeitung einer Funktion wartet. Über Rückruffunktionen erreicht man eine lose Kopplung zwischen einzelnen Komponenten. So kann der Aufrufer ohne Unterbrechung seine Arbeit fortsetzen. (Sch08)

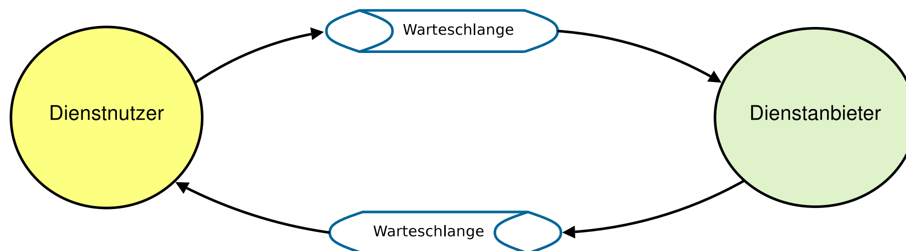


Abbildung 2.11: Asynchrone Kommunikation unter Einsatz von Warteschlangen für die Nachrichten.

ordnen. Ausführungen zu diesen Konzepten erfolgen im Kapitel Message-Exchange Pattern, S. 30ff..

Heterogene Datentypen. Aufgrund der Vielzahl von Diensten in einer SOA existiert auch eine Vielzahl an Datenformaten. Es lässt sich jedoch wegen des verteilten Charakters des Systems und infolge seiner dezentralen Struktur kein einheitliches Datenmodell entwickeln. Das wird auch in absehbarer Zeit nicht gelingen, da sich die Teile des Systems ständig weiterentwickeln und die Ansprüche an dieses gewünschte einheitliche Datenformat fortlaufenden Änderungen unterliegen. Um trotz dieser Widrigkeiten die Zusammenarbeit der unterschiedlichen Services nicht einzuschränken, überträgt man auf der Grundlage von Basistypen (Ganzzahlen, Unicode Zeichenketten etc.) die verschiedenen Strukturen auf die Schnittstellen. So wird ausgeschlossen, dass bei einer Erweiterung der Strukturen die Leistungsfähigkeit des Systems durch umständliche Datentypschachtelungen eingeschränkt wird. So wird es z.B. bei einer kontinuierlich angestrebten Datentypharmonisierung immer einen Zeitpunkt geben, zu welchem die Anforderungen durch Umwege erfüllt werden, obwohl die Datenstrukturen dies nicht ermöglichen. Dabei werden die Umwege oft nur unter dem Gesichtspunkt der Anforderungen, nicht jedoch unter dem Aspekt der Performanz gewählt.

Es ist festzustellen, dass die separaten Anforderungen an die Austauschformate erhalten bleiben, zwischen den Diensten aber eine entsprechende Umsetzung erfolgen muss.

Vermittler. Im Unterschied zu Punkt-zu-Punkt-Verbindungen, bei denen die genaue Adresse bekannt sein muss, übernimmt diese Aufgabe der Vermittler. Dabei sind zwei Formen von Vermittlern zu unterscheiden:

Broker Der Broker hilft dem Nutzer beim Finden der Zieladresse. Er übernimmt die Suche. Mit der gefundenen Adresse wird eine Punkt-zu-Punkt-Verbindung aufgebaut, Nutzer und Anbieter kommunizieren direkt miteinander. Somit erscheint die lose Kopplung nur in der Adressierung, nicht in der Kommunikation. Aufgrund der beiden Aufgaben – Namensauflösung und Übermittlung – entsteht bei der Umsetzung dieses Ansatzes in der Regel mehr Aufwand.
Der Broker findet im Alltag bei der Namensauflösung im Internet als DNS–Server Verwendung. Analog führen wir vor der Kontaktaufnahme zu einer Internetadresse die Ermittlung einer zur DNS-Domäne gehörenden IP durch.

ESB Während der Broker nur kurzzeitig Nutzer und Anbieter entkoppelt, erfolgt die Entkopplung bei der Nutzung eines ESB über den gesamten Zeitraum der Kommunikation. Der Nutzer schickt seine Nachricht an den ESB, dieser ermittelt die Adresse und leitet sie selbstständig weiter. Die Aufgabe des Nutzers besteht lediglich darin, die Adresse des ESB zu kennen und ihm die Nachricht zu übergeben.
Man kann die Arbeitsweise des ESB mit einem Briefkasten vergleichen. Wir werfen die Nachrichten ein – adressiert (und frankiert) – und die Post (in der Rolle des Enterprise Service Bus) übernimmt die Zustellung.

Deployment. Deployment kennzeichnet das „Ausrollen" einer neuen Softwareversion. In diesem Zusammenhang ist auch die Frage zu beantworten, ob alle Bestandteile synchron oder zeitlich entkoppelt ausgetauscht werden können. Um hier eine Antwort zu finden, bedarf es der Betrachtung der Versionierung.

Versionierung. Die Durchführung der Versionierung von Diensten entscheidet auch u.a. darüber, ob eng oder lose gekoppelt wird. Sollte einer der Dienste seine Schnittstellen oder Datenformate ändern, wird danach entschieden, ob andere Dienste durch Abwärtskompatibilität lose oder durch Anpassung ihrer Logik neu übersetzt und damit eng gekoppelt werden.

Ein Beispiel dafür sind Anwendungsschnittstellen - kurz APIs. Die Initialisierung der Kommunikation wird sehr oft damit eingeleitet, dass beide Teilnehmer ihre Versionsnummer mitteilen. Erst danach wird entschieden, wie man weiter miteinander kommuniziert. Sollte die entsprechende Anwendung mehrere Versionen unterstützen, kann man von loser Kopplung sprechen (siehe Abb. 2.12).

```
↼ Protocol 999
⇀ Protocol 8
↼ OPEN IM echo123 this is a prefilled chatmessage
⇀ CHAT #anappo/$echo123;ebe5311cdd203657 NAME #anappo/$echo123;ebe5311cdd203657
⇀ MESSAGE 1259 STATUS SENDING
⇀ CHAT #anappo/$echo123;ebe5311cdd203657 TYPE DIALOG
⇀ CHATMEMBER 1257 ROLE USER
⇀ ...
```

Abbildung 2.12: Austausch von Versionsnummern bei Beginn einer Kommunikation über die Skype-API.

Ein Nachteil bei der Unterstützung von Abwärtskompatibilität besteht darin, dass die Verarbeitungslogik älterer Versionen berücksichtigt werden muss .

2.5 Enterprise Service Bus

Die Basis der Kommunikation in einer SOA bildet in dieser Arbeit der *Enterprise Service Bus* – kurz als ESB bezeichnet. Er stellt das Rückgrat der SOA dar und dient als Integrationsplattform für verschiedene Dienste in einem verteilten System. Als Teil der Infrastruktur übernimmt er den Transport der Nachrichten in Form einer Middleware[16].

Die Vergangenheit hat gezeigt, dass die Bemühungen um eine Integration verschiedener Systeme nicht neu sind (vgl. (Dem08)). Ziel der Integrationsbestand immer darin, alle zu integrierenden Anwendungen lose zu koppeln. Daher gibt es Formen der Integration, die schon aus der

[16] „Middleware (auf Deutsch etwa Zwischenanwendung) bezeichnet in der Informatik anwendungsneutrale Programme, die zwischen Anwendungen vermitteln, so dass die Komplexität dieser Applikationen und ihrer Infrastruktur verborgen wird" (oce)

Interprozesskommunikation bekannt sind. Einige sollen hier kurz erwähnt und später (siehe 'Integrationsstile' auf Seite 46ff.) ausführlicher besprochen werden. Die einfachste Form ist die Nutzung *gemeinsamer Dateien*. Mehrere Anwendungen tauschen Informationen über festgelegte Dateien aus. Etwas allgemeiner ist der Ansatz einer gemeinsamen Datenbank, in welche Anwendungen geschrieben und aus der diese gelesen werden. Es ist auch möglich, die Anwendungen direkt über den Aufruf entfernter Funktionen (RPC) bzw. Methoden (RMI) zu adressieren. Bei dieser Art des Informationsaustausches muss jedoch geklärt werden, wie sich die Anwendungen finden oder wo die Adressen nachzufragen sind. Deshalb ist man im Laufe der Zeit dazu übergegangen, im Zuge der *Enterprise Application Integration* auf eine nachrichtenorientierte Middleware zu setzen.

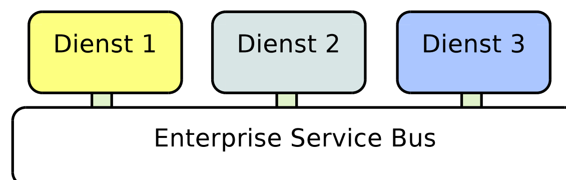


Abbildung 2.13: Der *Enterprise Service Bus* als Rückgrat der Kommunikation.

Der Enterprise Service Bus baut auf einer nachrichtenorientierten Middleware [17] auf und verpackt diese in einer zusätzlichen Abstraktionshülle. Die MOM stellt den Bus innerhalb eines ESB dar.

Dabei wird nach solchen Konzepten, wie synchrones/asynchrones Messaging, Request/Reply, Publish/Subscribe, Store-and-Forward oder die Garantierte Zustellung von Nachrichten (siehe Abschnitt 3.3.5, S. 63f.) gearbeitet. Im Zusammenhang mit dem Messaging wird oft auch der Begriff „Broker“ (siehe S. 25) genannt. Er nimmt Nachrichten an und leitet diese an die ausgewiesenen Empfänger weiter.

Im Gegensatz zur MOM enthält der Enterprise Service Bus jedoch zahlreiche Erweiterungen, wie z.B. Konnektoren zu verschiedenen Protokollen,

[17] Auch als MOM abgekürzt, engl.: message-oriented middleware.

Transformatoren für Protokolle und Daten und eine dynamische Konfiguration. Außerdem bietet er zusätzliche Möglichkeiten zur Steuerung des Nachrichtenflusses (siehe Tabl. 2.3). Der Enterprise Service Bus unterstützt damit die Erreichung des Ziels, nämlich die Interoperabilität von SOA.

	MOM	ESB
Transport	JMS	Anbieter für JMS, SOAP, File, Stream, usw.
Vermittler	Routing und Datentransformation	Protokolltransformation Routing Datentransformation
Konfiguration	Quellcode	dynamisch

Tabelle 2.3: Gegenüberstellung und Abgrenzung von MOM und ESB (Dem08).

Eine der wichtigsten Aufgaben des Enterprise Service Bus ist die Interoperabilität. Diese befasst sich mit den Problemen der Konnektivität von Diensten untereinander, der Datentransformation zwischen den Diensten und mit dem Routing der Nachrichten zwischen dem Nutzer und dem Anbieter eines Dienstes.

In einem System können durchaus mehrere hundert Dienste existieren. Infolgedessen kann der ESB sehr heterogen sein und aus vielen – technologisch verschiedenen – miteinander verbundenen Bussystemen bestehen. Dieser Fall tritt ein, wenn Dienste von Fremdanbietern eingebunden werden und es mehrere Eigentümer gibt, die auf unterschiedliche Produkte setzen und zu deren Bussystemen eine Verbindung hergestellt wird. Trotz dieser Teilung handelt es sich logischerweise um *einen* ESB (siehe Abb. 2.14, S. 29).

Der Austausch von Informationen zwischen Diensten kann auf einfache Weise als Punkt-zu-Punkt-Verbindung erfolgen. In diesem Fall müsste jeder Nutzer die physikalische Adresse vom Endpunkt des jeweiligen Anbieters kennen. Sollte der Anbieter ausfallen, laufen die Anfragen ins Leere und schlagen fehl.

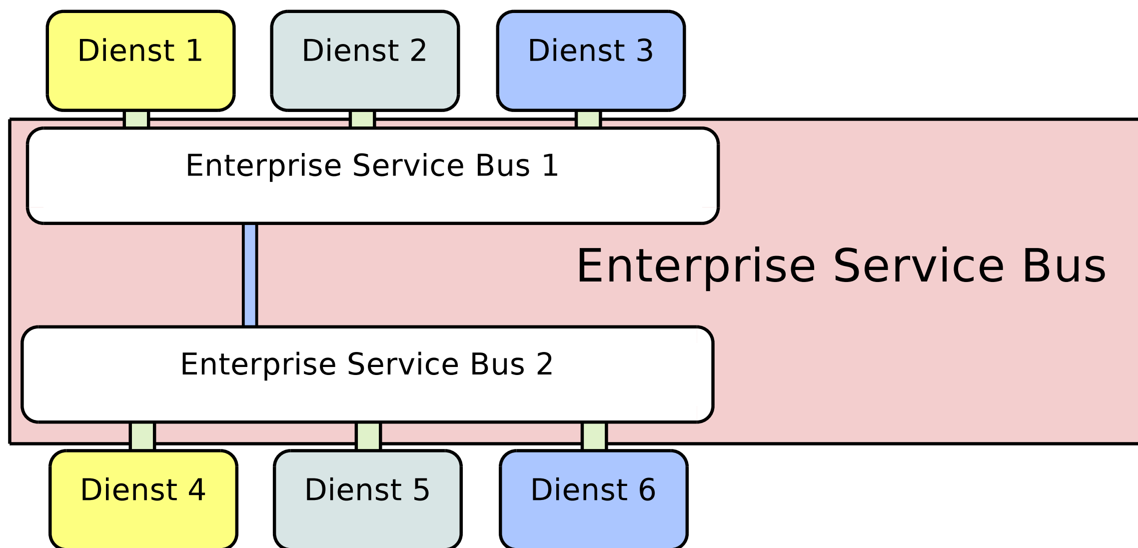
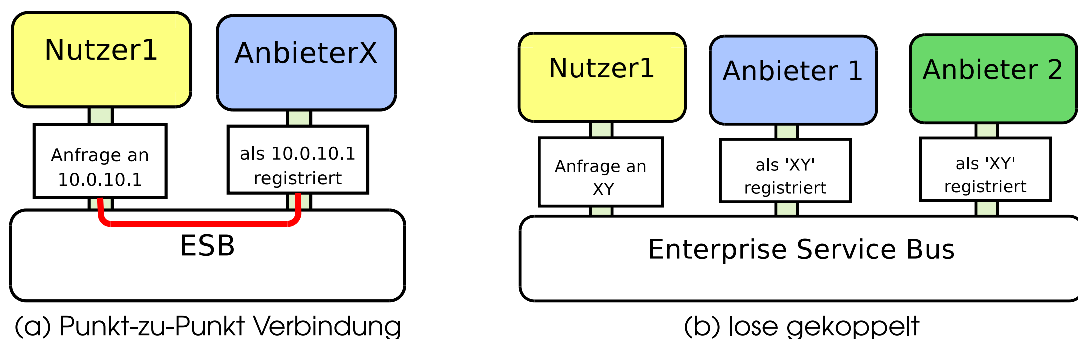


Abbildung 2.14: Architektur mit verteilten Diensten und mehreren Enterprise Service Bus Instanzen.

Mit dem Einsatz des ESB können symbolische Namen für die Endpunkte vergeben und damit eine losere Kopplung erzielt werden. Dieser indirekte Ansatz befähigt dazu, flexibler zu reagieren. So kann u.a. in Problemsituationen auf Ausweichanbieter transparent ausgewichen und bei Bedarf eine Lastverteilung vorgenommen werden (Abb. 2.15). Im Abschnitt 'Lose Kopplung' auf Seite 18ff. wurde auf diese Problematik näher eingegangen.



(a) Punkt-zu-Punkt Verbindung (b) lose gekoppelt

Abbildung 2.15: Die direkte und indirekte Kommunikation.

Transformation

Aufgrund der Tatsache, dass die SOA kein technisches Konzept, sondern ein Architekturstil ist, können unterschiedliche Systeme und Programmiersprachen bei der Implementierung der Dienste zur Anwendung kommen. Diese transparent miteinander zu verbinden, übernimmt der ESB. Dabei besteht die Möglichkeit, sich auf ein gemeinsames Protokoll zu einigen bzw. eine gemeinsame API zu wählen, um die korrekte Transformation der Daten zwischen den Diensten zu gewährleisten.

Dabei kann es sich um die Transformation von Protokollen oder um die Konvertierung von Geschäftsdaten in andere Formate handeln. Hierbei sei auf die Enterprise Integration Pattern (siehe Kapitel 3, S. 45ff.) verwiesen, die für die Datentransformation spezielle Muster bereitstellen.

Routing

Für die Abbildung der Geschäftsprozesse reicht es nicht aus, mithilfe technischer Parameter den Nachrichtenfluss zu steuern. Dafür wird *intelligentes Routing* zur Berücksichtigung der fachlichen Aspekte von Nachrichten (Tageszeiten, Prioritäten, Inhalten usw.) benötigt. Die dazu notwendigen Enterprise Integration Pattern werden noch ausführlich behandelt (Kapitel 3, S. 45ff.).

2.6 Message-Exchange Pattern

Bisher haben wir gesehen, welche Komponenten an der Kommunikation beteiligt sind. Im Weiteren soll untersucht werden, nach welchen Mustern der Nachrichtenaustausch abläuft. Diese sind vom World Wide Web Consortium (CHL+06) für *Web Services* beschrieben und dargestellt. Sie veranschaulichen Folgen von Nachrichten in einem Dienstaufruf oder einer Dienstoperation, wo die Richtung, die Anzahl und die Reihenfolge festgelegt werden. Im Folgenden wird auf die verschiedenen Muster eingegangen, welche anhand der SOA-spezifischen Rollen *Dienstnutzer* (Nutzer) und *Dienstanbieter* (Service) beschrieben werden.

Im Weiteren werden acht Muster genannt, die jedoch in zwei Typklassen (siehe Tabl. 2.4) unterteilt werden können:

Unidirektional. Dieser Typ wird auch als „Einweg“- Kommunikation bezeichnet. Es werden nur in eine Richtung Nachrichten verschickt, man erwartet jedoch keine Antworten.

Bidirektional. Dieser Typ wird auch als „Frage–Antwort“-Kommunikation bezeichnet. Nach jeder gesendeten Nachricht wird eine Antwort erwartet.

Typ	Muster
unidirektional	in-only
	out-only
	robust in-only
	robust out-only
bidirektional	in-out
	out-int
	in-optional-out
	out-optional-in

Tabelle 2.4: Die Klassifikation der Muster.

In-Only

Das Muster *in-only* beschreibt das einseitige Versenden einer Nachricht ohne Rückkopplung. Der Sender wird nicht darüber informiert, ob seine Nachricht angekommen ist oder nicht. Übertragungsfehler bleiben hier unberücksichtigt.

Abbildung 2.16: Die Nachricht wird versendet - *„fire and forget"*.

Robust In-Only

Das Muster kommt ähnlich wie In-Only (s.o.) zur Anwendung, jedoch mit dem Unterschied, dass jede versendete Nachricht eine Fehlernachricht auslösen kann. Übertragungsfehler werden hier festgestellt.

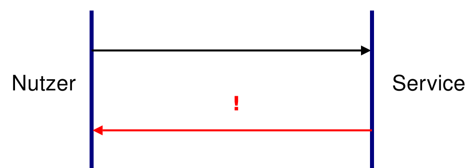


Abbildung 2.17: Nur im Fehlerfall erfolgt eine Rückmeldung.

In-Out

Jeder Nachricht folgt eine Antwort, wobei die Antwort auch eine Fehlermeldung sein kann.

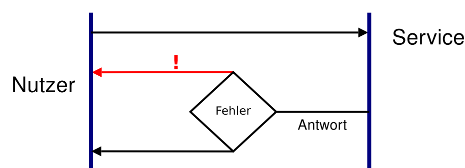


Abbildung 2.18: Jeder Anfrage folgt eine Rückmeldung – entweder eine Antwort oder ein Fehler.

In-Optional-Out

Bei diesem Muster folgt einer Anfrage nicht unbedingt eine Antwort, zusätzlich kann eine Fehlermeldung erhalten werden.

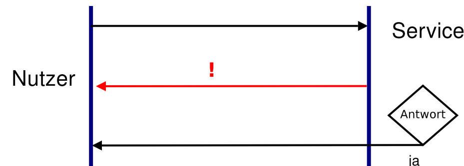


Abbildung 2.19: Die Rückmeldung ist optional, jedoch kann zusätzlich ein Fehler gesendet werden.

In diesem Fall verändern sich die bisher genannten Muster dahingehend, dass nämlich der Anbieter selbst aktiv wird. Das kann bei einer ereignisorientierten Kommunikation dann der Fall sein, wenn der Anbieter auf Vorkommnisse mit Nachrichten an den Nutzer reagiert.

Out-Only

Analog zum Muster *In-Only* (siehe S. 31) bleiben Zustellungsfehler unbemerkt.

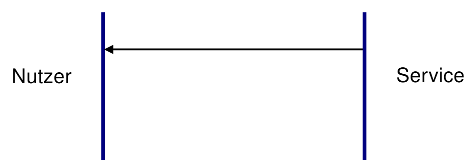


Abbildung 2.20: Die Nachricht wird versendet - „*fire and forget*“.

Robust Out-Only

Auf *Out-Only* (s.o.) aufbauend, werden Übertragungsfehler bemerkt und dem Anbieter (Sender) gemeldet.

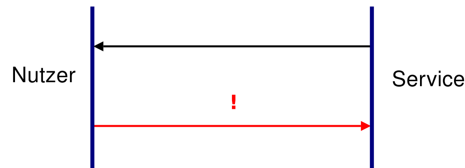


Abbildung 2.21: Nur im Fehlerfall erfolgt eine Rückmeldung.

Out-In

Jeder Nachricht vom Anbieter folgt eine Antwort, wobei die Antwort auch eine Fehlermeldung sein kann.

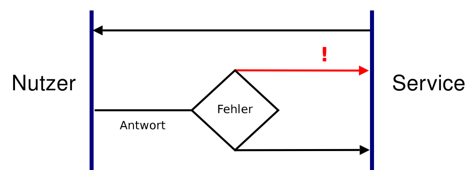


Abbildung 2.22: Jeder Anfrage folgt eine Rückmeldung.

Out-Optional-In

Bei diesem Muster folgt einer Anfrage nicht unbedingt eine Antwort, zusätzlich kann eine Fehlermeldung erhalten werden.

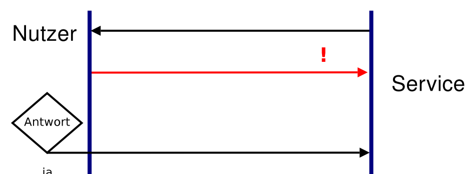


Abbildung 2.23: Die Rückmeldung ist optional, jedoch kann zusätzlich ein Fehler gesendet werden.

Am Ende dieses Abschnitts werden die einzelnen Muster in Form einer Zusammenfassung gegenübergestellt, um die Gemeinsamkeiten, Analogien und Gegensätze zu veranschaulichen (Tabl. 2.5).

Muster	Kommunikation	Fehlerbehandlung	Initiator	Anzahl der Nachrichten
in-only	asynchron	–	Nutzer	1(1)[1]
robust in-only	asynchron	Rückmeldung bei Fehler	Nutzer	1(2)[1]
out-only	asynchron	–	Anbieter	1(1)[1]
robust out-only	asynchron	Rückmeldung bei Fehler	Anbieter	1(2)[1]
in-out	synchron	Fehler ersetzt Nachricht	Nutzer	2(2)[1]
out-in	synchron	Fehler ersetzt Nachricht	Anbieter	2(2)[1]
in-optional-out	asynchron	optional	Nutzer	1-2(2-3)[1]
out-optional-in	asynchron	optional	Anbieter	1-2(2-3)[1]

[1] durch Fehlerbehandlung

Tabelle 2.5: Gegenüberstellung der Muster.

2.7 SOA-Entwurfsmuster

SOA als Architekturstil mit seinen Haupteigenschaften *lose Kopplung*, *Flexibilität*, *Modularität*, *Autonomie* und *Interoperabilität* hat im Laufe der Zeit typische architektonische Strukturen hervorgebracht. Diese werden im Zusammenhang mit der Softwareentwicklung *Entwurfsmuster* genannt. Prinzipiell ist festzustellen, dass allen SOA-Entwurfsmustern die Konzepte der losen Kopplung (Kapitel 2.4, S. 19ff.) und des standardisierten Servicevertrages zu Grunde liegen. Im Weiteren folgen Muster, die sich nahezu in allen Umsetzungen wiederfinden. Deshalb sollen hier einige näher erläutert werden.

2.7.1 Service Messaging

Das **Problem** sehen wir in Folgendem:

Wie kann verhindert werden, dass Dienste zu eng miteinander gekoppelt sind und keine ständige Verbindung haben?

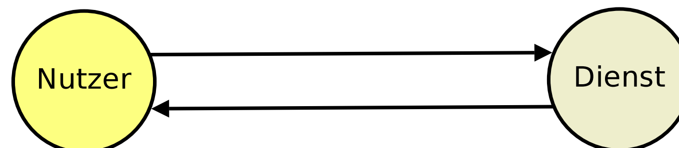


Abbildung 2.24: Die enge Kopplung zweier Dienste.

Die **Lösung** des Problems besteht darin, die entkoppelten Dienste auf eine andere Kommunikationsbasis umzustellen, so dass sie nicht mehr auf ständige Verbindungen angewiesen sind. Deshalb werden die Nachrichten auf einer darunterliegenden Schicht zugestellt.

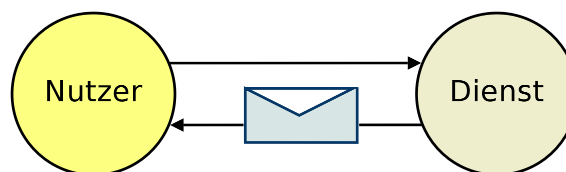


Abbildung 2.25: Kommunikation auf der Grundlage von Nachrichten.

Die Lösung des Problems hat folgende **Auswirkungen**:

Dieses Muster erfordert den Einsatz eines Kommunikationsframework, das grundlegenden Anforderungen genügt, um einen reibungsfreien Nachrichtenaustausch zu gewährleisten. Dazu gehören die Garantierte Zustellung (siehe Abschnitt 3.3.5, S. 63f.) oder zumindest die garantierte Benachrichtigung vom eventuellen Scheitern der Zustellung, die Sicherheit auf Nachrichtenebene (Verschlüsselung der Nutzdaten u.a.), die Übertragung der Nachrichten möglichst in Echtzeit und die Unterstützung für dienstübergreifende Transaktionen (Erl08c). Als eines der fundamentalsten Entwurfsmuster in SOA umfasst das *Service Messaging* den Interaktionsprozess im gesamten System.

2.7.2 Asynchronous Queuing

Das **Problem** sehen wir in Folgendem:

Synchrone Kommunikation erfordert die unmittelbare Antwort auf eine Anfrage und impliziert somit immer eine Zwei-Wege-Kommunikation für jeden Dienstaufruf. Das ist so, weil der synchrone Nachrichtenaustausch zur Folge hat, dass der Aufrufer nach dem Senden der Nachricht so lange wartet, bis er eine Antwort auf die von ihm gestellte Anfrage erhält. Erst danach führt er weitere Operationen durch. Auf der Gegenseite wird der Dienst so lange blockiert, bis er die Bestätigung erhält, dass seine Antwort beim Aufrufenden angekommen ist. Sollte der Dienst nicht erreichbar oder der Aufrufer zur Annahme der Antwort nicht bereit sein, tritt eine Blockierung ein. Diese kann durch die Verzögerung der Abarbeitungszeiten innerhalb des Systems zu einem Leistungseinbruch führen und die Verfügbarkeit des blockierten Dienstes in Frage stellen (vgl. (Erl08a)).

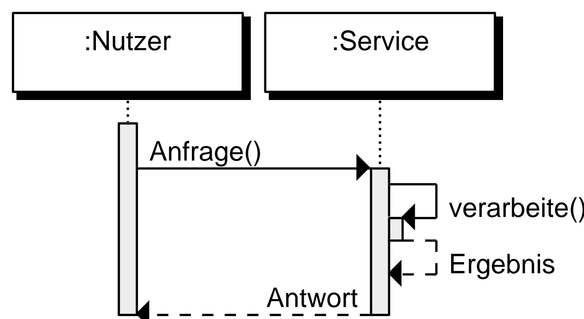


Abbildung 2.26: Für die Dauer des Aufrufs ist der Aufrufende blockiert.

Die **Lösung** des Problems ist durch die Nutzung eines Puffers in Form einer Nachrichtenwarteschlange, in der alle Nachrichten zwischen dem Aufrufer und dem Dienst zeitweise zwischengelagert werden, zu erreichen. Es werden zwei unterschiedliche Warteschlangen verwendet, eine vom Aufrufer zum Anbieter und eine vom Anbieter zum Aufrufer (siehe Abb. 2.11, S. 24). Der Aufrufer setzt seine Nachricht in die Warteschlange und kann sofort andere Tätigkeiten durchführen. Ist der Dienst bereit, holt er sich Nachricht für Nachricht ab, bearbeitet sie und schickt gegebenenfalls seine Antwort für den Anbieter an die Warteschlange zurück. Somit können Aufrufer und Dienst zeitlich *entkoppelt* operieren.

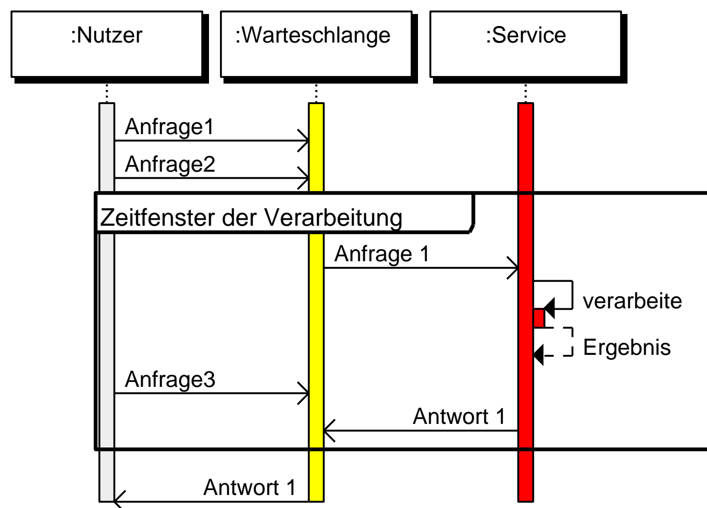


Abbildung 2.27: Der Aufrufer ist nicht blockiert und kann in dieser Zeit weitere Anfragen stellen.

Die Lösung des Problems hat folgende **Auswirkungen**:

Die vermittelnde Warteschlange ermöglicht eine bessere Interaktion mit den Diensten. Das kann durch die zeitliche Entkopplung aber auch dazu führen, dass zusätzliche Logik für die Ausnahmebehandlung notwendig ist. Dadurch würde der gesamte Dienst eine höhere Komplexität erreichen.

Durch den Einsatz einer Warteschlange sind die Kontrolle und Verwaltung des Nachrichtenflusses problematisch. Transaktionen – damit verbunden ist häufig die Abhängigkeit von der Zeit – sind mithilfe eines asynchronen Nachrichtenaustausches nicht möglich. In dem Fall muss auch beachtet werden, dass atomare Transaktionen während der Abarbeitung bis zu einem Commit oder Rollback bestimmte Ressourcen sperren.

Trotz dieser Einschränkungen gibt es jedoch den Vorteil, dass alle Kommunikationspartner sicher sein können, dass der Nachrichtenaustausch erfolgreich abgewickelt wurde. Mit der Quittierung der Nachricht von der Warteschlange nach der Datenübergabe erhält der Sender sofort eine Bestätigung dafür, dass seine Nachricht zugestellt und verarbeitet worden ist (Abb. 2.11, S. 24).

Es muss jedoch beachtet werden, dass diesem Entwurfsmuster zusätzlich das Konzept der *Zustandslosigkeit eines Dienstes*[18] zu Grunde liegt.

2.7.3 Event-Driven Messaging

Das **Problem** sehen wir in Folgendem:

Dienste warten auf Ereignisse und sollen automatisch darauf reagieren. Das allgemein bekannte *Polling*[19] soll dabei nicht verwendet werden, da es eine zusätzliche Belastung für Dienst und Infrastruktur darstellt. Es wird eine andere Lösung erwartet.

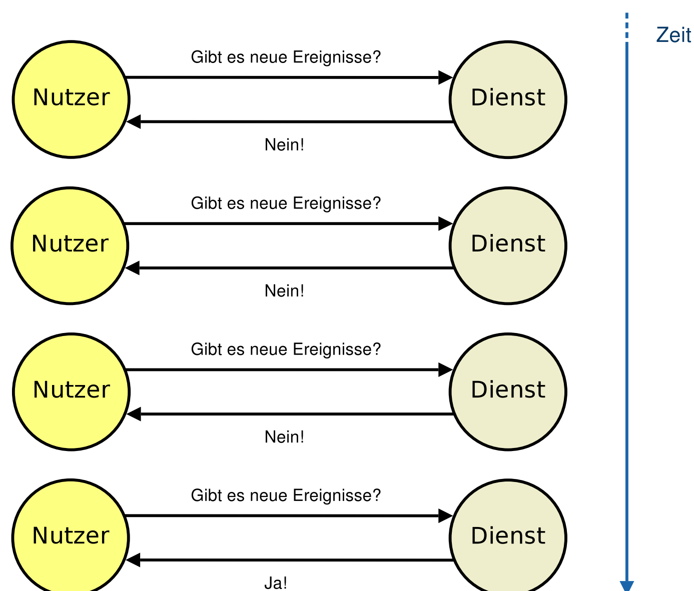


Abbildung 2.28: Kontinuierliches Abfragen eines Dienstes.

Die **Lösung** des Problems besteht darin, dass eine unnötige zusätzliche Belastung des Dienstes und der Infrastruktur in Bezug auf Rechenzeit und Verkehrsaufkommen vermieden werden soll. Normalerweise würde nämlich der Dienst den ereigniserzeugenden Service kontinuierlich abfragen und bei einer positiven Antwort die Aktion ausführen (Polling).

[18] „Services minimize resource consumption by deferring the management of state information when necessary." (Erl07, S. 333) (Dienste minimieren ihren Ressourcenverbrauch durch das Aufschieben der Verwaltung der Zustandsinformation bis zu dem Zeitpunkt, wo es notwendig wird. Übersetzung aus dem Englischen, d.V.).

[19] Zyklisches Abfragen einer Ressource.

Gelöst werden kann dieses Problem durch die Anfertigung einer Liste, in die sich alle interessierten Dienste einschreiben, mit dem Hinweis darauf, an welchen Ereignissen sie interessiert sind. Bei dem dann folgenden Ereignis werden alle auf der Liste eingetragenen Dienste die gewünschte Nachricht erhalten. Man hat damit das Verkehrsaufkommen in der Infrastruktur und die Rechenleistung für die Negativbeantwortung der Anfragen eingespart.

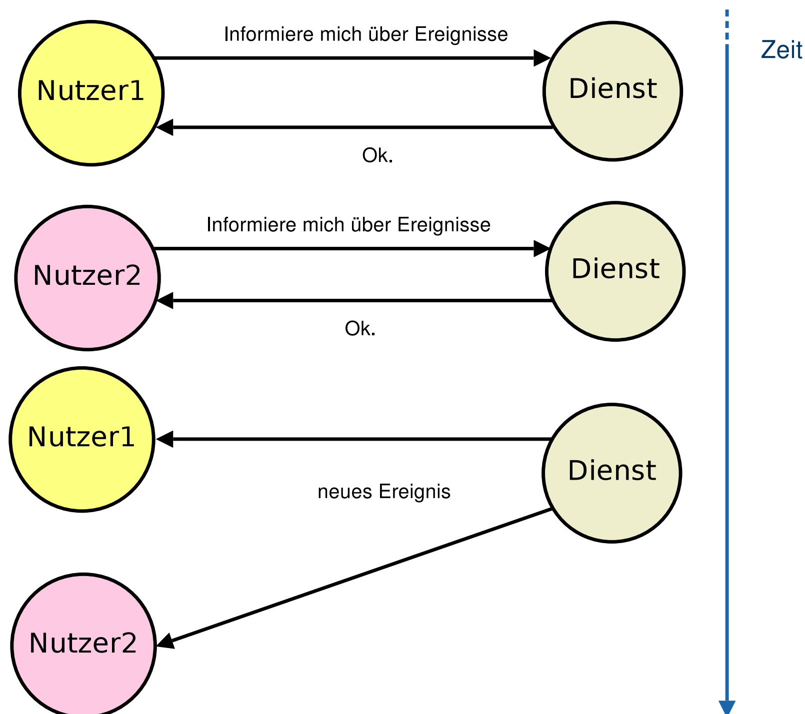


Abbildung 2.29: Alle Nutzer eines Dienstes werden bei einem Ereignis selbstständig informiert.

Dies kann an einem Beispiel demonstriert werden. Dabei wird davon ausgegangen, dass jede Anfrage 50ms dauert, 200 Dienste pro Sekunde nachfragen und alle 10s ein Ereignis stattfindet. Der Dienst würde in diesem Fall die Zeit zwischen den Ereignissen mit dem Beantworten der Nachfragen ausfüllen und eine konstante Vollauslastung aufweisen. Es ist gut vorstellbar, dass bei einem Anstieg der Anzahl der Interessenten sehr schnell eine Überlastung eintritt und der Dienst unerreichbar wird.

Würde nun aber das Verschicken einer Nachricht zeitlich nicht ins Gewicht fallen, können mit diesem Muster problemlos auch einige tausend Interessenten den Dienst nutzen.

Die Lösung des Problems hat folgende **Auswirkungen**:

Event-driven Messaging baut auf asynchroner Kommunikation auf. Dadurch gibt es keine Möglichkeiten zur Abwicklung von Transaktionen.

Weil Benachrichtigungen nicht vorhersagbar sind, müssen die Interessenten immer erreichbar sein. Hier empfiehlt es sich, zusätzlich das Muster *Asynchronous Queuing* einzusetzen (siehe Kapitel 2.7.2, S. 37ff.). In dem Fall muss jedoch beachtet werden, dass diesem Entwurfsmuster zusätzlich die Serviceautonomie zu Grunde liegt.

2.7.4 Legacy Wrapper

Das **Problem** sehen wir in Folgendem:

Beim Aufbau einer SOA müssen sehr oft Altanwendungen integriert werden. Normalerweise (Koc07) wird die Kapselung der API–Zugriffe als Service direkt an den Enterprise Service Bus angeschlossen. Wie kann jedoch verhindert werden, dass die API–Zugriffslogik zu eng mit der Logik zur Kommunikation über den ESB verknüpft wird? Wie können später die Schnittstellen zum ESB ausgetauscht werden, ohne dabei den Adapter[20] zu verändern?

Die **Lösung** des Problems besteht darin, durch eine Dekomposition des Adapterservice die Zuständigkeiten in zwei Services zu trennen – auf der einen Seite in die des Adapters zur API und auf der anderen Seite in die des Adapters zum Enterprise Service Bus.

Der Service Bus Adapter hat die Aufgabe, die Kommunikation zum ESB umzusetzen. Dazu gehören Auf- und Abbau der Verbindung und die Assemblierung der benutzen Protokolle mit den Nutzdaten. Die Trennung reduziert den Code des Adapters auf die Logik zur Ansteuerung der API der Altanwendung und das Versenden der Nachrichten auf Grundlage der Message-Exchange Pattern (Kapitel 2.6, S. 30ff.).

[20]Zur Übersetzung zwischen verschiedenen Schnittstellen (oca).

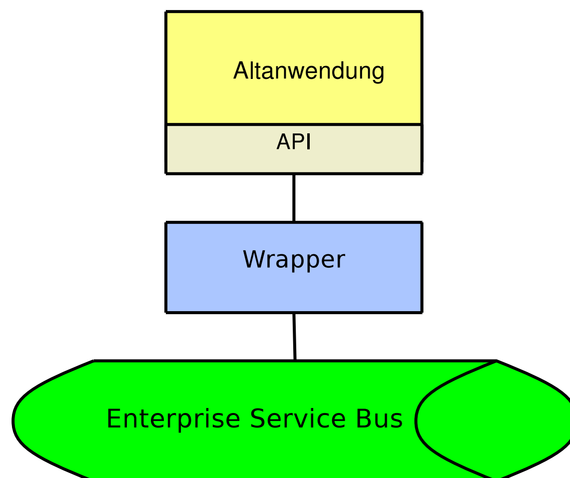


Abbildung 2.30: Variante mit enger Kopplung zum Service Bus Adapter.

Somit kann der Adapter stabil gehalten und der Service Bus Adapter ausgetauscht bzw. angepasst werden. Auf der anderen Seite kann das Legacy-System mit minimalen Auswirkungen auf die Nutzung des Dienstes ausgetauscht werden (Erl08b).

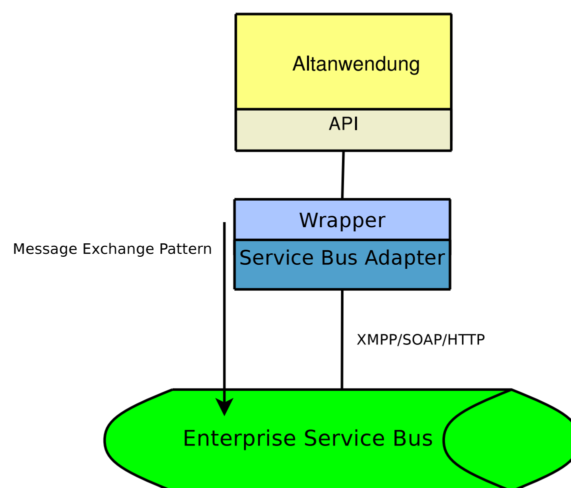


Abbildung 2.31: Dekomposition des Adapters.

Die Lösung des Problems hat folgende **Auswirkungen**:

Der Einsatz dieses Musters bewirkt eine zusätzliche Abstraktion und somit Leistungseinbußen für die Serviceaufrufe und Datentransformationen.

In diesem Fall muss beachtet werden, dass diesem Entwurfsmuster zusätzlich die Serviceabstraktion[21] zu Grunde liegt.

[21] Jeder Dienst sollte eine klare Aufgabe haben und sich auf diese beschränken, um so die Komplexität und Fehleranfälligkeit zu senken.

3 Enterprise Integration Pattern

Im Laufe der Zeit sind die Anwendungslandschaften von Unternehmen unstrukturiert erweitert worden – sie sind in einem kleinen Rahmen geplant, jedoch nicht in den Gesamtzusammenhang eingeordnet worden. Mit jeder neuen Anforderung haben die einzelnen Abteilungen eigene Lösungen mit selbstgewählter Technologie erstellt. Gründe dafür sind meist Zeitdruck und Personalengpässe. Dieses Konglomerat aus vielen verschiedenen Lösungen und historisch gewachsenen Technologien (Betriebssystemen, Programmiersprachen und –modellen etc.) wird in den Fallstudien zur Umstellung auf SOA bei der *Deutschen Post* und *Credit Suisse* als Haupthinderungsgrund für eine fristgerechte Umsetzung neuer Projekte und der damit verbundenen langsameren Reaktion auf Geschäftsanforderungen angesehen. Weiterhin verhindern steigende Anteile der IT-Budgets für Wartung und Instandhaltung die Innovation in den IT-Abteilungen (Heu07, S.64ff.).

Die Deutsche Post z.B. kann nach der Umstellung und Integration 7% des IT-Budgets mehr als bisher für neue Projekte ausgeben. Die Laufzeit der Projekte – einschließlich der Tests – hat sich generell von einigen Monaten auf einige Wochen bis Tage reduziert (Heu07, S.71).

Integration bezeichnet die Zusammenführung verschiedener Anwendungen. Im Gegensatz zur Kopplung ist es das Ziel, die Anzahl der Schnittstellen und die Komplexität zu reduzieren. Darin enthalten sind die Funktions-, Daten- und Geschäftsprozessintegration (wil08a).

Die *Funktionsintegration* beinhaltet die Zusammenfassung von gleichartigen Funktionsmodulen, um die mehrmalige Implementierung in verschiedenen Anwendungsteilen zu vermeiden. Die *Datenintegration* zentralisiert die Daten und löst das Problem der doppelten Datenhaltung und der damit verbundenen Synchronisation. Die *Geschäftsprozessintegration* führt verschiedene Abläufe innerhalb und außerhalb eines Unternehmens zusammen und ermöglicht dadurch die bessere Kommunikation und Verzah-

nung (im Sinne eines reibungsfreieren Ablaufes) der Prozesse untereinander.

Die technische Umsetzung von Integrationen ist als *Enterprise Application Integration* (kurz EAI) sehr gut bekannt. (GH03) beschreibt Herausforderungen, denen man sich aus Gründen der Integration stellen muss.

Die Kommunikationsstruktur im Unternehmen muss sich nach einer erfolgten Integration ändern, da die einzelnen Anwendungen nunmehr als Teil eines Ganzen zu betrachten sind. Bildhaft ausgedrückt, es reicht nicht mehr aus, dass nur die Rechner miteinander kommunizieren, sondern es müssen auch die Abteilungen miteinander reden, um den Fluss der Geschäftsprozesse zu gewährleisten. Jede Abteilung arbeitet mit ihrer bereitgestellten Funktionalität einer anderen zu.

Wichtige Geschäftsabläufe sind von der Integration betroffen und stellen somit das Rückgrat des Unternehmens dar. Ein Ausfall oder eine Fehlfunktion hätten beträchtliche finanzielle Schäden zur Folge.

Eine der großen Herausforderungen besteht darin, die fehlende Kontrolle der Entwickler über zu integrierende Altanwendungen wettzumachen. Es ist dabei gleich, ob sich diese Einschränkungen aus technischen oder politischen Gründen ergeben.

Trotz der hohen Nachfrage an Integrationslösungen gibt es nur wenige technische Standards. Um in Zukunft nicht wiederum einen Mangel an Interoperabilität auch nur im Ansatz entstehen zu lassen, hat sich z.B. die Deutsche Post dafür entschieden, keine proprietären Erweiterungen etablierter Industriestandards zu nutzen (vgl. (Heu07, S.70)).

Die Durchführung und Wartung einer Enterprise Application Integration erfordert spezielle Kenntnisse, wie z.B. die Verteilung und Überwachung der Dienste u.a.. Über dieses Wissen verfügen mehrere Personen, wobei es durchaus sein kann, dass diese nicht im Unternehmen tätig sind.

Integrationsstile

Eine Integration per se schreibt nicht vor, wie integriert wird. Daher sollen einige Integrationsstile betrachtet werden. Mithilfe dieser Stile, aufbauend auf einem älteren Ansatz, hat man immer wieder versucht, die Integration zu verfeinern. Dabei kristallisieren sich einige Kriterien heraus, die laut

(GH03) Beachtung finden sollten und im Folgenden genannt werden:

- Wie eng sollen Anwendungen gekoppelt sein, um der Funktionalität zu genügen, aber dabei auch weiterhin flexibel genug für Veränderungen sein?
- Wie stark passt man die zu integrierende Anwendung an?
- Welche Technologie soll gewählt werden? Dabei sollten der Aufwand für die Einarbeitung der Entwickler, die Kosten bei der Anschaffung der Werkzeuge und das Problem der zu großen Abhängigkeit vom Hersteller[1] beachtet werden.
- Welches Austauschformat benutzen die Anwendungen? Damit verbunden ist auch die Frage nach ihrer Erweiterbarkeit im Laufe der Zeit.
- Soll nur *eine* Daten- oder Funktionsintegration oder sollen sogar *beide* erfolgen?
- Bei der Nutzung nicht lokaler Kommunikation über Fernfunktionsaufrufe sollte berücksichtigt werden, dass synchrone Aufrufe langsamer sind und damit den Aufrufer während der Abarbeitung blockieren können. Eine Lösung können asynchrone Aufrufe sein, die jedoch komplexer in der Entwicklung sind und eine Fehlersuche erschweren.
- Verfügbarkeit spielt bei Fernfunktionsaufrufen eine entscheidende Rolle. Kann die Anwendung mit temporären Ausfällen umgehen?

Gregor Hohpe und Bobby Woolf sind sich darin (GH03) darin einig, dass keiner der folgenden Ansätze alle Kriterien gleichermaßen erfüllen kann.

Der einfachste Ansatz zur Integration von Anwendungen ist der *Dateiaustausch*. Auf diese Weise kann man ohne spezielle Werkzeuge sehr einfach

[1] Tritt auch als Muster *vendor-lockin* auf. Es beschreibt die große Abhängigkeit von einem Hersteller und den damit verbundenen hohen Kosten bei einer Umstellung.

Daten zwischen den Anwendungen austauschen. Dazu sind in der datenabgebenden Anwendung eine Export- und in der annehmenden eine Importfunktion notwendig. Meist einigt man sich auf standardisierte Datenformate, wie z.B. XML. Dafür gibt es eine Reihe von Werkzeugen und Bibliotheken, die den Einsatz erheblich erleichtern.

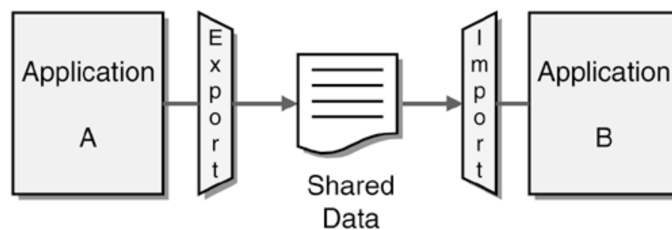


Abbildung 3.1: Der Datenaustausch zwischen Anwendungen mit den zwischengeschalteten Import- und Exportkomponenten (GH03).

Diese Form der Integration über Dateien ist problemlos. Sie lässt sich auf allen gängigen Plattformen und unter allen üblichen Betriebsystemen durchführen. Allerdings können Probleme bei der Synchronisation auftreten, da die Dateien weitaus seltener ausgetauscht werden als beim Einsatz einer gemeinsamen Datenbank.

Die Nutzung einer *gemeinsamen Datenbank* beschleunigt den Datenaustausch und erzeugt zu jeder Zeit einen konsistenten Datenbestand. Synchronisationsprobleme treten nicht auf.

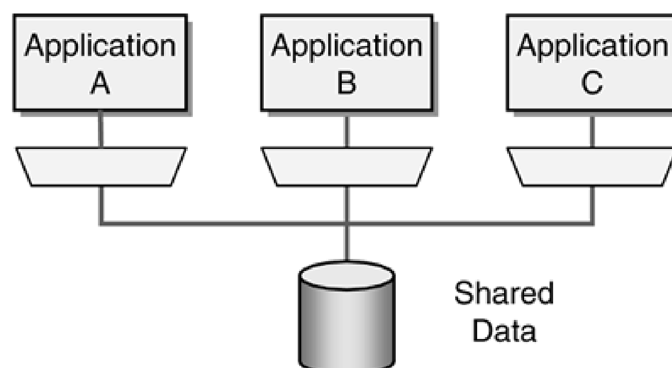


Abbildung 3.2: Keine Konsistenz- und Synchronisationsprobleme beim konkurrierenden Zugriff auf eine gemeinsamen Datenbank (GH03).[2]

Den Vorteilen hinsichtlich der Verwendung einer Datenbank und des Datenaustausches über Dateien stehen einige Nachteile gegenüber. Durch den Zugriff auf Datenbanken müssen die Anwendungen Kenntnis von den Datenbankschemata haben, wobei Probleme auftreten, wenn diese geändert werden (müssen). So müssen infolge einer Integration durch die Verwendung mehrerer verschiedener Anwendungen die Daten teilweise in ein anderes Schema konvertiert werden. Sollten nun viele Anwendungen auf eine Datenbank zugreifen und dabei zahlreiche Lese- und Schreibzugriffe tätigen, können sehr schnell Leistungsengpässe auftreten. Die Lösung der Performanzprobleme durch verteilte Datenbanken kann schnell zu einem Desaster führen, da niemand genau weiß, wo die Daten liegen und Zugriffe auf entfernte Bestände erheblich länger dauern als auf lokale. Für den häufigen Austausch von vielen kleinen Daten – ohne sich dabei auf ein globales Datenmodell zu einigen – eignet sich Messaging besonders gut. Darauf wird der Verfasser später eingehen.

Die Integration mit Hilfe von Dateien und gemeinsamen Datenbanken bezweckt oft nur eine Datenintegration. Wie aber kann die Funktionalität integriert werden? Wie können Anwendungen die Fähigkeiten anderer Applikationen nutzen?

Hierbei hilft der Einsatz von RPC. Anwendungen bieten eine Fähigkeit nach außen über eine Schnittstelle für einen Fernfunktionsaufruf an. Dabei können die Daten über verschiedene Schnittstellen angeboten werden. Es gibt seit vielen Jahren auch mehrere Standards, hierzu zählen u.a. CORBA, COM und Java RMI.

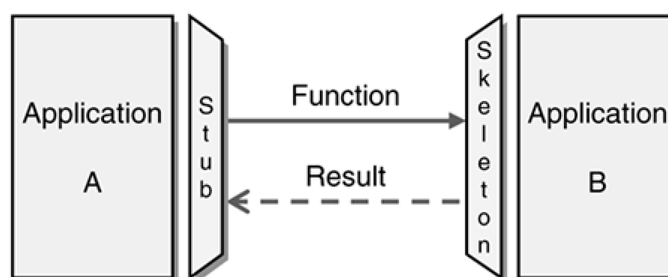


Abbildung 3.4: RPC zur Kommunikation zwischen Anwendungen (GH03).

[2]Vorrausetzung ist, dass die Datenbank ACID (atomarity concurrency isolation durability) unterstützt.

Fernfunktionsaufrufe ermöglichen die transparente Nutzung lokaler als auch entfernter Funktionen. Dass dabei entfernte Aufrufe durch die Kopplung mehrerer Anwendungen zu einem sehr viel langsameren und unzuverlässigeren Gesamtsystem führen können, belegen (WWWK94) und (Fow03) in ihren Schriften. Außerdem führt die enge Kopplung in der Regel zu einem unflexiblen System hinsichtlich künftiger Innovationen und Veränderungen.

Benötigt wird eine Lösung, die flexibler und gleichzeitig genauso leistungsfähig bzw. leistungsfähiger ist als die genannten Ansätze. *Messaging* gilt im Verständnis dieser Arbeit n.u.M. als der beste Kompromiss und ist damit als Lösung zu betrachten. Daraus ergibt sich, ähnlich wie bei einem Dateitransfer, die Möglichkeit, Daten in kleinen Portionen schnell und einfach auszutauschen. Nebenbei wird der Empfänger automatisch über neue Datenpakete informiert, welche er abholen kann. Somit ist kein gemeinsames Datenmodell zwingend erforderlich, so dass sich Änderungen im Unternehmen problemloser umsetzen lassen. Mit dem Senden von Nachrichten lassen sich auch Fähigkeiten im Sinne von Fernfunktionsaufrufen gezielt ansprechen. Beim asynchronen Funktionsaufruf ist der Aufrufer auch nicht direkt vom Scheitern betroffen. Darauf kann mit erneutem Senden oder einer Fehlerverarbeitungslogik reagiert werden.

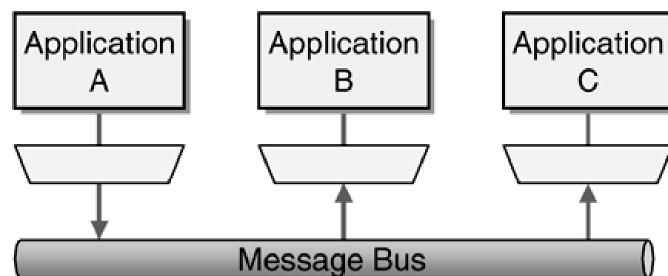


Abbildung 3.5: Integration von Anwendungen durch Messaging (GH03).

Messaging erfüllt unsere Anforderungen, erhöht jedoch in einem geringen Maße die Komplexität durch Abstraktion und zwar meist[3] durch die Nutzung von Asynchronität.

Asynchrone Kommunikation ist eine grundlegende Herangehensweise zur Lösung der Probleme in verteilten Systemen. Um das jedoch zu vermei-

[3]Man kann auch synchron Nachrichten verschicken.

den (vgl. Kapitel 3, S. 46ff.), muss auf eine hohe Kohäsion (viel lokale Arbeit ist notwendig) und eine geringe Adhäsion (wenig entfernte Aufrufe sind nötig) bei der Entwicklung geachtet werden (GH03, S.53ff.).

Im Gegensatz zur Datenbank ist in den Anwendungen keine Datenharmonisierung erforderlich. Die Transformatoren zwischen den Anwendungen lassen sich durch die lose gekoppelte Nachrichtenkommunikation sehr leicht einbinden und anpassen.

Im Zuge einer nachrichtenorientierten Integration treten u.a. Probleme, wie z.B. *langsame Konsumenten*[4] oder *Garantierte Zustellung* (siehe Abschnitt 3.3.5, S. 63f.), auf. Es haben sich Muster herausgebildet, die zur Lösung dieser wiederholt auftretenden Probleme beitragen.

Wie können Integrationsmuster diese Herausforderungen bewältigen helfen?

Muster werden als „im Allgemeinen die Gesamtheit nicht signifikanter, besonderer Unterschiede" definiert(wil08b). So lassen sich auch Muster für Architekturen finden. Die Architekturmuster, zu denen auch die *Enterprise Integration Pattern* (abgekürzt EIP, auf deutsch: Unternehmensintegrationsmuster) gehören, sind im Ergebnis von Erfahrungen der Leute entstanden, die mehrfach eine solche Integration begleitet und Gemeinsamkeiten erkannt haben. Prinzipiell verstehen sich Muster im Allgemeinen und EIP im Speziellen als *Erfahrungen und bewährte Lösungen immer wiederkehrender Problematiken*. Es wird dabei betont, dass Integrationsmuster keine copy-paste Codeschnipsel oder gekapselte Komponenten sind, die es gilt einzubauen (GH03, S. 4). Vielmehr handelt es sich um eine Brücke zwischen Visionen der Softwarearchitekten und den Problemen der Softwareentwickler.

3.1 Grundlegende Konzepte

Im Umfeld von Messaging-Systemen haben sich mehrere Konzepte herausgebildet. Auf einige, wie z.B. *Nachricht*, *Endpunkt* und *Nachrichtenkanal*,

[4] Dazu gibt es ein SOA-Muster mit gleichem Namen.

wird im Weiteren näher eingegangen. Andere, wie z.B. *Pipes and Filter*, *Message Router* und *Message Translator*, werden, weil es einfach zuviele sind, im Rahmen dieser Arbeit nicht berücksichtigt. Sie werden nur der Vollständigkeit halber erwähnt.

In dem weiteren Text wird des öfteren von *Messaging-System*[5] (auch als Middleware bezeichnet) die Rede sein. Messaging-System meint das vermittelnde Element als Kern einer nachrichtenorientierten Infrastruktur. Diesem fallen Aufgaben zu, die das Annehmen, das Vermitteln und das Zustellen von Nachrichten, den Auf- und Abbau und die Organisation von Verbindungen zwischen den teilnehmenden Anwendungen umfassen.

Die teilnehmenden Anwendungen haben im Kontext einer nachrichtenorientierten Kommunikation gebräuchliche Bezeichnungen, wie Sender/-Empfänger oder Produzent/Konsument. Analog dazu spricht man in einer serviceorientierten Umgebung von Anbieter und Nachfrager. Im Sonderfall einer Kommunikation von einem Sender zu mehreren Empfängern über einen *Publish-Subscribe*-Kanal verwendet man die Begriffe *Publisher* und *Subscriber*.

Ein *Nachrichtenkanal* bezeichnet einen unidirektionalen Verbindungsweg für die Daten zwischen Anwendungen (siehe Abb. 3.6b, S. 53). Kanäle senden deshalb nur in eine Richtung, weil sonst ein Produzent seine eigenen Nachrichten konsumieren würde. Sollen Daten bidirektional ausgetauscht werden, kommen mehrere Kanäle zum Einsatz. Diese werden über logische Adressen angesprochen. Es ist durchaus üblich, dass Anwendungen mehrere Kanäle benutzen.

Man unterscheidet zwei Typen von Nachrichtenkanälen, nämlich *Punkt--zu-Punkt* und *Publish-Subscribe*. Der *Punkt-zu-Punkt*-Kanal verschickt die Nachricht an nur einen Empfänger,vom *Publish-Subscribe*-Kanal dagegen werden alle eingeschriebenen[6] Empfänger mit der Nachricht beliefert.

Eine *Nachricht* stellt ein atomares Paket von Daten dar. Ist ein Datenpaket zu groß, kann es in mehrere Nachrichten aufgeteilt werden (siehe

[5]Da eine deutsche Übersetzung – wie z.B. ein Nachrichtensystem – nicht vollständig die Bedeutung wiedergibt, wird im Folgenden der englische Ausdruck verwendet.

[6]Im Zusammenhang mit einem *Publish-Subscribe*-Kanal wird der logische Endpunkt als *topic* (engl.: Thema) bezeichnet. Dies geschieht in Anlehnung an die Metapher einer Diskussionsrunde, bei der alle Teilnehmer die Beiträge aufnehmen und dann ihrerseits einen Beitrag leisten).

(a) eine Nachricht (b) ein Nachrichtenkanal (c) ein Endpunkt

Abbildung 3.6: Piktogramme: Nachricht, Endpunkt und Nachrichtenkanal (GH03).

Abb. 3.6a). In der Nachricht werden die Daten als Bitdatenstrom transportiert. Das Ein- und Auspacken der Daten in eine bzw. aus einer Nachricht wird dabei analog zu RPC als Marshalling und Unmarshalling bezeichnet.

Nachrichten bestehen aus einem Kopfdaten- und Hauptdatenteil – dem Header und dem Body. Der Kopf einer Nachricht enthält die logische Zieladresse, die logische Absenderadresse und eventuell Beschreibungen des Inhalts. Neben der Aufteilung nach strukturellen Gesichtspunkten werden Nachrichten auch nach Aufgabentypen unterschieden. Im Weiteren wird auf folgende Nachrichtenmuster eingegangen: *Befehlsnachricht*, *Dokumentennachricht*, *Anfrage-Antwort-Nachricht* und *Nachrichtenverfall* (Kapitel 3.4, S. 64).

SOA verdankt seine lose Kopplung der reinen nachrichtenorientierten Ausrichtung der Kommunikation.

Dabei stellt sich jedoch die Frage, ob in jeder Anwendung die Logik zur Interaktion mit dem Messaging-System implementiert werden muss, oder ob es vielleicht eine bessere Lösung gibt. Ein Lösungsmuster ist der *Endpunkt* (siehe Abb. 3.6c). Der Endpunkt stellt die Schnittstelle zum Messaging-System – meist in Form einer Client-API – dar. Diese ist nicht anwendungs-, sondern domänenspezifisch, kann somit wiederverwendet werden und stellt die Anbindung dar. Diese API kann soweit allgemeingültig sein, dass sie Industriestandards, wie JMS z.B., umsetzt und dabei dem Entwickler die Wahl lässt, welches JMS-konforme Produkt und welche Bibliotheken er einbinden möchte. Mit dem Endpunkt wird das Messaging für die Anwendung gekapselt und stellt somit eine spezielle Form des Kanaladapters dar (siehe Abschnitt 3.3.2, S. 59f.).

3.2 Nachrichtenendpunkte

Endpunkte sind die angepassten Schnittstellen an die Messaging-Systeme. Die Schnittstellen sind zur Annahme und Abgabe von Nachrichten bestimmt. Dabei muss man die Frage stellen, auf welche Weise der Konsument an der Schnittstelle vom Vorhandensein einer Nachricht erfährt. Die einfachste Vorgehensweise wäre es, regelmäßig den Datenkanal, an dem man sich registriert hat, zu fragen, ob neue Nachrichten eingetroffen sind. Bei dieser Vorgehensweise – bekannt als *Fragender Konsument* bzw. *Polling Consumer* – würde man die wenigen bzw. seltenen Nachrichten zyklisch abfragen und die Rechenlast konstant (hoch) halten[7].

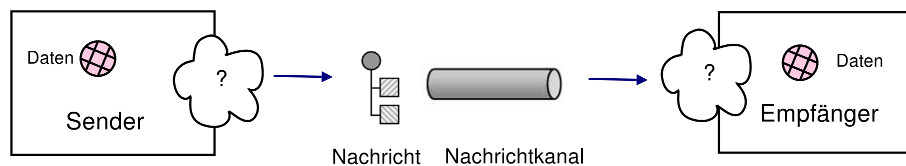


Abbildung 3.7: Aufgabe der Endpunkte im Messaging-System.

3.2.1 Ereignisgetriebener Konsument

Die Systemlast kann auf ein notwendiges Minimum reduziert werden, indem man dem Nachrichtenkanal mitteilt, dass wir bei einem Ereignis informiert werden wollen (engl.: eventdriven). Infolge dieser asynchronen Arbeitsweise kann der untätige Konsument und Empfänger in der freien Zeit schlafen und Ressourcen freigeben (Abschnitt 2.7.3, S. 39).

Der *Ereignisgetriebene Konsument* ist ein Objekt, welches genutzt wird, um den Anwendungen mitzuteilen, dass ihnen Nachrichten zugestellt werden sollen. Der Empfänger der Nachrichten erstellt einen neuen Konsumenten, der sich mit einem bestimmten Nachrichtenkanal verbindet. Je-

[7] Die Rechenlast ist so hoch, dass sich selbst Betriebssystementwickler Gedanken über ihre Minimierung gemacht haben. Selbst auf einer sehr tiefen Ebene, wie z.B. dem Dateisystem, wird versucht, Polling einzuschränken. So ist seit der Version 2.6.13 im Linuxkernel das *inotify* Framework integriert, (Lov05),(cor04), um so Änderungen im Dateisystem – auf eine ereignisgetriebene Art und Weise – melden zu lassen (siehe Abschnitt 2.7.3, S. 39).

des Mal, wenn eine neue Nachricht im Kanal eingeht, wird dieser Konsument über eine api-spezifische Methode informiert und ihm die Nachricht als Parameter übergeben. Dieser ruft seinerseits die anwendungsspezifische Callback-Funktion auf und übergibt dem Empfänger (der eigentlichen Anwendung) die Nachricht.

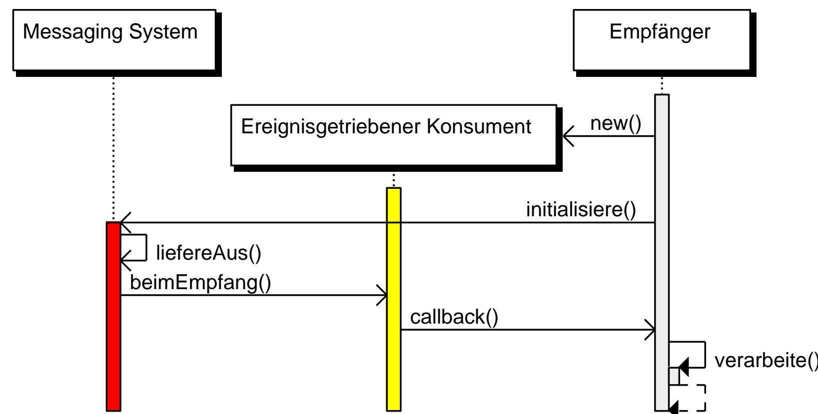


Abbildung 3.8: Zeitliche Abfolge beim Einsatz eines Ereignisgetriebenen Konsumenten.

Ereignisgetriebene Konsumenten nehmen die Nachricht ohne Umschweife sofort an. Bei (GH03, S. 498ff.) wird der Einsatz eines *Polling Consumer* empfohlen, um so die Entnahme besser kontrollieren zu können. Während der Abarbeitung nimmt der Empfänger keine weiteren Nachrichten an, es sei denn, er registriert mehrere Konsumer. Mit einem *Message Dispatcher* wäre es möglich, die entnommenen Nachrichten auf Verarbeitungsmodule zu verteilen. Dadurch könnten mehrere Konsumenten simuliert werden[8]. Die sequenzielle Entnahme neuer Nachrichten kann nicht in allen Situationen die Nachfrage der Umgebung befriedigen, so dass sich die Forderung nach einer schnelleren Verarbeitung neuer Nachrichten ergibt.

[8]Intern wird beim *Konkurrierenden Konsumenten* ein Message Dispatcher eingesetzt. Dadurch werden Synchronisationsprobleme verhindert und die Nachrichten effektiv verteilt.

3.2.2 Konkurrierende Konsumenten [9]

Eine einfache Möglichkeit, diesem Engpass zu begegnen, sind mehrere Konkurrierende Konsumenten. Alle Konsumenten melden sich an dem Kanal an und entnehmen, miteinander in Konkurrenz stehend, Nachrichten.

Alle Konsumenten laufen parallel und können somit auch parallel auf den Kanal zugreifen. Beim Zugriff auf die Nachrichten übernimmt der Nachrichtenkanal die Synchronisation, so dass keine Nachricht doppelt ausgeliefert wird. Für jeden angemeldeten Konsumenten muss das Messaging-System einen Thread zur Übergabe der Nachricht bereitstellen. Freie verfügbare Konsumenten können Nachrichten entnehmen, während andere dies schon getan haben. Wenn man davon ausgeht, dass das Entnehmen der Nachricht nur 10ms beansprucht, ihre Verarbeitung ca. 30s dauert und pro Minute drei neue Nachrichten eintreffen, so würde der Bestand an Nachrichten im Kanal bei nur einem Konsumenten zunehmen. Gäbe es aber drei Konsumenten, bliebe nach der Verarbeitung noch Zeit zum Warten.

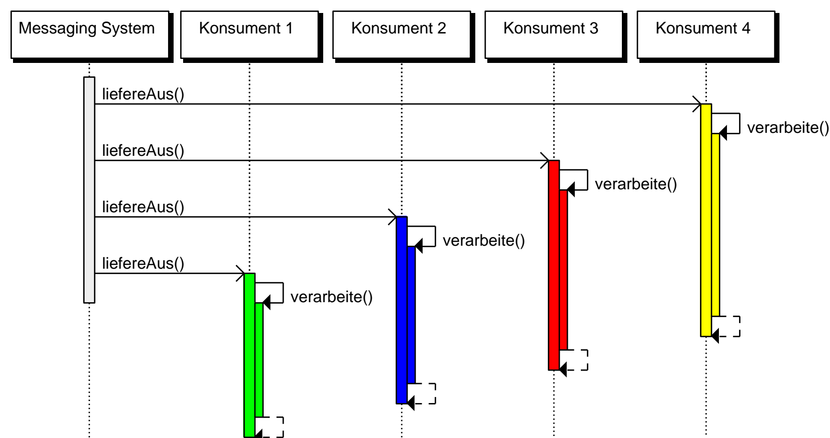


Abbildung 3.9: Hier wird das zeitlich parallele Nachrichtenzustellen beim Einsatz Konkurrierender Konsumenten deutlich.

Konkurrierende Konsumenten sind auf *Punkt-zu-Punkt*-Nachrichtenkanäle beschränkt, weil bei einem *Publish-Subscribe*-Nachrichtenkanal die Nachricht an alle gleichermaßen kopiert und zugestellt wird.

[9] engl.: Competing Consumer

3.3 Nachrichtenkanäle

Der Nachrichtenaustausch erfolgt, wie schon festgestellt, über sogenannte Nachrichtenkanäle. Diese Kanäle können sich jedoch grundlegend unterscheiden. Wenn wir genau einem Endpunkt die Nachricht zukommen lassen wollen, während wir in einem anderen Szenario eine Nachricht an viele Endpunkte zu schicken beabsichtigen, werden wir dazu zwei unterschiedliche Typen nutzen. Im ersten Fall wird ein *Punkt-zu-Punkt*-Kanal, im zweiten ein *Publish-Subscribe*-Kanal eingesetzt.

Im Weiteren müssen in diesem Zusammenhang mehrere Fragen gestellt werden: Wie können wir sichergehen, dass unsere Nachricht zugestellt wurde und nicht auf dem Transport verloren gegangen ist? Welche Sicherungsmaßnahmen helfen uns beim Ausfall des Messaging-Systems? Sind die Nachrichten verloren, wenn sie nicht übermittelt worden sind?

Diese Probleme werden mit dem Muster ‘Garantierte Zustellung’ auf Seite 63f. gelöst.

Wohin führen die Nachrichtenkanäle, wenn sie die Anwendungen nicht direkt miteinander verbinden? Welche Komponente vermittelt zwischen den angeschlossenen Applikationen? Welche Komponente verbindet die unterschiedlichen Systeme und Plattformen, die am Messaging-System angeschlossen sind?

Die Anwort auf diese Fragen finden wir im folgenden Kapitel.

3.3.1 Nachrichtenbus [10]

Der Nachrichtenbus als integraler Bestandteil in einer SOA ermöglicht erst die lose Kopplung. Durch ihn sind die angeschlossenen Anwendungen bzw. Dienste in der Lage, transparent angesprochen zu werden. Er schafft die Bedingungen dafür, dass ohne Einwirkung auf andere Teile des Systems Dienste ausgetauscht werden. Ohne einen Nachrichtenbus müssten alle Anwendungen für eine Kommunikation aneinander angepasst werden. Die dabei auftretenden Probleme bei einer systemübergreifenden Datenmodellharmonisierung und einer einheitlichen, alle Aspekte berücksichtigenden Befehlsstruktur, seien hier nur am Rande erwähnt.

[10] engl.: Message Bus

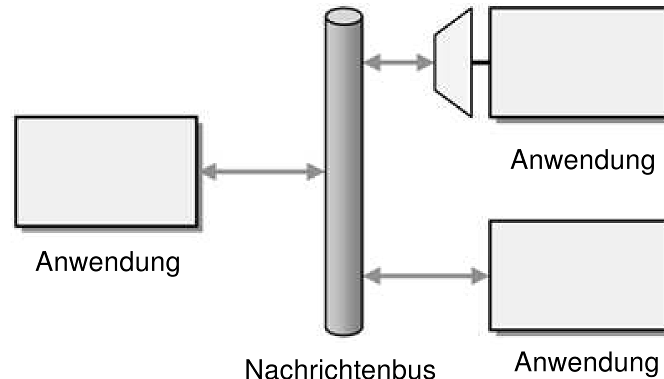


Abbildung 3.10: Die Anwendungen verbinden sich mit Adaptern oder direkt mit dem Nachrichtenbus.

Nach (GH03, S. 137ff.) gehören zum Nachrichtenbus:

- Eine *gemeinsame Kommunikationsinfrastruktur*, die sprach- und plattformübergreifende Schnittstellen zur Anbindung unterschiedlicher Systeme bereitstellt. Die einfachste Möglichkeit besteht darin, einen *Publish-Subscriber-Kanal* einzusetzen und allen Anwendungen alle Nachrichten zuzustellen. Dass dabei Anforderungen wie Routing[11], Load Balancing[12] und Throttling[13] nicht abgedeckt sind, ist offensichtlich.

- *Adapter* zwecks Verbindung zum Nachrichtenbus, die sich auf der Grundlage eines vorher vereinbarten minimalen Datenmodells verbinden, um auf diese Weise eine lose Kopplung zu erreichen (siehe dazu: Kapitel 2.4, S. 24).

- Eine *gemeinsame Befehlsstruktur*. Sie ist – ähnlich wie bei einem Prozessor – erforderlich, um die essentiellen Operationen der Nachrichtenübergabe zu ermöglichen.

[11] Auf der Grundlage verschiedener Kriterien werden Nachrichten unterschiedlichen Empfängern zugestellt.
[12] Lastabhängiges Verteilen der Verarbeitung auf viele gleichartige Dienste.
[13] Abbremsen des Senders, um eine Überlastung des Empfängers zu verhindern.

Der *Nachrichtenbus* stellt die Basis für eine nachrichtenorientierte Middleware dar und baut auf anderen Mustern auf, von denen einige im Folgenden genannt werden sollen.

3.3.2 Kanaladapter

Wir stehen vor dem Problem einer Integration von Anwendungen in eine SOA. Da SOA nachrichtenorientiert ist, müssen die Anwendungen auch über Messaging kommunizieren können. Weil die Zuverlässigkeit eines Messaging- Systems genutzt werden soll, verzichten wir auf HTTP-, TCP- oder RPC-basierte Ansätze. Benötigt wird eine verbindende Komponente, die die Altanwendung Messaging-fähig macht, nämlich ein *Kanaladapter*. Dieser Kanaladapter sendet und empfängt Nachrichten auf der einen Seite und nutzt die Funktionen der zu integrierenden Anwendung auf der anderen Seite (siehe Abschnitt 2.7.4, S. 41ff.). Der Adapter kann auf verschiedenen Ebenen – der Nutzungsoberfläche, der Geschäftslogik mittels einer bereitgestellten API oder der Datenbank – die einander fremden Systeme verbinden. Man bezeichnet die Adapter deshalb als *Oberflächen-*, *Geschäftslogik-* und *Datenbankadapter*. Wir werden uns auf die Anpassung an die Geschäftslogik bzw. den Geschäftslogikadapter konzentrieren, weil in der Referenzanwendung beschrieben in Kapitel 4, S. 67, die anderen beiden Typen keine Rolle spielen.

Die Geschäftslogik wird meist über eigene Schnittstellen in Form einer API für bestimmte Technologien geöffnet. Bei der Verwendung von programmiersprachenorientierten Schnittstellen treten in der Regel die wenigsten Probleme auf, weil weiterhin die gleiche Technologie genutzt wird. Protokollorientierte APIs dagegen erfordern einen zusätzlichen Adapter für das verwendete Protokoll. Im Vergleich zu den anderen beiden Adaptern stellt dieser (der Geschäftslogikadapter) den besten und effizientesten Ansatz dar, um Legacy-Anwendungen einzubinden.

Der Kanaladapter muss nicht unbedingt lokal installiert sein. Somit ist es auch möglich, bestehende Anwendungsserver zu integrieren, ohne diese zu verändern. Der Adapter würde dann z.B. über ODBC oder HTTP angesprochen. Diese Vorgehensweise kann ratsam sein, wenn das System aufgrund mangelnder personeller Möglichkeiten (Entwickler pensioniert, we-

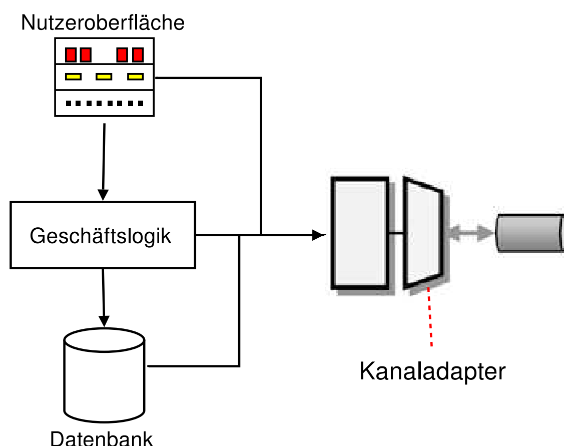


Abbildung 3.11: Der *Kanaladapter* im Zusammenspiel mit den Anwendungsschichten.

nig verbreitete oder alte Programmiersprache etc.) so wenig wie möglich verändert werden soll, gemäß dem Grundsatz „Never touch a running System"[14].

3.3.3 Punkt-zu-Punkt-Kanal

Wie können wir sichergehen, dass eine Nachricht von nur einem Konsumenten abgerufen wird? Wie wird abgesichert, dass keine Nachricht mehrmals an unterschiedliche Empfänger versandt wird?

Das können wir durch den Einsatz eines Punkt-zu-Punkt-Kanals erreichen. Dabei ist nicht auszuschließen, dass sich viele an diesem Kanal registrieren. Jedoch wird, ganz im Sinne der Konkurrierenden Konsumenten, nur jeweils einer die neue Nachricht herausnehmen. Somit erhält nur ein Konsument die neue Nachricht, viele (unterschiedliche) Nachrichten können parallel abgearbeitet werden. Der Wettstreit der Konsumenten um die nächste Nachricht ist auch der Einstiegspunkt, um eine Lastverteilung auf mehrere Konsumenten zu installieren (GH03, S. 103ff.).

[14] Grundsatz in der praktischen Informatik: Ein funktionierendes, allen Anforderungen genügendes System sollte man nicht verändern. Darf nicht so verstanden werden, dass man sich notwendigen Veränderungen gegenüber (Sicherheitsupdates, Systemintegration etc.) verschließt.

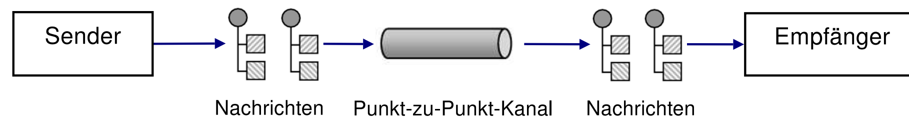


Abbildung 3.12: Die Verwendung eines *Punkt-zu-Punkt*-Kanals zwischen zwei Endpunkten.

Wollen wir nun allerdings viele Konsumenten mit *der gleichen* Nachricht versorgen, wird ein anderes Muster benötigt.

3.3.4 Publish-Subscribe-Kanal

Wie können wir eine Nachricht an viele Empfänger in einem Broadcast[15] versenden? Wie kann verhindert werden, dass irgendein Empfänger eine Nachricht oder eine Benachrichtigung über eine neue Nachricht mehrmals erhält?

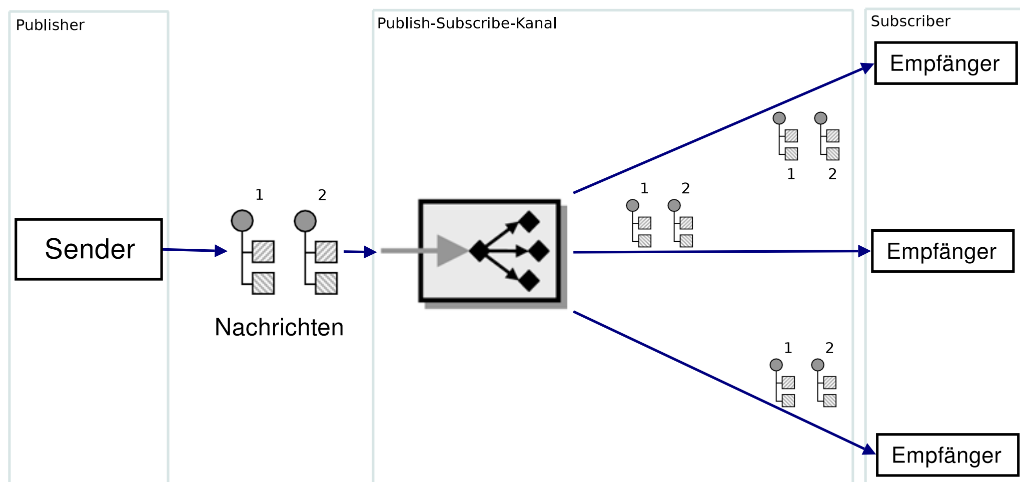


Abbildung 3.13: Prinzip eines *Publish-Subscribe-Kanals*.

Das erreicht man, indem zu jedem Empfänger ein separater Kommunikationskanal aufgebaut wird und die Ereignisse bzw. Nachrichten in diesen hineinkopiert werden. Der Vorgang läuft wie folgt ab: Der Produzent im Kontext eines Publish-Subscribe-Kanals wird *Publisher*, der Konsument

[15] deutsch.: Rundruf, Nachricht an alle Benutzer

Subscriber genannt. Der Publisher erzeugt eine Nachricht und setzt sie in den Kanal. Dieser liefert sie an jeden der Subscriber aus, indem er zu jedem einen separaten Kanal aufbaut, wobei nur ein Konsument erlaubt ist. Dann wird die Nachricht jeweils in die Kanäle kopiert. Dadurch werden Doppelauslieferungen und Konkurrenzverhalten (siehe Abschnitt 3.2.2, S. 56f.) vermieden.

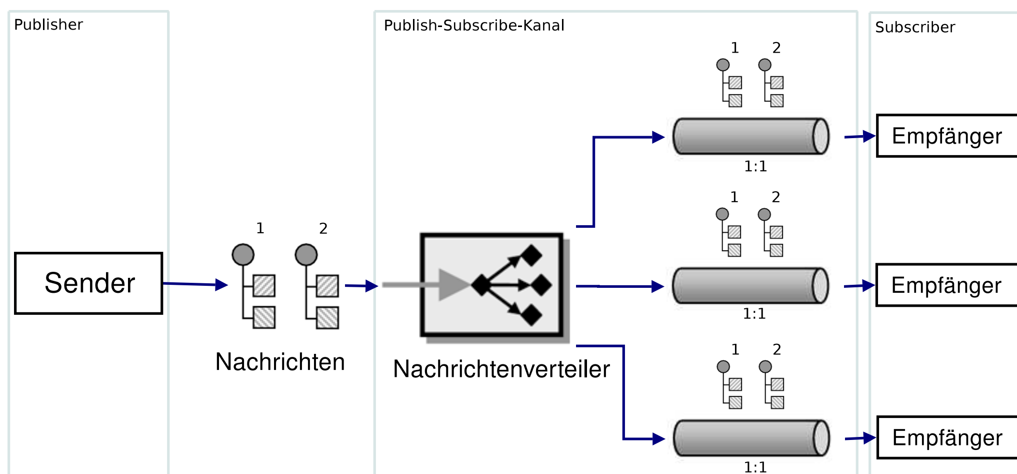


Abbildung 3.14: Arbeitsweise eines *Publish-Subscribe-Kanals*.

Mithilfe der Auslieferung der Nachrichten durch Kopien an die Subscriber kann laut (GH03, S. 106ff.) sehr einfach Fehlersuche betrieben werden. Durch Hinzufügen eines weiteren Subscriber können alle Nachrichten auf dem Kanal mitgelesen werden. Dieser Vorteil ist aber auch ein Sicherheitsrisiko, welches durch Zugangskontrolle zum Kanal behoben werden kann. Auch sollte vermieden werden, Nachrichten – wie z.B. solche zur Rechnungsabwicklung – an mehrere Empfänger zu schicken, da es dann evtl. zu nicht gewollten Seiteneffekten (doppelter Abrechnung) kommen kann. Deshalb sollte ein Punkt-zu-Punkt-Kanal genutzt werden, zu dem viele Messaging-Systeme eine ähnliche Funktion zum Mitschneiden von Nachrichten zur Fehlerverfolgung anbieten, ohne diese jedoch zu konsumieren und damit aus dem Kanal zu entfernen.

3.3.5 Garantierte Zustellung [16]

Eine wesentliche Frage ist, wie der Sender absichern kann, dass auch bei Ausfällen des Messaging-Systems seine Nachrichten ankommen. Durch die Entkopplung der Anwendungen über das Messaging-System allein können wir nicht sichergehen, dass die von uns versendete Nachricht angekommen ist. Ihr Empfang wurde zwar quittiert, es ist jedoch keine direkte Antwort erfolgt. Obwohl die Middleware die Nachrichten nach dem „store-and-forward“ Prinzip[17] sendet, kann bei einem unerwarteten Ausfall die Pufferung im Hauptspeicher nicht ausreichen, weil dieser nicht persistent und somit nur zur Laufzeit abrufbar ist. Was benötigt wird, ist ein nichtflüchtiges Speichermedium, welches Ausfälle überbrücken kann. Dafür eignet sich u.a. eine Datenbank. Dabei wird die Nachricht erst auf dieser zwischengespeichert, dann quittiert und erst danach eine Auslieferung an das Ziel versucht. Mit dieser *Garantierten Zustellung*[18] kann man längere Ausfälle ohne Datenverlust verkraften.

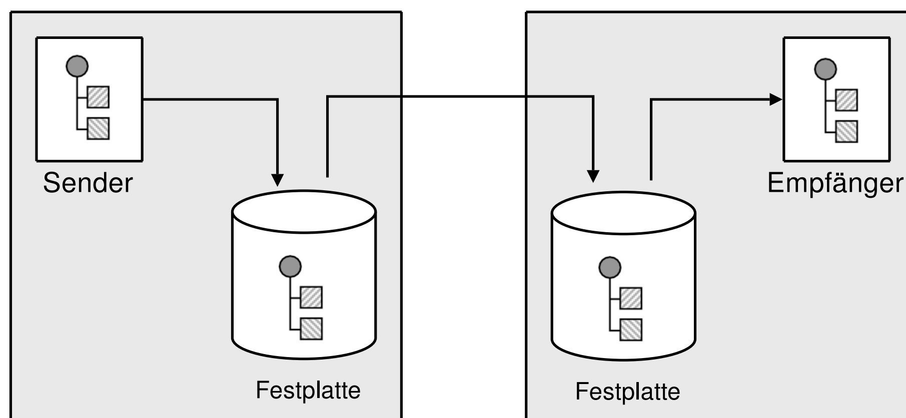


Abbildung 3.15: Konzept der *Garantierten Zustellung*.

Allerdings sollte beachtet werden, dass die Absicherung auf Kosten der Leistung erkauft wird. Da, wie unser Beispiel zeigt, Hauptspeicher naturgemäß wesentlich schneller als Festplatten sind, sollte überlegt werden, ob der Verlust von Nachrichten durch einen Ausfall tolerierbar ist. Durch die

[16] engl.: Guaranteed Delivery

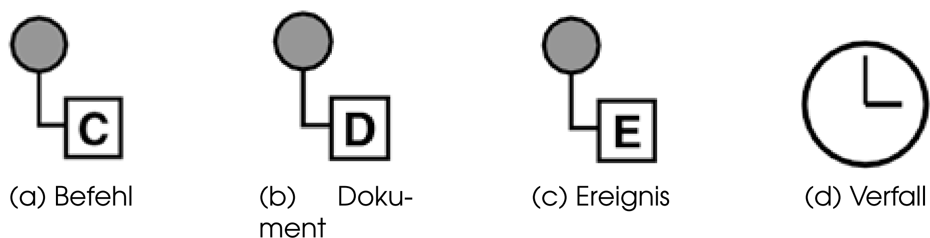
[17] Nachrichten werden erst dann als zugestellt quittiert, wenn die Zwischenspeicherung abgeschlossen ist. Erst danach erfolgt der Weitertransport.

[18] JMS bezeichnet dieses Muster als „persistente“ Zustellung (occ),(Inc98).

Pufferung muss auch ausreichend Speicherplatz vorhanden sein, weil das Messaging-System die Annahme sonst verweigert. In Systemen mit sehr hohem Verkehrsaufkommen kann Abhilfe dadurch geschaffen werden, dass bei wichtigen Nachrichten die Zustellung garantiert und bei wiederkehrenden Ereignisnachrichten der Verlust toleriert wird.

3.4 Nachrichten

Nachrichten können allgemeiner Natur sein oder einen speziellen Typ charakterisieren. Unterschieden werden dabei drei Grundtypen: Die Befehls-, die Dokumenten- und Ereignisnachricht (siehe Abb. 3.16).



(a) Befehl (b) Dokument (c) Ereignis (d) Verfall

Abbildung 3.16: Nachrichtentypen und -attribute.

- **Befehlsnachricht.** Dieses Muster folgt dem bekannten Entwurfsmuster „Befehl" (vgl. (GHJV04)) und ermöglicht, entfernte Funktionsaufrufe durchzuführen. Dabei wird die Anfrage in ein Objekt verpackt und dem entsprechenden Modul übergeben. Der Zweck besteht darin, über Nachrichten Fernfunktionsaufrufe durchzuführen. Dazu werden die Befehle mit ihren Parametern im Body verpackt.

- **Dokumentennachricht.** Mit der *Dokumentennachricht* werden Nutzdaten versendet. Im Gegensatz zur *Befehlnachricht* soll der angesprochene Empfänger keine Aktion ausführen, sondern ihm werden lediglich Daten zur Verarbeitung übergeben.

- **Ereignisnachricht.** Die *Ereignisnachricht* ähnelt der *Dokumentenachricht*, unterscheidet sich jedoch in dem Punkt, dass sie für gewöhnlich

kürzer ist und zur Koordination von Diensten genutzt wird. Sie wird häufig in Verbindung mit dem Muster Publish-Subscribe-Kanal eingesetzt (Abschnitt 3.3.4, S. 61f.).

Messaging stellt im Allgemeinen die Zustellung einer Nachricht an einen Empfänger sicher (siehe Abschnitt 3.3.5, S. 63f.). In einigen Fällen kann es jedoch erforderlich sein, dies von der Dauer der Übertragung abhängig zu machen. Mit dem Muster „**Nachrichtenverfall**" wird dieser Anforderung genügt. Nähere Erläuterungen finden sich in (GH03). Der Nachricht wird im Kopfdatenteil – dem Header – als Attribut eine Verfallszeit mitgegeben. Sollte dieser Zeitraum abgelaufen sein, wird die Nachricht aus dem regulären Transport aussortiert und nicht mehr zugestellt[19].

[19]Für diesen Fall gibt es das Muster *Dead Letter Channel*, welches sich der Verarbeitung dieser verfallenen Nachrichten annimmt.

4 Referenzanwendung

Das Ziel der praktischen Umsetzung ist die Entwicklung eines serviceorientierten Multi-Messaging-Dienstes. Dieser soll am Beispiel von Skype implementiert werden. Skype soll somit als Service einen Gateway in die Skype--Community darstellen. Dem Betreiber ist es freigestellt, mehrere Instanzen von Skype einzubinden[1]. Die interne Nachrichteneinspeisung zum Versand und zum Empfang eines der eingebundenen Nachrichtenmodule erfolgt transparent über den ESB. Darüber hinaus besitzt jedes eingebundene Modul auch eine Monitoring-Komponente, welche der Backend-Logik die Überwachung der Dienste ermöglicht.

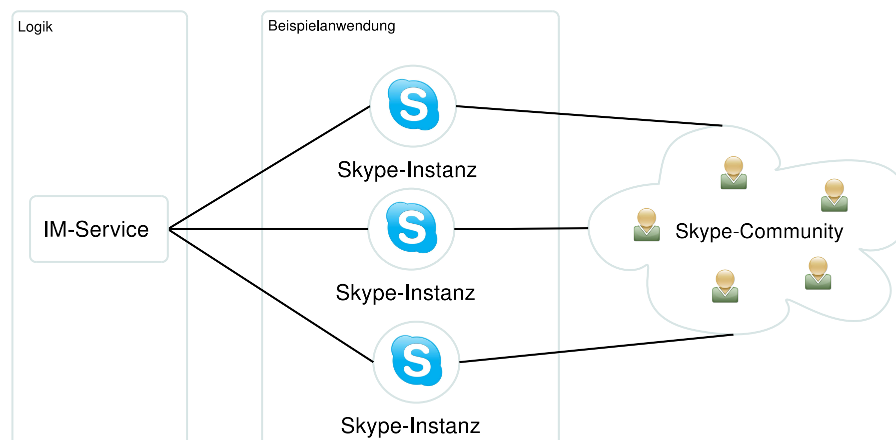


Abbildung 4.1: Gesamtsicht auf das System. Das Backend (links) wird in (Pra08) gesondert behandelt. Die Beispielanwendung (Mitte) ist Teil dieser Arbeit.

[1] Die Schnittstellen sind ausgelegt und erprobt auch für den Einsatz anderer Messaging-Module, beispielsweise ICQ/AIM/MSN/YAHOO etc..

Die Verwaltungslogik, im Folgenden als *IM-Service*[2] bezeichnet, wird in dieser Arbeit keine weitere Erwähnung finden. Sie wird von Stefan Pratsch in (Pra08) separat behandelt (Abb. 4.1, S. 67).

Diese Anwendung soll Teil eines Dienstangebots sein und im Sinne von SOA nur eine primäre Funktion erfüllen – das „Instant-Messaging". Das *Instant-Messaging* bezeichnet eine Art der Kommunikation, welche ein verzögerungsfreies Übermitteln der Nachricht vom Sender zum Empfänger ermöglicht. Sender und Empfänger sind dabei Menschen – selten Stellvertreter – wie in diesem Fall der Dienst.

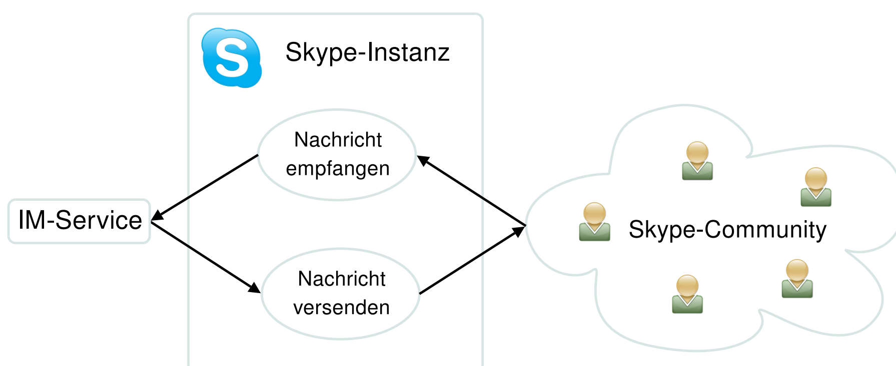


Abbildung 4.2: Der allgemeine Anwendungsfall beim Einsatz des Messaging-Dienstes befasst sich mit dem Versenden und Entgegennehmen von Nachrichten.

Bei der Übermittlung der Nachrichten wird vorausgesetzt, dass Sender und Empfänger zur gleichen Zeit im Netzwerk angemeldet sind. Dieses Netzwerk bezeichnet bei einer zentralen Struktur den Adressraum, in welchem sich alle Nutzer dieses Protokolls befinden. Als Beispiele seien hier ICQ/AIM/Yahoo/MSN und Skype angeführt. Im Gegensatz dazu bezeichnet der Begriff Netzwerk im Zusammenhang mit einer dezentralen Struktur – so z.B. IRC/JABBER/SILC – den abgegrenzten privaten Adressraum, in dem sich der Nutzer befindet. Dabei können viele Nutzer online sein. Sie erhalten jedoch dadurch nicht unbedingt die Möglichkeit, über die Grenzen

[2] „Jeglicher Informationsaustausch erfolgt personalisiert und nicht anonym wie bei einem herkömmlichen Bot. Damit ermöglicht dieser Service die Verknüpfung von eingehenden und ausgehenden Anfragen mit bestehenden persönlichen Profilen der Anbieter-Portale. Das wird über eine ausgelagerte Komponente erledigt – dem Backend." (Pra08)

des eigenen Netzwerkes hinaus, mit anderen Nutzern in Kontakt zu treten (Tabl. 4.1).

Dienst	Art der Kommunikation	Nachricht hinterlassen	Struktur [1]	Adressraum der Buddy-ID
ICQ	asynchron	ja	zentral	global
AIM	asynchron	ja	zentral	global
YAHOO	asynchron	ja	zentral	global
Jabber	asynchron	ja	dezentral	privat
IRC	(a)synchron	möglich	dezentral	privat
SILC	synchron	nein	dezentral	privat
Skype	synchron	nein	zentral	global

[1] Zentral: Es ist nicht möglich, ein lokales Netzwerk dieser Art einzurichten. Dezentral: Server-Software ist verfügbar.

Tabelle 4.1: Vergleich von Instant-Messaging-Diensten.

Die Mitglieder in einer Community werden grundsätzlich in zwei Rollen eingeteilt – dem Buddy und den Kontakten (des Buddy). Der Buddy ist die eigene ID, mit der man sich im Netzwerk bewegt. Die Kontakte sind alle IDs, mit denen eine Kommunikation erfolgen kann. Die IDs sind zum Teil numerisch als auch zum Teil alphanumerisch[3].

Spam

Mit der Möglichkeit der einfachen Kommunikation verbinden sich sowohl Fehler- als auch Missbrauchspotentiale. Auf den Missbrauch zum Versenden von SPAM wird nicht eingegangen, weil es den Rahmen dieser Arbeit sprengen würde. Allerdings können Fehler in der Konzeption sehr schnell Nachrichten zu SPAM werden lassen. Das ist dann die Folge der Missachtung sekundärer Informationen, wie des Online-, des Authentifizierungs- und des Störungs-Level-Status.

Der *Online-Status* kennt nur zwei Zustände: **Offline** und **online**. Er dient dem Auf- oder Abbau einer Verbindung. Dieser Status ist physischer Natur.

[3] So sind IDs bei ICQ und SILC numerisch, aber im IRC und bei Skype alphanumerisch.

Der *Authentifizierungsstatus* unterscheidet zwischen *beglaubigt* bzw. *nicht beglaubigt* und *blockiert* bzw. *nicht blockiert*. Beglaubigt werden Kontakte. Die Beglaubigung befähigt zur Einsicht des Online-Status eines Kontaktes. Anderenfalls ist nicht ersichtlich[4], ob ein Kontakt online oder offline ist. Deshalb wird er meist als offline bewertet.

- **Beglaubigt.** Dem Kontakt ist es erlaubt, den Online-Status einzusehen.
- **Nicht beglaubigt.** Dem Kontakt ist es verwehrt, den Online-Status einzusehen. Der Kontakt wird als *offline* geführt.
- **Blockiert.** Dem Kontakt ist jegliche Kommunikationsaufnahme verboten.[5]
- **Nicht blockiert.** Dem Kontakt ist jegliche Kommunikationsaufnahme erlaubt. Das ist in der Regel der Startzustand bei der Inbetriebnahme eines Buddy.

Der *Status des Störungs-Level* bestimmt die individuelle Toleranz gegenüber eventuellen Störungen durch Kontakte. Dabei wird unterschieden zwischen:

- **Verfügbar.** Es handelt sich um einen neutralen Zustand, der eine Kontaktaufnahme ermöglicht. Visuelle und akustische Signale, die dem Nutzer in Form von Hinweisfenstern und Tönen ein neues Ereignis mitteilen sollen, sind auf einem sehr hohen Niveau. Sie lenken mit sehr hoher Wahrscheinlichkeit die Aufmerksamkeit des Nutzers auf sich und führen so zu einer Unterbrechung seiner Tätigkeit.
- **Kontaktsuchend.** Dieser Zustand ist dem Status *verfügbar* ähnlich. Allerdings weist der Nutzer explizit darauf hin, dass er jede Kontaktaufnahme begrüsst[6]. Damit verbunden sind temporäre Veränderungen

[4]Bei Skype wird ein Fragezeichen über dem Kontakt eingeblendet.
[5]In der Linuxversion 2.0.72 von Skype wird infolge der Aufnahme des Kontaktes in die Blockierungsliste gleichzeitig die Authentifizierung entzogen. Die Skype-API lässt vermuten, dass damit beide Aktionen nur in der Oberfläche zusammengeführt sind ((Lim)). Bei ICQ bzw. Pidgin als Client sind **blockieren** und **authentifizieren** zwei unterschiedliche Aktionen.
[6]Beim Skype-Client heißt dieser Status *skypeme* – http://support.skype.com/index.php?_a=knowledgebase&_j=questiondetails&_i=442

des Authentifizierungsstatus. Jeder (fremde) Kontakt darf den Buddy ansprechen, obwohl es die Einstellungen sonst nicht zulassen würden.

- **Abwesend.** Dieser Zustand zeigt an, dass während einer gewissen Zeitspanne – bei Skype standardmäßig 5 Minuten (einstellbar) – keine Aktivität[7] des Benutzers registriert wurde. Das Hinweisniveau gleicht dem Status *verfügbar*.

- **Nicht verfügbar.** Dieser Status bezeichnet eine längere Phase der Abwesenheit. Er tritt automatisch nach einer weiteren Inaktivitätsdauer nach dem Status *abwesend* ein. Dieser Zeitraum kann, ähnlich wie zuvor, entsprechend den jeweiligen Wünschen verändert werden.

- **Nicht stören.** Um erreichbar zu sein und mitzuteilen, dass man im Prinzip nicht gestört werden möchte, kann dieser Zustand gewählt werden. Die visuellen und akustischen Signale sind hierbei bei einer Kontaktaufnahme auf ein Minimum reduziert. So ist z.B. bei Skype lediglich eine Veränderung des Programmbildes in der Systemleiste zu beobachten.

- **Unsichtbar.** Dieser Modus wird unter Skype auch als *als offline anzeigen* bezeichnet. Er behält den Online-Status bei, suggeriert allerdings allen Kontakten, dass der Buddy offline gegangen sei. Der Nutzer nimmt hier die Position des Beobachters ein, da bei Beachtung des Online-Status keiner seiner Kontakte den Versuch einer Kontaktaufnahme unternehmen wird. Sollte jedoch ein Kontakt diesen Status ignorieren und versuchen, eine Kommunikation zu initiieren, gelingt dies auch, da der Online-Status nur anscheinend *offline* ist.

Die Status *abwesend* und *verfügbar* werden selbstständig vom Client gesetzt, jedoch können beide auch manuell gesetzt werden.

Diese sekundären Informationen – der *Online-*, der *Authentifizierungs-* und der *Störungs-Level-Status* – dienen der Feststellung, ob ein Kontakt

[7] Diese Einschätzung hängt vom Chatprogramm ab. Pidgin – ein Multiprotokollclient – verbindet z.B. mit dem Begriff Aktivität Oberflächenereignisse (u.a. Mausbewegungen und Tastatureingaben).

Status	Hinweise	Veränderung des Authentifizierungs-Level
verfügbar abwesend nicht verfügbar unsichtbar	alle	nein
kontaktsuchend	alle	ja
nicht stören	kaum	nein

Tabelle 4.2: Übersicht der Störungs-Level-Status.

erreichbar ist bzw. ob er erreicht werden möchte. Damit kann ein versehentliches Versenden von SPAM verhindert werden. So kann z.B. die Nichtbeachtung des Online-Status dazu führen, dass der Kontakt, wenn er das nächste Mal online geht, mit Nachrichten überhäuft wird. Es muss also immer, auf der Grundlage aller technischen Möglichkeiten, in Betracht gezogen werden, welche Wünsche der Nutzer hat. Zu beachten ist weiterhin, dass nicht jedes IM-Protokoll alle Zustände unterstützt.

4.1 Architektur

Der Skype-Service versteht sich als *Composite-Service* (Kapitel 2.2, S. 12ff.). Er besteht aus zwei *Component*-Diensten – dem Skype-Anwendungsadapter und dem Skype-Konnektor (siehe Abb. 4.3, S. 73).

Die Architektur besteht aus dem IM-Service und den Diensten. Die Dienste werden von einem *Serviceprovider* ausgeliefert. Dieser startet bzw. stoppt die Dienste und liefert Statusinformationen an den IM-Service. Der Serviceprovider, der IM-Service und die ausgewählten Dienste [8] kommunizieren über einen Enterprise Service Bus (Kapitel 2.5, S. 26) auf der Grundlage des Protokolls JMS. Deshalb nutzen der Serviceprovider und der Skype--Konnektor zwecks Kommunikation mit dem ESB einen JMS-Kanaladapter (siehe Abb. 4.4, S. 73).

[8] Nicht alle Dienste benötigen die Steuerung durch den IM-Service. Sie werden dann nur über den Serviceprovider gestartet und gestoppt.

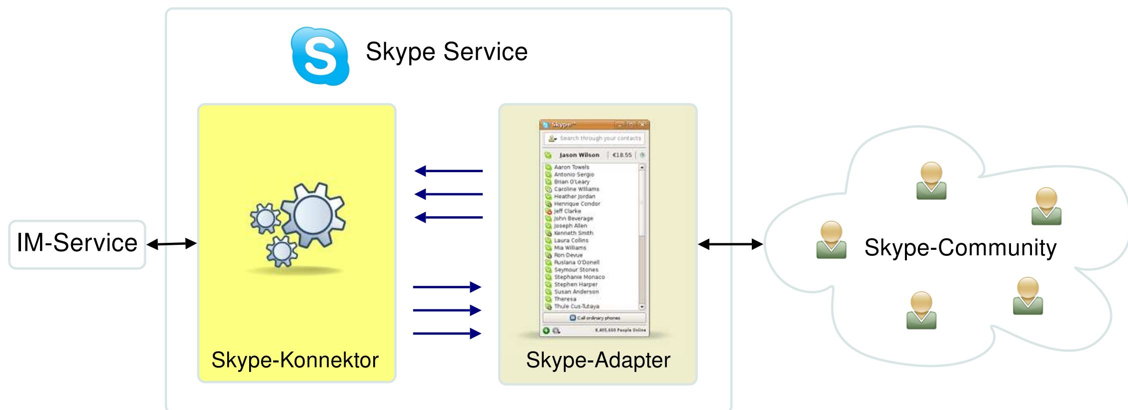


Abbildung 4.3: Die logische Sicht mit Kapselung der Anwendung Skype in einem Composite-Service, bestehend aus dem Konnektor und dem Adapter.

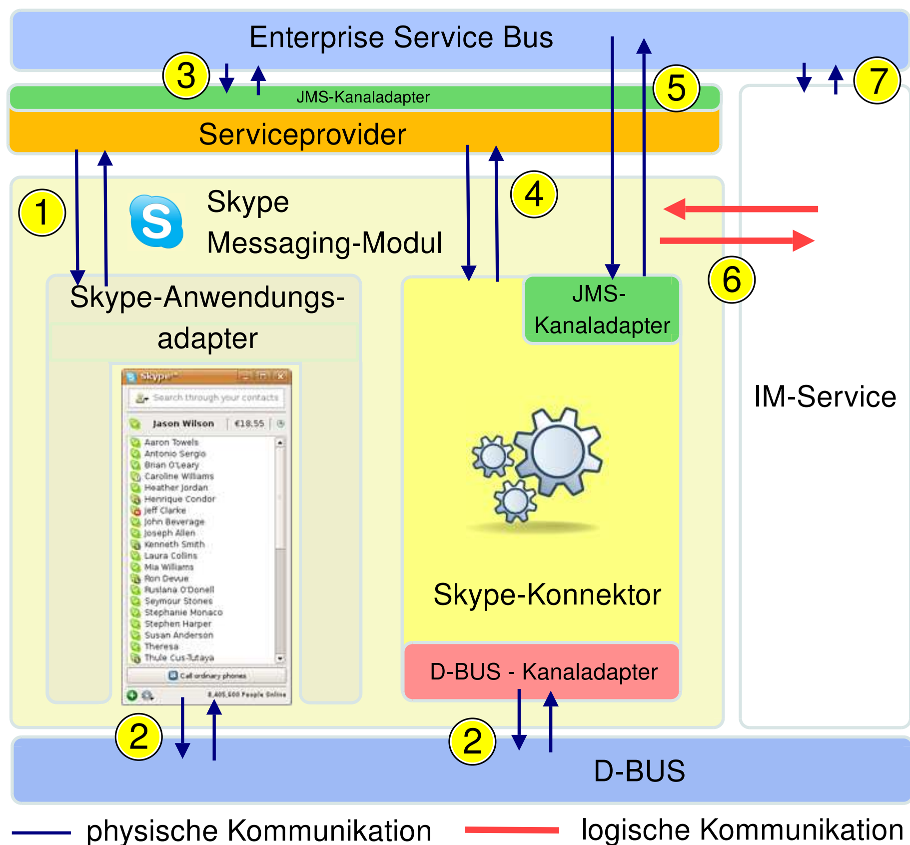


Abbildung 4.4: Kommunikation der Komponenten – logische und physische Sicht (Kommunikationspunkte siehe unten).

Erläuterung der Kommunikationspunkte:

1. Darstellung des Steuerungskanals (über eine interne Objektreferenz) zwischen *Serviceprovider* und dem Dienst *Skype--Anwendungsadapter*. Über diesen Kanal werden wird der Service gestartet, gestoppt und Informationen erfragt.
2. Versand der Ereignisnachrichten des Skype-Clients mithilfe des D-BUS an den Konnektor. Auf der anderen Seite werden Ereignisnachrichten des Skype-Clients empfangen, und Befehlsnachrichten zur Steuerung des Clients und Dokumentennachrichten als Textnachrichten an die Kontakte des Buddy versendet.
3. Aufbau eines Steuerungskanals zum IM-Service mithilfe des ESB. Darüber werden Befehlsnachrichten zur Steuerung der Dienste empfangen und Benachrichtigungen weitergegeben.
4. Analog 1) (s.o.).
5. Registrierung als Konsument an der Warteschlange vom IM-Service für zu versendende Nachrichten und als Produzent an der Warteschlange an den IM-Service für empfangene Nachrichten.
6. Transport der Nachrichten aus 3) und 5) (s.o.).
7. Nutzung des Dienstes *Skype*: Versand von Dokumentennachrichten an Kontakte in das Skype-Netzwerk und Empfang von Nachrichten aus dem Skype-Netzwerk.

Die Steuerung des Skype-Clients [9] übernimmt der Skype-Konnektor. Dieser gibt seine Befehle an den D-Bus[10] (Systemnachrichtenbus unter Linux) und entnimmt auch von dort die Antworten. Die Anwendung Skype arbeitet nicht nur reaktiv, sondern auch *ereignisgetrieben* und informiert eigenständig den Konnektor über neue Ereignisse (z.B. über den Online-/Offlinestatus, siehe S. 69 und Abb. 4.4, S. 73). Die eingegangenen Informationen werden aufbereitet[11] und an den IM-Service weitergegeben.

[9] Gemeint ist das eigentliche Programm, welches von der Website des Herstellers heruntergeladen werden muss und den einzigen Zugang zum Skype-Netzwerk gewährt.
[10] http://freedesktop.org/wiki/Software/dbus –
[11] Transformation der Daten in ein eigenes Format.

Ein Beispiel dafür wäre, dass ein Kontakt online geht, der IM-Service über dessen Online-Status mit einer Ereignisnachricht (Kapitel 3.4, S. 64ff.) informiert wird und daraufhin mit einem Nachrichtenversand an den Kontakt reagiert. Nach dem Versand informiert Skype den Konnektor über das erfolgreiche Versenden der Nachricht (Abb. 4.5).

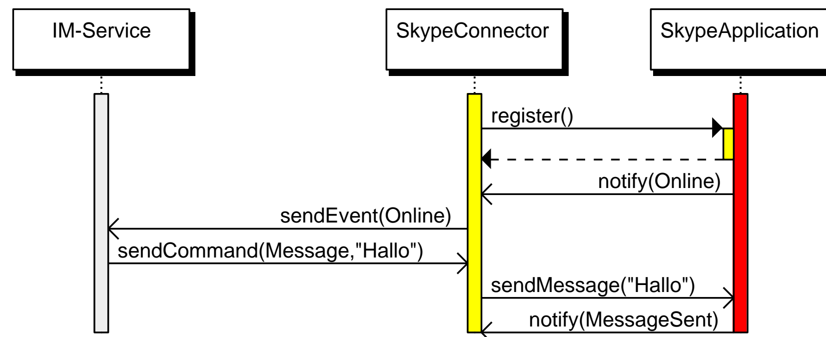


Abbildung 4.5: Praktische Ausprägung eines ereignisgetriebenen Konsumenten am Beispiel des Skype-Konnektors.

Die Integration der bestehenden Skype-Applikation benötigt einen *Legacy Adapter* (Abschnitt 2.7.4, S. 41). Auf diese Weise kann die Anwendung gesteuert werden. Der Adapter teilt sich auf beide Dienste auf. Zum einen übernimmt der Skype-Anwendungsadapter die Prozesssteuerung auf Betriebsystemebene – das Starten, das Stoppen und das Überwachen der Anwendung – und zum anderen integriert der Skype-Konnektor die Funktionalität nahtlos als Service in Form eines Geschäftslogikadapters (Abschnitt 3.3.2, S. 59f.).

4.2 Auflösung der Dienstabhängigkeiten

Dienste können voneinander abhängig sein. Ein Dienst kann es erforderlich machen, dass ein weiterer aktiv ist. Am Beispiel der beiden Skype-Dienste wird das sichtbar. Jeder Service führt eine Liste von Abhängigkeiten, so dass beim Starten der Dienste immer auf eine korrekte Reihenfolge geachtet wird (siehe Abb. 4.6).

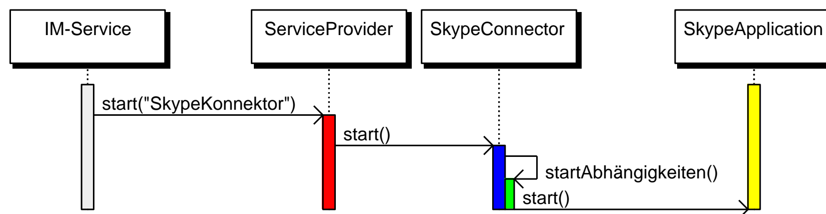


Abbildung 4.6: Ablauf der Auflösung der Abhängigkeiten beim Starten des Dienstes *Skype-Konnektor*.

Beim Start der Anwendung ist nur der *Serviceprovider* einsatzbereit. Auf Anweisung des IM-Service starten die einzelnen Dienste unter Beachtung der Abhängigkeiten. Es werden iterativ alle benötigten Dienste, erst danach der eigentliche Service gestartet. Deshalb muss in diesem Fall aufgrund der Abhängigkeit des Dienstes *Skype-Konnektor* vom *Skype--Anwendungsadapter* nur der Konnektor explizit gestartet werden (siehe auch Abb. 4.6 und Abb. 4.7).

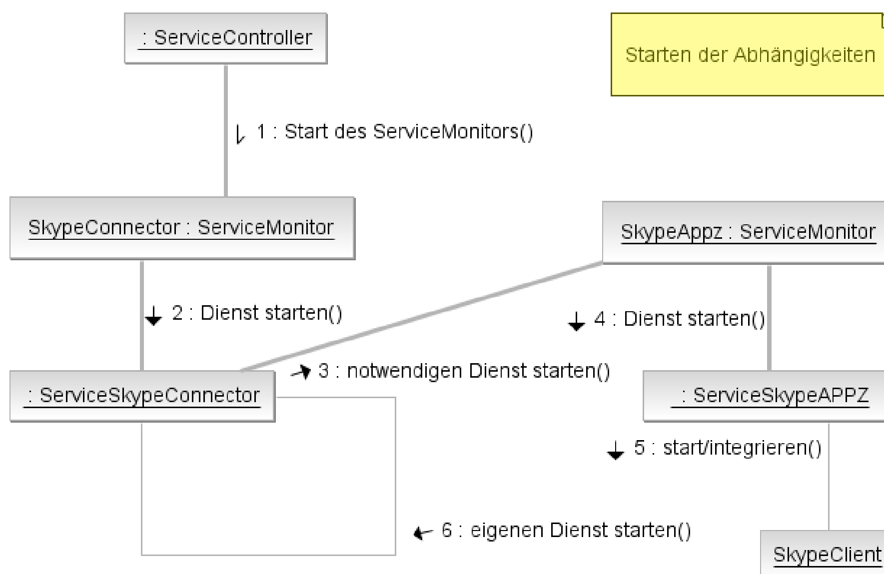


Abbildung 4.7: Ablauf der Auflösung der Abhängigkeiten beim Starten des Dienstes *Skype-Konnektor* unter Einbeziehung von Service-Monitoren.

Sollte beim Start der Skype-Anwendung ein Fehler auftreten, wird der Konnektor nicht starten können und beide Dienste werden eine Fehlermeldung an den IM-Service weiterleiten. Diese Abhängigkeiten bestehen nicht nur beim Start, sondern auch bei Beendigung der Dienste (auch in Fehlersituationen). Somit kann die Abhängigkeit als beiderseitig angesehen werden. Sollte ein Dienst ausfallen, wird dies von anderen durch *Polling* erkannt. In diesem Fall ist der Einsatz von Polling wichtig, da Ausfälle nicht vorhersehbar sind und man sich nicht darauf verlassen kann, dass diese Dienste die von ihnen abhängigen Dienste informieren. Da das Polling in der Anwendung jedoch intern abläuft, ist es sehr ressourcensparend.

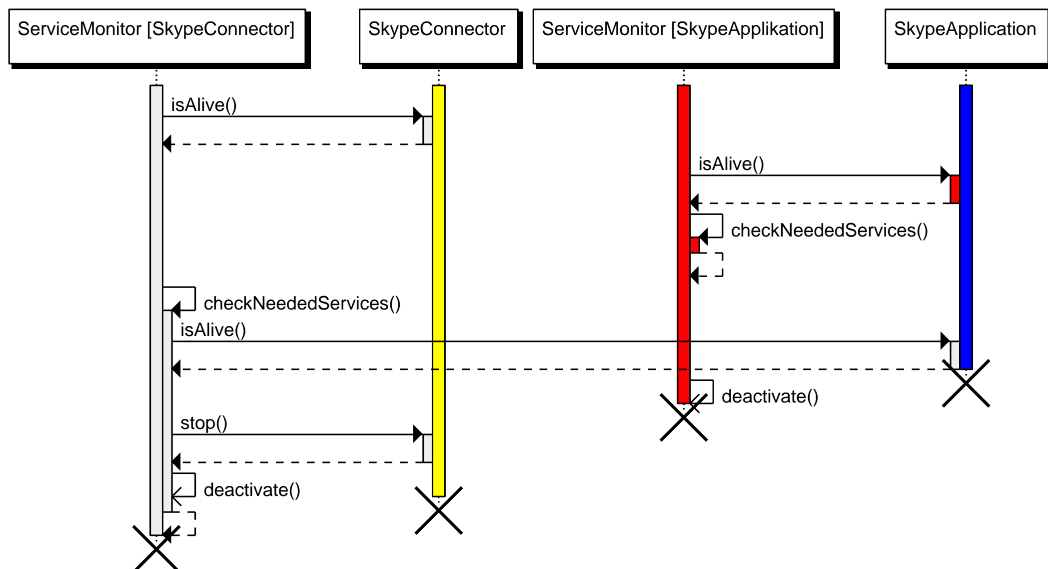


Abbildung 4.8: Selbstständige Beendigung beim Ausfall eines notwendigen Dienstes.

Jeder der Dienste erhält zwecks Überwachung einen *Service-Monitor*[12]. Diesem wird per Dependency Injection sein Dienst zugeteilt. Der Monitor hat die Aufgabe, den Dienst zu überwachen. Sollte einer der für den Service notwendigen Dienste ausfallen, wird dieser von seinem Service-Monitor angehalten.

[12]Hier sei auf das Muster *Beobachter* der „Gang of Four“ verwiesen.

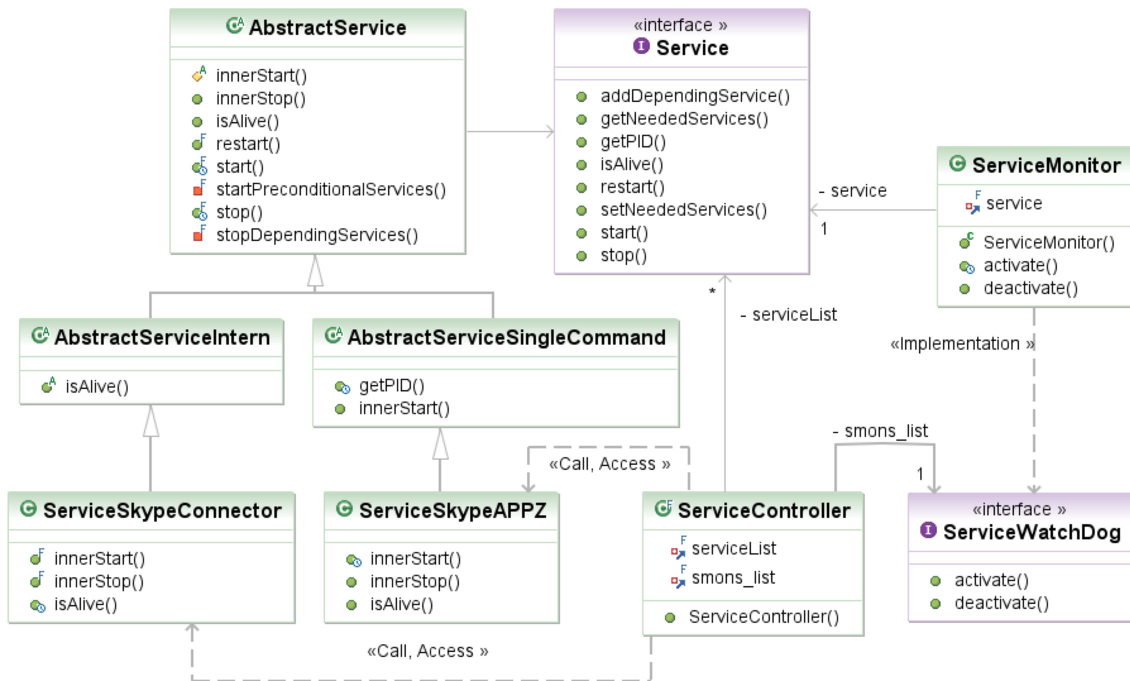


Abbildung 4.9: Zusammenhänge und Struktur der Dienste. Der *Servicecontroller* als Hauptklasse enthält eine Liste der Dienste und (de)aktiviert diese auf Anfragen.

Für einen Dienst ist es erforderlich, das Verhalten bei einem Start, bei einer Beendigung und einer Statusabfrage festzulegen. Es werden dabei zwei Serviceklassen unterschieden: Der *interne* Dienst, welcher eine homogene Javastruktur besitzt und der *externe* Dienst, welcher einen Adapter für einen externen Prozess einsetzt (Abb. 4.9, S. 78). Externe unterscheiden sich von internen Diensten durch eine plattformabhängige Prozesskennung[13]. Das ist deshalb notwendig, um Prozesse, die von diesen Diensten umhüllt werden, sicher kontrollieren zu können. Zur Kontrolle gehört u.a. auch das sichere Entfernen dieser aus dem Prozessraum. Dabei kann der Fall eintreten, dass ein Prozess nicht mehr nur beendet, sondern sogar „getötet"[14] werden muss. So war es u.a. auch nicht möglich, beste-

[13] PID

[14] Prozesse werden normalerweise dadurch beendet, dass sie auf Prozesssignale reagieren. Sollte einer der Prozesse nicht darauf reagieren, wird er mit Gewalt aus dem Prozessraum entfernt.

hende Prozesse in Java ohne den Bruch der Plattformunabhängigkeit, die diese Programmiersprache Java auszeichnet, anzusprechen[15].

Die Integration eines neuen Service erfordert zunächst die Unterscheidung nach *internen* und *externen* Diensten. Im Weiteren müssen der Start und die Beendigung des Service klar definiert werden (Abb. 4.9, S. 78).

Im Folgenden werden Aspekte betrachtet, die für einen hochverfügbaren Betrieb von Bedeutung sind.

4.3 Robustheit und Skalierbarkeit

Die Implementierung konzentriert sich nur auf das Versenden und Empfangen von Textnachrichten. Fähigkeiten von Skype[16] darüber hinaus werden zu diesem Zeitpunkt nicht genutzt. Eine sehr wichtige Aufgabe bestand darin, eine robuste und skalierbare Lösung zu finden.

Skalierbarkeit

Unter Skalierbarkeit wird der problemfreie Einsatz der Anwendung unter steigenden Anforderungen verstanden. Im Rahmen dieser Anwendung können die Nutzerzahlen und somit der Verkehr zwischen Skype--Anwendung und IM-Service steigen. Dadurch kann es zu einer Überlastung kommen, dem jedoch entgegengewirkt werden kann, indem das Konzept des *Load Balancing*[17] aufgegriffen wird. Die Applikation Skype unterstützt die Nutzung simultaner Verbindungen unter gleicher Kennung. Das heißt, man kann mit demselben Buddy mehrmals im Skype-Netzwerk angemeldet sein. Ein gutes Beispiel bietet Skype mit dem Buddy „echo123". Dieser ist für den individuellen Test der Audioeinstellung eingerichtet, den *alle* Teilnehmer kontaktieren können und dazu auch in der Lage sein müssen. Würde nun die Last, hervorgerufen durch das Skype-Netzwerk, zu groß

[15] Notwendig, um bestehende Prozesse zwecks Einsparung von Systemressourcen einzubinden und keine doppelten Prozesse zu erzeugen.
[16] Sprach- und Videofähigkeiten, wie Telefonie, Voicemail und Videokonferenzen.
[17] Verteilung der Anwendungslast auf mehrere gleichartige Komponenten im System.

werden, um sie mit der vorhandenen Anzahl an Instanzen zu verarbeiten, könnten auf einfache Art und Weise weitere Skype-Instanzen in den Pool eingegliedert werden. Die Verteilung der Nachrichten des IM-Service erfolgt durch die Weitergabe an eine Warteschlange. Die Warteschlange fungiert dabei als *Punkt-zu-Punkt*-Kanal (Abschnitt 3.3.3, S. 60f.), der zwei virtuelle Endpunkte miteinander verbindet. Die Entnahme doppelter Nachrichten durch die Instanzen wird entsprechend dem Muster *Konkurrierende Konsumenten* (Abschnitt 3.2.2, S. 56) verhindert.

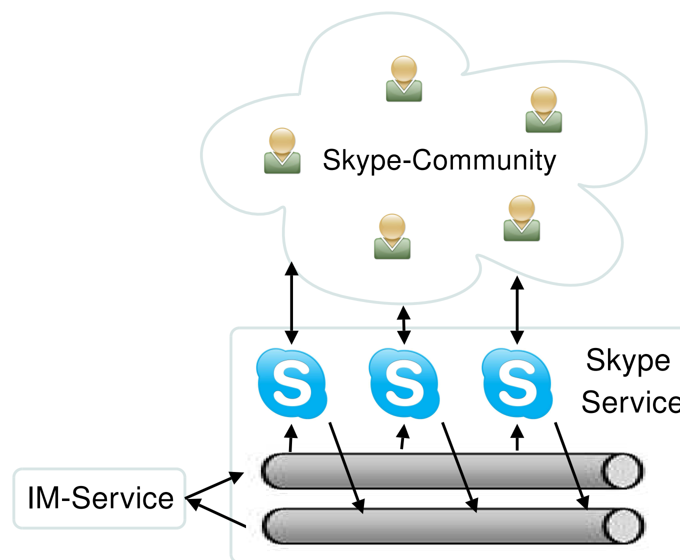


Abbildung 4.10: Skype-Dienst, verteilt auf mehrere Instanzen.

Ein weiterer Nachrichtenkanal, ebenfalls in Form einer Warteschlange, führt von den Skype-Instanzen zum IM-Service (Abb. 4.10). Dieser Kanal besitzt einen Puffer, so dass für eine gewisse Zeit nicht unbedingt Konsumenten registriert sein müssen, ohne dabei einen Nachrichtenverlust zu befürchten.

Robustheit

Der Begriff Robustheit drückt aus, dass weder beim Transport noch bei der Übergabe der Nachrichten diese verloren gehen dürfen. Durch den

Einsatz eines Enterprise Service Bus[18] und den darin verankerten *Enterprise Integration Pattern* (Kapitel 3, S. 45) können wir sicher sein, dass nach dem Muster *Garantierte Zustellung* (Abschnitt 3.3.5, S. 63) keine Nachricht auf dem Transport verloren geht. Alle Komponenten des Systems arbeiten transaktionsorientiert: Die Logik und der Skype-Konnektor mit einem JMS-Kanaladapter quittieren erst bei einer erfolgreichen Annahme/Übergabe der Nachricht. Die Skype-Anwendung arbeitet nach ähnlichen Prinzipien[19]. Dabei wurde Folgendes beobachtet:

- Nach dem gescheiterten Versuch der Versendung von Nachrichten und einem folgenden Neustarten der Anwendung – Beenden der aktuellen Instanz der Skype-Anwendung und Start einer neuen Instanz – wird erneut der Versuch unternommen, erfolglos gesendete Nachrichten zuzustellen. Es sollte eine Verbindung zum Kontakt aufgebaut werden, um auf diese Weise die Nachrichten zu übermitteln.
- Bei längeren Tests[20] zwischen Skype-Instanzen ist kein Datenverlust aufgetreten.
- Beim Neustarten einer empfangenen Skype-Anwendung konnte der Nachrichtenfluss ohne Probleme und Verluste fortgesetzt werden.

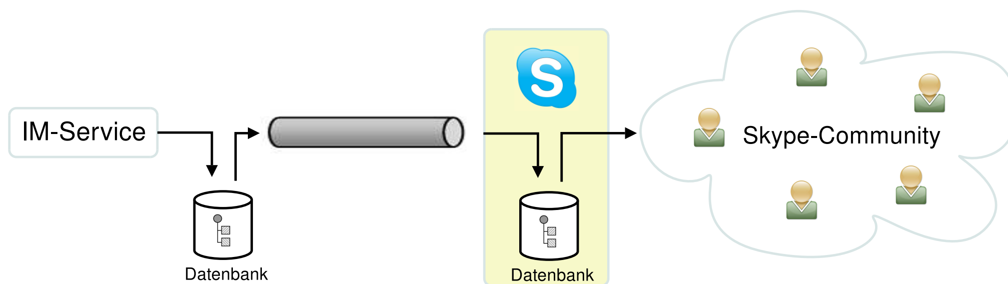


Abbildung 4.11: Prinzip der *Garantierten Zustellung* auf dem Weg *zu* einem Kontakt.

[18] Activemq v5.1 – `http://activemq.apache.org`
[19] Der Autor kann in diesem Zusammenhang nur das von ihm beobachtete Verhalten einschätzen. Durch die proprietäre Lizenz und dem nicht öffentlich zur Verfügung stehendem Quelltext kann keine Sicherheit über die Korrektheit der Einschätzung gegeben werden.
[20] 100.000 Nachrichten sequentiell.

An dieser Stelle muss angemerkt werden, dass zu diesem Zeitpunkt die Größe des Empfangs- bzw. Sendepuffers auf der Grundlage experimenteller Analysen noch nicht bestimmt worden ist. Somit kann das Verhalten des Skype-Clients in solch einem Grenzfall nicht vorhergesagt werden. Allerdings lässt sich über die Skype-API die Anzahl ungelesener Nachrichten abfragen und somit vom Skype-Konnektor eine Beschränkung setzen. Dadurch wird die Wahrscheinlichkeit, in eine solche Situation zu geraten, wesentlich geringer. Eine Beschränkung kann so erreicht werden: Bei der Überschreitung einer festgelegten Grenze wird der *Online-Status* des Skype-Clients geändert, bei einer eventuellen Unterschreitung wird erneut ein Wechsel vorgenommen.

Fehlertoleranz

Fallen innerhalb dieser Anwendung als Teil des Systems (IM-Service, Skype--Dienst und ESB) einzelne Komponenten aus, wird darauf mit vereinbarten Notfallroutinen reagiert.

Beim Ausfall des ...

- ... **JMS-Kanaladapters** des Skype-Konnektors und der damit einhergehenden Verbindung zum Enterprise Service Bus wird die Skype-Instanz deaktiviert und im Falle einer erfolgreichen Wiederverbindung reaktiviert. Die Aktivierung bzw. Deaktivierung erfolgt über einen Wechsel des *Online-Status*. Der Ausfall des Kanaladapters bedeutet normalerweise den Ausfall des ESB (Abb. 4.12, S. 83).

- ... **D-BUS-Kanaladapters** des Skype-Konnektors wird der Skype--Anwendungsadapter beendet, um die Verbindung des Skype--Clients mit dem Netzwerk zu trennen. Eine andere Möglichkeit gibt es in dieser Situation nicht, weil die Kommunikation mit dem Skype-Client über eben diese ausgefallene Schnittstelle abläuft (Abb. 4.13a, S. 84).

- ... **Skype-Clients** werden zwei Fälle unterschieden, einerseits das „Einfrieren" der Anwendung und andererseits die unerwartete Beendigung des Clients. Das „Einfrieren" kann nur durch eine gewaltsame

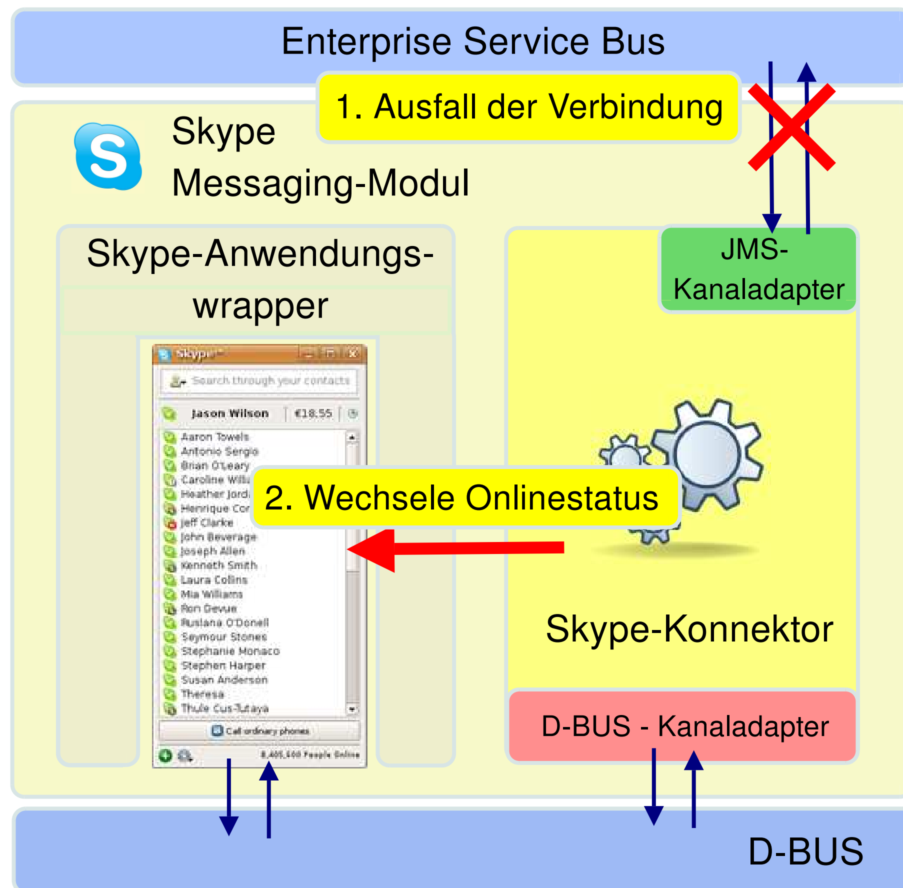
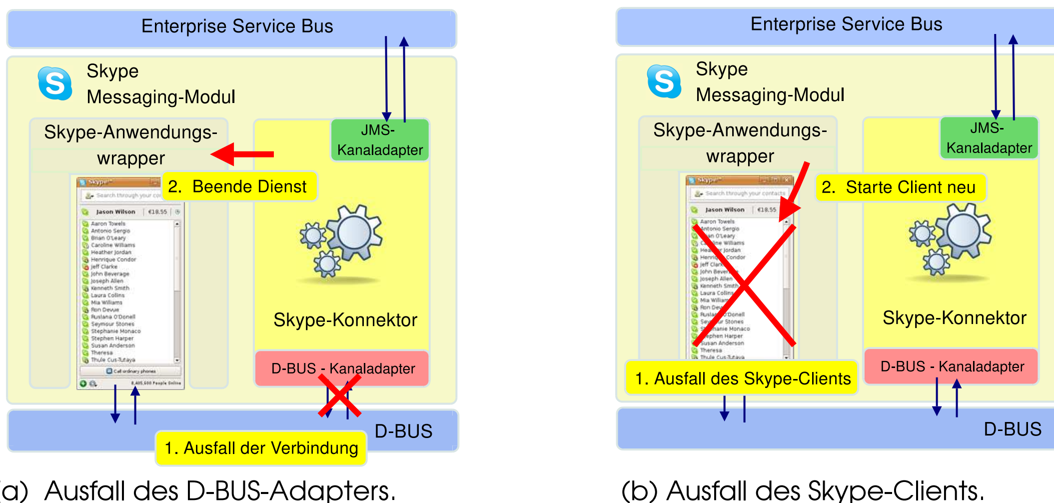


Abbildung 4.12: Ausfall des JMS-Kanaladapters.

Terminierung des Prozesses beendet werden. Da der Service-Monitor lediglich den Prozess beobachtet und das „Einfrieren" ein Merkmal des Verhaltens ist, wird er davon nichts zur Kenntnis nehmen. Etwas bemerken wird lediglich der Kommunikationspartner, nämlich der Skype-Konnektor. Er wird beim Auftreten dieses Fehlers einen Neustart des Skype-Anwendungsadapters mit Hilfe seines Service-Monitors[21] durchführen (Abb. 4.13b, S. 84). Beim Absturz im zweiten Fall wird der Service-Monitor des Skype-Anwendungsadapters selbstständig einen Neustart durchführen.

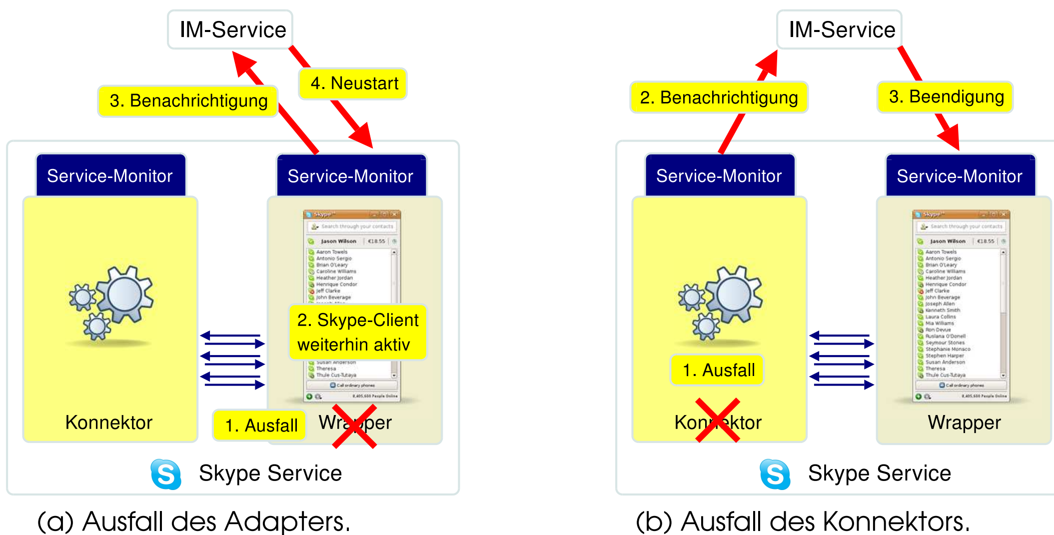
[21] Service-Monitor des Skype-Anwendungsadapters.



(a) Ausfall des D-BUS-Adapters. (b) Ausfall des Skype-Clients.

Abbildung 4.13: Ausfall des D-Bus-Adapters und des Skype-Clients.

- ... **Skype-Anwendungsadapters** genügt ein einfaches Neustarten desselben. Ein Absturz hat deshalb keine wesentlichen Auswirkungen auf den Betrieb des Systems, da er nur zum Starten und Beenden des Skype-Clients erforderlich ist (Abb. 4.14a).



(a) Ausfall des Adapters. (b) Ausfall des Konnektors.

Abbildung 4.14: Ausfall des Skype-Adapters und des -Konnektors.

- ... **Skype-Konnektors** ist einer der neuralgischsten Punkte des Skype--Dienstes betroffen. Wenn der Konnektor ausfällt und der Skype-Client weiterhin online ist, gehen auf jeden Fall Informationen verloren. Weil

der D-BUS keinen Nachrichtenpuffer besitzt, gehen die Daten beim Ausfall des Empfängers verloren. In diesem Fall wird der IM-Service durch den Service-Monitor über den Ausfall des Skype-Konnektors informiert (Abb. 4.14b, S. 84).

Die exemplarisch dargestellten Ausfallszenarien beschreiben einerseits den Handlungsablauf beim Auftreten von Fehlern. Andererseits können so auf einfache Art und Weise Komponenten des Skype-Dienstes ausgewechselt werden.

5 Zusammenfassung

Das angestrebte Ziel dieser Diplomarbeit war die serviceorientierte Implementierung eines Multi-Messaging-Dienstes in Java mit Skype. Dieses Ziel ist mithilfe des Einsatzes moderner Entwurfsmuster erreicht worden (siehe Kapitel 2.7, *SOA-Entwurfsmuster*, Seite 35 - 43 und Kapitel 3, *Enterprise Integration Pattern*, Seite 45 - 65).

Es wurde ein IM-Gateway zur Übermittlung von Nachrichten unter Zuhilfenahme der Skype-Kommunikationsplattform erstellt. Mit einem Skype-Buddy ist der Empfang offener Fragen und der Versendung von weiteren Antworten auf gestellte Fragen aus der hiogi-Community möglich (siehe dazu Kapitel 4, S. 67ff.).

Dabei wird die zunehmende Verknüpfung und Einbettung einzelner Kommunikationsmittel in ein Gesamtsystem zur besseren Erreichbarkeit der Menschen angestrebt.

Die im Kap 4, Abb. 4.1, S. 67 dargestellte Beispielanwendung soll Teil eines Dienstangebots sein und im Sinne von SOA nur eine primäre Funktion erfüllen – das „Instant-Messaging". Eine sehr wichtige Aufgabe bestand darin, eine robuste und skalierbare Lösung zu finden.

Im Rahmen der Zusammenarbeit mit *hiogi* gibt es für die Zukunft mindestens zwei Richtungen, in die eine Weiterentwicklung stattfinden kann. Das betrifft zum einen den Ausbau der Fähigkeiten, die Skype bietet und zum anderen die Einbeziehung weiterer Instant-Messaging-Plattformen.

Der erste Weg ist aufgrund der Vertrautheit des Autors mit Skype und der angebotenen API kurzfristig der erfolgversprechendere. Skype ist hochgradig skalierbar, äusserst leistungsfähig und bietet z.B. erweiterte Fähigkeiten wie Telefonie, das Versenden von Sprachnachrichten, die Nutzung für Videokonferenzen. Durch Skype ergibt sich die Möglichkeit zur Transferierung von Geld. Zudem ist eine hohe Verbreitung und zunehmende Akzeptanz

auch in der Geschäftswelt zu verzeichnen, vor allem durch die Öffnung von Skype gegenüber Sicherheitsverantwortlichen in Firmennetzen. Durch Portierungen von Skype auf mobile Plattformen scheint dies auch viel versprechend und investitionssicher zu sein, da eine Firma hinter dieser Software steht, welche deren Entwicklung absichert und seit Jahren erfolgreich auf dem Markt tätig ist.

Da diese Anwendung unter dem Gesichtspunkt einer wirtschaftlichen Ausrichtung entwickelt worden und *hiogi* ein wichtiger Kooperationspartner ist, wird der zweiten Möglichkeit, nämlich der Einbeziehung weiterer Instant-Messaging-Plattformen, den Vorrang eingeräumt. Mit der Ausweitung auf mehrere Kommunikationskanäle, in Gestalt von weiteren Instant-Messaging-Protokollen können weitaus mehr Menschen angesprochen werden, als das mit Skype möglich wäre. Es soll hier noch einmal in Erinnerung gerufen werden, dass hiogi eine Community für Fragesteller und -beantworter ist, die wie jede andere Community von der Größe ihrer Nutzerbasis abhängig ist. Mit ICQ/AIM/MSN/YAHOO u.a. können noch weitere Nutzer angesprochen werden. Diese Entscheidung wird auch von der aktuellen Entwicklung großer Social-Community-Portale gestützt, die sich immer mehr öffnen und ausgewählte Instant-Messaging Dienste integrieren. Diese sind deshalb maßgebend, weil sie strukturell und aufgrund ihres Erfolges Vorbildwirkung für hiogi haben.

Der Verfasser ist abschließend der Meinung, dass auf der Grundlage der in den Kapiteln 2 und 3 gewonnenen theoretischen Erkenntnisse im Kapitel 4 nachgewiesen werden konnte, wie deren praktische Umsetzung erfolgen kann.

Tabellenverzeichnis

2.1 Übersicht der strukturellen Einteilung von Dienstarten in einer SOA. 13
2.2 Nicolai Josuttis listet in (Jos07) diese Unterschiede zwischen enger und loser Kopplung auf. 22
2.3 Gegenüberstellung und Abgrenzung von MOM und ESB (Dem08). 28
2.4 Die Klassifikation der Muster. 31
2.5 Gegenüberstellung der Muster. 35

4.1 Vergleich von Instant-Messaging-Diensten. 69
4.2 Übersicht der Störungs-Level-Status. 72

Abbildungsverzeichnis

2.1 Die strukturellen Bestandteile von SOA (ohne Servicevertrag). 11
2.2 Ein einfacher Dienst, der Ereignisse in einer Datei mitloggt. . . 12
2.3 Ein Registrierungsdienst, aufgespaltet in Teildienste. 13
2.4 Kapselung der Persistenzschicht durch einen Data Service. . 14
2.5 Der Service Broker verteilt die Nachrichten. 14
2.6 Beziehung zwischen dem Nutzer und Anbieter eines Dienstes. 15
2.7 Die schematische Darstellung des *design-by-contract* Aufrufs eines Dienstes. 16
2.8 Arten und Stärke der Kopplung. 19
2.9 Ausfallkosten pro Minute aus einer Umfrage der Standish Group (vgl. (Sys00)). 20
2.10 Synchron arbeitende Services. 23
2.11 Asynchrone Kommunikation unter Einsatz von Warteschlangen für die Nachrichten. 24
2.12 Austausch von Versionsnummern bei Beginn einer Kommunikation über die Skype-API. 26
2.13 Der *Enterprise Service Bus* als Rückgrat der Kommunikation. . 27
2.14 Architektur mit verteilten Diensten und mehreren Enterprise Service Bus Instanzen. 29
2.15 Die direkte und indirekte Kommunikation. 29
2.16 Die Nachricht wird versendet - *„fire and forget“*. 32
2.17 Nur im Fehlerfall erfolgt eine Rückmeldung. 32
2.18 Jeder Anfrage folgt eine Rückmeldung – entweder eine Antwort oder ein Fehler. 32
2.19 Die Rückmeldung ist optional, jedoch kann zusätzlich ein Fehler gesendet werden. 33
2.20 Die Nachricht wird versendet - *„fire and forget“*. 33

2.21 Nur im Fehlerfall erfolgt eine Rückmeldung. 34
2.22 Jeder Anfrage folgt eine Rückmeldung. 34
2.23 Die Rückmeldung ist optional, jedoch kann zusätzlich ein Fehler gesendet werden. 34
2.24 Die enge Kopplung zweier Dienste. 36
2.25 Kommunikation auf der Grundlage von Nachrichten. 36
2.26 Für die Dauer des Aufrufs ist der Aufrufende blockiert. 37
2.27 Der Aufrufer ist nicht blockiert und kann in dieser Zeit weitere Anfragen stellen. 38
2.28 Kontinuierliches Abfragen eines Dienstes. 39
2.29 Alle Nutzer eines Dienstes werden bei einem Ereignis selbstständig informiert. 40
2.30 Variante mit enger Kopplung zum Service Bus Adapter. 42
2.31 Dekomposition des Adapters. 42

3.1 Der Datenaustausch zwischen Anwendungen mit den zwischengeschalteten Import- und Exportkomponenten (GH03). 48
3.2 Keine Konsistenz- und Synchronisationsprobleme beim konkurrierenden Zugriff auf eine gemeinsamen Datenbank (GH03). 48
3.4 RPC zur Kommunikation zwischen Anwendungen (GH03). . . 49
3.5 Integration von Anwendungen durch Messaging (GH03). . . 50
3.6 Piktogramme: Nachricht, Endpunkt und Nachrichtenkanal (GH03). 53
3.7 Aufgabe der Endpunkte im Messaging-System. 54
3.8 Zeitliche Abfolge beim Einsatz eines Ereignisgetriebenen Konsumenten. 55
3.9 Hier wird das zeitlich parallele Nachrichtenzustellen beim Einsatz Konkurrierender Konsumenten deutlich. 56
3.10 Die Anwendungen verbinden sich mit Adaptern oder direkt mit dem Nachrichtenbus. 58
3.11 Der *Kanaladapter* im Zusammenspiel mit den Anwendungsschichten. 60
3.12 Die Verwendung eines *Punkt-zu-Punkt*-Kanals zwischen zwei Endpunkten. 61
3.13 Prinzip eines *Publish-Subscribe-Kanals*. 61

3.14 Arbeitsweise eines *Publish-Subscribe-Kanals*. 62
3.15 Konzept der *Garantierten Zustellung*. 63
3.16 Nachrichtentypen und -attribute. 64

4.1 Gesamtsicht auf das System. Das Backend (links) wird in (Pra08) gesondert behandelt. Die Beispielanwendung (Mitte) ist Teil dieser Arbeit. 67
4.2 Der allgemeine Anwendungsfall beim Einsatz des Messaging-Dienstes befasst sich mit dem Versenden und Entgegennehmen von Nachrichten. 68
4.3 Die logische Sicht mit Kapselung der Anwendung Skype in einem Composite-Service, bestehend aus dem Konnektor und dem Adapter. 73
4.4 Kommunikation der Komponenten – logische und physische Sicht (Kommunikationspunkte siehe unten). 73
4.5 Praktische Ausprägung eines ereignisgetriebenen Konsumenten am Beispiel des Skype-Konnektors. 75
4.6 Ablauf der Auflösung der Abhängigkeiten beim Starten des Dienstes *Skype-Konnektor*. 76
4.7 Ablauf der Auflösung der Abhängigkeiten beim Starten des Dienstes *Skype-Konnektor* unter Einbeziehung von Service-Monitoren. 76
4.8 Selbstständige Beendigung beim Ausfall eines notwendigen Dienstes. 77
4.9 Zusammenhänge und Struktur der Dienste. Der *Servicecontroller* als Hauptklasse enthält eine Liste der Dienste und (de)aktiviert diese auf Anfragen. 78
4.10 Skype-Dienst, verteilt auf mehrere Instanzen. 80
4.11 Prinzip der *Garantierten Zustellung* auf dem Weg *zu* einem Kontakt. 81
4.12 Ausfall des JMS-Kanaladapters. 83
4.13 Ausfall des D-Bus-Adapters und des Skype-Clients. 84
4.14 Ausfall des Skype-Adapters und des -Konnektors. 84

Quellen- und Literaturverzeichnis

(CHL+06) Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 2: Adjuncts. http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060327, mar 2006. Zugriffsdatum: 2008.07.24 13:07.

(cor04) corbet. Watching filesystem events with inotify. Technical report, OSS, September 2004. Zugriffsdatum: 2008.07.08 12:05.

(CtAtPPotDoEtD99) Manage Procure Environmental Restoration Waste Management Other Construction Projects National Research Council Committee to Assess the Policies Practices of the Department of Energy to Design. *Improving Project Management in the Department of Energy*. The National Academic Press, 1999.

(Dem08) Markus Demolsky. Enterprise service bus. *entwickler.de*, April 2008. Zugriffsdatum: 2008.07.24 13:07.

(DGH03) Schahram Dustdar, Harald Gall, and Manfred Hauswirth. *Software-Architekturen für Verteilte Systeme*. Springer, Berlin, July 2003.

(Erl07) Thomas Erl. *SOA Principles of Service Design*. Prentice Hall/PearsonPTR, July 2007.

(Erl08a) Thomas Erl. Asynchronous queuing. http://www.soapatterns.org/asynchronous_queuing.asp, jul 2008. Zugriffsdatum: 2008.07.23 17:17.

(Erl08b) Thomas Erl. Legacy wrapper. http://www.soapatterns.org/legacy_wrapper.asp, jul 2008. Zugriffsdatum: 2008.07.05 22:10.

(Erl08c) Thomas Erl. Service messaging. http://www.soapatterns.org/service_messaging.asp, jul 2008. Zugriffsdatum: 2008.07.03 12:10.

(Fow03) Martin Fowler. *Patterns of Enterprise Application Architecture*. Mitp-Verlag; Auflage: 1 (Oktober 2003), 2003.

(Gar96a) Gartner. Service oriented' architectures, part 1. Research Note SPA-401-068, SSA, April 1996.

(Gar96b) Gartner. Service oriented' architectures, part 2. Research Note SPA-401-069, SSA, April 1996.

(GH03) Bobby Woolf Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Oktober 2003.

(GHJV04) Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Juli 2004.

(Heu07) Roger Heutschi. *Serviceorientierte Architektur*. Springer, Berlin, 2007.

(Inc98) Sun Inc. Interface deliverymode. http://java.sun.com/products/jms/javadoc-102a/javax/jms/DeliveryMode.html, August 1998. Zugriffsdatum: 2008.07.18 12:00.

(Jos07) Nicolai M. Josuttis. *SOA in Practice*. Dpunkt Verlag, first edition edition, August 2007.

(Koc07) Andres Koch. Service-design in der soa-praxis. http://www.ifi.uzh.ch/sise/sise2007/slides/

SISE2007_Koch.pdf, 2007. Zugriffsdatum: 2008.07.03 15:10.

(Lim) Skype Limited. Overview of the skype api. https://developer.skype.com/Docs/ApiDoc/src. Zugriffsdatum: 2008.08.08 12:00.

(Lov05) Robert Love. (patch) inotify. Technical report, Kernel, July 2005. Zugriffsdatum: 2008.07.08 12:00.

(oca) opensource community. Adapter pattern. http://en.wikipedia.org/wiki/Adapter_pattern. Zugriffsdatum: 2008.08.03 12:00.

(ocb) opensource community. Business-rule-engine. http://de.wikipedia.org/wiki/Business-Rule-Engine. Zugriffsdatum: 2008.08.04 9:00.

(occ) opensource community. Guaranteed delivery. http://activemq.apache.org/camel/guaranteed-delivery.html. Zugriffsdatum: 2008.07.18 12:00.

(ocd) opensource community. loose coupling. http://en.wikipedia.org/wiki/Loose_coupling. Zugriffsdatum: 2008.08.01 12:00.

(oce) opensource community. Middleware. http://de.wikipedia.org/wiki/Middleware. Zugriffsdatum: 2008.07.29 12:00.

(ocf) opensource community. service-oriented architecture. http://en.wikipedia.org/wiki/Service-oriented_architecture. Zugriffsdatum: 2008.07.18 12:00.

(ocg) opensource community. Soa antipatterns. http://www.ibm.com/developerworks/webservices/

library/ws-antipatterns/. Zugriffsdatum: 2008.06.08 13:00.

(Pra08) Stefan Pratsch. Ejb 3.0 service komponenten für multi-messaging konzeption und umsetzung eines esb-dienstes. Master's thesis, University of Applied Sciences Brandenburg, August 2008.

(RGO07) Arnon Rotem-Gal-Oz. *SOA Patterns*. not yet, 2007.

(Rit06) Axel Rittershaus. Kurzeinführung in soa. Technical report, Machold Systemhaus 21, Mar. 2006. Zugriffsdatum: 2008.07.24 13:07.

(Sch08) Dr. Holger Schwichtenberg. Erklärung des begriffs: Callback. http://www.it-visions.de/glossar/alle/4860/Callback.aspx, jun 2008. Zugriffsdatum: 2008.07.24 13:07.

(Sti02) Roland Stigge. Antipatterns: Stovepipe enterprise stovepipe system. http://www.rolandstigge.de/studium/stovepipe.pdf, 2002. Zugriffsdatum: 2008.07.08 12:30.

(Sys00) Cisco Systems. Eine starke basis für erfolgreiches e-business. Technical report, Cisco Systems, 2000. Zugriffsdatum: 2008.07.24 13:08.

(TTW05) Stefan Tilkov, Marcel Tilly, and Hartmut Wilms. Lose kopplung mit web-services einfach gemacht. *Java-SPEKTRUM*, May 2005. Zugriffsdatum: 2008.07.04 13:07.

(WH04) Steve Wilkes and John Harby. Soa blueprints concepts. http://www.cs.drexel.edu/~yfcai/CS575/Lectures/SOABlueprints/SOA_Blueprints_Concepts_v0_5.pdf, June 2004. Zugriffsdatum: 2008.06.08 10:30.

(wil08a) Integration. http://de.wikipedia.org/wiki/Integration_(Software), jul 2008. Zugriffsdatum: 2008.07.13 12:00.

(wil08b) Muster. http://de.wikipedia.org/wiki/Muster, jul 2008. Zugriffsdatum: 2008.07.12 12:00.

(WWWK94) Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*. Sun Microsystems, Inc., 1994.

Erklärung zur Diplomarbeit

Hiermit erkläre ich, die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst zu haben. Auf die verwendeten Quellen habe ich in entsprechender Weise hingewiesen und diese im Literaturverzeichnis aufgeführt.

Brandenburg a. d. H., den 20. August 2008

Lars Gohlke