Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Modellierung nebenläufiger, komponentenbasierter Software-Systeme mit Entwurfsmustern

Diplomarbeit am
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Software-Entwurf und -Qualität
Prof. Dr. Ralf H. Reussner
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. inform.
Holger Friedrich

Betreuer:
Prof. Dr. Ralf H. Reussner
Dipl.-Inform. Jens Happe

Tag der Anmeldung: 15. März 2007
Tag der Abgabe: 15. September 2007

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 15. September 2007

Inhaltsverzeichnis

1. Einleitung

Die Entwicklung komplexer Software wird mithilfe von Entwicklungsprozessen von der Konzeption bis hin zum Einsatz strukturiert. Zu Beginn erfolgt die Analyse der funktionalen und extra-funktionalen Anforderungen, wie z. B. Performance- oder Verfügbarkeitsanforderungen. Beim Software-Entwurf wird die Grundlage dafür gelegt, dass das spätere Endprodukt sowohl die funktionalen als auch die extra-funktionale Anforderungen erfüllt. Oftmals werden jedoch die extra-funktionalen Anforderungen erst in späteren Prozessphasen wie z. B. beim Testen beachtet oder deren Nichterfüllung während des laufenden Betriebs festgestellt. Um in diesen Fällen dennoch die gewünschte Qualität zu erreichen, ist es meist erforderlich, dass frühe Entwurfsentscheidungen geändert werden müssen. Die damit verbundenen Kosten steigen, je später die Änderungen durchgeführt werden, da sie im Allgemeinen mit größeren Änderungen am Quellcode verbunden sind. Die Methode der modellbasierten Performance-Vorhersage verspricht hier Abhilfe zu leisten, da sie die Bewertung des Performance-Einflusses von Entwurfsentscheidungen ermöglicht. Dabei wird ein Software-Modell um Performance-relevante Informationen ergänzt, die auf Schätzungen oder Messungen basieren. Mithilfe von Analyseverfahren oder Simulationen können auf Basis dieser Informationen Vorhersagen über die zu erwartende Performance des Modells getroffen werden. Somit können verschiedene Entwurfsalternativen miteinander vergleichen werden.

Eine Möglichkeit zur Verbesserung der Performance ist der Einsatz von Nebenläufigkeit. Nebenläufige Software gewinnt mit der zunehmenden Verbreitung von Mehrkernprozessoren immer mehr an Bedeutung. Diese Art von Software besteht aus mehreren, parallel zueinander ausführbaren Programmteilen, so genannten Prozessen oder Threads. Dadurch ist sie in der Lage, die vorhandene Parallelität auf Hardware-Ebene auszunutzen, was bei rein sequenzieller Programmausführung nicht möglich ist. Nebenläufige Software weist im Allgemeinen eine sehr hohe Komplexität auf. Dies liegt an den verwendeten Programmierkonstrukten, wie Threads oder spezielle Synchronisationsmechanismen, da beispielsweise die Ablaufreihenfolge von Threads erst zur Ausführungszeit vom Scheduler des Betriebssystems festgelegt wird. Wegen der hohen Komplexität von nebenläufiger Software gestalten sich auch Performance-Vorhersagen als schwierig. Der Einsatz von Entwurfsmustern stellt eine Möglichkeit

dar, der hohen Komplexität entgegenzuwirken, da mit ihnen eine Abstraktion von Nebenläufigkeitsaspekten erreicht wird. Für die Entwurfsmuster können Modellkonstrukte geschaffen werden, die deren Verhalten und deren Performance-Einflüsse kapseln. Diese Modellkonstrukte versprechen eine Vereinfachung der Modellierung von nebenläufiger Software, da durch die Kapselung der Entwurfsmuster deren Komplexität für den Modellierer transparent ist. Mithilfe der Modellierungskonstrukte kann darüber hinaus eine Vereinfachung der Performance-Vorhersage erreicht werden, da über die Verwendung der Modellkonstrukte auch die darin gekapselten Performance-Einflüsse in den Software-Entwurf einfließen.

1.1 Beitrag der Arbeit

In dieser Diplomarbeit werden Entwurfsmuster für nebenläufige Systeme entsprechend ihrer Funktionalität und ihrem Verwendungszweck für komponentenbasierte Software-Systeme klassifiziert. Damit soll die Auswahl der Entwurfsmuster beim Software-Entwurf vereinfacht werden.

Die Erstellung von Modellkonstrukten erfolgt exemplarisch für jene Entwurfsmuster, die sich auf Problemstellungen in Zusammenhang mit nachrichtenorientierter Kommunikation beziehen. Das Ziel ist, dass die Modellkonstrukte das Verhalten der Entwurfsmuster möglichst gut abbilden und deren Performance-Relevanz berücksichtigen. Dafür ist eine umfassende Evaluierung der Performance-relevanten Parameter erforderlich.

Für die Anreicherung eines Software-Entwurfs um Performance-relevante Parameter für nachrichtenorientierte Kommunikation wird ein Annotationsmodell entwickelt, über das sich die Performance-relevanten Parameter spezifizieren lassen. Mithilfe einer entwickelten Transformation werden diese Parameter ausgewertet und der Software-Entwurf um die benötigten Modellkonstrukte erweitert. Damit können Performance-Vorhersagen für Software-Systeme, die die nachrichtenorientierte Kommunikationsform verwenden, durchgeführt werden. Die Vorhersage-Qualität wird in einer abschließenden Fallstudie bewertet.

1.2 Gliederung der Arbeit

In Kapitel 2 wird der entwickelte Ansatz zur Performance-Vorhersage von nebenläufigen Systemen auf Basis von Entwurfsmustern in Bezug zu verwandten Arbeiten gesetzt.

Die Grundlagen dieser Arbeit werden in Kapitel 3 ausführlich erklärt. Unter anderem umfasst dieses Kapitel eine Einführung in das Palladio Komponentenmodell, das Performance-Vorhersagen für Software-Architekturen ermöglicht. Dieses Komponentenmodell wird durch die erstellten Modellkonstrukte erweitert, um dessen Unterstützung für Nebenläufigkeit zu verbessern.

In Kapitel 4 werden Entwurfsmuster für nebenläufige Systeme beschrieben und entsprechend ihrer Anwendbarkeit bei der Entwicklung von komponentenbasierter Software kategorisiert.

Die Integration von Entwurfsmustern für nachrichtenorientierte Kommunikation in das Palladio Komponentenmodell wird in Kapitel 5 detailliert beschrieben. Dabei wird besonders auf die Evaluierung der Performance-relevanten Parameter und die Erstellung der Modellkonstrukte eingegangen. Außerdem wird ein Annotationsmodell vorgestellt, mit dem ein Software-Entwurf um Performance-relevante Informationen für nachrichtenorientierte Kommunikation ergänzt werden kann. Darüber hinaus werden die Schritte einer Transformation erläutert, die einen so angereicherten Entwurf um die entsprechenden Modellkonstrukte modifiziert.

Die Evaluierung des Ansatzes wird in Kapitel 6 anhand einer Fallstudie durchgeführt. Dabei steht die Bewertung der Vorhersage-Qualität der erstellten Modellkonstrukte im Mittelpunkt. Als Vergleichsmaßstab für die Simulationsergebnisse dienen Messwerte, die für die implementierte Anwendung der Fallstudie ermittelt wurden.

Die Annahmen und Einschränkungen, die in dieser Arbeit getroffen bzw. vorgenommen wurden, sind in Kapitel 7 zusammengefasst.

Abschließend erfolgt eine Zusammenfassung der durchgeführten Erstellung von Modellkonstrukten und der Integration in das Palladio-Komponentenmodell, um dessen Unterstützung für nebenläufige Systeme zu verbessern. Außerdem wird ein Ausblick auf weitere Ansatzpunkte gegeben.

2. Verwandte Arbeiten

In der Literatur gibt es zahlreiche Arbeiten, die nebenläufige Software-Systeme analysieren und Möglichkeiten aufzeigen, wie der Problematik von Nebenläufigkeit beim Software-Entwurf entgegnet werden kann.
Lee [Lee06] sieht im Thread-Modell die Quelle für die schwere Verständlichkeit von nebenläufiger Software. Alternativen zum Thread-Modell finden kaum Verbreitung, da mit ihnen im Allgemeinen syntaktische Änderungen verbunden sind, die von Programmierern kaum angenommen werden. Lee schlägt deshalb vor, etablierte Programmiersprachen nicht zu ersetzen, sondern auf ihnen aufzubauen. Dazu sollen Koordinierungssprachen eingesetzt werden, mit denen ausschließlich die Koordination und Kommunikation zwischen Komponenten spezifiziert wird. Die neue Syntax der Koordinierungssprachen ist orthogonal zu jener der etablierten Programmiersprachen. Somit können diese weiterhin für alle anderen Zwecke außer der Komponentenkoordination und -kommunikation verwendet werden. Einige Entwurfsmuster können auf der Ebene von Koordinierungssprachen angesiedelt sein und stellen somit eine Grundlage für die mit der Verwendung von Koordinierungssprachen versprochene Verbesserung der Verständlichkeit von nebenläufiger Software dar. Diese Arbeit kann als Bestandsaufnahme von nebenläufiger Software betrachtet werden, die zugleich mögliche Lösungsansätze aufzeigt. Hierbei stehen hauptsächlich qualitative Eigenschaften wie Verklemmungsfreiheit im Blickpunkt. Quantitative Eigenschaften, wie z.B. Performance und Zuverlässigkeit werden nicht betrachtet.
Im Rahmen des Ptolemy-Projekts [Lee03] wird die Modellierung, der Entwurf und die Simulation von nebenläufigen, echtzeitfähigen eingebetteten Systemen untersucht. Dabei steht das Zusammenspiel von nebenläufigen Komponenten im Mittelpunkt. Berechnungsmodelle wie *Communicating Sequential Processes* [Hoar83] oder *Process Networks* [Kahn74] werden zur Modellierung der Interaktion von Komponenten verwendet und können als Entwurfsmuster für die Komponenteninteraktion betrachtet werden. In Ptolemy sind verschiedene Berechnungsmodelle als so genannte Domänen implementiert, die grafische Darstellungen zur Spezifizierung von Modellen verwenden. Dadurch soll die Komplexität der heterogenen Modellierung mit verschiedenen Berechnungsmodellen verringert werden. Im Ptolemy-Projekt werden ähnliche Ansätze verfolgt wie in dieser Diplomarbeit, da jeweils Entwurfsmuster für die Modellierung von Nebenläufigkeit eingesetzt werden und Modellkonstrukte zur

Verringerung der Komplexität beitragen sollen. Jedoch werden Aspekte wie Performance und Zuverlässigkeit nicht thematisiert.
Hilderink stellt in [Hild02] eine grafische Modellierungssprache zur Spezifikation von Nebenläufigkeit beim Software-Entwurf vor. Als Basis dient die Prozessalgebra *Communicating Sequential Processes* (CSP) von Hoare [Hoar83]. Diese betrachtet Nebenläufigkeit auf einer hohen Abstraktionsebene und bietet fundamentale Primitive für die Spezifizierung, den Entwurf und die Analyse von nebenläufiger Software. Mithilfe der grafischen Modellierungssprache sollen diese beim Entwurf von nebenläufigen Systemen verwendet werden können. Entwürfe, die mit der Modellierungssprache erstellt werden, bilden die so genannten CSP-Diagramme. Ein CSP-Diagramm ist ein Graph, der aus Prozessen und ihren Beziehungen besteht und sowohl Datenfluss-, als auch Kontrollflussaspekte widerspiegelt. Somit kann mit einem CSP-Diagramm das Ausführungsmodell eines Prozessgeflechts dargestellt werden. Für die Spezifikation der Beziehungen zwischen Prozessen werden grafische Konstrukte für die Modellierung in CSP-Diagrammen zur Verfügung gestellt. Diese Konstrukte repräsentieren die unterschiedlichen Beschreibungsprimitive von CSP. Für die Spezifikation der Prozesse selbst sind CSP-Diagrammen nicht geeignet, da von prozessinternen Details abstrahiert wird. Zudem stellen Prozesse ausschließlich Einheiten dar, die eine Folge von Ereignissen ausführen, bei denen mindestens zwei Einheiten beteiligt sind. CSP-Diagramme sind mit UML kombinierbar und können als Prozessdiagramme für UML verwendet werden. Mittels Bibliotheken kann CSP in verbreitete Programmiersprachen integriert werden. Dieser Ansatz geht in Richtung der von Lee vorgeschlagenen Koordinierungssprache und teilt mit diesem den Nachteil, dass die Performance-Eigenschaften weniger essentiell sind als Konflikterkennung, mögliche Verklemmungen oder Prioritätsumkehrungen.
Kumar et al. sehen in [KWSS+04], dass Spezifikations- und Verifikationstechniken für sequenzielle Systeme mit jenen für nebenläufige Systeme in Einklang gebracht werden müssen, um eine erfolgreiche Spezifikation und Verifikation von komponentenbasierter Software zu erreichen. Die Kapselung von Nebenläufigkeit in Komponenten stellt einen Ansatz dazu dar. Für die Modellierung von möglichen Beeinflussungen von nebenläufigen Komponenten werden relationale Spezifikationen eingesetzt. Die in den Spezifikationen modellierte Nebenläufigkeit wird für aufrufende Komponenten verdeckt, indem Zwischenkomponenten eingeführt werden. Diese dienen als Stellvertreter für die nebenläufigen Komponenten und stellen eine vereinfachte sequenzielle Komponentenspezifikation bereit. Dafür ist es erforderlich, dass ein zusätzlicher Spezifikationsausdruck eingeführt wird, der die Anforderungen an das zukünftige Verhalten einer aufrufenden Komponente ausdrückt. Mithilfe der relationalen Spezifikationen, der Zwischenkomponenten und dem zusätzlichen Spezifikationsausdruck kann die Nebenläufigkeit innerhalb von Komponenten gekapselt werden, sodass diese nach außen als sequenzielle Komponenten erscheinen. Damit erhält jeder Aufrufer einer Komponente eine vereinfachte Sicht auf das System, bei der die Nebenläufigkeit für den Aufrufer transparent ist. Dieser Ansatz ermöglicht die Spezifizierung von komponentenbasierter Software unter Gewährleistung von Thread-Sicherheit und Vermeidung von verhungernden Threads. Allerdings entsteht durch die zusätzlichen Komponenten und umfangreicheren Komponentenspezifikationen ein deutlicher Mehraufwand. Damit verbunden ist eine Erhöhung der Komplexität des Entwurfs. Hinzu kommt, dass die Nebenläufigkeit auf einer niedrigen Abstraktionsebene in den Komponentenspezifikationen erfasst wird. Eine Betrachtung der Auswirkungen der Spezifikation von Nebenläufigkeit erfolgt nicht. Für Performan-

ce-Probleme in Zusammenhang mit Nebenläufigkeit bieten Entwurfsmuster oft Lösungsschemata (siehe Kapitel 4). Solche Entwurfsmuster müssten detailliert gemäß diesem Ansatz spezifiziert werden, wodurch dessen praktische Anwendbarkeit stark eingeschränkt wird.

Pettit und Gomaa modellieren in [PeGo04] und [PeGo06] Verhaltensmuster von nebenläufiger Software mit UML-Stereotypen, für welche Vorlagen für farbige Petri-Netze (*Coloured Petri Nets*, CPNs) [JeKW07] zur Verfügung gestellt werden, sodass UML-Kollaborationsdiagramme in CPNs transformiert werden können. CPNs ermöglichen neben der Modellierung von nebenläufigen Systemen auch deren Analyse bezüglich funktionalen Aspekten und Performance. Dazu nutzen Gomaa und Pettit das Werkzeug DesignCPN [Desi], sodass die Analysefähigkeit dieser Arbeiten stark von diesem Werkzeug abhängen. Wie diese Diplomarbeit verfolgen beide Arbeiten die Idee, dass Muster für die Modellierung und die Performance-Vorhersage von nebenläufigen Systemen verwendet werden und dass durch die Abbildung auf Abstraktionen wie CPN-Vorlagen eine Vereinfachung der Modellierung erreicht werden soll. Die beiden Arbeiten setzen jedoch auf eine Kombination von UML und CPNs, wohingegen diese Diplomarbeit das Palladio Komponentenmodell erweitert. Dies ist von Vorteil, denn das Palladio Komponentenmodell vereinigt Konzepte der komponentenbasierten Software-Entwicklung und der Performance-Vorhersage. Dabei kommen modellgetriebene Techniken zum Einsatz, um Modellinstanzen in Vorhersagemodelle zu transformieren. Ansätze, die auf UML basieren, haben hier deutliche Defizite, da bezüglich modellgetriebener Entwicklung bei UML noch deutlicher Verbesserungsbedarf besteht (siehe [Hend05]) und zu sehr komplexen Transformationen führt. Für Transformationen ist beispielsweise problematisch, dass UML mehrere Konstrukte bietet, mit denen der selbe Sachverhalt auf unterschiedlicher Weise modelliert werden kann.

Zdun und Avgeriou befassen sich in [ZdAv05] mit der Modellierung von Entwurfsmustern. Sie stellen fest, dass die Modellierung von Entwurfsmustern schlecht unterstützt wird, da sich die Musterelemente im Allgemeinen nicht direkt auf Elemente von Modellierungssprachen abbilden lassen. Sie gehen dieses Problem an, indem sie fundamentale Modellierungselemente von Entwurfsmustern identifizieren und für diese Elemente UML-Erweiterungen erstellen. Diese Erweiterungen lassen sich dann beim Software-Entwurf nutzen und ermöglichen dadurch eine einfachere Verwendung von Entwurfsmustern. Bei diesem Ansatz steht grundsätzlich die Modellierung von Entwurfsmustern beim Software-Entwurf im Mittelpunkt, so dass die Modellierungselemente möglichst allgemein sind. Dieser Ansatz lässt sich jedoch auf den Entwurf von nebenläufiger Software übertragen, so dass Modellierungskonstrukte für Entwurfsmuster Erleichterungen bei der Modellierung versprechen.

Eine Reihe von Arbeiten befassen sich mit Performance-Aspekten beim Entwurf von Software. Methoden zur modellbasierte Performance-Vorhersage wie das *Software Performance Engineering* (SPE) [Smit90] oder die Analyse bzw. Simulation von UML-Modellen, die gemäß dem SPT-Profil [Grou05] der Object Management Group annotiert sind, ermöglichen die Spezifikation und Analyse von dienstgüterelevanten Parametern einer Software-Architektur. Diese Methoden betrachten jedoch Aspekte wie Nebenläufigkeit, Kommunikation oder Synchronisation auf einer sehr niedrigen Abstraktionsebene. Einen Schritt in diese Richtung gehen Smith und Wil-

liams in [SmWi02], indem sie den SPE-Ansatz auf verteilte Systeme anwenden. Zur Modellierung von verteilten Systemen werden vier verschieden Kommunikations- und Synchronisationstypen für UML-Sequenzdiagramme zur Verfügung gestellt. Damit ist es möglich, die Interaktion von Komponenten zu modellieren und deren Auswirkungen auf die Performance vorherzusagen, jedoch werden die Probleme von Nebenläufigkeit, wie z.B. Verklemmungen oder Laufzeiteffekte nicht thematisiert. Darüber hinaus werden dem Software-Architekten nur geringfügige Abstraktionen für die Nebenläufigkeit geboten, sodass die Komplexität beim Entwurf von nebenläufiger Software kaum reduziert wird.
Mit der Performance-Vorhersage bei Entwurf von komponentenbasierten, Server-seitigen Anwendungen beschäftigen sich Liu et al. in [LiFG05]. Sie untersuchen dabei spezielle Architekturmuster für Anwendungs-Server. Ihr Ansatz besteht aus vier Teilschritten. Zunächst wird ein allgemeines Modell für den Komponentencontainer von Anwendungs-Servern entworfen, das dessen Hauptbestandteile und Verzögerungsquellen repräsentiert. Im zweiten Schritt erfolgt die Analyse und Modellierung von Architekturmustern in Form von Aktivitätsdiagrammen, um daraus deren spezifischen Ressourcenbedarf zu ermitteln. Der dritte Schritt sieht die Erstellung eines parametrisierten Performance-Modells für einen bereits erstellten Anwendungsentwurf vor, bei dem erfasst wird, welchen Bedarf die Komponenten des Anwendungsentwurfs an den Komponentencontainer stellen. Für die Performance-Vorhersage dieses Modells müssen die Charakteristiken der verwendeten Plattform einbezogen werden. Dafür wird im letzten Schritt mithilfe einer einfachen Testanwendung ein Profil der Plattform erstellt. Dieser Ansatz zielt hauptsächlich auf die Erfassung der Performance-relevanten Aspekte von Komponentencontainern, mit denen ein bestehender Anwendungsentwurf angereichert wird. Diese Anreicherung zu einem parametrisierten Performance-Modell muss jedoch manuell erfolgen, wodurch der Entwurfsaufwand deutlich erhöht wird. Damit verbunden ist eine Steigerung der Komplexität des Entwurfs, da die Ermittlung des Ressourcenbedarfs der Anwendungskomponenten an dem Komponentencontainer auf einem niedrigen Abstraktionsniveau erfolgt. Somit ist dieser Ansatz für Performance-Vorhersagen im praktischen Einsatz weniger geeignet. In dieser Arbeit erfolgt die Spezifizierung von extra-funktionalen Performance-relevanten Parametern über ein Annotationsmodell, anhand derer der Anwendungsentwurf mittels Transformationen mit Modellkonstrukten für die Funktionalität von Nachrichtensystemen erweitert wird. Dies ist mit einem deutlich geringerem Aufwand und einer geringeren Komplexität beim Software-Entwurf verbunden. Dennoch bietet der Ansatz von Liu et al. Erkenntnisse für die Modellierung von Ressourcen, beispielsweise um ein exakteres Modell für Nachrichtensysteme zu erstellen, als dies exemplarisch in dieser Arbeit erfolgt.

Chen und Greenfield führen in [ChGr04] empirische Untersuchungen für die Ermittlung von Dienstgütemerkmalen des *Java Message Service* (JMS) [Java02] durch. Dazu entwickeln sie eine Testanwendung, die die drei JMS-Optionen „persistente Nachrichtenkanäle“, „dauerhafte Empfängerregistrierung“ und „transaktionaler Nachrichtenversand“ unterstützt und zur Evaluation die Metriken „maximaler nachhaltiger Nachrichtendurchsatz“, „Latenz“, „Zeitspanne für den Versand einer Menge von Nachrichten“ und „Verlust von persistierten Nachrichten nach einem Neustart“ verwendet. Mithilfe dieser Testanwendung können verschiedene JMS-Implementierungen hinsichtlich Dienstgüte miteinander verglichen werden. Für die im Rahmen dieser Arbeit durchgeführte Evaluierung der Performance-relevanten Parameter ist

diese Testanwendung nur unzureichend geeignet, da nicht alle evaluierten JMS-Optionen unterstützt werden.
In dieselbe Richtung wie die Arbeit von Chen und Greenfield zielt die Entwicklung des standardisierten Benchmarks *SPECjms2007* zur Evaluierung der Performance und Skalierbarkeit von JMS-Nachrichtensystemen. Ein bedeutender Unterschied zur vorherigen Methode ist, dass für den Benchmark repräsentative Lastszenarien entworfen wurden, die in [SKCB07] und [SKBB07] vorgestellt werden. Für die Lastszenarien wird zudem ein breiteres Spektrum an Dimensionen betrachtet, als die drei zuvor genannten JMS-Optionen. Jedoch werden nicht alle der JMS-Optionen abgedeckt, die in dieser Arbeit den Entwurfsmustern für nachrichtenorientierte Kommunikation zugeordnet werden. Weiterhin ist beiden Ansätzen gemein, dass die jeweils zur Evaluierung eingesetzte Anwendung nicht innerhalb eines Java EE-basierten Anwendungs-Servers läuft, so dass von den Evaluationsergebnissen nicht ohne weiteres auf die Performance von nachrichtenorientierter Kommunikation zwischen Java EE-Komponenten geschlossen werden könnte. Aus diesem Grund erfolgt die Evaluierung der Performance-relevanten Parameter mithilfe einer eigens entwickelten Testanwendung.

3. Grundlagen

In diesem Kapitel werden die Grundlagen dieser Diplomarbeit beschrieben, um den entwickelten Ansatz besser verstehen zu können. Die einzelnen Themenbereiche werden getrennt in eigenen Unterkapiteln vorgestellt. Im Mittelpunkt stehen dabei Begriffserklärungen und die Vorstellung ihrer Konzepte.

Das erste Unterkapitel gibt einen Überblick über die Entwicklung von nebenläufigen Systemen auf Hardware-Ebene und deren Bedeutung für die Software-Entwicklung. Dabei werden bedeutende Probleme bei der Entwicklung von Anwendungen für nebenläufige Systeme identifiziert. Diese Probleme motivieren diese Arbeit, denn der entwickelte Ansatz soll dazu beitragen, diesen Problemen zu entgegnen. Im zweiten Unterkapitel wird der Einsatz von Entwurfsmustern für den Entwurf von nebenläufiger Software motiviert, da die Verwendung von Entwurfsmustern eine Möglichkeit darstellt, den Problemen von nebenläufiger Software entgegenzuwirken. Anschließend werden die wichtigsten Konzepte des Palladio Komponentenmodells erläutert, das in dieser Diplomarbeit so erweitert wird, dass es eine bessere Unterstützung von Nebenläufigkeit bietet. Anschließend erfolgt eine Vorstellung der *Java Platform, Enterprise Edition*, denn diese wird im Rahmen dieser Arbeit als Referenzplattform für die Entwicklung von komponentenbasierten Anwendungen betrachtet. Im vorletzten Unterkapitel wird das Konzept der nachrichtenorientierten Kommunikation vorgestellt und die wichtigsten Bestandteile der Schnittstellenspezifikation JMS für den Zugriff von Java-Programmen auf Nachrichtensysteme beschrieben. Diese Kommunikationsform steht im besonderen Fokus dieser Arbeit, da der entwickelte Ansatz auf die asynchrone Kommunikation mittels Nachrichtenaustausch ausgerichtet ist. Den Abschluss bildet das Unterkapitel zu evolutionären Algorithmen. Dabei wird deren Evolutionsprozess erklärt und die Besonderheit der genetischen Programmierung als spezieller Vertreter dieser Algorithmenfamilie aufgezeigt. Die genetische Programmierung wird in dieser Arbeit verwendet, um aus den Ergebnisse der Leistungsbewertung von speziellen Performance-relevanten Parametern ein Modell ableiten zu können.

3.1 Nebenläufige Systeme

In den letzten Jahrzehnten hat sich die Leistung der Mikroprozessoren exponentiell gesteigert. Dies wurde vor allem durch die Erhöhung der Taktfrequenz, die Optimierung der Befehlsverarbeitung und dem Einsatz von größeren Caches erreicht. Verbesserungen in diesen Bereichen beschleunigen sowohl sequenzielle als auch nebenläufige Anwendungen. Seit ungefähr drei Jahren sind die Leistungssteigerung der Mikroprozessoren basierend auf den genannten Bereichen an physikalische Grenzen gestoßen, da beispielsweise mit der exponentiellen Leistungssteigerung auch eine exponentielle Steigerung der Leistungsaufnahme einherging. Die exponentiellen Leistungssteigerungen der vergangenen Jahrzehnte können mit bisherigen Mikroprozessorarchitekturen nur noch schwer erreicht werden [OlHa05]. Seit einiger Zeit vollzieht sich daher ein Wandel bei Architekturen von Mikroprozessoren. Die modernen Architekturen basieren anstatt auf einem auf mehreren Rechenkernen. Mehrkernprozessoren bieten eine höhere Parallelität auf Hardware-Ebene und versprechen, die in Software vorhandene Parallelität effizienter auszunutzen als Prozessoren mit nur einem Kern. Auf diese Weise sollen weiterhin Leistungssteigerungen bzgl. der Verarbeitungsgeschwindigkeit von Mikroprozessoren in den bisherigen Größenordnungen möglich sein [Cree05].
In den letzten Jahren finden Mehrkernprozessoren im Server- und Desktop-Bereich eine immer größere Verbreitung. Von den Leistungssteigerungen dieser Prozessoren können nur nebenläufige Anwendungen profitieren. Sequenzielle Anwendungen werden allenfalls geringe Beschleunigung aufgrund von größeren Caches erfahren. Nebenläufige Anwendungen werden schon seit vielen Jahren geschrieben, die überwiegende Mehrheit der existierenden Anwendungen ist aber sequenziell. Die Entwicklung von nebenläufiger Software gewinnt somit an Bedeutung, da die einzelnen Threads und Prozesse auf die verfügbaren Prozessorkerne verteilt werden können und somit eine effiziente Ausnutzung der Hardware ermöglicht wird [Cree05, SuLa05].
Klassische Programmierkonstrukte zur Unterstützung von Nebenläufigkeit, wie Semaphoren, Sperren oder Threads, die in den meisten objektorientierten Sprachen eingesetzt werden, bereiten Probleme bei der Programmierung [Sutt05, Lee06]. Der Grund dafür ist, dass sie eine hohe Komplexität aufweisen, zu Fehlern verleiten und Programme schwerer verständlich machen. Im Bereich des High Performance Computings werden zur Parallelisierung Implementierungen von Programmiermodellen wie Message Passing oder Datenparallelität (z. B. MPI[1] oder OpenMP[2]) eingesetzt. Diese Programmiermodelle stehen in den objektorientierten Sprachen nicht zur Verfügung.
Sutter und Larus [SuLa05] sehen den Bedarf an Objektorientierung für Nebenläufigkeit und prophezeien eine neue Ära in der Software. Objektorientierung für Nebenläufigkeit stellt dabei eine Abstraktion auf hoher Ebene dar, die dabei hilft, nebenläufige Programme zu schreiben. Von den bisherigen nebenläufigen Programmiermodellen stellen sich Threads und Sperren als problematisch dar [Lee06, SuLa05], da z. B. Deadlocks, Starvation oder Race Conditions [Tane02, S. 50, S. 143, S. 118] auftreten können. Threads sind nichtdeterministisch, da ihre Ausführungsreihenfolge erst zur Laufzeit vom Scheduler des Betriebssystems bestimmt wird. Die Aufgabe von Entwicklern ist es, alle theoretischen Ausführungsreihenfolgen zu beachten und mittels Semaphoren, Monitoren oder anderen Programmierkonstrukten die Ausfüh-

[1] http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
[2] http.//www.openmp.org

rung an bestimmten Stellen zu serialisieren, um eine deterministische Reihenfolge zu erreichen.

Nebenläufige Software nutzt nicht grundsätzlich die vorhandene Parallelität auf Hardware-Ebene aus, da in Abhängigkeit von der Ausführungsreihenfolge mit der Synchronisation von nebenläufigen Komponenten ein Blockieren einiger dieser Komponenten verbunden sein kann. Die Effizienz von nebenläufiger Software kann demnach über die Anzahl der rechnenden und rechenbereiten nebenläufigen Komponenten bestimmt werden. Ist beispielsweise die Anzahl der rechnenden oder rechenbereiten Komponenten einer Anwendung geringer als die Zahl der Prozessorkerne, so kann die Anwendung nur einen Teil der Prozessorkerne verwenden. Der Synchronisationsaufwand in nebenläufiger Software ist abhängig von der Art der Parallelität. Dabei kann nach [SuLa05] zwischen unabhängiger, regulärer und unstrukturierter Parallelität unterschieden werden. Die Art ist abhängig vom Grad der Interaktion der nebenläufigen Komponenten. Bei unabhängiger Parallelität sind die Komponenten komplett unabhängig und können die Parallelität auf Hardware-Ebene effizient ausnutzen, da keinerlei Synchronisation erforderlich ist. Bei regulärer Parallelität werden die selben Operationen auf unterschiedlichen Daten ausgeführt, wobei eine Operation auf einem Teil der Daten von den Ergebnisse der Operationen auf anderen Teilen der Daten abhängt. Aus diesem Grund erfordern Anwendungen mit regulärer Parallelität entweder Synchronisationsmechanismen oder spezielle Algorithmen, um eine Ausführungsreihenfolge zu gewährleisten, die keine Synchronisation erfordert. Unstrukturierte Parallelität ist die allgemeinste Art, bei der sich die nebenläufigen Operationen unterscheiden, so dass ihre Datenzugriffe nicht vorhersagbar sind und deshalb explizite Synchronisationsmechanismen eingesetzt werden müssen, um ein deterministisches Verhalten zu erreichen. Anwendungen, die Threads verwenden, weisen meist eine unstrukturierte Parallelität auf.

3.2 Entwurfsmuster für nebenläufige Systeme

Ein Entwurfsmuster beschreibt ein wiederkehrendes Problem und zeigt dazu ein generisches Lösungsschema [AlIS77]. Übertragen auf Software beschreibt ein Entwurfsmuster eine Familie von Lösungen für ein Software-Entwurfsproblem. Entwurfsmuster helfen Entwicklern, bei bestehender Software die Robustheit zu erhöhen und stellen eine Hilfe beim Software-Entwurf dar, damit neue Software effektiv entwickelt werden kann. Bei nebenläufiger und verteilter Software richten Entwurfsmuster den Fokus der Entwickler auf Entwurfsangelegenheiten und höhere Ebenen der Software-Architektur, wie beispielsweise der Spezifikation von geeigneten Dienstzugriffen, der Ereignisbehandlung oder Thread-Modellen [SRSS+00].
Mit der Nebenläufigkeit und Verteilung von Software steigt deren Komplexität weiter an. Um die Komplexität zu reduzieren und dadurch die Programme verständlicher machen zu können, wurden spezielle Entwurfsmuster für diese Aspekte identifiziert. Schmidt et al. [SRSS+00] präsentieren Entwurfsmuster für die Aspekte „Dienstkonfiguration und Dienstzugriff“, „Ereignisbehandlung“, „Nebenläufigkeit“ sowie „Synchronisation“, die sie als zentrale Herausforderungen für den Entwurf und die Programmierung von nebenläufiger und verteilter Software ansehen. In Kapitel 4 werden diese Entwurfsmuster detailliert vorgestellt.

3.3 Das Palladio Komponentenmodell

Das Palladio Komponentenmodell (PCM) ist ein Meta-Modell zur Spezifizierung von komponentenbasierten Software-Architekturen und deren extra-funktionalen Eigenschaften. Es verfolgt die Konzepte der komponentenbasierten Software-Entwicklung (CBSE) [Szyp02], ist jedoch speziell für die Vorhersage von Dienstgüte konzipiert, wie z. B. der Performance oder der Zuverlässigkeit. Damit können Software-Architekturen bereits in der Entwurfsphase des Software-Entwicklungsprozesses anhand solcher Dienstgüteparameter bewertet und folglich Entwurfsentscheidungen in einer Weise getroffen werden, um eine gewünschte Dienstgüte möglichst zu erreichen. Dadurch soll vermieden werden, dass in späten Phasen des Software-Entwicklungsprozesses Entwurfsentscheidungen kostspielig geändert werden müssen, um die gewünschte Qualität erzielen zu können.
Eine detaillierte Beschreibung des Palladio Komponentenmodell ist im technischen Bericht „The Palladio Component Model“ [RBKH+07] zu finden, in dieser Arbeit wird nur ein Überblick über die wichtigsten Konzepte gegeben:

Komponenten

Komponenten bilden den Kern des Komponentenmodells. Sie kapseln funktionale und extra-funktionalen Eigenschaften und sollen ohne Kenntnis ihres inneren Aufbaus verwendbar sein. Deshalb ist die Spezifikation von Schnittstellen erforderlich, die die angebotenen und benötigten Funktionen enthalten. Die Verbindung zwischen Schnittstellen und Komponenten erfolgt über Rollen. Bietet eine Komponente eine Schnittstelle an, so wird die Assoziation zwischen der Komponente und der Schnittstelle als *Provided Role* bezeichnet. Benötigt eine Komponente eine Schnittstelle, werden die Komponente und die Schnittstellen über eine *Required Role* assoziiert. Jede Komponente muss alle Dienste der angebotenen Schnittstellen bereitstellen. Dazu nimmt sie wiederum jene Dienste in Anspruch, die über ihre benötigten Schnittstellen spezifiziert werden.
Im PCM existieren für Komponenten die drei unterschiedlichen Typen *Provided Component Type*, *Complete Component Type* und *Implementation Component Type*. Bei Komponenten vom Typ *Provided Component Type* werden lediglich die angebotenen Schnittstellen definiert, über benötigte Schnittstellen und Implementierungsdetails werden keine Aussagen getroffen. Komponenten vom Typ *Complete Component Type* abstrahieren von Implementierungsdetails, bieten jedoch eine vollständige Spezifikation der angebotenen und benötigten Schnittstellen. Komponenten sind vom Typ *Implementation Component Type*, wenn sie vollständig spezifiziert sind. Das bedeutet, dass sowohl ihre angebotenen und benötigten Schnittstellen spezifiziert sind als auch ihre interne Struktur. Solche vollständig spezifizierten Komponenten können entweder Basiskomponenten oder zusammengesetzte Komponenten sein. In einer zusammengesetzten Komponente sind mehrere Komponenten zusammengefasst und ihr Verhalten setzt sich aus dem Verhalten der inneren Komponenten zusammen. Im Gegensatz dazu wird das Verhalten einer Basiskomponente nur vom Verhalten der Komponente selbst bestimmt. Die Charakterisierung des Verhaltens einer Basiskomponenten erfolgt über eine *Service Effect Specification*.

Schnittstellen

Im PCM bestehen Schnittstellen vor allem aus einer Menge von Signaturspezifikationen und optionalen Protokollspezifikationen. Mithilfe der Signaturspezifikationen

werden die Dienste einer Schnittstelle definiert. Eine Signatur repräsentiert einen Dienst und ist mit einer Methodensignatur von Programmiersprachen vergleichbar. Wie diese besteht eine Signatur des PCM aus einem Rückgabewert, einem Namen, einer geordneten Parameterliste und einer Menge von Ausnahmen. Ein Protokoll legt die gültigen Aufrufreihenfolgen von Signaturen fest.

Service Effect Specification

Für jeden angebotenen Dienst einer Komponente kann eine *Service Effect Specification* (SEFF) erstellt werden, die festlegt, wie der angebotene Dienst die benötigten Dienste der Komponente in Anspruch nimmt. Vom internen Kontrollfluss der Komponente wird jedoch abstrahiert, sofern diese keine Auswirkungen auf die externen Aufrufe haben. Für die Performance-Vorhersage werden spezielle SEFFs eingesetzt, mit denen der Ressourcenbedarf, der Datenfluss und Parameterabhängigkeiten spezifiziert werden können. Dafür stehen unter anderem interne und externe Aktionen, Schleifen oder Verzweigungen zur Verfügung.

Kontexte und Konnektoren

Für die Verknüpfung von Komponenten zu Komponentenarchitekturen wird das Konzept des Kontexts benötigt, das die Verwendung einer Komponente in einer Komponentenarchitektur eindeutig macht. Eine eindeutig identifizierbare Komponentenverwendung in einer Komponentenarchitektur wird als Kontext-Komponente bezeichnet, dabei können für eine Komponente mehrere Kontext-Komponenten existieren. Die Verknüpfung der Kontext-Komponenten erfolgt über Konnektoren, die genau eine benötigte Rolle einer Komponente mit einer angebotenen Rolle einer anderen Komponente verbinden. Eine solche Verknüpfung bedeutet, dass jeder Aufruf der Kontext-Komponente, die einen Dienst der benötigten Rolle braucht, an die verknüpfte Kontext-Komponente weitergeleitet wird, die diesen Dienst über ihre Rolle anbietet. Konnektoren müssen die Anforderung erfüllen, dass die angebotene Rolle ein Supertyp der benötigten Rolle ist, damit die Interoperabilität der Kontext-Komponenten gewährleistet ist.

Ressourcenallokation

Kontext-Komponenten und Konnektoren müssen auf den Ressourcen der Ressourcenumgebung allokiert werden. Dies erfolgt über einen Allokationskontext, mit dem die Zuordnung zwischen einer Kontext-Komponente und einem Ressourcen-Container der Ressourcenumgebung festgelegt wird. Die Ressourcen der Ressourcenumgebungen werden in berechnende und passive Ressourcentypen unterteilt. Beispiele für berechnende Ressourcen sind Prozessoren oder Festplatten. Semaphoren oder Ressourcen-Pools sind hingegen passive Ressourcen. Ein Spezialfall der berechnenden Ressourcen stellen die kommunizierenden Ressourcen dar, die zur Beschreibung der Verbindungen zwischen Ressourcen-Containern verwendet werden. Ein Konnektor wird implizit einer kommunizierenden Ressource zugeordnet, falls die Kontext-Komponenten, die durch diesen Konnektor verknüpft werden, auf unterschiedlichen Ressourcen-Containern allokiert werden.

Verwendungsmodell

In das Palladio Komponentenmodell ist ein Verwendungsmodell integriert, mit dessen Instanzen Benutzerinteraktionen mit einem System spezifiziert werden können. Eine Instanz des Verwendungsmodells beschreibt mittels Verwendungsszenarien jene Dienste, die direkt von Benutzern aufgerufen werden sowie die Reihenfolge der Aufrufe. Zusätzlich wird in einem Verwendungsszenario die Last der Benutzerinteraktion festgelegt.

3.4 Java Platform, Enterprise Edition

Die *Java Platform, Enterprise Edition* (Java EE) [Java06] ist Spezifikation einer Plattform, die auf die Entwicklung von komplexen, verteilten Geschäftsanwendungen mit der Programmiersprache Java ausgerichtet ist. Das mit Java EE verbundene Ziel ist die Reduktion der Kosten und der Komplexität bei der Entwicklung solcher Anwendungen. Java EE sieht dafür ein spezielles Programmiermodell mit einer Menge von spezifizierten Schnittstellen und Diensten vor. Beispiele für solche Plattformdienste sind ein Transaktionsdienst, ein Persistenzdienst, ein Sicherheitsdienst, ein Namensdienst, das Ressourcenmanagement oder ein Nachrichtendienst. Dies hat den Vorteil, dass solche Dienste von den Anwendungen in Anspruch genommen werden können und nicht bei der Anwendungsentwicklung implementiert werden müssen. Die Architektur der Plattform selbst besteht aus den vier Containern *Applet Container*, *Application Client Container*, *Web Container* und *EJB Container*. Diese Container sind Laufzeitumgebungen, die jene Dienste anbieten, welche von Anwendungskomponenten benötigt werden. Innerhalb dieser Container kommunizieren die Anwendungskomponenten niemals direkt miteinander. Die Anwendungskomponenten nutzen vielmehr die Funktionalität des Containers, um miteinander zu kommunizieren oder um Dienste der Plattform in Anspruch zu nehmen. Damit ist der Container in der Lage, transparent Dienste zu verwenden, die von einer Komponente benötigt werden. Im Kontext dieser Arbeit ist vor allem der *EJB Container* von Bedeutung, da dieser als Laufzeitumgebung von *Enterprise JavaBeans* fungiert. Im folgenden Unterkapitel werden die Konzepte dieser Anwendungskomponenten erläutert.

3.4.1 Enterprise JavaBeans

Enterprise JavaBeans (EJB) [Ente06] sind standardisierte wiederverwendbare Anwendungskomponenten, die typischerweise die Geschäftslogik kapseln. Seit Java EE Version 5 gibt es zwei unterschiedliche EJB-Typen, *Session-Beans* und *Message-Driven Beans*. Die *Entity Beans* vorheriger Java EE-Versionen wurden durch Entitäten des Persistenzdiensts ersetzt. Seit dieser Version sind EJBs einfache Java-Objekte, die durch Annotationen um Metadaten ergänzt werden, um die Plattformdienste zu spezifizieren, die eine EJB-Komponente benötigt. Der Container wertet diese Annotationen aus und realisiert die Verwendung der Dienste.

3.4.2 Session Beans

Session Beans werden von Klienten aufgerufen, um eine Operation der Geschäftslogik durchzuführen. Die Klienten können sich dabei sowohl innerhalb als auch außerhalb des Containers befinden. Eine *Session Bean* kann nur von einem Klienten

verwendet werden und sie persistiert ihre Daten nicht. Es gibt zwei unterschiedliche Typen dieser Komponenten, zustandslose (*stateless*) und zustandsbehaftete (*stateful*) *Session Beans*. *Stateful Session Beans* speichern automatisch den Zustand zwischen den Aufrufen eines Klienten. Dies erfolgt allerdings nur für den Zeitraum einer Sitzung zwischen dem Klienten und der *Stateful Session Bean*. Wird diese Sitzung beendet, so wird der Zustand der *Stateful Session Bean* verworfen. *Stateless Session Beans* unterhalten hingegen keinen Zustand, so dass sie einen möglichen Zustand nur für die Dauer eines einzigen Aufrufs speichern. Nach Beendigung eines Aufrufs wird der interne Zustand nicht weiter beibehalten. Alle Instanzen einer *Stateless Session Bean* sind gleichwertig, mit Ausnahme für die Dauer eines Methodenaufrufs. Dies ermöglicht dem Container, einem Klienten eine beliebige Instanz zuzuweisen.

3.4.3 Message-Driven Beans

Genauso wie *Session Beans* kapseln *Message-Driven Beans* Teile der Geschäftslogik. Sie werden jedoch niemals über ihre Schnittstellen direkt von Klienten aufgerufen. Stattdessen verwenden sie den Nachrichtendienst JMS der Java EE Plattform, um Nachrichten über einen Nachrichtenkanal zu senden, an dem die benötigte *Message Driven Bean* registriert ist. Die Ausführung der Logik einer *Message Driven Bean* wird über den Empfang einer Nachricht ausgelöst, die der *Message Driven Bean* von einem Nachrichtensystem zugestellt werden. Somit können *Message Driven Beans* für die asynchrone Ausführung der Geschäftslogik eingesetzt werden. Instanzen von *Message Driven Beans* speichern darüber hinaus keinen Zustand und sind alle gleichwertig, so dass der Container einer beliebigen Instanz eine Nachricht zuweisen kann.

3.5 Nachrichtenorientierte Kommunikation

Nachrichtenorientierte Kommunikation ist eine asynchrone Kommunikationsform zwischen lose gekoppelten Software-Komponenten, die durch den Austausch von Nachrichten miteinander kommunizieren. Der Nachrichtentransport erfolgt entlang von Nachrichtenkanälen, die eine logische Verbindung zwischen den kommunizierenden Komponenten darstellen. Ein Nachrichtenkanal kann als Warteschlange betrachten werden, zu welcher sendende Komponenten Nachrichten hinzufügen und aus welcher empfangende Komponenten Nachrichten entnehmen. Eine Nachricht ist eine einfache Datenstruktur und besteht aus einem Nachrichtenkopf und einem Nachrichtenrumpf. Im Nachrichtenkopf werden Meta-Informationen über die Nachricht aufgenommen, wie z. B. eine Identifikation des Senders oder das Ziel der Nachricht. Diese Informationen werden von den kommunizierenden Komponenten meistens ignoriert. Für sie ist vielmehr der Nachrichtenrumpf von Bedeutung, da dieser die auszutauschenden Daten aufnimmt. Für die Kommunikation mittels Nachrichtenaustausch ist ein Nachrichtensystem essenziell notwendig, da es die Prozesse des Nachrichtenversands und -empfangs koordiniert und verwaltet sowie den Transfer von der Senderkomponente zur Empfängerkomponente durchführt. Dabei können Sender und Empfänger auf verschiedene Rechner verteilt sein. Daraus ergibt sich die Zuverlässigkeit als weitere Anforderung an ein Nachrichtensystem, da Nachrichten auch über unzuverlässige Kommunikationsverbindungen zuverlässig zugestellt werden müssen. Nachrichtenorientierte Kommunikation bietet Vorteile für eine Reihe von Anwendungskontexte. Ein Beispiel ist die Integration von Anwendungen, da Anwendungen,

die möglicherweise in verschiedenen Programmiersprachen implementiert wurden, auf verschiedenen Plattformen über den Austausch von Nachrichten kommunizieren können. Mit dem Einsatz von asynchroner Kommunikation sind jedoch auch besondere Herausforderungen verbunden. Zum einen ist das ereignisorientierte Programmiermodell komplexer als das kontrollflussorientierte, beispielsweise können mehrere Nachrichtenkanäle erforderlich sein, um einen einfachen Methodenaufruf repräsentieren zu können. Darüber hinaus wird keine Reihenfolgetreue garantiert, so dass dies bei Bedarf explizit behandelt werden muss. Mit nachrichtenorientierter Kommunikation ist ein Aufwand verbunden, der durch das Erstellen einer Nachricht aus den Anwendungsdaten, dem Nachrichtenversand, dem Nachrichtentransport und dem Nachrichtenempfang entsteht. In [HoWo03] stellen Hohpe und Woolf Entwurfsmuster vor, die für diese Herausforderungen Lösungsmöglichkeiten aufzeigen.

3.5.1 Java Message Service

Java Message Service (JMS) [Java02] ist eine Spezifikation einer Menge an Schnittstellen und der damit verbundenen Semantik für den Zugriff von Java-Anwendungen auf Nachrichtensysteme. Eine Anwendung, die über JMS kommuniziert, besteht gemäß der Spezifikation aus *JMS Clients*, Nachrichten, einem *JMS Provider* und administrierten Objekten. *JMS Clients* sind hier die Java-Anwendungen, die Nachrichten versenden bzw. empfangen. Mit *JMS Provider* ist das Nachrichtensystem bezeichnet, das die JMS-Spezifikation implementiert. Administrierte Objekte sind vorkonfigurierte JMS-Objekte, die bei der nachrichtenorientierten Kommunikation verwendet werden können. Konkret sind dies die Typen *ConnectionFactory* und *Destination*, wobei ersteres von Sendern bzw. Empfängern zur Herstellung einer Verbindung mit dem Nachrichtensystem verwendet wird. Letzteres bezeichnet hingegen einen speziellen Nachrichtenkanal. JMS unterstützt zwei verschiedene Nachrichtenkanaltypen, einen Punkt-zu-Punkt-Nachrichtenkanal, an dem nur ein Empfänger registriert sein kann und einen *Publish/Subscribe*-Kanal, bei dem jede versendete Nachricht an alle registrierten Empfänger übertragen wird. Das Programmiermodell von JMS besteht neben den bereits erwähnten *ConnectionFactory* und *Destination* aus den Objekten *Connection*, *Session*, *MessageProducer* und *MessageConsumer*. Die *Connection* repräsentiert eine aktive Kommunikationsverbindung zum Nachrichtensystem, eine *Session* stellt einen Kontext für den Nachrichtenversand bzw. -empfang dar. Ein *MessageProducer* ist ein Objekt, das von einem *Session*-Objekt erzeugt wird und für den Nachrichtenversand verwendet wird. Dessen Gegenstück auf Empfängerseite ist der *MessageConsumer*. Dabei bestehen die folgenden Beziehungen zwischen diesen Objekten:

- mit einer *ConnectionFactory* wird eine *Connection* erzeugt
- für eine *Connection* wird eine *Session* erzeugt
- eine *Session* erzeugt einen *MessageProducer* bzw. einen *MessageConsumer*
- für eine *Session* wird eine Nachricht erzeugt

JMS-Nachrichten enthalten zusätzlich zum Nachrichtenkopf und Nachrichtenrumpf einen Teil für Nachrichteneigenschaften. Damit ist es möglich zu einer Nachricht

optionale Meta-Informationen hinzuzufügen, die nicht über die spezifizierten Eigenschaften des Nachrichtenkopfs erfasst werden können. JMS bietet verschiedene Mechanismen, wie ein Empfänger den Empfang einer Nachricht bestätigt. Zum einen kann dies automatisch im Nachrichtensystem erfolgen oder explizit vom Nachrichtenempfänger durchgeführt werden. JMS bietet die Möglichkeit, dass das Nachrichtensystem für Empfänger eine Filterung von Nachrichten durchführt und spezifiziert dafür eine Filtersyntax, die an eine Teilmenge der Sprache SQL92 angelehnt ist.

3.6 Evolutionäre Algorithmen

Evolutionäre Algorithmen [EiSm03] sind heuristische Optimierungsverfahren, die sich für die Lösung von schwer lösbaren Problemen eignen. Evolutionäre Algorithmen sind an die Evolutionstheorie der Biologie angelehnt, nach der in einer begrenzten Umgebung eine Auslese unter den einzelnen konkurrierenden Individuen stattfindet. Dabei werden jene Individuen bevorzugt, die am besten an die Umgebung angepasst sind. Ein weiterer Grundpfeiler dieser Theorie leitet sich aus den phänotypischen Variationen unter den Mitgliedern einer Population ab. Jedes Individuum besitzt einzigartige phänotypische Züge, die direkt seine Anpassungsfähigkeit an die Umwelt bestimmen. Wenn sich eine solche Kombination durch die Auslese als vorteilhaft erweist, wird diese an die Nachkommen weitergegeben, ansonsten hat diese Kombination schlechte Überlebenschancen. Zusätzlich können über Mutationen zufällige Phänotypkombinationen entstehen. Die phänotypischen Züge werden durch die inneren Merkmale eines Individuums bestimmt, dabei stellen Gene funktionale Vererbungseinheiten dar, die phänotypische Eigenschaften kodieren. Somit werden phänotypische Variationen stets durch genotypische Variationen verursacht. Evolutionäre Algorithmen nutzen diesen Prozess, um automatisch für einen breiten Problembereich möglichst gute Ergebnisse in akzeptabler Zeit zu liefern. Allgemein betrachtet wird bei evolutionären Algorithmen eine zufällige Menge an Lösungskandidaten für das Problem anhand einer Qualitätsfunktion bewertet und einige der besten Kandidaten für die Erzeugung der nächsten Generation ausgewählt. Für die Erzeugung einer neuen Generation werden für diese Kandidaten mittels Rekombination oder Mutation Nachkommen erzeugt. Diese neuen Kandidaten konkurrieren mit den bestehenden um einen Platz in der neuen Generation. Dieser Prozess kann iterativ durchgeführt werden, bis ein Kandidat mit ausreichender Qualität gefunden ist. Im Allgemeinen führt die Anwendung der Prinzipien der Variation und der Selektion zu einer Verbesserung bzw. einer Beibehaltung der Qualität der gefundenen Lösung zwischen aufeinanderfolgenden Lösungen. Die Hauptbestandteile eines genetischen Algorithmus sind eine Individuendefinition, eine Fitnessfunktion, eine Population, ein Mechanismus zur Auswahl von Eltern, Variationsoperatoren und ein Auslesemechanismus. Bei der Individuendefinition werden Objekte, die mögliche Lösungen im originalen Problemkontext bilden, auf Individuen des evolutionären Algorithmus abgebildet. Die Fitnessfunktion repräsentiert die Anforderungen, an die sich die Individuen anpassen müssen und bildet damit den Kern für die Individuenauslese. Eine Population beinhaltet Individuen als mögliche Lösungskandidaten und stellt somit die Einheit dar, die sich durch Selektion und Variation ändert und an die Anforderungen anpasst, wohingegen Individuen statische Elemente mit einem spezifischen Phänotyp sind. Der Mechanismus zur Auswahl von Eltern ist erforderlich, um gute Individuen anhand ihrer Qualität für die Erzeugung der Folgegeneration auszuwählen. Die Erzeugung von neuen Individuen erfolgt über Variationsoperatoren, dies

sind die Mutation und Rekombination. Bei der Mutation wird aus einem einzelnen Individuum ein Mutant erzeugt, der einen veränderten Genotyp aufweist. Die Rekombination erzeugt mehrere Nachfahren mittels Verschmelzung der Genotypen mehrerer Elternindividuen. Der Auslesemechanismus wird für die Individuenauslese verwendet und ähnelt dem Mechanismus zur Elternauswahl, jedoch kommt dieser im Evolutionszyklus zur Anwendung, wenn bereits Nachkommen für die ausgewählten Elternindividuen erzeugt wurden.

3.6.1 Genetische Programmierung

Das Verfahren der genetischen Programmierung [EiSm03, S. 101-114] gehört zur Familie der evolutionären Algorithmen. Folglich wendet sie deren Evolutionsprozess an, zeichnet sich jedoch dadurch aus, dass Syntaxbäume als Individuen verwendet werden. Über diese Syntaxbäume können Ausdrücke einer gegebenen formalen Syntax erfasst werden. Beispiele hierfür sind arithmetische Ausdrücke, Formeln in Prädikatenlogik erster Stufe oder *if-then-else*-Verzweigungskonstrukte. Die genetische Programmierung ist allgemeiner als andere evolutionäre Algorithmen, da aufgrund der speziellen Individuen ein größerer Suchraum abgedeckt werden kann und weniger Annahmen über die mögliche Lösung getroffen werden. Die genetische Programmierung eignet sich besonders für Probleme, für welche die Ein- und Ausgabemengen bekannt sind, jedoch ein Modell gesucht wird, das für jede Eingabe die korrekte Ausgabe liefert. Damit lässt sie sich im Bereich des maschinellen Lernens ansiedeln.

4. Kategorisierung der Entwurfsmuster

Im Rahmen dieser Arbeit werden mehrere Entwurfsmuster aus [SRSS+00], [HoWo03], [Doug04], [PeSo97] und [CoDK05] betrachtet und zunächst in die Gruppen Ereignisbehandlung, Synchronisation, Nebenläufigkeit und nachrichtenorientierte Kommunikation eingeteilt [SRSS+00], da diese die Problembereiche sind, für die die Entwurfsmuster Lösungsschemata anbieten. Für Software-Architekten und Komponentenentwickler lässt sich daraus jedoch nicht ohne Detailkenntnisse der Entwurfsmuster ableiten, ob ein Entwurfsmuster für die Spezifikation einer Komponente oder für die Spezifikation der Interaktion von Komponenten geeignet ist. Eine Klassifizierung, die an das Vorgehen beim Entwurf von komponentenbasierter Software angepasst ist, verspricht eine einfachere Auswahl dieser Entwurfsmuster beim Software-Entwurf. Beispielsweise müsste Der Komponentenentwickler für die Spezifikation von Komponenten das geeignete Entwurfsmuster nur aus den jeweils relevanten Entwurfsmustern ausgesucht werden. Nebenläufige Software zeichnet sich vor allem dadurch aus, dass Dateninkonsistenzen bei der Ausführung verhindert werden müssen und die Nebenläufigkeit der Hardware effizient genutzt werden soll (siehe 3.1). Die Entwurfsmuster helfen dabei, diese Aspekte bei der Spezifikation der Komponenten und deren Interaktion zu berücksichtigen. Einige der Entwurfsmuster betreffen Funktionalität, die bereits von einer Infrastruktur aus Java und Java EE-konformem Anwendungs-Server erbracht wird.

4.1 Definition der Kategorien

Zwei zentrale Herausforderungen bei der Entwicklung von nebenläufiger Software sind Korrektheit und Effizienz [GPBB+05]. Im Gegensatz zu sequenzieller Software müssen bei nebenläufiger Software im Allgemeinen zusätzliche Maßnahmen, wie Koordination der beteiligten Komponenten, ergriffen werden, um einen korrekten Programmablauf zu gewährleisten. Der Grund dafür ist, dass die Ablaufreihenfolge erst zur Laufzeit vom Scheduler des Betriebssystems bestimmt wird. Zur Entwicklungszeit müssen daher bestimmte Ablaufkonstellationen mittels Koordinationsmechanismen verhindert werden, um dadurch einen korrekten Programmablauf zu erreichen,

der nicht vom Scheduling abhängig ist [GPBB+05]. Nebenläufige Software weist eine hohe Effizienz auf, wenn die auf Hardware-Ebene verfügbare Parallelität ausgenutzt werden kann (siehe Abschnitt 3.1). Dafür ist es erforderlich, dass die nebenläufigen Komponenten und die Interaktion zwischen den Komponenten optimiert werden, damit möglichst viele der Komponenten ablaufbereit sind.
Die hohe Komplexität von nebenläufiger, komponentenbasierter Software ist eng mit mit den Aspekten der Korrektheit und Effizienz verknüpft [SuLa05]. Im Vergleich mit sequenzieller Software sind für die Korrektheit und Effizient von nebenläufiger Software zusätzliche Maßnahmen wie z. B. Synchronisationsmechanismen erforderlich. Ein erster Schritt zur Verringerung der Komplexität ist die Verwendung von Entwurfsmustern, denn diese bieten Lösungsschemata für häufig auftretende Probleme von nebenläufigen Systemen an. Ein weiterer Schritt ist eine hierarchische Betrachtung. Zunächst kann sich der Fokus auf die Korrektheit und Effizienz jeder einzelnen Komponenten richten, bevor die Interaktion der Komponenten in den Vordergrund rückt.
Diese hierarchische Vorgehensweise lässt sich mit dem Palladio Komponentenmodell verbinden, das einen hierarchischen Ansatz bei der Spezifikation von Komponenten und ihrer Zusammensetzung zu einer Anwendung verfolgt [RBKH+07]. Eine Zuordnung der Entwurfsmuster zu den Hierarchiestufen Komponentenspezifikation und Komponenteninteraktion erleichtert ihre Anwendung beim Entwurf und ermöglicht die Erstellung von Modellkonstrukten für die Entwurfsmuster. Die Verwendung von Modellkonstrukten vereinfacht die Modellierung und verbessert die Vorhersagbarkeit, da die Komplexität der Entwurfsmuster in den Entwurfskonstrukten gekapselt ist und die Festlegung der Performance-relevanten Parameter für die Performance-Vorhersage in den Mittelpunkt rückt.
Die rein funktionale Gruppierung der Entwurfsmuster, wie sie in [SRSS+00] verfolgt wird, lässt sich nicht direkt mit dem Entwurf von nebenläufiger, komponentenbasierter Software verbinden, da die Problembetrachtung auf einer Ebene mit geringerem Abstraktionsgrad erfolgt als beim Software-Entwurf. Eine Gruppierung auf vergleichbarer Abstraktionsebene verspricht eine einfachere Auswahl der Entwurfsmuster beim Software-Entwurf. Beispielsweise sind Entwurfsmuster, die die Komponenteninteraktion betreffen bei der Spezifikation von Basiskomponenten nicht relevant und müssen daher nicht betrachtet werden. Aus diesem Grund wird in dieser Arbeit eine Kategorisierung der Entwurfsmuster entsprechend ihrer Anwendbarkeit bei der hierarchischer Spezifikation von Komponenten und ihrer Zusammensetzung vorgeschlagen. Die Kategorie „Thread-Sicherheit bei Komponenten“ umfasst alle Entwurfsmuster, die bei der Spezifikation von Basiskomponenten eingesetzt werden können und darauf ausgerichtet sind, Dateninkonsistenzen bei konkurrierendem Zugriff zu verhindern. In der Kategorie „Komponenteninteraktion“ sind diejenigen Entwurfsmuster zusammengefasst, die bei der Spezifikation der Komponenteninteraktion verwendbar sind. Diese bieten Lösungsstrategien für die Koordination und Optimierung der Interaktion von Komponenten. Eine dritte Kategorie ist orthogonal zu den beiden genannten Kategorien. Sie ist darin begründet, dass im Rahmen dieser Arbeit komponentenbasierte Anwendungen betrachtet werden, die auf einer Middleware basieren. Konkret erfolgt sogar eine Einschränkung auf die Programmiersprache Java und auf einen Java EE-konformem Anwendungs-Server als Laufzeitumgebung. Die Kategorie umfasst die Entwurfsmuster, deren Funktionalität von der Infrastruktur aus Java und Java EE-konformem Anwendungs-Server erbracht wird. Diese Entwurfsmuster sind daher beim Entwurf und der Implementierung die-

ser Infrastruktur von Bedeutung. Für nebenläufige, komponentenbasierte Software, die auf dieser Infrastruktur basiert, sind diese Muster hingegen nur indirekt relevant. Diese Kategorie wird daher als „Infrastruktur“ bezeichnet.
Im Folgenden werden die Kategorien im Detail beschrieben und anschließend die Kategorisierung der in [SRSS+00], [HoWo03], [Doug04], [PeSo97] und [CoDK05] beschriebenen Entwurfsmuster erläutert.

4.1.1 Kategorie Thread-Sicherheit bei Komponenten

Bei nebenläufiger Software ist es möglich, dass mehrere Komponenten auf eine weitere gemeinsame Komponente parallel zugreifen. Der Ausführungszustand dieser Komponente wird dabei von den anderen Komponenten gemeinsam genutzt. Beispielsweise umfasst der Ausführungszustand eines Objekts alle Daten des Objekts. Diese Daten müssen gegen unkontrollierten nebenläufigen Zugriff geschützt werden, damit beim Zugriff von mehreren Threads keine unerwünschten Folgen wie korrumpierte Daten auftreten. Komponenten werden als Thread-sicher bezeichnet, wenn sie sich auch bei nebenläufigem Zugriff entsprechend ihrer Spezifikation verhalten. Thread-Sicherheit ist unabhängig vom Scheduling und darf keine zusätzliche Koordination von Komponenten erfordern, die diese Komponente verwenden [GPBB+05, S.18].
Entwurfsmuster, die Lösungsschemata anbieten, um Komponenten Thread-sicher zu machen, werden in die Kategorie Thread-sichere Komponenten eingeteilt. Für diese Entwurfsmuster ist die Kommunikation und Interaktion der Komponenten transparent. Die Zugriffskoordination beeinflusst die Effizienz, da bei der Koordination des Zugriffs auf den gemeinsamen Ausführungszustand nebenläufige Komponenten blockiert werden können. In dieser Kategorie wird lediglich die Thread-Sicherheit einer Komponente selbst betrachtet, da eine effiziente Zugriffskoordination erreicht werden kann, ohne dass Kenntnis über die Form der Interaktion mit anderen Komponenten vorhanden sein muss.
Thread-Sicherheit kann im Palladio Komponentenmodell nur auf die Basiskomponenten angewandt werden, da zusammengesetzte Komponenten nur eine Zusammenfassung von einer Komponentenmenge darstellt, aber kein eigenes Verhalten haben. Der Komponentenentwickler ist im Palladio Komponentenmodell für die Spezifikation und Implementierung von Basiskomponenten verantwortlich und kann deshalb die Entwurfsmuster dieser Kategorie verwenden, falls Komponenten Thread-sicher sein müssen.

4.1.2 Kategorie Komponenteninteraktion

Nach Sutter und Larus [SuLa05] ist in den meisten Fällen die Parallelität in nebenläufiger Software unstrukturiert. Im Allgemeinen interagieren bei unstrukturierter Parallelität mehrere nebenläufige Komponenten miteinander und weisen eine hohe Komplexität auf, da sich die nebenläufigen Aktivitäten stark unterscheiden (siehe Abschnitt 3.1). Dabei wird jedoch nicht ohne weiteres die Parallelität auf Hardware-Ebene effizient ausgenutzt. Es ist beispielsweise möglich, dass schlimmstenfalls alle Threads sequenziell laufen, weil der laufende Thread alle anderen Threads blockiert. Für eine effiziente Ausnutzung der vorhandenen Hardware-Parallelität muss daher die Interaktion der nebenläufigen Komponenten optimiert werden. Die Interaktion von nebenläufigen Komponenten kann eine Synchronisation der beteiligten Komponenten erfordern. Zum Beispiel könnte der Programmablauf bei Systemen mit

Replikation erst dann fortgesetzt werden, wenn alle Datenkopien geändert wurden. Entwurfsmuster, die Lösungsschemata anbieten, um die Interaktion zwischen nebenläufigen Komponenten zu koordinieren und optimieren, werden der Kategorie Komponenteninteraktion zugeordnet. Diese Muster abstrahieren vom internen Verhalten der Komponenten, essentiell ist lediglich deren Interaktion mit anderen Komponenten. Für einen korrekten Programmablauf kann es erforderlich sein, dass die Komponenteninteraktion synchronisiert erfolgt. Beispielsweise kann eine Barriere dafür eingesetzt werden, dass alle nebenläufigen Komponenten an der Barriere aufeinander warten, bis jede der Komponenten ihre Berechnungen abgeschlossen hat, um alle Berechnungsergebnisse als konsistente Datenbasis für die darauffolgende Berechnungsiteration zur Verfügung zu haben. Für die Gewährleistung der Korrektheit sind hier im Gegensatz zur Kategorie Thread-sichere Komponenten (siehe Abschnitt 4.1.1) mehrere Komponenten beteiligt, die miteinander interagieren.
Im Palladio Komponentenmodell übernimmt der Software-Architekt die Aufgabe der Spezifikation der Zusammensetzung von Komponenten. Der Komponentenentwickler spezifiziert bei zusammengesetzten Komponenten die enthaltenen Komponenten und deren Verknüpfung. Beide Rollen befassen sich somit mit der Komponenteninteraktion, so dass beide die Entwurfsmuster dieser Kategorie verwenden können.

4.1.3 Kategorie Infrastruktur

Im Rahmen dieser Arbeit wird nur nebenläufige, komponentenbasierte Software betrachtet, die in der Programmiersprache Java erstellt wird und einen Java EE-konformen Anwendungs-Server [Ente06] als Laufzeitumgebung verwendet. Die Aufgaben eines solchen Anwendungs-Servers umfassen das Ressourcenmanagement sowie Nebenläufigkeit, Transaktionsdienst, Persistenzdienst, Objekt-Verteilung, Namensdienst und Sicherheit als elementaren Dienste [BuMH06]. Einige der Entwurfsmuster betreffen Funktionalität, die von konkreten Anwendungen unabhängig ist und entsprechend der Aufgaben von der Infrastruktur aus Java und Java EE Anwendungs-Servern erbracht wird. Die interne Struktur der bereitgestellten Konstrukte ist für die Verwendung transparent.

4.2 Kategorisierung

Den im vorherigen Abschnitt definierten Kategorien werden Entwurfsmuster aus [SRSS+00], [HoWo03], [Doug04], [PeSo97] und [CoDK05] zugeordnet. Als Kriterium dient die Verwendbarkeit beim hierarchischen Vorgehen beim Entwurf komponentenbasierter Software. Für die Kategorie „Infrastruktur“ sind die Programmiersprache Java und Java EE-konforme Anwendungs-Server relevant. Die Tabelle 4.1 zeigt das Resultat der Kategorisierung, die in den folgenden Unterabschnitten für jedes Entwurfsmuster begründet wird. Dabei werden für die Entwurfsmuster die englische Namen verwendet und diese durch kursive Schriftform hervorgehoben, um so Missverständnisse zu vermeiden.

4.2.1 Entwurfsmuster zur Ereignisbehandlung

Für große verteilte Software-Systeme stellt die ereignisorientierte Kommunikation ein effektiver Kommunikationsmechanismus zwischen Komponenten dar [SRSS+00]

	Ereignis-behandlung	Synchronisation	Neben-läufigkeit	Nachrichten-orientierte Kommunikation
Thread-Sicherheit bei Komponenten		Scoped Locking Strategized Locking Thread-safe Interface Double-Checked Locking Optimization	Thread-specific Storage Monitor Object	
Komponenten-interaktion	Asynchro-nous Completion Token	Rendezvous	Replikation	Nachrichten-kanäle Nachrichten-Routing Nachrichten-endpunkte
Infrastruktur	Reactor Proactor Acceptor-Connector		Active Object Half-Sync/ Half-Async Leader-Followers Thread Pool	

Tabelle 4.1: Kategorisierung der Entwurfsmuster

Ihre Vorteile sind die lose Kopplung der Kommunikationsteilnehmer und ihre sehr hohe Skalierbarkeit [PiSB03]. Das Verhalten von ereignisorientierten Anwendungen wird im Unterschied zu kontrollflussorientierten Anwendungen durch interne oder externe asynchrone Ereignisse ausgelöst, wobei bedeutend ist, das die Ereignisse schnell bearbeitet werden. Die Ereignisverarbeitung kann mittels endlicher Automaten kontrolliert werden. Die Entwurfsmuster *Reactor*, *Proactor*, *Asynchronous Completion Token* und *Acceptor-Connector* aus [SRSS+00] zeigen Lösungsschemata für Probleme bei der Entwicklung von ereignisorientierten Software-Systemen.

4.2.1.1 Reactor

Ereignisorientierte Software muss in der Lage sein, mehrere Anfragen gleichzeitig abarbeiten zu können, ohne dass eine Ereignisquelle andere Ereignisquellen blockiert. Bedeutend ist weiterhin, die Minimierung von Kontextwechseln, Synchronisation sowie Datentransfers zwischen Prozessoren. Die Komplexität von Nebenläufigkeit und Synchronisation soll zudem möglichst vor der Anwendung verborgen bleiben. Das Entwurfsmuster *Reactor* [SRSS+00, S. 179-214] sieht für jeden Anwendungsdienst eine separate `Event Handler`-Komponente vor, die die Ereignisse eines bestimmten Typs von bestimmten Ereignisquellen bearbeiten. Jede dieser Komponenten registriert sich bei einer `Reactor`-Komponente. Die `Reactor`-Komponente wird von einem `SynchronousEventDemultiplexer` über auftretende Ereignisse benachrichtigt. Ein Beispiel für einen `SynchronousEventDemultiplexer` ist die von vielen Betriebssystemen unterstützte `select()`-Funktion für Ein-/Ausgabeereignisse. Nach der Ereignisbenachrichtigung initiiert die `Reactor`-Komponente die Weiterverarbeitung des Ereignisses durch die zuständige `Event Handler`-Komponente.
Das Entwurfsmuster *Reactor* ermöglicht somit ereignisorientierten Anwendungen das Demultiplexen von Ereignissen und die Steuerung von deren Weiterverarbeitung. Betrachtet werden Ereignisse auf der Ebene des Zugriffs auf Ein/Ausgabeprimitive, wie zum Beispiel ein TCP-Socket. Die Komponenten dieses Entwurfsmusters können fast vollständig auf Klassen des `java.nio` -Pakets abgebildet werden. Einzig der Rückfragemechanismus zum Starten der Weiterverarbeitung nach dem Demultiplexen müsste ergänzt werden [dBur02]. Bei der Entwicklung komponentenbasierte Software unter Verwendung eines Anwendungs-Servers, übernimmt der Anwendungs-Server die Kommunikationsaufgaben. Der Anwendungs-Server hat somit die vom *Reactor* adressierten Probleme zu lösen, so dass diese für die Anwendung selbst transparent sind. Das Entwurfsmuster *Reactor* wird in Folge dessen in die Kategorie Infrastruktur eingeordnet.
Das Demultiplexen und die weitere Bearbeitung der eintreffenden Ereignisse durch `Event Handler`-Komponente kann auf verschiedenen Varianten erfolgen, die die Performance beeinflussen können. Die `Event Handler`-Komponenten können entweder im Thread der `Reactor`-Komponente oder in einem eigenen Thread laufen, wobei letzteres es ermöglicht, dass die Ereignisbearbeitung nebenläufig erfolgen kann. Werden nebenläufige `SynchronousEventDemultiplexer` eingesetzt, können mehrere Threads gleichzeitig Ereignisse demultiplexen.

4.2.1.2 Proactor

Eine Verbesserung der Performance kann bei ereignisorientierter Software oftmals dadurch erreicht werden, dass Dienstanfragen asynchron bearbeitet werden. Dazu sollte eine Anwendung mehrere Ereignisse, mit denen das Betriebssystem die Beendigung der asynchronen Verarbeitung anzeigt, gleichzeitig behandeln können. Die

Performance soll nicht durch unnötige Kontextwechsel, Synchronisation oder Datentransfers zwischen Prozessoren beeinträchtigt werden. Das Entwurfsmuster *Proactor* [SRSS+00, S. 215-260] sieht eine Aufteilung der Anwendung als Lösung. Dabei werden langlaufende, asynchron auszuführende Operation von der weiteren Bearbeitung ihrer Ergebnisse durch `CompletionHandler`-Komponenten getrennt. Nachdem die Bearbeitung einer asynchronen Operation beendet ist, muss die Anwendung das entsprechende Ereignis behandeln. Zum Beispiel muss eine Anwendung jedes Ereignis, das die Beendigung einer asynchronen Operation anzeigt, demultiplexen und an eine interne Komponente zur Verarbeitung der Ergebnisse übergeben. Dazu wird zunächst das Beendigungsereignis in eine `Completion Event Queue` eingefügt. Eine `Proactor`-Komponente, demultiplext die asynchronen Ereignisse, indem sie sich dieser Warteschlange bedient und die Weiterverarbeitung der Ereignis durch die zugehörigen `CompletionHandler`-Komponenten veranlasst.
Das Entwurfsmuster *Proactor* kann als asynchrone Variante des *Reactor*-Musters betrachtet werden, denn es stellt transparent Funktionalität für das Demultiplexen und die Steuerung der Weiterverarbeitung von Ereignissen bereit, die von asynchronen Operationen ausgelöst werden. Analog zum *Reactor*-Muster kann es der Kategorie Infrastruktur zugeordnet werden, da seine Funktionalität auf einer sehr betriebssystemnahen Ebene anzusiedeln ist. Die Funktionalität des *Proactor*-Musters ist von konkreten Anwendungen unabhängig, ist für diese somit transparent und wird deshalb vom Anwendungs-Server übernommen.
Einfluss auf die Performance kann die Art des Demultiplexens und der Weiterverarbeitung von Ereignissen haben. Werden beispielsweise asynchrone `CompletionHandler` eingesetzt, so kann auch eine langlaufende Weiterverarbeitung von Ereignissen erfolgen, ohne die Reaktionsfähigkeit der Anwendung signifikant zu beeinträchtigen. Der Einsatz eines nebenläufigen `AsynchronousEventDemultiplexers` ermöglicht es der `Proactor`-Komponente, mehrere Ereignisse gleichzeitig zu demultiplexen.

4.2.1.3 Asynchronous Completion Token

Nach dem Aufruf asynchroner Dienste benötigt der Aufrufer zur Weiterverarbeitung der Dienstantwort den Kontext der Antwort. Dazu sieht das Entwurfsmuster *Asynchronous Completion Token* [SRSS+00, S. 261-284] einen eindeutigen Identifikator, den sog. `Asynchronous Completion Token` vor, der für jeden asynchronen Dienstaufruf erzeugt wird. Damit kann der Initiator des Dienstaufrufs die zugehörige Antwort identifizieren und die weitere Verarbeitung einleiten.
Das Entwurfsmuster *Asynchronous Completion Token* ermöglicht es Anwendungen, die Antworten auf asynchrone Dienstaufrufe effizient zu demultiplexen und zu verarbeiten. Es dient somit der Koordination von Komponenten die asynchron miteinander kommunizieren. Die Erzeugung des eindeutigen Identifikators für jeden Dienstaufruf sollte von der Infrastruktur übernommen werden und für die Anwendung selbst transparent sein, um die Eindeutigkeit des Identifikators zu gewährleisten und um Details der asynchronen Kommunikation vor der Anwendung zu verbergen. Ein Beispiel für *Asynchronous Completion Tokens* sind HTTP-Cookies [SRSS+00, S. 279]. Bei nachrichtenorientierter Kommunikation mittels des *Java Message Service* (siehe Abschnitt 3.5.1, [Java02]) kann das Feld `JMSCorelationID` im Nachrichtenkopf der Antwortnachricht auf die `JMSMessageID` der Anfragenachricht gesetzt werden, um so die Antwortnachricht mit der Anfragenachricht zu verknüpfen. Die `JMSCorelationID` wird jedoch im Gegensatz zur `JMSMessageID` nicht vom Nachrichtensystem generiert und gesetzt, sondern vom Sender der Antwort. Dieses Ent-

wurfsmuster wird für viele Anwendungskontexte bereits zur Verfügung gestellt. Es ist jedoch bei der Spezifikation der Interaktion von Komponenten, die durch Austausch von JMS-Nachrichten asynchron kommunizieren sollen, nicht transparent. Aus diesem Grunde wird das Entwurfsmuster *Asynchronous Completion Token* der Kategorie Komponenteninteraktion zugeordnet.

4.2.1.4 Acceptor-Connector

Anwendungen, die die verbindungsorientierte Kommunikationsform nutzen, beinhalten oft eine bedeutende Menge an Konfigurations-Code zur Herstellung von Verbindungen und der Initialisierung von Diensten. Da dieser Konfigurations-Code weitgehend unabhängig von den Anwendungsdiensten selbst ist, ist eine enge Kopplung zwischen beiden nicht gewünscht. Die Lösung des Entwurfsmusters *Acceptor-Connector* [SRSS+00, S. 285-322] kapselt jeden Anwendungsdienst in einer `Service Handler`-Komponente. Mittels einer `Acceptor`- und einer `Connector`-Fabrik kann eine Verbindung zu einer `Service Handler`-Komponente hergestellt und die Komponente initialisiert werden. Bei der `Acceptor`-Fabrik erfolgt der Verbindungsaufbau passiv, bei der `Connector`-Fabrik hingegen aktiv. Nach der Initialisierung einer `Service Handler`-Komponente kann diese die anwendungsspezifische Ausführung beginnen.
Das Muster *Acceptor-Connector* entkoppelt somit die Herstellung einer Verbindung zwischen Anwendungskomponenten von der darauf folgenden Bearbeitung. Für die Entwicklung von Server-Anwendungen ist dies vorteilhaft in Bezug auf Performance und Flexibilität. Bei Verwendung eines Anwendungs-Servers wird die Verbindungsherstellung von diesem durchgeführt und erfolgt transparent für die Anwendungskomponenten. Dieses Entwurfsmuster kann somit in die Kategorie Infrastruktur eingeordnet werden.
Für die Komponente `Connector` stehen zwei unterschiedliche Strategien zur Verfügung wie die `ServiceHandler`-Komponenten initialisiert werden. Abhängig vom Einsatzszenario kann die synchrone Initialisierung effizienter sein als die asynchrone Variante.

4.2.2 Synchronisationsmuster

Bei nebenläufiger Software kann der Ausführungszustand einer Komponente gleichzeitig von mehreren anderen Komponenten geändert werden, wodurch Daten korrumpiert werden können. Der Zugriff auf den Ausführungszustand nebenläufiger Komponenten muss mittels Synchronisationsmechanismen kontrolliert werden, damit Datenkorrumption verhindert wird. Die Entwurfsmuster *Scoped Locking*, *Strategized Locking*, *Thread-Safe Interface* und *Double-Checked Locking Optimization* aus [SRSS+00] bieten Lösungen zu Problemen, die in Zusammenhang mit der Synchronisation von nebenläufigen Komponenten auftreten können.

4.2.2.1 Scoped Locking

Der gemeinsame Ausführungszustand einer nebenläufigen Komponente muss mithilfe von Sperren vor unkontrolliertem Zugriff geschützt werden. Das Entwurfsmuster *Scoped Locking* [SRSS+00, S. 325-332] sieht eine `Guard`-Komponente vor, die vor dem Zugriff automatisch eine Sperre erlangt und diese nach dem Zugriff automatisch wieder freigibt.

Java verfügt über das Programmierkonstrukt des `synchronized`-Blocks, das Scoped Locking implementiert. Für einen Code-Block, der von `synchronized` umschlossen ist, generiert der Java-Compiler die Bytecode-Anweisungen `monitorenter` für den Eintritt und `monitorexit` für den Austritt aus dem `synchronized`-Block. Der Compiler generiert ebenfalls die Ausnahmebehandlung, so dass die Sperren in jedem Fall wieder freigegeben werden.
Das Entwurfsmuster *Scoped Locking* wird der Kategorie Thread-Sicherheit bei Komponenten zugeordnet, da es explizit über das `synchronized`-Konstrukt bei der Entwicklung von Basiskomponenten vom Komponentenentwickler eingesetzt wird.

4.2.2.2 Strategized Locking

Nebenläufige Komponenten müssen den Zugriff auf ihren Ausführungszustand koordinieren. Die Synchronisationsstrategie kann an die Anforderungen einer Anwendung angepasst werden, indem beispielsweise Semaphoren oder Leser-/Schreiber-Sperren eingesetzt werden. Das Entwurfsmuster *Strategized Locking* [SRSS+00, S. 333-343] ermöglicht einen Synchronisationsmechanismus, bei dem per Parameter die benötigte Synchronisationsstrategie gewählt werden kann.
Seit der Version 5.0 bietet Java im Paket `java.util.concurrent` verschiedene Sperrprimitive an, z.B. Semaphoren, Mutexe oder Leser-/Schreiber-Sperren. Zusammen mit den ebenfalls seit Version 5.0 eingeführten generischen Datentypen kann dieses Entwurfsmuster vom Komponentenentwickler bei der Entwicklung von Basiskomponenten angewandt werden, um für diese Thread-Sicherheit entsprechend der erforderlichen Synchronisationsstrategie zu erreichen. Strategized Locking stellt somit ein Entwurfsmuster der Kategorie Thread-Sicherheit bei Komponenten dar.

4.2.2.3 Thread-Safe Interface

Komponenten von nebenläufigen Anwendungen können komponenteninterne Methodenaufrufe enthalten. Um beim Aufruf solcher Methoden zu verhindern, dass eine Komponente versucht eine Sperre zu erlangen, die sie bereits selbst hält, kann das Entwurfsmuster *Thread-Safe Interface* [SRSS+00, S. 345-352] eingesetzt werden. Demzufolge sollen Komponenten mit internen Methodenaufrufen gemäß zweier Konventionen strukturiert sein. Die erste Konvention besagt, dass alle Schnittstellenoperationen nur Sperren der Komponente erlangen bzw. freigeben sollen. Zwischen dem Erlangen der Sperre und der Freigabe erfolgt die Methodenausführung durch die Komponente, die die Schnittstellenmethode implementiert. Die zweite Konvention sieht vor, dass Methodenimplementierungen nur dann ausgeführt werden sollen, wenn sie von Schnittstellenmethoden aufgerufen wurden. Dadurch wird sichergestellt, dass Sperren niemals von den Methodenimplementierungen erlangt oder freigegeben werden. Darüber hinaus hilft das Entwurfsmuster *Thread-Safe Interface* bei der Minimierung des Overheads, der in Zusammenhang mit Sperren entsteht.
Bei korrekter Verwendung des `synchronized`-Konstrukts von Java kann es nicht dazu kommen, dass eine Komponente auf eine Sperre wartet, die sie bereits selbst besitzt [GPBB+05, S. 26]. Dennoch bietet sich der Einsatz des Entwurfsmuster *Thread-Safe Interface* an, um den Sperr-Overhead zu verringern. Seit Java 5.0 ist dieses Muster bereits in einigen Bibliotheken implementiert. Beispiele sind `ConcurrentHashMap` oder `ConcurrentSkipListMap` des Pakets `java.util.concurrent`.
Dieses Entwurfsmuster kann bei der Entwicklung von Basiskomponenten angewandt

werden, um Thread-Sicherheit in Verbindung mit sparsamer Nutzung von Sperren zu erreichen. Da bisher nur wenige Bibliotheken vorhanden sind, die dieses Entwurfsmuster implementieren und der Einsatz im Allgemeinen anwendungsspezifisch ist, wird *Thread-Safe Interface* der Kategorie Thread-Sicherheit bei Komponenten zugeordnet.

4.2.2.4 Double-Checked Locking Optimization

Das Entwurfsmuster *Double-Checked Locking Optimization* [SRSS+00, S. 353-363] dient dazu, den Synchronisations-Overhead zu reduzieren, falls Sperren nur ein einziges Mal während der Programmausführung in Thread-sicherer Weise erlangt werden muss. Dazu wird ein Flag eingesetzt, das anzeigt, ob es nötig ist, einen kritischen Bereich auszuführen, bevor dessen Sperre erlangt werden muss. Unnötiger Synchronisations-Overhead wird dadurch vermieden, da die Sperre des kritischen Bereichs nur dann erlangt werden muss, falls der kritische Bereich ausgeführt werden muss. Beispielsweise kann dieses Entwurfsmuster bei der Thread-sicheren Initialisierung eines Singleton-Objekts verwendet werden. Das Flag zeigt in diesem Falle an, ob das Singleton-Objekt bereits instantiiert wurde.
In [GPBB+05, S. 348-349] wird dieses Entwurfsmuster als Anti-Muster angesehen und sollte somit in Verbindung mit Java nicht angewandt werden. Der Grund dafür liegt im Speichermodell von Java, so dass es möglich ist, dass eine Komponente auf eine Singleton-Komponente zugegriffen wird, die noch nicht vollständig initialisiert wurde. Seit Java 5.0 kann *Double-Checked Locking Optimization* eingesetzt werden, da das Speichermodell von Java geändert wurde. Dennoch werden stattdessen Alternativen empfohlen, da die ursprünglichen Gründe für *Double-Checked Locking Optimization*, wie z.B. langsame Synchronisation oder langsamer Start der Java Virtual Machine, keine Rolle mehr spielen.
Abgesehen von den Problemen bei der Verwendung unter Java kann das Entwurfsmuster *Double-Checked Locking Optimization* vom Komponentenentwickler eingesetzt werden, um effizient Thread-Sicherheit bei Basiskomponenten zu erreichen.

4.2.2.5 Rendezvous

Bei nebenläufiger Software ist es in bestimmten Situation erwünscht, dass die nebenläufigen Komponenten strukturiert zusammenarbeiten. Die Synchronisation von nebenläufigen Komponenten soll unabhängig von Scheduling-Strategien und Prioritäten erfolgen, sondern durch eine Synchronisationsstrategie bestimmt sein. Entsprechend dem Entwurfsmuster *Rendezvous* [Doug04, S. 579-584] registrieren sich die nebenläufigen Komponenten bei einer `Rendezvous`-Komponente, sobald sie für die Synchronisation bereit sind, und blockieren. Gemäß der Synchronisationsstrategie der `Rendezvous`-Komponente wird die weitere Ausführung der blockierenden nebenläufigen Komponenten initiiert. Das *Rendezvous*-Muster ermöglicht so die Synchronisation mehrerer Threads an einem Synchronisationspunkt. Eine einfache Variante des *Rendezvous*-Muster ist die Barriere, bei der eine Anzahl von nebenläufigen Komponenten aufeinander warten müssen, bevor sie ihre Abarbeitung fortsetzen können. Die Synchronisationsstrategie wäre in diesem Fall die Anzahl der erforderlichen Komponenten an der Barriere.
Dieses Entwurfsmuster bietet ein Lösungsschema für die Koordination und Optimierung von Komponenteninteraktion und kann vom Software-Architekt bei der Spezifikation des Zusammenspiels von Komponenten oder vom Komponentenentwickler

bei der Spezifikation von zusammengesetzten Komponenten angewendet werden. Das seit Java 5.0 vorhandene Paket `java.utli.concurrent` enthält die Klasse `CyclicBarrier`, die eine rudimentäre Implementierung des *Rendezvous*-Musters darstellt. Das Entwurfsmuster *Rendezvous* kann somit als Komponenteninteraktionsmuster kategorisiert werden.
Die Wartezeit der Threads am Synchronisationspunkt wird maßgeblich von der Synchronisationsstrategie, die die `Rendezvous`-Komponente verwendet, beeinflusst.

4.2.3 Nebenläufigkeitsmuster

Für nebenläufige Anwendungen können verschiedene Nebenläufigkeitsmechanismen ausgewählt werden, da jeder Nebenläufigkeitsmechanismus auf bestimmte Anforderungen abzielt. Für einige Probleme in Zusammenhang mit Nebenläufigkeit bieten die Entwurfsmuster *Active Object*, *Monitor Object*, *Half-Sync/Half-Async*, *Leader/-Followers* und *Thread-Specific Storage* Lösungen (siehe [SRSS+00]). Im Gegensatz zu *Thread-Safe Interface* wird das Muster *Monitor Object* als Nebenläufigkeitsmuster gruppiert, da hier die Koordination der Ablaufreihenfolge im Vordergrund steht.

4.2.3.1 Active Object

Mithilfe des Entwurfsmuster *Active Object* [SRSS+00, S. 369-398] kann die Ausführung einer Methode von ihrem Aufruf entkoppelt werden. Das Ziel ist die Verbesserung der Nebenläufigkeit und ein einfacherer synchronisierter Zugriff auf nebenläufige Komponenten. Eine Besonderheit dieses Entwurfsmusters ist, dass der Zugriff auf nebenläufige Komponenten in deren eigenem Ausführungskontext erfolgt und die Kontrolle nach dem Aufruf der entsprechenden Operation sofort zum Aufrufer zurückkehrt. Im Detail wird ein Methode auf einer `Proxy`-Komponente aufgerufen, die die entsprechende Schnittstelle anbietet. In Folge dessen wird eine Methodenanfrage ausgelöst, die von der `Proxy`-Komponente an eine `Scheduler`-Komponente übergeben wird. Die `Scheduler`-Komponente, die in einem eigenen Thread läuft, fügt anstehende Methodenanfragen in eine Warteschlange ein und bestimmt welche der anstehenden Methodenanfragen ausgeführt werden können. Eine ablaufbereite Methodenanfrage wird an die `Servant`-Komponente, die die gewünschte Methode implementiert, übergeben und wird von dieser ausgeführt.
Dieses Entwurfsmuster dient der Koordination und Optimierung von Komponenteninteraktion, indem der nebenläufiger Zugriff auf eine Komponente nicht den gesamten Prozess für unbestimmte Zeit blockiert und dadurch die verfügbare Parallelität der Plattform ausnutzt. Oftmals werden komponentenbasierte Anwendungen auf Basis eines Anwendungs-Servers realisiert. Dieser übernimmt die Aufgaben, auf die das *Active Object*-Muster ausgerichtet ist. Als Beispiel kann hier der Zugriff auf mehrere TCP-Verbindungen genannt werden. Für die komponentenbasierte Anwendung selbst erfolgt dies transparent, sodass dieses Entwurfsmuster in die Kategorie Infrastruktur eingeordnet wird.
Die Auswahlstrategie des Schedulers und die Rendezvous- und Rückgabestrategie kann sich auf die Performance auswirken, da dadurch effiziente Aufrufreihenfolgen möglich sind bzw. bestimmt wird, wie Dienstnehmer die Rückgabewerte erhalten. Bei letzterem sind beispielsweise die Strategien „synchrones Warten“ oder „asynchroner Aufruf“ möglich. Der Einsatz von *Timed Method Invocations* [SRSS+00, S. 392] ermöglicht es, dass das Einfügen und Entfernen von Methodenanforderungen in die bzw. aus der Activation List mit einem Timeout versehen wird. Davon können viele Anwendungen profitieren.

4.2.3.2 Monitor Object

Das Entwurfsmuster *Monitor Object* [SRSS+00, S. 399-422] ermöglicht es, dass von nebenläufigen Methodenausführungen einer Komponente nur eine aktiv ist, um so den Zugriff auf Komponenten zu synchronisieren und dadurch eine korrekte Ausführung der nebenläufigen Anwendung zu erreichen. Dazu wird jede Komponente, auf die nebenläufig zugegriffen wird, als `Monitor Object` definiert, worauf nur über dessen synchronisierten Methoden zugegriffen werden kann. Dabei kommt das Entwurfsmuster *Thread-Safe Interface* (siehe Abschnitt 4.2.2.3) zur Anwendung. Zur Vermeidung von Race Conditions kann nur eine der synchronisierten Methoden ausgeführt werden. Dazu enthält jedes `Monitor Object` ein `Monitor Lock`. Des Weiteren ist es möglich, dass mehrere synchronisierten Methoden, die in unterschiedlichen Threads laufen, ihre Ablaufreihenfolge kooperativ planen, indem sie auf Monitorbedingungen warten und sich über Monitorbedingungen benachrichtigen. Die Monitorbedingungen sind mit dem entsprechenden Monitorobjekt verknüpft.
Bei Java kann jedes Objekt ein Monitorobjekt sein, das eine Monitorsperre und eine einzelne Monitorbedingung enthält. Das Wurzelobjekt aller Java-Objekte ist `java.lang.Object` und beinhaltet die Methoden `wait()`, `notify()` und `notifyAll()`, die jedoch nur aufgerufen werden dürfen, wenn der Aufrufer der Besitzer des Monitors des Objekts ist. Das *Monitor Object*-Muster kann somit vom Komponentenentwickler für die Entwicklung von Basiskomponenten verwendet werden, um den Zugriff auf die Komponente zu synchronisieren und zu koordinieren. Dazu muss explizit das `synchronized`-Konstrukt zur Deklaration der synchronisierten Methoden verwendet werden. Mittels der `wait()`, `notify()` und `notifyAll()` wird der Zugriff auf synchronisierten Methoden explizit koordiniert. Das *Monitor Object*-Muster ein Entwurfsmuster der Kategorie Thread-Sicherheit bei Komponenten, da auf Ebene der Basiskomponenten erreicht wird, dass der Ausführungszustand einer Komponente bei nebenläufigem Zugriff nicht korrumpiert wird. Trotz der Synchronisation der Komponentenzugriffe wird dieses Entwurfsmuster nicht als Synchronisationsmuster, sondern als Nebenläufigkeitsmuster gruppiert, da es ermöglicht, dass die Komponentenmethoden ihre Ablaufreihenfolge kooperativ planen.
Die Wahl der Sperre und der Monitorbedingung kann Auswirkungen auf die Performance haben.

4.2.3.3 Half-Sync/Half-Async

Nebenläufige Systeme bestehen oft aus einer Kombination aus asynchronen und synchronen Diensten. Für Anwendungsentwickler soll die Komplexität der asynchronen Ausführung verborgen bleiben, für Systementwickler ist jedoch die Maximierung der Performance maßgebend, sodass die synchrone Ausführung für sie nicht von Bedeutung sein soll. Die Software-Architektur soll es dennoch ermöglichen, dass die synchronen und asynchronen Dienste miteinander kommunizieren, ohne dass die Komplexität steigt oder sich die Performance übermäßig verschlechtert. Das Entwurfsmuster *Half-Sync/Half-Async* [SRSS+00, S. 423-446] trennt eine Anwendung in eine synchrone Schicht und eine asynchrone Schicht. In der synchronen Schicht werden anwendungsnahe Dienste ausgeführt, in der synchronen Schicht systemnahe Dienste. Beide Schichten kommunizieren über eine Warteschlangenschicht. Dadurch ist es möglich die Abarbeitung asynchroner Dienste von der Abarbeitung synchroner Dienste zu entkoppeln.
Die Funktionalität dieses Entwurfsmusters wird bei der betrachteten Infrastruktur

aus Java und einem Java EE-konformen Anwendungs-Server vom Anwendungs-Server erbracht. Die asynchronen systemnahen Dienste sind für komponentenbasierte Anwendungen transparent. Die Anwendungen nutzen meist die synchrone Ausführung, um die Komplexität der asynchronen Ausführung zu vermeiden. Somit wird *Half-Sync/Half-Async* in die Kategorie Infrastruktur eingeordnet.
Für die Warteschlangenschicht können Pufferstrategien verwendet werden, die auf das Einsatzszenario ausgerichtet sind. So könnte sich beispielsweise eine prioritätsabhängige Pufferreihenfolge als effizienter erweisen als eine einfache FIFO-Strategie. Werden für die synchronen Dienste asynchrone Nachrichten für den Kontrollfluss verwendet, können diese Dienste asynchron über die in der Queue zur Verfügung stehenden Nachrichten benachrichtigt werden. Die darauffolgenden Datenoperationen werden synchron ausgeführt. Diese so genannte Variante *Asynchronous Control with Synchronisation Data I/O* [SRSS+00, S. 438] ermöglicht aufgrund der asynchronen Benachrichtigung eine höhere Reaktionsfähigkeit der synchronen Dienste. Beim der Variante *Half-Async/Half-Async* [SRSS+00, S.438] werden zusätzlich zur asynchronen Benachrichtigung auch asynchrone Ein-/Ausgabe-Operationen eingesetzt, sodass auf höherer Ebene von der Effizienz der asynchronen Mechanismen auf niedriger Ebene profitiert werden kann.

4.2.3.4 Leader/Followers

Bei hochleistungsfähiger nebenläufiger Software, die mehrere Ereignisse nebenläufig verarbeiten kann, ist ein effizientes Demultiplexen von Ereignissen ein wichtiger Faktor. Für die Maximierung der Performance muss der durch die Nebenläufigkeit bedingte Aufwand, wie z.B. für Kontextwechsel oder die Synchronisation von Threads, minimiert werden. Des Weiteren ist die Vermeidung von Race Conditions essentiell. Das Entwurfsmuster *Leader/Followers* [SRSS+00, S. 447-474] bietet für die genannten Probleme die folgende Lösung. Mehrere Threads wechseln sich ab, um Ereignisse zu bearbeiten. Dabei existiert maximal ein aktiver Leader-Thread, der die Möglichkeit hat, auf Ereignisse zu reagieren. Die restlichen Threads, die keine Ereignisse bearbeiten, sind in wartendem Zustand, bis sie jeweils zum `Leader`-Thread werden. Nachdem der `Leader`-Thread ein neues Ereignis empfangen hat, wählt dieser einen der `Follower`-Threads als neuen `Leader`-Thread aus. Anschließend veranlasst er die Bearbeitung des Ereignisses. Nach Abschluss der Ereignisbearbeitung wird er wieder in den wartenden Zustand versetzt.
Dieses Entwurfsmuster koordiniert die Interaktion mehrerer nebenläufiger Komponenten und zielt auf Maximierung der Performance, Vermeidung von Laufzeiteffekten und effiziente Demultiplexverknüpfungen zwischen nebenläufigen Komponenten und Ereignisquellen. Für nebenläufige, komponentenbasierte Anwendungen, die auf Java EE-konformen Anwendungs-Servern basieren, übernimmt der Anwendungs-Server die Funktionalität des *Leader/Followers*-Musters. Aus diesem Grund wird es als Infrastrukturmuster klassifiziert.
Mittels des *Follower Promotion Protocols* [SRSS+00, S. 463-465] bestimmt der aktuelle `Leader`-Thread einen `Follower`-Thread, welcher zum neuen `Leader`-Thread werden soll. Die Auswahlstrategie dieses Protokolls stellt einen Performance-relevanten Aspekt dar, da z.B. bei bestimmten Szenarien eine prioritätsabhängige Reihenfolge effizienter ist als eine LIFO-Strategie. Bei der Variante *Bound Handle/Thread Associations* [SRSS+00, S. 467] wird jeder Thread einem eigenen Handle zugeordnet, um bestimmte Ereignisse zu bearbeiten. Mehrere Threads können sich Ressourcen

teilen, da mittels zusätzlicher Zustandsvariablen der Threads bei eingehenden Ereignissen das entsprechende Handle identifiziert werden. Der zugehörige Thread kann darauf hin die Ereignisverarbeitung durchführen, unabhängig davon ob es sich dabei um den `Leader`-Thread oder einen der `Follower`-Threads handelt. Dies ist möglich da diese Variante einen Mechanismus zur Ereignisübergabe vorsieht. Diese Möglichkeit der Ressourcenteilung erhöht die Skalierbarkeit und damit auch hoher Last die Performance. Bei einer weiteren Variante werden die Serialisierungsbeschränkungen gelockert [SRSS+00, S. 468], um so die Parallelität auf Hardware-Ebene ausnutzen zu können. Dies geschieht dadurch, dass mehrere `Leader`-Threads gleichzeitig aktiv sein können.

4.2.3.5 Thread-Specific Storage

Nebenläufige Software bietet aufgrund von Sperr-Overhead oftmals keine Verbesserungen der Performance gegenüber sequenzieller Software. Erfolgt der atomare und global einheitliche Zugriff auf Variablen, von denen jeder beteiligte Thread seine eigene Kopie der Variablenwerte hat, ohne dass bei jedem Zugriff ein Sperr-Overhead in Kauf genommen werden muss, reduziert sich der Overhead in Zusammenhang mit Sperren. Dadurch sind Verbesserungen der Performance zu erreichen. Das Entwurfsmuster *Thread-Specific Storage* [SRSS+00, S. 475-504] ermöglicht es Threads, über eine `Proxy`-Komponente global einheitlich auf ein lokales Objekt zuzugreifen. Die `Proxy`-Komponente verwendet Identifikatoren zur Referenzierung der Thread-spezifischen Objekte jedes Threads, um anschließend auf das benötigte Thread-spezifische Objekt zuzugreifen.
In Java wird dieses Muster von der Klasse `java.lang.ThreadLocal` implementiert und kann bei der Entwicklung von Basiskomponenten vom Komponentenentwickler verwendet werden. Das Entwurfsmuster *Thread-Specific Storage* wird somit zur Kategorie Thread-Sicherheit bei Komponenten zugeordnet.

4.2.3.6 Replikation

Viele verteilte Software-Systemen müssen hohen Dienstgüteanforderungen gerecht werden. Replikation stellt eine Möglichkeit dar, wie die Performance, Verfügbarkeit und Fehlertoleranz verbessert werden [CoDK05] kann. Replikation bedeutet, dass mehrere Datenkopien auf mehreren Rechnern geführt werden. Dafür ist es erforderlich, dass bei Datenänderungen alle Kopien konsistent geändert werden müssen. Im Rahmen dieser Arbeit wird die aktive Replikation [CoDK05, S. 620-622] betrachtet. Für Client-Komponenten, die den Dienst der Replikation in Anspruch nehmen, soll der Ablauf der Replikation transparent sein. Dafür werden die Komponenten `Front-End Manager` und `Replication Manager` eingeführt. Der `Front-End Manager` nimmt Replikationsanfrage einer Client-Komponente entgegen und leitet diese per Gruppenkommunikation an alle `Replication Manager` weiter. Jeder `Replication Manager` führt die Operationen gemäß der Replikationsanfragen auf seiner Datenkopie aus. Alle `Replication Manager` senden für jede Replikationsanfrage eine Antwort an den `Front-End Manager`. Dieser übermittelt entsprechend seines Fehlermodells eine kumulative Antwort an die Client-Komponente.
Bei komponentenbasierten, nebenläufigen Anwendungen, die einen Java EE-konformen Anwendungs-Server verwenden, kann Replikation beim Anwendungs-Server selbst eingesetzt werden. Dabei sind Verbesserungen bei der Performance messbar

[SiPH06]. Für die Anwendung ist die Replikation im Anwendungs-Server transparent. Das Replikationsmuster lässt sich somit in die Kategorie Infrastruktur einordnen.
Auf die Dienstgüte haben verschiedene Parameter Einfluss. Neben der Anzahl der Kopien ist auch die Variante der Gruppenkommunikation von Bedeutung, da beispielsweise zuverlässige Gruppenkommunikation einen höheren Kommunikationsaufwand erfordert als einfache Gruppenkommunikation. Die Reihenfolge der Replikationsanfragen kann sich ebenfalls auf die Performance auswirken, da dadurch bestimmt wird, in welcher Reihenfolge die Replikationsanfragen bearbeitet werden. Bedeutend ist weiterhin das Fehlermodell des `Front-End Manager`. Sollen beispielsweise einzelne Abstürze toleriert werden, kann der `Front-End Manager` nach dem Eintreffen einer erfolgreichen Antwort von einem `Replication Manager` eine Antwort an die Client-Komponente senden, ohne weiter Antworten von den `Replication Managern` abwarten zu müssen.

4.2.3.7 Thread Pool

Nebenläufige Software soll die Möglichkeit bieten, mehrere Anfragen parallel zu bearbeiten. Ein naheliegender Ansatz für dieses Problem wäre die Bearbeitung jeder Anfrage in einem eigenen Thread. Dies hätte jedoch zur Folge, dass viel Zeit und Systemressourcen für die Erzeugung und Zerstörung der Threads aufgewendet werden müsste. Das Entwurfsmuster *Thread Pool* [PeSo97, S. 8-9] stellt eine effiziente Lösung für das Problem der parallelen Bearbeitung von Anfragen dar, indem mehrere Threads vorab erzeugt und in einem Thread-Pool gruppiert werden. Wartende Threads des Thread-Pools können Anfragen bearbeiten und stehen nach Beendigung der Bearbeitung wieder im Thread-Pool für weitere Anfragen bereit. Da bei der Ankunft einer Anfrage die Threads bereits existieren, kann die Bearbeitung sofort starten, sodass sich die Reaktionsfähigkeit verbessert.
Für komponentenbasierte nebenläufige Anwendungen, die einen Anwendungs-Server als Laufzeitumgebung nutzen, nimmt der Anwendungs-Server Anfragen entgegen und initiiert deren Verarbeitung. Für die Anwendungen erfolgt dies transparent. Aufgrund der Performance-Vorteile dieses Musters, wird es bei den meisten Implementierungen von Anwendungs-Servern eingesetzt. Somit lässt sich Das Muster *Thread Pool* als Infrastrukturmuster klassifizieren.
Die Anzahl der Threads eines Thread-Pools kann sich auf die Performance auswirken. Ist die Anzahl gering, ist es möglich, dass sich keine Threads des Thread-Pools in wartendem Zustand befinden und ankommende Anfragen warten müssen. Andererseits führt die Erzeugung vieler Threads zu hohem Speicherverbrauch.

4.2.4 Entwurfsmuster für nachrichtenorientierte Kommunikation

Nachrichtenorientierte Kommunikation stellt eine Kommunikationsform dar, bei der die Kommunikationsteilnehmer lose gekoppelt sind. Für den Austausch von Nachrichten zwischen Kommunikationspartnern ist ein Nachrichtensystem verantwortlich. Die Entwurfsmustergruppen „Nachrichtenkanäle“, „Nachrichten-Routing“ und „Nachrichtenendpunkte“ aus [HoWo03] bieten Lösungen für Probleme in Zusammenhang mit nachrichtenorientierter Kommunikation und werden im Folgenden den in Abschnitt 4.1 definierten Kategorien zugeordnet.

4.2.4.1 Nachrichtenkanäle

Bei nachrichtenorientierter Kommunikation stellt sich die Frage, wie die Nachrichten zwischen den beteiligten Komponenten übertragen werden. Der Versand von Nachrichten erfolgt über einen virtuellen Kanal zwischen dem Sender und Empfänger, dem so genannten Nachrichtenkanal. Nachrichtenkanäle können je nach Anforderungen in verschiedenen Varianten eingesetzt werden. Beispiele sind der *Point-to-Point Channel* oder der *Publish-Subscribe Channel* [HoWo03].
Hohpe und Woolf fassen in der Gruppe Nachrichtenkanäle [HoWo03, S. 99-141] eine Menge von Entwurfsmustern zusammen. Sie umfasst die Muster *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Datatype Channel*, *Invalid Message Channel*, *Dead Letter Channel*, *Guaranteed Delivery*, *Channel Adapter*, *Message Bridge* und *Message Bus*. Diese Menge von Entwurfsmustern wird in die Kategorie Komponenteninteraktion eingeordnet, da der Software-Architekt beim Entwurf festlegen kann, welche Komponenten durch Austausch von Nachrichten miteinander asynchron kommunizieren. Obwohl die Nachrichtenkanäle letztendlich im Nachrichtensystem implementiert sind und von diesem bereitgestellt werden, wird diese Mustermenge nicht der Kategorie Infrastruktur zugeordnet sondern der Komponenteninteraktion. Dies ist darin begründet, dass die Nachrichtenkanäle abhängig von ihrem Anwendungskontext sind. Das Nachrichtensystem wird zudem entsprechend der spezifizierten nachrichtenorientierten Komponentenkommunikation konfiguriert. Erst anhand der Konfiguration werden vom Nachrichtensystem die benötigten Nachrichtenkanäle erzeugt und der Anwendung zur Verfügung gestellt.

4.2.4.2 Nachrichten-Routing

Die Kommunikation zwischen Komponenten kann auch stattfinden, ohne dass es einen direkten Nachrichtenkanal zwischen dem Sender und Empfänger geben muss, sondern die Nachrichten mehrere Kanäle passieren bevor sie den Empfänger erreichen. Der Nachrichtenversand erfordert in diesem Fall, dass ein oder mehrere Nachrichten-Router die Nachrichten an den jeweils nächsten Nachrichtenkanal bis zum Empfänger weiterleiten. Für Nachrichten-Router existieren einige Varianten, die auf spezielle Probleme beim Weiterleiten von Nachrichten ausgerichtet sind, z.B. *Content-Based Router*, *Splitter* oder *Process Manager*.
Zu der Musterfamilie Nachrichten-Routing [HoWo03, S. 225-326] gehören nach Hohpe und Woolf die Entwurfsmuster *Content-Based Router*, *Message Filter*, *Dynamic Router*, *Recipient List*, *Splitter*, *Aggregator*, *Resequencer*, *Composed Message Processor*, *Scatter-Gather*, *Routing Slip*, *Process Manager* und *Message Broker*. Diese Entwurfsmusterfamilie wird in die Kategorie Komponenteninteraktion eingeordnet, da die einzelnen Muster für die Koordination der asynchronen Kommunikation zwischen Komponenten bedeutend. Zudem sind sie auf einen bestimmten Anwendungskontext ausgerichtet. Die Nachrichten-Router werden beim Software-Entwurf vom Software-Architekten spezifiziert und vom Komponentenentwickler implementiert. Aus Sicht des Nachrichtensystems ist ein Nachrichten-Router lediglich eine Komponente, die Nachrichten empfängt bzw. versendet. Die Routing-Funktionalität ist für das Nachrichtensystem transparent, da diese die interne Funktionalität des Nachrichten-Routers darstellt. Somit kann die Familie Nachrichten-Routing nicht der Infrastruktur zugeordnet werden.
Je nach Variante können verschiedene Parameter für die Performance relevant sein.

Beispielsweise wirkt sich die Nachrichtenlänge auf die Performance des *Content-Based Routers* aus.

4.2.4.3 Nachrichtenendpunkte

Komponenten können nicht ohne weiteres auf das Nachrichtensystem zugreifen, um die nachrichtenorientierte Kommunikationsform nutzen zu können. Sie müssen vielmehr mittels eines Nachrichtenendpunkts an das Nachrichtensystem angebunden werden. Hohpe und Woolf fassen in der Mustermenge Nachrichtenendpunkte [HoWo03, S. 463-535] die Entwurfsmuster *Messaging Gateway*, *Messaging Mapper*, *Transactional Client*, *Polling Consumer*, *Event-Driven Consumer*, *Competing Consumers*, *Message Dispatcher*, *Selective Consumer*, *Durable Subscriber*, *Idempotent Receiver* und *Service Activator* zusammen. Bei der Anbindung von Komponenten an das Nachrichtensystem können diverse Probleme auftreten, für die die einzelnen Entwurfsmuster der Mustermenge Lösungen anbieten. Beispielsweise kann die Variante *Selective Consumer* eingesetzt werden, falls eine Komponente nur einen bestimmten Nachrichtentyp empfangen möchte.
Die Art des Nachrichtenendpunkts hängt vom konkreten Verwendungskontext ab, der sich aus der Spezifikation der Interaktion der Komponenten ergibt und wird damit vom Software-Architekten festgelegt. Die Implementierung der benötigten Anwendung wird vom Komponentenentwickler durchgeführt. Bei der Verwendung von *Message-Driven Beans* (MDB) (siehe Abschnitt 3.4.3) wird die Kombination aus den Mustern *Event-Driven Consumer* und *Transactional Client* vom EJB-Container als Nachrichtenendpunkt zur Verfügung gestellt. Eine MDB kann dynamisch als Pool von *Competing Consumers* konfiguriert werden [HoWo03, S. 466]. Die Entwurfsmustermenge *Messaging Endpoint* wird der Kategorie Komponenteninteraktion zugeordnet, da die Verwendung eines Nachrichtenendpunkts beim Entwurf und der Implementierung von Komponenten nicht transparent ist. Die Art der benötigten Nachrichtenendpunkte muss beim Software-Entwurf spezifiziert und die Message-Driven Beans vom Komponentenentwickler entsprechend konfiguriert werden, falls die von *Message-Driven Beans* zur Verfügung gestellte Endpunkt-Funktionalität den Anforderungen genügt. Andernfalls müssen zusätzlich eigene Nachrichtenendpunkte spezifiziert und implementiert werden.

4.3 Zusammenfassung

Die in [SRSS+00], [HoWo03], [Doug04], [PeSo97] und [CoDK05] vorgestellten Entwurfsmuster lassen sich beim Software-Entwurf für die Spezifikation von Basiskomponenten oder des Zusammenspiels von Komponenten einsetzen. Mithilfe der in diesem Kapitel durchgeführten Kategorisierung wird die Auswahl des passenden Entwurfsmuster für den Software-Architekten und den Komponentenentwickler erleichtert. Die Infrastruktur aus Java und einem Java EE-konformen Anwendungs-Server stellt die Funktionalität von vielen der hier betrachteten Entwurfsmuster bereits den Anwendungen transparent zur Verfügung. Somit reduziert sich die Anzahl der für den Software-Architekten und den Komponentenentwickler relevanten Entwurfsmuster erneut.

5. Integration von Mustern ins PCM

Von den in Kapitel 4 vorgestellten Entwurfsmustern wird im ersten Unterkapitel eine Teilmenge für die Integration in das Palladio Komponentenmodell ausgewählt. Die Auswahlkriterien sind die Verwendbarkeit beim Entwurf von nebenläufigen, komponentenbasierten Anwendungen und die Integrierbarkeit ins Palladio Komponentenmodell. Das zweite Unterkapitel beschreibt die Durchführung von Performance-Messungen für die ausgewählten Entwurfsmuster, um die Parameter zu ermitteln, die Einfluss auf die Performance haben. Darauf aufbauend erfolgt im dritten Unterkapitel der Entwurf einer Vervollständigung [WoPS02], die die Erfassung von extra-funktionalen, Performance-relevanten Informationen für die nachrichtenorientierte Kommunikation ermöglicht. Die Vervollständigung bietet die Möglichkeit der Integration dieser Kommunikationsform in das Palladio Komponentenmodell und kann mit den ermittelten Performance-relevanten Parameter konfiguriert werden. Teil des dritten Unterkapitels ist auch die Modellierung der Performance-Auswirkungen dieser Parameter in Form von Schablonen, damit der Performance-Einfluss des Nachrichtensystems für die nachrichtenorientierte Kommunikation flexibel der Konfiguration der Vervollständigung angepasst werden kann. Mithilfe der im vorletzten Unterkapitel vorgestellten Transformation wird diese Vervollständigung zur Übersetzungszeit in die Modellinstanzen eines Entwurfs eingewoben. Abschließend werden die Ergebnisse der Musterintegration noch einmal zusammengefasst.

5.1 Auswahl der Entwurfsmuster

Die Arbeit richtet den Fokus auf den Entwurf von nebenläufiger, komponentenbasierter Software. Nicht alle Entwurfsmuster, die in Kapitel 4 vorgestellt wurden sind hierfür verwendbar. Aus diesem Grund wird für die weiteren Betrachtungen eine Auswahl aus der Menge der Entwurfsmuster getroffen. Neben der Verwendbarkeit ist die Integrierbarkeit der Muster ins PCM ein weiteres Kriterium für die Auswahl, da einige Muster besondere Anforderungen an das PCM stellen. Die Musterauswahl erfolgt in diesem Kapitel anhand der in Abschnitt 4.2 durchgeführten Kategorisierung.

Kategorie Infrastruktur

Alle Entwurfsmuster, die der Kategorie Infrastruktur zuzuordnen sind (siehe Tabelle 4.1), werden im Rahmen dieser Arbeit nicht weiter verfolgt, da sie bei der Entwicklung von Anwendungs-Servern, die die Plattform für komponentenbasierte Software bilden, von Bedeutung sind. Über eine Integration dieser Muster ins PCM kann dessen Ressourcenmodell verfeinert und erweitert werden, was zu einer Verbesserung der Performance-Vorhersage führt. Beim Entwurf von komponentenbasierter Software sind somit diese Muster für Software-Architekten und Komponentenentwickler nur von indirektem Nutzen.

Kategorie Thread-Sicherheit bei Komponenten

Alle Entwurfsmuster der Kategorie Thread-Sicherheit bei Komponenten (siehe Tabelle 4.1) thematisieren den spezifikationskonformen, nebenläufigen Zugriff auf Komponenten. Das Thread-Modell ist zustandsbehaftet, mit jedem Thread sind Kontrollinformationen verknüpft, die den aktuellen Ausführungszustand des Threads beinhalten. Eine Modellierung der Muster dieser Kategorie ist mit dem PCM möglich obwohl das PCM zustandslos ist. Dies kann über Semaphoren erreicht werden, da für die Performance nur entscheidend ist, dass ein gemeinsamer Zustand verändert wird und nicht wie diese Zustandsänderung erfolgt. Dennoch wird diese Kategorie in dieser Arbeit nicht thematisiert, da dies den Rahmen dieser Arbeit übersteigen würde.

Kategorie Komponenteninteraktion

Von den Mustern der Kategorie Komponenteninteraktion (siehe Tabelle 4.1) erweisen sich vor allem die Entwurfsmuster für nachrichtenorientierte Kommunikation als vielversprechend für die Integration in das PCM, da für asynchrone Komponentenkommunikation mittels Nachrichten viele Implementierungen existieren. Zudem findet diese Kommunikationsform breite Anwendung bei der Entwicklung von komponentenbasierter Software. Die Musterfamilien Nachrichtenkanäle und Nachrichtenendpunkte bieten allgemeine Lösungsschemata für Probleme in Zusammenhang mit nachrichtenorientierter Kommunikation und sind von konkreten Anwendungskontexten unabhängig. Eine Integration von Mustern dieser beiden Familien ins PCM verspricht daher eine vereinfachte Modellierung und eine bessere Performance-Vorhersage für ein breites Anwendungsspektrum. Die Entwurfsmuster der Musterfamilie Nachrichten-Routing sind hingegen abhängig vom speziellen Anwendungskontext, sodass keine generelle Verwendbarkeit gegeben ist. Zudem können sie durch Kombinationen der beiden anderen Musterfamilien Nachrichtenkanäle und Nachrichtenendpunkte, sowie spezieller Komponenten, die die Routing-Funktionalität kapseln, ausgedrückt werden (siehe 4.2.4.2). Aus diesem Grund werden diese Routing-Muster nicht ins PCM integriert.

Somit fällt die Wahl für die weiteren Betrachtungen auf die Musterfamilien Nachrichtenkanäle und Nachrichtenendpunkte. Für diese werden zunächst die Performancerelevanten Parameter evaluiert. Die Evaluierungsergebnisse bilden die Grundlage für die Erstellung von Modellkonstrukten. Software-Entwürfe können mittels einer Transformation um diese Modellkonstrukte erweitert werden.

5.2 Evaluierung der Performance-relevanten Parameter

Im Kontext der nachrichtenorientierten Kommunikation ist unter Performance vor allem die Übertragungsdauer einer Nachricht zu verstehen, d. h. die Zeitdauer zwischen dem Start des Versands einer Nachricht und deren Empfang. Für die Integration von Entwurfsmustern der beiden Musterfamilien Nachrichtenkanäle und Nachrichtenendpunkte in das PCM ist es entscheidend, dass diese einen Einfluss auf die Performance haben. Die diversen Muster weisen auf eine große Konfigurationsvielfalt bei nachrichtenorientierter Kommunikation hin, jedoch lassen sich nur Vermutungen und Schätzungen anstellen, ob und in welchem Maße ein Entwurfsmuster Auswirkungen auf die Performance hat. Aus diesem Grund wird eine Leistungsbewertung von Konfigurationsoptionen des *Java Message Service* (JMS) (siehe Abschnitt 3.5.1, [Java02]) durchgeführt. Viele dieser Optionen stellen Repräsentationen der Entwurfsmuster dar, so dass sich von der Performance-Relevanz der Optionen auf die der Entwurfsmuster schließen lässt.

Zunächst wird daher eine Zuordnung von Entwurfsmustern zu Konfigurationsoptionen der JMS-API vorgenommen. Darauf folgend werden der Entwurf und einige Implementierungsdetails der Testanwendung beschrieben, welche für die Leistungsbewertung der JMS-Optionen verwendet wird. Abschließend erfolgt die Diskussion der gemessenen Übertragungsdauer bei unterschiedlichen JMS-Optionen.

5.2.1 Entwurfsmuster und JMS-Optionen

Für die Ermittlung der Performance-Relevanz der Entwurfsmuster der Familien Nachrichtenkanäle und Nachrichtenendpunkte werden Performance-Messungen mithilfe einer Testanwendung durchgeführt, die auf einer JMS-Implementierung basiert. Um von den Messergebnissen der unterschiedlichen JMS-Konfigurationen auf die Performance-Relevanz von Entwurfsmustern schließen zu können, wird zunächst eine Zuordnung von Entwurfsmustern zu JMS-Optionen vorgenommen. Diese Zuordnung ist in Tabelle 5.1 zusammengefasst. Wie im vorherigen Kapitel werden zur Vermeidung von Missverständnissen die englischen Namen der Entwurfsmuster verwendet und durch kursive Schriftform hervorgehoben. Dasselbe gilt für die Bezeichnung der JMS-Optionen, die an die Terminologie der JMS-Spezifikation [Java02] und der JMS-Beispiele in [HoWo03] angelehnt ist.

Musterfamilie Nachrichtenkanäle

Zur Musterfamilie Nachrichtenkanäle gehören die Entwurfsmuster *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Datatype Channel*, *Invalid Message Channel*, *Dead Letter Channel*, *Guaranteed Delivery*, *Channel Adapter*, *Messaging Bridge* und *Message Bus*. Jedes dieser Muster wird zunächst kurz beschrieben und anschließend die getroffene Zuordnung zu JMS-Optionen gemäß Tabelle 5.1 erklärt.

Point-to-Point Channel Bei nachrichtenorientierter Kommunikation ist es oftmals erforderlich, dass eine Nachricht von genau einem Empfänger verarbeitet wird. Hierfür eignet sich das Entwurfsmuster *Point-to-Point Channel* [HoWo03, S. 103-105], da es einen Nachrichtenkanal spezifiziert Nachrichtenkanal, dessen Nachrichten nur von einem einzigen Empfänger entgegengenommen werden. Dies ist von

	Entwurfsmuster	JMS-Option
Nachrichtenkanäle	*Point-to-Point Channel*	*JMS Queue*
	Publish-Subscribe Channel	*JMS Topic*
	Datatype Channel	-
	Invalid Message Channel	-
	Dead Letter Channel	-
	Guaranteed Delivery	*JMS Persistent Messages*
	Channel Adapter	-
	Messaging Bridge	-
	Message Bus	-
Nachrichtenendpunkte	*Messaging Gateway*	-
	Messaging Mapper	-
	Transactional Client	*JMS Transacted Session*
	Polling Consumer	*JMS Receive*
	Event-Driven Consumer	*JMS Message Listener*
	Competing Consumers	*MDB-Pool*
	Message Dispatcher	-
	Selective Consumer	*JMS Message Selector*
	Durable Subscriber	*JMS Durable Subscription*
	Idempotent Receiver	*JMS Message Acknowledgment*
	Service Activator	-

Tabelle 5.1: Zuordnung von Entwurfsmuster zu JMS-Optionen

Bedeutung da es potenziell möglich ist, dass mehrere Empfänger feststellen, dass eine Nachricht in einem Nachrichtenkanal vorhanden ist und die Verarbeitung dieser Nachricht starten.
JMS bietet mit dem Nachrichtenziel `Queue` eine direkte Repräsentation des Musters *Point-to-Point Channel*. Die Eigenschaften der Option *JMS-Queue* entsprechen damit jenen des Entwurfsmusters, so dass sich beide einander zuordnen lassen.

Publish-Subscribe Channel Der Versand von Nachrichten an eine Menge von Empfängern wird von dem Entwurfsmuster *Publish-Subscribe Channel* [HoWo03, S. 106-110] thematisiert. Dafür existieren bereits etablierte Muster wie Beobachter [GHJV95] oder *Publisher-Subscriber* [BMRS+96], wobei das letztere auf ersterem basiert. Beide Muster ermöglichen, dass kooperierende Komponenten sich in synchronisiertem Zustand befinden, indem eine Komponente eine Menge von Beobachterkomponenten über seine Zustandsänderungen benachrichtigt. Das Entwurfsmuster *Publish-Subscribe Channel* überträgt das Prinzip dieser Muster auf nachrichtenorientierte Kommunikation und funktioniert folgendermaßen: Eine Senderkomponente sendet Nachrichten über einen Nachrichtenkanal, der in mehrere Ausgabekanäle aufgesplittet wird, einen für jeden teilnehmenden Empfänger. Dieser besondere Nachrichtenkanal wird als *Publish-Subscribe Channel* bezeichnet. Für jede Nachricht, die über einen solchen Nachrichtenkanal übertragen wird, schickt der Nachrichtenkanal eine Kopie der Nachricht an alle Ausgabekanäle. Jeder Empfänger erhält so über den ihm eigenen Ausgabekanal die Nachricht, die er nur einmal empfangen kann. Somit wird über den *Publish-Subscribe Channel* jede Nachricht zu jedem teilnehmenden

Empfänger genau einmal übertragen.
Neben `Queue` ist `Topic` ein weiteres Nachrichtenziel, das von JMS angeboten wird. Die Eigenschaften von `Topic` sind identisch mit den Eigenschaften das Musters *Publish-Subscribe Channel*. Somit kann dieses Entwurfsmuster der Option *JMS-Topic* zugeordnet werden.

Datatype Channel Das Entwurfsmuster *Datatype Channel* [HoWo03, S. 111-114] adressiert das Problem, dass Nachrichten mit unterschiedlichen Datentypen in einer Weise übertragen werden, dass der Empfänger in der Lage ist, die Nachrichten zu verarbeiten. Der Lösungsvorschlag dieses Musters verwendet getrennte Nachrichtenkanäle für jeden Datentyp, so dass über jeden dieser Nachrichtenkanäle ausschließlich Nachrichten mit gleichem Datentyp gesendet werden.
Dieses Entwurfsmuster ist sehr anwendungsspezifisch, da die verwendeten je nach Anwendung die Datentypen unterschiedlich sein können. Aus diesem Grund ist in JMS auch keine Repräsentation dieses Musters vorhanden, es müsste vielmehr bei Bedarf explizit eingesetzt werden.

Invalid Message Channel Bei nachrichtenorientierter Kommunikation kann das Problem auftreten, dass ein Empfänger eine empfange Nachricht nicht verarbeiten kann. Beispielsweise könnten benötigte Nachrichteneigenschaften unvollständig oder eine korrekte Nachricht über den falschen Kanal gesendet worden sein. Das Muster *Invalid Message Channel* [HoWo03, S. 111-118] stellt für solche Fälle eine geeignete Möglichkeit zur Fehlerbehandlung dar, indem alle Nachrichten, die ein Empfänger nicht verarbeiten kann, an einen speziellen Nachrichtenkanal, dem *Invalid Message Channel*, weitergeleitet werden.
Die Empfehlung der JMS-Spezifikation 1.1 zu dieser Problematik lautet folgendermaßen [Java02, S.69]:

> It is possible for a listener to throw a RuntimeException; however, this is considered a client programming error. Well behaved listeners should catch such exceptions and attempt to divert messages causing them to some form of application-specific 'unprocessable message' destination.

Die Fehlerbehandlung soll demnach in der Applikation selbst vorgenommen werden. JMS bietet selbst keine Fehlerbehandlungsmechanismen, für den Fall, dass ein Empfänger eine Nachricht nicht verarbeiten kann. Somit bietet JMS keine Repräsentation des Entwurfsmusters *Invalid Message Channel*.

Dead Letter Channel Es gibt eine Reihe von Gründen, warum das Nachrichtensystem nicht in der Lage ist, eine Nachricht an einen Empfänger zu senden. Zum Beispiel könnte die Lebensdauer einer Nachricht abgelaufen sein oder alle selektiven Empfänger ignorieren eine Nachricht aufgrund deren Selektionskriterien. Es stellt sich die Frage, wie das Nachrichtensystem mit solchen Nachrichten verfahren soll. Das Entwurfsmuster *Dead Letter Channel* [HoWo03, S. 119-121] bietet folgende Lösungsmöglichkeit: Stellt das Nachrichtensystem fest, dass eine Nachricht nicht einem Empfänger zugestellt werden kann, wird diese an einen eigens dafür vorgesehenen Nachrichtenkanal, der als *Dead Letter Channel* bezeichnet wird, weitergeleitet.

Die JMS-Spezifikation lässt den Implementierungen Freiraum wie mit Fehlern bei der Nachrichtenzustellung umgegangen wird. Für den Fall, dass die Lebensdauer einer Nachricht abgelaufen ist, empfiehlt die Spezifikation das Löschen der Nachricht. Einige JMS-Implementierungen bieten jedoch vielfältige Möglichkeiten, wie Fehler bei der Nachrichtenzustellung behandelt werden sollen. Der BEA WebLogic Server 10.0 lässt sich beispielsweise so konfigurieren [BeaA], dass das Muster *Dead Letter Channel* verwendet wird. Da die JMS-Spezifikation jedoch keine explizite Repräsentation dieses Musters beinhaltet, wird für dieses keine Zuordnung getroffen.

Guaranteed Delivery Das Entwurfsmuster *Guaranteed Delivery* [HoWo03, S. 122-126] sieht vor, dass Nachrichten persistiert werden, damit sie selbst beim Absturz des Nachrichtensystems nicht verloren gehen. Dadurch kann jeder Sender sicher sein, das alle Nachrichten an den Empfänger ausgeliefert werden.
JMS bietet die Möglichkeit, dass für jede Nachricht beim Versand bestimmt werden kann, ob sie persistiert werden soll. Hieraus resultiert ein Unterschied zum Entwurfsmuster *Guaranteed Delivery*, das nicht festlegt, ob Persistenz auf einzelne Nachrichten oder auf Nachrichtenkanäle angewandt werden soll. Der Unterschied ist jedoch marginal, da die Funktionalität der Persistierung von Nachrichten identisch ist. Die nachrichtenspezifische Persistenzmöglichkeit von JMS kann als Anwendung des Entwurfsmusters betrachtet werden, dessen Generalität aus der allgemeinen Anwendbarkeit von Mustern stammt. Insgesamt lässt sich das Entwurfsmuster *Guaranteed Delivery* den *JMS Persistent Messages* zuordnen.

Channel Adapter Das Problem der Anbindung einer Anwendung an ein Nachrichtensystem wird vom Entwurfsmuster Channel Adapter [HoWo03, S. 127-132] behandelt. Als Lösungsansatz sieht das Muster einen Channel Adapter vor, der die Programmierschnittstelle einer Anwendung ansprechen oder auf Anwendungsdaten zugreifen kann, um Daten per Nachrichten verschicken bzw. nach dem Empfang von Nachrichten Anwendungsfunktionalität aufrufen zu können.
Das Muster weist einen sehr starken Anwendungsbezug auf, sodass es keiner JMS-Option zugeordnet werden kann, da diese hauptsächlich das Nachrichtensystem selbst beeinflussen.

Messaging Bridge Die Verknüpfung von mehreren Nachrichtensystemen thematisiert das Entwurfsmuster *Messaging Bridge* [HoWo03, S. 133-136]. Der Lösungsvorschlag des Musters beinhaltet eine Nachrichtenbrücke als Verbindung zwischen unterschiedlichen Nachrichtensystemen, die die Nachrichten zwischen den Nachrichtensystemen repliziert. Die Nachrichtenbrücke kann als Menge von Adaptern für Nachrichtenkanäle betrachtet werden, bei dem die Anwendung, die an ein Nachrichtensystem angebunden wird, selbst wieder ein Nachrichtensystem ist. Dabei wird jedes Paar von zusammengehörigen Nachrichtenkanälen zweier Nachrichtensysteme durch ein Adapterpaar verbunden. Die Nachrichtenbrücke bildet demnach eine Menge von Nachrichtenkanälen auf eine andere ab und transformiert eingehende Nachrichten in das Format des anderen Nachrichtensystems.
In der JMS-Spezifikation ist keine Funktionalität festgelegt, die dem Muster *Messaging Bridge* entspricht. Dennoch bieten einige JMS-Implementierungen diese Funktionalität, beispielsweise der BEA WebLogic Server 10.0 [BeaM]. Das Entwurfsmus-

ter *Messaging Bridge* wird keiner JMS-Option zugeordnet, da von konkreten JMS-Implementierungen abstrahiert wird und stattdessen die JMS-Spezifikation als Referenz dient.

Messaging Bus Bei der Integration von Anwendungen ist eine Architektur wünschenswert, die es ermöglicht dass separate Anwendungen zusammenarbeiten und diese dennoch von einander entkoppelt sind. Dadurch ist es möglich, solche Anwendungen einfach hinzuzufügen bzw. zu entfernen ohne die restlichen Anwendungen zu beeinflussen. Das Entwurfsmuster *Message Bus* [HoWo03, S. 137-141] strukturiert die Middleware, die die einzelnen Anwendungen verbindet, als Nachrichtenbus. Dieser ermöglicht es den Anwendungen, dass sie mittels nachrichtenorientierter Kommunikation kooperieren. Dieses Muster kombiniert verschiedene andere Muster für nachrichtenorientierte Kommunikation, wie z. B. *Publish-Subscribe Channel*, *Channel Adapter* oder *Datatype Channel*.
Das Muster *Messaging Bus* kann keiner JMS-Option zugeordnet werden, da es ein Architekturmuster darstellt und von Details des Nachrichtensystems abstrahiert. Es basiert auf mehreren Mustern, die keine JMS-Repräsentation besitzen, unter anderem weil diese einen starken Anwendungsbezug aufweisen. Schon aus diesem Grund kann die Kombination der Muster nicht in JMS zur Verfügung gestellt werden.

Musterfamilie Nachrichtenendpunkte

Zur Musterfamilie Nachrichtenendpunkte gehören die Entwurfsmuster *Messaging Gateway*, *Messaging Mapper*, *Transactional Client*, *Polling Consumer*, *Event-Driven Consumer*, *Competing Consumers*, *Message Dispatcher*, *Selective Consumer*, *Durable Subscriber*, *Idempotent Receiver* und *Service Activator*. Nach einer kurzen Beschreibung jedes dieser Muster folgt eine Erläuterung der getroffene Zuordnung zu JMS-Optionen gemäß Tabelle 5.1.

Messaging Gateway Das Entwurfsmuster *Messaging Gateway* [HoWo03, S. 468-476] sieht einen Gateway vor, der die für die nachrichtenorientierte Kommunikation spezifischen Methodenaufrufe kapselt und einer Anwendung domänenspezifische Methoden zur Verfügung stellt. Dadurch können direkte Abhängigkeiten zwischen einer Anwendung und dem verwendeten Nachrichtensystem aufgelöst werden. Es wird eine höhere Flexibilität erreicht als ohne *Messaging Gateway*, da lediglich die Gateway-Implementierung ausgetauscht werden muss, falls eine andere Integrationstechnologie eingesetzt werden soll. Zudem folgt dieser Ansatz dem Gedanken der Trennung von Verantwortlichkeiten, da die Anwendungslogik vom kommunikationsspezifischen Code getrennt wird.
Dieses Entwurfsmuster weist einen hohen Anwendungsbezug auf. Von Details für die Nachrichtenorientierte Kommunikation wird weitgehend abstrahiert, so dass keine JMS-Option diesem Muster zugeordnet werden kann.

Messaging Mapper Die Daten, die mithilfe von Nachrichten übertragen werden, sind oftmals von Objekten der Anwendungsdomäne abgeleitet. Das Entwurfsmuster *Messaging Mapper* [HoWo03, S. 477-483] adressiert das Problem der Abbildung zwischen Domänenobjekten und dem Nachrichtensystem. Dem Muster entsprechend

wird die Abbildungslogik in einer getrennten *Messaging Mapper*-Komponente so gekapselt, dass weder die Domänenobjekte noch das Nachrichtensystem Kenntnis von der Existenz des *Messaging Mapper* haben. Dadurch werden diese beiden konsequent voneinander getrennt.
Die Komponente *Messaging Mapper* dieses Entwurfsmuster ist im Allgemeinen sehr anwendungsspezifisch und wird von Hohpe und Woolf sogar als Teil der Anwendung angesehen. Daraus leitet sich ab, dass dieses Muster keine Repräsentation in JMS aufweist, sondern bei Bedarf zusätzlich in die Anwendung integriert werden müsste.

Transactional Client Das interne Verhalten von Nachrichtensystemen ist im Allgemeinen transaktional, damit sie ihre Funktionalität überhaupt erbringen können. Ein Nachrichtensystem garantiert, dass eine Nachricht eine Nachricht zu einem Nachrichtenkanal hinzugefügt wird oder nicht bzw. dass eine Nachricht vom Nachrichtenkanal gelesen wurde oder nicht. Dazu verwendet das Nachrichtensystem interne Transaktionen. Diese werden ebenfalls zum Kopieren einer Nachricht vom Sender zum Empfänger verwendet. Für viele Anwendungsszenarien ist dieses transaktionale Verhalten von Nachrichtensystemen nicht ausreichend. Vielmehr benötigen Anwendungen oftmals ein umfassenderes Transaktionsverhalten, das mehr als nur eine einzelne Nachricht einschließt und das die internen Transaktionen des Nachrichtensystems kontrolliert. Beispielsweise sollen mehrere Nachrichten koordiniert oder die nachrichtenorientierte Kommunikation mit anderen Ressourcen abgestimmt werden. Das Entwurfsmuster *Transactional Client* [HoWo03, S. 484-493] ermöglicht hierfür, dass die Teilnahme an nachrichtenorientierter Kommunikation über einen transaktionalen Verbindungskontext erfolgen kann und dass der Teilnehmer die Transaktionsgrenzen selbst festlegen kann.
JMS bietet die Möglichkeit, dass der Kontext der Verbindung mit dem Nachrichtensystem als transaktional spezifiziert werden kann. Die Funktionalität dieser Option entspricht derer des Entwurfsmusters, sodass das Muster *Transactional Client* der Option *JMS Transacted Session* zugeordnet wird.

Polling Consumer Der Empfang von Nachrichten kann auf zwei Arten initiiert werden, aktiv vom Empfänger selbst oder von außen durch das Nachrichtensystem. Das Entwurfsmuster *Polling Consumer* [HoWo03, S. 494-497] bietet einen Ansatz wie ein Empfänger bei Bedarf Nachrichten empfangen kann. Dabei fordert der Empfänger explizit Nachrichten vom Nachrichtensystem an, so dass die Kontrolle über den Nachrichtenempfang bei der Anwendung liegt.
Der Nachrichtenempfang kann in JMS synchron erfolgen. Dafür werden den Empfängern verschiedene `receive`-Methoden zur Verfügung gestellt. Diese JMS-Methoden sind somit die Repräsentation des Entwurfsmusters *Polling Consumer*.

Event-Driven Consumer Eine weitere Möglichkeit des Nachrichtenempfangs ist, dass Nachrichten von Empfängern automatisch nach der Übertragung verarbeitet werden. Im Gegensatz zum *Polling Consumer* liegt hier die Kontrolle des Nachrichtenempfangs beim Nachrichtensystem, das einen Empfänger aktiviert, wenn diesem eine Nachricht zugestellt wurde. Diesen Ansatz beschreibt das Entwurfsmuster *Event-Driven Consumer* [HoWo03, S. 498-501].

In JMS kann eine Empfängerklasse das Interface `Mesasge Listener` implementieren. Die erforderliche `onMessage`-Methode wird vom Nachrichtensystem aufgerufen, sobald eine Nachricht zu diesem Empfänger übertragen wurde. Somit lässt sich die Zuordnung des Entwurfsmusters *Event-Driven Consumer* zur Option *JMS Mesasge Listener* vornehmen.

Competing Consumers Standardmäßig erfolgt der Empfang von Nachrichten und deren Weiterverarbeitung sequenziell durch einen Empfänger. Dies kann für Anwendungen jedoch zu langsam sein, sodass sich Nachrichten im Nachrichtenkanal stauen und das Nachrichtensystem zum Flaschenhals wird. Das Entwurfsmuster *Competing Consumers* [HoWo03, S. 502-507] adressiert dieses Problem und sieht als Lösungsmöglichkeit den Einsatz von konkurrierenden Empfängern für einen einzelnen Nachrichtenkanal vor, die die Nachrichten dieses Kanals nebenläufig empfangen und verarbeiten. Das Nachrichtensystem entscheidet, welcher der konkurrierenden Empfängern eine Nachricht empfängt. Die Nachrichtenverarbeitung kann beispielsweise dadurch nebenläufig erfolgen, indem jeder dieser konkurrierenden Empfängern in einem eigenen Thread läuft.
Die JMS-Spezifikation legt keine Verhaltensweisen von nebenläufigen Nachrichtenempfängern fest. Vielmehr fordert sie nicht, dass JMS-Implementierungen diese Funktionalität bereitstellen müssen. Erfolgt der Empfang durch *Message-Driven Beans* (MDB) in einem EJB-Container, so können Nachrichten nebenläufig empfangen und verarbeitet werden. Der EJB-Container kann eine Menge von MDB-Instanzen der selben Klasse in einem Bean-Pool halten, die dann abwechselnd bei der Nachrichtenzustellung vom EJB-Container aktiviert werden. Das Entwurfsmuster *Competing Consumers* kann dem MDB-Pool des EJB-Containers zugeordnet werden, da der Fokus dieser Arbeit auf der Entwicklung von komponentenbasierten Anwendung auf Basis von Java EE liegt.

Message Dispatcher Für Anwendungen, die nachrichtenorientierte Kommunikation verwenden, kann es erforderlich sein, dass Nachrichten eines Nachrichtenkanals, von mehreren Empfängern koordiniert verarbeitet werden. Das Entwurfsmuster *Message Dispatcher* [HoWo03, S. 508-514] bietet einen Lösungsansatz der auf dem Mediator-Muster [GHJV95] basiert. Ein so genannter *Message Dispatcher* agiert als Mediator, indem dieser alle Nachrichten eines Nachrichtenkanals empfängt und diese an Komponenten zur weiteren Verarbeitung verteilt. Für die verarbeitenden Komponenten erfolgt die Koordination transparent.
Dieses Entwurfsmuster weist einen starken Anwendungsbezug auf, da die verarbeitenden Komponenten und folglich auch der koordinierende *Message Dispatcher* anwendungsspezifisch ist. Soll für eine Anwendung dieses Muster eingesetzt werden, ist eine Implementierung des *Message Dispatchers* und der verarbeitenden Komponenten erforderlich. Das Muster *Message Dispatcher* kann somit keiner generell anwendbaren JMS-Option zugeordnet werden.

Selective Consumer Das Entwurfsmuster *Selective Consumer* [HoWo03, S. 515-521] bietet eine Antwort auf die Frage wie ein Nachrichtenempfänger die Nachrichten auswählen kann, die er empfangen möchte. Das Muster sieht einen *Selective Consumer* vor, der alle Nachrichten, die über einen Nachrichtenkanal versendet werden,

filtert und ausschließlich jene Nachrichten empfängt, die seine Filterkriterien erfüllen.
In JMS können Sender für Nachrichten spezielle Eigenschaften definieren und mit Werten belegen. Empfänger, die mit einem `messageSelector` als Parameter erzeugt werden, filtern eingehende Nachrichten auf Übereinstimmung der Nachrichteneigenschaften mit dem `messageSelector`. Die Funktionalität des Entwurfsmusters *Selective Consumer* und der Option *JMS Message Selector* stimmt somit überein. Aus diesem Grund wird eine Zuordnung zwischen beiden vorgenommen.

Durable Subscriber Bei einem *Publish-Subscribe Channel* werden die Nachrichten an die Empfänger versendet, die sich für diesen Nachrichtenkanal registriert haben. Falls ein Empfänger keine Nachrichten dieses Kanals mehr empfangen möchte, meldet er sich davon ab. Dies erfolgt im Allgemeinen dadurch, dass die Verbindung zu diesem Nachrichtenkanal geschlossen wird. Registriert sich der Empfänger wieder bei diesem Nachrichtenkanal, so werden ihm alle Nachrichten zugestellt, die ab dem Zeitpunkt der Registrierung versendet werden. Das Entwurfsmuster *Durable Subscriber* [HoWo03, S. 522-527] sieht vor, dass das Nachrichtensystem alle Nachrichten, die während der Zeit der Trennung eines *Durable Subscriber* vom Nachrichtenkanal versendet wurden, zwischenspeichert. Nachdem sich der *Durable Subscriber* wieder beim Nachrichtenkanal registriert hat, werden ihm die gespeicherten Nachrichten zugestellt. Dieses Verhalten ist oftmals wünschenswert. Beispielsweise sollte ein Empfänger, dessen Verbindung aufgrund von Wartungsarbeiten oder eines fehlerbedingten Neustarts der Anwendung vom Nachrichtenkanal getrennt wurde, nach erfolgter Neuregistrierung alle Nachrichten empfangen, die während der Zeit der Trennung versendet wurden. Hingegen stellt das Nachrichtensystem bei einem *Publish-Subscribe Channel* ohne *Durable Subscriber* nicht jene Nachrichten zu, die in dem Zeitraum versendet wurden, in dem der Empfänger nicht registriert war.
JMS bietet die Möglichkeit einen dauerhaften Teilnehmer zu erzeugen, für den sich JMS analog zum Entwurfsmuster *Durable Subscriber* verhält. Das Entwurfsmuster wird folglich der Option *JMS Durable Subscription* zugeordnet.

Idempotent Receiver Bei nachrichtenorientierter Kommunikation, die das Muster *Guaranteed Delivery* verwendet, können Duplikate von Nachrichten im Nachrichtensystem vorhanden sein, beispielsweise wenn eine Empfangsbestätigung aufgrund eines Netzwerkfehlers nicht beim Sender ankommt und dieser deshalb die Nachricht nochmals versendet bis er die zugehörige Empfangsbestätigung vom Empfänger erhält. Das Nachrichtensystem wendet im Allgemeinen Mechanismen an, um Nachrichtenduplikate zu entfernen, so dass die Anwendungen sich darum nicht kümmern müssen. Das Nachrichtensystem verursacht dadurch einen zusätzlichen Overhead. Das Entwurfsmuster *Idempotent Receiver* [HoWo03, S. 528-531] bietet jedoch die Möglichkeit, diesen Overhead zu reduzieren. Empfänger, die als *Idempotent Receiver* konzipiert sind, können eine Nachricht mehrmals empfangen, ohne Probleme zu verursachen.
JMS bietet die Möglichkeit, dass die Option `DUPS_OK_ACKNOWLEDGE` als Bestätigungsmechanismus für eine JMS-*Session* gewählt wird. Die JMS-Spezifikation empfiehlt, diese Option nur zu verwenden, wenn die Empfänger Nachrichtenduplikate tolerieren können. Als Vorteil wird eine Reduktion des *Session*-Overheads aufgeführt, der durch die Duplikateliminierung innerhalb des Verbindungskontextes verursacht wird.

Aufgrund der identischen Funktionsweise dieser JMS-Option und des Entwurfsmusters *Idempotent Receiver*, erfolgt eine Zuordnung zwischen beiden.

Service Activator Anwendungen, die ihre Dienste für verschiedener Kommunikationstypen anbieten, weisen im Zusammenspiel mit anderen Anwendungen eine höhere Flexibilität auf als wenn sie sich auf einen speziellen Typ beschränken. Zum Beispiel könnte eine EJB-basierte Anwendung ihre Dienste sowohl für synchronen Zugriff über *Session Beans* als auch für asynchronen Zugriff über *Message-Driven Beans* bereitstellen. Für Dienstaufrufe ausgehend vom Empfang von Nachrichten sieht das Entwurfsmuster *Service Activator* [HoWo03, S. 532-535] einen Nachrichtenempfänger vor, der die Nachrichten eines Nachrichtenkanals mit dem aufzurufenden Dienst verknüpft. Dieser Nachrichtenempfänger wird deshalb als *Service Activator* bezeichnet und kapselt alle Details, welche die nachrichtenorientierte Kommunikation betreffen. Für den aufgerufenen Dienst ist es dadurch transparent, dass der Dienstaufruf durch den Empfang einer Nachricht initiiert wird. Der *Service Activator* kann selbst als *Polling Consumer* oder als *Event-Driven Consumer* realisiert sein.
Dieses Entwurfsmuster kann keiner JMS-Option zugeordnet werden, da es aufgrund des Dienstaufrufs einen hohen Anwendungsbezug aufweist. Für das Nachrichtensystem ist ein *Service Activator* ein gewöhnlicher Nachrichtenempfänger. Der interne Aufbau des *Service Activator* hingegen ist für das Nachrichtensystem bedeutend, er ist jedoch im Allgemeinen eine Komposition von anderen Entwurfsmustern für nachrichtenorientierter Kommunikation, wie beispielsweise dem *Polling Consumer*.

Die getroffene Zuordnung von Entwurfsmustern für nachrichtenorientierte Kommunikation zu JMS-Optionen bildet die Grundlage für die Performance-Relevanz der Muster, da auf deren Performance-Auswirkungen durch eine Leistungsbewertung der unterschiedlichen JMS-Optionen geschlossen werden kann.

5.2.2 Testanwendung

Die Evaluierung der Performance-relevanten Parameter bei nachrichtenorientierter Kommunikation, denen Entwurfsmuster zugrunde liegen, erfolgt mittels einer Testanwendung. Diese Testanwendung misst die Übertragungsdauer von Nachrichten, die zwischen Komponenten innerhalb eines Java EE-basierten Anwendungs-Servers über JMS versendet werden. Diese Messungen können für diverse Konfigurationen durchgeführt werden, die weitgehend auf den Entwurfsmustern für nachrichtenorientierter Kommunikation beruhen (siehe Unterabschnitt 5.2.1). Im Folgenden wird zunächst eine kurze Anforderungsanalyse durchgeführt und der Entwurf der Testanwendung vorgestellt, bevor auf einige Implementierungsdetails eingegangen wird.

5.2.2.1 Anforderungsanalyse

An die Testanwendung können folgende Anforderungen identifiziert werden, welche die Grundlage für den Entwurf und die Implementierung bilden:

Unterstützung verschiedener JMS-Optionen: Die JMS-Optionen, deren Zuordnung zu den Entwurfsmustern für nachrichtenorientierte Kommunikation in Tabelle 5.1 zusammengefasst ist, sollen mit der Testanwendung evaluiert

werden können. Dafür muss die Testanwendung verschiedene Konfigurationen unterstützen, die im Folgenden mit der jeweils entsprechenden JMS-Option aufgeführt ist:

- *JMS-Queue*: Nachrichtenversand über den Nachrichtenkanal `Queue`
- *JMS-Topic*: Nachrichtenversand über den Nachrichtenkanal `Topic`
- *JMS Persistent Messages*: Persistenter bzw. nicht-persistenter Nachrichtenversand
- *JMS Transacted Session*: Transaktionales Versenden von Nachrichten
- *MDB-Pool*: Nebenläufiger Nachrichtenempfang durch die Empfänger eines Empfänger-Pools
- *JMS Message Selector*: Filterung von zu empfangenden Nachrichten
- *JMS Durable Subscription*: Dauerhafte Registrierung von Empfängern am Nachrichtenkanal `Topic`
- *JMS Message Acknowledgment*: Nachrichtenversand über den Nachrichtenkanal `Queue` mit automatischem und duplikattolerantem Bestätigungsmechanismus

Im Rahmen dieser Arbeit wird der Entwurf von nebenläufigen, komponentenbasierten Anwendungen auf Basis von Java EE betrachtet. Die Spezifikation von Java EE sieht in den *Message-Driven Beans* explizit Nachrichtenempfänger vor, die auf die Zustellung von Nachrichten reagieren und damit die Option *JMS Message Listener* realisieren. Die Testanwendung folgt den Konzepten von Java EE, so dass der synchrone Nachrichtenempfang über die Option *JMS Receive* nicht weiter verfolgt wird.

Ein Leistungsbewertung wird für weitere JMS-Optionen durchgeführt, die keine Entwurfsmuster für nachrichtenorientierte Kommunikation repräsentieren:

- Versand von Nachrichten mit unterschiedlichen JMS-Nachrichtentypen
- Versand von Nachrichten mit unterschiedlicher Größe des Inhalts
- Nachrichtenübermittlung bei verteiltem Empfänger
- Nachrichtenübermittlung bei verteiltem JMS-Provider

Performante Speicherung der Messergebnisse: Die Messergebnisse müssen in einer Weise gespeichert werden, die möglichst geringe Auswirkungen auf die eigentliche Performance-Messung hat. Ansonsten würden die Messungen durch die Speicherung der Ergebnisse verfälscht.

Vergleichbarkeit der Messwerte: Die durchschnittliche Anzahl der Nachrichten im System muss konfiguriert werden können, damit eine Vergleichbarkeit der Übertragungsdauermessungen gewährleistet ist.

Vergleichbarkeit mit Anwendungen aus der Praxis: Die Testanwendung soll sich am praktischen Einsatz von nachrichtenorientierter Kommunikation orientieren, damit die Messergebnisse in die Praxis übertragen werden können.

5.2.2.2 Entwurf

Der Entwurf der Testanwendung ist an den zuvor beschriebenen Anforderungen ausgerichtet und umfasst die Komponenten `AppClient`, `ExperimentManager`, `JmsProvider`, `MessagePublisher` und `MessageReceiver`. Die Architektur, die in Abbildung 5.1 illustriert ist, sieht die Anbindung an eine Datenbank zur Speicherung der Messergebnisse vor. In der Komponente `AppClient` ist die Auswahl der JMS-Optionen und die Initiierung des eigentlichen Nachrichtenversands gekapselt. Die Komponente `MessagePublisher` stellt die JMS-Optionen auf der Senderseite zur Verfügung. Auf Empfängerseite beinhaltet diese der `MessageReceiver`. Die Komponente `JmsProvider` ist eine Implementierung des *Java Message Service* (JMS) (siehe Abschnitt 3.5.1l, [Java02]) und repräsentiert das Nachrichtensystem. Dieses übernimmt den asynchronen Nachrichtenversand zwischen dem `MessagePublisher` und dem `MessageReceiver`. Die Verwaltung der Messergebnisse erfolgt im ExperimentManager. Diese veranlasst zudem die Speicherung der Messwerte in der angebundenen Datenbank. Die Komponente `AppClient` ist als eigenständige Client-Anwendung konzipiert, die Komponenten `MessagePublisher` und `MessageReceiver` hingegen als Komponenten in einem Anwendungs-Server. Der Grund für diese Entwurfsentscheidung ist, dass in der Praxis die nachrichtenorientierte Kommunikation zwischen Komponenten innerhalb von Anwendungs-Servern eine gängige Kommunikationsmethode ist. Der `AppClient` ist vom eigentlichen Nachrichtenversand unabhängig, er dient lediglich dazu, den `MessagePublisher` im Anwendungs-Server aufzurufen.

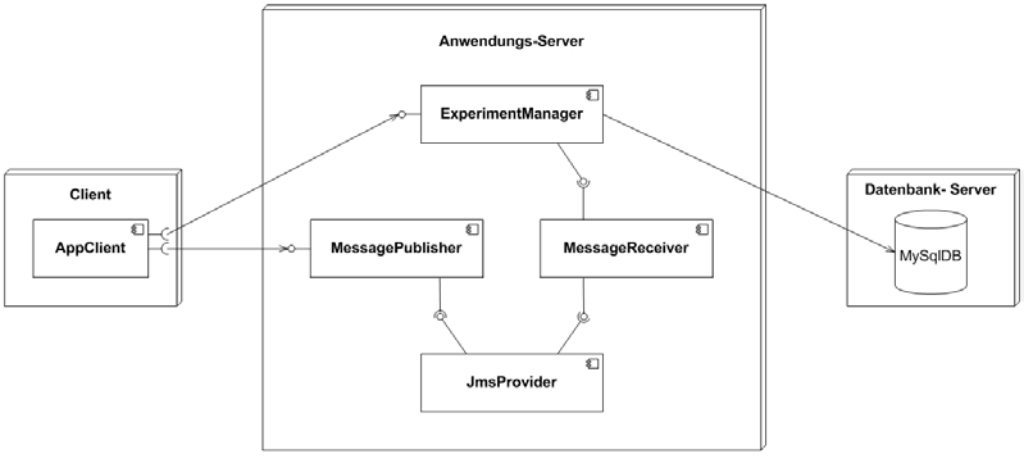


Abbildung 5.1: Architektur der Testanwendung

Die Komponente `AppClient` besteht aus den Klassen `AppClient` und `ExperimentClient`, die im Paket `de.uka.ipd.sdq.tests.jms.testsClient` zusammengefasst sind (siehe Abbildung 5.2). Die Klasse `AppClient` nimmt Aufrufparameter für die zu evaluierende JMS-Option entgegen und ruft entsprechend den Parametern eine der öffentlichen Methoden der Klasse `ExperimentClient` auf. Diese Methoden initiieren den Nachrichtenversand durch die Komponente `MessagePublisher` gemäß der gewünschten JMS-Option. Die Referenzierung des `MessagePublisher` erfolgt in der privaten Methode `lookupEnterpriseBean`, die dafür einen Namensdienst in Anspruch nimmt. Über die private Methode `waitForProceeding` kann zwischen

zwei aufeinander folgenden Initiierungen des Nachrichtenversands eine Warteperiode eingelegt werden, wodurch die Frequenz der Nachrichtensendungen und damit die durchschnittliche Anzahl der Nachrichten im Nachrichtensystem bestimmt wird.

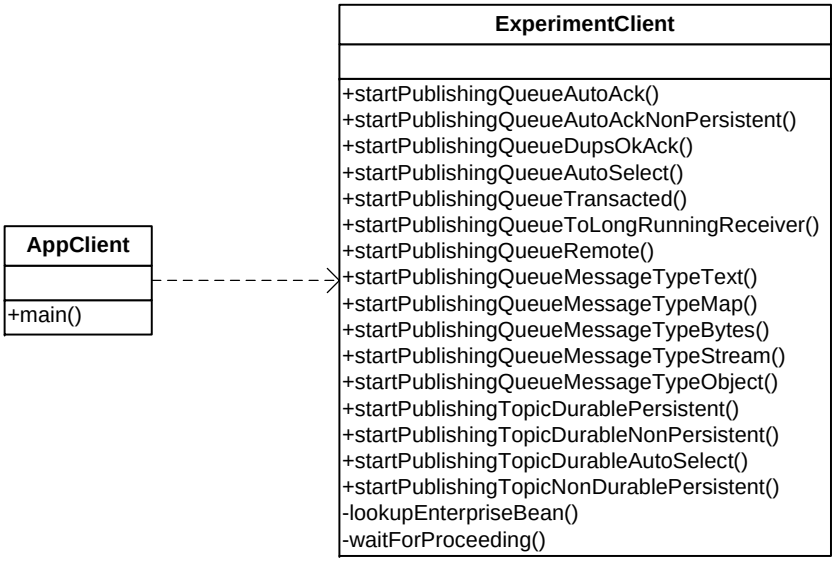


Abbildung 5.2: Klassen des Pakets `de.uka.ipd.sdq.tests.jms.testClient`

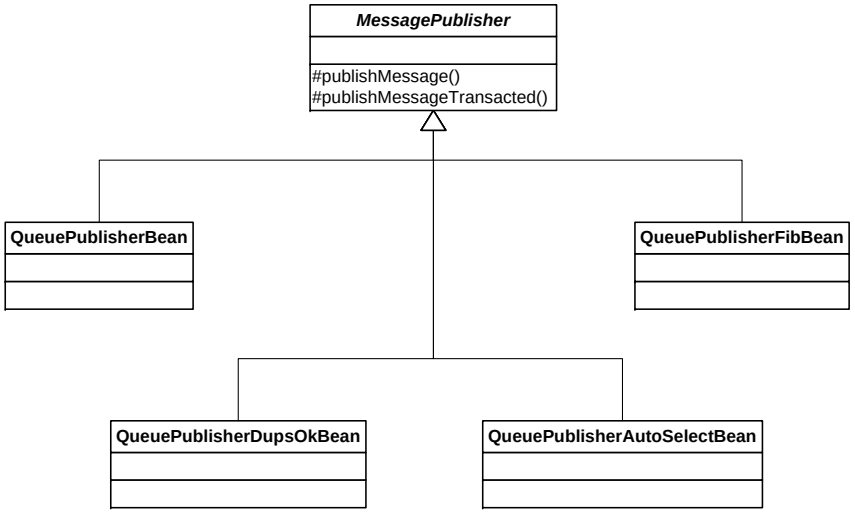


Abbildung 5.3: Klassen für den Nachrichtenversand über eine JMS-Queue

Die Klassen für den Versand von Nachrichten über eine JMS-Queue, wie sie in Abbildung 5.3 dargestellt sind, realisieren einen Teil der Komponente `MessagePublisher`. Die Klasse `MessagePublisher`, kapselt die Logik für den Nachrichtenversand. Die Unterklassen beinhalten jeweils Konfigurationen für unterschiedliche JMS-Optionen. Der Grund für diese Entwurfsentscheidung ist, dass für einige der JMS-Optionen spezielle Empfänger erforderlich sind. Eine JMS-Queue stellt jedoch eine Punkt-zu-Punkt-Kommunikation dar, sodass an einer JMS-Queue nicht mehrere Empfänger registriert werden können. Somit muss für solche JMS-Optionen jeweils eigene JMS-Queue verwendet werden, was zu unterschiedlichen Konfigurationen auf der Senderseite führt. Der Vorteil dieses Entwurfs ist die Trennung der Konfigurationen

untereinander, sowie die Trennung der Logik für den Nachrichtenversand von den einzelnen Konfigurationen. Dadurch ist eine eine gute Wartbarkeit und Änderbarkeit der Konfigurationen gegeben. Beispielsweise kann die Testanwendung für zusätzliche JMS-Optionen angepasst werden, ohne dass andere Konfigurationen davon betroffen wären. Der Start des Sendevorgangs erfolgt über die Methode `publishMessage`. Bei transaktionalem Nachrichtenversand wird stattdessen die Methode `publishMessageTransacted` aufgerufen, da zusätzlich zum Start des Nachrichtenversands noch die Transaktionsgrenzen festgelegt werden müssen.

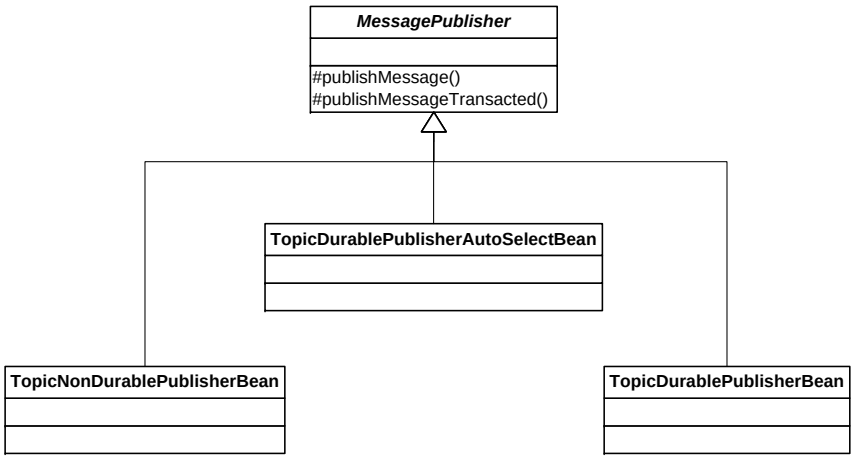


Abbildung 5.4: Klassen für den Nachrichtenversand über ein JMS-Topic

In Abbildung 5.4 sind die Klassen für den Nachrichtenversand über ein JMS-Topic illustriert, welche zusammen mit jenen für eine JMS-Queue (siehe Abbildung 5.3) die Komponente `MessagePublisher` bilden. Die Unterklassen erben ebenfalls von der Klasse `MessagePublisher` und umfassen die unterschiedliche Konfigurationen für die JMS-Optionen, die für ein JMS-Topic relevant sind. Wie beim Nachrichtenversand über eine JMS-Queue ist die Begründung hierfür, dass die Konfigurationen, die in den Unterklassen gekapselt sind, aufgrund separater JMS-Topics erforderlich sind. Die separaten JMS-Topics werden benötigt, da einige JMS-Optionen spezielle Empfänger erfordern. Diese sind jeweils an einem separaten JMS-Topic registriert. An einem JMS-Topic können zwar mehrere verschieden Empfänger registriert sein, jedoch wird davon abgesehen, da möglicherweise eine gegenseitige Beeinflussung der Empfänger auftreten kann. Bei einem Empfänger pro JMS-Topic kann dies ausgeschlossen werden. Die Konfigurationen werden durch die Kapselung in separaten Unterklassen voneinander getrennt.

Die für den Nachrichtenempfang zuständige Komponente `MessageReceiver` besteht für den Empfang über eine JMS-Queue aus fünf Unterklassen, die speziell konfigurierte Nachrichtenempfänger für unterschiedliche JMS-Optionen sind (siehe Abbildung 5.5). Diese Klasseneinteilung bietet eine strikte Trennung der Konfigurationen und eine leichte Abbildung auf *Message-Driven Beans*, die als Empfängerimplementierung vorgesehen sind. Diese Klassen erben von der Oberklasse `MessageReceiver` und implementieren dessen abstrakte Methode `onMessage`. Diese wird vom Nachrichtensystem bei der Zustellung einer Nachricht aufgerufen. Über die Methode `storeTimeSpan` wird die Übertragungsdauer zwischen dem Versende- und Empfangs-

zeitpunkt berechnet und zur Speicherung an die Komponente `ExperimentManager` übergeben.

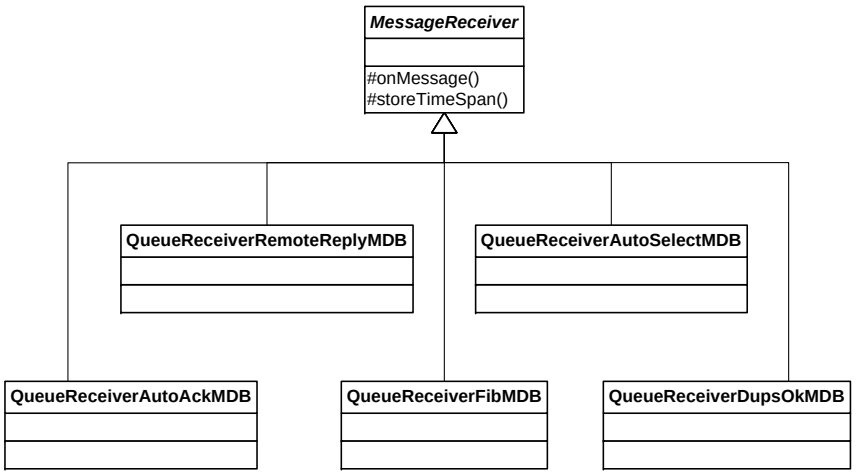


Abbildung 5.5: Klassen für den Nachrichtenempfang über eine JMS-Queue

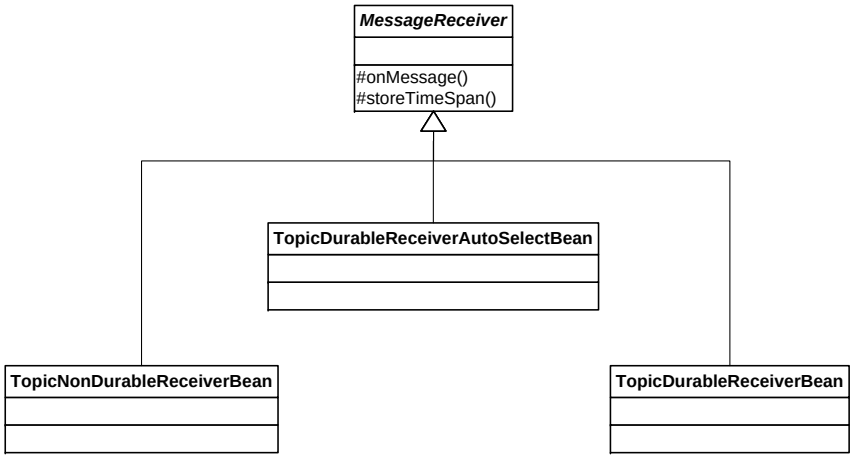


Abbildung 5.6: Klassen für den Nachrichtenempfang über ein JMS-Topic

Die Realisierung des Nachrichtenempfangs über ein JMS-Topic durch die Komponente `MessageReceiver` erfolgt in den Unterklassen der zuvor beschriebenen Klasse `MessageReceiver`, die in Abbildung 5.6 aufgeführt sind. Analog zu den Klassen für den Empfang über eine JMS-Queue, sind diese speziell konfigurierte Empfänger für unterschiedliche JMS-Optionen, die für ein JMS-Topic auf der Empfängerseite die Übertragungsdauer einer Nachricht beeinflussen können.

Für die Verwaltung und Speicherung der gemessenen Übertragungsdauerwerte ist die Komponente `ExperimentManager` zuständig. Sie umfasst die Klassen des Pakets `de.uka.ipd.sdq.tests.jms.experiment`, dessen Struktur in Abbildung 5.7 dargestellt ist. Die Klasse `ExperimentManager` ist als Einzelstück konzipiert. Dadurch wird vermieden, dass der Anwendungs-Server mehrere Objektinstanzen dieser Klasse erzeugt. Damit können alle Messwerte in der selben Instanz gesammelt werden. Über die Methode `getInstance` erfolgt die Referenzierung dieser Instanz. Über die Klasse

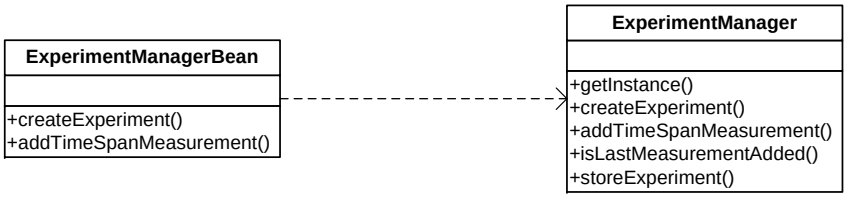


Abbildung 5.7: Klassen des Pakets `de.uka.ipd.sdq.tests.jms.experiment`

`ExperimentManagerBean` wird auf den `ExperimentManager` zugegriffen. Der Erstellung eines Experiments, in dem alle für die Messungen relevanten Daten aggregiert werden, dient die Methode `createExperiment`. Die Methode `addTimeSpanMeasurement` ermöglicht die Aufnahme eines Messwerts für die Übertragungsdauer einer Nachricht. Die Speicherung der Messwerte erfolgt kumulativ, nachdem die Messwerte für Übertragungsdauer aller Nachrichten eines Experiment gesammelt wurden. Der Grund für diese Entscheidung ist, dass die zeitintensiven Datenbankzugriffe erst nach Übertragung aller Nachrichten durchgeführt werden sollen, damit die Messergebnisse nicht verfälscht werden. Mithilfe der Methode `isLastMeasurementAdded` kann ermittelt werden, ob die Messwerte aller übertragenen Nachrichten in der Instanz gesammelt wurden. Dies ist erforderlich, da der `ExperimentManager` aufgrund der asynchronen Kommunikationsform selbst feststellen können muss, wann alle Messwerte gesammelt wurden. Die Persistierung der Messwerte in der angeschlossenen Datenbank erfolgt über die Methode `storeExperiment`.

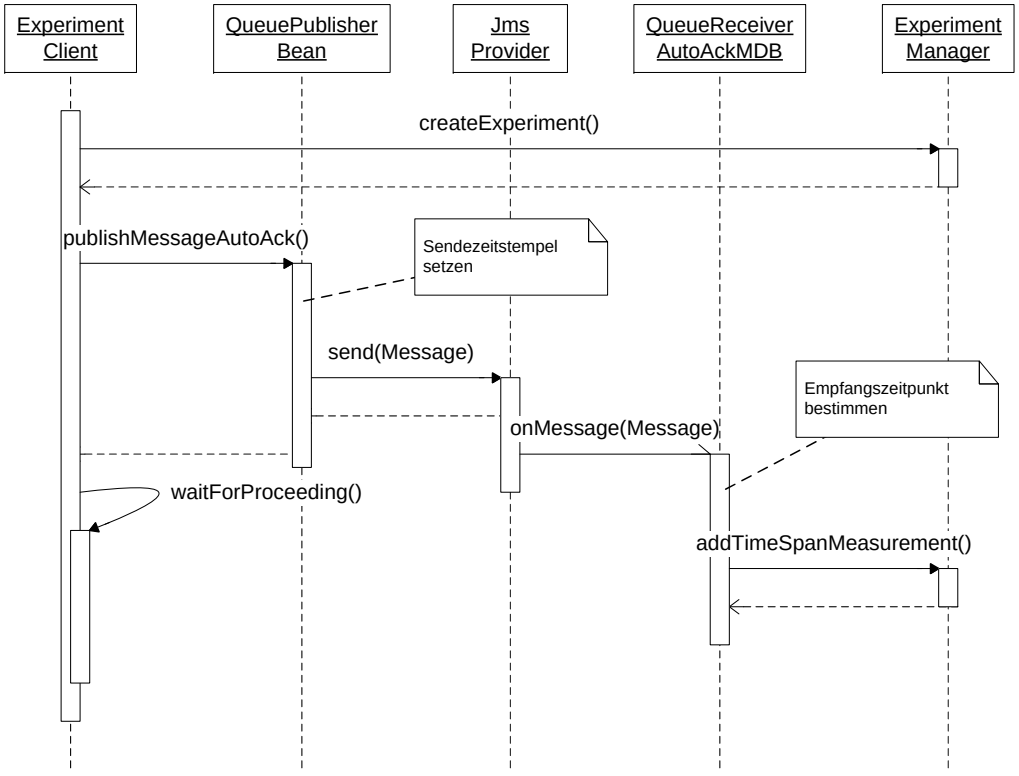


Abbildung 5.8: Sequenzdiagramm für den Ablauf eines Tests für JMS-Optionen

Der zeitliche Ablauf eines Tests zur Evaluierung der Performance von JMS-Optionen ist in Abbildung 5.8 an einem vereinfachten Beispiel mit den wichtigsten Inter-

aktionsfolgen veranschaulicht. Die JMS-Option dieses Beispiels ist der Nachrichtenversand über eine *Queue*, mit automatischem Bestätigungsmechanismus. Ausgehend vom `ExperimentClient` wird die Methode `createExperiment()` eines `ExperimentManager`-Objekts aufgerufen, um dieses für die Aufnahme der Messergebnisse zu initialisieren. Danach initiiert der `ExperimentClient` über die Methode `publishMessageAuto()` des Objekts `QueuePublisherBean` den Nachrichtenversand. Dabei wird eine Nachricht inklusive deren Sendezeitstempel per `send(Message)` an das Nachrichtensystem übergeben, das in der Abbildung vereinfacht durch den `JmsProvider` repräsentiert wird. Der Kontrollfluss kehrt zum `ExperimentClient` zurück, da die `send(Message)`-Methode nicht blockiert. Das Nachrichtensystem stellt die Nachricht dem registrierten Empfänger `QueueReceiverAutoAckMDB` asynchron zu, was im Sequenzdiagramm durch den asynchronen Methodenaufruf von `onMessage(Message)` dargestellt ist. Innerhalb dieser Methode wird der Empfangszeitpunkt bestimmt, sodass über der Differenz mit dem Sendezeitstempel, der mit der Nachricht mitgeschickt wurde, die Übertragungsdauer der Nachricht berechnet werden kann. Dieser Messwert wird mittels des Aufrufs der Methode `addTimeSpanMeasurement()` an das `ExperimentManager`-Objekt übergeben. Parallel zu den Aktivitäten des Nachrichtensystems, des Empfängers `QueueReceiverAutoAckMDB` und des `ExperimentManager`, ruft der `ExperimentClient` die Methode `waitForProceeding()` auf, um eine Warteperiode zwischen einem möglichen weiteren Nachrichtenversand einzulegen.

5.2.2.3 Implementierung

Die Beschreibung der Implementierung beschränkt sich auf zwei Aspekte, die Struktur der Anwendung auf Basis der *Enterprise JavaBeans*-Technologie und die performante Speicherung der Messergebnisse.

Die Implementierung des zuvor erläuterten Entwurfs erfolgt in der Programmiersprache Java. Die Komponenten `ExperimentManager`, `MessagePublisher` und `MessageReceiver` basieren auf der Enterprise JavaBeans-Technologie [Ente06] und verwenden einen *EJB-Container* eines Java EE-Anwendungs-Server als Laufzeitumgebung. Die Anbindung des Nachrichtensystems erfolgt über die Programmierschnittstelle des JMS. Abbildung 5.9 zeigt schematisch die Struktur der Testanwendung, die an ein Beispiel aus dem JMS-Tutorial von Sun [Java] angelehnt ist. Die Klassen `ExperimentManagerBean` und alle Unterklassen des `MessagePublisher` sind als *Session Beans* konzipiert, die Unterklassen des `MessageReceiver` als *Message-Driven Beans*.

Der `AppClient` ist eine Java-Anwendung, die getrennt von den restlichen Komponenten in einer eigenen virtuellen Maschine läuft. Mittels des *Java Naming and Directory Interface* (JNDI) [JNDI] des Anwendungs-Servers referenziert der `AppClient` Objektinstanzen von *Session Beans* innerhalb des *EJB-Container* über deren zugehörige *Remote Interfaces*, da ansonsten nicht von einer anderen virtuellen Maschine auf die *Session Beans* zugegriffen werden kann. Der `AppClient` ruft die Methoden zur Initialisierung des `ExperimentManager` über die *Session Bean* `ExperimentManagerBean`. Des Weiteren initiiert er den Nachrichtenversand. Die Referenz auf eine Instanz der `ExperimentManagerBean` bleibt zur gesamten Laufzeit des `AppClient` bestehen, so dass diese Instanz ihrerseits die Referenz auf das Einzelstück `ExperimentManager` beibehält. Dadurch ist sichergestellt, dass während der gesamten

Laufzeit der Testanwendung immer dieselbe Instanz des `ExperimentManager` referenziert wird.

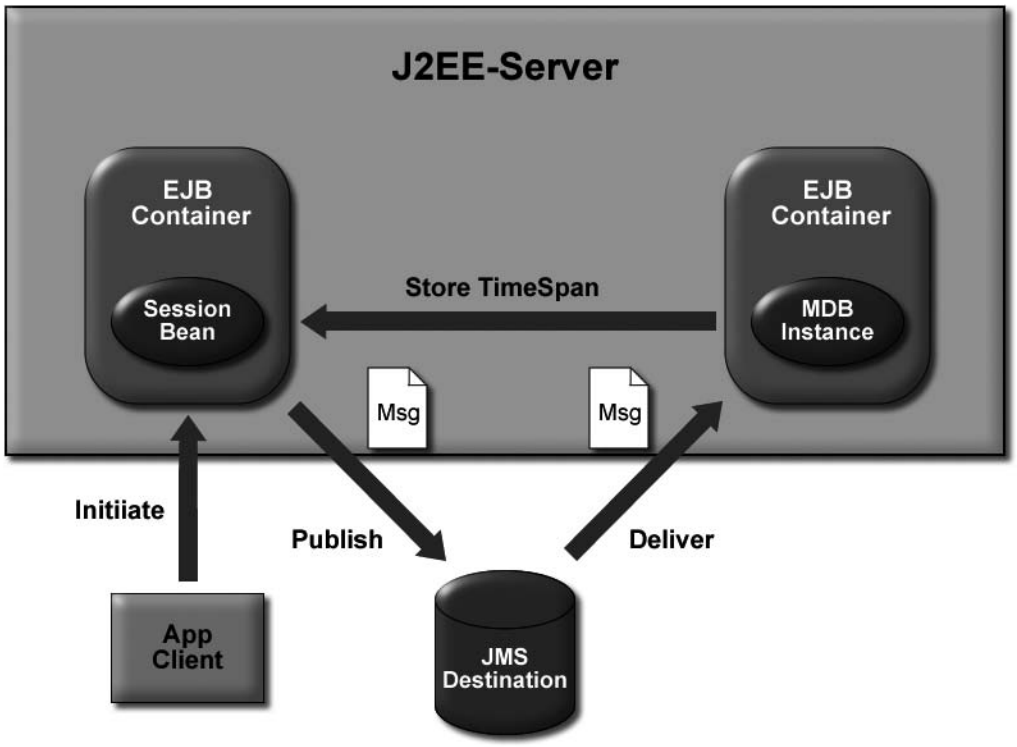


Abbildung 5.9: Struktur der Testanwendung (angelehnt an [Java, S. 104])

```
23 /**
24  * Singleton that holds an experiment instance and offers
        methods
25  * to add related objects to this instance.
26  */
27 public class ExperimentManager {
28
29         // thread"=safe counter for storing operation calls
30         private AtomicInteger storeOperationsCounter
31                 = new AtomicInteger(0);
32
33         // array for storing the measured time spans
34         private long[] measuredTimeSpans;
35
36         // array for storing the measured event times
37         private long[] eventTimes;
38
39         // array for the sensors corresponding to the
40         // measurements
41         private String[] measurementSensors;
```

Listing 5.1: Teilmenge der Attribute der Klasse `ExperimentManager`

Für die performante Speicherung der Messergebnisse, muss bei der Implementierung der Klasse `ExperimentManager` berücksichtigt werden, dass nebenläufig auf dieses Einzelstück zugegriffen werden kann. Dabei sollen sich die nebenläufigen Zugriffe möglichst nicht blockieren und in einer Weise erfolgen, dass dabei möglichst geringer Aufwand für die Synchronisation entsteht. Dies wird dadurch erreicht, dass die Messwerte in Arrays gespeichert werden, auf die über einen Index zugegriffen wird, der als atomare Variable realisiert ist. Atomare Variablen sind feingranularer und leichtgewichtiger als Sperren, da sie den Bereich des Wettbewerbs auf eine einzelne Variable beschränken. Dadurch ist die Wahrscheinlichkeit größer, dass ohne Wartezeit auf den Wettbewerbsbereich zugegriffen werden kann [GPBB+05, S. 324-329]. Listing 5.1 zeigt die für die Speicherung relevanten Attribute der Klasse. Das Array `measuredTimeSpans` nimmt die gemessenen Zeitspannen auf, im Array `eventTimes` werden die Zeitstempel des Sendezeitpunktes gespeichert. Die Zuordnung zu dem Sensor, der die Messwerte ermittelt hat, erfolgt über das Array `measurementSensors`. Die Aufrufe der Speicheroperationen werden in `storeOperationsCounter` gezählt, das gleichzeitig als Index für die Array-Position des nächsten Messwerts dient. Durch die Typisierung als `AtomicInteger` wird eine wechselseitige Beeinflussung von Threads verhindert, ohne dass dafür eine Synchronisation mittels Sperren erforderlich ist.

Beim Erzeugen eines neuen Experiments über die `createExperiment`-Methode (siehe Listing 5.2) erfolgt die Initialisierung all dieser Attribute mit der Anzahl der Nachrichtensendungen in der privaten Methode `initializeMeasurementFields`, so dass bei der Speicherung von Messwerten keine dynamischen Größenanpassungen erfolgen, welche sich negativ auf die Messungen auswirken könnten.

```
241 /**
242  * Initializes the fields that are relevant for storing
243  * measuremt values. This method should only be called
244  * from the createExperiment()-Method
245  *
246  * @param numberOfExperimentMessages the number of messages
247  * that will be transferred in this experiment
248  */
249 private void initializeMeasurementFields(
250                     int numberOfExperimentMessages){
251     this.numberOfExperimentMessages
252             = numberOfExperimentMessages;
253     measuredTimeSpans
254             = new long[numberOfExperimentMessages];
255     eventTimes = new long[numberOfExperimentMessages];
256     measurementSensors
257             = new String[numberOfExperimentMessages];
258 }
```

Listing 5.2: Methode zur Initialisierung der messwerterelevanten Attribute

Die Speicherung von Messwerten erfolgt über die öffentliche Methode `addTimeSpanMeasurement` (siehe Listing 5.3). Zunächst wird der aktuelle Wert für die Position

des nächsten Messwerts in den Arrays ermittelt und danach inkrementiert. An der ermittelten Position werden in allen drei Arrays die entsprechenden übergebenen Parameter gespeichert. Mithilfe des Zählers `storeOperationsCounter` kann bestimmt werden, ob die Anzahl der Methodenaufrufe mit der Anzahl der Nachrichtensendungen übereinstimmt, um alle gespeicherten Messwerte zu persistieren, falls die Messwerte aller Nachrichtensendungen gespeichert wurden. Die Persistierung erfolgt im Datenformat des Palladio-Sensorframeworks. Aus diesem Grund werden auch dessen *Entity*-Klassen verwendet.

```
86 /**
87  * Adds a new time span measurement to the experiment with
88  * the given experimentID
89  *
90  * @param sensorName The Name of the sensor that measured
91  * the values
92  * @param eventTime The time stamp when the event occured
93  * @param timeSpan The measured time span
94  */
95 public void addTimeSpanMeasurement(String sensorName,
96                 long eventTime, long timeSpan){
97     int currentPosition = storeOperationsCounter.
           getAndIncrement();
98     measuredTimeSpans[currentPosition] = timeSpan;
99     eventTimes[currentPosition] = eventTime;
100    measurementSensors[currentPosition] = sensorName;
101 }
```

Listing 5.3: öffentliche Methode zum Hinzufügen eines Messwerts

5.2.3 Evaluierungsergebnisse

In diesem Unterabschnitt werden die Ergebnisse der evaluierten JMS-Optionen vorgestellt. Zuerst werden die verwendete Testumgebung der Testablauf beschrieben. Es folgt für jede JMS-Optionen eine kurze Erklärung und die Erläuterung der Messergebnisse. Der vorletzte Teil dieses Unterabschnitts thematisiert Fragen, die bei der Auswertung der Messergebnisse aufgeworfen wurden, jedoch in dieser Diplomarbeit nicht weiter betrachtet werden. Den Abschluss bildet eine Zusammenfassung der Ergebnisse.

5.2.3.1 Testumgebung und Testablauf

Die Messungen der Übertragungsdauer von JMS-Nachrichten bei unterschiedlichen JMS-Optionen erfolgt auf einem System mit einem `Intel Pentium 4 2,80 GHz` als Prozessor, das über 2 GB Hauptspeicher verfügt. Als Betriebssystem wird `Microsoft Windows XP Professional SP2` verwendet. Als Laufzeitumgebung für den in Java implementierten `AppClient`-Komponente der Testanwendung kommt die `Java Runtime Environment` von Sun Mircosystems in der Version `1.6.0_01-b06` zum Einsatz. Als *EJB-Container* wurde Java EE-Anwendungs-Server `Glassfish V2 Beta 3` ausgewählt. Als Nachrichtensystem wird `Sun Java System Message Queue 4.1`

eingesetzt, das standardmäßig mit dem verwendeten Anwendungs-Server ausgeliefert wird. Entsprechend den Standardeinstellung des Anwendungs-Servers wird das Nachrichtensystem in dessen Prozess gestartet und von diesem verwaltet. Die Speicherung der Messergebnisse erfolgt in einer `MySQL 5`-Datenbank, die ebenfalls auf diesem System installiert ist.
Für zwei JMS-Optionen ist ein zweites System erforderlich, da bei diesen das Nachrichtensystem bzw. der Nachrichtenempfänger verteilt ist. Dieses System verfügt über einen `Intel Pentium M 1,50 GHz`-Prozessor und 2 GB Hauptspeicher. Die verwendete Software unterscheidet sich von derer des ersten Systems lediglich in der Version des Java EE-Anwendungs-Server, da bei der Version `V2 Beta 3` die erforderliche Registrierung eines Nachrichtenempfängers bei einem entfernten Nachrichtensystem fehlschlägt. Aus diesem Grund ist auf diesem System `Glassfish V2 Milestone 4` installiert. Beide Systeme sind über eine Netzwerkverbindung mit einer Geschwindigkeit von 100 MBit miteinander verbunden.

Bei jedem Durchlauf der Testanwendung werden 10000 Nachrichtensendungen initiiert. Die Übertragungsdauer jeder Nachricht wird in Nanosekunden mittels der Java-Funktion `System.nanoTime()` gemessen. Die Auflösung dieser Funktion auf dem verwendeten System beträgt ungefähr $50ns$. Bestimmt wurde dies mit einem Werkzeug des Lehrstuhls [Kupe07]. Für jeden Durchlauf wird der Java EE-Anwendungs-Server neu gestartet und nach Beendigung wieder heruntergefahren. Dessen Protokollierungsdateien werden vor jeden Start ebenso gelöscht wie die Datei, in der das Nachrichtensystem zu persistierende Nachrichten ablegt. Dadurch läuft die Testanwendung für alle JMS-Optionen unter vergleichbaren Bedingungen ab, so dass eine Vergleichbarkeit der Ergebnisse gewährleistet ist. Der Inhalt der gesendeten Nachrichten besteht aus einer Zeichenkette mit zufälligen Zeichen der Länge „n“, je nach JMS-Konfiguration variiert die Länge der Zeichenkette, beispielsweise um den Einfluss der Nachrichtengröße auf die Übertragungsdauer ermitteln zu können.

5.2.3.2 Evaluierung

Die evaluierten JMS-Konfigurationen werden im Folgenden kurz erklärt. Für jede der Konfigurationen erfolgt eine Beschreibung der gemessenen Werte für die Übertragungsdauer. Von Bedeutung sind vor allem der Median der Übertragungsdauer, sowie die Verteilung der Übertragungsdauerwerte. Ein einfacher Vergleich der Übertragungsdauerwerte bei unterschiedlichen JMS-Konfigurationen kann anhand eines zentralen Repräsentanten durchgeführt werden. Hierfür wird der Median verwendet, da sich unter den Messergebnisse einige Ausreißer befinden. Gegen jene ist der Median resistenter als andere statistische Kenngrößen wie z. B. der Mittelwert [SaHe06, S. 69]. Die Verteilung ermöglicht detailliertere Aussagen, da sie die Streuung und mögliche Konzentrationen der Messwerte zeigt. Sie bildet die Grundlage für die Modellierung des Ressourcenbedarf beim Entwurf der Modellkonstrukte in Abschnitt 5.3.
Der Verlauf der Werte für die Übertragungsdauer in Abhängigkeit von der Empfangsreihenfolge zeigt eine Aufwärmphase bevor sich die Messwerte auf einem stabilen Niveau bewegen. Der Grund für diese Aufwärmphase liegt in internen Abläufen im Anwendungs-Server, da beispielsweise Instanzen von *Enterprise Java Beans* erzeugt werden müssen. In Abbildung 5.10 zeigt sich am Beispiel des Nachrichtenversands über eine `Queue` mit automatischem Bestätigungsmechanismus, dass diese Aufwärmphase erst nach der Übertragung von 2000 bis 2500 Nachrichten beendet ist. Extreme

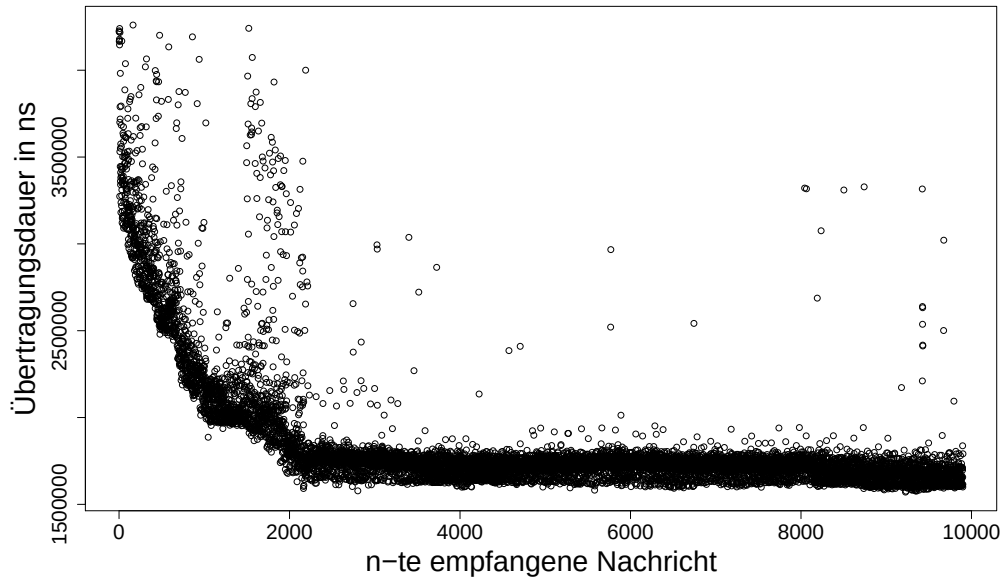


Abbildung 5.10: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ inklusive der Aufwärmphase

Ausreißer wurden für die Analyse der Aufwärmphase ausgeschlossen, indem die 100 größten Werte für die Übertragungsdauer abgeschnitten werden.

Als Folge der Bestimmung der Aufwärmphase, werden die Ergebnisse der Übertragungsdauermessungen erst ab der 2500. empfangenen Nachricht für die Analyse der Messwerte herangezogen. Dadurch wird vermieden, dass die Aufwärmphase zu falschen Interpretationen führt. Für die Berechnung des Medians werden somit 7500 der 10000 Nachrichteninitiierungen berücksichtigt. Für Verlauf der Übertragungsdauer und die Verteilung der Übertragungsdauer wird der Umfang der Messwerte zusätzlich gestutzt, um durch den Ausschluss der Ausreißer die Interpretation der Messergebnisse zu erleichtern. Da nur sehr wenige extreme Ausreißer nach oben festzustellen sind, wird das oberste Perzentil abgeschnitten, so dass schließlich 7425 Messwerte für die Auswertung relevant sind. Eine deutlichere Stutzung, beispielsweise durch das Abschneiden des oberen und unteren Quartils, hätte zur Folge, dass nicht nur die Ausreißer ausgeschlossen würden, sondern dass auch die vorhandene Streuung der Messwerte nicht ausreichend berücksichtigt werden könnte.

Queue mit automatischem Bestätigungsmechanismus

Bei dieser Konfiguration erfolgt der Nachrichtenversand über eine `Queue` als Nachrichtenkanal, bei dem der automatische Bestätigungsmechanismus des Nachrichtensystems verwendet wird. Jede Nachricht enthält als Inhalt eine Zeichenkette von fünf zufälligen Zeichen und wird entsprechend den Standardeinstellungen von JMS persistent gespeichert. Zwischen zwei Nachrichteninitiierungen wird eine Warteperiode von $50ms$ eingelegt.
Der Median der Messwerte für die Übertragungsdauer liegt bei ca. $1.7ms$ (siehe Tabelle 5.2). Der Verlauf der Übertragungsdauer in Abhängigkeit von der Emp-

Konfiguration	Median	Verhältnis zu „Queue mit automatischem Bestätigungsmechanismus“
Queue mit automatischem Bestätigungsmechanismus	1715840ns	-
Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung	1240779ns	0,72
Queue mit duplikattolerantem Bestätigungsmechanismus	1698024ns	0,99
Queue mit automatischem Bestätigungsmechanismus und automatischer Filterung	1836094ns	1,07

Tabelle 5.2: Median der Übertragungsdauer für verschiedene Konfigurationen des Nachrichtenversands über den Nachrichtenkanal `Queue`

fangsreihenfolge, der in Abbildung 5.11 dargestellt ist, zeigt bis zur 6000. empfangenen Nachricht zwei Häufungsbereiche, bei Übertragungsdauerwerten, die geringfügig größer sind als der Median. Die nachfolgenden Nachrichten weißen jedoch vornehmlich Übertragungsdauerwerte auf, die kleiner als der Median sind. Entsprechende Häufungen sind dem Histogramm (siehe Abbildung 5.12) zu entnehmen. Über die Gründe für diese Verringerung der Übertragungsdauer lassen sich nur Vermutungen anstellen, zum Beispiel können dynamische Lastanpassungen im Anwendungs-Server bzw. im Nachrichtensystem dafür verantwortlich sein. Zur Klärung dieses Phänomens sind weitergehende Untersuchungen erforderlich. Im Rahmen dieser Arbeit wird dieses Verhalten jedoch nicht weiter betrachtet, da dies nur ein kleiner Teil der Messwerte betrifft und deshalb von dieser Häufung abstrahiert wird. Des Weiteren sind die beiden Häufungsbereiche bei 1,71ms und 1,75ms nicht sehr deutlich voneinander abgetrennt, denn das Intervall dazwischen kann nahezu genauso viele Messwerte auf sich vereinigen. Aus diesem Grund werden diese Häufungsbereiche bei der Modellierung des Ressourcenbedarfs zu einem Häufungsbereich kumuliert.
Wird die Größe des Nachrichteninhalts variiert, so sind bis zu einer Größe von 1000 Zeichen nur marginale Unterschiede bei der Übertragungsdauer zu messen (siehe Tabelle 5.3). Bei einem Nachrichteninhalt, der mehr als 1000 Zeichen umfasst sind, werden mit wachsender Zeichenzahl deutlich steigende Übertragungsdauerwerte gemessen.

Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung

Diese Konfiguration entspricht weitgehend der zuvor beschriebenen Konfiguration „Queue mit automatischem Bestätigungsmechanismus“, jedoch werden die Nachrichten nicht auf der Festplatte persistent gespeichert. Durch den Vergleich dieser Konfiguration mit der Vorherigen kann bestimmt werden, in wie sich die JMS-Option *JMS Persistent Messages* auf die Performance auswirkt.
Der Verzicht auf die Persistierung von Nachrichten und damit verbundene Wegfall des Zugriffs auf den persistenten Speicher hat zur Folge, dass die Übertragungsdauer von Nachrichten im Vergleich mit der Persistierung bei der Konfiguration „Queue

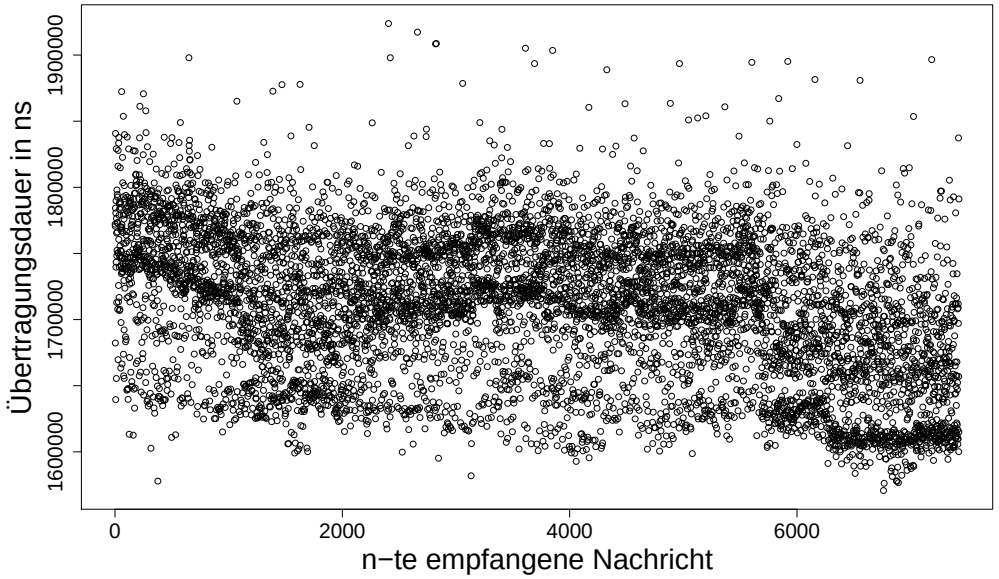


Abbildung 5.11: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus“

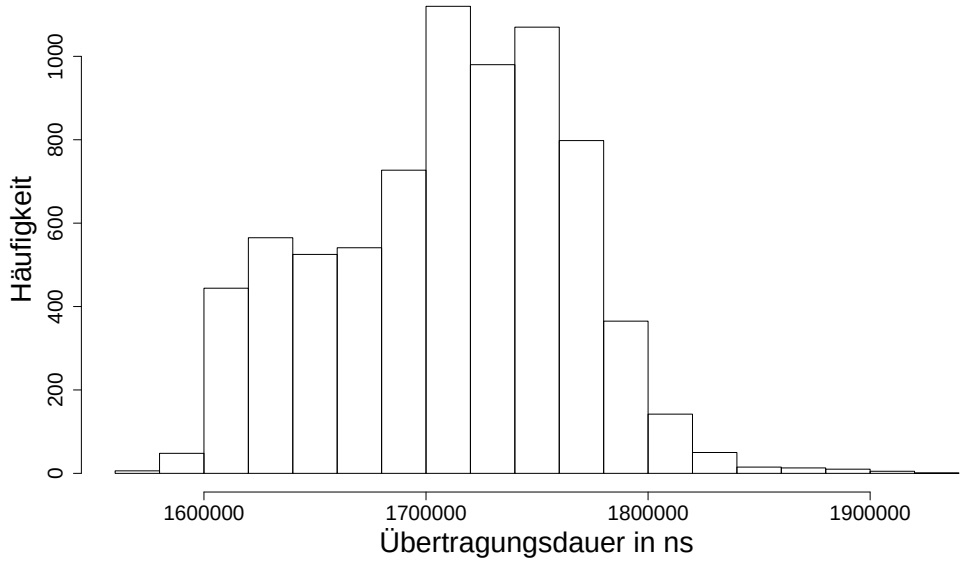


Abbildung 5.12: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus“

Anzahl der Zeichen als Nachrichteninhalt	Median	Verhältnis zu einem Inhaltszeichen
1	1746585,5ns	-
10	1783596,5ns	1,02
100	1724385,5ns	0,99
1000	1801700ns	1,03
10000	2077454,5ns	1,19
100000	4403709ns	2,52

Tabelle 5.3: Median der Übertragungsdauer in Abhängigkeit von der Größe des Nachrichteninhalts für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus“

mit automatischem Bestätigungsmechanismus“ deutlich geringer ist. Der Median der Übertragungsdauer liegt bei etwa 1,2ms, was eine Verringerung der Übertragungsdauer von 28% bedeutet (siehe Tabelle 5.2). Charakteristisch für die Verteilung der Übertragungsdauer bei dieser Konfiguration ist, dass es zwei Bereiche gibt, bei denen sich die Messwerte stark häufen (siehe Histogramm in Abbildung 5.13). In den Spannen 1,15ms - 1,20ms und 1,25ms - 1,30ms befinden sich ungefähr 34% bzw. 27% der Messwerte, im Bereich zwischen 1,20ms und 1,25ms liegen hingegen lediglich ca. 16% der Messwerte.
In Tabelle 5.4 sind die Werte für die Übertragungsdauer in Abhängigkeit von der Nachrichtengröße aufgelistet. Wie bei der persistenten Nachrichtenübertragung sind erst ab einem Inhalt, der aus mehr als 1000 Zeichen besteht, deutlich höhere Werte für die Übertragungsdauer messbar. Im Verhältnis zu einem Nachrichteninhalt von einem Zeichen steigt die Übertragungsdauer bei steigender Nachrichtengröße ohne Persistierung schneller als beim persistenten Nachrichtenversand. Insgesamt lässt sich durch den Vergleich der Konfigurationen “Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“ und „Queue mit automatischem Bestätigungsmechanismus“ feststellen, dass die JMS-Option *JMS Persistent Messages* mit einer Erhöhung der Übertragungsdauer verbunden ist, die bei großen Nachrichten vermindert ausfällt.

Anzahl der Zeichen als Nachrichteninhalt	Median	Verhältnis zu einem Inhaltszeichen
1	1192717ns	-
10	1207990ns	1,01
100	1196674ns	1,00
1000	1223311ns	1,03
10000	1456620ns	1,22
100000	3582204ns	3,00

Tabelle 5.4: Median der Übertragungsdauer in Abhängigkeit von der Größe des Nachrichteninhalts für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“

Queue mit duplikattolerantem Bestätigungsmechanismus

Bei dieser Konfiguration wird im Vergleich mit der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ mit `DUPS_OK_ACKNOWLEDGE` ein Bestätigungs-

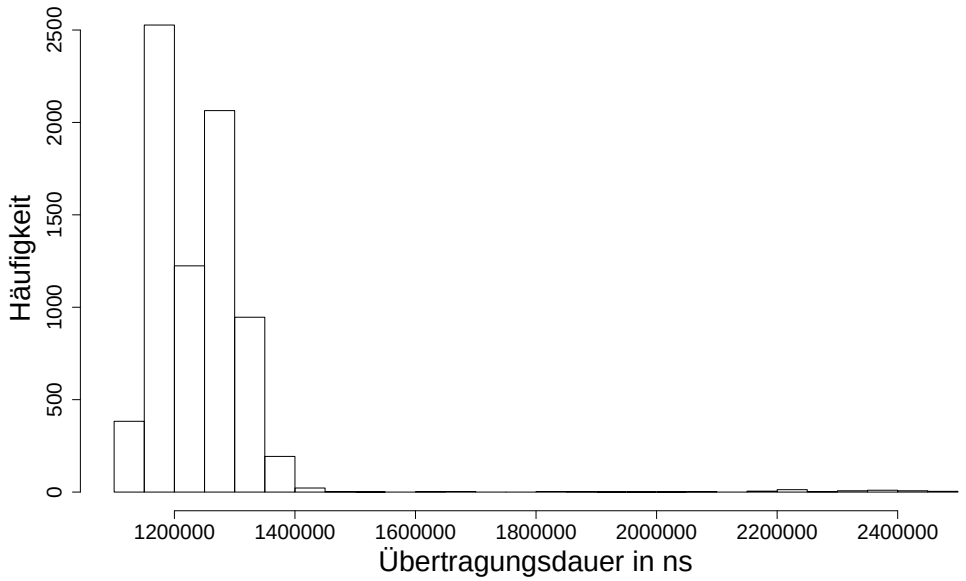


Abbildung 5.13: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“

mechanismus verwendet, bei dem Nachrichtenduplikate toleriert werden. Somit kann durch den Vergleich beider Konfigurationen bestimmt werden, in welchem Maße sich die JMS-Option *JMS Message Acknowledgment* die Performance beeinflusst.
Der Bestätigungsmechanismus `DUPS_OK_ACKNOWLEDGE` ist mit einen geringeren Verwaltungsaufwand im Nachrichtensystem verbunden als der Automatische, was eine geringere Nachrichtenübertragungsdauer vermuten lässt. Die Messergebnisse bestätigen diese Vermutung nicht. Gemäß Tabelle 5.2 kann zwischen den Konfigurationen „Queue mit automatischem Bestätigungsmechanismus“ und „Queue mit duplikattolerantem Bestätigungsmechanismus“ nur ein marginaler Unterschied beim Median der Übertragungsdauer festgestellt werden. Die Verteilungen der Messwerte für die Übertragungsdauer beider Konfigurationen unterscheiden sich zwar darin, dass bei der Konfiguration „Queue mit duplikattolerantem Bestätigungsmechanismus“ zwei ausgeprägtere Häufungsbereiche aufweist. Wie der Vergleich der Mediane zeigt, unterscheiden sich die Werte für die Übertragungsdauer der Nachrichten nur unwesentlich, sodass die JMS-Option *JMS Message Acknowledgment* nicht als Performancerelevant eingestuft wird.

Queue mit automatischem Bestätigungsmechanismus und automatischer Filterung

Diese Konfiguration ist identisch mit der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“, jedoch wird zusätzlich jede zugestellte Nachricht vor dem Empfang automatisch auf die Erfüllung eines Filterkriteriums überprüft. Hiermit wird der Einfluss der JMS-Option *JMS Message Selector* auf die Übertragungsdauer ermittelt.
Die Filterung von eingehenden Nachrichten auf Empfängerseite erfordert aufgrund der Überprüfung der Filterkriterien zusätzlichen Aufwand. Dieser Aufwand spiegelt

sich in den Messwerten der Übertragungsdauer für diese Konfiguration wider. Der Median weist einen um 7% größeren Wert auf als jener der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ (siehe Tabelle 5.2). Die Verteilung der Übertragungsdauerwerte zeigt zwar eine gleichmäßigere Streuung der Messwerte um den Median, als dies bei der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ festzustellen ist. Dennoch können die Einbußen bei der Übertragungsrate, die von der JMS-Option *JMS Message Selector* bedingt sind, mit etwa 7% beziffert werden.

Queue mit transaktionalem Nachrichtenversand

Bei der Anwendung einer transaktionalen Sitzung werden mehrere Nachrichten in einem Paket zusammengefasst. Es wurden Messungen für 1, 2, 4, 10, 20, 100, 200, 400, 1000 und 2000 Nachrichten pro Paket durchgeführt. Damit kann bezogen auf die 7425 relevanten Nachrichten das gesamte Spektrum von Transaktionen mit sehr wenigen Nachrichten bis hin zu Transaktionen, die einen sehr großen Umfang an Nachrichten haben, abgedeckt werden. Die Wahl der Transaktionsgrößen ist darüber hinaus in der Vermutung begründet, dass eine Verdoppelung bzw. Verzehnfachung der Transaktionsgröße sich in einer Verdoppelung bzw. Verzehnfachung der maximalen Übertragungsdauer innerhalb einer Transaktion auswirkt. Mit der transaktionalen Nachrichtenübertragung geht eine automatische Bestätigung der Nachrichten einher, so dass hier kein zusätzlicher Bestätigungsmechanismus ausgewählt werden muss. Je nach Anzahl an Nachrichten pro Paket wird die Dauer der Warteperiode auf einen Wert zwischen $100ms$ und $15000ms$ festgelegt, damit im Durchschnitt immer nur die Nachrichten eines Pakets versendet werden. Über die Messergebnisse kann die Performance-Relevanz der JMS-Option *JMS Transacted Session* bestimmt werden.
Beim transaktionalen Nachrichtenversand ist die Anzahl der Nachrichten pro Transaktion für die Übertragungsdauer einzelner Nachrichten von Bedeutung, wie Tabelle 5.5 zu entnehmen ist. Besteht eine Transaktion lediglich aus einer Nachricht, so ist der Median der Übertragungsdauerwerte im Bereich von jenem der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“. Sobald mehrere Nachrichten in einer Transaktion zusammengefasst werden, spielt zusätzlich auch die Position einer Nachricht innerhalb einer Transaktion eine Rolle. Der Grund dafür ist, dass die Nachrichten einer Transaktion als Paket übertragen werden, so dass die Nachricht, die als erste der Transaktion hinzugefügt wurde, erst dann zusammen mit allen anderen Nachrichten der Transaktion dem Empfänger zugestellt wird, nachdem die letzte Nachricht in die Transaktion aufgenommen wurde. Hieraus lässt sich die Vermutung anstellen, dass die erste Nachricht die längste Übertragungsdauer aufweist und die letzte Nachricht die kürzeste Übertragungsdauer. Diese Vermutung wird jedoch durch die gemessenen Werte für die Übertragungsdauer widerlegt. Am Beispiel des in Abbildung 5.14 dargestellten Verlaufs der Übertragungsdauerwerte bei Transaktionen mit einem Umfang von jeweils 10000 Nachrichten ist zu erkennen, dass die Messwerte das genau gegenteilige Verhalten aufweisen. Die Begründung dafür ist, dass der Empfangsprozess offenbar mit einem höheren Aufwand verbunden ist als der Sendevorgang, denn die Differenz der Empfangszeitpunkte zweier aufeinanderfolgender Nachrichten ist im Durchschnitt deutlich größer als die Differenz der Sendezeitpunkte. Die erste empfange Nachricht ist davon noch nicht betroffen, für alle folgenden Nachrichten der Transaktion summiert sich der Empfangsaufwand,

so schließlich die letzte Nachricht innerhalb der Transaktion wesentlich länger im Nachrichtenkanal verbleibt als die erste. Die Verzögerung beim Nachrichtenversand, die bis auf die letzte Nachricht alle anderen Nachrichten einer Transaktion erfahren, wird von der Empfangsverzögerung überlagert. In Abbildung 5.14 sind deutlich der Verlauf der unterschiedlichen Übertragungsdauerwerte für die Nachrichten einer Transaktion sowie die Transaktionsgrenzen erkennbar. Die Vermutung, dass eine Verdoppelung bzw. Verzehnfachung der Nachrichtenanzahl pro Transaktion ungefähr eine Verdopplung bzw. Verzehnfachung der maximalen Übertragungsdauer zur Folge hat, bestätigt sich durch die Messergebnisse. Zum Beispiel befinden sich die maximalen Werte für die Übertragungsdauer bei 200 Nachrichten pro Transaktion im Intervall zwischen $260ms$ und $270ms$, bei 400 Nachrichten pro Transaktion verteilen sich die maximalen Werte auf die Spanne $520ms$ - $540ms$. Für die Verteilung der Übertragungsdauerwerte bedeutet die unterschiedliche Übertragungsdauer für die Nachrichten einer Transaktion eine Auffächerung des Bereichs mit großen Häufigkeiten. Bei einer Nachricht pro Transaktion weist die Verteilung noch eine starke Häufung der Messwerte im Intervall zwischen $1{,}6ms$ und $1{,}75ms$ auf. Abbildung 5.15 zeigt exemplarisch, dass sich mit steigender Nachrichtenanzahl pro Transaktion die Häufigkeit der Übertragungsdauerwerte einem einheitlichen Niveau für die Intervalle zwischen der minimalen und maximalen Übertragungsdauer von Nachrichten einer Transaktion annähert. Insgesamt lässt sich auf Basis der durchgeführten Messungen feststellen, dass die JMS-Option *JMS Transacted Session*, abhängig von der Anzahl an Nachrichten pro Transaktion, einen sehr großen Einfluss auf die Übertragungsdauer einer Nachricht hat.

Nachrichtenanzahl pro Transaktion	**Median**	**Verhältnis zu einer Nachricht pro Transaktion**
1	$1665917ns$	-
2	$2506566ns$	1,50
4	$4157104ns$	2,50
10	$9145595ns$	5,49
20	$17012373ns$	10,21
100	$82752583ns$	49,67
200	$172879011ns$	103,77
400	$356843626ns$	214,20
1000	$943539863ns$	566,38
2000	$1992597735ns$	1196,10

Tabelle 5.5: Median der Übertragungsdauer beim transaktionalen Nachrichtenversand über den Nachrichtenkanal `Queue`

Queue mit Empfänger-Pool

Der Nachrichtenversand erfolgt bei dieser Konfiguration über eine `Queue` mit automatischem Bestätigungsmechanismus. Jeder Nachricht wird persistiert und ist eine Textnachricht mit einer Zeichenkette von fünf zufälligen Zeichen als Inhalt. Die Warteperiode zwischen zwei Nachrichteninitiierungen beträgt $100ms$. Die Nachricht werden von mehreren konkurrierenden Empfängern in einem Empfänger-Pool empfangen, der im verwendeten Java EE-Anwendungs-Server als *Message-Driven Bean-*

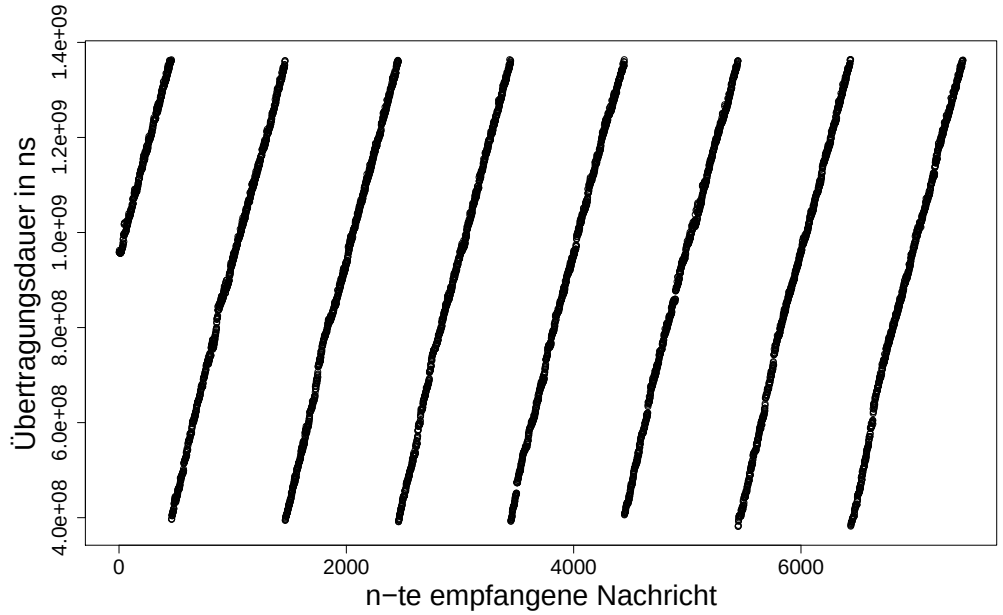


Abbildung 5.14: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand“ bei 1000 Nachrichten pro Transaktion

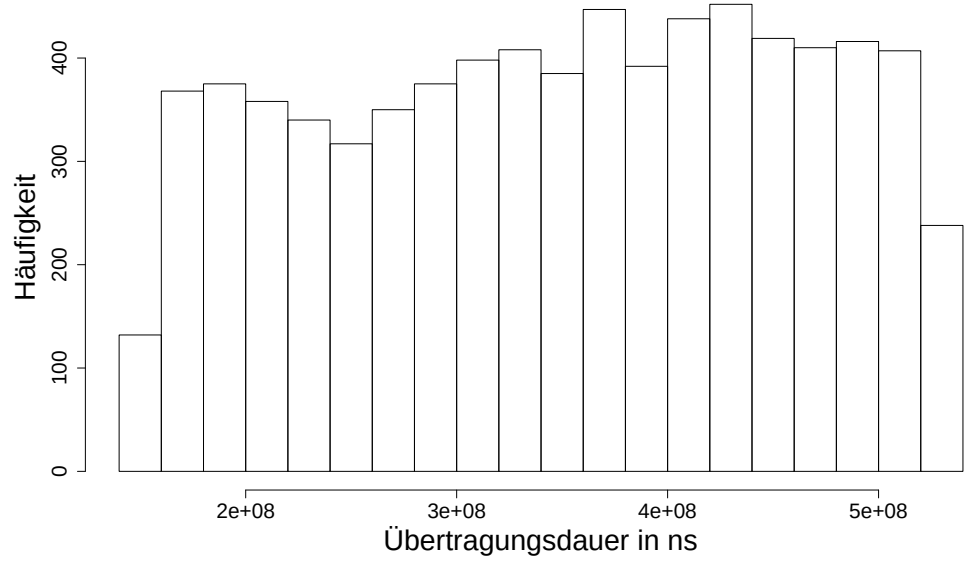


Abbildung 5.15: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand“ bei 400 Nachrichten pro Transaktion

Pool realisiert ist. Messungen wurden für dynamische Poolgrößen von 8-32, 0-1, 0-8, 0-32 und 0-128 *Message-Driven Beans*, sowie für statische Poolgrößen von 8, 32 und 128 *Message-Driven Beans* durchgeführt. Bei den dynamischen Poolgrößen kennzeichnet die erste Zahl die initiale Anzahl an Nachrichtenempfängern, die im Pool vorhanden sind. Reicht diese Anzahl nicht aus um auf Anfragen reagieren zu können, wird die Anzahl dynamisch bis zum Maximalwert erhöht, der durch die zweite Zahl der Poolgrößen bestimmt wird. Jeder Nachrichtenempfänger führt nach dem Empfang einer Nachricht die rekursive Berechnung von der Fibonacci-Zahl von 35 aus, so dass ein Nachrichtenempfänger erst nach Beendigung dieser Berechnung wieder empfangsbereit ist und nicht sofort nach dem Empfang einer Nachricht. Dadurch ist es möglich den Performance-Einfluss der JMS-Option *MDB-Pool* zu bestimmen.
Diese Konfiguration kann durch ein Warteschlangenmodell ausgedrückt werden, bei dem Kunden mit einer Zwischenankunftszeit von $100ms$ in das System eintreten und die von einer oder mehreren Bedienstationen bedient werden. Die Anzahl der Bedienstationen entspricht der Größe des Empfänger-Pools, die Bedienzeiten sind die Ausführungszeiten der Fibonacci-Berechnung. Die Warteschlange wächst ins Unendliche, wenn mehr Kunden ins System eintreten, als das System nach der Beendigung ihrer Bedienung wieder verlassen. Dieses Verhalten zeigen die Messungen bei den Poolgrößen 0-1, 0-8 und 8 *Message-Driven Beans*. Die Übertragungsdauer einer Nachricht steigt proportional mit der Empfangsnummer (siehe Abbildung 5.16), bzw. annähernd proportional bei acht konkurrierenden Empfängern. Bei diesen Poolgrößen kommt es somit zu einem Überlauf der Warteschlange, so dass die Übertragungsdauer gegen unendlich wächst. Die Ausführungszeit der Fibonacci-Berechnung ist bei 0-1 Empfängern fast konstant, bei Poolgrößen von maximal acht *Message-Driven Beans* ist bereits eine leichte Auffächerung innerhalb eines breiteren Streifens erkennbar. Der Grund dafür, dass die einzelnen Empfänger-Threads nicht gleichmäßig den Prozessor zugeteilt bekommen. Bei den Poolgrößen mit 32 und mehr Empfängern liegen die Medianwerte für die Übertragungsdauer nahe beieinander. Abbildung 5.17 zeigt beispielhaft, dass sich bei allen Poolgrößen mit mindestens 32 Empfängern die Ausführungszeiten für die Fibonacci-Berechnung auf ein breites Spektrum verteilen. Diese Zeiten steigen mit wachsender Poolgröße. Dies ist darin begründet, dass immer mehr Empfänger vorhanden sind, die um die Hardware-Ressourcen konkurrieren und somit öfter unterbrochen werden. Analog zur Übertragungsdauer erreicht auch die Ausführungszeit ab einer Poolgröße von 32 Beans ein stabiles Niveau, was darauf hinweist, dass sich ab dieser Größe nahezu durchgängig wartende Empfänger im Empfänger-Pool befinden, so dass eine eingehende Nachricht direkt bedient werden kann. Die Werte für die Übertragungsdauer weisen bei bei der Poolgröße von 0-1 *Message-Driven Beans* keine Häufung auf, bei den beiden Poolgrößen mit acht konkurrierenden Empfängern gibt es eine leichte Häufung. Bei den Poolgrößen mit mehr als 32 *Message-Driven Beans* ist eine starke Konzentration im Bereich des jeweiligen Medians festzustellen. Die Poolgröße hat entsprechend der Warteschlangentheorie einen entscheidenden Einfluss auf die Übertragungsdauer der Nachrichten, wenn auf der Empfängerseite aufwändige Berechnungen durchgeführt werden. Ein zu kleiner Pool führt zu einem Stau der Nachrichten und damit zu einer vielfach größeren Übertragungsdauer.

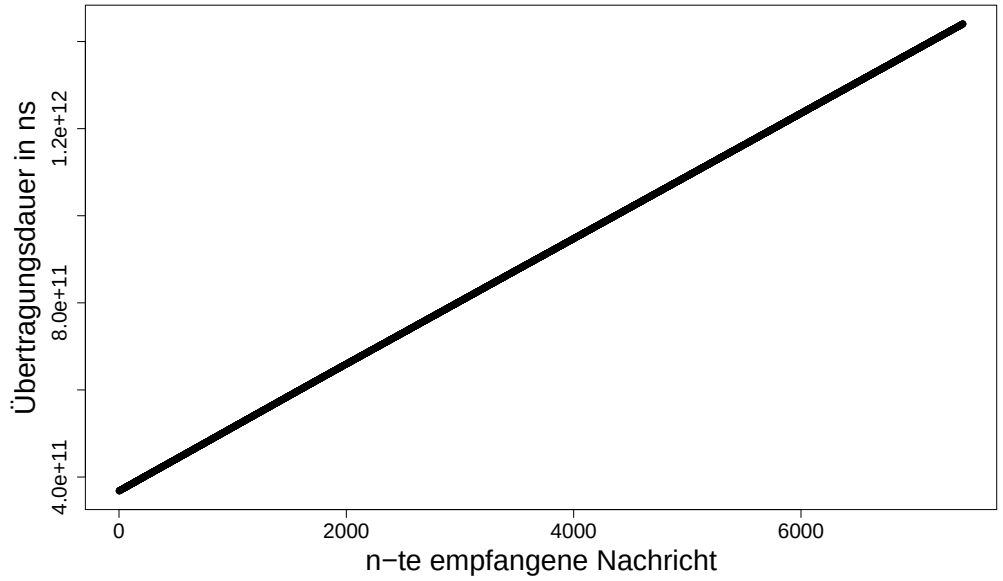


Abbildung 5.16: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit Empfänger-Pool“ bei einer Poolgröße von 0-1 *Message-Driven Beans*

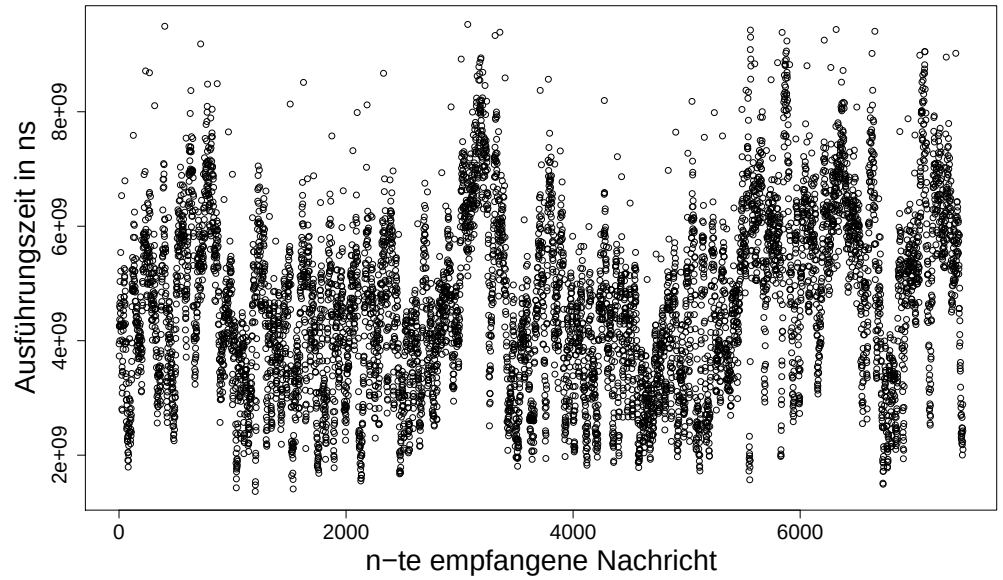


Abbildung 5.17: Verlauf der Ausführungszeit der Berechnungen auf Empfängerseite für die Konfiguration „Queue mit Empfänger-Pool“ bei einer Poolgröße von 0-128 *Message-Driven Beans*

Queue mit verschiedenen Nachrichtentypen

JMS bietet mit *TextMessage*, *MapMessage*, *BytesMessage*, *StreamMessage* und *ObjectMessage* unterschiedliche Nachrichtentypen. Deren Auswirkung auf die Performance wird durch den Versand von Nachrichten mit unterschiedlicher Inhaltsgröße ermittelt. Gemessen wurde die Übertragungsdauer für Zeichenketten als Nachrichteninhalt, die aus 1, 10, 100, 1000, 10000 und 100000 zufälligen Zeichen bestehen. Der Typ *BytesMessage* unterstützt Zeichenketten mit maximal 65535 Zeichen, so dass hierfür keine Messung für 100000 Zeichen erfolgte. Bei dieser Konfiguration werden die Nachrichten über eine `Queue` mit automatischem Bestätigungsmechanismus übertragen, jedoch ohne sie zu persistieren. Zwischen zwei Nachrichteninitiierungen wird eine Warteperiode von $100ms$ eingelegt. Diese Konfiguration basiert auf den JMS-spezifischen Nachrichtentypen und entspricht keiner der JMS-Optionen, die die Entwurfsmuster für nachrichtenorientierte Kommunikation repräsentieren. Dennoch wird sie in die Evaluierung miteinbezogen, da die Nachrichtentypen in [SJSM] als Performance-relevant eingestuft werden.
Der Nachrichtentyp wirkt sich bei kurzen Nachrichten geringfügig auf die Übertragungsdauer aus. Zwischen *TextMessage*, *BytesMessage*, *StreamMessage* und *ObjectMessage* sind bei Nachrichten mit einem Inhalt mit weniger als 1000 Zeichen nahezu keine Unterschiede messbar. Lediglich der Typ *MapMessage* zeigt 13% größerer Medianwerte bei der Übertragungsdauer. Mit steigender Nachrichtenlänge sind unterschiedliche Tendenzen feststellbar. Bezogen auf 100000 Zeichen pro Nachrichteninhalt verdreifacht sich der Median der Übertragungsdauer beim Typ *TextMessage* im Vergleich zu Nachrichten mit einem Zeichen. Bei *MapMessage* beträgt der Faktor 3,4; *StreamMessage* und *ObjectMessage* weisen hierfür jeweils den Faktor 1,4 auf. Die Messergebnisse für *ObjectMessage* sind jedoch stark von der Struktur des Objekts abhängig, das in einer Nachricht versendet werden soll, da komplexe Objekte einen höheren Aufwand erfordern, beispielsweise für die Serialisierung. Für eine Vergleichbarkeit mit den anderen Nachrichtentypen wurden die Messungen für einfache Objekte durchgeführt, die nur eine Zeichenkette enthalten. Bei Nachrichten, die aus Zeichenketten mit weniger als 10000 Zeichen bestehen, zeigen die Verteilungen der Messergebnisse weitgehend zwei Häufungsintervalle, die charakteristisch für die nicht-persistente Übertragung sind. Bei längeren Nachrichten ist meist nur noch ein Häufungsbereich um den Median herum vorhanden, beim Typ *BytesMessage* ist dies jedoch bereits ab 1000 Zeichen pro Nachricht der Fall. Bei einer Größe von 100000 Zeichen existieren beim Typ *ObjectMessage* erneut zwei deutlich getrennte Häufungsbereiche.
Zusammenfassend beurteilt, weisen die gemessenen Werte darauf hin, dass sich der Nachrichtentyp erst bei längeren Nachrichten bemerkbar macht. Diese Messungen basieren jedoch auf einer Zeichenkette als spezifischer Nachrichteninhalt, sodass einzig für den Typ *TextMessage* aussagekräftige Schlüsse gezogen werden können, da dieser auf Zeichenketten als Inhalt beschränkt ist. Für die übrigen Nachrichtentypen sind hingegen weitere Untersuchungen erforderlich, um den generellen Einfluss des Nachrichtentyps und des Inhalts auf die Übertragungsdauer bestimmen zu können. Für den Entwurf der Modellkonstrukte in Abschnitt 5.3 wird somit lediglich der Nachrichtentyp *TextMessage* betrachtet.

Queue mit entferntem Nachrichtensystem

Diese Konfiguration sieht den Nachrichtenversand über ein entferntes Nachrichtensystem vor. Als Nachrichtenkanal kommt eine `Queue` mit automatischem Bestätigungsmechanismus zum Einsatz. Jede Nachricht wird persistiert und ist vom Typ *TextMessage*. Es wurden Messungen für Inhalte mit Zeichketten durchgeführt, die aus 1, 10, 100, 1000, 10000 und 100000 zufälligen Zeichen bestehen. Die Konfiguration entspricht ebenfalls keiner der JMS-Optionen aus Abschnitt 5.2.1, jedoch ist bei entferntem Nachrichtensystem zusätzlicher Aufwand zu erwarten, so dass dieser Performance-Einfluss evaluiert wird.
Die Übertragungsdauer von Nachrichten ist deutlich größer, wenn das verwendete Nachrichtensystem sich auf einem entfernten System befindet. Auf Grundlage der Medianwerte der verwendeten Testumgebung hat der Faktor einen Wert von drei, wenn man einen Vergleich des Medians dieser Konfiguration bei einer Zeichenkettenlänge von 10 und jenem der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ heranzieht. Dies ist unter anderem darin begründet, dass über eine Netzwerkverbindung kommuniziert werden muss, was mit zusätzlichem Aufwand verbunden ist. Die Länge des Inhalts spielt ab 1000 Zeichen eine signifikante Rolle (siehe Tabelle 5.6), bei 100000 Zeichen ist die Übertragungsdauer fast sechs mal so lang als bei einem Zeichen als Nachrichteninhalt. Die erforderliche Kommunikation über eine Netzwerkverbindung ist hierfür ebenfalls von Bedeutung, entscheidend sind aber vermutlich zusätzlich Implementierungsdetails des Nachrichtensystems. Die Häufigkeitsverteilungen der Messreihen für unterschiedliche Nachrichtengrößen ähneln sich sehr stark.
Insgesamt ist festzustellen, dass sich die Verwendung eines entfernten Nachrichtensystems in einer deutlich höheren Übertragungsdauer auswirkt. Durch einen großen Nachrichteninhalt wird diese noch zusätzlich erhöht. Dies ist jedoch nicht zwangsläufig auf das Nachrichtensystem zurückzuführen, da hierfür zunächst der Einfluss des Netzwerks eindeutig bestimmt werden müsste.

Anzahl der Zeichen als Nachrichteninhalt	**Median**	**Verhältnis zu einem Inhaltszeichen**
1	5077956ns	-
10	5111247ns	1,01
100	5192908ns	1,02
1000	5461432ns	1,08
10000	8156925ns	1,61
100000	29213002ns	5,75

Tabelle 5.6: Median der Übertragungsdauer beim Nachrichtenversand über den Nachrichtenkanal `Queue` mit entferntem Nachrichtensystem

Queue mit entferntem Empfänger

Die Performance-Auswirkungen des Nachrichtenversands an entfernte Empfänger wird in dieser Konfiguration bestimmt. Sie ähnelt weitgehend der Konfiguration „Queue mit entferntem Nachrichtensystem“, jedoch befindet sich hier der Nachrichtenempfänger auf einem entfernten System. Dieser schickt nach dem Nachrichtenempfang eine Antwortnachricht über einen eigenen Nachrichtenkanal an das System

des ursprünglichen Senders zurück. Die Übertragungsdauer ist in diesem Fall die Zeitdauer zwischen Versand der Nachricht an den entfernten Empfänger und dem Empfang von dessen Antwortnachricht. Davon wird die Zeit, die der Empfänger zwischen dem Nachrichtenempfang und dem Versand der Antwortnachricht benötigt, abgezogen. Die Übertragungsdauer wurde für Nachrichten mit unterschiedlich großem Inhalt gemessen. Als Nachrichteninhalt wurde Zeichenketten mit 1, 10, 100, 1000, 10000 und 100000 zufälligen Zeichen gewählt.
Beim Versand von Nachrichten zu einem entfernten Empfänger und zurück, sind erst ab einem Inhalt mit einem Umfang von mehr als 1000 Zeichen signifikante Unterschiede bei der Übertragungsdauer festzustellen (siehe Tabelle 5.7). Dies ist dadurch zu erklären, dass ab dieser Nachrichtenlänge eine Segmentierung der zu übertragenden Daten erfolgt, da die Datenrahmen von Ethernet auf eine maximale Nutzdatengröße von 1500 Bytes beschränkt sind. Als Folge müssen die zu übertragenden Daten der Nachrichten segmentiert, mittels mehrerer Ethernet-Datenrahmen übertragen und wieder reassembliert werden. Die in der Tabelle aufgeführten Werte der Mediane für die Übertragungsdauer der Nachrichten können mit denen der anderen Konfigurationen verglichen werden, wenn sie halbiert werden. Sie entsprechen dann ungefähr der Dauer vom Nachrichtenversand bis zum Empfang der Nachricht, anstatt der Dauer für die Hin-und Rücksendung einer Nachricht zwischen Sender und Empfänger. Vergleicht man den Median für einen Inhalt bestehend aus 10 Zeichen mit jenem der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“, so zeigt sich dass die Übertragungsdauer beträchtlich größer ist, der Faktor beträgt $40{,}6ms \ / \ (2 * 1{,}7ms) \approx 11{,}9$. Die überwiegende Mehrheit der Messwerte gruppiert sich bei allen Zeichenkettenlängen in einem breiten Streifen um den Median, wobei die sich Breite des Streifens mit der Länge der Zeichenkette vergrößert.
Zusammenfassend lässt sich feststellen, dass die Nachrichtenübertragung zwischen verteilten Kommunikationspartnern mit einer deutlich größeren Übertragungsdauer verbunden ist, als wenn diese lokal via Nachrichten kommunizieren. Zudem spielt die Größe des Nachrichteninhalts eine Rolle. Analog zur Konfiguration „Queue mit entferntem Nachrichtensystem“ müssten genaue Untersuchungen des Netzwerkverkehrs angestellt werden, um für das Nachrichtensystem und das Netzwerk jeweils ihren Anteil an der Verzögerung bestimmen zu können.

Anzahl der Zeichen als Nachrichteninhalt	**Median**	**Verhältnis zu einem Inhaltszeichen**
1	$40971428ns$	-
10	$40557864ns$	0,99
100	$40801943ns$	1,00
1000	$41285136ns$	1,01
10000	$44075089ns$	1,08
100000	$70953438ns$	1,73

Tabelle 5.7: Median der Übertragungsdauer beim Nachrichtenversand über den Nachrichtenkanal `Queue` mit entferntem Empfänger

Topic mit dauerhafter Empfängerregistrierung

Bei dieser Konfiguration erfolgt der Nachrichtenversand über ein `Topic` als Nachrichtenkanal mit automatischem Bestätigungsmechanismus. Jede Nachricht wird gemäß

den Standardeinstellungen von JMS persistent gespeichert. Die Zeichenkette des Nachrichteninhalts hat eine Länge von 5 Zeichen. Die Warteperiode zwischen zwei Nachrichteninitiierungen beträgt $50ms$. Die Nachrichtenempfänger sind dauerhaft am Nachrichtenkanal registriert.
Die Messungen der Übertragungsdauer wurden ausschließlich bei Empfängern durchgeführt, die ihre Verbindung zum Nachrichtensystem nicht unterbrechen. Eine Unterbrechung der Verbindung eines Empfängers zum Nachrichtensystem, seiner erneuten Registrierung und der Zustellung aller Nachrichten, die innerhalb des Unterbrechungszeitraums versendet wurden, verlängert die Übertragungsdauer einer Nachricht mindestens um die Dauer der Unterbrechung. Dieser Fall wird nicht in die Messungen einbezogen, um die Analyse des Aufwands für die dauerhafte Registrierung nicht zu verfälschen.
Beim Nachrichtenversand über ein `Topic` sind ähnliche Messwerte für die Übertragungsdauer von Nachrichten festzustellen wie über eine `Queue`. Der Median dieser Konfiguration unterscheidet sich mit $1{,}77ms$ nur um etwa 3% von jenem der vergleichbaren Konfiguration „Queue mit automatischem Bestätigungsmechanismus“, der bei $1{,}72ms$ liegt. Die Häufigkeitsverteilung der Werte für die Übertragungsdauer weist eine deutliche Häufung der Messwerte im Intervall zwischen $1{,}70ms$ und $1{,}82ms$ auf, ungefähr 47% der Werte liegen in diesem Intervall (siehe Abbildung 5.18). Im Vergleich zur Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ zeigt sich eine etwas breitere Streuung der Übertragungsdauerwerte ohne deutliches Häufungsintervall. Insgesamt ergibt sich kein bedeutender Unterschied bezüglich der Übertragungsdauer zwischen den beiden Nachrichtenkanaltypen `Topic` und `Queue`.

Konfiguration	**Median**	**Verhältnis zu „Topic mit dauerhafter Empfängerregistrierung“**
Topic mit dauerhafter Empfängerregistrierung	$1770230ns$	-
Topic mit dauerhafter Empfängerregistrierung und nicht-persistenter Nachrichtenübertragung	$1271604ns$	0,72
Topic mit dauerhafter Empfängerregistrierung und automatischer Filterung	$1910017ns$	1,08
Topic ohne dauerhafte Empfängerregistrierung	$1415697ns$	0,80

Tabelle 5.8: Median der Übertragungsdauer für verschiedene Konfigurationen des Nachrichtenversands über den Nachrichtenkanal `Topic`

Topic mit dauerhafter Empfängerregistrierung und nicht-persistenter Nachrichtenübertragung

Diese Konfiguration entspricht weitgehend der zuvor beschriebenen Konfiguration „Topic mit dauerhafter Empfängerregistrierung“, jedoch werden die Nachrichten nicht auf der Festplatte persistent gespeichert. Durch den Vergleich dieser Konfiguration mit der Vorherigen kann bestimmt werden, wie sich die JMS-Option *JMS*

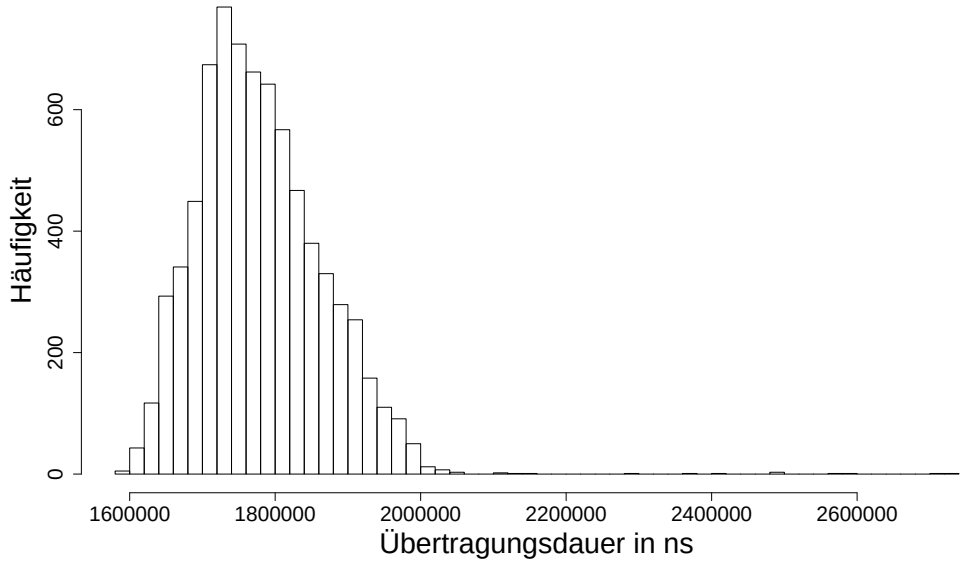


Abbildung 5.18: Histogramm der Übertragungsdauer für die Konfiguration „Topic mit dauerhafter Empfängerregistrierung“

Persistent Messages beim Nachrichtenkanal `Topic` auf die Performance auswirkt. Die nicht-persistente Nachrichtenübertragung über ein `Topic` als Nachrichtenkanal wirkt sich nahezu identisch auf die Übertragungsdauer aus wie bei der Übertragung über eine `Queue`. Im Vergleich zur persistenten Übertragung ist die Übertragungsdauer bezogen auf den Median um 28% kürzer (siehe Tabelle 5.8). Analog zur Konfiguration „Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“ sind bei der Verteilung der Werte für die Übertragungsdauer zwei deutliche Häufungsintervalle festzustellen. Diese Konfiguration bestätigt, dass die JMS-Option *JMS Persistent Messages* sowohl bei einer `Queue` als auch bei einem `Topic` mit einer deutlichen Erhöhung der Übertragungsdauer verbunden ist.

Topic mit dauerhafter Empfängerregistrierung und automatischer Filterung

Diese Konfiguration ist fast identisch mit der Konfiguration „Topic mit dauerhafter Empfängerregistrierung“, jedoch erfolgt vor dem Empfang für jede zugestellte Nachricht eine automatisch Überprüfung auf die Erfüllung eines Filterkriteriums. Hiermit wird für den Nachrichtenkanal `Topic` der Einfluss der JMS-Option *JMS Message Selector* auf die Übertragungsdauer ermittelt.
Die Filterung der Nachrichten auf Empfängerseite wirkt sich bei einem `Topic` auf die Übertragungsdauer mit einer geringen Verlängerung um 8% aus (siehe Vergleich der Mediane in Tabelle 5.8). Dieser Einfluss ist mit jenem bei einer `Queue` fast identisch, so dass der JMS-Option *JMS Message Selector* eine geringe Verlängerung der Übertragungsdauer zugeschrieben wird. Dieser Wert wird im Rahmen dieser Arbeit auf Grundlage der durchgeführten Messungen angenommen, er ist jedoch stark vom verwendeten Selektor abhängig, so dass der ermittelte Wert nicht als genereller Einflussfaktor betrachtet werden kann. Auf die Häufigkeitsverteilung der Werte für die Übertragungsdauer hat die Filterung vernachlässigbaren Einfluss.

Topic ohne dauerhafte Empfängerregistrierung

Der Nachrichtenempfang durch Empfänger, die nicht dauerhaft am Nachrichtenkanal registriert sind, unterscheidet diese Konfiguration von der Konfiguration „Topic mit dauerhafter Empfängerregistrierung“. Durch den Vergleich beider Konfigurationen kann der Performance-Einfluss der JMS-Option *JMS Durable Subscription* bestimmt werden.
Die mit der nicht-dauerhaften Registrierung von Nachrichtenempfängern verbundene Aufwand im Nachrichtensystem ist geringer als bei dauerhafter Registrierung (siehe Abschnitt 5.2.1). Wie die Messungen zeigen, resultiert dies auch in einer kürzeren Übertragungsdauer. Tabelle 5.8 ist zu entnehmen, dass sich die Übertragungsdauer um etwa 20% verringert, wenn die Empfänger sich nicht dauerhaft registrieren. Ungefähr 60% der Übertragungsdauerwerte befinden sich in einem Intervall zwischen 1,34*ms* und 1,44*ms*. Die übrigen Werte verteilen sich fast ausnahmslos auf die Spanne von 1,44*ms* bis 1,72*ms*. Insgesamt lässt sich aus den Messergebnissen schließen, dass die die JMS-Option *JMS Durable Subscription* eine signifikante Verlängerung der Übertragungsdauer bewirkt, die im Bereich von etwa 25% liegt.

5.2.3.3 Offene Fragen

Die Messergebnisse werfen einige Fragen auf, die im Rahmen dieser Arbeit nicht weiter behandelt werden, da diese vermutlich JMS-spezifische Details bzw. Implementierungsdetails des JMS-Nachrichtensystems betreffen. Folgende Phänomene sind bei der Analyse der Messergebnisse aufgetreten:

- Bei der Konfiguration „Queue mit automatischem Bestätigungsmechanismus“ ist ab der 5000. empfangenen Nachricht eine sprunghafte Verringerung der Übertragungsdauer von fast 10% zu messen.
- Alle Konfigurationen, bei denen keine persistente Speicherung der Nachrichten erfolgt, zeigt die Verteilung der Werte für die Übertragungsdauer zwei Häufungsbereiche.
- Bei der Konfiguration „Queue mit Empfänger-Pool“ mit Poolgrößen von mindestens 32 Empfängern liegen die Werte für die Übertragungsdauer auf unterschiedlichen, deutlich getrennten Niveaus, die große Mehrheit der Nachrichten weist jedoch eine Übertragungsdauer auf, die auf dem untersten Niveau liegt.
- Der Verlauf der Werte für die Übertragungsdauer der Nachrichten weist bei der Konfiguration „Queue mit entferntem Empfänger“ deutliche trapezförmige Muster auf. Innerhalb dieses Musters reihen sich diese Werte zum Teil entlang von Parallelen auf.

Für die Erklärung dieser Phänomene, ist eine gründliche Untersuchung des JMS-Nachrichtensystems und dessen Nutzung der Hardware-Ressourcen erforderlich. Die Frage, ob diese Phänomene spezifisch für das verwendete Nachrichtensystem `Sun Java System Message Queue 4.1` sind, lässt sich durch einen Vergleich unterschiedlicher JMS-Nachrichtensysteme beantworten.

5.2.3.4 Gruppierung der Ergebnisse

In Tabelle 5.9 sind die Auswirkungen der JMS-Optionen und weiterer Einflussfaktoren wie z. B. dem Nachrichtentyp oder der Nachrichtengröße auf die Übertragungsdauer zusammengefasst. Die Evaluierung der Messergebnisse zeigt, dass die JMS-Optionen *JMS Queue*, *JMS Topic* und *JMS Message Acknowledgment* keinen messbaren Einfluss auf die Übertragungsdauer von Nachrichten haben. Ein geringer Einfluss bedeutet hierbei eine Verringerung bzw. Verlängerung der Übertragungsdauer um weniger als 20%, da dies in etwa dem Toleranzbereich für Performance-Vorhersagen entspricht. Größere Abweichungen sind für Performance-Vorhersagen hingegen ausschlaggebend, so dass ein Einfluss von JMS-Optionen auf die Übertragungsdauer als signifikant eingestuft, wenn dieser sich im Bereich zwischen 20% und 100% bewegt. Alle Faktoren, die eine Vervielfachung der Übertragungsdauer verursachen, werden der Kategorie „großer Einfluss“ zugeordnet. Relevant für die Einordnung sind maximalen Auswirkungen. So wird beispielsweise der Nachrichtengröße ein großer Einfluss beigemessen, da sich die Übertragungsdauer bei einer Größe von 100000 Zeichen im Vergleich zu einer Nachrichtengröße von einem Zeichen verdreifacht.

JMS-Option	**kein Einfluss**	**geringer Einfluss**	**signifikanter Einfluss**	**großer Einfluss**
JMS Queue	X			
JMS Topic	X			
JMS Persistent Messages			X	
JMS Message Acknowledgment	X			
JMS Message Selector		X		
JMS Transacted Session				X
JMS Durable Subscription			X	
MDB-Pool				X
Nachrichtengröße				X
Nachrichtentyp				X
entfernter Empfänger				X
entferntes Nachrichtensystem				X

Tabelle 5.9: Klassifizierung der JMS-Optionen und weiterer Parameter nach ihrem Einflusses auf die Übertragungsdauer von Nachrichten

Zusammenfassend lässt sich auf Grundlage der Messergebnisse feststellen, dass die Nachrichtenkanaltypen `Queue` und `Topic` nahezu gleiche Werte für durchschnittliche Übertragungsdauer einer Nachricht erreichen. Sie werden dennoch beim folgenden Entwurf der Modellkonstrukte berücksichtigt, da sie unterschiedliche Häufungen der Übertragungsdauerwerte zeigen. Der verwendete Bestätigungsmechanismus wird hingegen nicht weiter verfolgt, da damit keine messbaren Auswirkungen auf die Performance festgestellt wurden. Über die JMS-Optionen „Nachrichtentyp“, „entferntes Nachrichtensystem“ und „entfernter Nachrichtenempfänger“ können keine eindeutigen Aussagen getroffen werden, da bei diesen mögliche zusätzliche Einflussfaktoren untersucht werden müssen. Beim Nachrichtentyp sind die Messwerte vermutlich stark vom Inhalt abhängig, was jedoch genauer analysiert werden muss. Bei einem entfernten Nachrichtensystem oder Nachrichtenempfänger ist es erforderlich den Einfluss des Netzwerks und dem Kommunikationsaufwand auf Seiten der Kommunikationsteilnehmer zu untersuchen, bevor die höheren Werte für die Übertragungsdauer eindeutig begründet werden können. Die übrigen JMS-Optionen wirken sich gering bis stark auf die Übertragungsdauer von Nachrichten aus, so dass sie für den Entwurf der Modellkonstrukte von Bedeutung sind.
Aufgrund der in Abschnitt 5.2.1 getroffenen Zuordnung von JMS-Optionen zu Entwurfsmustern für nachrichtenorientierte Kommunikation, können diese Performance-Auswirkungen der JMS-Optionen auf ihre entsprechenden Entwurfsmuster übertragen werden.

5.3 Entwurf der Modellkonstrukte

Vervollständigungen [WoPS02] sind ein generelles Mittel zur Erfassung von extrafunktionalen Performance-Aspekten, um alle erwarteten Elemente eines Entwurfs beschreiben und eine Spezifikation modifizieren zu können. Für die Integration von nachrichtenorientierter Kommunikation ins Palladio Komponentenmodell bietet sich eine Vervollständigung an, da die Performance-Auswirkungen der nachrichtenorientierten Kommunikation maßgeblich vom Einsatz eines Nachrichtensystems bestimmt werden, anstatt von der zu entwickelnden Anwendung. Darüber hinaus ermöglicht eine solche Vervollständigung, dass beim Entwurf einer Anwendung implizit eine Komponente einbezogen und konfiguriert wird, die das Nachrichtensystem realisiert. Der Entwurf dieser Vervollständigung erfolgt in diesem Abschnitt. Dafür ist es zunächst nötig, das Palladio Komponentenmodell um ein Annotationsmodell für nachrichtenorientierte Kommunikation zu erweitern. Damit kann die Vervollständigung mit den zuvor ermittelten Performance-relevanten Parametern konfiguriert werden. Anschließend wird der Entwurf der Vervollständigung selbst beschrieben. Abschließend wird eine Komponente entworfen, die die asynchrone Kommunikationsform eines Nachrichtensystems widerspiegelt und dessen Performance-Einfluss kapselt. Hierfür werden die Performance-Auswirkungen dieser Parameter in Form von Schablonen modelliert, sodass der Performance-Einfluss dieser Komponente entsprechend der Konfiguration der Vervollständigung angepasst werden kann.

5.3.1 Annotationsmodell

Für die zu entwerfende Vervollständigung sind die in Abschnitt 5.2 evaluierten Performance-relevanten Parameter von nachrichtenorientierter Kommunikation von zentraler Bedeutung, da sich ihr Entwurf nach diesen Parametern ausrichtet. In diesem

Unterabschnitt werden zunächst die Performance-relevanten Parameter strukturiert und miteinander in Beziehung gesetzt, um die erlaubten Parameterkombinationen bestimmen zu können. Ausgehend davon erfolgt der Entwurf eines Annotationsmodells, damit diese extra-funktionalen Performance-relevanten Parameter im Palladio Komponentenmodell erfasst werden können. Damit wird die Konfiguration der Vervollständigung für nachrichtenorientierte Kommunikation ermöglicht.

Die über die unterschiedlichen JMS-Optionen ermittelten Performance-relevanten Parameter für nachrichtenorientierte werden als *Feature*-Diagramm [CzEi00, S. 87] modelliert, wie es in in Abbildung 5.19 dargestellt ist. Durch diese Strukturierung lassen sich die erlaubten Parameterkonfigurationen bestimmen. Die Bezeichnung der Parameter entspricht dabei der Bezeichnung der JMS-Optionen in Abschnitt 5.2.1.

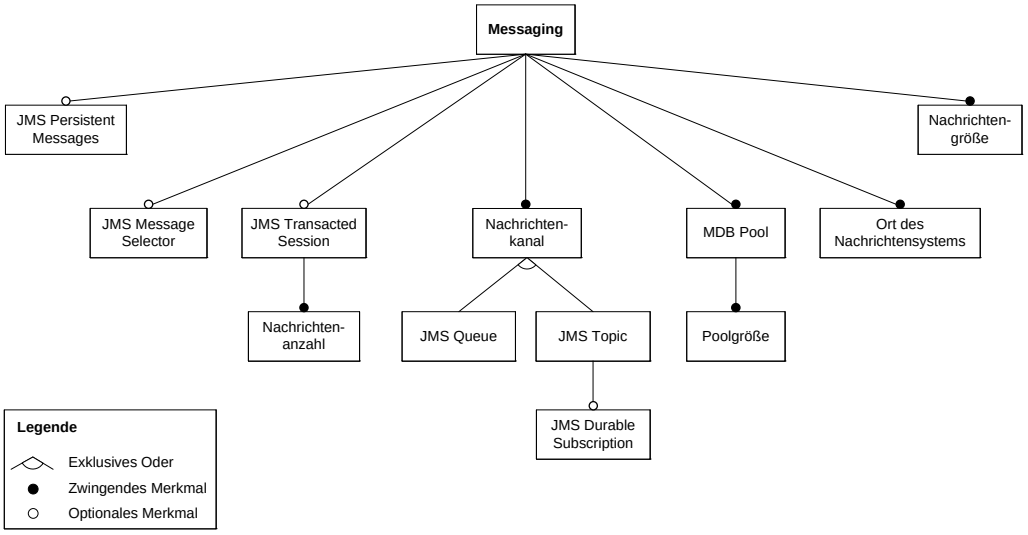


Abbildung 5.19: *Feature*-Diagramm für die Performance-relevanten Parameter von nachrichtenorientierter Kommunikation

Für die nachrichtenorientierte Kommunikation ist es zwingend erforderlich, den Typ des Nachrichtenkanals festlegen zu können. Für die Performance ist die Wahl des Nachrichtenkanaltyps ebenso zwingend erforderlich, da davon für einen Nachrichtenkanaltyp spezifische, Performance-relevante Parameter abhängen, obwohl die Wahl des Nachrichtenkanaltyps selbst keine signifikanten messbaren Auswirkungen auf die Performance hat. Von den untersuchten JMS-Optionen ist *JMS Durable Subscription*, dessen Verwendung die Übertragungsdauer von Nachrichten um ungefähr 25% verlängert, auf die Option *JMS Topic* beschränkt, denn die dauerhafte Registrierung eines Empfängers am Nachrichtensystem kann nur auf den Nachrichtenkanaltyp `Topic` angewendet werden. Erforderlich ist *JMS Durable Subscription* nicht, da ansonsten angenommen wird, dass der Empfänger nicht dauerhaft am Nachrichtensystem registriert ist. Die Optionen *JMS Queue* und *JMS Topic* schließen sich gegenseitig aus, da entsprechend der JMS-Spezifikation [Java02] ein einzelner Empfänger nur Nachrichten von einem Nachrichtenkanal empfangen kann.
Entsprechend den zuvor durchgeführten Leistungsbewertungen ist die Option *MDB Pool* und die Größe des Empfänger-Pools für die Übertragungsdauer von Nachrichten von großer Bedeutung. Java EE-konforme Anwendungs-Server verfügen über einen Empfänger-Pool für Nachrichten. Da diese Anwendungs-Server die in dieser

Arbeit betrachtete Infrastruktur für die Entwicklung von nebenläufiger, komponentenbasierter Software darstellen, wird die Option *MDB Pool* als zwingendes Merkmal eingestuft, um diesem technischen Sachverhalt Rechnung zu tragen. Die Festlegung der Poolgröße ist ebenfalls verpflichtend, da diese für die Performance-Auswirkungen der Option *MDB Pool* entscheidend ist. Die Option *MDB Pool* ist von keinen weiteren Optionen abhängig und ist deshalb als eigenständiges Merkmal im *Feature*-Diagramm modelliert.
Der Einsatz eines entfernten Nachrichtensystems wirkt sich verglichen mit der Verwendung eines lokalen Nachrichtensystems in einer erheblich längeren Übertragungsdauer der Nachrichten aus. Somit ist der Ort des Nachrichtensystems für die Performance relevant. Er wird als zwingendes Merkmal eingestuft, da die Konfiguration des Nachrichtensystemorts bei vielen Java EE-konforme Anwendungs-Servern durchgeführt werden kann. Wie die Option *MDB Pool* ist der Ort des Nachrichtensystems von allen anderen Merkmalen unabhängig, da hiermit die Verbindung zwischen Anwendungs-Server und Nachrichtensystem bestimmt wird, über welche der Nachrichtentransport erfolgt, ohne dass weitere Optionen darauf Einfluss nehmen können.
Ein ebenfalls verbindliches Merkmal stellt die Nachrichtengröße dar, denn sie wirkt sich deutlich auf die Performance des Nachrichtenversands aus. Verbindlich ist die Nachrichtengröße deshalb, weil nur damit verlässliche Aussagen über die Performance von nachrichtenorientierter Kommunikation getroffen werden können. Sie wird als unabhängig von anderen Optionen betrachtet, da im Rahmen dieser Arbeit die Nachrichtengröße auf den Nachrichteninhalt reduziert wird, welcher unabhängig von den übrigen Konfigurationen ist.
Eine Option, die auf jeden Nachrichtenkanaltyp angewendet werden kann und einen sehr großen Einfluss auf die Übertragungsdauer von Nachrichten hat, ist *JMS Transacted Session*. Mit ihr kann optional ein transaktionaler Versand von Nachrichten durchgeführt werden. Standardmäßig erfolgt der Nachrichtenversand allerdings nicht transaktional. Entsprechend dieses Verhaltens von JMS wird *JMS Transacted Session* als optionales Merkmal modelliert. Mit dieser Option können Transaktionsgrenzen in der Anwendung festgelegt werden, wodurch die Anzahl der Nachrichten einer Transaktion festgelegt werden. Somit wird die Nachrichtenanzahl beim Entwurf bestimmt, so dass sie ein zwingendes Merkmal für die Option *JMS Transacted Session* darstellt.
Die Verwendung der Option *JMS Message Selector* führt zu einer geringen Verlängerung der Übertragungsdauer von Nachrichten. Damit werden Nachrichtenempfänger optional mit einem Nachrichtenfilter ausgestattet, was unabhängig von Typ des Nachrichtenkanals oder sonstiger JMS-Optionen ist. Aus diesem Grund stellt *JMS Message Selector* ein optionales Performance-relevantes Merkmal für die nachrichtenorientierte Kommunikation dar.
Eine Verlängerung der Übertragungsdauer von Nachrichten um fast 40% wurde in Abschnitt 5.2.3.2 bei der Verwendung der nicht verbindlichen Option *JMS Persistent Messages* ermittelt, da hierbei die Nachrichten vom Nachrichtensystem persistiert werden. Diese Form der Erhöhung der Zuverlässigkeit kann bei beiden Nachrichtenkanaltypen eingesetzt werden und ist von allen anderen JMS-Optionen unabhängig. Dementsprechend ist die Option *JMS Persistent Messages* im *Feature*-Diagramm in Abbildung 5.19 modelliert.

Der Entwurf des Annotationsmodell ist an den zuvor beschriebenen Parameterkom-

binationen ausgerichtet. Für das Modell werden jedoch anstatt der Bezeichnungen der JMS-Optionen die Namen der Entwurfsmuster für nachrichtenorientierte Kommunikation verwendet, deren Zuordnung zu JMS-Optionen in Tabelle 5.1 zusammengefasst ist. Damit soll von JMS als konkrete nachrichtenorientierte Schnittstelle abstrahiert werden und die Anwendbarkeit der Entwurfsmuster erleichtert werden. Eine Annotation mit den Performance-relevanten Parametern von nachrichtenorientierter Kommunikation muss zudem mit bestehenden Elementen des PCM in Verbindung gebracht werden, damit die Verbindung spezifiziert werden kann, über welche Komponenten in einem bestimmten Kontext per Nachrichtenaustausch miteinander kommunizieren.

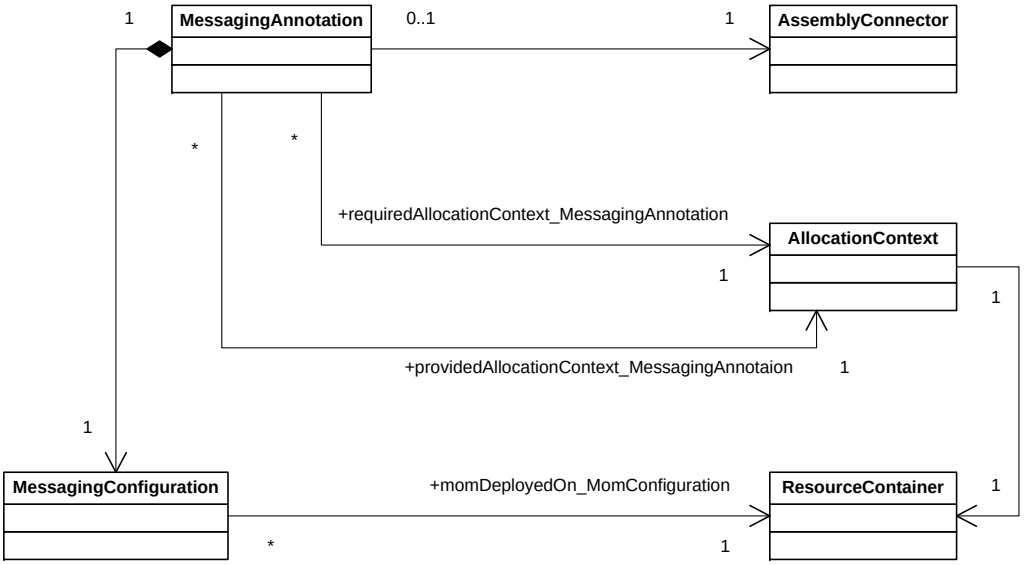


Abbildung 5.20: Klassendiagramm des Annotationsmodells für nachrichtenorientierte Kommunikation

In Abbildung 5.20 ist das Klassendiagramm des Annotationsmodells dargestellt. Von zentraler Bedeutung ist die Klasse `MessagingAnnotation`, da sie die Verbindung zwischen einer Parameterkonfiguration für nachrichtenorientierte Kommunikation und bestehenden Elementen des PCM, die in Abschnitt 3.3 beschrieben sind, herstellt. Der Entwurf des Annotationsmodells sieht die Verknüpfung von `MessagingAnnotation` mit der Klasse `AssemblyConnector` vor. Damit wird spezifiziert, auf welchen `AssemblyConnector` sich die Annotation bezieht und dass dieser die Kommunikation über den Austausch Nachrichten repräsentiert. Demzufolge muss eine `MessagingAnnotation` zwingend ein `AssemblyConnector` referenzieren, da ansonsten nicht festgestellt werden kann, auf welchen `AssemblyConnector` sie sich bezieht. Umgekehrt ist es nicht erforderlich, dass ein `AssemblyConnector` von einer `MessagingAnnotation` referenziert wird. Denn damit wird ausgedrückt, dass dieser `AssemblyConnector` einen gewöhnlichen Dienstaufruf repräsentiert, ohne dass mittels Nachrichtenaustausch kommuniziert wird.
Im vorherigen Unterkapitel wurde am Beispiel des Nachrichtenversands zu entfernten Empfängern festgestellt, dass die Allokation der Komponenten von großer Bedeutung für die Übertragungsdauer von Nachrichten ist. Das Annotationsmodell berücksichtigt dies, indem einer `MessagingAnnotation` ein `AllocationContext` zugeordnet wird. Dies ist für die eindeutige Identifizierung der Verbindung wichtig, da es für

die durch einen AssemblyConnector verbundenen Komponenten mehrere Kontexte geben kann. Der AllocationContext der Komponente, welche den Dienst anbietet, der mithilfe des Nachrichtenaustauschs in Anspruch genommen werden soll, wird mit der Rolle providedAllocationContext_MessagingAnnotation bezeichnet. Dem AllocationContext der Komponente, die den Dienst benötigt, wird die Rollenbezeichnung requiredAllocationContext_MessagingAnnotation zugeordnet. In beiden Fällen hat die Klasse AllocationContext eine Multiplizität von eins, da ein AllocationContext die Ressourcenallokation einer der beiden Komponenten darstellt, die per Nachrichtenaustausch miteinander kommunizieren. Hingegen kann ein AllocationContext von keinem oder mehreren MessagingAnnotation referenziert werden, da die dadurch allokierten Komponenten nicht zwangsläufig die nachrichtenorientierte Kommunikationsform in Anspruch nehmen müssen bzw. mit mehr als einer weiteren Komponente mittels Nachrichtenaustausch kommunizieren können. Über die bereits im PCM existierende Assoziation zwischen den Klassen AllocationContext und ResourceContainer kann somit bestimmt werden, ob in einer Systemumgebung die Senderkomponente dem selben System zugeteilt wurde wie die Empfängerkomponente. Dies ist von Bedeutung, da im verteilten Fall die Nachrichtenübertragungsdauer von der Last bzw. vom Durchsatz des Netzwerks beeinflusst wird.
Die Klasse MessagingAnnotation enthält genau eine MessagingConfiguration, in der die Parameterkonfiguration für die nachrichtenorientierte Kommunikation gekapselt ist. Wie zuvor beschrieben ist es für die Übertragungsdauer bedeutend, auf welchem Ressourcen-Container das Nachrichtensystem allokiert ist. Bei einem entfernten Nachrichtensystem sind die kommunizierenden Komponenten einem anderen Ressourcen-Container zugeordnet als das Nachrichtensystem. Über die Assoziation von einer MessagingConfiguration zu einem ResourceContainer wird festgelegt, wo das Nachrichtensystem in einer Systemumgebung allokiert werden soll. Über die beiden Assoziationen zwischen MessagingAnnotation und AllocationContext kann festgestellt werden, ob das Nachrichtensystem dem selben Ressourcen-Container zugeteilt ist oder ob es sich um ein entferntes Nachrichtensystem handelt. Im zweiten Fall wird die Übertragungsdauer von Nachrichten durch die Netzwerkverbindung zwischen den Ressourcen-Containern beeinflusst.

Die detaillierte Modellierung der Klasse MessagingConfiguration ist in Abbildung 5.21 dargestellt und umfasst die weiteren Performance-relevanten Parameter, die im *Feature*-Diagramm in Abbildung 5.19 aufgeführt sind. Die Attribute GuaranteedDelivery, SelectiveConsumer und MessageSize entsprechen den Parametern JMS Persistent Messages, JMS Message Selector bzw. Nachrichtengröße. Der Grund für die Modellierung dieser Merkmale als Attribute ist, dass diese Merkmale keine weiteren abhängigen Merkmale haben. Mithilfe des Datentyps Boolean wird erreicht, dass die Attribute GuaranteedDelivery und SelectiveConsumer optional sind, so dass sie nur dann Berücksichtigung finden, wenn ihnen der Wahrheitswert true zugewiesen wird. Die Spezifizierung der Nachrichtengröße ist verbindlich und erfolgt über die Zuweisung eines Zahlenwerts zum Attribut MessageSize. Die Alternative, dass die Nachrichtengröße als Methodenparameter modelliert wird, ist nicht anwendbar, da aufgrund der Modellierung des asynchronen Kommunikation als Erzeuger-Verbraucher-System keine Methodenparameter übergeben werden können (sieh Abschnitte 5.3.2 und 5.3.3).

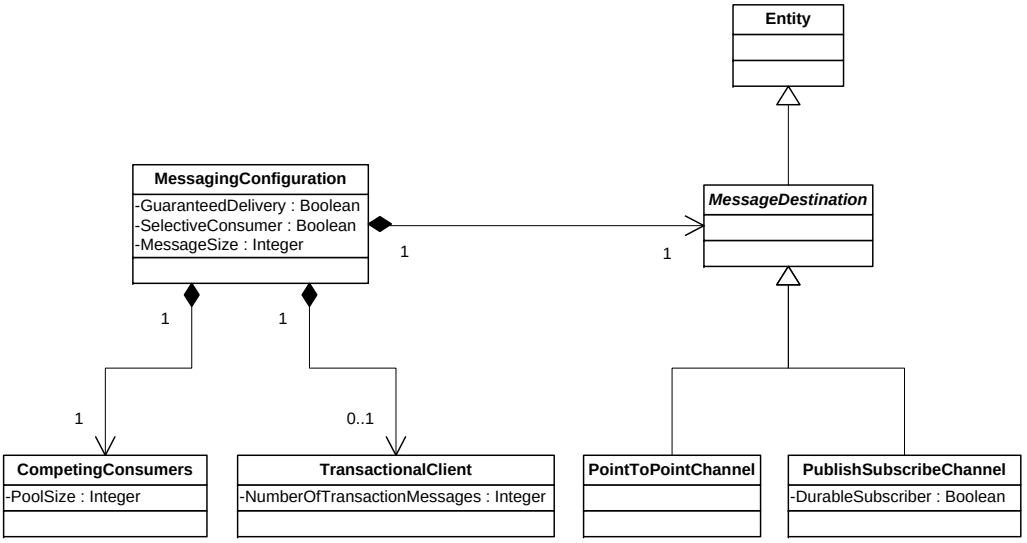


Abbildung 5.21: detaillierte Modellierung der Klasse `MessagingConfiguration` des Annotationsmodell für nachrichtenorientierte Kommunikation

Die Modellierung des verbindlichen Parameters `MDB Pool` erfolgt über die Komposition der Klassen `MessagingConfiguration` und `CompetingConsumers`. Das Attribut `PoolSize` dient der Spezifizierung der Anzahl konkurrierender Empfänger im Empfänger-Pool.
Der optionale Parameter `JMS Transacted Session` wird dadurch ausgedrückt, dass eine `MessagingConfiguration` ein `TransactionalClient` enthalten kann. Über dessen Attribut `NumberOfTransactionMessages` kann die Anzahl der Nachrichten in einer Transaktion bestimmt werden, welches der entscheidenden Faktor für die Übertragungsdauer der Nachrichten ist.
Die Auswahl des Nachrichtenkanaltyps wird über die Komposition zwischen den Klassen `MessagingConfiguration` und `Message Destination` erreicht. Demnach kann eine `MessagingConfiguration` genau eine `Message Destination` enthalten. Die Klasse `Message Destination` ist ihrerseits eine abstrakte Oberklasse der Klassen `PointToPointChannel` und `PublishSubscribeChannel`. Somit kann entweder ein `PointToPointChannel` oder `PublishSubscribeChannel` in einer `MessagingConfiguration` enthalten sein. Diese Auswahl ist essenziell, da nur ein `PublishSubscribeChannel` das Attribut `DurableSubscriber` besitzt, über welches die Performance-relevante dauerhafte Registrierung von Nachrichtenempfängern ausgewählt werden kann. Die Vererbungsbeziehung zwischen der Klasse `Entity` und der Klasse `Message Destination` erlaubt die Zuweisung eines eindeutigen Identifikators und eines Namens zu einem Nachrichtenkanaltyp. Der Grund hierfür ist, dass die Bezeichnung der Nachrichtenkanäle beim Software-Entwurf erfasst werden kann. Dadurch können Probleme in Zusammenhang mit der Bezeichnung bei der Implementierung sowie der Konfiguration des Anwendungs-Servers vermieden werden.

Mit den bisher vorgestellten Teilen des Annotationsmodell ist es möglich eine Kommunikationsbeziehung zwischen zwei Kontext-Komponenten als nachrichtenorientierte Kommunikation samt den Performance-relevanten Parametern zu spezifizieren. Abbildung 5.22 zeigt die Modellierung der Sammlung von mehreren Annotationen in einem `AnnotationRepository`, damit mehrere Annotationsinstanzen spezifiziert und zentral gebündelt werden können. Dies bietet Vorteile für die in Abschnitt 5.4

beschriebene Transformation der Vervollständigungen, da über die Modellinstanz des `AnnotationRepository` alle enthaltenen Annotationen für die Transformation zugänglich sind. Die Komposition zwischen der Klasse `AnnotationRepository` und der abstrakten Klasse `Annotation` bietet die Möglichkeit, verschiedene Ausprägungen von Annotationen in einem `AnnotationRepository` zusammenzufassen. Mithilfe dieser Modellerweiterung können in das PCM weitere konkrete Annotationsmodelle flexibel integriert werden. Die Vererbungsbeziehung zwischen der Klasse `Entity` und der Klasse `Annotation` ermöglicht die Bezeichnung von konkreten Annotationsinstanzen und die Zuweisung eines eindeutigen Identifikators.

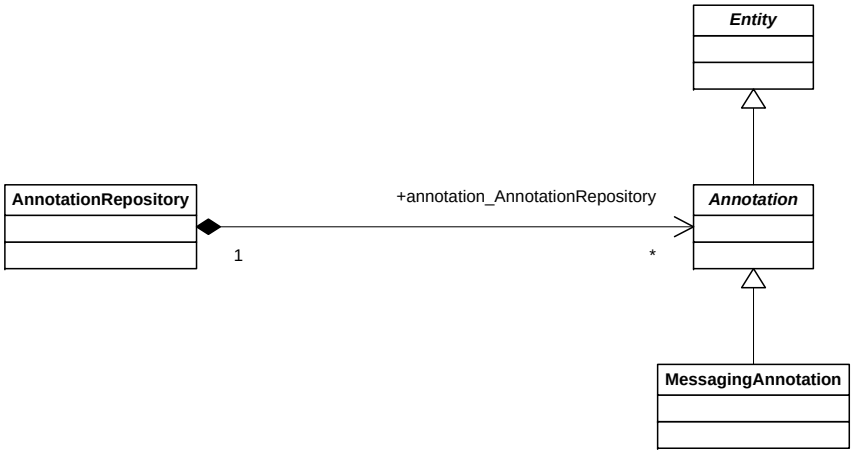


Abbildung 5.22: Klassendiagramm für die Aufnahme der `MessagingAnnotation` in einem `AnnotationRepository`

5.3.2 Vervollständigung

Mit dem im vorherigen Unterabschnitt beschriebenen Annotationsmodell kann eine Kommunikationsbeziehung zwischen Komponenten als nachrichtenorientierte Kommunikation spezifiziert und mit den zugehörigen Performance-relevanten Parameter konfiguriert werden. Die Modifikation eines Software-Entwurfs entsprechend einer oder mehrerer Instanzen dieses Annotationsmodell erfolgt über eine Vervollständigung (siehe [WoPS02]) für nachrichtenorientierte Kommunikation, die im Folgenden detailliert beschrieben wird.

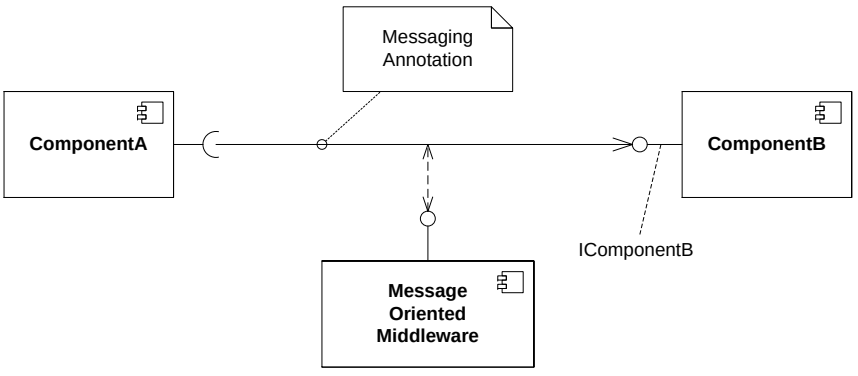


Abbildung 5.23: Problemszenario beim Einsatz von nachrichtenorientierter Komponentenkommunikation

Abbildung 5.23 zeigt ein Beispiel für die Kommunikation per Nachrichtenaustausch zwischen den Komponenten `ComponentA` und `ComponentB`. Die Annotation für die Konfiguration der Performance-relevanten Parameter ist zur Vereinfachung nur schematisch dargestellt, da zunächst von konkreten Parameterkonfigurationen abstrahiert werden kann. Für den Nachrichtenaustausch zwischen beiden Komponenten wird die Komponente `MessageOrientedMiddleware` benötigt, die ein Nachrichtensystem repräsentiert und dessen Verhalten kapselt. Hier tritt jedoch das Problem auf, dass die Verwendung der Komponente für das Nachrichtensystem beim Software-Entwurf transparent sein soll. Eine Einbeziehung dieser Komponente, würde die Verringerung des Abstraktionsgrads bedeuten und die Komplexität des Entwurfs erhöhen.

Das Problem der transparenten Verwendung des Nachrichtensystems kann mit einer Vervollständigung für nachrichtenorientierte Kommunikation gelöst werden, indem diese die Verwendung der Komponente für das Nachrichtensystem kapselt und mittels Transformation anstelle der direkten Kommunikationsbeziehung zwischen den Schnittstellen zweier Komponenten eingesetzt werden kann. Die Vervollständigung benötigt die Dienste eines Nachrichtensystems. In dieser Arbeit wird die Komponente `MessageOrientedMiddleware` als konkreter Repräsentant eines Nachrichtensystems verwendet, die in 5.3.3 genauer beschrieben ist. Der Vorteil dieser Trennung der Vervollständigung von der Nachrichtensystemkomponente ist, dass letztere leicht ausgetauscht werden kann, ohne dass die Vervollständigung grundlegend geändert werden müsste.

Das Hauptmerkmal der nachrichtenorientierten Kommunikation ist, dass die Kommunikation zwischen der Senderkomponente und der Empfängerkomponente asynchron erfolgt. Dieses Verhalten muss bei der Modellierung der Vervollständigung berücksichtigt werden. Das PCM unterstützt jedoch nativ keine asynchronen Dienstaufrufe, so dass sich die Modellierung der asynchronen Kommunikation als problematisch erweist. Dieses Problem kann umgangen werden, indem die Vervollständigung als Erzeuger-Verbraucher-System modelliert wird. Dazu umfasst sie die Basiskomponenten `MessageSenderAdapter` und `MessageReceiverAdapter`, die indirekt über eine Komponente `MessageOrientedMiddleware` miteinander kommunizieren (siehe Abbildung 5.24). Der `MessageSenderAdapter` stellt dabei den Erzeuger von Nachrichten dar, der `MessageReceiverAdapter` ist der Verbraucher. Die Komponente `MessageOrientedMiddleware` wird als Warteschlange für Nachrichten betrachtet. Das Sequenzdiagramm in Abbildung 5.25 zeigt den Kommunikationsverlauf, bei dem die Komponente `MessageSenderAdapter` synchron den Dienst `publishMessage` der Komponente `MessageOrientedMiddleware` zum Versenden einer Nachricht aufruft. Letztere nimmt die Nachricht entgegen und der Kontrollfluss kehrt sofort zur Komponente `MessageSenderAdapter` zurück. Die Komponente `MessageReceiverAdapter` führt proaktiv den Nachrichtenempfang durch, indem sie den Dienst `deliverMessage` auf der Komponente `MessageOrientedMiddleware` aufruft. Dabei übergibt die Komponente `MessageOrientedMiddleware` Nachrichten an die Komponente `MessageReceiverAdapter`, die sie selbst von der Komponente `MessageSenderAdapter` entgegengenommen hat. Die Entkopplung des `MessageSenderAdapter` vom `MessageReceiverAdapter` erfolgt somit mithilfe der Komponente `MessageOrientedMiddleware`. Als Folge dieser Modellierung ist es erforderlich, dass ein Verwendungsszenario erstellt werden muss, das den Empfangsprozess

über die Komponente `MessageReceiverAdapter` startet. Im Sequenzdiagramm ist dies über den externen Aufruf `onMessage` auf dieser Komponente angedeutet. Für den Nachrichtenversand muss kein spezielles Verwendungsszenario erstellt werden, da die Komponente `MessageSenderAdapter` von der Komponente aufgerufen wird, die Nachrichten versenden möchte. Insgesamt ermöglicht diese Modellierung, dass der Nachrichtenversand und -empfang vollständig unabhängig voneinander ausgeführt werden können.

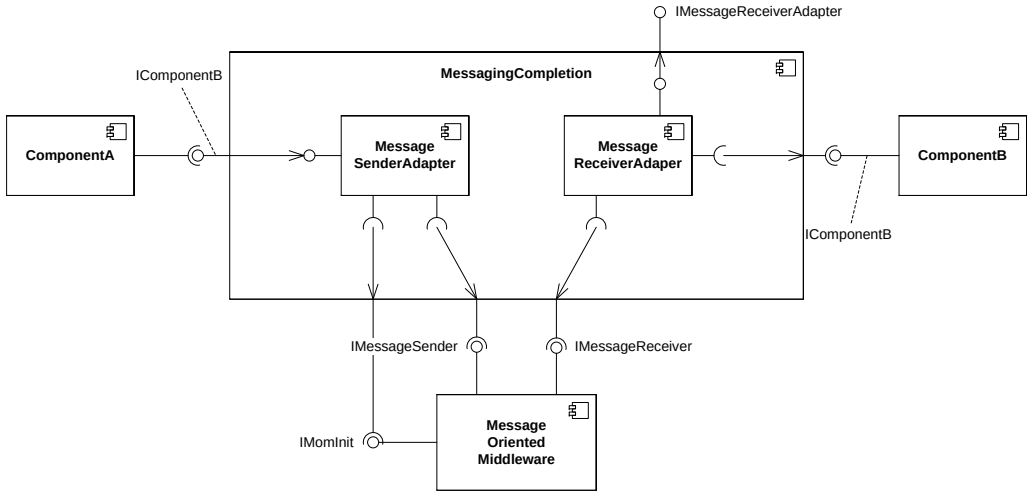


Abbildung 5.24: Modell der Vervollständigung

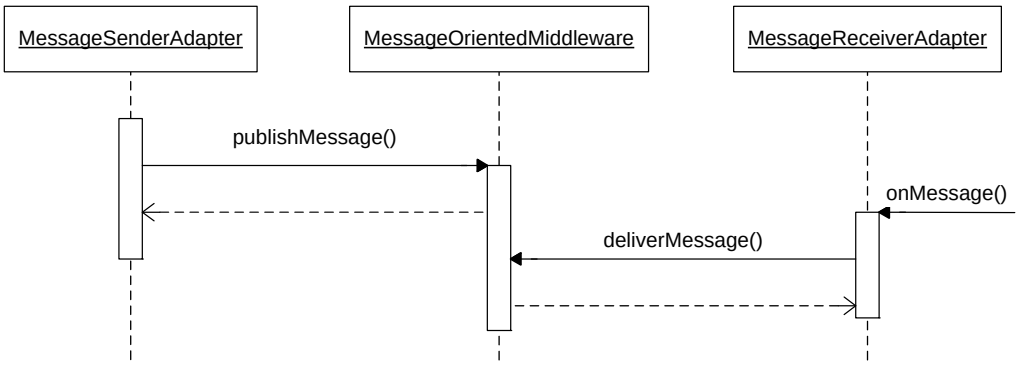
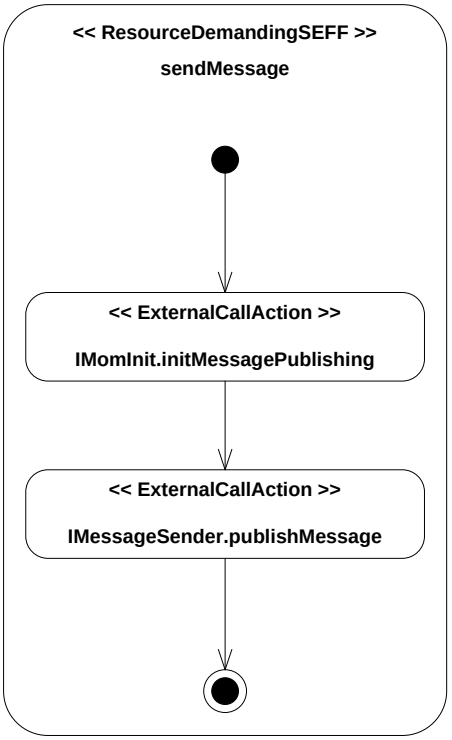


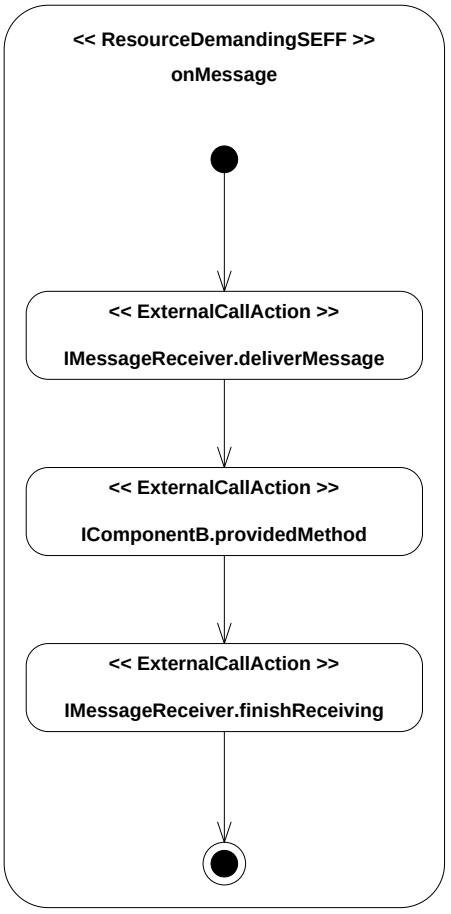
Abbildung 5.25: Sequenzdiagramm der Vervollständigung für den Versand und Empfang von Nachrichten

Das Komponentendiagramm in Abbildung 5.24 zeigt die Modellierung der Vervollständigung als zusammengesetzte Komponente. Diese umfasst die Basiskomponenten `MessageSenderAdapter` und `MessageReceiverAdapter`. Die Vervollständigung fungiert als Adapter für die Komponente `ComponentB` und bietet dementsprechend einen Dienst über die Schnittstelle `IComponentB` an. Aufrufe der Komponente `ComponentA` auf dieser Schnittstelle werden an die Komponente `MessageSenderAdapter` delegiert und deren Dienst `sendMessage` aufgerufen. Abbildung 5.26(a) zeigt anhand des SEFF das Verhalten dieses Dienstes. Die Komponente `MessageSenderAdapter` ruft demnach zunächst auf der Schnittstelle der Komponente `MessageOriented-Middleware` den Dienst `initMessagePublishing` auf, um das Nachrichtensystem für die Nachrichtenversand zu initialisieren. In der zweiten Aktion erfolgt der Nachrichtenversand über den externen Aufruf des Dienstes `publishMessage`, der über die

Schnittstelle `IMessageSender` von der Komponente `MessageOrientedMiddleware` angeboten wird. Den Dienst des Empfangsprozesses von Nachrichten bietet die Vervollständigung über die Schnittstelle `IMessageReceiverAdapter` an. Das Verwendungsszenario für den Nachrichtenempfang kann den Dienst über diese Schnittstelle in Anspruch nehmen. Die Aufrufe auf dieser Schnittstelle werden an die Komponente `MessageReceiverAdapter` weitergeleitet. Dabei wird der Dienst `onMessage` dieser Komponente aufgerufen, dessen Verhalten durch den in Abbildung 5.26(b) dargestellten SEFF charakterisiert wird. Die Komponente ruf den Dienst `deliverMessage` auf der Schnittstelle `IMessageReceiver` der Komponente `MessageOrientedMiddleware` auf, um darüber Nachrichten zu empfangen. In der sich anschließenden Aktion erfolgt ein Dienstaufruf auf der Schnittstelle der Komponente `ComponentB`, die das Ziel der nachrichtenorientierten Kommunikation darstellt. Dabei ist der aufgerufene Dienst genau jener, den die Komponente `ComponentA` benötigt. Die letzte Aktion führt erneut einen Aufruf auf der Schnittstelle `IMessageReceiver` der Komponente `MessageOrientedMiddleware` aus. Dieses Mal wird jedoch der `finishReceiving` in Anspruch genommen wird, um der Komponente `MessageOrientedMiddleware` den Abschluss des Empfangsprozesses mitzuteilen.



(a) SEFF der Komponente `MessageSenderAdapter` für die Initiierung des Nachrichtenversands



(b) SEFF der Komponente `MessageReceiverAdapter` für den Empfang von Nachrichten

Abbildung 5.26: Verhaltensspezifikationen für die Basiskomponenten der Vervollständigung

In Abbildung 5.27 ist der komplette Ablauf der nachrichtenorientierten Kommunikation am Beispiel der Komponenten `ComponentA` und `ComponentB` dargestellt. Dazu ist zu bemerken, dass das Verwendungsszenario zum Nachrichtenempfang jederzeit den Dienst `onMessage` aufrufen kann und der Aufruf nicht zwingend dann erfolgen muss, wenn über den Dienst `publishMessage` bereits Nachrichtenversendungen stattgefunden haben. Sollte dies nicht der Fall sein, würde der Dienst `onMessage` so lange blockieren, bis der Dienst `publishMessage` aufgerufen wurde. Auf der der anderen Seite kann die Komponente `MessageSenderAdapter` den Dienst `publishMessage` mehrmals aufrufen, unabhängig davon, ob ein Nachrichtenempfang über den Dienst `deliverMessage` erfolgt oder nicht.

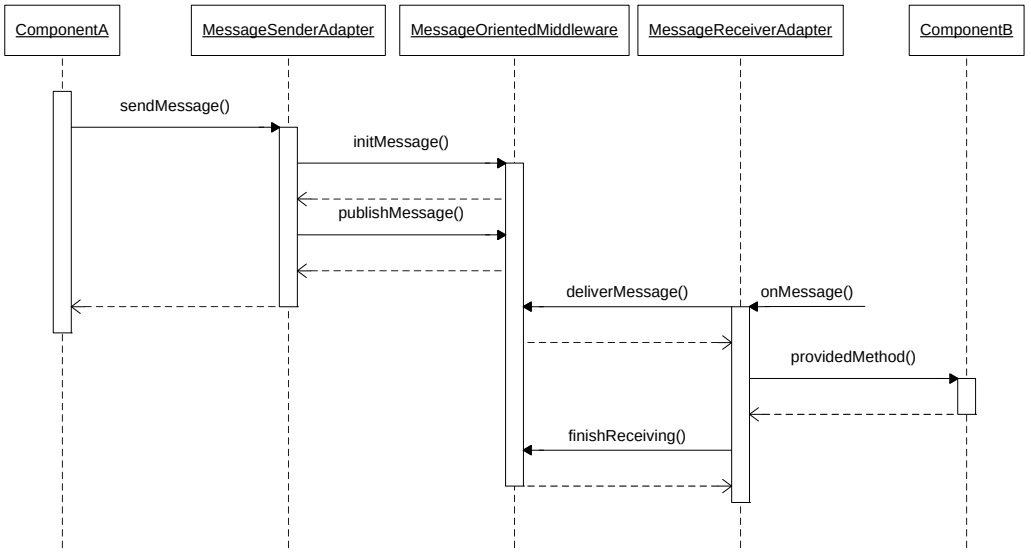


Abbildung 5.27: Sequenzdiagramm der nachrichtenorientierten Kommunikation unter Verwendung der Vervollständigung

Insgesamt ermöglicht diese Modellierung der Vervollständigung, dass der Versand und Empfang von Nachrichten über die Basiskomponenten `MessageSenderAdapter` bzw. `MessageReceiverAdapter` asynchron zueinander ablaufen können. Erreicht wird dies vor allem durch das spezielle Verwendungsszenario für den Nachrichtenempfang und die Entkopplung der beiden Komponenten mithilfe der Komponente `MessageOrientedMiddleware`. Diese wird bisher als Warteschlange für Nachrichten betrachtet. Das genaues Verhalten dieser Komponente wird im folgenden Abschnitt beschrieben.

5.3.3 Modellierung eines Nachrichtensystems

Bisher wurde die Komponente `MessageOrientedMiddleware`, die das Nachrichtensystem repräsentiert, lediglich als Warteschlange betrachtet, der Nachrichten hinzugefügt bzw. entnommen werden können. Die Modellierung dieser Komponente muss jedoch nicht nur dies ermöglichen, sondern auch das Verhalten eines Nachrichtensystems möglichst gut abbilden. Dafür ist es zum Beispiel erforderlich, den Ressourcenbedarf entsprechend der Performance-relevanten Parameter zu berücksichtigen. Entsprechend diesen Anforderungen wird im Folgenden ein detaillierter Entwurf dieser Komponente vorgestellt. Die Modellierung des Ressourcenbedarfs basiert auf den

Messergebnissen aus Abschnitt 5.2.3.2, so dass das Modell die dafür verwendete spezifische Kombination aus Nachrichtensystem und Rechensystem repräsentiert. Dies stellt jedoch keine große Einschränkung dar, denn der Ressourcenbedarf lässt sich im Modell sehr leicht an andere Kombinationen aus Nachrichtensystem und Rechensystem anpassen.

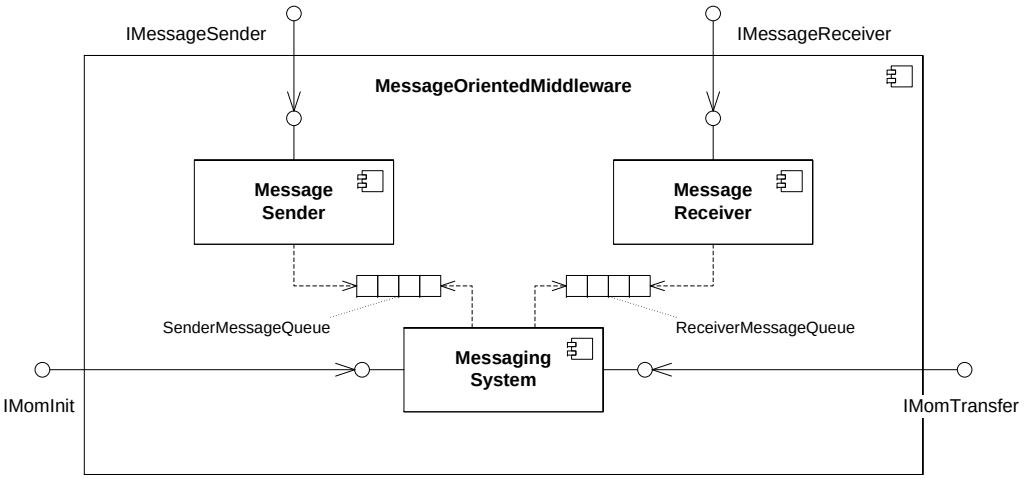


Abbildung 5.28: Modell der Komponente `MessageOrientedMiddleware`, die ein Nachrichtensystems repräsentiert

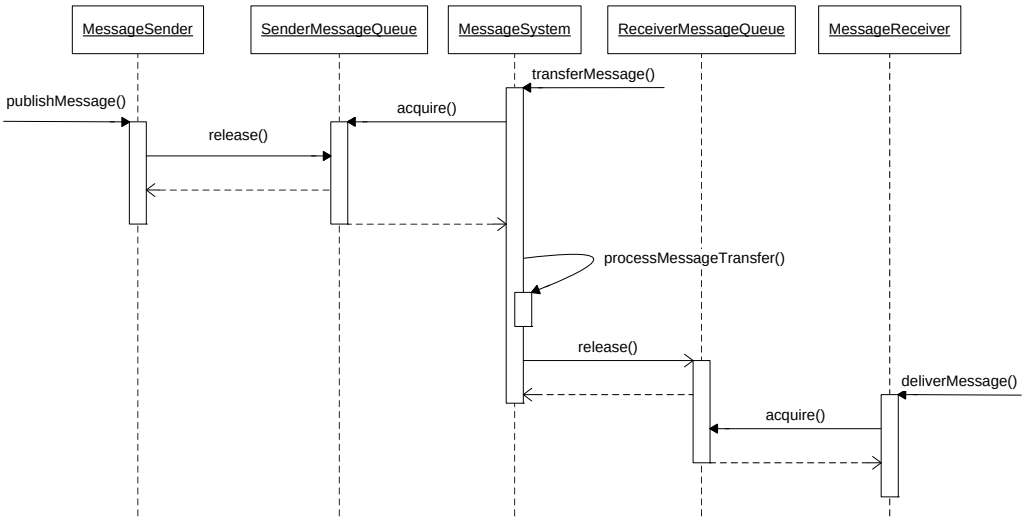


Abbildung 5.29: Sequenzdiagramm für die asynchrone Kommunikation mittels Warteschlangen

Das Komponentendiagramm in Abbildung 5.28 zeigt, dass die Komponente `MessageOrientedMiddleware` als zusammengesetzte Komponente modelliert ist. Sie besteht aus den Basiskomponenten `MessageSender`, `MessagingSystem` und `MessageReceiver`, die ausschließlich über die passive Ressourcen `SenderMessageQueue` und `ReceiverMessageQueue` kommunizieren. Das Verhalten der beiden passiven Ressourcen entspricht jenem von Semaphoren. Auf diese Weise können Warteschlangen im PCM modelliert werden. In diesem Fall fungieren die passiven Ressourcen als Warteschlangen für Nachrichten. In Abbildung 5.29 ist das Sequenzdiagramm für die

Kommunikation der Basiskomponenten abgebildet. Dabei ist zu erkennen, dass die Übertragung einer Nachricht in zwei Schritten erfolgt. Im ersten Schritt wird eine Nachricht von der Komponenten `MessageSender` an die Komponente `MessagingSystem` übergeben. Im zweiten Schritt wird die Nachricht von der Komponente `MessagingSystem` an die Komponente `MessageReceiver` weitergereicht. Damit diese beiden Schritte asynchron zueinander ablaufen können, sind diese jeweils als Erzeuger-Verbraucher-System modelliert. Dabei wird vom `MessageSender` über eine Freigabe auf der passiven Ressource `SenderMessageQueue` eine Nachricht in die erste Warteschlange eingefügt. Falls die Warteschlange Nachrichten enthält, kann die Komponente `MessagingSystem` Nachrichten daraus entnehmen, indem sie die passive Ressource `SenderMessageQueue` erlangt. Die Kommunikation zwischen den Komponenten `MessagingSystem` und `MessageReceiver` erfolgt analog mithilfe der passiven Ressource `ReceiverMessageQueue`, die die zweite Warteschlange repräsentiert. Zwischen den beiden Schritten der Nachrichtenübertragung kann ein Ressourcenbedarf entsprechend der Performance-relevanten Parameter erfolgen. Das Sequenzdiagramm zeigt auch die Notwendigkeit, dass der Dienst `transferMessage` der Komponente unabhängig von den Diensten der Komponenten `MessageSender` und `MessageReceiver` aufgerufen werden muss. Aus diesem Grund wird zusätzlich ein Verwendungsszenario erstellt, das diesen Dienst für den Nachrichtentransfer aufruft.

Dem Komponentendiagramm in Abbildung 5.28 ist zu entnehmen, dass die Komponente `MessageOrientedMiddleware` ihre Dienste über vier verschiedene Schnittstellen anbietet. Aufrufe des Diensts `initMessagePublishing` auf der Schnittstelle `IMomInit` werden an die Komponente `MessagingSystem` weitergereicht. Das Verhalten dieses Diensts bei solchen Aufrufen charakterisiert den Ressourcenbedarf, der bei der Initialisierung des Nachrichtensystems vor dem eigentlichem Nachrichtenversand festzustellen ist. Für ein JMS-Nachrichtensystem wäre dies konkret jener Aufwand, der mit Erzeugung der Objekte für die *Session*, für den *MessageProducer* und für die Nachricht selbst verbunden ist (siehe Abschnitt 3.5.1). An die Komponente `MessageSender` werden Aufrufe auf der Schnittstelle `IMessageSender` delegiert, um über den angebotenen Dienst `publishMessage` den Versand einer Nachricht zu starten. Der Dienst `transferMessage` wird von der Komponente `MessagingSystem` über die Schnittstelle `IMomTransfer` angeboten. Damit kann über einen Aufruf aus dem Verwendungsszenario, das speziell für den Start des Nachrichtentransfers ausgelegt ist, die Durchführung des Übertragungsvorgangs unabhängig von den anderen Komponenten initiiert werden. Für den Empfang von Nachrichten bietet die Komponente `MessageReceiver` über die Schnittstelle `IMessageReceiver` den Dienst `deliverMessage` an. Über diese Schnittstelle kann auch der Dienst `finishReceiving` aufgerufen werden, der den Nachrichtenempfang abschließt.
Das Besondere an dem Entwurf der drei Basiskomponenten ist, dass sie sich über passive Ressourcen miteinander synchronisieren, anstatt ihre angebotenen direkt aufzurufen. Damit kann die Asynchronität von nachrichtenorientierter Kommunikation modelliert werden, obwohl das PCM bis dato keine nativen asynchronen Aufrufe unterstützt. Die Abbildungen 5.31 bis 5.34 zeigen dies anhand der SEFFs, mit denen das interne Verhalten der Komponenten charakterisiert wird.

Mithilfe der SEFFs der Komponenten `MessageSender`, `MessagingSystem` und `MessageReceiver` wird im Folgenden das modellierte Verhalten für die Nachrichtenübertragung exemplarisch beschrieben.

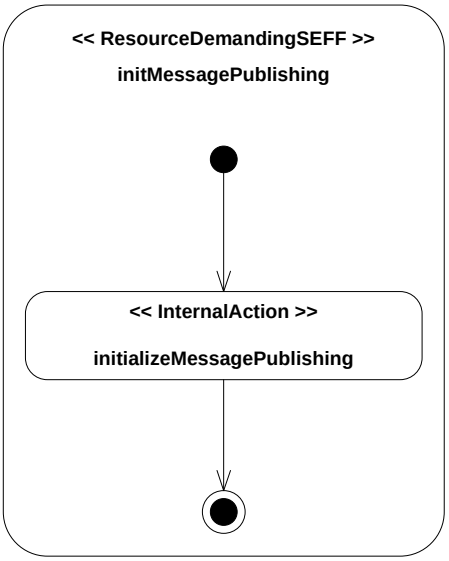


Abbildung 5.30: SEFF für den Dienst `initMessagePublishing` der Komponente `MessagingSystem` zur Charakterisierung des Verhaltens bei der Initialisierung des Nachrichtensystems vor dem Nachrichtenversand

Vor dem eigentlichen Nachrichtenversand erfolgt die Initialisierung der Nachrichtenübertragung über den Dienst `initMessagePublishing` der Komponente `MessagingSystem`. Der SEFF zu diesem Dienst, der in Abbildung 5.30 dargestellt ist, besteht nur aus der internen Aktionen `initializeMessagePublishing`. In dieser Aktion wird der Ressourcenbedarf spezifiziert, der bei der Initialisierung anfällt.

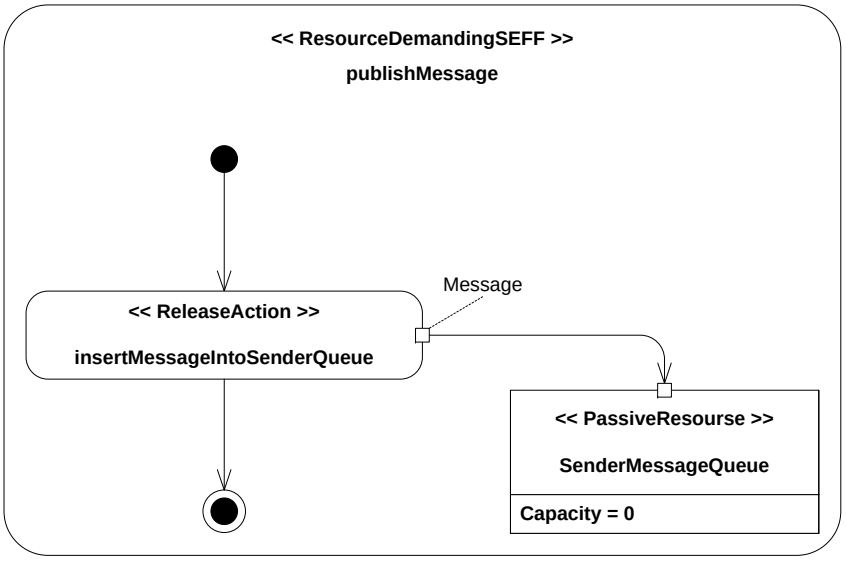


Abbildung 5.31: SEFF der Komponente `MessageSender`, das den Versand von Nachrichten modelliert

Ausgangspunkt für die Nachrichtenübertragung ist der Aufruf des Diensts `publishMessage` auf der Komponente `MessageSender` zur Initiierung eines Nachrichtenversands. Auf einen solchen Aufruf reagiert die Komponente mit der Ausführung der Aktion `insertMessageIntoSenderQueue`, die eine Nachricht in der passiven Res-

source SenderMessageQueue freigibt (siehe Abbildung 5.31). Die Semantik dieser Freigabe basiert auf jener von Semaphoren, so dass die Kapazität der passiven Ressource mit jeder Freigabeaktion um eins erhöht wird. Der Nachrichtenversand ist damit abgeschlossen und der Kontrollfluss kehrt zur aufrufenden Komponente zurück, ohne dass auf das Ende der Nachrichtenübertragung gewartet werden muss.

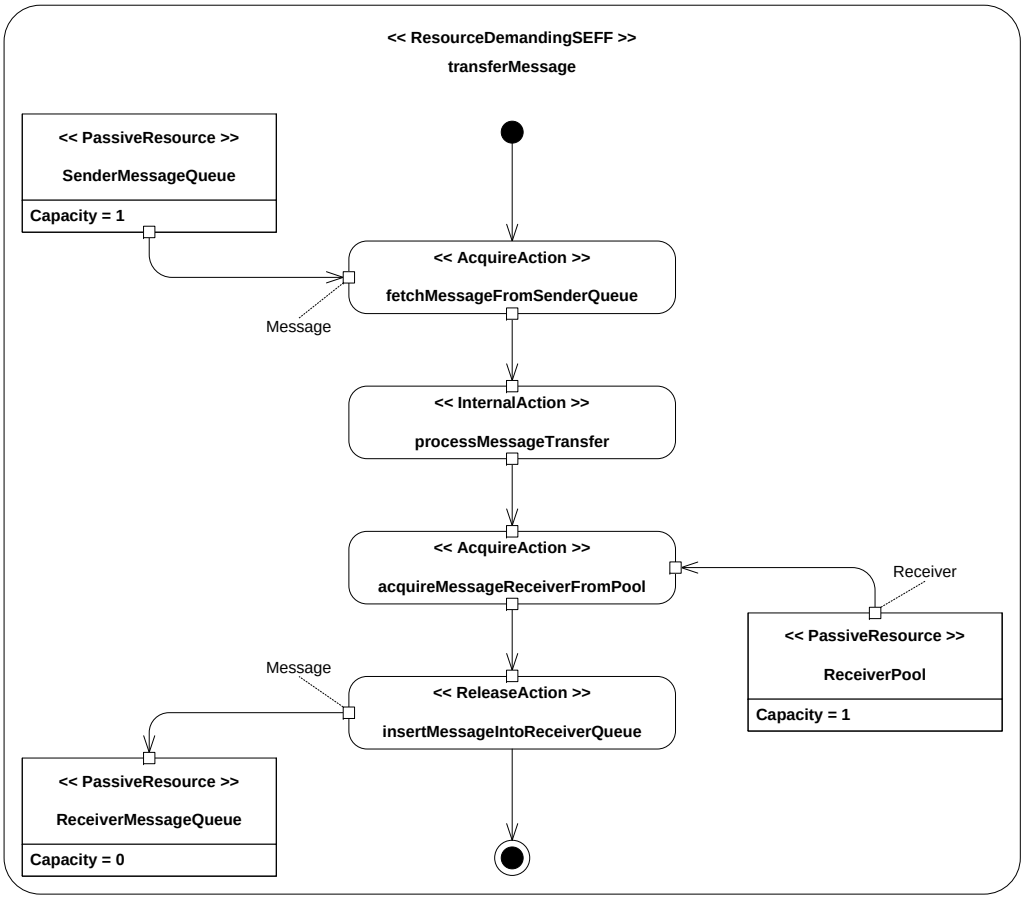


Abbildung 5.32: SEFF für den Dienst transferMessage der Komponente MessagingSystem als Modellierung des Verhaltens bei der Übertragung von Nachrichten

Die Nachrichtenübertragung erfolgt im Modell durch den Aufruf des Diensts transferMessage der Komponente MessagingSystem. Das interne Verhalten sieht zunächst in der Aktion fetchMessageFromSenderQueue das Erlangen einer Nachricht von der passiven Ressource SenderMessageQueue vor. Diese Aktion blockiert so lange bis Nachrichten in der Ressource verfügbar sind, d. h. bis deren Kapazität einen Wert von mindestens eins aufweist. Im Beispiel in Abbildung 5.32 hat die passive Ressource SenderMessageQueue als Resultat des zuvor beschrieben Hinzufügens einer Nachricht zur SenderMessageQueue eine Kapazität von eins. Damit kann eine Nachricht von der Ressource erlangt werden ohne dass diese Aktion blockiert. Diese Aktion stellt somit das Gegenstück zu der Freigabeaktion der Komponente MessageSender dar. Dementsprechend ist mit dem Erlangen eine Verringerung der Kapazität der passiven Ressource verbunden. In der folgenden internen Aktion processMessageTransfer wird die Übertragungsdauer von Nachrichten als Ressourcenbedarf modelliert, um so eine entsprechende Verzögerung realisieren zu können, wie sie beim realen Einsatz von Nachrichtensystemen zu messen ist. Im Allgemeinen stehen für den Empfang von Nachrichten eine begrenzte Anzahl von konkurrieren-

den Empfängern zur Verfügung. In Anlehnung an den *Message-Driven-Bean*-Pool von Java EE-basierten Anwendungs-Servern (siehe Abschnitt 5.2.3.2) wird dafür die passive Ressource `ReceiverPool` eingesetzt. Der `ReceiverPool` zeigt an, wie viele wartende Empfänger existieren. Bevor eine Nachricht einem Empfänger zugestellt werden kann, ist es erforderlich, dass mindestens ein wartender Empfänger im Empfänger-Pool vorhanden ist. Im SEFF der Komponente `MessagingSystem` erfolgt dies durch das Akquirieren eines Empfängers von der passiven Ressource `ReceiverPool`. Im Beispiel kann in der entsprechende Aktion `acquireMessageReceiverFromPool` ohne zu blockieren ein Empfänger akquiriert werden, da die Kapazität des `ReceiverPool` eins beträgt. Den Abschluss der Übertragungsvorgangs bildet die Aktion `insertMessageIntoReceiverQueue`, bei der eine Nachricht in der passiven Ressource `ReceiverMessageQueue` freigegeben wird.

Das Verhalten der Komponente `MessageReceiver` wird für den Nachrichtenempfang und für die Beendigung des Empfangs getrennt betrachtet, da dazwischen die Abarbeitung der empfangenen Nachricht erfolgt, die unabhängig vom Nachrichtensystems durchgeführt wird. Der Zusammenhang zwischen dem Nachrichtenempfang und dessen Beendigung ist Abbildung 5.26(b) zu entnehmen.
Der in Abbildung 5.33 dargestellte SEFF für den Dienst `deliverMessage` dieser Komponente charakterisiert das Verhalten beim Nachrichtenempfang. Er hat als erste Aktion das Erlangen einer Nachricht von der passiven Ressource `ReceiverMessageQueue`. Im Beispiel kann dies ohne zu blockieren durchgeführt werden, da `ReceiverMessageQueue` eine Kapazität von eins besitzt. Als Folge dieser Aktion wird analog zu allen anderen Akquirierungsaktionen die Kapazität der `ReceiverMessageQueue` um den Wert eins dekrementiert. Die letzte Aktion `setMessageContentSize` des SEFF ist eine `SetVariableAction`, mit der die Größe des Nachrichteninhalts festgelegt wird. Damit ist es möglich einen Ressourcenbedarf auf einer Netzwerkressource zu realisieren, falls diese für die Zustellung zu jener Komponente in Anspruch genommen wird, die das Ziel der nachrichtenorientierten Kommunikationsverbindung ist.

Nach Abschluss der Bearbeitung einer Nachricht muss der Empfänger wieder dem Empfänger-Pool hinzugefügt werden. Dieses Verhalten realisiert der SEFF für den Dienst `finishReceiving` der Komponente `MessageReceiver` (siehe Abbildung 5.34). Dieser besteht lediglich aus der Aktion `insertMessageReceiverIntoPool`, die einen Empfänger in der passiven Ressource `ReceiverPool` freigibt. Im Beispiel hat diese Ressource eine Kapazität von null, was bedeutet, dass keine bereiten Empfänger im Pool vorhanden sind. Durch diese Freigabeaktion erhöht sich die Kapazität um den Wert eins, so dass wieder ein Empfänger für den Nachrichtenempfang zur Verfügung steht und gemäß Abbildung 5.32 in Anspruch genommen werden kann.

Insgesamt wird jeweils ein Erzeuger-Verbraucher-System als Synchronisationsmechanismus für die Modellierung des Nachrichtenversands zwischen den Komponenten `MessageSender` und `MessagingSystem` sowie zwischen `MessagingSystem` und `MessageReceiver` verwendet. Dadurch werden diese Komponenten voneinander entkoppelt und die Asynchronität von nachrichtenorientierter Kommunikation abgebildet. Der Modellierung des Empfänger-Pools liegt ebenfalls ein Erzeuger-Verbraucher-System zugrunde.

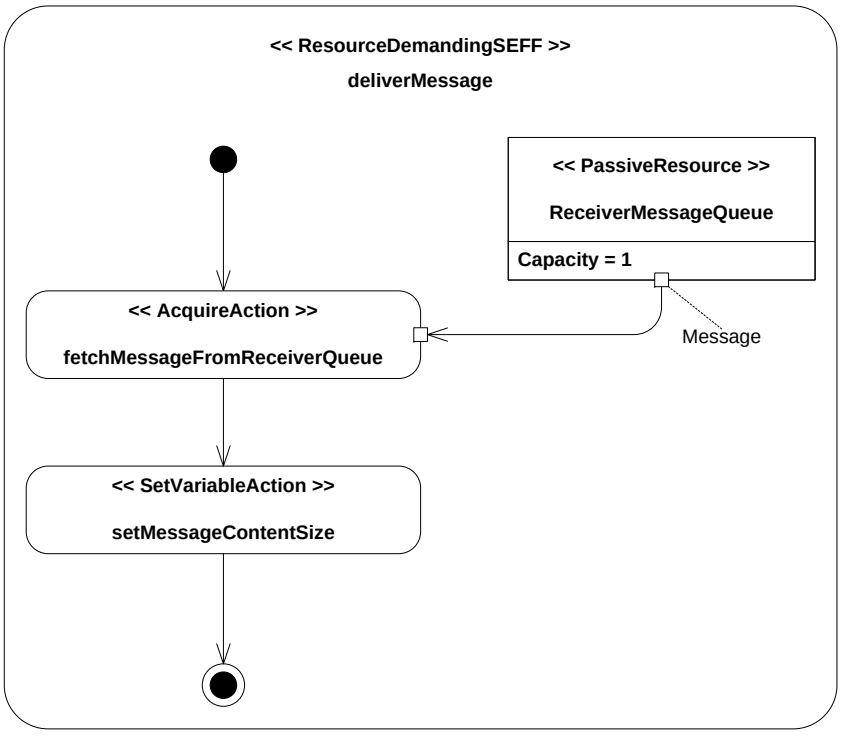


Abbildung 5.33: SEFF für den Dienst `deliverMessage` der Komponente `MessageReceiver`, das den Empfang von Nachrichten modelliert

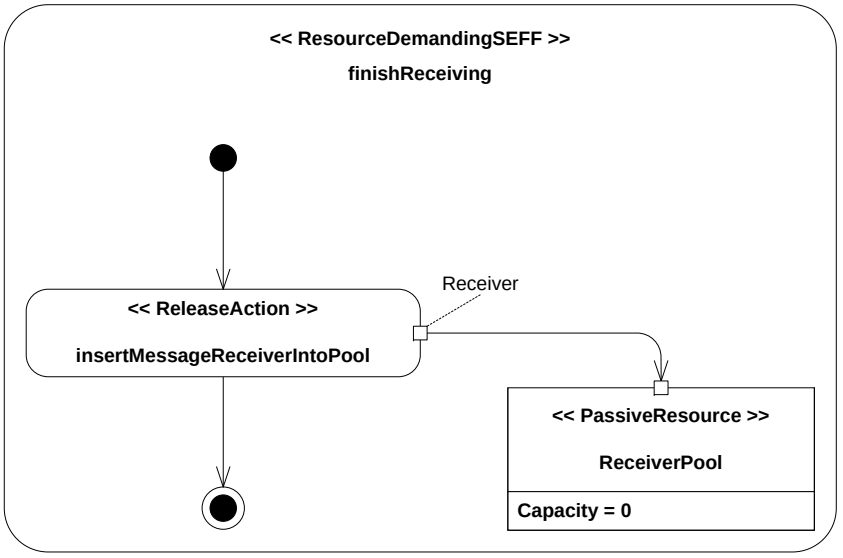


Abbildung 5.34: SEFF für den Dienst `finishReceiving` der Komponente `MessageReceiver` zur Beendigung des Nachrichtenempfangs

Variationen und Schablonen

Das modellierte Nachrichtensystem soll die Unterschiede bei der Übertragungsdauer bei unterschiedlichen Parameterkonfigurationen widerspiegeln. In dieser Arbeit wurde ein Ansatz gewählt, bei dem für einen Parameter eine Schablone erstellt wird, die das Verhalten des Nachrichtensystem bei diesem Parameter modelliert. Der Grund hierfür ist, dass dafür ein spezieller SEFF nötig ist. Dies hat den Vorteil, dass bei der in Abschnitt 5.4 beschriebenen Transformation lediglich die benötigte Schablone ausgewählt werden muss, anstatt ein Modell des Verhaltens des Nachrichtensystems während der Transformation zu generieren. Infolgedessen sinkt die Komplexität der Transformation. Für die übrigen Parameter genügt es, das Verhalten der Komponente `MessagingSystem` durch eine geringfügige Anpassung des Ressourcenbedarf der internen Aktion `processMessageTransfer` (siehe Abbildung 5.32) zu ändern. Im Folgenden werden die Schablonen bzw. Anpassungen des Ressourcenbedarfs für die Parameter des Annotationsmodells erläutert (siehe Abbildung 5.21):

MessageDestination Für die beiden Nachrichtenkanaltypen `PointToPointChannel` und `PublishSubscribeChannel` sind in Abschnitt 5.2.3.2 unterschiedliche Häufungen der Messwerte für die Übertragungsdauer von Nachrichten ermittelt worden. Dem wird bei der Modellierung des Verhaltens eines Nachrichtensystems dahingehend Rechnung getragen, als dass die Verteilungsfunktion des Ressourcenbedarfs der internen Aktion `processMessageTransfer` eine Annäherung an die jeweilige Verteilung der Messergebnisse für diese beiden Parameter darstellt. Bei der Transformation wird die Verteilungsfunktion des Ressourcenbedarfs entsprechend des gewählten Nachrichtenkanaltyps angepasst.
Die Verwendung der dauerhaften Registrierung an einem `PublishSubscribeChannel` über den Parameter `DurableSubscriber` ist gemäß den Messergebnissen aus Abschnitt 5.2.3.2 mit einer Verlängerung der Nachrichtenübertragungsdauer von ungefähr 25% aus. Dementsprechend wird bei der Transformation der Ressourcenbedarf in der internen Aktion `processMessageTransfer` mit dem Faktor 1,25 multipliziert.

TransactionalClient Bei transaktionalem Nachrichtenversand unterscheidet sich das Verhalten des Nachrichtensystems deutlich vom Verhalten beim nicht-transaktionalen Versand. Aus diesem Grunde wird dafür eine eigene Schablone entworfen, die ein spezieller SEFF für den Transfervorgang in der Komponente `MessagingSystem` beinhaltet. Für die Komponente `MessagingSystem` bedeutet der transaktionale Nachrichtenversand, dass sie die Übertragung und Zustellung aller Nachrichten einer Transaktion auf einmal erledigen muss. Das transaktionale Verhalten der `MessagingSystem`-Komponente zeigt Abbildung 5.35, das sich durch zwei Schleifenaktionen auszeichnet. Die erste Schleifenaktion `iterateTaMessagesFromSender` iteriert über die Anzahl der Nachrichten in einer Transaktion und führt in jeder Iteration den bereits vorgestellten Ablauf des Erlangens einer Nachricht und des anschließenden Ressourcenbedarf in der internen Aktion `processMessageTransfer` aus. Darauf folgend werden in der zweiten Schleifenaktion `iterateTaMessagesToReceiver` für jede der Nachrichten ein Empfänger aus dem Empfänger-Pool akquiriert und die Nachricht selbst in der passiven Ressource `ReceiverMessageQueue` freigegeben. Es ist jedoch nicht ausreichend, dass nur der Vorgang der Nachrichtenübertragung transaktionales Verhalten aufweist. Zusätzlich muss auch Versand und Empfang von

Nachrichten transaktional erfolgen. Dafür sind spezielle SEFFs für die Dienste `sendMessage` und `onMessage` der Vervollständigungskomponenten `MessageSenderAdapter` und `MessagereceiverAdapter` (siehe Abschnitt 5.3.2) erforderlich. Abbildung 5.36 zeigt beide SEFFs. Beim Nachrichtenversand über den Dienst `sendMessage` erfolgt pro Transaktion einmal die Initialisierung des Nachrichtensystems über den Aufruf des externen Diensts `initMessagePublishing`. Anschließend wird für jede Nachricht der Transaktion der Sendevorgang gestartet. Auf Empfängerseite wird über die Anzahl der Nachrichten innerhalb einer Transaktion iteriert. In jeder Iteration wird dabei der Empfangsvorgang des nicht-transaktionalen Nachrichtenempfangs ausgeführt.

SelectiveConsumer Für die Verwendung des Parameters `SelectiveConsumer`, der den Einsatz einer Nachrichtenfilterung auf Empfängerseite spezifiziert, genügt es die Verteilungsfunktion für den Ressourcenbedarf mit dem Faktor 1,07 zu skalieren, da die in Abschnitt 5.2.3.2 ermittelte Verlängerung der Übertragungsdauer ungefähr 7% beträgt. Diese geringfügige Änderung des Ressourcenbedarfs kann problemlos während der Transformation durchgeführt werden, so dass hierfür keine eigene Schablone erforderlich ist.

GuaranteedDelivery Bei der in Abschnitt 5.2.3.2 durchgeführten Evaluierung der Performance-relevanten Parameter wurde festgestellt, dass sich die Verteilung der Übertragungsdauerwerte bei persistenter Speicherung der zu übertragenden Nachrichten von jener bei nicht-persistenter Nachrichtenübertragung unterscheidet. Ein Unterschied bei der Verteilung der Messwerte ist ebenfalls für die beiden untersuchten Nachrichtenkanaltypen feststellbar. Aus diesem Grund wird für beide Nachrichtenkanaltypen während der Transformation in der internen Aktion `processMessageTransfer` eine Verteilungsfunktion für Ressourcenbedarf eingesetzt, die die entsprechende Verteilung der Übertragungsdauerwerte bei persistenter Nachrichtenübertragung annähert.

CompetingConsumers Die Modellierung eines Nachrichtensystems berücksichtigt grundsätzlich einen Pool von konkurrierenden Empfängern, da dies auch als verbindlicher Parameter eingestuft wurde. Somit ist es nicht nötig dafür eine Schablone bereitzustellen. Bei der Transaktion wird lediglich die initiale Kapazität der passiven Ressource `ReceiverPool` auf den Wert gesetzt, wie er in der Annotation spezifiziert wurde.

MessageSize Die Messergebnisse aus Abschnitt 5.2.3.2 zeigen, dass sich die Übertragungsdauer für Nachrichten mit großem Inhalt deutlich verlängert. Die Berücksichtigung dieses Parameters im Modell für das Nachrichtensystem erfolgt bei der Transformation, bei der die Verteilungsfunktion für den Ressourcenbedarf der internen Aktion `processMessageTransfer` in Abhängigkeit von Nachrichtengröße skaliert wird, deren Wert in der Annotation spezifiziert wird. Der Skalierungsfaktor für die spezifizierte Nachrichtengröße wird über das Verhältnis zwischen den mittleren Übertragungsdauerwerten bei dieser Nachrichtengröße und bei einer minimalen Nachrichtengröße von einem Zeichen berechnet. Für die Bestimmung der Werte für

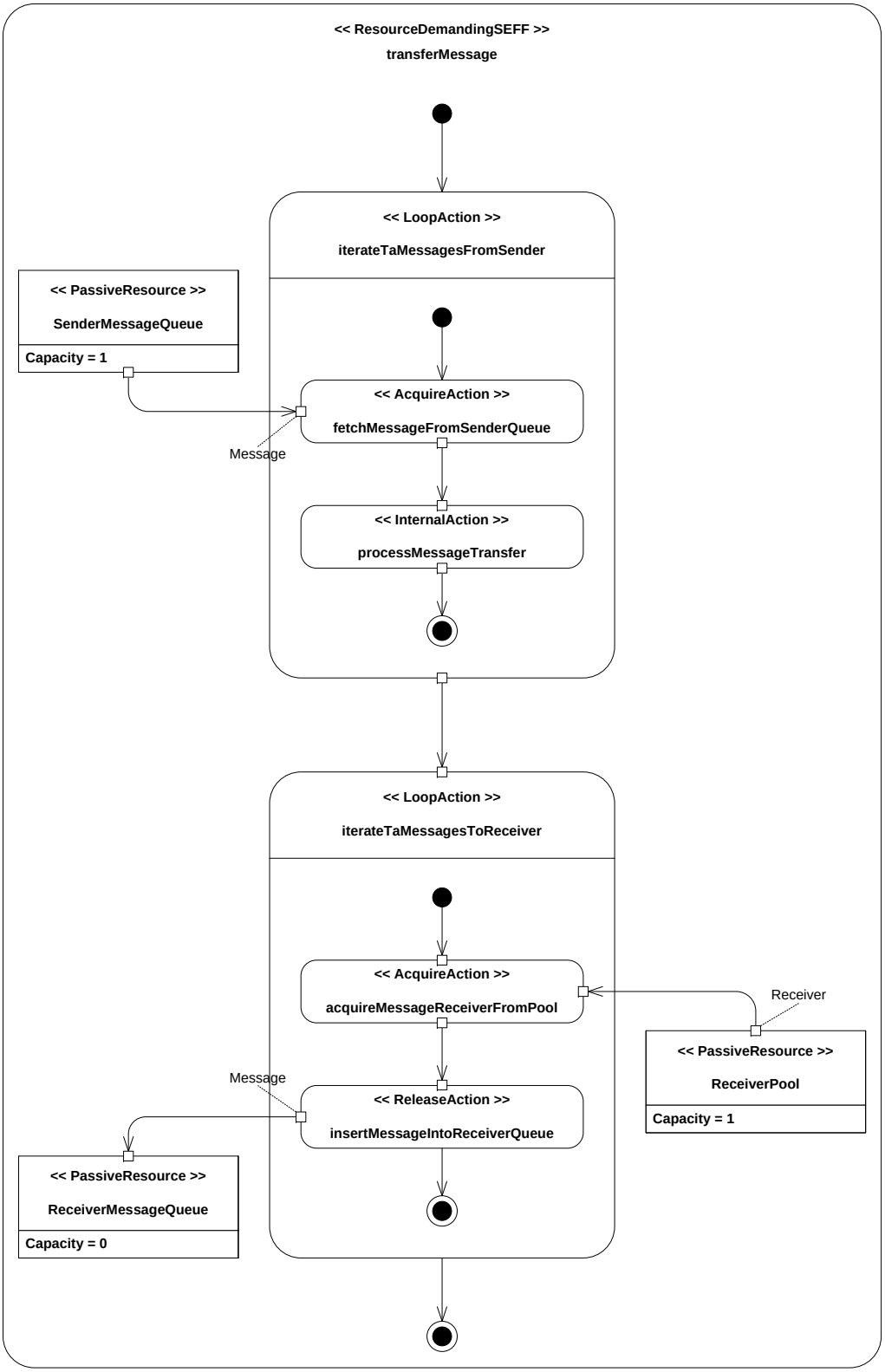
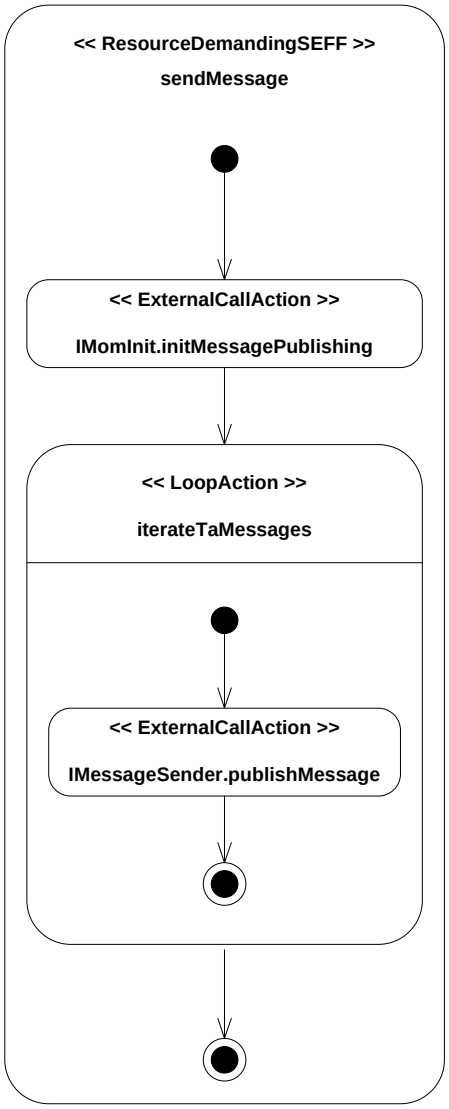
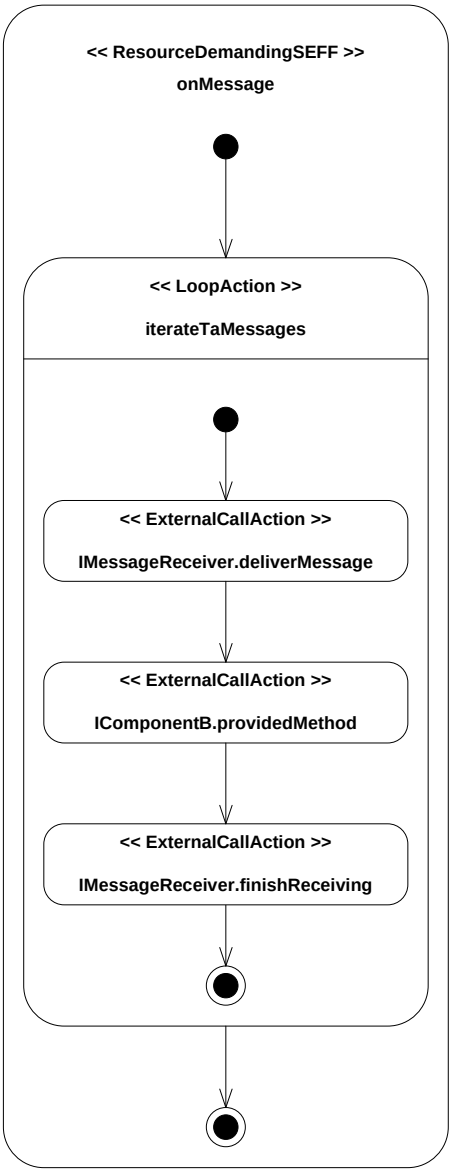


Abbildung 5.35: SEFF für den Dienst `transferMessage` der Komponente `MessagingSystem` als Modellierung des Verhaltens bei der transaktionalen Übertragung von Nachrichten



(a) SEFF der Komponente `MessageSenderAdapter` für die Initiierung des transaktionalen Nachrichtenversands



(b) SEFF der Komponente `MessageReceiverAdapter` für den transaktionalem Empfang von Nachrichten

Abbildung 5.36: Verhaltensspezifikationen für die Basiskomponenten der Vervollständigung bei transaktionalem Nachrichtenversand

die Übertragungsdauer wird das Verfahren der genetischen Programmierung (siehe Abschnitt 3.6.1) angewandt, mit dem aus allen relevanten Messwerten ein Berechnungsalgorithmus für die mittlere zu erwartende Übertragungsdauer in Abhängigkeit von der Nachrichtengröße generiert wird. Dies hat gegenüber statistischen Regressionsverfahren den Vorteil, dass keine konkrete Vermutung über eine Funktion zwischen Nachrichtengröße und Übertragungsdauer angestellt werden muss. Mit dem Verfahren der genetischen Programmierung wird selbständig eine Kombination aus Individuen bestimmt, sodass der resultierende Berechnungsalgorithmus möglichst genaue Ergebnisse liefert. Zudem bietet sich die Möglichkeit, dass die Bestimmung der zu erwartenden Übertragungsdauer automatisiert während der Transformation durchgeführt wird und das Resultat sofort in die Transformation einbezogen wird. Weiterhin kann mithilfe der genetischen Programmierung der Einfluss der Nachrichtengröße auf die Übertragungsdauer für andere Nachrichten- und Rechensysteme bestimmt werden, indem dieser die gemessenen Werte dieser Systeme einliest. Für die Bestimmung der Nachrichtenübertragungsdauer wurde die genetische Programmierung mit einem Individuenraum konfiguriert, der eine Menge von arithmetischen Ausdrücken umfasst. Dazu zählen die Operationen Addition, Substraktion, Multiplikation und Division sowie Potenz- und Exponentialfunktionen. Zusätzlich zu den arithmetischen Ausdrücken stehen Verzweigungskonstrukte zur Verfügung. Die Qualität einer Zwischenlösung wird mithilfe einer Fitnessfunktion bestimmt, die den Gesamtfehler zwischen allen berechneten Übertragungsdauerwerten und dem entsprechenden gemessen Wert errechnet.
Tabelle 5.10 zeigt für zehn Nachrichtengrößen einen Vergleich der Medianwerte mit den Übertragungsdauerwerten, die mit dem besten Berechnungsalgorithmus berechnet wurden, der mit dem Verfahren der genetischen Programmierung generiert wurde. Die Basis dafür bilden die Übertragungsdauermesswerte für Konfiguration „Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“. Die genetische Programmierung wurde mit Nachrichtengrößen von 1, 10, 100, 1000, 10000 und 100000 Zeichen trainiert. Die Abweichungen der berechneten Übertragungsdauer vom Median der Messwerte beträgt maximal 1,28% für Trainingsdaten. Für Nicht-Trainingsdaten sind Abweichungen von maximal 3,48% festzustellen. Diese Ergebnisse zeigen, dass mit der Anwendung der genetischen Programmierung für die Approximation der Übertragungsdauer in Abhängigkeit von der Nachrichtengröße sehr gute Ergebnisse erzielt werden können.

Ort des Nachrichtensystems Der Ort an dem das verwendete Nachrichtensystem eingesetzt wird, wirkt sich den Messergebnisse aus Abschnitt 5.2.3.2 zufolge deutlich auf die Übertragungsdauer von Nachrichten aus, da möglicherweise für den Nachrichtenaustausch über eine Netzwerkverbindung kommuniziert werden muss. Der Einfluss der Netzwerkverbindung konnte mit den durchgeführten Messungen nicht eindeutig ermittelt werden. Ausgehend von den Messergebnissen wird deshalb ein konstanter Ressourcenbedarf angenommen, der die Verzögerung bei der Nachrichtenübertragung repräsentiert, die durch die Netzwerkverbindung verursacht wird. Der Ressourcenbedarf der internen Aktion `processMessageTransfer` wird dann um diesen konstanten Wert erhöht. Dies geschieht während der Transformation (siehe Abschnitt 5.4), falls die Annotationsinstanz für die gewünschte Parameterkonfiguration die Kommunikation über eine Netzwerkressource spezifiziert.

Anzahl der Zeichen als Nachrichteninhalt	Median der gemessenen Übertragungsdauerwerte	berechnete Übertragungsdauer	Abweichung der Berechnung vom Median
1	1192717ns	1198094,75ns	0,45%
10	1207990ns	1198193,76ns	0,81%
50	1212818ns	1198633,75ns	1,17%
100	1196674ns	1199183,76ns	0,21%
500	1219459ns	1210025,75ns	0,77%
1000	1223311ns	1222025,76ns	0,11%
5000	1315154ns	1318025,75ns	0,22%
10000	1456620ns	1438025,76ns	1,28%
50000	2317321ns	2398025,75ns	3,48%
100000	3582204ns	3598025,76ns	0,44%

Tabelle 5.10: Vergleich zwischen dem Median der Übertragungsdauer und der Übertragungsdauer, die mittels eines Algorithmus berechnet wurde, der durch Generierung mit dem Verfahren der genetischen Programmierung entstand. Die Werte beziehen sich auf die Konfiguration „Queue mit automatischem Bestätigungsmechanismus und nicht-persistenter Nachrichtenübertragung“

5.4 Transformation der Modellkonstrukte

Die Spezifizierung eines Konnektors zwischen zwei Komponenten als nachrichtenorientierte Kommunikationsbeziehung erfolgt mithilfe des in Abschnitt 5.3.1 vorgestellten Annotationsmodells. Für die Performance-Vorhersage ist es nun erforderlich, dass jeder mit einer Annotation versehene Konnektor durch die in Abschnitt 5.3.2 vorgestellte Vervollständigung ersetzt wird und die Komponente für das Nachrichtensystem entsprechend den Parametern der Annotation konfiguriert wird. Im Folgenden wird eine Transformation beschrieben, die genau dies leistet.

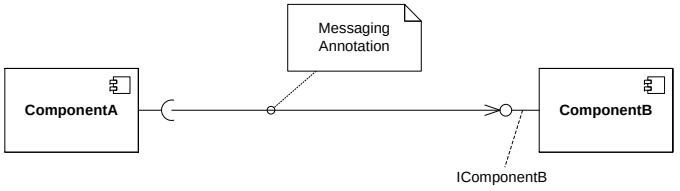


Abbildung 5.37: Annotierter Entwurf als den Ausgangspunkt für die Transformation

Den Ausgangspunkt für die Transformation bildet ein Entwurf, bei dem mindestens ein Konnektor mit einer Annotation für nachrichtenorientierte Kommunikation versehen wurde. In Abbildung 5.37 ist ein Beispiel eines solchen Entwurfs dargestellt. Der erste Schritt des Transformationsvorgangs ist das Einlesen aller Modellinstanzen, inklusive dem Annotationsmodell, den Modellinstanzen für die Vervollständigung und das Nachrichtensystem sowie deren Verwendungsszenarien. Für jeden Komponentenkonnektor, der mit einer Annotationsinstanz für nachrichtenorientierte Kommunikation spezifiziert wurde, werden die folgenden Schritte durchgeführt. Zunächst werden für die einzufügenden Basiskomponenten der Vervollständigung die Referenzen auf Ressourcen an die Ressourcenumgebung angepasst. Das selbe wird für

die Basiskomponenten der Nachrichtensystemkomponente durchgeführt. Dies erfolgt anhand der in der Annotation spezifizierten Referenzen `requiredAllocationContext_MessagingAnnotation`, `providedAllocationContext_MessagingAnnotaion` und `momDeployedOn_MomConfiguration` (siehe Abbildung 5.20). Die Transformation wendet für diesen Schritt das Besucher-Entwurfsmuster [GHJV95, S. 331ff] an, indem ein Besucher durch die Komponentenstrukturen traversiert und die jeweils benötige Ressource referenziert. Im nächsten Transformationsschritt erfolgt die Erzeugung eines `AssemblyContext` für die Vervollständigung und die Nachrichtensystemkomponente. Der `AssemblyConnector`, der in der Annotation spezifiziert wurde, wird durch zwei neue Konnektoren ersetzt, die die beiden Komponenten mit der Vervollständigung verbinden. Den Abschluss dieses Transformationsschritts bildet Verbindung der Vervollständigung mit der Nachrichtensystemkomponente, wofür ebenfalls neue Konnektoren erzeugt werden. Im folgenden Schritt der Transformation wird der `AssemblyContext` jeder Basiskomponente der Vervollständigung und der Komponente für das Nachrichtensystem jener Ressource zugeteilt, wie sie in der Annotation spezifiziert wurde. Der vierte Schritt umfasst die Anpassung der Signaturen und SEFFs. Die Signatur des Dienstes *sendMessage* muss an die Signatur des Diensts angepasst werden, den die aufrufende Komponente benötigt. Zusätzlich muss der Aufruf genau dieses Diensts im SEFF für den Dienst *onMessage* der Komponente `MessageReceiverAdapter` eingefügt werden. Dadurch wird ermöglicht, dass die Vervollständigung als Adapter für die Zielkomponente fungieren kann. Insgesamt ergibt sich dann eine Kommunikationsbeziehung wie sie in Abbildung 5.24 dargestellt ist. Teil dieses Transformationsschritts ist auch die Auswahl und Anpassung einer Schablone für das Verhalten der Nachrichtensystemkomponente gemäß der Parameterkonfiguration, wie sie in der Annotation vorgenommen wurde. Des Weiteren werden die Komponenten der Vervollständigung und des Nachrichtensystems zur Komponentensammlung hinzugefügt. Im letzten Schritt werden die Verwendungsszenarien für die Vervollständigung und das Nachrichtensystem in die Verwendungsmodellinstanz eingebunden, die vom Domänenexperten erstellt wurde.
Abbildung 5.38 zeigt den modifizierten Entwurf des Beispiels nachdem die Transformation durchgeführt wurde

5.5 Zusammenfassung

Ein Teil der Entwurfsmuster für nachrichtenorientierte Kommunikation können auf Optionen für existierende Nachrichtensystem abgebildet werden, wie in Abschnitt 5.2.1 am Beispiel von JMS gezeigt wurde. Auf Grundlage dieser Zuordnung kann mithilfe einer Testanwendung eine Leistungsbewertung von Parametern durchgeführt werden. Für die so ermittelten Performance-relevanten Parameter lässt sich ein Annotationsmodell erstellen, so dass eine Kommunikationsbeziehung zwischen zwei Komponenten als nachrichtenorientierte Kommunikation spezifiziert und mit den gewünschten Parametern konfiguriert werden kann. Die Modifikation eines auf diese Weise mit extra-funktionalen Performance-relevanten Informationen angereicherten Entwurfs erfolgt über eine Transformation, die die direkte Kommunikationsbeziehung durch eine Vervollständigung ersetzt. Diese Vervollständigung benötigt wiederum das Modell eines Nachrichtensystems, das entsprechend der Parameterkombination der Annotation konfiguriert werden kann. Somit können die nachrichtenorientierte Kommunikation und die Performance-relevanten Aspekte der Entwurfsmuster auf einfache Weise in den Software-Entwurf einbezogen werden.

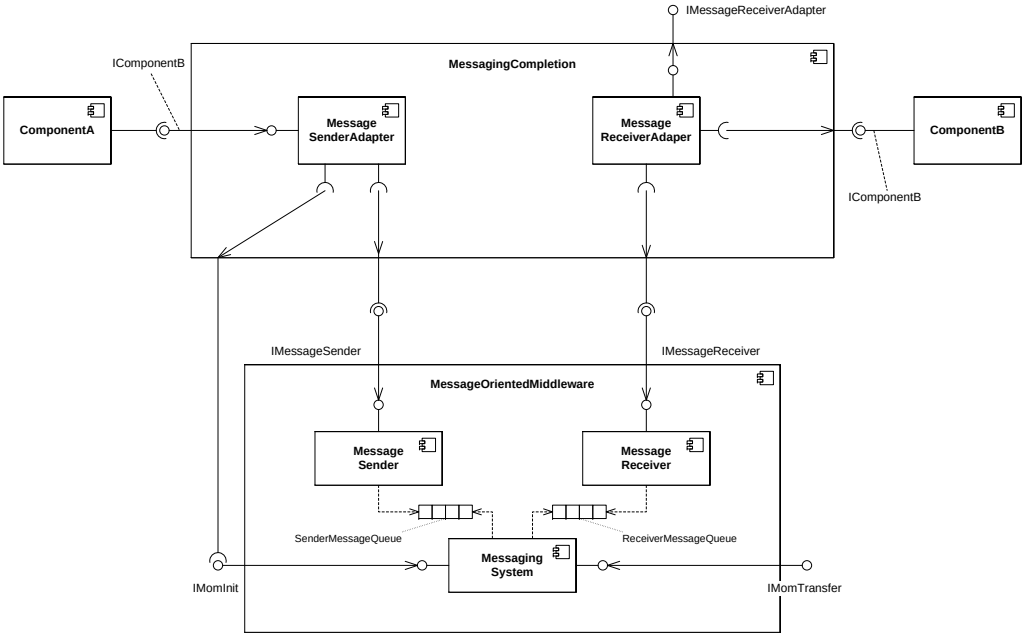


Abbildung 5.38: Modifizierter Entwurf, der durch Transformation des annotierten Entwurfs erzeugt wurde

6. Fallstudie

Die durchgeführte Integration von nachrichtenorientierter Kommunikation ins Palladio Komponentenmodell wird in diesem Kapitel in einer Fallstudie bewertet. Dazu wird eine Anwendung entworfen und implementiert, bei der einige Komponente über den Austausch von Nachrichten kommunizieren. Zusätzlich werden Performance-Vorhersagen für den Anwendungsentwurf durchgeführt. Die Beurteilung der Qualität der Performance-Vorhersage erfolgt durch den Vergleich der Vorhersagewerte mit den Messwerten für die Implementierung.

Der Entwurf der Anwendung wird im ersten Unterkapitel beschrieben. Die Evaluierung der Qualität der Performance-Vorhersagen erfolgt im zweiten Unterkapitel.

6.1 Entwurf

Für die Anwendung der Fallstudie wurde das Szenario eines Handelsprozesses gewählt, bei dem ein Klient eine Bestellung mit einer Menge an gewünschten Produkten an ein Handelssystem übergibt. Das Handelssystem bearbeitet einen solchen Auftrag indem es die Auslagerung der gewünschten Produkte durchführt, eine Rechnung erstellt und das Kaufverhalten des Kunden aktualisiert.
Der Entwurf der Anwendung „Trading System“ (siehe Abbildung 6.1) besteht aus den drei Schichten Klient und Anwendungs-Server und Datenbank-Server. Ein Kunde interagiert über einen Browser mit der Anwendung, die auf der mittleren Schicht angesiedelt ist. Die Komponente `OrderUI_Module` nimmt Bestellungen von Kunden entgegen und leitet diese an die angebundene Komponente `Order_Module` weiter. Die Komponente `Order_Module` steuert die weiteren Schritte des Bestellungsprozesses. Dazu kommuniziert sie mit den Komponenten `ERP_Module` und `CRM_Module`. Ihr internes Verhalten wird durch den in Abbildung 6.2 dargestellten SEFF beschrieben. Demnach ruft die Komponente `Order_Module` zunächst den Dienst zur Auslagerung der gewünschten Produkte auf der Komponente `ERP_Module` auf. Anschließend erfolgt in einer internen Aktion die Rechnungserstellung, die durch die Festlegung eines Ressourcenbedarf an eine rechnende Ressource charakterisiert wird. In der letzten Aktion wird der Dienst zur Aktualisierung des Kaufverhalten des Kunden auf der Komponente `CRM_Module` aufgerufen. Das Verhalten der beiden Komponenten `ERP_Module` und `CRM_Module` besteht jeweils aus einer internen Aktion, die

einen Ressourcenbedarf an eine berechnende Ressource stellt. Als Vereinfachung repräsentiert der Ressourcenbedarf auch jeweils die Kosten für die Zugriffe auf die angeschlossene Datenbank.

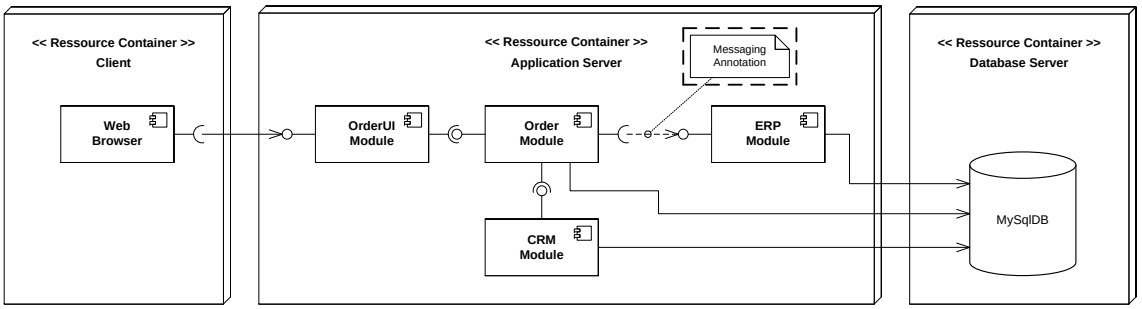


Abbildung 6.1: Architektur der Anwendung „Trading System“

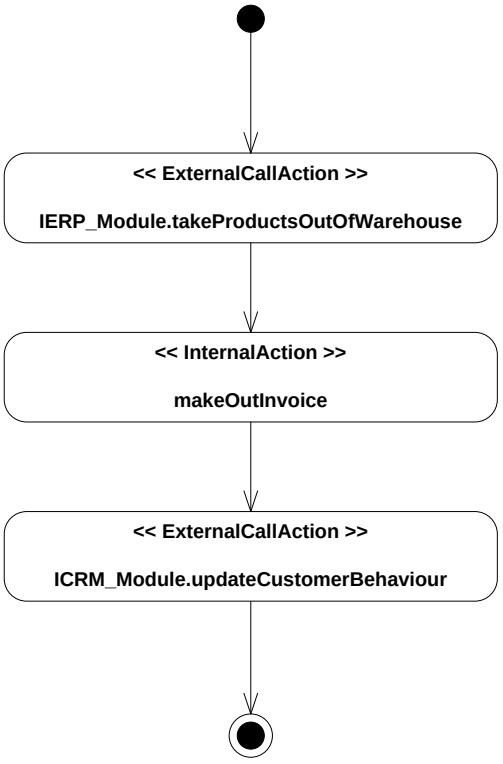


Abbildung 6.2: SEFF der Komponente `Order_Module`für den Dienst `processOrder`

Ein Alternativentwurf ist fast identisch mit dem ersten Entwurf, jedoch wird die Kommunikationsbeziehung zwischen den Komponenten `Order_Module` und `ERP_Module` mit einer Annotation für nachrichtenorientierte Kommunikation versehen. Dabei wird das Ziel verfolgt, die Antwortzeit zu verringern, indem die als zeitintensiv betrachtete Ausführung des Diensts zur Produktauslagerung asynchron erfolgt. Für die Annotation werden zwei Varianten erstellt, die sich lediglich darin unterscheiden, dass bei der ersten die Persistierung der Nachrichten ausgewählt wird, jedoch nicht bei der zweiten Variante. Ansonsten sieht die Parameterkonfiguration der Annotation einen Punkt-zu-Punkt-Kanal als Nachrichtenkanaltyp, eine Nachrichtengröße von

zehn Bytes und eine Größe von eins für den Empfänger-Pool vor. Alle übrigen verfügbaren Parameter sind nicht ausgewählt. Die beiden alternativen Annotationen zielen darauf ab, dass mit der Nachrichtenpersistierung die Zuverlässigkeit erhöht wird. Somit bestehen insgesamt folgende drei Entwurfsalternativen:

1. Entwurf der Anwendung „Trading System“ ohne Verwendung der nachrichtenorientierten Kommunikation
2. Entwurf der Anwendung „Trading System“ mit persistenter Nachrichtenübertragung
3. Entwurf der Anwendung „Trading System“ mit nicht-persistenter Nachrichtenübertragung

Die beiden letzten Alternativen versprechen eine höhere Performance als Alternative 1, da aufgrund der asynchronen Kommunikation über den Versand von Nachrichten die Dienste mehrerer Komponenten nebenläufig ausgeführt werden können. Die Entscheidung zwischen Alternative 2 und 3 bedeutet die Abwägung zwischen den Kriterien Performance und Zuverlässigkeit. Von Alternative 3 wird die höchste Performance erwartet, jedoch ist damit die geringste Zuverlässigkeit verbunden, da die Nachrichten nicht persistiert werden.

6.2 Evaluierung

Alle drei Entwurfsalternativen wurden mit dem PCM modelliert und simuliert. Die Ressourcenanforderungen in allen internen Aktionen wurden mit Wahrscheinlichkeitsverteilungen von gemessenen Werten modelliert. Dabei wird kein Anspruch auf Repräsentativität für existierende Systeme erhoben. Vielmehr erfolgt die Modellierung der Messwerte in einer Weise, dass die zu erwartende Nachrichtenübertragungsdauer einen deutlich messbaren Einfluss hat. Für die beiden Alternativen, bei denen die Spezifizierung von extra-funktionalen Parametern für nachrichtenorientierte Kommunikation über Annotationen erfolgte, wurde vor der Simulation noch die in Abschnitt 5.4 beschriebene Transformation durchgeführt, um die Entwürfe entsprechend den Annotationen zu modifizieren. In diesen beiden Fällen wird dadurch der Entwurf um die Komponenten der Vervollständigung und des Nachrichtensystems erweitert (siehe Abschnitt 5.3.2 bzw. 5.3.3). Alle drei alternativen Entwürfe wurden mit Java auf Basis von Java EE implementiert und jeweils die Antwortzeiten für die Dienstaufrufe gemessen. Für jene Alternativen, die unter anderem über den Austausch von Nachrichten kommunizieren, erfolgte zusätzlich die Messung der Übertragungsdauer der ausgetauschten Nachrichten. Als Umgebung für die Messungen kommt erneut das in Abschnitt 5.2.3.1 beschrieben System zum Einsatz.

Sowohl die Simulationsergebnisse als auch die Messwerte zeigen für die Alternativen 2 und 3 eine deutliche Verringerung der Antwortzeit für den Aufruf des Dienstes der Komponente `Order_Module`. Die asynchrone Kommunikation zwischen den Komponenten `Order_Module` und `ERP_Module` über den Austausch von Nachrichten führt somit zu einer deutlichen Performance-Steigerung. Diese Verringerung fällt bei der Simulation jedoch deutlicher aus, als dies bei den tatsächlichen Messwerten der Fall ist. Bezogen auf den Median beträgt bei den Messwerten die Reduktion ungefähr

86% für die persistente Nachrichtenübertragung bzw. ca. 89% im nicht-persistenten Fall. Im Vergleich dazu zeigen die Simulationsergebnisse für beide Varianten der nachrichtenorientierten Kommunikation eine Verringerung der Antwortzeit um etwa 94%. Vergleicht man den Median der Simulationsergebnisse mit dem entsprechendem Median der Messwerte (siehe Tabelle 6.1), so lässt für die Alternative 1 eine Abweichung von 1% feststellen. Bei den Alternativen 2 und 3 ist die simulierte Antwortzeit jedoch um 45% bzw. 28% geringer. Abbildung 6.3 zeigt für die Alternative 2 den Vergleich zwischen den Histogrammen der gemessenen und der simulierten Werte. Die Simulationswerte für die Antwortzeit sind dabei ausnahmslos geringer als die gemessenen Werte. Die Ursache liegt vermutlich darin, dass durch den Bestätigungsmechanismus des Nachrichtensystems eine Verzögerung beim Sender auftritt. Dieser Einfluss wurde jedoch im Modell des Nachrichtensystems nicht berücksichtigt, da hierfür Kenntnisse über internen Details des Nachrichtensystems erforderlich sind.

Entwurfs-alternative	**Median der Messwerte**	**Median der Simulationser-gebnisse**	**Abweichung zwischen Simulation und Messung**
Alternative 1	9118393,5ns	9031582ns	0,95%
Alternative 2	1311948,5ns	723550ns	44,85%
Alternative 3	1009638ns	723561ns	28,33%

Tabelle 6.1: Vergleich der Medianwerte für die Antwortzeit des Dienstaufrufs auf der Komponente `Order_Module`

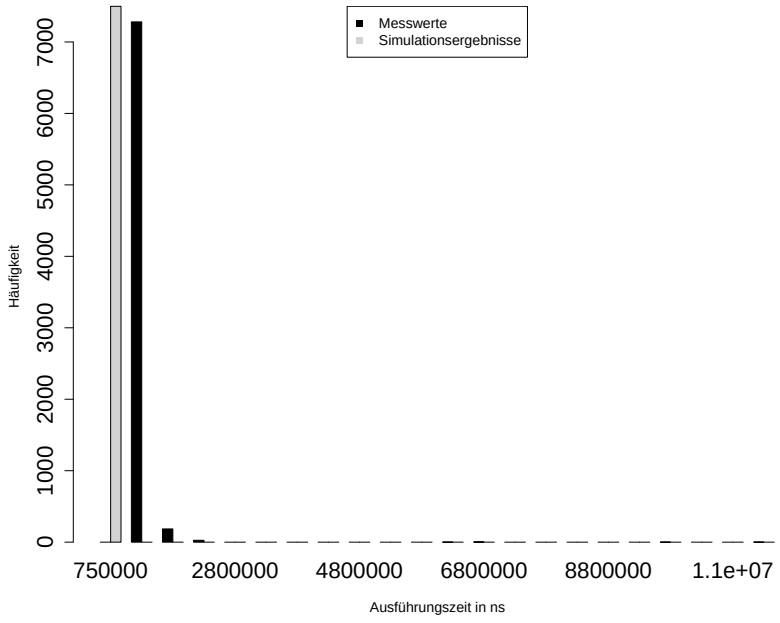


Abbildung 6.3: Histogramm der gemessenen und simulierten Antwortzeit der Dienstaufrufe auf der Komponente `Order_Module` für die Entwurfsalternative 2

Wie Tabelle 6.2 zu entnehmen ist, fallen die Unterschiede bei den Antwortzeiten für die Dienstaufrufe auf der Komponente `CRM_Module` zwischen den Simulations-

Entwurfs-alternative	**Median der Messwerte**	**Median der Simulationser-gebnisse**	**Abweichung zwischen Simulation und Messung**
Alternative 1	184222ns	184459ns	0,13%
Alternative 2	190747,5ns	368914ns	93,40%
Alternative 3	184440ns	368924ns	100,02%

Tabelle 6.2: Vergleich der Medianwerte für die Antwortzeit des Dienstaufrufs auf der Komponente `CRM_Module`

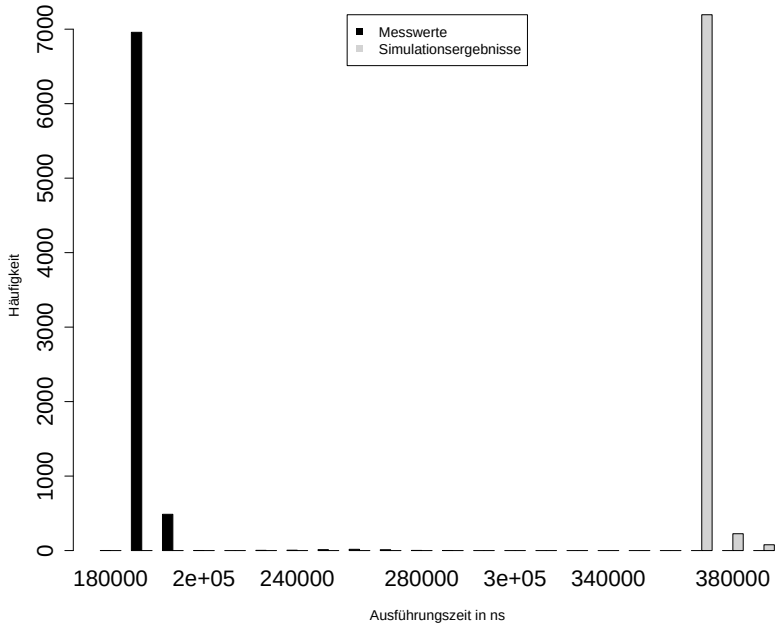


Abbildung 6.4: Histogramm der gemessenen und simulierten Antwortzeit der Dienstaufrufe auf der Komponente `CRM_Module` für die Entwurfsalternative 3

ergebnissen und den gemessenen Werte sehr deutlich aus. Bei Alternative 2 ist die Antwortzeit um 93% höher und bei Alternative 3 verdoppelt sie sich sogar. In Abbildung 6.4 ist dies anhand der Histogramme für die gemessenen und simulierten Antwortzeiten zu erkennen. Der Grund für diese deutlichen Unterschiede liegt im Verhalten des Schedulings. Bei der Simulation teilen sich die internen Aktionen der Komponente `CRM_Module` gleichmäßig die gemeinsame berechnende Ressource des Ressourcen-Containers mit der internen Aktion der Komponente für das Nachrichtensystem. Dadurch ist die Antwortzeit der Dienstaufrufe höher, als wenn die berechnende Ressource exklusiv für die Dauer des Dienstaufrufs zugeteilt wird. Ganz besonders deutlich wird dies beim Vergleich der Antwortzeiten für die Dienstaufrufe auf der Komponente `CRM_Module`, die sich für beiden nebenläufigen Alternativen 2 und 3 im Vergleich zum sequentiellen Fall der Alternative 1 ungefähr verdoppelt. Beim Verfahren der Scheduler von Betriebssystemen bekommen nebenläufige Prozesse den Prozessor jeweils abwechselnd für die Dauer einer Zeitscheibe zugewiesen. Die Messungen erfolgen unter Windows XP mit der Scheduler-Einstellung `Programme`, bei der die Dauer einer Zeitscheibe 30 Millisekunden beträgt [RuSo04, S. 340]. Den Tabellen 6.1 und 6.2 ist anhand der gemessenen Werte zu entnehmen, dass die Dienstaufrufe der Komponente `CRM_Module` im Mittel einen Ressourcenbedarf an die berechnende Ressource haben, der geringer ist als 30 Millisekunden. Dies hat zur Folge dass die Abarbeitung eines Dienstaufrufs immer komplett innerhalb einer Zeitscheibe erfolgt und es niemals zu einer Unterbrechung kommt. Aus diesem Grund sind die gemessenen Antwortzeiten für den Dienstaufruf auf der Komponente `CRM_Module` bei allen drei Alternativen nahezu identisch. Da dieses Scheduling-Verhalten im PCM nicht modelliert werden kann, ergeben sich die deutlichen Unterschiede zu den Simulationsergebnissen.

Für Dienstaufrufe auf der Komponente `ERP_Module` sind die Mediane der Simulationsergebnisse nahezu mit jenen der Messwerte identisch. In Tabelle 6.3 sind diese aufgeführt und zeigen Abweichungen von maximal 2%. Der zuvor beschriebene Scheduling-Effekt tritt hier nicht auf, da während der Ausführung dieses Dienstes alle anderen Dienstausführungen beendet sind, so dass die berechnende Ressource nicht geteilt werden muss. Der Vergleich der Verteilungen in Abbildung 6.5 zeigt für die Alternative 2, dass sich die Simulationswerte stärker um den Median konzentrieren, wohingegen sich die Messwerte auf ein etwas breiteres Intervall verteilen. Dennoch ist die Vorhersagequalität sehr gut.

Die Qualität der Simulationsergebnisse ist bezüglich der Übertragungsdauer von Nachrichten ebenfalls sehr gut. Die in Tabelle 6.4 aufgelisteten Resultate zeigen für die Simulationsergebnisse Abweichungen von 3,3% bei persistentem Nachrichtentransfer und 10,4% wenn keine Nachrichtenpersistierung stattfindet. Für letzteren Fall ist in die Abbildung 6.6 den Vergleich der Verteilung der gemessenen Übertragungsdauerwerte mit jener der Simulationsergebnisse dargestellt. Es zeigt sich, dass nahezu alle Simulationswerte im gleichen Intervall konzentriert sind wie die überwiegende Mehrheit der Messwerte. Der in Abschnitt 5.2.3.2 für den Parameter `GuaranteedDelivery` ermittelte Unterschied zwischen der Persistierung und Nicht-Persistierung von Nachrichten zeigt sich sowohl bei den Messwerten als auch bei den Simulationsergebnissen. Die gemessene Übertragungsdauer reduziert sich beim Verzicht auf die Persistierung von Nachrichten um 27%, bei den Simulationsergebnissen beträgt die Reduzierung ca. 22%.

Entwurfs-alternative	**Median der Messwerte**	**Median der Simulationser-gebnisse**	**Abweichung zwischen Simulation und Messung**
Alternative 1	8757409,5ns	8777270ns	0,23%
Alternative 2	8605662,5ns	8777691ns	2,00%
Alternative 3	8629340,5ns	8777810ns	1,72%

Tabelle 6.3: Vergleich der Medianwerte für die Antwortzeit des Dienstaufrufs auf der Komponente `ERP_Module`

Entwurfs-alternative	**Median der Messwerte**	**Median der Simulationser-gebnisse**	**Abweichung zwischen Simulation und Messung**
Alternative 2	2115955,5ns	2194724ns	3,72%
Alternative 3	1540192ns	1699689ns	10,36%

Tabelle 6.4: Vergleich der Medianwerte für die Übertragungsdauer von Nachrichten

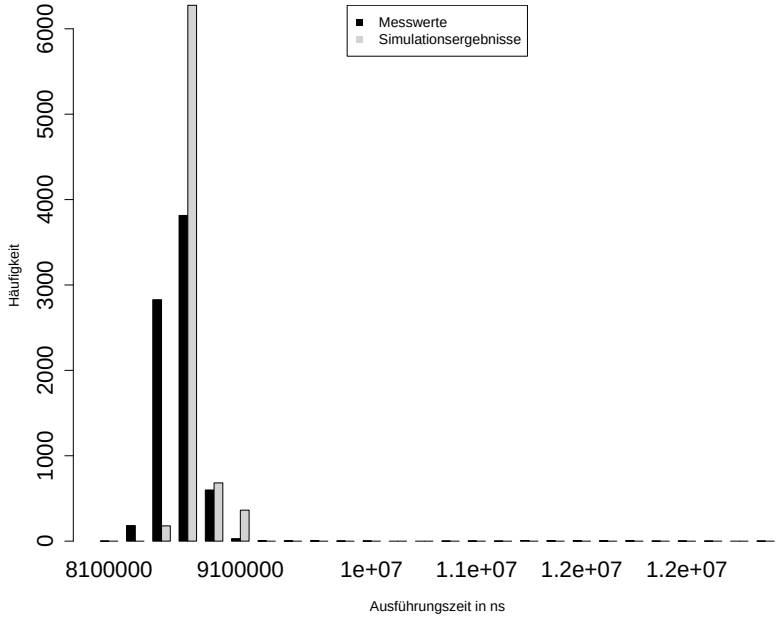


Abbildung 6.5: Histogramm der gemessenen und simulierten Antwortzeit der Dienstaufrufe auf der Komponente `ERP_Module` für die Entwurfsalternative 2

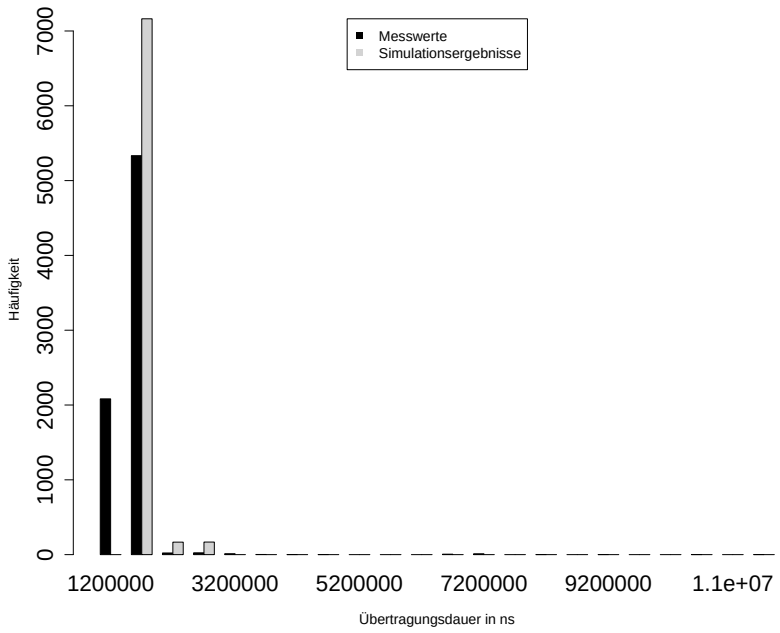


Abbildung 6.6: Histogramm der gemessenen und simulierten Übertragungsdauerwerte für die Entwurfsalternative 3

6.3 Zusammenfassung

Die Fallstudie zeigt, dass sich mit der durchgeführten Integration von nachrichtenorientierter Kommunikation ins Palladio Komponentenmodell gute Performance-Vorhersagen für diese asynchrone Kommunikationsform durchführen lassen. Zum einen stimmen die Vorhersage und die Messwerte in Bezug auf die Performance-Vorteile beim Einsatz nachrichtenorientierter Kommunikation im Vergleich zu strikt sequentieller Ausführung nahezu überein. Zum andren konnte für den Parameter `GuaranteedDelivery` annähernd der gemessene Performance-Einfluss vorhergesagt werden. Aufgrund der großen Anzahl an Parameterkonfigurationen wurde die Evaluierung der Vorhersagequalität in dieser Arbeit exemplarisch für diesen Parameter durchgeführt. Die Bewertung der Vorhersagequalität für die übrigen Performance-relevanten Parameter kann in einer anschließenden umfangreicheren Evaluation erfolgen.
Die festgestellten Schwächen bei der Performance-Vorhersage in Zusammenhang mit der nebenläufigen Nutzung von gemeinsamen Ressourcen ist ist nicht auf die erstellten Konstrukte für die nachrichtenorientierte Kommunikation zurückzuführen. Dies ist vielmehr ein Resultat des unterschiedlichen Scheduling-Verhaltens zwischen dem verwendeten Betriebssystem und der Simulation.

7. Annahmen und Einschränkungen

Für den in dieser Diplomarbeit entwickelten Ansatz wurden für die Modellierung der nachrichtenorientierte Kommunikation einige Annahmen getroffen bzw. Einschränkungen vorgenommen.

Für die nachrichtenorientierte Kommunikation wird angenommen, dass keine Rückgaben erwartet werden. Dies stellt jedoch keine große Einschränkung dar, da es verschiedene Möglichkeiten gibt, Rückgaben zu integrieren. Eine Möglichkeit ist, dass der Nachrichtenempfänger eine Nachricht mit den Rückgabewerten an den Empfänger zurückschickt. JMS bietet die Möglichkeit, dass im Nachrichtenkopf ein Antwortkanal spezifiziert wird. Damit wird dem Nachrichtenempfänger mitgeteilt, über welchen Kanal dieser die Antwortnachricht verschicken muss.

Der Entwurf der erstellten Modellkonstrukte basiert auf den Messergebnissen aus Abschnitt 5.2.3.2 und sind damit spezifisch für die Plattform, auf denen die Messwerte ermittelt werden. Die Modellkonstrukte können jedoch auf andere Plattformen angepasst werden, indem für diese die Messungen durchgeführt werden.

Der Ressourcenbedarf, der in Zusammenhang mit der nachrichtenorientierten Kommunikation entsteht, wurde komplett dem Prozessor als berechnende Ressource zugeschrieben, da ohne Kenntnisse über die internen Abläufe im Nachrichtensystem keine verlässlichen Aussagen getroffen werden können. Die Modellkonstrukte könnten jedoch leicht geändert werden, um den realen Ressourcenbedarf exakter zu modellieren. Beispielsweise könnte beim persistenten Nachrichtentransfer zwischen dem Ressourcenbedarf an den Prozessor und jenem an die Festplatte differenziert werden.

Für den Ressourcenbedarf von nachrichtenorientierter Kommunikation wird angenommen, dass dieser komplett im Nachrichtensystem anfällt. Eine genaue Aufteilung zwischen Sender, Nachrichtensystem und Empfänger konnte nicht durchgeführt werden, da hierfür detaillierte Kenntnisse des verwendeten Nachrichtensystems erforderlich wären.

Für die Modellierung der nachrichtenorientierten Kommunikation wurden die Kriterien Robustheit und Zuverlässigkeit nicht untersucht. Beispielsweise wurden beim transaktionalen Nachrichtenversand nur erfolgreiche Transaktionen betrachtet. Der Grund für diese eingeschränkte Betrachtung ist, dass der Fokus dieser Arbeit auf der Untersuchung der Performance-Einflüsse liegt.

8. Zusammenfassung und Ausblick

In dieser Diplomarbeit wurden Entwurfsmuster für Nebenläufigkeit hinsichtlich ihrer Funktionalität untersucht und klassifiziert. Im Fokus stand dabei ihre Anwendung beim Entwurf von komponentenbasierter Software. Mithilfe der Klassifizierung wird die Musterauswahl beim Software-Entwurf vereinfacht.

Aus der Menge der Entwurfsmuster wurden jene ausgewählt, die Lösungsschemata für Probleme in Zusammenhang mit nachrichtenorientierter Kommunikation bieten. Dies erfolgte mit dem Ziel, die nachrichtenorientierte Kommunikationsform ins Palladio Komponentenmodell zu integrieren, um dadurch eine bessere Unterstützung von Nebenläufigkeit zu erreichen. Für die Integration war es erforderlich, die Performance-relevanten Parameter dieser Muster zu ermitteln. Dies erfolgte, indem diesen Parametern Optionen des Java Message Service (JMS) zugeordnet wurden und anschließend eine Leistungsbewertung dieser JMS-Optionen durchgeführt wurde. Für die Leistungsbewertung der JMS-Optionen wurde eigens eine Anwendung implementiert, die diese Optionen unterstützt und die Übertragungsdauer der Nachrichten ermittelt. Die Integration der Entwurfsmuster erfolgte über das Konzept der Vervollständigung. Demnach werden beim Software-Entwurf extra-funktionale spezifiziert und der Entwurf über Transformationen entsprechend der Parameterspezifikation modifiziert. Dafür wurde ein Annotationsmodell für das Palladio Komponentenmodell erstellt, das die Spezifizierung der ermittelten Performance-relevanten Parametern für nachrichtenorientierte Kommunikation beim Software-Entwurf ermöglicht. Anschließend wurde die Vervollständigung selbst modelliert, die implizit eine Komponente einbezieht, die das Nachrichtensystem repräsentiert. Die Modellierung einer solchen Komponente erfolgte auf Grundlage der Messwerte für die JMS-Optionen. Hierbei standen besonders die Modellierung der Asynchronität von nachrichtenorientierten Kommunikation und das Verhalten bei unterschiedlichen Performance-relevanten Parametern im Mittelpunkt. Den Abschluss der Integration bildete die Beschreibung einer Transformation, die Instanzen des Annotationsmodells auswertet und den Software-Entwurf um die Vervollständigung und die Komponenten für das Nachrichtensystem erweitert.

Im Rahmen einer Fallstudie wurde die Qualität der Performance-Vorhersagen unter

Berücksichtigung der Modellkonstrukte für nachrichtenorientierte Kommunikation bewertet. Dazu wurden für drei Entwurfsalternativen die Ergebnisse der Simulationen mit den gemessenen Werten verglichen. Es zeigte sich, dass mit den erstellten Modellkonstrukten verlässliche Performance-Vorhersagen getroffen werden können. Die Vorhersage der Übertragungsdauer von Nachrichten wich um maximal 10% von den gemessenen Werten ab. Die Bewertung der Performance-Vorhersage für alle weiteren Parameterkonfigurationen kann bei einer anschließenden umfangreichen Evaluierung durchgeführt werden.

Bei der Ermittlung der Performance-relevanten Parameter konnte für zehn JMS-Optionen aussagekräftige Ergebnisse erzielt werden. Die Bestimmung des Einflusses des Nachrichtentyps konnte nicht endgültig durchgeführt werden. Darüber hinaus konnte der Einfluss einer Netzwerkverbindung auf die nachrichtenorientierte Kommunikation nicht eindeutig bestimmt werden. Für beides bietet es sich an, tiefer gehende Untersuchungen anzustellen und die erstellten Modellkonstrukte um diese Aspekte zu erweitern. Damit kann die Vorhersagequalität für nachrichtenorientierte Kommunikation erhöht werden.

Weitere Verbesserungen bei der Performance-Vorhersage sind von einem Schedulermodell zu erwarten. Eine Erweiterung des Palladio Komponentenmodells um asynchrone Dienstaufrufe hätte eine deutliche Vereinfachung der Konstrukte für die nachrichtenorientierte Kommunikation zur Folge.

Die erstellten Modellkonstrukte sind bisher sehr plattformspezifisch, da sie direkt auf Basis der Messwerte modelliert wurden. Dennoch bietet der Ansatz ein hohes Automatisierungs- und Anpassungspotenzial, indem für die gewünschte Zielplattform ein Benchmarking des Nachrichtensystems durchgeführt wird und die Modellkonstrukte automatisch diesen Ergebnissen angepasst werden. Dafür kann die genetische Programmierung für die Analyse des Verhaltens des Nachrichtensystems eingesetzt werden. Die automatische Anpassung der Modellkonstrukte kann während der Transformation des annotierten Software-Entwurfs erfolgen.

Das entwickelte Verfahren zur Integration von nachrichtenorientierter Kommunikation in das Palladio Komponentenmodell kann für die übrigen Entwurfsmuster für Nebenläufigkeit angewendet werden, um damit eine breite Unterstützung von Nebenläufigkeit im Palladio Komponentenmodell zu erreichen.

A. Messergebnisse

In Abschnitt 5.2.3.2 wurden in Zusammenhang mit der Erläuterung der Messergebnisse für einige der untersuchten Konfigurationen der Verlauf der Antwortzeiten und deren Verteilung graphisch dargestellt. Die folgenden Abbildungen zeigen besonders charakteristische Verläufe oder Histogramme für die Konfigurationen „Queue mit transaktionalem Nachrichtenversand“, „Queue mit Empfänger-Pool“, „Queue mit entferntem Nachrichtensystem“, „Queue mit entferntem Empfänger“ und „Topic ohne dauerhafte Empfängerregistrierung“.

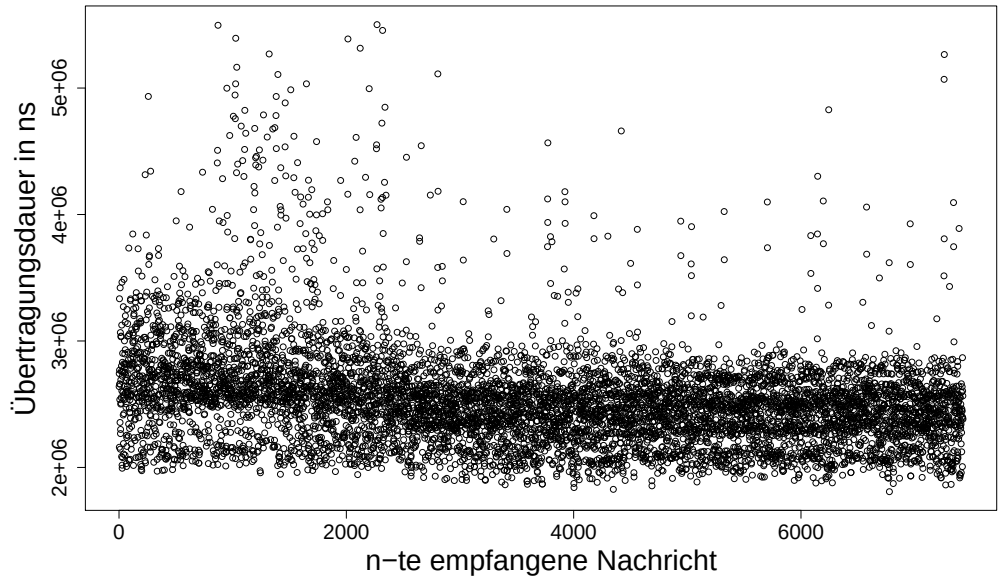


Abbildung A.1: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand„ bei 2 Nachrichten pro Transaktion

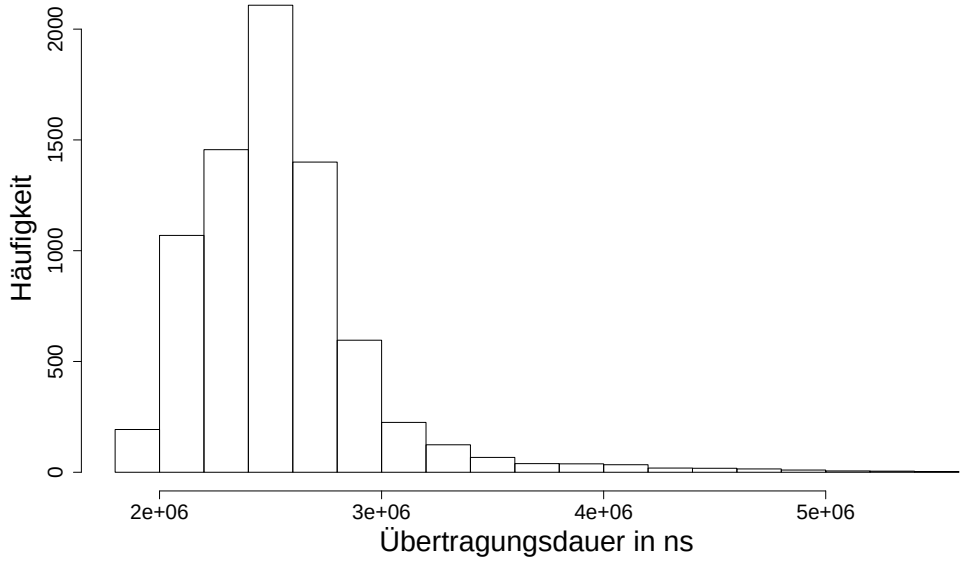


Abbildung A.2: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand„ bei 2 Nachrichten pro Transaktion

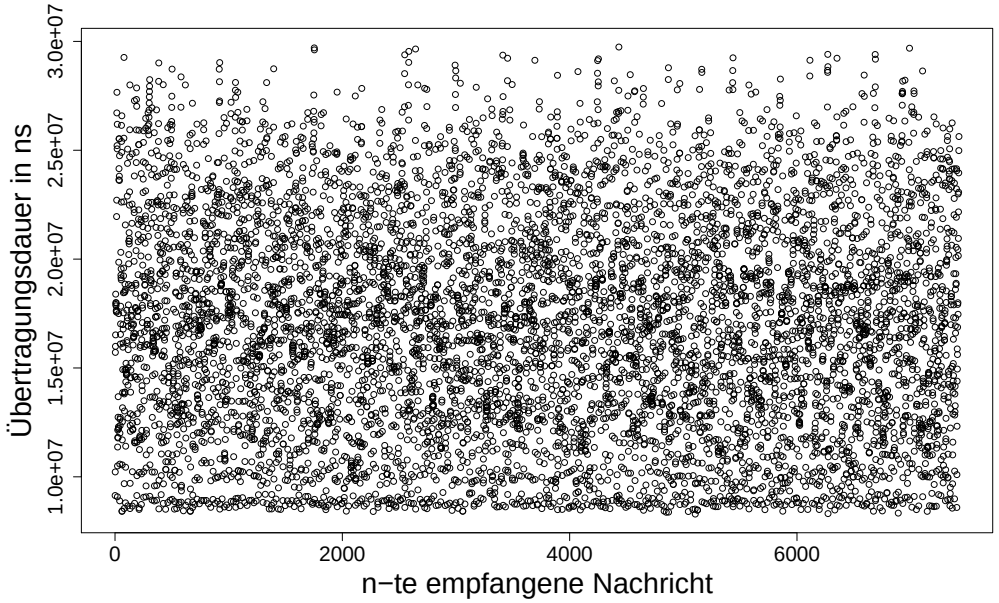


Abbildung A.3: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand„ bei 20 Nachrichten pro Transaktion

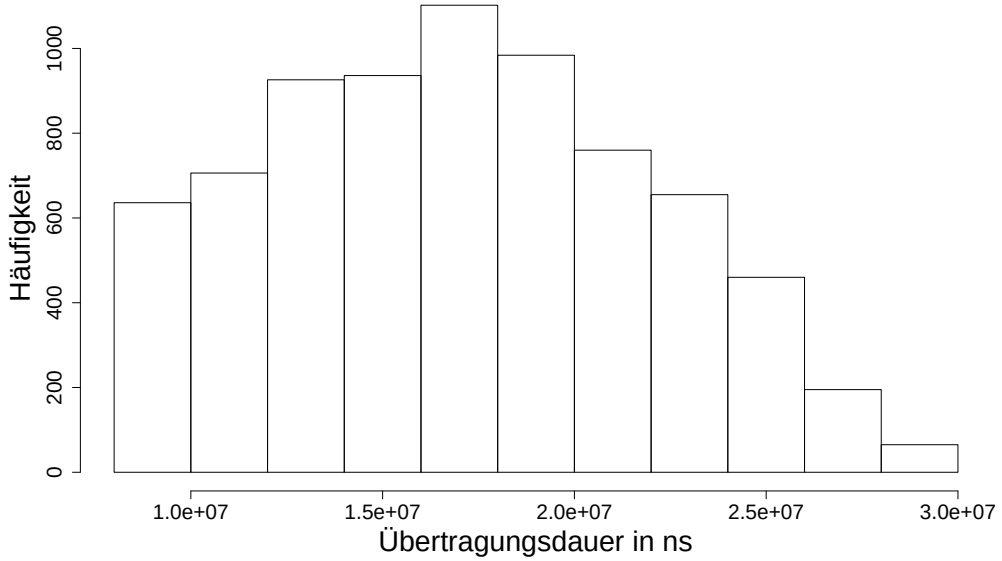


Abbildung A.4: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit transaktionalem Nachrichtenversand„ bei 20 Nachrichten pro Transaktion

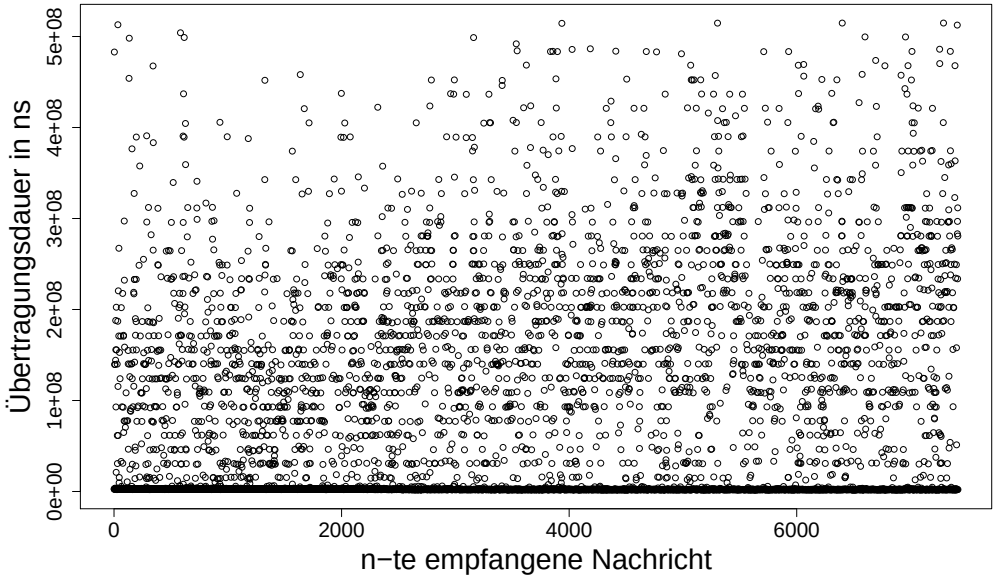


Abbildung A.5: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit Empfänger-Pool„ bei einer festen Poolgröße von 8 Empfängern

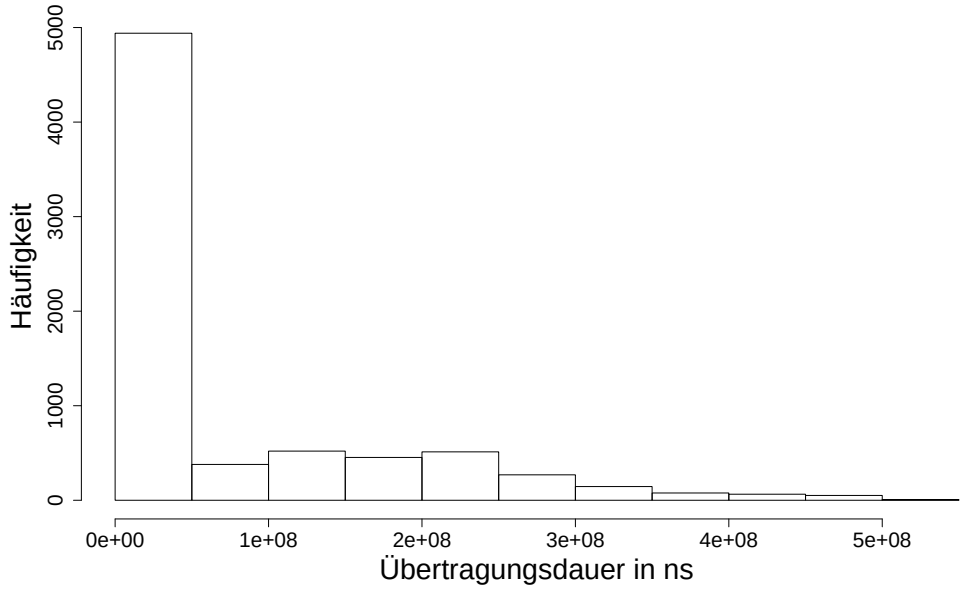


Abbildung A.6: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit Empfänger-Pool„ bei einer festen Poolgröße von 8 Empfängern

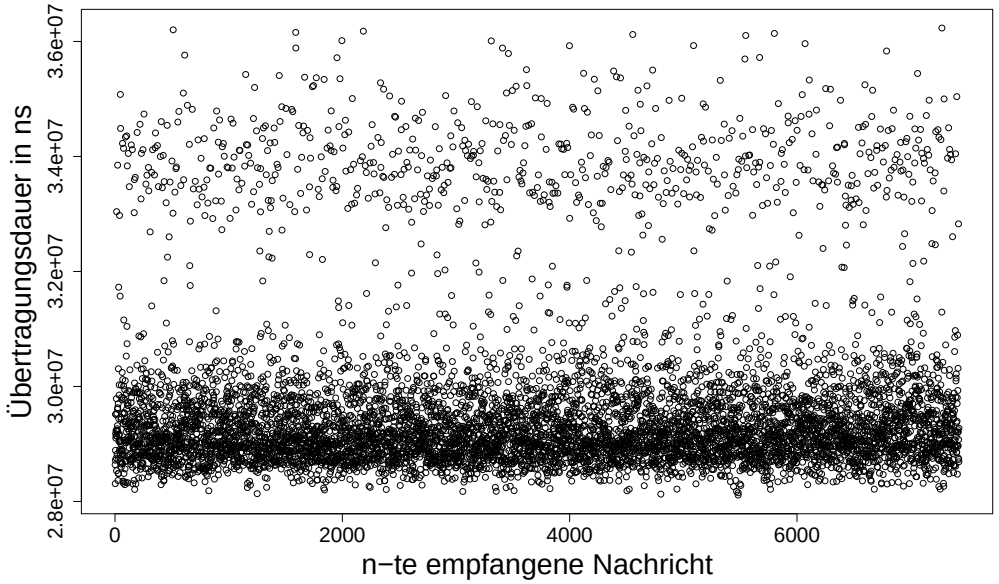


Abbildung A.7: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit entferntem Nachrichtensystem„ bei einem Nachrichteninhalt von 100000 Zeichen

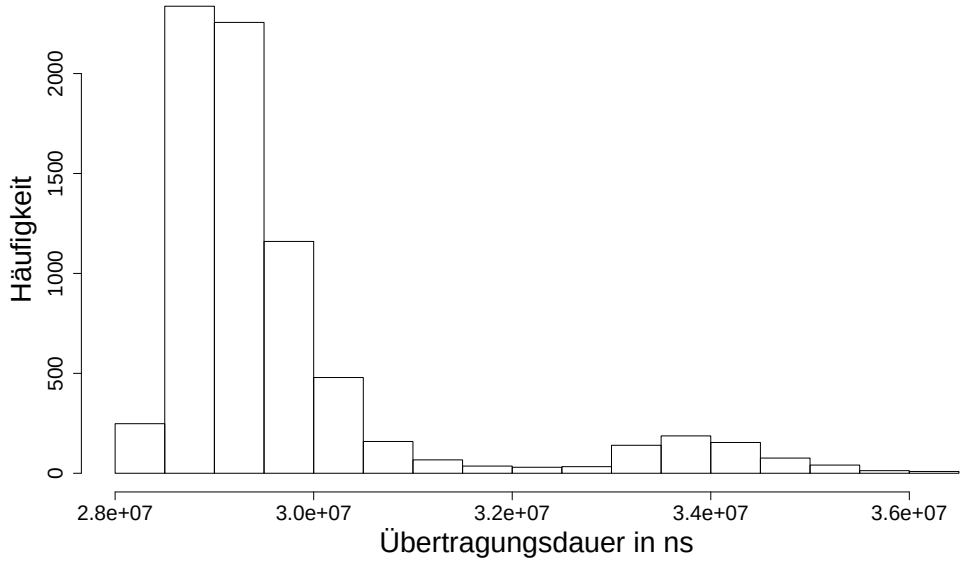


Abbildung A.8: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit entferntem Nachrichtensystem„ bei einem Nachrichteninhalt von 100000 Zeichen

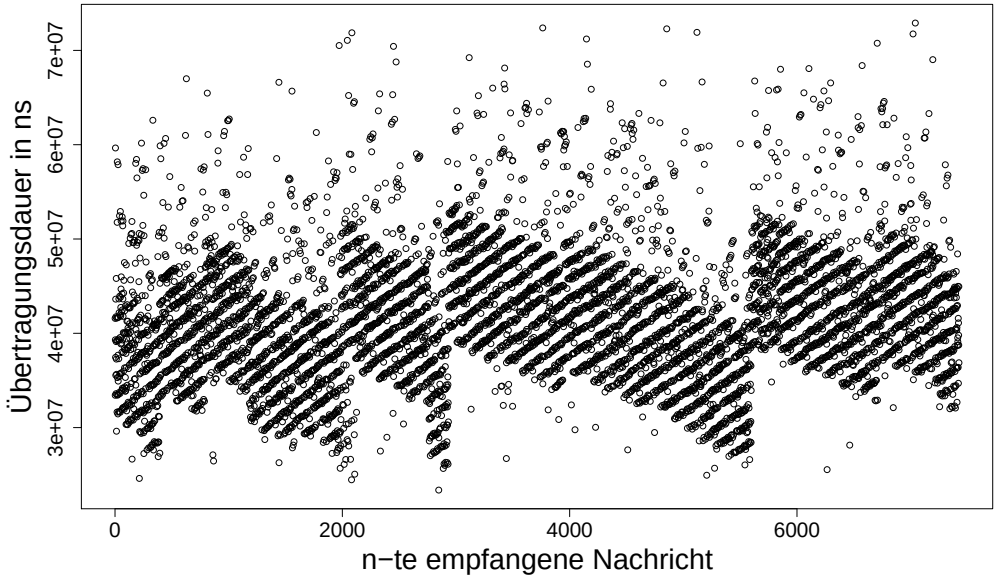


Abbildung A.9: Verlauf der Übertragungsdauer für die Konfiguration „Queue mit entferntem Empfänger„ bei einem Nachrichteninhalt von 1000 Zeichen

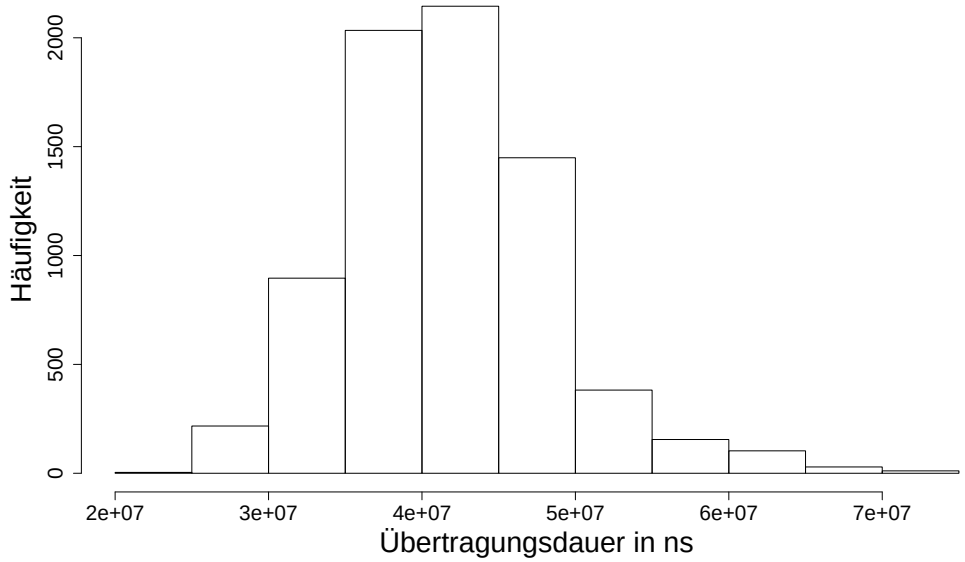


Abbildung A.10: Histogramm der Übertragungsdauer für die Konfiguration „Queue mit entferntem Empfänger„ bei einem Nachrichteninhalt von 1000 Zeichen

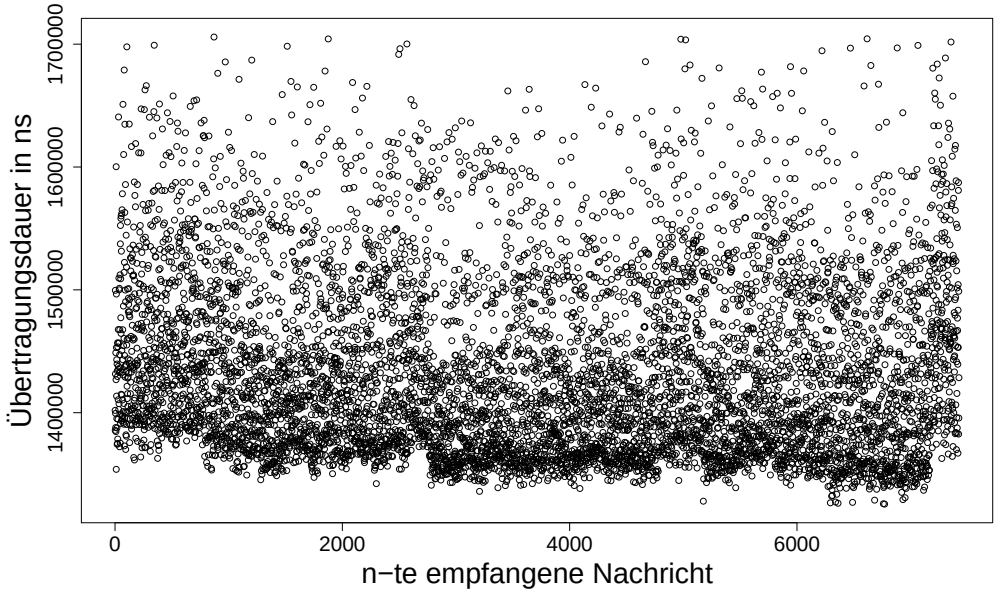


Abbildung A.11: Verlauf der Übertragungsdauer für die Konfiguration „Topic ohne dauerhafte Empfängerregistrierung„

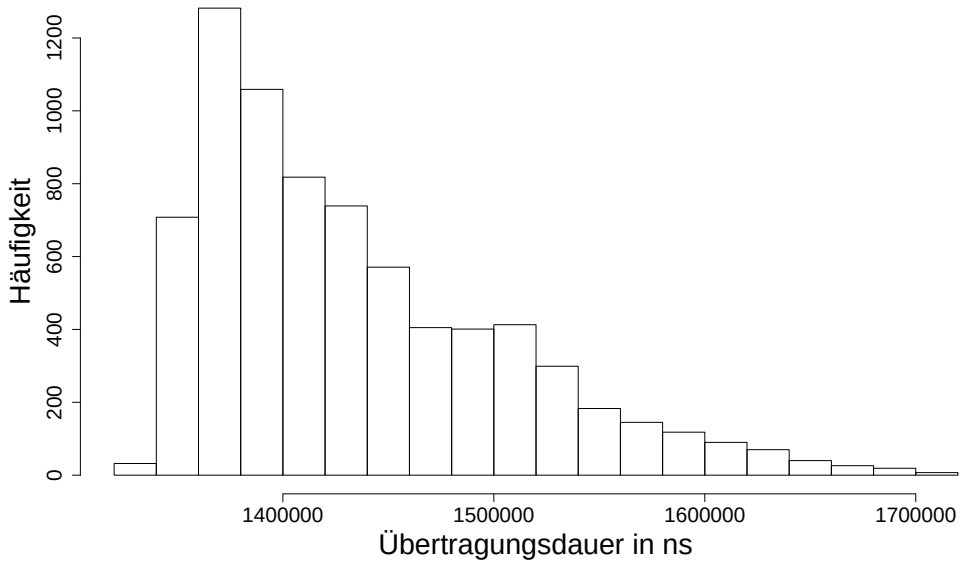


Abbildung A.12: Histogramm der Übertragungsdauer für die Konfiguration „Topic ohne dauerhafte Empfängerregistrierung„

Literatur

[AlIS77] Christopher Alexander, Sara Ishikawa und Murray Silverstein. *A Pattern Language*. Oxford University Press. August 1977.

[BeaA] BEA WebLogic Server 10.0 Documentation - Administration Console Online Help. [Online; accessed 2007-08-04].

[BeaM] BEA WebLogic Server 10.0 Documentation - Configuring and Managing the WebLogic Messaging Bridge. [Online; accessed 2007-08-04].

[BMRS+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA. 1996.

[BuMH06] Bill Burke und Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc. 2006.

[ChGr04] Shiping Chen und Paul Greenfield. QoS Evaluation of JMS: An Empirical Approach. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, Washington, DC, USA, 2004. IEEE Computer Society, S. 90276.2.

[CoDK05] Coulouris, Jean Dollimore und Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2005.

[Cree05] Mache Creeger. Multicore CPUs for the masses. *Queue*, 3(7), 2005, S. 64–ff.

[CzEi00] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 2000.

[dBur02] Mathijs den Burger. Design Patterns for Networking Applications in Java, January 2002. [Online; accessed 2007-05-06].

[Desi] Design/CPN. [Online; accessed 2007-06-20].

[Doug04] Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. 3. Auflage, 2004.

[EiSm03] Agoston E. Eiben und J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag. 2003.

[Ente06] Enterprise JavaBeans 3.0 Specification- version 1.1, May 2006. [Online; accessed 2007-05-09].

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1995.

[GPBB+05] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional. May 2005.

[Grou05] Object Management Group. UML Profile for Schedulability, Performance, and Time, v1.1, January 2005. [Online; accessed 2007-06-18].

[Hend05] Brian Henderson-Sellers. UML - the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and System Modeling*, 4(1), 2005, S. 4–13.

[Hild02] Gerald H. Hilderink. A Graphical Modeling Language for Specifying Concurrency based on CSP. In James Pascoe, Roger Loader und Vaidy Sunderam (Hrsg.), *Communicating Process Architectures 2002*, sep 2002, S. 255–284.

[Hoar83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1), 1983, S. 100–106.

[HoWo03] Gregor Hohpe und Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2003.

[Java] Java Message Service Tutorial. [Online; accessed 2007-08-07].

[Java02] Java Message Service Specification - version 1.1, April 2002. [Online; accessed 2007-05-07].

[Java06] Java Platform, Enterprise Edition 5 (Java EE 5) Specification, May 2006. [Online; accessed 2007-09-09].

[JeKW07] Kurt Jensen, Lars Michael Kristensen und Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2007.

[JNDI] Java Naming and Directory Interface. [Online; accessed 2007-08-27].

[Kahn74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld (Hrsg.), *Information processing*, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam, S. 471–475.

[Kupe07] Michael Kuperberg. Java TimerResolutionRetriever, University of Karlsruhe, 2007.

[KWSS+04] Santosh Kumar, Bruce W. Weide, Paolo A.G. Sivilotti, Nigamanth Sridhar, Jason O. Hallstrom und Scott M. Pike. Encapsulating Concurrency as an Approach to Unification. In *FSE Workshop on Specification and Verification of Component-Based Systems*, Newport Beach, CA, October 2004.

[Lee03] Edward A. Lee. Overview of the Ptolemy Project. Technischer Bericht UCB/ERL M03/25, EECS Department, University of California, Berkeley, Jul 2003.

[Lee06] Edward A. Lee. The Problem with Threads. *Computer*, 39(5), 2006, S. 33–42.

[LiFG05] Yan Liu, Alan Fekete und Ian Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11), 2005, S. 928–941.

[OlHa05] Kunle Olukotun und Lance Hammond. The future of microprocessors. *Queue*, 3(7), 2005, S. 26–29.

[PeGo04] Robert G. Pettit und Hassan Gomaa. Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets. *WICSA*, Band 00, 2004, S. 57–68.

[PeGo06] Robert G. Pettit und Hassan Gomaa. Modeling behavioral design patterns of concurrent objects. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, New York, NY, USA, 2006. ACM Press, S. 202–211.

[PeSo97] Dorina Petriu und Gurudas Somadder. A Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers. In *Proceedings of EuroPLoP'97*, July 1997.

[PiSB03] Peter R. Pietzuch, Brian Shand und Jean Bacon. A Framework for Event Composition in Distributed Systems. In *Proc. of the 4th Int. Conf. on Middleware (MW'03)*, Rio de Janeiro, Brazil, June 2003.

[RBKH+07] Ralf Reussner, Steffen Becker, Heiko Koziolek, Jens Happe, Michael Kuperberg und Klaus Krogmann. The Palladio Component Model. Technischer Bericht, University of Karlsruhe, 2007. To be published.

[RuSo04] Mark E. Russinovich und David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer).* Microsoft Press, Redmond, WA, USA. 2004.

[SaHe06] Lothar Sachs und Jürgen Hedderich. *Angewandte Statistik.* Springer-Verlag, Berlin Heidelberg, Germany. 2006.

[SiPH06] Sylvain Sicard, Noel De Palma und Daniel Hagimont. J2EE server scalability through EJB replication. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, New York, NY, USA, 2006. ACM Press, S. 778–785.

[SJSM] Sun Java System Message Queue 3.7 UR1 Administration Guide - Application Design Factors Affecting Performance. [Online; accessed 2007-08-15].

[SKBB07] Kai Sachs, Samuel Kounev, Jean Bacon und Alejandro Buchmann. Workload Characterization of the SPECjms2007 Benchmark. In *EPEW*, Lecture Notes in Computer Science, Berlin Heidelberg, Germany, 2007. Springer-Verlag, S. 228–244.

[SKCB07] Kai Sachs, Samuel Kounev, Marc Carter und Alejandro Buchmann. Designing a Workload Scenario for Benchmarking Message-Oriented Middleware. In *Proceedings of the 2007 SPEC Benchmark Workshop*, Austin, Texas, Januar 2007. SPEC.

[Smit90] Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1990.

[SmWi02] C. Smith und L. Williams. Performance and Scalability of Distributed Software Architectures: an SPE Approach, 2002.

[SRSS+00] Douglas C. Schmidt, Hans Rohnert, Michael Stal, Dieter Schultz und Frank Buschmann. *Pattern-Oriented Software Architecture*, Band Vol. 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc., New York, NY, USA. 2000.

[SuLa05] Herb Sutter und James Larus. Software and the concurrency revolution. *Queue*, 3(7), 2005, S. 54–62.

[Sutt05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[Szyp02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2002.

[Tane02] Andrew S. Tanenbaum. *Moderne Betriebssysteme*, Band 2. Pearson Education Deutschland GmbH. 2002.

[WoPS02] Murray Woodside, Dorin Petriu und Khalid Siddiqui. Performance-related completions for software specifications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, 2002. ACM Press, S. 22–32.

[ZdAv05] Uwe Zdun und Paris Avgeriou. Modeling architectural patterns using architectural primitives. *SIGPLAN Not.*, 40(10), 2005, S. 133–146.

Index