Skalierbare Ausführung von Prozessanwendungen in dienstorientierten Umgebungen

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Wirt.-Inf.(FH) Steffen Preißler
geboren am 22. September 1979 in Freital

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Arbeitsgruppe Datenbanken
39016 Magdeburg

Tag der Verteidigung: 25. Oktober 2012

Dresden, im November 2012

Zusammenfassung

Die Strukturierung und Nutzung von unternehmensinternen IT-Infrastrukturen auf Grundlage dienstorientierter Architekturen (engl. *service-oriented architecture*, *SOA*) und etablierter XML-Technologien ist in den vergangenen Jahren stetig gewachsen. Lag der Fokus anfänglicher SOA-Realisierungen auf der flexiblen Ausführung klassischer, unternehmensrelevanter Geschäftsprozesse, so bilden heutzutage zeitnahe Datenanalysen sowie die Überwachung von geschäftsrelevanten Ereignissen weitere wichtige Anwendungsklassen.

Aufgrund der datenzentrischen Charakteristik dieser neuen Anwendungsklassen und der dafür ungeeigneten kontrollflussbasierten Ausführung klassischer Geschäftsprozesse, werden diese Anwendungsklassen jedoch häufig mit spezialisierten, orthogonalen Systemen umgesetzt und den Geschäftsprozessen nur über Dienstschnittstellen bereitgestellt. Dies führt zu einer Vielzahl heterogener Anwendungssysteme und strikt getrennten Anwendungsteilen. Die Kernpunkte einer SOA – die flexible Kombination verteilter Dienste und die konsequente Ausrichtung an Prozessflüssen – lassen es daher sinnvoll erscheinen, solche Anwendungen ebenfalls als SOA-basierte Prozesse abzubilden.

Aus diesem Grund beschäftigt sich die vorliegende Arbeit mit dem Problem der skalierbaren prozessorientierten Ausführung dieser Anwendungsklassen in einer SOA. Dabei werden die Probleme datenzentrischer Anwendungen bei der klassischen, kontrollflussbasierten Prozessausführung auf drei Ebenen adressiert. Auf der ersten Ebene wird die Kommunikation zwischen einzelnen Diensten betrachtet. Es wird ein Konzept präsentiert, welches durch die Integration von Datenstromsemantik eine skalierbare Verarbeitung großer Datenmengen durch Webdienste erlaubt. Dadurch können Dienste datenzentrische Funktionen bereitstellen, welche auf großen, strukturierten Datenmengen verteilt arbeiten. Auf der zweiten Ebene, der Ebene der Prozessausführung, werden die Datencharakteristiken der Anwendungsklassen analysiert und ein datenstrombasiertes Verarbeitungsmodell abgeleitet. Das zuvor entwickelte Konzept der Dienstebene wird dabei aufgegriffen und in das Verarbeitungsmodell integriert. Abschließend wird die Ebene der physischen Prozess- und Dienstverteilung betrachtet. Hierbei wird ein Konzept vorgestellt, welches durch das Zusammenbringen von Prozess- und Dienstverarbeitung auf gemeinsamen Serverknoten den Netzwerktransfer und den XML-Overhead reduziert. Dadurch werden Netzwerkkapazitäten für andere Aufgaben freigehalten und der Durchsatz von Prozessausführungen erhöht.

Danksagung

Zum erfolgreichen Abschluss dieser Arbeit haben eine Reihe von Personen auf verschiedene Art und Weise beigetragen. Diesen Personen möchte ich hiermit danken.

Mein größter Dank gilt meinem Doktorvater Herrn Prof. Dr.-Ing. Wolfgang Lehner für die Möglichkeit, an seinem Lehrstuhl zu promovieren; für seine zahlreichen Ideen und Anregungen, für die mir gegebenen wissenschaftlichen Freiräume und seine fortwährende Unterstützung. Ich bedanke mich für das in mich gesetzte Vertrauen und nicht zuletzt für die angenehme Zusammenarbeit. Meinem Zweitgutachter Herrn Prof. Dr. Gunter Saake danke ich für die zügige Begutachtung meiner Arbeit und für die hilfreichen Tipps bei der Vorstellung des Themas in Magdeburg.

Weiterhin möchte ich mich bei meinen ehemaligen Kollegen am Lehrstuhl für Datenbanken der Technischen Universität Dresden bedanken, die mit einem angenehmen und kreativen Arbeitsklima wesentlich zu dieser Dissertation beigetragen haben. Ein besonderer Dank gilt dabei Dr.-Ing. Dirk Habich, mit dem ich regelmäßig bei intensiven und kreativen Gesprächen im Büro, am Kaffeeautomaten oder auf Dienstreisen neue Ideen diskutiert und auch wieder verworfen habe. Vielen Dank für die wertvolle Unterstützung während meiner Zeit am Lehrstuhl. Ich danke Frank Rosenthal, der vor allem in der Schreibphase mit fachlichen Diskussionen und weiterführenden, nichtfachlichen Gesprächsthemen zum Gelingen dieser Arbeit beitrug. Er schrieb während dieser Zeit ebenfalls an seiner Arbeit und teilte mit mir Freud und Leid. Außerdem danke ich Dr.-Ing. Maik Thiele, Dr.-Ing. Matthias Böhm und Hannes Voigt, die mir insbesondere während der Anfangsphase meiner Dissertation mit wertvollem Rat zur Seite standen. Meinen ehemaligen Studenten Markus Dumat und Fanny Dittmar gilt mein Dank für ihre engagierte Arbeit und ihre Hilfe bei der prototypischen Umsetzung der in dieser Arbeit entwickelten Konzepte. Weiterhin danke ich auch Katrin Braunschweig, Ulrike Fischer, Martin Hahmann und Benjamin Schlegel für die unterhaltsamen Kaffeepausen, die gemeinsamen sportlichen Unternehmungen und für erlebnisreiche Lehrstuhl-Wandertage.

Meiner Familie und meinen Freunden danke ich für Ihre Unterstützung, Ihr Verständnis für häufig abgesagte Termine, den Zuspruch und die vielen Dinge, durch die ich neue Energie schöpfte. Ein ganz besonderer Dank gilt meiner Ehefrau Antonia – für ihre langjährige Unterstützung, ihre Liebe und ihre Geduld. Ihre Korrekturen, ihre aufmunternden Worte sowie die interdisziplinären Diskurse mit ihr während der Schreibphase waren für die Erstellung dieser Arbeit von unschätzbarem Wert.

Abschließend möchte ich meinem Vater Ekkehard Preißler danken, der die Fertigstellung dieser Arbeit leider nicht mehr miterleben kann. Ich danke ihm für seine

vorbehaltlose Unterstützung auf meinem bisherigen Lebensweg und den Glauben an mich. Er hat mir damit so viele Dinge ermöglicht. Ihm widme ich diese Arbeit von ganzem Herzen.

Steffen Preißler
31. August 2012

Inhaltsverzeichnis

1 Einleitung

Die Strukturierung und Nutzung von IT-Systemen in Unternehmen nach dem Konzept dienstorientierter Architekturen (engl. *service-oriented architecture*, SOA) ist in den vergangenen Jahren stetig gewachsen [78, 180, 181]. Wenngleich der Grundgedanke einer SOA bereits Mitte der 90er Jahre beschrieben wurde, so begann ihr Aufstieg erst mit der Entwicklung der Extensible Markup Language (XML) [200] und der darauf aufbauenden Web Service-Spezifikationen [209]. Die Kernidee einer SOA liegt darin, alle relevanten Anwendungsfunktionen eines IT-Systems als Dienste verteilt in einem Netzwerk bereitzustellen und deren Funktionen standardisiert und implementierungsunabhängig zu beschreiben. Die Kommunikation mit den Diensten erfolgt lose gekoppelt über Nachrichtenaustausch in standardisierten Formaten und Protokollen. Langfristig, so die Hoffnung, sollen dadurch die heute vorherrschenden monolithischen Anwendungssysteme abgelöst werden [114]. Die Abstraktion von Funktionalität zu lose gekoppelten Diensten mit standardisierten Schnittstellenbeschreibungen hat mehrere Vorteile. Zum einen wird dadurch die einfache Wiederverwendung von Dienstkomponenten in einer Vielzahl von Anwendungen möglich. Zum anderen können diese Dienste flexibel zu höherwertigen Prozessen komponiert werden. Die Abbildung von realen, wirtschaftlichen Geschäftsprozessen (engl. *business processes*) auf äquivalente technische, dienstbasierte Prozesse wird dadurch vereinfacht und erlaubt eine flexible Synchronisierung beider Welten. Anbieter traditioneller Systeme für automatisierte Arbeitsabläufe (engl. *workflow)* begannen, ihre Produkte für die Geschäftsprozessmodellierung und Ausführung auf Basis der Web Service-Spezifikation anzupassen und zu vermarkten [196].

Lag der Fokus anfänglicher SOA-Realisierungen der Hersteller also zunächst auf der Transformation monolithischer IT-Systeme hin zu einer Geschäftsprozessorientierung auf Basis von Dienstkomponenten, so wurde in den folgenden Jahren der Begriff des Business Process Management (BPM) geprägt. Dieser beschreibt die Verwaltung des gesamten Lebenszyklus von Diensten und Prozessen und betrachtet alle Phasen von der Konzipierung über die Entwicklung und Nutzung bis zum Feedback und den daraus folgenden Aktualisierungen der Dienstkomponenten und Prozesse. In den letzten Jahren wurden, neben dem Ausbau dieser ganzheitlichen Sichtweise, auch zwei weitere Anwendungsklassen mit der SOA-Infrastruktur und den darin laufenden dienstbasierten Prozessen verknüpft, um die dienstbasierte Verarbeitung der zentralen geschäftsrelevanten Prozesse zu unterstützen [121, 195].

Die erste dieser weiteren Anwendungsklassen beinhaltet die Anwendungstypen *Datenintegration* und *Datenanalyse* (engl. *data integration and analytics* (DIA)). *Da-*

tenintegration bildet dabei einen der klassischen und wichtigsten Anwendungstypen in Unternehmen. Dabei steht die Konsolidierung und Bereinigung von Daten heterogener Datenquellen und Formate im Fokus. Klassische Systeme zur unternehmensinternen Datenintegration sind Extraktion-Transformation-Load (ETL)-Systeme, welche für diese Konsolidierung entwickelt wurden und meist in Data Warehouse-Umgebungen Anwendung finden [98, 119, 141, 153]. Die Notwendigkeit dieser Anwendungen hat sich durch die dienstorientierte Strukturierung der IT-Systeme nicht verringert und bleibt ein integraler Bestandteil der Unternehmensinfrastruktur [134]. Der Anwendungstyp *Datenanalyse* hat sich im Gegensatz dazu in den letzten Jahren stark geandelt. Trends zur zeitnahen Analyse von Unternehmensdaten, zu deren Visualisierung sowie zur daraus resultierenden Unterstützung zeitnaher Entscheidungsfindung lassen sich in derzeitigen Anwendungspaketen verschiedener Hersteller erkennen [190, 150]. Durch die intensive Nutzung konsolidierter Ergebnisse dieser Anwendungsklasse in den zentralen *SOA-basierten Prozessen* lässt sich auch ein immer stärkerer Wunsch zur direkten Integration von Teilprozessen der DIA-Anwendungsklasse in die *SOA-basierten Prozesse* feststellen [195].

Die zweite, weitere wichtige Anwendungsklasse ist die Überwachung geschäftsrelevanter Ereignisse (engl. *business activity monitoring* (BAM)) mithilfe der Nachrichtenstromanalyse [36, 37]. Grundlage bilden Nachrichtenströme, die innerhalb der Ausführungsumgebungen einer SOA generiert werden und Statusinformationen über die einzelnen Dienst- und Prozesskomponenten enthalten. Ziel der Nachrichtenstromanalyse in SOA-Umgebungen ist es dabei, geschäftsrelevante Metriken von Diensten, Prozessen und anderen Infrastrukturkomponenten zu überwachen. Entsprechende Stromsysteme werten die Nachrichtenströme aus und können über die Aggregation von Metriken sowie über eine zeitbasierte Mustererkennung (engl. *complex event processing* (CEP)) weiterführende Aktionen auslösen. Auch in dieser Anwendungsklasse lässt sich durch die starke Interaktion dieser Klasse mit der zentralen Klasse der technischen, dienstbasierten Geschäftsprozesse der Wunsch zu integrierten Ausführungsplattformen und damit zur Integration von Teilprozessen dieser Anwendungsklasse erkennen [195].

Im Gegensatz zur kontrollflussorientierten Ausführung von *SOA-basierten Geschäftsprozessen* haben Anwendungen der beiden beschriebenen Klassen ihren Ursprung im Datenmanagementbereich und basieren auf dort etablierten Modellierungsmethodiken und einer datenflussorientierten Ausführung. Diese Eigenschaften ermöglichen es den Ausführungssystemen, die Datencharakteristiken ihrer Anwendungen effizient zu unterstützen. Jedoch begründet dies auch die Tatsache, dass Anwendungen dieser beiden Klassen anders als die zentralen SOA-basierten Geschäftsprozessen sowohl bei ihrer Anwendungsentwicklung als auch bei ihrer Ausführung mit orthogonalen Methodiken und Systemen umgesetzt werden. Da die Resultate der Anwendungen jedoch für die eigentlichen, SOA-basierten Prozesse relevant sind, werden sie meist nachträglich als Dienstaufruf in die Prozessausführung integriert. Dies führt zum einen zu strikt getrennten Modellierungs- und Ausführungsumgebungen. Zum anderen sind dadurch eine Vielzahl heterogener Anwendungssysteme involviert, deren

gegenseitige Interaktion und Integration oft als fehleranfällig und schwer wartbar bewertet wird [195, 197]. Die Kernpunkte einer SOA – die flexible Kombination verteilter Dienste einerseits und die konsequente Ausrichtung an Prozessflüssen andererseits – sowie der Trend zu datenintensiveren Geschäftsprozessen [70, 95] lassen es daher sinnvoll erscheinen, solche Anwendungen ebenfalls als SOA-basierte Prozesse abzubilden. Eine solche ganzheitliche dienstbasierte Umsetzung der Anwendungsklassen ist derzeit jedoch durch die Ausführungssemantik von Geschäftsprozessen sowie durch die Verwendung XML-basierter Austauschformate nicht direkt möglich.

In dieser Arbeit werden deshalb Methoden und Techniken entwickelt, welche die Vorteile datenflussorientierter Ausführungskonzepte aus dem Datenmanagementbereich in die klassische Prozess- und Dienstwelt integrieren. Globale Zielstellung ist dabei, eine einheitliche Ausführungsplattform auf Dienst- und Prozessebene zu entwickeln, welche die skalierbare Verarbeitung von Datenintegration- und Nachrichtenstromanalyseprozessen in einer service-orientierten Umgebung und deren prozess- und dienstbasierten Eigenschaften ermöglicht. Da die XML-basierte Nachrichtenkommunikation zudem den größten Engpass bei der dienstbasierten Verarbeitung darstellt [50, 108, 152], wird zusätzlich die kostenbasierte Platzierung von Prozessen zu abhängigen Diensten adressiert, welche die entfernte Kommunikation reduziert und damit weitere Performancesteigerungen ermöglicht.

Beiträge der Arbeit

Unter den im vorherigen Abschnitt beschriebenen Rahmenbedingungen umfasst die vorliegende Arbeit die folgenden Beiträge für ein Gesamtkonzept zur Unterstützung einer effizienten und skalierbaren Ausführung der vorgestellten Anwendungsklassen in dienstorientierten Umgebungen:

- Das Hauptziel dieser Arbeit besteht darin, eine konsolidierte Ausführungsplattform für SOA-Umgebungen zu präsentieren, welche die drei Anwendungsklassen *BPM*, *DIA* und *BAM* und ihre spezifischen Anforderungen unterstützt und somit die Komplexität der bisherigen orthogonalen Systeme sowie ihrer Interaktion miteinander reduziert und eine integrierte Methodik für die Entwicklung und Ausführung solcher erweiterter Prozesse bietet.
- Auf Ebene der Dienstkomponenten wird dazu ein strombasierter Ansatz erarbeitet, welcher die skalierbare und korrelierte Verarbeitung beliebiger Datenmengen ermöglicht. Dies umfasst auch die strombasierte Kommunikation zwischen Dienstanbieter und Dienstnutzer auf Grundlage von Nachrichtenkommunikation.
- Dieser Ansatz wird weiterhin um die Möglichkeit der Initialisierung und Rekonfiguration bereits laufender Dienstinstanzen erweitert, sodass eine allgemeine Nutzung von Dienstinstanzen als beliebige entfernte Operatoren möglich wird.

- Auf Ebene der SOA-basierten Dienstorchestrierung wird ein skalierbares Verarbeitungsmodell vorgestellt, welches durch die Partitionierung der mithilfe des Prozesses zu verarbeitenden Daten und eine strombasierte Verarbeitungssemantik die Anforderungen der hier diskutierten Anwendungen erfüllt.
- Die Integration der erarbeiteten Konzepte auf Dienst- und Orchestrierungsebene erlaubt es, Dienste beliebiger Funktion als skalierbare, entfernte Aktivitäten in einem Prozessgraphen zu verwenden. Diese Aktivitäten werden im Kontext der Anwendungsklassen klassifiziert und deren Umsetzung diskutiert. Weiterhin wird die Abbildung der verschiedenen Anwendungsklassen auf das Verarbeitungsmodell erläutert.
- Durch den verteilten Charakter einer dienstbasierten Infrastruktur und den erheblichen Mehraufwand XML-basierter Nachrichtenkommunikation gegenüber einer proprietären, binären Kommunikation erscheint eine effiziente bzw. intelligente Platzierung der zueinander abhängigen Dienste und Prozesse innerhalb der Infrastruktur sinnvoll. Deshalb wird auf Grundlage der entwickelten Ansätze ein Advisor vorgestellt, der eine intelligente Abbildung von Komponenten auf Serverknoten berechnet und vorschlägt.
- Zur Problemdefinition werden die bereits genannten Anwendungsklassen detailliert erläutert, deren Anwendungs- und Systemeigenschaften diskutiert und Beispielszenarien zur späteren Bewertung erarbeitet.
- Es werden das prototypische Rahmenwerk SOA-XS vorgestellt und die Bewertung der hier beschriebenen Ansätze sowohl anwendungsinvariant als auch anwendungsspezifisch vorgenommen.

Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in acht Kapitel. Ihr Aufbau ist in Abbildung 1.1 dargestellt. Im folgenden Kapitel 2 werden die drei bereits genannten technischen Anwendungsklassen diskutiert und neben den grundlegenden Systemarchitekturen auch deren Datencharakteristiken gegenübergestellt. Auf Basis dieser Gegenüberstellung werden Anforderungen definiert, welche im Rahmen einer skalierbaren SOA-basierten Anwendungsverarbeitung beachtet werden müssen. In Kapitel 3 werden

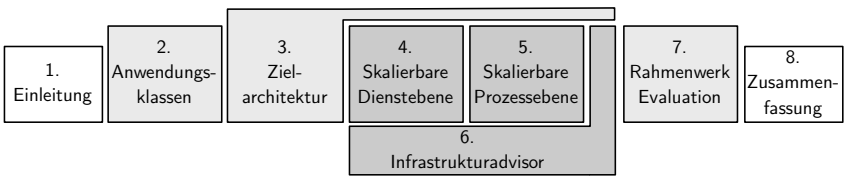


Abbildung 1.1: Struktur dieser Arbeit.

die definierten Anforderungen im Kontext dienstbasierter Prozesse bewertet, die für diese Arbeit relevante Problemstellung abgeleitet und eine zunächst abstrakte Zielarchitektur definiert. Kapitel 4 bis 6 bilden den fachlichen Kern dieser Arbeit. Dabei beschäftigt sich Kapitel 4 mit der Umsetzung einer skalierbaren Datenverarbeitung auf Dienstebene und schließt sowohl die Verarbeitung als auch die Kommunikation dieser Dienstkomponenten ein. Darauf aufbauend adressiert Kapitel 5 die skalierbare Anwendungsverarbeitung auf Prozessebene und stellt das grundlegende Verarbeitungsmodell sowie weitergehende Optimierungen vor. Im anschließenden Kapitel 6 wird auf Grundlage der beiden vorherigen Kapitel die effiziente Verteilung der entwickelten Dienst- und Prozesskomponenten in einer Infrastruktur diskutiert und ein Infrastruktur-Design-Advisor präsentiert, welcher auf Grundlage von Arbeitslast- und Ausführungsstatistiken eine kommunikationseffiziente Abbildung von Komponenten auf vorhandene Serverknoten berechnet. Kapitel 7 befasst sich daraufhin mit dem für diese Dissertation entwickelten prototypischen Rahmenwerk und beschreibt dessen Komponenten. Abschließend fasst Kapitel 8 die Ergebnisse der Arbeit zusammen und diskutiert weiterhin offene Forschungsfragen.

2 Dienstorientierte Architekturen und dienstbasierte Anwendungsklassen

Diese Arbeit beschäftigt sich in ihrem Kern mit der effizienten Ausführung dienstbasierter Prozessanwendungen im Unternehmensumfeld. Die Notwendigkeit dieser dienstbasierten Anwendungen resultiert aus der Tatsache, dass heutige unternehmensinterne IT-Infrastrukturen zunehmend nach dem Konzept der dienstorientierten Architekturen strukturiert sind. Die Grundannahme einer SOA besteht darin, dass jegliche Funktionalitäten als verteilte Dienstkomponenten angeboten werden. Dadurch wird eine flexible Komposition dieser Komponenten als dienstbasierte Prozesse (engl. *processes*) möglich. Lag der zentrale Fokus von SOA-Projekten in der Vergangenheit auf der Ausrichtung und Ausführung von dienstbasierten Prozessen an den wirtschaftlichen Abläufen (*dienstbasierte Geschäftsprozesse* (BPM)), so bilden heute *Ereignismonitoring* (BAM), d.h. die Überwachung dieser Geschäftsprozesse und deren relevanter Ereignisse, sowie zeitnahe *Datenintegration und Datenanalysen* (DIA) der Geschäftsdaten die zwei ergänzenden Anwendungsklassen im SOA-basierten Unternehmensumfeld [121, 195]. Die beiden Anwendungsklassen Datenintegration/Datenanalyse und Ereignismonitoring ergänzen die *dienstbasierten Geschäftsprozesse* auf verschiedenen Abstraktionsebenen. So werden DIA-Prozesse meist auf der technischen Ebene der Dienstimplementierung verwendet, um Daten aus heterogenen, proprietären Datenquellen aufzubereiten und gegebenenfalls zu analysieren. Die daraus resultierenden Ergebnisse werden daraufhin standardisiert über Dienstschnittstellen bereitgestellt, sodass BPM-Prozesse damit arbeiten können. Im Gegensatz dazu abstrahieren BAM-Prozesse von den konkreten BPM-Prozessen und überwachen deren Laufzeit sowie vorher definierte, geschäftsrelevante Performance-Kennzahlen (engl. *key performance indicators* (KPI)). Diese Kennzahlen werden in den BAM-Systemen ausgewertet und den BPM-Prozessen zur automatischen Entscheidungsfindung und Auswertung wiederum über Dienstschnittstellen zur Verfügung gestellt.

Die folgende Auflistung gibt zunächst einen kurzen Überblick zu den drei in dieser Arbeit adressierten Anwendungsklassen:

Dienstbasierte Geschäftsprozesse (BPM) Diese Anwendungsklasse beschreibt die technischen Geschäftsprozesse der Systemwelt, welche die wirtschaftlichen Prozesse der realen Welt abbilden und mithilfe SOA-basierter Technologien ausgeführt werden. BPM-Prozesse sind die zentralen Anwendungen in Unternehmen.

Ihre Ausführung erfolgt dabei kontrollflussorientiert, ihre Definition durch die standardisierte, grafische Modellierung des Kontrollflusses [60, 91, 100].

Datenintegration und Datenanalyse (DIA) Diese Klasse beschreibt die traditionellen Batchprozesse zur Datenintegration sowie die darauf aufbauende Analyse der integrierten Daten. Die Ausführung von DIA-Prozessen erfolgt datenflussorientiert, ihre Definition über proprietäre, grafische Modellierungswerkzeuge [18, 21, 98, 150].

Ereignismonitoring (BAM) Diese Anwendungsklasse umfasst die Überwachung von dienstbasierten Geschäftsprozessen und deren Ausführung auf Basis von Nachrichtenströmen, welche innerhalb der Unternehmensinfrastruktur versandt werden. Die Ausführung von BAM-Prozessen erfolgt datenflussorientiert, ihre Definition zumeist deklarativ über Anfragesprachen [33, 82, 89].

Abbildung 2.1 zeigt einen Überblick der *Anwendungsklassen*, deren typische *Ausführungsumgebungen* sowie ihre Interaktionen miteinander. Dienstbasierte Geschäftsprozesse bilden die zentrale Anwendungsklasse. Sie sind an der Umsetzung der realen Geschäftsprozesse und damit auch an der eigentlichen Wertschöpfung des Unternehmens beteiligt. DIA- und BAM-Anwendungen ergänzen diese zentrale Anwendungsklasse auf unterschiedlichen Abstraktionsebenen wie zuvor erläutert. Aus Sicht der *dienstbasierten Geschäftsprozesse* existiert somit die in Abbildung 2.1 gezeigte Interaktion mit den Anwendungsklassen *Datenintegration/Datenanalyse* und *Ereignismonitoring*. Die geschichtliche Entwicklung von DIA- und BAM-Anwendungen zeigt, dass diese Anwendungsklassen, im Gegensatz zu den workflowbasierten BPM-Prozessen, ihren Ursprung im Datenmanagementbereich haben.

Aufgrund ihrer spezifischen Anwendungseigenschaften basieren die beide Anwendungsklassen DIA und BAM auf einer datenzentrischen Beschreibung der Prozesse und einer datenflussorientierten Ausführung. Dennoch haben beide Klassen im Detail eine zueinander unterschiedliche Ausführungssemantik. Deshalb werden die beiden, und damit alle drei hier beschriebenen Anwendungsklassen, auf unterschiedliche, zueinander orthogonale Systeme abgebildet und ausgeführt.

Dienstbasierte Prozesse werden in einer entsprechenden Prozessausführungsumgebung verarbeitet. Diese Ausführungsumgebung ist primär auf die Komposition von Diensten und die Kommunikation über Nachrichtenaustausch zwischen den Komponenten ausgelegt und agiert somit vollständig innerhalb der SOA (Abbildung 2.1). Die Ausführungsumgebungen für die Anwendungsklassen *Datenintegration und Datenanalyse* sowie *Ereignismonitoring* bilden bisher eigenständige Systeme, deren Fokus auf Effizienz und Schnelligkeit liegt und die nicht für die standardisierte Integration in die dienstorientierte Umgebung ausgelegt sind. Zur Integration der vorher definierten DIA- und BAM-Prozesse in die unternehmensinterne SOA werden diese Prozesse über Dienstschnittstellen (engl. *SOA wrapper* [38, 92]) im Sinne eines Adapters [58] gekapselt und damit den standardisierten Zugriff von BPM-Prozessen auf diese Anwendungen ermöglicht. In aktuellen Ausführungsumgebungen

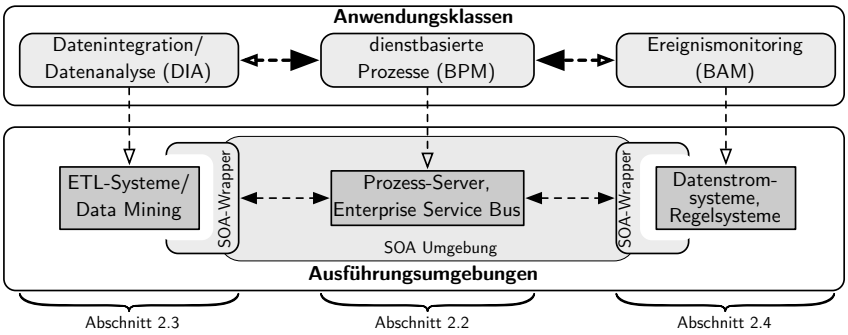


Abbildung 2.1: Die in dieser Arbeit betrachteten Anwendungsklassen.

für Datenintegrations- und Ereignismonitoring-Prozesse lassen sich zudem bereits Dienstaufrufe auf Basis von Web Services realisieren. Der Effizienzaspekt solcher Dienstaufrufe und somit der Effizienzaspekt der SOA-Interaktion wird dabei jedoch bisher nicht betrachtet.

Der verbleibende Teil des Kapitels diskutiert zunächst SOA als Basis *dienstbasierter Geschäftsprozesse* ausführlicher. Danach werden die einzelnen Anwendungsklassen detailliert beschrieben. Es werden Referenzarchitekturen der einzelnen Anwendungsdomainen vorgestellt und deren Eigenschaften und Anforderungen anhand von Beispielprozessen abgeleitet und gegenübergestellt. Abbildung 2.1 ordnet die jeweiligen Abschnitte graphisch ein.

2.1 Grundlagen der dienstorientierten Architektur

Den Grundgedanken des klassischen Paradigmas einer service-orientierten Architektur bilden, wie bereits diskutiert, die Zerlegung der Funktionalität einer Infrastruktur in spezialisierte, autonome *Dienste* und deren anschließende Komposition zu höherwertigen Prozessen. Da SOA zunächst nur ein Paradigma und keine konkrete Technik darstellt, werden im folgenden Abschnitt sowohl die Grundlagen einer SOA als auch eine konkrete Umsetzung der Teilaspekte mit dem Defacto-Standard der XML-basierten Web Service-Technologien [209] beschrieben.

2.1.1 Grundlegende Aspekte und Architektur

Zur Charakterisierung der an einer SOA beteiligten Akteure, wurde bereits in [187] ein einfaches SOA-Referenzmodell definiert. Es setzt die grundlegenden Rollen von *Dienstnutzer*, *Dienstanbieter* und *Dienstverzeichnis* miteinander in Verbindung. Abbildung 2.2 zeigt diese technische Sichtweise und die Beziehungen zwischen den Rollen.

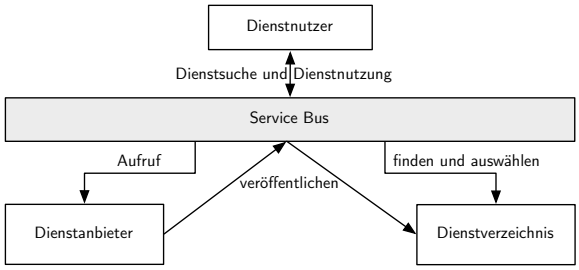


Abbildung 2.2: Einfache technische Sichtweise auf eine SOA.

Ausgangspunkt der einfachen technischen Sichtweise ist der *Dienstanbieter*. Dieser stellt neben der eigentlichen Dienstimplementierung auch deren Dienstbeschreibung zur Verfügung. Die Dienstbeschreibung (vgl. Abschnitt 2.1.2) enthält dabei alle Angaben zur Funktionalität des Dienstes, dessen Ort sowie die Art und Weise seiner Nutzung. Diese Beschreibung wird im *Dienstverzeichnis* veröffentlicht und ist dort abrufbar. Möchte der *Dienstnutzer* einen Dienst aufrufen, stellt er eine Anfrage an das Dienstverzeichnis und erhält die Dienstbeschreibung des jeweiligen Dienstes. Mit den darin enthaltenen Informationen kann der Dienst letztendlich aufgerufen werden. Die Kommunikation zwischen Dienstnutzer und Dienstanbieter basiert dabei auf dem Austausch von Nachrichten und ermöglicht somit eine lose Kopplung der Partner zueinander. Eine Erweiterung dieser dezentralen Architektur um eine zentrale Komponente bildet der Enterprise Service Bus (ESB) [114, 159]. Der ESB fungiert als Abstraktionsschicht und entkoppelt die drei Grundkomponenten voneinander. Er realisiert bzw. vermittelt die Kommunikation zwischen den Komponenten und verbirgt somit die zugrunde liegende Komplexität der Infrastruktur [114, 130, 159]. Neben der Abstraktion der physischen Orte einzelner Dienste und Nutzer übernimmt der ESB zudem Aufgaben der Datentransformation zwischen Dienstnutzer und Dienstanbieter, der Anwendungsintegration auf Protokoll- und Technologieebene sowie eines intelligenten Routings der Nachrichten in der Infrastruktur.

Aufgrund der Tatsache, dass die sehr einfache Sichtweise in Abbildung 2.2 lediglich die Kommunikation zwischen den beteiligten Partnern widerspiegelt, wird diese Sichtweise im Unternehmensumfeld um im Unternehmen relevante Komponenten ergänzt. Abschnitt 2.2 beschäftigt sich detaillierter mit der dazugehörigen Referenzarchitektur im Kontext *dienstbasierter Geschäftsprozesse*.

2.1.2 Dienstbeschreibung

Wie bereits in Abbildung 2.2 dargestellt, veröffentlicht der Dienstanbieter eine Dienstbeschreibung, damit ein potentieller Nutzer die Funktionen eines Dienstes finden und entsprechend der Beschreibung nutzen kann. Im Kontext der Web Service-Standards wurde dazu die *Web Service Description Language (WSDL)* spezifiziert [205], welche einen Dienst entsprechend des SOA-Konzeptes einheitlich und plattformunabhängig

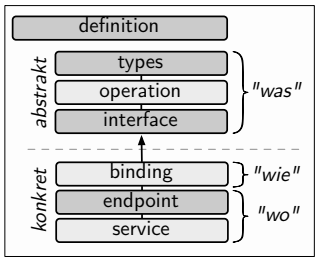


Abbildung 2.3: Schematischer Aufbau eines WSDL-Dokumentes.

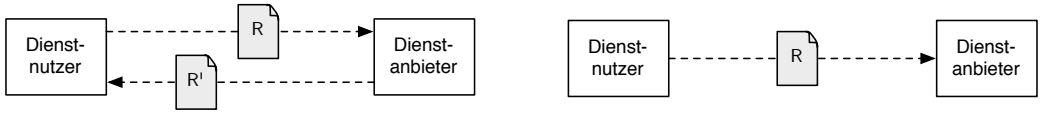
beschreibt und aktuell in Version 2.0 vorliegt. WSDL basiert auf XML und definiert blockorientierte Konstrukte und dazugehörige Attribute, um Dienstschnittstellen mit deren Operationen, Parametern und Zugriffsmethoden zu beschreiben. Abbildung 2.3 zeigt die Basiskonstrukte eines WSDL-Dokuments. Ein solches Dokument besteht grundsätzlich aus zwei Teilen. Der erste, *abstrakte* Teil beschreibt plattform- und protokollunabhängig die zur Verfügung stehenden Operationen des Dienstes mit deren Eingabe- und Ausgabe-Parametern („*was*“). Ein *operation*-Konstrukt repräsentiert dabei eine Dienstoperation und definiert ihren Namen und ihre Parameter. Das *interface*-Konstrukt gruppiert wiederum mehrere *operation*-Konstrukte und deren Fehlernachrichten zu einer logischen Einheit. Die Beschreibung der Eingabe- und Ausgabeparameter jeder Operation erfolgt mithilfe von XML Schema [203] im *types*-Abschnitt. Diese wird in den einzelnen Operationen referenziert.

Der zweite, *konkrete* Teil eines *WSDL*-Dokumentes beschreibt Protokolle und Nachrichtenformate, mit denen auf die im abstrakten Teil definierten Operationen zugegriffen werden kann („*wie*“). Als Beispiel sei hier auf die Nutzung des Nachrichtenformates SOAP im nächsten Abschnitt verwiesen. Des Weiteren wird im konkreten Teil festgelegt, „*wo*“ sich ein Dienst innerhalb der Infrastruktur befindet. WSDL als rein technische Beschreibung einer Dienstschnittstelle hat sich in den letzten Jahren als Defacto-Standard bei SOA-Produkt-Herstellern etabliert. Es existieren jedoch auch andere Ansätze, vor allem im Bereich der semantischen Dienstbeschreibung [8, 126, 142, 201]. Weiterhin wurde im Rahmen des THESEUS-TEXO-Projektes[1] die Universal Service Description Language (USDL) entwickelt. USDL beinhaltet WSDL als technische Beschreibung und erweitert diese um umfassende, nichttechnische Beschreibungskomponenten wie beispielsweise die semantische Dienstbeschreibung, AGBs, Preismodelle oder Dienstgüteeigenschaften [194].

2.1.3 Dienstkommunikation

Die klassische Kommunikation zwischen SOA-Komponenten ist bidirektional. Dabei nehmen die teilnehmenden Partner die Rollen des *Dienstnutzers* (engl. *client*) und des *Dienstanbieters* (engl. *service provider*) an. Abbildung 2.4(a) zeigt die bidirek-

[1] `http://www.theseus-programm.de/de/texo.php`



(a) Bidirektionaler Dienstaufruf. (b) Unidirektionaler Dienstaufruf.

Abbildung 2.4: Nachrichtenbasierte Dienstkommunikation.

tionale Kommunikation zwischen den beiden Rollen. Grundlage dieser Dienstkommunikation ist das Anfrage-Antwort-Paradigma (engl. *request–response paradigm*). Dabei sendet ein Dienstnutzer eine Anfrage R (engl. *request*) in Form einer Nachricht an den Dienst, der nur dadurch seine implementierte Funktion ausführt, und die Anfrage wird beim Dienst verarbeitet. Je nach Funktion des Dienstes sendet dieser eine Antwort R' (engl. *response*) an den Dienstnutzer zurück. Bei einer solchen bidirektionalen Kommunikation werden synchrone Aufrufe, bei denen der Dienstnutzer direkt auf die Antwortnachricht wartet, von asynchronen Aufrufen, bei denen der Dienstnutzer weiterarbeitet und nebenläufig auf die Antwortnachricht des Dienstes wartet, unterschieden.

Neben bidirektionalen Aufrufen kann ein Dienstnutzer eine Nachricht an einen Dienstanbieter senden, ohne eine Antwortnachricht zu erwarten (Abbildung 2.4(b)). Die Semantik basiert auf der Event-Driven Architecture (EDA) [121] bzw. einer Ausprägung dieser Architektur in Form der Message-Queuing-Systeme (MQ) [183, 118], bei der unidirektionale Verbindungen vorherrschen, die das Triggern bzw. Anzeigen eines Ereignisses zum Ziel haben [116]. Auch wenn diese Semantik im klassischen SOA-Paradigma nicht vorgesehen ist, scheint eine solche Kommunikationsmethodik für moderne service-orientierte Architekturen unabdingbar [37, 121].

Das Nachrichtenformat für den Nachrichtenaustausch ist in einer SOA nicht festgelegt. Es muss sich jedoch um ein plattformunabhängiges, standardisiertes Format und entsprechende Zugriffsmethoden handeln. In den letzten Jahren haben sich in der Literatur und in den Produkten drei Ausprägungen in Bezug auf Dienstkommunikation etabliert: das Simple Object Access Protocol (SOAP), der Representational State Transfer (REST) und die JavaScript Object Notation (JSON) [170, 114]. Diese werden im Folgenden kurz charakterisiert.

SOAP Für die Nachrichtenkommunikation wurde vom World Wide Web Consortium (W3C) im Rahmen der Web Service Activity [213] das Nachrichtenformat SOAP standardisiert und liegt aktuell in Version 1.2 vor [204]. Es basiert auf XML und abstrahiert somit von einer konkreten Implementierung des Dienstes. Außerdem ist das zu verwendende Transportprotokoll nicht näher definiert. In den letzten Jahren hat sich jedoch das Hyper Text Transfer Protocol (HTTP) als Protokoll der Wahl etabliert, da es heutzutage in jeder IT-Infrastruktur zu finden ist und auch das grundlegende Protokoll für Webanfragen darstellt. Abbildung 2.5 zeigt den Aufbau einer SOAP-Nachricht. Diese enthält drei Hauptelemente: Einen Kopfbe-

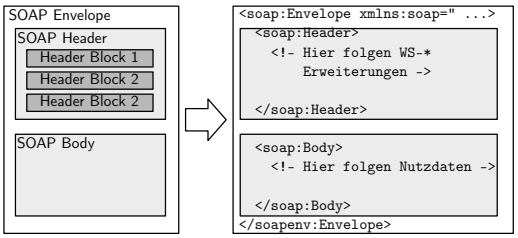


Abbildung 2.5: SOAP Nachrichtenaufbau.

reich (*header*), den Bereich für Nutzdaten (*body*) sowie den Umschlag (*envelope*), der Kopf- und Nutzdatenbereich umschließt. Dabei ist der Kopfbereich optional und beinhaltet vor allem Metadaten für den Nachrichtentransport, wie beispielsweise Empfänger, Zwischenstationen, Zugriffsschutz oder Transaktionsrealisierung. Die Endpunkte bzw. die Zwischenstationen, über welche die Nachricht transportiert wird, müssen diese Metadaten verstehen, unterstützen und umsetzen. Weiterhin können im Kopfbereich anwendungsspezifische Kontextinformationen zu den Nutzdaten enthalten sein, welche beispielsweise die Zuordnung der Nutzdaten zu bestimmten logischen Ressourcen eines Dienstes beim Endpunkt ermöglichen. Die Inhalte des Nutzdatenbereichs werden in XML als Kinder des *body*-Elementes in die Nachricht integriert. Dadurch kann XML Schema [203] mit seinen Datentypen direkt bei der Beschreibung und Validierung der Nutzdaten verwendet werden. Die Spezifikation unterscheidet zwei Formen der Nutzdatenstruktur: Bei der Darstellung der Nutzdaten als *Remote Procedure Call* (RPC) wird eine fest definierte Struktur für den Aufruf klassischer entfernter Methoden angenommen. Dabei werden Funktionsname und die Funktionsparameter in fester Struktur übergeben. Bei der Darstellung der Nutzdaten in dokumentenbasierter Form (*document/literal style*) ist die Nutzdatenstruktur anwendungsabhängig und die Validierung bzw. die korrekte Verarbeitung der Nutzdaten muss vom Empfänger sichergestellt werden.

Zusammenfassend lässt sich festhalten, dass SOAP durch seine Unabhängigkeit vom Transportmedium, die Unterstützung von komplexen Kommunikationsmustern sowie seine Erweiterungsmöglichkeiten eine Vielzahl von verteilten Anwendungen unterstützt. Dadurch ist jedoch eine einfache Nutzung und Implementierung nicht möglich. Auch der Mehraufwand zum Erzeugen einer SOAP-Nachricht bei einfachen Punktanfragen bietet häufig Anlass zur Kritik [51, 152, 108].

REST REST [56] wurde bereits im Jahre 2000 von Roy Fielding in seiner Dissertation beschrieben. Im Gegensatz zu SOAP, welches ein echtes Nachrichtenformat darstellt, ist REST ein Architekturstil zur Abfrage und Verwaltung von Ressourcen. Eine Ressource ist nach der Definition des W3C eine beliebige, logische oder physische Entität, auf deren technische Repräsentation über einen Uniform Resource Locator (URL) zugegriffen werden kann. Die Beschreibung der Ressource zur Abfrage wird über Schlüssel-Wert-Paare direkt in der URL angegeben und über HTTP

entsprechend versandt. Dadurch können auch standardisierte Werkzeuge, wie beispielsweise Web-Browser, problemlos mit diesen Ressourcen interagieren. Zum Abfragen, Einfügen, Aktualisieren und Löschen von Ressourcen werden die HTTP-Methoden *GET*, *PUT*, *POST* und *DELETE* unterstützt. In welchem Format die Anfrage das Resultat an den Aufrufer zurücksendet, ist in REST nicht vorgegeben. Das Format wird über den MIME-Typ im HTTP-Kopfteil angezeigt und bedarf einer Interpretation des Clients. Alle REST-basierten Dienste sind zustandslos. Ein jeweiliger Zustand wird per Definition vom Client verwaltet und alle Informationen zum Verarbeiten einer Anfrage beim Dienst müssen bei dessen Aufruf angegeben werden.

Schließlich bleibt festzuhalten, dass REST die Kommunikation mit Webressourcen vereinfacht und aufgrund der Interaktion mit diesen Ressourcen durch standardisierte URL-Mechanismen stark an Verbreitung gewonnen hat. Nachteile dieser HTTP-gebundenen Form der Nachrichtenkommunikation sind die punktbasierte Kommunikation zwischen Dienstnutzer und Dienstanbieter sowie die fehlende Möglichkeit für die Definition erweiterter Metadaten wie beispielsweise Sicherheitsrichtlinien, komplexe Kommunikationsmuster mit Zwischenstationen oder transaktionale Sicherheit.

JSON JSON [175] stellt nicht wie SOAP ein Nachrichtenformat dar, sondern ein zu XML alternatives textuelles Datenformat für den Austausch strukturierter Daten. Die Grundlage von JSON bilden dabei Listen von Schlüssel-Wert-Paaren, welche auch hierarchisch ineinander geschachtelt sein können. Jedes gültige JSON-Dokument ist gleichzeitig eine gültige Javascript-Datenstruktur und kann direkt zur Laufzeit in ein Javascript-Objekt umgewandelt werden. JSON hat seinen Ursprung in AJAX-basierten Webseiten und wird dabei hauptsächlich in Verbindung mit REST-Anfragen verwendet. Im Vergleich zu XML ist JSON ein reines Datenaustauschformat mit einfacherer Syntax. So fehlen beispielsweise die Unterstützung von Namensräumen [208], die Möglichkeit, Elemente eines Dokumentes an beliebigen Stellen im Dokument oder außerhalb des Dokumentes zu referenzieren [210] bzw. Dokumente mit Anfragesprachen zu durchsuchen [211, 212]. Somit lassen sich mit JSON zwar nicht alle Dokumentstrukturen von XML abbilden, jedoch reduziert sich der Mehraufwand für die Dokumentinterpretation erheblich.

2.1.4 Dienstkomposition und Prozessbeschreibung

Die lose Kopplung von Diensten über Nachrichtenaustausch sowie die standardisierte Dienstbeschreibung ermöglichen es, Dienste und deren Daten einfach und flexibel miteinander zu kombinieren. Grundlage für deren technische Spezifikation sind Arbeitsabläufe (engl. *workflows*) bzw. Prozesse. In einem Prozess werden Aktivitätsketten definiert, deren Gesamtheit das Erreichen eines bestimmten, höherwertigen Ergebnisses zum Ziel hat. Aktivitäten repräsentieren dabei technische Dienste

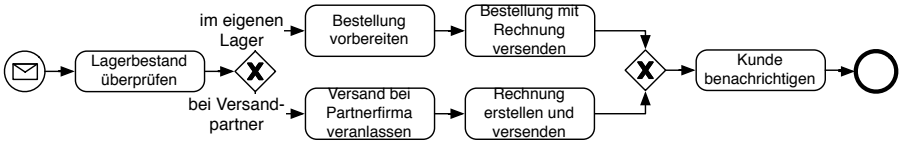


Abbildung 2.6: Beispielprozess „Bearbeitung eingehender Bestellungen“.

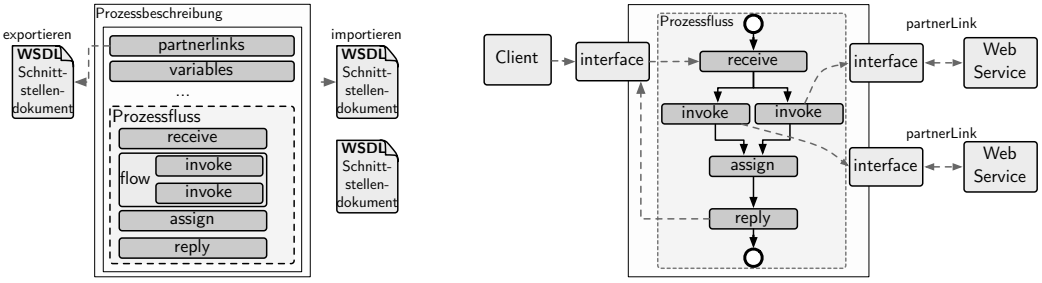
und deren Funktionalität. Kanten zwischen den Aktivitäten stellen die temporalen Beziehungen der Aktivitäten zueinander, d.h. deren Abarbeitungsreihenfolge, dar. Diese Art der Komponentenbereitstellung und deren Orchestrierung zu höherwertigen Ergebnissen wird auch als Paradigma der zweistufigen Programmierung bezeichnet [99, 165]. „Programming in the large“ beschreibt dabei die Kommunikation zwischen beteiligten Komponenten und entspricht im Falle einer SOA der Komposition vorhandener Dienste zu höherwertigen Prozessen. „Programming in the small“ hingegen kennzeichnet die Entwicklung der eigentlichen Komponenten mithilfe klassischer Programmiersprachen.

Abbildung 2.6 zeigt einen einfachen Beispielprozess für eine Bearbeitung eingehender Bestellungen in einem Versandgroßhandel. Eine eintreffende Bestellung startet den Prozess, welcher zunächst veranlasst, die Verfügbarkeit des bestellten Artikels im eigenen Lager zu überprüfen. Befindet sich der Artikel im eigenen Lager, wird er zum Versand vorbereitet und die dazugehörige Rechnung erstellt. In einem zweiten Schritt wird der Versand der Bestellung mit beiliegender Rechnung angestoßen. Befindet sich der bestellte Artikel hingegen nicht im eigenen Lager, veranlasst der Prozess den Versand des Artikels bei einer Partnerfirma, die den gewünschten Artikel vorrätig hat. Danach wird die Rechnung für den Bestellvorgang generiert und separat an den Kunden versandt. In beiden Fällen wird der Kunde zum Abschluss des Bestellprozesses über den erfolgten Versand informiert. Im Kontext einer SOA entsprechen die im Prozess dargestellten Aktivitäten jeweils verteilten Dienstaufrufen, welche die jeweiligen Funktionalitäten bereitstellen bzw. veranlassen.

Im Bereich der Web Service-Technologien hat sich zur Spezifikation von Dienstkompositionen bzw. automatisierten Workflows die Business Process Execution Language (BPEL) [188] als Defacto-Standard etabliert. Im Folgenden werden die Kerneigenschaften von BPEL dargestellt, da diese Sprache im anschließenden Kapitel 3 im Kontext der skalierbaren Anwendungsverarbeitung näher betrachtet wird.

Business Process Execution Language (BPEL)

BPEL ist eine XML-basierte Prozessbeschreibungssprache, welche auf dem Dienstmodell der WSDL-Spezifikation aufbaut und deren Aktivitäten durch Web Services und deren Aufrufe repräsentiert werden. Ein BPEL-Prozess kann seinerseits selbst eine Web Service-Schnittstelle als WSDL-Dokument anbieten, wodurch dieser wiederum einen Dienst repräsentiert und in anderen Prozessen als Web Service-Aktivität



(a) Struktur eines BPEL-Dokumentes.

(b) Visuelle Darstellung des Prozessflusses und dessen Kommunikation.

Abbildung 2.7: Die Prozessbeschreibungssprache BPEL.

integriert werden kann. Somit ist es möglich, eine hierarchische Struktur von atomaren und kompositen Diensten aufzubauen. Die Grundstruktur eines BPEL-Prozesses ist in Abbildung 2.7(a) skizziert und orientiert sich an einer Blockstruktur. Diese Blöcke beinhalten unterschiedliche Informationen zu den wichtigsten Aspekten des aktuellen Prozessplans. Die drei Hauptblöcke, welche im Folgenden näher beschrieben werden, sind dabei Partnerlinks (*partnerLinks*), Variablen (*variables*) sowie der eigentliche Prozessfluss. Neben diesen drei Hauptblöcken existieren noch eine Reihe weiterer Funktionsblöcke, wie beispielsweise *scopes* für die Kompensation im Fehlerfall oder *correlationSets* für die Korrelation von Nachrichten zu Prozessinstanzen. Diese Funktionalitäten sind jedoch nicht Gegenstand der weiteren Arbeit und werden deshalb nicht näher betrachtet. Der interessierte Leser sei deshalb auf die Spezifikation unter [188] verwiesen.

Der Block der Partnerlinks (*partnerLinks*) beschreibt alle Kommunikationsendpunkte der beteiligten Web Services sowie die eigene Web Service-Schnittstelle. Diese Endpunkte werden abstrakt auf Ebene der *interface*-Abschnitte der Partner-WSDL-Beschreibungen definiert (vgl. Abschitt 2.1.2, Dienstbeschreibung). Im Block der Prozessvariablen (*variables*) werden Platzhalter für zustandsbehaftete Informationen zur Prozesslaufzeit definiert. Diese Variablen werden durch XML Schema-Elemente beschrieben und von den einzelnen Aktivitäten im Prozessfluss referenziert. Sie dienen somit als Eingabe- bzw. Ausgabeparameter der Aktivitäten und beschreiben somit auch die Nachrichtentypen, welche zwischen Prozess und Web Service ausgetauscht werden. Der eigentliche Prozessfluss wird im dritten Hauptblock beschrieben. Beispielhaft sind in Abbildung 2.7(a) sequenziell die Aktivitäten *receive*, *flow*, *assign* und *reply* definiert, wobei die *flow*-Aktivität zwei nebenläufige *invoke*-Aktivitäten enthält, welche jeweils einen Web Services aufrufen. Diese Web Services wurden über deren WSDL-Dokumente in den Block der Partnerlinks importiert. Abbildung 2.7(b) zeigt dasselbe BPEL-Dokument mit Fokus auf dem Prozessfluss und den beteiligten Kommunikationspartnern. Die Menge der in der *BPEL*-Spezifikation definierten Aktivitäten werden in die *zwei* Klassen *Basisaktivitäten* und *strukturierte Aktivitäten* unterteilt:

Basisaktivitäten Die Klasse der Basisaktivitäten beschreibt die elementaren Aktivitäten. Sie beinhalten selbst keine anderen Aktivitäten und bilden somit die grundlegenden Ausführungseinheiten. Die für diese Arbeit wichtigen Basisaktivitäten werden nachfolgend näher beschrieben:

receive Diese Aktivität blockiert den Prozess, bis eine vordefinierte Nachricht eintrifft. Existiert nur eine *receive*-Aktivität im gesamten Prozessfluss, startet und initialisiert diese Aktivität eine neue Prozessinstanz. Die Struktur der Nachricht wird in der Dienstschnittstellenbeschreibung des Prozesses definiert.

reply Das Gegenstück zur *receive*-Aktivität bildet *reply*. Die *reply*-Aktivität fungiert als Rückgabewert des Prozesses und beschreibt damit seinen synchronen Aufruf. Die Struktur des Rückgabewertes wird in der WSDL-Beschreibung des Prozesses definiert. Fehlt diese Aktivität im Prozessfluss, wird ein asynchroner Prozessaufruf angeboten.

invoke Die Aktivität *invoke* ist die Kernaktivität der Prozessbeschreibungssprache BPEL. Damit ist es möglich, die vielfältigen, durch Dienste bereitgestellten Funktionen in den Prozessfluss zu integrieren und die eigentliche Dienstorchestrierung zu realisieren. Es werden synchrone von asynchronen Dienstaufrufen unterschieden, die entweder den Prozessfluss bis zu einer Antwort blockieren oder keine Antwort erwarten und den Prozessfluss deshalb sofort fortsetzen.

assign Diese Aktivität ermöglicht die Transformation bzw. Manipulation von Prozessvariablen und deren Daten. Ausprägungen dieser Manipulationen sind einfache Wertzuweisungen, *XPath*-Ausdrücke sowie *XSLT*-Transformationsbeschreibungen [206].

Weitere beispielhafte Aktivitäten, die der Klasse der Basisaktivitäten zugeordnet werden, sind **throw** für das Aufzeigen eines Fehlerfalles, **exit** zum expliziten Beenden einer Prozessinstanz, **wait** für das Blockieren des Prozesses für eine definierte Dauer sowie **empty** für eine leere Anweisung oder als Platzhalter in der Modellierungsphase.

Strukturierte Aktivitäten Diese Klasse beinhaltet Aktivitäten zur Strukturierung, Kapselung und rekursiven Komposition von Aktivitäten. Die für diese Arbeit wichtigen Aktivitäten werden nachfolgend näher beschrieben:

sequence Die in einer *sequence*-Aktivität enthaltenen Aktivitäten werden sequenziell, d.h. zeitlich strikt nacheinander abgearbeitet.

if Diese Aktivität ermöglicht die Ausführung von exakt einer Folgeaktivität aus einer vordefinierten Menge an Aktivitäten aufgrund einer Bedingung. Zusätzlich können beliebige *elseif*- bzw. *else*-Elemente definiert werden, welche in der Reihenfolge ihrer Definition überprüft und ausgewertet werden.

while Dieses Konstrukt beschreibt eine kopfgesteuerte Schleife und ermöglicht die iterative Abarbeitung einer Sequenz von Aktivitäten. Hierbei werden die einzelnen Iterationen sequenziell abgearbeitet.

flow Die in dieser Aktivität definierten Aktivitäten werden parallel ausgeführt. Zusätzlich können zwischen Untermengen der darin definierten Aktivitäten zeitliche Abhängigkeiten über sog. *links* definiert werden.

Als weitere Aktivitäten der Klasse der *strukturierten Aktivitäten* gelten beispielsweise **repeatUntil** und **forEach** als zusätzliche Ausprägungen von Schleifenkonstrukten sowie **pick** zur Integration von ereignisgesteuerter Logik in den Prozessfluss.

2.2 Dienstbasierte Geschäftsprozesse

Die erste der hier diskutierten Anwendungsklassen bilden *dienstbasierte Geschäftsprozesse* (engl. *business processes (BPM)* oder verallgemeinert *workflows*). Sie bilden die technische Realisierung von realen Geschäftsprozessen eines Unternehmens mit deren Wertschöpfungskette durch automatisierte Arbeitsabläufe und durch die Integration der unterschiedlichen Anwendungssysteme auf Basis des Nachrichtenaustauschs. Deshalb bilden sie die zentrale Anwendungsklasse einer modernen Unternehmensinfrastruktur. Grundlage dieser *dienstbasierten Geschäftsprozesse* bildet eine serviceorientierte Architektur (SOA), wie sie im vorherigen Abschnitt beschrieben wurde.

2.2.1 Beispielprozess

Dieser Abschnitt beschreibt einen Beispielprozess der Klasse BPM, welcher zum einen die grundlegenden Eigenschaften dieser Anwendungsklasse widerspiegelt und zum anderen als Ausgangspunkt für die Interaktion mit den anderen Anwendungsklassen in den Abschnitten 2.3.1 und 2.4.1 dient (Abbildung 2.8). Alle Abbildungen der Beispielprozesse in den einzelnen Unterabschnitten dieses Kapitels sind gleich strukturiert: Es werden jegliche Partnerkomponenten der in diesem Kapitel beschriebenen Beispielprozesse in den unterschiedlichen Anwendungsklassen dargestellt. Komponenten mit grauem Rahmen und grauer Schrift dienen dabei nur der Visualisierung der klassenübergreifenden Interaktion und werden nicht tiefgehender erläutert. Die Komponenten mit schwarzem Rahmen und schwarzer Schrift repräsentieren jeweils einen konkreten Beispielprozess einer der drei Anwendungsklassen, welche im jeweiligen Unterabschnitt detaillierter beschrieben werden.

Ausgangspunkt des klassenübergreifenden Beispielszenarios sei ein Betreiber eines Online-Webshops, welcher vierteljährlich den BPM-Prozess „Kundenpflege“ in Abbildung 2.8 ausführt. Jede Aktivität entspricht dabei einem Unterprozess. Von diesen Unterprozessen wird jedoch in dieser Abbildung zunächst abstrahiert. Dabei startet eine Eingangsnachricht den Prozess, die den von den Kunden erbrachten Mindes-

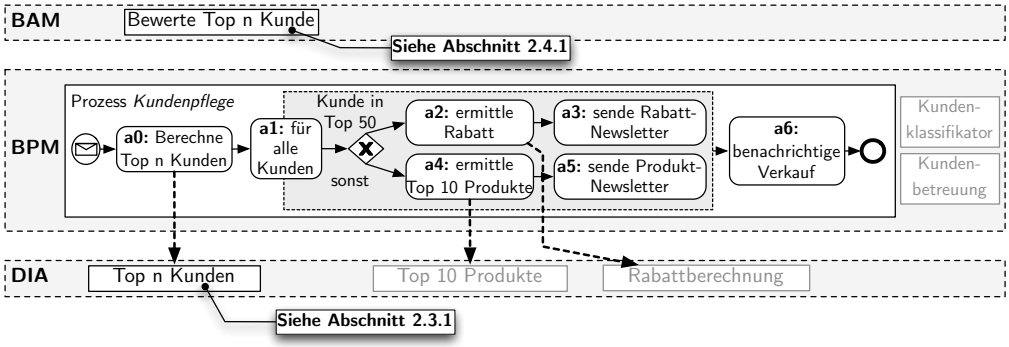


Abbildung 2.8: Geschäftsprozess „*Kundenpflege*“.

tumsatz sowie zur Begrenzung des Ergebnisses die maximale Anzahl n der zu analysierenden Kunden als Parameter an den Prozess übergibt. In der ersten Aktivität *a0* werden die Top n Kunden der letzten drei Monate berechnet und die entsprechende Kundenliste mit den Platzierungen der einzelnen Kunden zurückgegeben. Danach werden für jeden der n Kunden über ein Schleife die nachfolgenden Aktivitäten ausgeführt (*a1*). Gehört der Kunde zu den Top n (beispielsweise $n = 50$) Kunden des Unternehmens, wird der spezifische Rabatt des Kunden in Abhängigkeit seines Gesamtumsatzes sowie seines Kreditlimits berechnet (*a2*). Dieser wird in einem nächsten Schritt an einen Unterprozess übergeben (*a3*), welcher einen Newsletter mit einem spezifischen Rabattgutschein erstellt und diesen an den Kunden versendet. Gehört der Kunde nicht zu den 50 umsatzstärksten Kunden des Unternehmens, so werden die zehn aktuell meistverkauften Produkte mit ihren aktuellen Preisen und Produktbeschreibungen ermittelt (*a4*), diese in einen Werbenewsletter integriert und an den Kunden versandt (*a5*). Wurden alle Kunden verarbeitet, wird die Verkaufsabteilung des Unternehmens per E-mail über die Abarbeitung des Prozesses informiert.

Abbildung 2.9 zeigt den beschriebenen abstrakten Prozess als technische Realisierung in Anlehnung an BPEL. Die jeweiligen Aktivitätstypen sind in eckigen Klammern annotiert. *invoke*-Aktivitäten kommunizieren mit den dargestellten Dienst-

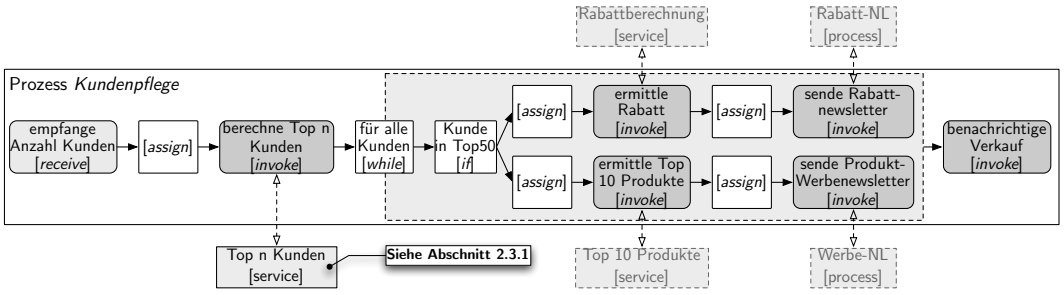


Abbildung 2.9: BPEL-Repräsentation des Prozesses „Kundenpflege“.

komponenten zur Ausführung der jeweiligen Funktion. Die Aktivität *berechne Top n Kunden* wird dabei durch den DIA-Prozess *Top n Kunden* aus Abschnitt 2.3.1 realisiert, der dem Prozess über eine Dienstschnittstelle somit standardisiert zur Verfügung steht.

2.2.2 SOA-Referenzarchitektur

Betrachtet man eine SOA im Unternehmensumfeld, so muss die einfache Sichtweise in Abbildung 2.2 (Seite 10), welche nur die Kommunikation zwischen den beteiligten Partnern widerspiegelt, um unternehmensrelevante Komponenten ergänzt werden. Abbildung 2.10 zeigt eine entsprechend erweiterte Referenzarchitektur. Diese Abbildung konzentriert sich auf die technischen Aspekte der erweiterten Architektur und stellt dabei ausschließlich die Phasen der Anwendungsentwicklung sowie der Anwendungsausführung dar.

Entwicklungsphase

Die Entwicklungsphase dient der Konzeption und Implementierung von Diensten und Prozessen. Die Prozessentwicklung kann dabei sowohl einstufig als auch zweistufig umgesetzt werden. Die zweistufige Entwicklung beinhaltet zunächst die abstrakte Modellierung des technischen Prozesses, bei der die technischen Details wie beispielsweise konkrete Datenstrukturen, deren Abbildung auf Nachrichten oder die Endpunkte spezifischer Dienstkomponenten ausgeblendet werden. Die grafische Notation des Beispielprozesses *Kundenpflege* aus Abbildung 2.8 (Seite 19) stellt eine solche abstrakte Modellierung dar. Diese Art der Prozessmodellierung basiert auf der grafischen Modellierung von Aktivitäten und des dazugehörigen Kontrollflusses. In den letzen Jahren hat sich dafür die Business Process Modeling Notation (BPMN) [189] etabliert, aber auch andere Notationen, wie beispielsweise die UML-Aktivitätsdiagramme [88], werden verwendet. Diese abstrakte Modellierung wird in einem zweiten Schritt durch eine Transformationsvorschrift in eine technische Prozessbeschreibung umgeformt und in einer weiteren manuellen Anpassung um konkrete technische Informationen angereichert (siehe auch Beispielprozess *Kundenpflege* in Abbildung 2.9 (Seite 19)). Zu diesen Informationen gehören die Bestimmung konkreter Dienste, die Handhabung der dabei anfallenden Datenstrukturen sowie die Festlegung weiterer Eigenschaften des Prozesses. Fehlt die abstrakte Modellierung, spricht man von einer einstufigen Prozessdefinition.

Ausführungsphase

Liegt die vollständige technische Prozessbeschreibung aus der vorherigen Entwicklungsphase vor, wird diese kompiliert und in der Ausführungsplattform registriert (engl. *deploy*). Zeitliche Ereignisse oder externe Anfragen starten die Ausführung

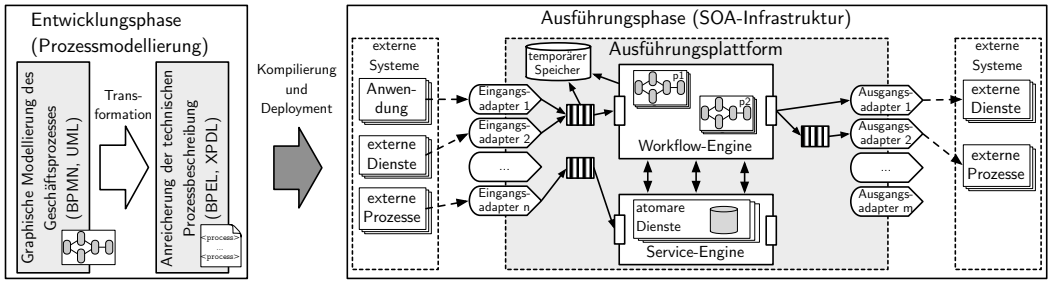


Abbildung 2.10: Technische SOA-Referenzarchitekur in Anlehnung an [26].

der installierten Prozesse. Meist beinhalteten diese Anfragen konkrete Nutzdaten (wie beispielsweise eine Kundenbestellung), welche der Prozess direkt verarbeitet. Aus technischer Sicht besteht eine typische Ausführungsplattform grundsätzlich aus Eingabeadaptern, den Laufzeitumgebungen für Prozesse und atomare Dienste sowie Ausgabeadaptern.

Eingabeadapter Eingabeadapter warten auf externe Anfragen von Anwendungen wie beispielsweise grafischen Benutzerschnittstellen oder anderen dienstbasierten Komponenten. Dabei konvertieren sie das ankommende Nachrichtenformat gegebenenfalls in eine einheitliche Datenrepräsentation, welche die Dienst- und Prozesskomponenten verarbeiten können. Die konvertierten Daten werden danach in Eingabewarteschlangen eingefügt. Diese Warteschlangen dienen als Puffer sowie zur Erhaltung der Eingangsordnung der Nachrichten und werden seriell an die Dienst- und Prozessausführungsumgebung (engl. *service* | *workflow engine*) weitergegeben. Man unterscheidet dabei transiente von persistenten Warteschlagen.

Workflow-Engine Die Workflow-Engine beinhaltet die kompilierten und abgelegten Prozessbeschreibungen aus der Entwicklungsphase und führt die Prozesse entsprechend der Beschreibung aus. Die Ausführungssemantik dieser Prozesse entspricht dem kontrollflussbasierten Charakter des modellierten Prozesses [22, 91]. Kommuniziert eine Prozessinstanz mit einem Dienst, der in derselben Ausführungsumgebung und somit in der lokalen Service-Engine residiert, so wird die Kommunikation effizient über interne Objekte realisiert. Die Kommunikation mit externen Systemen erfolgt entweder synchron direkt über die Ausgabeadapter oder asynchron über eine Warteschlange [26].

Service-Engine Die Service-Engine enthält die atomaren Dienste, d.h. Dienste, welche keine kompositen Prozessgraphen kapseln. Sie werden in traditionellen Programmiersprachen verfasst, wobei die verwendeten Sprachen von der übergeordneten Ausführungsumgebung abhängig sind. Mögliche Funktionen dieser Dienste werden somit nur durch die jeweilige Programmiersprache begrenzt. Typische Funktionen sind beispielsweise der Datenbankzugriff mit der dazugehörigen Vorverarbeitung und Transformation der Daten. Die Dienste der Service-Engine können lokal von der Workflow-Engine aufgerufen werden. Dabei wird die performance-kritische Kommu-

nikation über XML vermieden und so eine effiziente Ausführung der lokalen Dienste erreicht [184]. Die Kommunikation mit externen Systemen kann dabei analog zur Workflow-Engine synchron oder asynchron erfolgen.

Ausgabeadapter Die Ausgabeadapter realisieren die Kommunikation zwischen atomaren Diensten bzw. Prozessen und externen Systemen. Ihre Aufgabe besteht dabei neben dem eigentlichen Kommunikationsaufbau auch in der Transformation der internen Datenstruktur der Ausführungsumgebung in das spezifische Format des externen Systems.

2.2.3 Charakteristika

Dieser Abschnitt fasst die für diese Arbeit relevanten System- und Anwendungseigenschaften der Anwendungsklasse der *dienstbasierten Geschäftsprozesse* zusammen und definiert notwendige Metriken. Einen Vergleich der Anwendungsklassen untereinander nimmt Abschnitt 2.5 vor. Im Allgemeinen kann die Struktur eines dienstbasierten Prozesses P_{BPM} in dieser Arbeit wie folgt definiert werden:

Definition 1 P_{BPM}: *Ein dienstbasierter Prozess P_{BPM} ist definiert mit $P_{BPM} = (A, K, S)$ als gerichteter, azyklischer Graph (DAG). Dabei beschreibt A mit $A = (a_1, \ldots, a_i, \ldots, a_k)$ die Menge der im Prozessplan definierten Aktivitäten als Knoten des Graphen und K die Menge an Kanten, welche die Aktivitäten miteinander verbinden und ihre zeitliche Reihenfolge festlegen. Weiterhin beschreibt S die Menge an externen Diensten, welche über spezielle Aktivitätstypen aufgerufen werden und mit denen der Prozess interagiert.*

Systemeigenschaften

Das Verarbeitungsmodell für derartige Prozesse entspricht dem klassischer Workflow-Systeme [7] und beschreibt die Abarbeitung des definierten Kontrollflusses und damit die kontrollflussorientierte, schrittweise und atomare Abarbeitung der darin definierten Aktivitäten [23]. Dabei wird eine Aktivität ausgeführt und erst nach deren erfolgreicher Beendigung die nachfolgende Aktivität begonnen. Eine Eingangsnachricht m_j generiert jeweils eine neue Prozessinstanz p_j und wird darin isoliert von anderen Instanzen abgearbeitet. Prozessinstanzen des gleichen Prozesstyps werden von der Ausführungsumgebung seriell verarbeitet und unterbinden dadurch das Überholen einzelner Nachrichten und eine zeitlich inkonsistente Verarbeitung [30]. Die Datenkommunikation erfolgt standardisiert auf Basis von Nachrichten. Die Strukturierung der kommunizierten Anwendungsdaten basiert derzeit weitestgehend auf XML, wobei in speziellen Anwendungsdomainen das Datenformat JSON aufgrund seiner einfachen Struktur und der direkten Nutzung in JavaScript an Bedeutung gewinnt [170]. Um die baumartige Struktur der Daten effizient im System abbilden und verarbeiten zu können, arbeiten heutige Systeme intern nativ mit einem

Datenmodell für hierarchisch strukturierte Daten [179]. Damit lassen sich XPath- oder XQuery-Ausdrücke aus der Entwicklungsphase effizient zur Ausführungszeit anwenden.

Anwendungseigenschaften

Neben dem eigentlichen Format der Daten lässt sich die Charakteristik der Anwendungsprozesse anhand zweier Dimensionen näher beschreiben. Nachfolgend werden die Dimensionen der *Nachrichtengröße* und der *Nachrichtenrate* definiert, welche in den Abschnitten der anderen Anwendungsklassen wieder aufgegriffen werden.

Definition 2 Nachrichtengröße: *Die Nachrichtengröße G beschreibt die Summe aller Daten, welche eine Prozessinstanz p durch eine eintreffende* Eingangsnachricht *m verarbeiten muss. Dabei können Datenmengen sowohl in der eintreffenden Eingangsnachricht enthalten sein oder erst im Laufe der Prozessabarbeitung über externe Dienste dazugeladen werden. Da ein Prozess aus einer Menge A von Aktivitäten a_i besteht, berechnet sich die Nachrichtengröße G mit $G = \sum_{i=1}^{|A|} sizeof_{in,out}(a_i)$ als Summe der Größen von Ein- und Ausgabedaten der einzelnen Aktivitäten a_i. Da diese Aktivitäten auch die Kommunikation mit externen Diensten realisieren, beinhaltet G auch die Summe aller Nachrichten an diese Dienste.*

Bei der Analyse des Beispielprozesses in Abschnitt 2.2.1 bleibt festzuhalten, dass die Nachrichtengröße (auch im Hinblick auf die noch zu diskutierenden Anwendungsklassen) gering bzw. moderat ausfällt. Dies liegt einerseits in der Natur der dienstbasierten Geschäftsprozesse begründet, welche zum Großteil reale Unternehmensprozesse abbilden und somit der eigentliche Kontrollfluss und damit die korrekte zeitliche Abfolge der Aktivitäten im Vordergrund steht. Andererseits beinhaltet der Nachrichtenaustausch zwischen den Partnern meist nur Kontrollnachrichten zum Anzeigen von Zustandsänderungen bei den externen Diensten.

Definition 3 Nachrichtenrate: *Die Nachrichtenrate R gibt die Ankunftsrate der* Eingangsnachrichten *m_j innerhalb einer definierten Zeiteinheit an, welche vom System verarbeitet werden muss. R berechnet sich aus $R = \frac{n}{\Delta t}$ mit n als der Anzahl der Eingangsnachrichten m_j und Δt als dem dazugehörigen Zeitintervall.*

Da im Kontext der Anwendungsklasse der dienstbasierten Prozesse eine Eingangsnachricht m_j eine neue Prozessinstanz p_j generiert [30], impliziert die Nachrichtentrate R die Häufigkeit der seriellen Abarbeitung von Prozessinstanzen in einer Zeiteinheit. Bewertet man R in dieser Anwendungsklasse, so ist die Nachrichtenrate bzw. die Anzahl der zu startenden Prozessinstanzen in einer Zeiteinheit im Vergleich zu den noch zu diskutierenden Anwendungsklassen als gering einzuschätzen. Bei der Erweiterung der hier beschriebenen dienstbasierten Prozessausführung um Szenarien der Anwendungsintegration wie in [30], erhöht sich die Nachrichtenrate auf ein moderates Maß.

Zusammenfassend bleibt festzuhalten, dass dienstbasierte Prozesse durch ihre Nähe zu realen Geschäftsprozessen als Kernanwendungen im Unternehmensumfeld angesehen werden. Dabei ist das Konzept der verteilten Komponenten mit seiner standardisierten, nachrichtenbasierten Kommunikation auch in andere Anwendungsdomainen übertragbar. Im Hinblick auf eine skalierbare Anwendungsverarbeitung mithilfe dieses Konzeptes wird ihre kontrollflussbasierte Ausführung sowie die nachrichtenbasierte Kommunikation in Kapitel 3 näher diskutiert. Die Anwendungseigenschaften der dienstbasierten Geschäftsprozesse in den Dimensionen *Nachrichtengröße* und *Nachrichtenrate* lassen sich jeweils im unteren Bereich einordnen.

2.3 Datenintegration und Datenanalyseprozesse

Die zweite zu diskutierende Anwendungsklasse beinhaltet die Prozesse der Datenintegration und Datenanalyse (DIA). Sie umfasst dabei sowohl die Anwendungsfelder der Datenintegrations- und Datenanalyseprozesse als auch deren Kombination. Datenintegration ist ein historisch gewachsenes Anwendungsfeld, welches die Notwendigkeit von Datenextraktion, Datentransformation und Datenbereinigung aufgrund heterogener Datenquellen adressiert sowie dem Nutzer die Operationen über Werkzeugunterstützung ermöglicht [141, 153]. Datenintegration wird häufig im Bereich von Datawarehouse-Anwendungen eingesetzt, um Daten heterogener Quellsysteme in ein konsolidiertes, vereinheitlichtes Zielsystem zu überführen [18, 119]. Die Gruppierung derartiger Systeme wird häufig unter dem Akronym ETL vollzogen [18, 177, 191]. Es beschreibt die drei Hauptschritte von der *Extraktion* der Daten aus den Quellsystemen über deren *Transformation* in eine Zielstruktur bis zum *Laden* der modifizierten Daten in das entsprechende Zielsystem. Die Datenintegration bildet dabei die Voraussetzung der Datenanalyse, welche auf den konsolidierten Daten des Zielsystems arbeitet [98, 119]. Während dedizierte Datenintegrationssysteme für die effiziente und performante Datentransformation optimiert sind, liegt der Fokus von Datenanalysewerkzeugen vornehmlich auf der Vielfältigkeit der angebotenen Algorithmen zur Datenanalyse sowie auf der Darstellung der Resultate selbiger. Beiden Anwendungsfeldern ist gemein, dass die jeweiligen Datenflüsse als direkter, azyklischer Graph visuell modelliert und als Prozess bzw. Workflow im System ausgeführt werden. Somit ähnelt die Modellierung der hier diskutierten Prozesse der Modellierung von *dienstbasierten Geschäftsprozessen*, jedoch fehlt bei der Ausführung von DIA-Prozessen der stark verteilte Charakter der einzelnen Operatoren. Der Ansatz einer XML-basierten technischen Prozessbeschreibung ähnlich zu BPEL wurde in aktuellen Datenanalysewerkzeugen integriert [192].

2.3.1 Beispielprozess

Der in Abbildung 2.11 aufgezeigte Beispielprozess beschreibt die Implementierung des *Top n Kunden*-Dienstes, welcher von dem BPM-Beispielprozess *Kundenpflege*

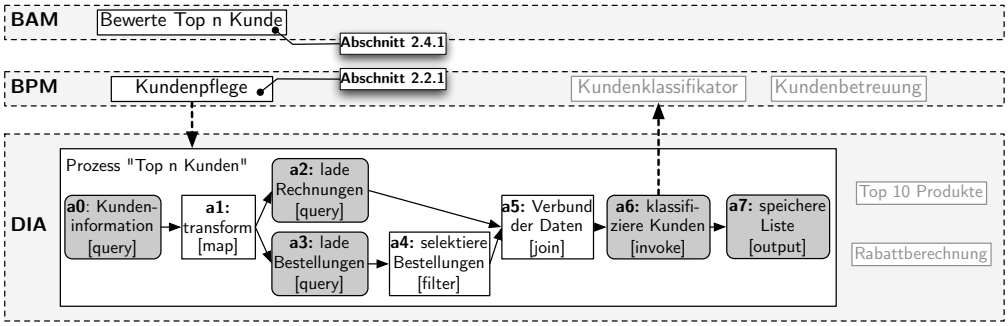


Abbildung 2.11: DIA-Prozess „*Top n Kunden*“

aus Abbildung 2.9 aufgerufen wird. Ziel dieses Dienstes ist die Berechnung der Top n Kunden anhand ihrer Umsätze in den letzten drei Monaten. Der Mindestumsatz u pro Kunde sowie die maximale Anzahl n an zurückzugebenden Kunden werden dem DIA-Prozess übergeben. Daraufhin extrahiert der Prozess alle Kundennummern der Kunden des Unternehmens mithilfe dieser zwei Prädikate (*a0*) und bereitet die Extraktion der Rechnungs- und Bestellungsdaten auf (*a1*). Danach werden nebenläufig alle Rechnungen und Bestellungen in den Prozess geladen (*a2* und *a3*). Bei den Bestellungen werden zudem offene, noch nicht bestätigte Bestellungen verworfen (*a4*). Nachfolgend werden alle Rechnungen und Bestellungen pro Kunde miteinander verbunden (*a5*) und an die Klassifizierungskomponente *Kundenklassifikator* weitergegeben, welche die Kundennummern mit den entsprechenden Verkaufsrängen zurückgibt (*a6*). Diese Komponente und deren Funktion wird über eine Dienstschnittstelle bereitgestellt. Schließlich ist das berechnete Ergebnis zu speichern bzw. an den aufrufenden Client zurückzugegeben (*a7*).

2.3.2 Referenzarchitektur

Die technische Perspektive einer gemeinsamen Referenzarchitektur für Datenintegration und Datenanalyse ist in Abbildung 2.12 dargestellt. Dabei liegt der Fokus dieser Architektur auf der Integration und Aufbereitung der Daten mithilfe von ETL bzw. Datenanalysealgorithmen. Beide basieren auf einem Datenflussgraph und interagieren mit Quell- und Zielsystemen. Die visuelle Darstellung von Analyseergebnissen bei Datenanalyseprozessen wird in dieser Abbildung vernachlässigt.

Entwicklungsphase

Wie auch in der SOA-Referenzarchitektur in Abschnitt 2.2.2 werden die Datenintegrations- bzw. Datenanalyseprozesse zunächst als Graph modelliert [191, 192]. Im Gegensatz zu den kontrollflussbasierten Prozessgraphen der dienstbasierten Geschäftsprozesse, wird in der Anwendungsklasse der Datenintegration- und Datenana-

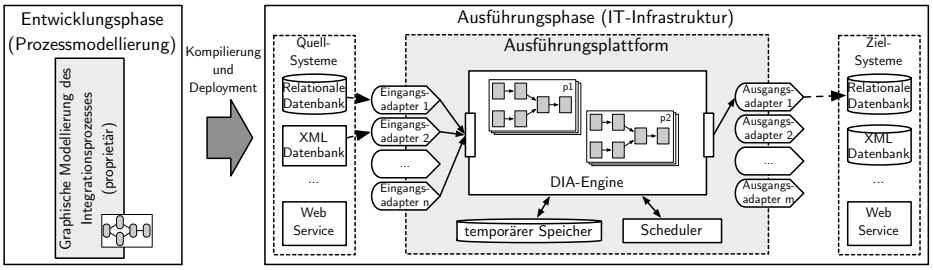


Abbildung 2.12: Technische DIA-Referenzarchitektur.

lyseprozesse ein datenflussorientierter Prozessgraph modelliert. Dieser beschreibt die Funktionen und Algorithmen, welche die Daten nacheinander durchlaufen. Da bisher keine standardisierte Modellierungsnotation in diesem Anwendungsgebiet existiert, werden in den einzelnen Systemen proprietäre Notationen und Symbole verwendet. Diese proprietären Notationen sind stark an die Leistungsfähigkeit der Systeme gekoppelt und spiegeln die Transformations- oder Analysealgorithmen der Systeme mit ihrer festgelegten Semantik direkt wider. Somit weisen diese Notationen bereits einen sehr hohen technischen Detailgrad auf und sind mit der *technischen Prozessbeschreibung* der dienstbasierten Prozesse vergleichbar. Eine abstrakte Modellierungsschicht wie beispielsweise BPMN bei den *dienstbasierten Geschäftsprozessen* fehlt derzeit. Auch die Integration beliebiger, verteilter und unabhängiger Funktionskomponenten über einheitliche Schnittstellen wurde bisher nicht betrachtet. Im Zuge der bisherigen Bereitstellung von Datenintegrationssystemen in SOA-Umgebungen wurde lediglich nachträglich die Möglichkeit geschaffen, Dienstaufrufe in den Datenflussgraphen des Integrationsprozesses zu integrieren. Die Effizienz dessen wird in Kapitel 3 näher betrachtet.

Ausführungsphase

Die in der Entwicklungsphase modellierten Prozesse werden analog zu den dienstbasierten Prozessen auf der proprietären Ausführungsplattform installiert und entweder zeit- oder ereignisbasiert gestartet. Im Gegensatz zu dienstbasierten Prozessen enthält die Eingangs- bzw. Initialisierungsnachricht eines DIA-Prozesses nur Parameter zur Konfiguration des Prozesses [177]. Alle zu integrierenden Nutzdaten werden daraufhin von den im Prozess angegebenen Quellsystemen geladen, in der DIA-Engine verarbeitet und letztendlich in die Zielsysteme geschrieben. Im Folgenden werden die in Abbildung 2.12 dargestellten Komponenten näher erläutert:

Eingabeadapter Eingabeadapter sind für die Kommunikation mit den Quellsystemen verantwortlich und realisieren den lesenden Zugriff auf diese Komponenten und die nachfolgende Transformation in das interne Datenformat der DIA-Engine. Sie sind direkt mit den Extraktionsaktivitäten des Prozesses verbunden, wobei meist de-

dizierte Extraktionsaktivitäten für den Zugriff auf herstellerspezifische Quellsysteme existieren [177].

DIA-Engine Die DIA-Engine führt die als Datenflussgraph modellierten Integrations- und Analyseprozesse aus. Diese Graphen werden zur Ausführungszeit in physische Datenflussgraphen transformiert, welche die Modifikationen der Daten zwischen Quell- und Zielsystemen beschreiben [177]. Die Verarbeitung dieser Daten auf Basis von klassischen, relationalen Datentupeln in DIA-Prozessen ermöglicht deren unabhängige Bearbeitung als Teilmenge der gesamten zu ladenden Relation [18]. Durch den expliziten Datenfluss und die Verarbeitungsgranularität der Datentupel werden die Daten entlang des Aktivitätsgraphs strombasiert verarbeitet. Die strombasierte Verarbeitung wird auf der technischen Ebene durch das *Pipes–and–Filters-Ausführungsmodell* [3, 115] realisiert. Prozessinstanzen werden voneinander isoliert verarbeitet und erlauben somit keine instanzübergreifende Korrelation.

Temporärer Speicher Dieser Speicher wird von zustandsbehafteten Aktivitäten zur Zwischenspeicherung von Daten und Ergebnissen genutzt, wenn das zu verarbeitende Datenvolumen die Größe des zugewiesenen Hauptspeichers überschreitet [44, 69]. Beispiele solcher zustandsbehafteten Aktivitäten sind Verbundoperationen auf unsortierten Daten oder holistische statistische Algorithmen wie beispielsweise die Berechnung des Medians. Des Weiteren wird der temporäre Speicher auch zur Speicherung von Kontrollpunkten (engl. *check points*) verwandt, um bei einem Systemfehler den gestarteten Prozess wieder an einem dieser Kontrollpunkte fortzusetzen. Dadurch kann die Latenz des Prozesses nach einem solchen Fehler reduziert werden [14, 34].

Scheduler Der Scheduler bearbeitet die Anfragen und initialisiert die entsprechenden Prozessinstanzen. Dabei werden Instanzen des gleichen Prozesstyps je nach Konsistenzanforderungen seriell ausgeführt, Instanzen unterschiedlicher Prozesstypen in der Regel parallel.

Ausgabeadapter Analog zu den Eingabeadaptern sind die Ausgabeadapter für die Kommunikation mit den Zielsystemen verantwortlich und realisieren den schreibenden Zugriff auf diese Komponenten. Sie sind direkt mit den Ladeaktivitäten des Prozesses verbunden. Meist existieren dedizierte Aktivitäten für den Zugriff auf herstellerspezifische Zielsysteme [177].

2.3.3 Charakteristika

Dieser Abschnitt fasst die für diese Arbeit relevanten System- und Anwendungseigenschaften der Anwendungsklasse *Datenintegration und Datenanalyse (DIA)* zusammen, um auf Basis dessen in Abschnitt 2.5 die Anforderungen an eine skalierbare Ausführungsplattform abzuleiten.

Die Struktur eines DIA-Prozesses P_{DIA} kann analog zur Definition des BPM-Prozesses (vgl. Abschnitt 2.2.3, Seite 22) wie folgt definiert werden:

Definition 4 *P_{DIA}: Ein Datenintegrations- und Datenanalyseprozess P_{DIA} wird definiert mit $P = (A, K, S)$ als gerichteter, azyklischer Graph (DAG). Dabei beschreibt A mit $A = (a_1, \ldots, a_i, \ldots, a_k)$ die Menge der im Prozessplan definierten Aktivitäten zur Verarbeitung der Daten als Knoten des Graphen. K beschreibt die Menge an Kanten, welche die Aktivitäten miteinander verbinden und ihre zeitliche Reihenfolge festlegen. Des Weiteren beschreibt S die Menge an externen Systemen, welche über Aktivitäten aufgerufen werden und mit denen der Prozess interagiert.*

Systemeigenschaften

Trifft eine Eingangsnachricht m_j oder ein zeitliches Ereignis im System ein, wird eine Prozessinstanz p_j eines definierten Prozesses generiert. Die in dieser Instanz zu verarbeitenden Daten werden isoliert von anderen Instanzen betrachtet. Dabei werden Instanzen gleichen Prozesstyps seriell ausgeführt [30]. Das Verarbeitungsmodell für derartige Prozessinstanzen ist an das physische Anfrageverarbeitungsmodell von Datenbankmanagementsystemen angelehnt und basiert auf einem definierten Datenfluss und einer strombasierten Verarbeitung der Verarbeitungseinheiten in einem Operatorgraphen [65, 67]. Das interne Datenmodell der meisten Systeme basiert aufgrund ihrer historischen Entwicklung und ihrer engen, geschichtlichen Verzahnung mit klassischen Datenbanksystemen auf dem Relationenmodell [21, 141, 153]. Dessen Verarbeitungseinheiten bilden flache, attributbasierte Tupel. Strukturierte Informationen auf Basis von XML werden in einem solchen Tupel als einfaches Attribut in XML-Notation abgelegt und entsprechend verarbeitet [21, 191]. Für die Kommunikation mit den jeweiligen externen Datenquellen unterstützen die Systeme der DIA-Anwendungsklasse eine Vielzahl von Datenformaten und Methodiken. So werden neben proprietären Formaten von Punkt-zu-Punkt-Verbindungen auch Formate für die nachrichtenbasierte XML-Kommunikation unterstützt.

Anwendungseigenschaften

Hinsichtlich der Beispielprozesse aus dem Abschnitt 2.3.1 lässt sich feststellen, dass diese Prozesse im Allgemeinen sehr hohe Datenvolumen verarbeiten müssen. Diese großen Datenmengen entstehen sowohl während des Prozesses zwischen den Aktivitäten als auch bei der Kommunikation mit externen Systemen. Bei Einordnung dieser Eigenschaft in die in Abschnitt 2.2.3 definierte Dimension *Nachrichtengröße* G, repräsentieren DIA-Prozesse die obere Grenze aller hier eingeführten Klassen. Hinsichtlich der Dimension *Nachrichtenrate* R ist die DIA-Anwendungsklasse als gering einzuschätzen, da DIA-Prozesse je nach Prozesstyp traditionell nur im Tage- oder Wochenrythmus ausgeführt werden [150].

Zusammenfassend lässt sich festhalten, dass sich die Anwendungsklasse der Datenintegration- und Datenanalyseprozesse und ihre Systeme durch hohe zu verarbeitende Datenvolumen innerhalb einer Prozessinstanz und zwischen den dazugehörigen ex-

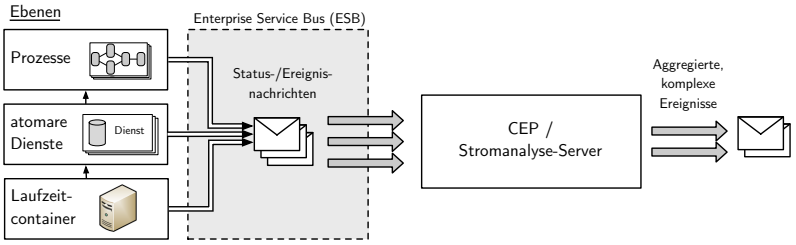


Abbildung 2.13: Überblick einer SOA-Monitoring-Infrastruktur.

ternen Partnern auszeichnen. Dadurch definieren DIA-Prozesse die obere Grenze der Dimension *Nachrichtengröße* aller hier betrachteten Anwendungsklassen. Die effiziente Abarbeitung der im Prozess zu verarbeitenden Daten erfolgt über eine Partitionierung der Datenmenge und eine strombasierte Verarbeitung der internen Datentupel im Operatorgraphen. Wie diese System- und Anwendungseigenschaften im Hinblick auf eine einheitliche Ausführungsplattform für dienstbasierte Anwendungen zu bewerten sind, wird in Kapitel 3 detailliert diskutiert.

2.4 Ereignismonitoring

Die dritte für diese Arbeit relevante Anwendungsklasse und die zweite unterstützende Klasse für *dienstbasierte Prozesse* ist die Nachrichtenstromanalyse und damit die Überwachung (engl. *monitoring*) von Ereignissen in SOA-Infrastrukturen. Grundlage dafür sind Ereignis- und Statusnachrichten verschiedener Software-Komponenten, welche in den Ausführungsumgebungen dieser Komponenten generiert und über eine gemeinsame zentrale Komponente wie den ESB verteilt werden [84]. Abbildung 2.13 gibt einen Überblick über eine solche grundlegende Infrastruktur. Über definierte Metriken lassen sich die Statusnachrichten auswerten und eine Aussage über die aktuelle Situation der Infrastruktur treffen. Folgende drei Überwachungsebenen der SOA-Infrastruktur können unterschieden werden:

Auf der *Ebene der Prozesse* werden die orchestrierten technischen Geschäftsprozesse überwacht. Es lassen sich dabei zwei Arten von Statusnachrichten unterscheiden. Zum einen werden Eigenschaften der Prozessausführung überwacht und in der Infrastruktur propagiert. Sensoren für die Überwachung solcher Eigenschaften sind inhärent in der Ausführungsumgebung der technischen Geschäftsprozesse verankert und unabhängig vom jeweiligen Prozesstyp. Beispiele solcher Statusnachrichten sind die Anzahl der Prozessaufrufe pro Zeiteinheit oder die Laufzeit der einzelnen Prozessinstanzen. Damit lassen sich Engpässe bei Hardwareressourcen oder auch technisch auffällige Geschäftsprozesse überwachen. Zum anderen können prozessspezifische Statusnachrichten in den Prozessen selbst verankert werden [89]. Diese geben Auskunft über prozessinterne Entscheidungen und über Instanzzustände zu definierten Zeitpunkten. Ein Beispiel für eine prozessspezifische Statusnachricht ist die Wahl

des Pfades bei einer Prozessverzweigung. Durch prozessspezifische Statusnachrichten werden prozessübergreifende und zeitbezogene Analysen über die Prozesshistorie möglich und es können Rückschlüsse auf reale Geschäftsprozesse gezogen und eventuelle Anpassungen veranlasst werden.

Auf der *Ebene der atomaren Dienste* werden die Dienste und deren Instanzen überwacht. Statusnachrichten auf dieser Ebene beschränken sich meist auf technische Aspekte der Ausführung wie die Laufzeit und die Arbeitslast. Auch auf der *Ebene der Laufzeitcontainer*, auf der alle Prozesse und atomaren Dienste verteilt ausgeführt werden, fallen Statusinformationen an. In diesem Fall werden jedoch dienst- und prozessunabhängige Status überwacht und als Nachricht propagiert. Beispiele hierfür sind die Gesamtauslastung des Systems in Bezug auf CPU und Hauptspeicher oder auch die Netzwerkbandbreite, mit der auf den Laufzeitcontainer zugegriffen wird. Im Rahmen des Projektes THESEUS.TEXO wurden zur Anwendungsklasse des Ereignismonitorings zahlreiche Arbeiten veröffentlicht, die sich mit der Generierung von Statusnachrichten auf den verschiedenen Ebenen [89, 144], der Definition von ebenenspezifischen Metriken [89, 154] und somit mit deren Überprüfbarkeit im Rahmen von *Service Level Agreements* (SLA) [143, 145] befassen. Eine Ausführungsumgebung zur Überprüfung dieser SLAs wurde jedoch auch hier nicht betrachtet.

2.4.1 Beispielprozess

Dieser Abschnitt beschreibt den in der vorliegenden Arbeit zur Veranschaulichung genutzten Beispielprozess *Bewerte Top n Kunde* der BAM-Anwendungsklasse, welcher die Ausführung des BPM-Prozesses *Kundenpflege* analysiert und gegebenenfalls weitere Ereignisse generiert.

Datengrundlage für den BAM-Prozess *Bewerte Top n Kunde* bilden die Nachrichten von definierten Sensoren [89] im BPM-Prozess *Kundenpflege* an verschiedenen Stellen entlang des Prozesspfades. Diese propagieren neben den Zeitstempeln ihrer Generierung weitere Zustandsinformationen. Folgende Sensoren und Inhalte sind im BPM-Prozess *Kundenpflege* definiert:

SensorID	Enthaltene Statusinformationen
Sensor1	Signalisierung des Prozessstarts.
Sensor2	Anzahl tatsächlich berechneter Top n Kunden.
Sensor3	Pro Kunde, der nicht unter den aktuellen Top 50 Kunden ist, eine Nachricht mit der Kundennummer, dem aktuellen Umsatz und dem Rang des Kunden.
Sensor4	Zeitstempel und Anzahl der insgesamt verarbeiteten Kunden.

Abbildung 2.14 visualisiert neben den oben genannten Sensoren den eigentlichen, hier betrachteten BAM-Prozess *Bewerte Top n Kunde* und stellt die Kommunikation mit externen Systemen der anderen Anwendungsklassen dar. Aufgabe des BAM-Prozesses ist es, ausschließlich Kunden zu erkennen, die im Vergleich zum

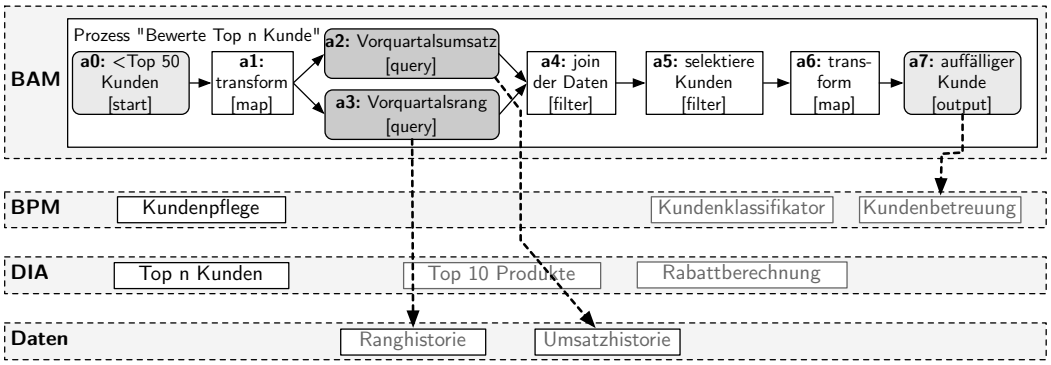


Abbildung 2.14: BAM-Beispielprozess „*Bewerte Top n Kunde*“.

Vorquartal im Rang sowie im Umsatz signifikant gesunken sind. Der BAM-Prozess wurde für diese Aufgabe nur für den Empfang von Nachrichten des Sensors *Sensor3* registriert und verarbeitet somit auch nur Nachrichten von Kunden, welche nicht zu den Top 50 Kunden des Unternehmens gehören. Empfängt nun dieser BAM-Prozess eine Statusnachricht, wird diese zunächst transformiert (*a1*). Auf Grundlage der Kundennummer werden aus den historischen Daten der vorherige Quartalsumsatz (*a2*) sowie der vorherige Quartalsrang (*a3*) des Kunden abgefragt und beide Ergebnisse miteinander vereinigt (*a4*). Die darauf folgende Filter-Aktivität (*a5*) überprüft zum einen, ob der Kunde im Vorquartal noch zu den Top 50 Kunden gehörte. Ist dies der Fall, war der Kunde bereits Premiumkunde des Unternehmens und ist nun zurückgefallen. Zum anderen wird überprüft, ob der Umsatz des Kunden um mehr als 20 Prozent im Vergleich zum Vorquartal zurückgegangen ist. Trifft dies nicht zu, so hat das Unternehmen andere, umsatzstärkere Kunden hinzugewonnen, wodurch sich zwar die Rangplatzierung des betrachteten Kunden verschlechtert hat, aber der Kunde selbst unauffällig geblieben ist. Sank jedoch zusätzlich der Umsatz des Kunden um mindestens 20 Prozent im Vergleich zum Vorquartal, leidet der Kunde 2) entweder unter finanziellen Problemen, 2) bezieht den Großteil seiner Produkte nun von einem anderen Anbieter oder 3) benötigt diese Produkte nicht mehr in dieser Menge. Somit wird der Kunde als auffällig eingestuft, die Daten dem Ausgangsformat entsprechend transformiert (*a6*) und eine manuelle Beurteilung und Betreuung des Kunden durch eine Ereignisnachricht angestoßen (*a7*).

2.4.2 Referenzarchitektur

Abbildung 2.15 zeigt analog zu den vorherigen Anwendungsklassen die technische Referenzarchitektur für eine Monitoring-Ausführungsumgebung. Dabei liegt der Fokus auf der Definition und Überwachung von Anfragemustern und Metriken, d.h. dem zeitlich abhängigen Auftreten von Ereignissen sowie von prozessübergreifenden, historienbasierten Kennzahlen. Es lassen sich verschiedene Systemtypen für derartige Aufgaben unterscheiden, die in ihren Fähigkeiten zur Unterstützung der

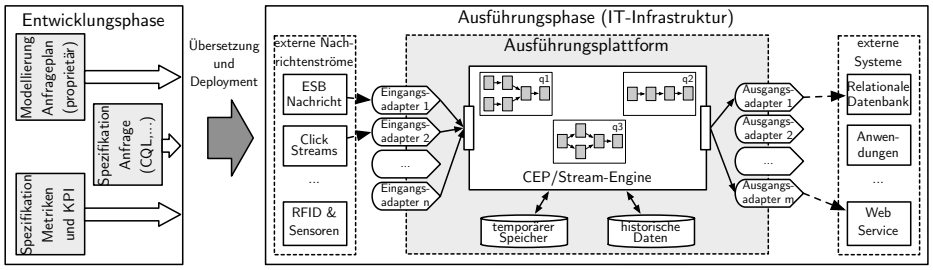


Abbildung 2.15: Technische Monitoring-Referenzarchitekur.

definierten Anfragemuster und Metriken voneinander abweichen. Neben den klassischen Datenstromsystemen [9, 76, 199] existieren auch Regelmaschinen [185] oder Systeme basierend auf endlichen Automaten [49].

Entwicklungsphase

Die Definition der Anfragemuster bzw. der zu überwachenden Metriken unterscheidet sich in Abhängigkeit des Systemtyps. Die erste Möglichkeit für die Definition von Datenstromanfragen bilden Datenstromanfragesprachen. Heutige Datenstromanfragesprachen basieren stets auf einer Erweiterung von SQL und beinhalten den Ansatz von Fenstersemantiken über Datenströmen, um relationale Operatoren auf einen theoretisch unendlichen Datenstrom anzuwenden [10, 198]. Im Detail unterscheiden sich die Anfragesprachen der verschiedenen Hersteller in dem zugrundeliegenden Strommodell sowie der Unterstützung zeitlich bedingter Anfragemuster [49, 82] für das Generieren von höherwertigen Ereignissen aus mehreren einfachen Ereignissen (engl. *complex event processing* (CEP)). Die zweite Möglichkeit für die Definition von Datenstromanfragen besteht in der graphischen Modellierung der Datenflüsse und ihrer Operatorensequenz. Diese Modellierung gliedert sich in die graphenbasierte Modellierung der ETL- bzw. Analyseprozesse ein und ist auch hier proprietär [199]. Den dritten Ansatz zur Definition von Datenstromanfragen bildet die Spezifikation von Metriken und Kennzahlen, den sogenannten *Key Performance Indicators* (KPI). Diese stellen wirtschaftliche Sachverhalte dar und ermöglichen die Ableitung der Qualität und Effizienz der eigentlichen Geschäftsprozesse [89].

Ausführungsphase

In dieser Phase werden die vorher definierten Anfragemuster bzw. Metriken ausgeführt. Die Grundlage der hier beschriebenen Ausführungsplattform bildet der Systemtyp der Datenstrommanagementsysteme, da die Mehrzahl der derzeitigen Produkte auf diesem Systemtyp basiert [176, 199]. Im Folgenden werden die einzelnen Komponenten der Ausführungsphase detaillierter beschrieben:

Eingabeadapter Eingabeadapter sind für die Kommunikation und Transformation von Eingangsnachrichten in das interne Datenmodell des Datenstromsystems verantwortlich. Eingangsdatenströme bilden beispielsweise einfache binäre Sensordatenströme, einfache textuelle Webprotokoll-Ereignisse oder auch XML-basierte Nachrichten der drei zuvor genannten Ebenen, welche über den ESB verteilt werden.

Stream-Engine Die zentrale Komponente der Ausführungsplattform ist die Stream-Engine. Diese beinhaltet die physische Repräsentation der zuvor definierten Anfragemuster und Metriken in Form eines Operatorgraphen [1, 13, 96, 132]. Durch eine Instanz dieses Graphen fließen die Stromelemente der externen Datenquellen. Im Kontext des Ereignismonitorings entspricht ein solches Stromelement dem Inhalt einer Nachricht. Aufgrund der Semantik von statischen, langlaufenden Anfragen (Operatorgraphen) an den Datenstrom wurde der Begriff „stehende“ bzw. kontinuierliche Anfrage geprägt [13, 149] und eine Korrelation der Nachrichteninhalte im Operatorgraphen der Instanz mit ihrer zeitlichen Reihenfolge möglich.

Temporärer Speicher Analog zur DIA-Referenzarchitektur wird der Temporäre Speicher zum Zwecke der Wiederherstellung und Ausfallsicherheit benötigt, da in periodischen Abständen die Zustände der Operatoren in diesem persistenten Speicher abgelegt werden [34, 77].

Speicher für historische Daten Dieser persistente Speicher ermöglicht den Zugriff auf vergangene Nachrichten, Metriken und generierte Ereignisse, wodurch globale Aussagen und Langzeitanalysen mit den aktuellen Nachrichten verknüpft werden können. Außerdem lassen sich durch die Speicherung vergangener Ereignisse zeitliche Abfolgen und Korrelationen auswerten und damit CEP realisieren [49, 107].

Scheduler Der Scheduler überwacht die Zuteilung von Hardwareressourcen zu den verschiedenen Operatoren der einzelnen Anfragen im System. Dies ermöglicht eine priorisierte Verarbeitung der unterschiedlichen Anfragen.

Ausgabeadapter Ausgabeadapter transformieren die durch die *Stream-Engine* generierten komplexen Ereignisse vom internen Datenmodell in das externe Nachrichtenformat und realisieren den Versand der Ausgangsnachricht.

2.4.3 Charakteristika

Der folgende Abschnitt fasst die für diese Arbeit relevanten System- und Anwendungseigenschaften der Anwendungsklasse *Ereignismonitioring (BAM)* zusammen, um in Abschnitt 2.5 die Anforderungen an eine einheitliche skalierbare Ausführungsplattform abzuleiten. Der Aufbau eines BAM-Prozesses kann analog zu den Definitionen für BPM- und DIA-Prozesse wie folgt beschrieben werden und ist unabhängig vom verwendeten Systemtyp:

Definition 5 *Ein Prozess zur Analyse von Nachrichtenströmen P_{BAM} wird definiert mit $P = (A, K, S)$ als gerichteter, azyklischer Graph (DAG). Dabei beschreibt A mit $A = (a_1, \ldots, a_i, \ldots, a_k)$ die Menge der im Prozessplan definierten Aktivitäten als Knoten des Graphen und K die Menge an Kanten, welche die Aktivitäten miteinander verbinden und ihre zeitliche Reihenfolge festlegen. Des Weiteren beschreibt S die Menge an externen Systemen, mit denen der Prozess zur Erfüllung seiner Aufgabe über spezielle Aktivitätstypen interagiert.*

Systemeigenschaften

Die hier beschriebenen Systemeigenschaften betrachten den Systemtyp der Datenstrommanagementsysteme (DSMS) als traditionellen Vertreter dieser Anwendungssysteme [176, 199]. Im Gegensatz zu den Systemen der vorherigen beiden Anwendungsklassen werden in der Anwendungsklasse des Ereignismonitorings pro Eingangsnachricht m_j nicht jeweils eine Prozessinstanz generiert und deren Daten isoliert voneinander verarbeitet. Vielmehr wird die Prozessdefinition bzw. die „Anfrage“ an den Nachrichtenstrom direkt in der Ausführungsumgebung registriert und instanziiert. Eintreffende Nachrichten werden in einer gemeinsamen Instanz verarbeitet und über eine strombasierte Verarbeitung entlang des Datenflusses im Operatorgraph weitergegeben. Somit arbeiten die Operatoren nachrichtenübergreifend und können notwendige inhaltliche und zeitliche Korrelationen zwischen den Nachrichten auswerten. Die Verarbeitungseinheiten dieser kontinuierlichen Prozesse bilden die einzelnen Nachrichten bzw. deren Nutzlast.

Das interne Datenmodell orientiert sich am Relationenmodell der klassischen Datenbanksysteme und beschreibt eine Verarbeitungseinheit als flaches, attributbasiertes Datentupel [1, 96], welches sich nur bedingt zur Verarbeitung hierarchisch strukturierter XML-Daten eignet und eine Abbildungsvorschrift für die Transformation von externen hierarchischen Nachrichtenformaten auf das interne Datenmodell benötigt. Für die Kommunikation mit externen Partnern unterstützen die Systeme dieser Anwendungsklasse historisch gewachsen eine Vielzahl von Protokollen, Formaten und Methodiken ähnlich denen der DIA-Systeme aus Abschnitt 2.3.2.

Anwendungseigenschaften

Die hier dargestellten Anwendungen mit ihren XML-basierten Nachrichtenströmen sind zum einen durch eine hohe Eingangsnachrichtenrate in einem definierten Zeitintervall, welche eine Prozessinstanz verarbeiten muss, geprägt. Diese Eigenschaft definiert die obere Grenze der Dimension *Nachrichtenrate R*. Zum anderen zeichnen sich die Eingangsnachrichten und die durch jede Eingangsnachricht bedingten Datenvolumina in der Prozessinstanz durch ihre einfache Struktur und ihr geringes Datenvolumen aus. Somit ist die *Nachrichtengröße G* als relativ gering einzustufen.

Merkmal/Klasse	Datenintegration/ Datenanalyse	dienstbasierte Geschäftsprozesse	Ereignismonitoring
Flussmodellierung	graphisch, datenflussbasiert	graphisch, kontrollflussbasiert	graphisch, datenflussbasiert, deklarativ
Physische Realisierung	datenflussbasierter Aktivitätsgraph	kontrollflussbasierter Aktivitätsgraph	datenflussbasierter Aktivitätsgraph
Verarbeitungsmodell	strombasierte Verarbeitung der Daten, nebenläufig in allen Aktivitäten im Graph	schrittweise, blockbasierte Abarbeitung der Aktivitäten im Graph	strombasierte Verarbeitung der Daten, nebenläufig in allen Aktivitäten im Graph
Internes Datenformat	flach, attributbasiert	hierarchisch strukturiert, baumbasiert	flach, attributbasiert
Kommunikation	proprietär und nachrichtenbasiert	nur nachrichtenbasiert	proprietär und nachrichtenbasiert
Instanzgenerierung	eine Instanz pro Eingangsnachricht	eine Instanz pro Eingangsnachricht	eine Instanz für eine Menge von Eingangsnachrichten

Tabelle 2.1: Merkmalszusammenfassung der vorgestellten Anwendungsklassen.

Zusammenfassend lässt sich festhalten, dass die Anwendungen und Systeme der Anwendungsklasse *Ereignismonitoring* eine hohe *Eingangsnachrichtenrate* R verarbeiten und zudem nachrichtenübergreifende Korrelationen ermöglichen müssen. Dies wird durch das Konzept der kontinuierlichen Prozessinstanzausführung realisiert, in welchem alle Eingangsnachrichten über eine strombasierte Semantik in einer Instanz des Prozessgraphen verarbeitet werden.

2.5 Zusammenfassung

In diesem Kapitel wurden die drei in der vorliegenden Arbeit betrachteten Anwendungsklassen, welche direkt oder indirekt an der Realisierung der realen Geschäftsprozesse beteiligt sind und über eine SOA-Infrastruktur miteinander kommunizieren, eingeführt und analysiert. Dabei nimmt die Klasse der *dienstbasierten Geschäftsprozesse* die zentrale Rolle ein. Die Hauptmerkmale der in den vorherigen Abschnitten diskutierten Anwendungsklassen und ihrer Systeme sind in Tabelle 2.1 nochmals zusammengefasst. Den Hintergrund der erfolgten Betrachtung bildet der Trend, dass diese drei vormals orthogonalen und domainspezifisch entstandenen Anwendungsklassen immer weiter miteinander integriert werden müssen [195], da deren Synergien für eine erfolgreiche Unternehmensentwicklung notwendig sind [181]. Eine Integration der Modellierung und Ausführung von Teilprozessen der beiden Anwendungsklassen DIA und CEP in die dienstbasierte Infrastruktur und in die Definition der dienstbasierten Geschäftsprozesse ist dabei wünschenswert; entsprechende Beispielprozesse wurden präsentiert. Außerdem reduziert Integration der Anwendungsklassen die Komplexität und Fehleranfälligkeit der beteiligten Systeme [195, 197] und eine skalierbare Ausführungsumgebung ermöglicht die Nutzung des verteilten Kom-

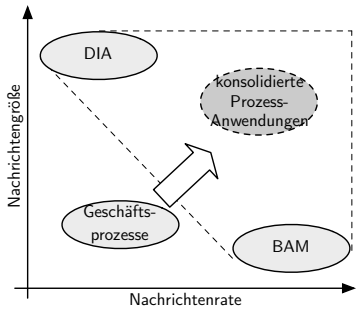


Abbildung 2.16: Gegenüberstellung der Anwendungseigenschaften.

ponentenmodells einer SOA, mit seiner prozessorientierten Orchestrierung, auch für weitere Anwendungsklassen.

Die Referenzarchitekturen der diskutierten Anwendungsklassen sind in ihren Grundkomponenten ähnlich aufgebaut. Alle Systeme nutzen das Konzept der Ein- bzw. Ausgabeadapter, instrumentalisieren temporäre persistente Datenspeicher für Wiederherstellung und Ausfallsicherheit und haben eine zentrale Ausführungsumgebung für zuvor definierte Prozesse. Eine im Grundsatz einheitliche Architektur wäre somit möglich. Auch die allgemeine Struktur der Prozesse ist bei allen drei unterschiedlichen Anwendungsklassen äquivalent. Der Hauptunterschied der Anwendungsklassen liegt innerhalb der jeweiligen Ausführungsumgebungen und spiegelt sich in deren unterschiedlichen Verarbeitungsmodellen sowie deren internen Datenmodellen wider. Diese Hauptunterschiede sind Folge der spezifischen Datencharakteristiken der Anwendungsklassen, da die Verarbeitungsmodelle sowie die internen Datenmodelle speziell für die jeweiligen Datencharakteristiken konzipiert wurden.

Abbildung 2.16 zeigt die Dimensionen *Nachrichtengröße* und *Nachrichtenrate* der drei Anwendungsklassen. Beim Vergleich der drei Anwendungsklassen zeichnet sich die Hauptanwendungsklasse der *dienstbasierten Geschäftsprozesse* dadurch aus, dass die pro Prozessinstanz zu verarbeitende Nachrichtengröße gering ausfällt und dass auch die Frequenz der Eingangsnachrichten im unteren Bereich der hier betrachteten Anwendungen liegt. Im Gegensatz dazu haben Anwendungen der *Datenintegration und Datenanalyse*-Klasse die Eigenschaften, dass deren zu verarbeitende *Nachrichtengröße* sehr groß ausfällt und entsprechende Systeme dafür ausgelegt sein müssen, auch Datenmengen zu verarbeiten, welche nicht vollständig zur Bearbeitung im Hauptspeicher abgelegt werden können. Derartige Anwendungen werden im Gesamtvergleich sehr selten aufgerufen, meist auf Tages- oder Wochenbasis. Ein Trend zu zeitnäheren Integrationsflüssen und Analysen ist jedoch erkennbar [150] und wird durch die Integration in die dienstbasierte Prozesswelt begünstigt. Schlussendlich zeichnet sich die Klasse der *Nachrichtenstromanalyse* dadurch aus, dass die zu verarbeitende Nachrichtengröße aufgrund einfacher Ereignisnachrichten zwar sehr klein

ausfällt, jedoch die Nachrichtenrate, mit welcher die Ereignisnachrichten im System eintreffen, die obere Grenze der drei Anwendungsklassen markiert.

Der Trend zu erweiterten bzw. konsolidierten dienstbasierten Prozessanwendungen, welche Teilprozesse der DIA- und BAM-Klassen beinhalten, setzt die Unterstützung aller bisher genannten Datencharakteristiken durch eine konsolidierte Ausführungsumgebung voraus. Aus dem Vergleich der notwendigen Eigenschaften von Anwendungen und Systemen der einzelnen Anwendungsklassen sowie aus den detaillierten Betrachtungen der vorangegangenen Abschnitte lassen sich die sechs folgenden Anforderungen an ein integriertes Verarbeitungssystem ableiten. Dabei beschreibt Skalierbarkeit im Folgenden sowohl die Fähigkeit der fehlerfreien Verarbeitung als auch die effiziente Ausnutzung der zur Verfügung stehenden Ressourcen:

Anforderung A1 (Verarbeitung): *Skalierbare Datenverarbeitung zwischen und innerhalb der Aktivitäten eines Prozesses.* Diese Anforderung beschreibt die Eigenschaft eines Systems, beliebig große Datenmengen in einer Prozessinstanz zu verarbeiten ohne beispielsweise an Hauptspeichergrenzen gebunden zu sein.

Anforderung A2 (Datenaustausch): *Skalierbarer Austausch beliebiger Datenmengen zwischen Prozess und entfernten Komponenten.* Diese Anforderung beschreibt zum einen die effiziente Kommunikation mit externen Systemen und zum anderen die Fähigkeit dieser externen Systeme, diese beliebigen Datenmengen skalierbar zu verarbeiten. Diese Anforderung ist gerade im Hinblick auf die Verteilung von dienstähnlichen Berechnungskomponenten für vielseitige Algorithmen grundlegend.

Anforderung A3 (Latenz): *Skalierbare und effiziente Verarbeitung hoher Eingangsnachrichtenraten.* Im Zuge einer steigenden Anzahl von Überwachungskomponenten in der IT-Infrastruktur und der Integration von Teilprozessen der Nachrichtenstromanalyse ist die effiziente und zeitnahe Verarbeitung der anfallenden Nachrichtenmenge bedeutend. Da bei Nutzung entfernter externer Systeme in der Prozessinstanz deren Kommunikationsrate durch die Eingangsnachrichtenrate vorgegeben wird, gilt diese Anforderung auch für die Verarbeitung bei den extern genutzten Systemen.

Anforderung A4 (Korrelation): *Inhärente (und damit effiziente) Möglichkeit zur Korrelation aufeinanderfolgender Nachrichten bzw. Datenelementen.* Diese Anforderung ist eine Notwendigkeit für nachrichten- bzw. datenelementübergreifende Operationen wie beispielsweise Musterkennung oder holistische Algorithmen. Sie gilt innerhalb der Aktivitäten einer Prozessinstanz sowie bei der Kommunikation mit involvierten externen Komponenten.

Anforderung A5 (Datenmodell): *Native Unterstützung von hierarchisch strukturierten Daten im systeminternen Datenmodell.* Diese Anforderung trägt der starken Nutzung und Verbreitung strukturierter Austauschformate, welche ohne eine Abbildungsvorschrift auf flache Datenmodelle von System unterstützt werden sollte, Rechnung. Im Speziellen die native Unterstützung von XML im

Datenmodell würde es erlauben, die ausgereiften Spezifikationen zur Abfrage [211] und Modifikation [20, 206, 212] von XML-Daten zu nutzen und somit die Effizienz und Nutzbarkeit der Anwendung maßgeblich positiv zu beeinflussen.

Anforderung A6 (Dienstsemantik): *Native Integration der Dienstsemantik für beliebige dienstbasierte Operatoren und deren skalierbare Kommunikation.* Diese Anforderung beschreibt die Eigenschaft, beliebige Aktivitäten der Prozessinstanz als externe Komponenten zu integrieren, ohne die Anforderungen A1 bis A5 zu verletzen. Dabei soll die standardisierte Nutzung der Komponenten über XML-basierte Nachrichtenkommunikation gewährleistet bleiben.

Diese sechs Anforderungen beschreiben die grundsätzlichen Eigenschaften, die ein skalierbares Verarbeitungssystem für erweiterte dienstbasierte Anwendungen aufweisen muss. Dabei spielen die verteilten Komponenten als beliebig entfernte Aktivitäten eine besondere Rolle. Die hier abgeleiteten Anforderungen sind Grundlage der weiteren Diskussion in den folgenden Kapiteln. Kapitel 3 bewertet existierende Ansätze der unterschiedlichen Ausführungsebenen im Hinblick auf diese Anforderungen und leitet eine Zielarchitektur ab. Die Anforderungen A2, A3 und A4 und deren Umsetzung auf Dienstebene sind Gegenstand von Kapitel 4. Die Ebene der Dienstorchestrierung und die damit zusammenhängenden Anforderungen A1, A2, A3, A4, A5 und A6 werden in Kapitel 5 betrachtet.

3 Problembeschreibung und Zielarchitektur

Auf Grundlage der sechs im vorherigen Kapitel definierten Anforderungen an ein skalierbares Verarbeitungssystem beschreibt dieses Kapitel nun die Verarbeitungsmodelle der einzelnen Anwendungsklassen in Abschnitt 3.1 sowie deren Kommunikationseigenschaften in Abschnitt 3.2 und bewertet die Teilaspekte hinsichtlich ihrer Eignung, die definierten Anforderungen zu erfüllen. Dabei werden verwandte Arbeiten dieser Teilaspekte diskutiert und schlussendlich in Abschnitt 3.3 eine neue Zielarchitektur abgeleitet.

3.1 Verarbeitungsmodelle der Prozessausführung

Dieser Abschnitt beschreibt und bewertet die *Verarbeitungsmodelle* der einzelnen Anwendungssysteme bezüglich der aufgestellten Anforderungen *A1* bis *A6*. Im Kontext dieser Arbeit lassen sich drei Klassen von Anwendungssystemen unterscheiden: Workflow-Systeme, Datenmanagementsysteme sowie hybride bzw. alternative Ansätze. Workflow-Systeme repräsentieren dabei die Anwendungsklasse der *dienstbasierten Prozesse*, wohingegen Datenmanagementsysteme die Anwendungsklassen *DIA* und *BAM* dominieren.

3.1.1 Workflow-Systeme

Wie bereits beschrieben, basieren Ausführungssysteme der *dienstbasierten Prozesse* auf den Konzepten der synchronen, nachrichtenbasierten Kommunikation traditioneller Workflow-Systeme [99, 121]. Zusätzlich integrieren aktuelle Systeme jedoch auch die asynchrone Kommunikation der Prozesse über Nachrichtenwarteschlangen (engl. *message queueing*) von dedizierten Message-Queueing-Systemen wie beispielsweise Flow Mark MQ [118], WebSphere MQ [183] oder ActiveMQ [172]. Dabei platzieren Prozesse ihre Anfragen in ausgehenden Warteschlangen und blockieren ihre Ausführung nicht bis zum Eintreffen der Antwortnachricht. Dies ermöglicht die Entkopplung von Anfragen und deren Antwortnachrichten und verstärkt somit auch die lose Kopplung zwischen Prozessen und anderen Komponenten. Jedoch steigt die Komplexität der Kommunikationsmuster und damit auch die Komplexität der Prozessdefinition [86]. Als Prozessbeschreibung definieren Workflow-Systeme die

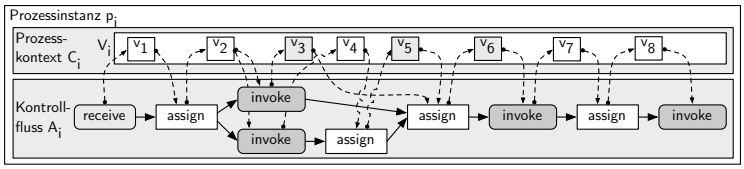


Abbildung 3.1: Kontrollflussbasierte Ausführung traditioneller Workflow-Systeme.

notwendigen Aktivitäten und deren zeitliche Abfolge mit den dabei auftretenden Informations- bzw. Datenflüssen [60]. Durch die strikte zeitliche Ordnung und die schrittweise Verarbeitung der Aktivitäten entspricht die Ausführungssemantik von Workflow-Systemen einer kontrollflussbasierten Ausführung [22, 91]. Ein Prozessplan P wird auf Grundlage von Definition 1 somit wie folgt definiert:

Definition 6 Prozessplan P_{BPM}: *Ein Prozessplan P_{BPM} wird definiert mit $P = (C, A, K, S)$ als gerichteter, azyklischer Graph (DAG). Dabei beschreibt C den Prozesskontext und A mit $A = (a_1, \ldots, a_i, \ldots, a_k)$ die Menge an Aktivitäten als Knoten des Graphen. Weiterhin definiert K die Menge an Kanten des Graphen, welche die Aktivitäten miteinander verbinden und ihre zeitliche Reihenfolge als Kontrollfluss festlegen. Abschließend beschreibt S die Menge von externen Diensten, mit denen ein Prozess interagiert. Diese Dienste werden durch einen speziellen Aktivitätstypen im Graph aufgerufen und darin als Eigenschaften referenziert. Der Prozesskontext C wird definiert als $C = (S_C, V)$ mit S_C als einer Menge von Systemeigenschaften, wie beispielsweise der Prozess-ID, und V als einer Menge von Variablen mit $V = (v_1, \cdots, v_i, \cdots, v_m)$, die von den Aktivitäten als deren Datenfluss referenziert werden. Mit jeder Eingangsnachricht $m_i \in M$ $(1 \leq i \leq n)$ wird eine Prozessinstanz $p_i \in P$ mit $p_i = (C_i, A_i, S_i)$ generiert und einmalig ausgeführt, sodass $m_i \rightarrow p_i$.*

Kontrollflussbasierte Ausführung

Zur Bewertung der Anforderungen *A1 (Verarbeitung)* und *A2 (Datenaustausch)*, d.h. der effizienten Verarbeitung großer Datenvolumen zwischen den Aktivitäten A eines Prozesses und dem effizienten Austausch beliebiger Datenmengen zwischen dem Prozess und seinen entfernten Datencontainern S, werden zunächst die Abarbeitung von Aktivitäten und die Datenverwaltung in einer Prozessinstanz p_i näher diskutiert. Abbildung 3.1 basiert auf Beispielprozess 2 aus Abbildung 2.11 und visualisiert den schematischen Aufbau einer solchen Instanz. Der Kontrollfluss A besteht neben den Aktivitäten auch aus den temporalen Beziehungen der Aktivitäten zueinander. Die Abarbeitung dieses Kontrollflussgraphen wird für jede Eingangsnachricht m_i gestartet und erfolgt schrittweise entlang des Graphen. Dabei wird eine Aktivität ausgeführt, ihre erfolgreiche Abarbeitung abgewartet und erst dann die nachfolgende Aktivität gestartet [23, 30]. Diese ganzheitliche Abarbeitung einzelner Aktivitäten

entspricht dem synchronen Aufrufmuster externer Workflows und korreliert direkt mit dem strikten *Anfrage-Antwort-Paradigma* von Dienstkomponenten, welches in Abschnitt 2.1.3 eingeführt wurde und dessen Bewertung in Abschnitt 3.2 folgt. Für den Datenfluss zwischen den Aktivitäten in A_i wird die Menge an Variablen V_i im Prozesskontext C_i genutzt. Jede Aktivität $a_j \in A_i$ einer Prozessinstanz referenziert eine oder mehrere Variablen $v_k \in V_i$ für deren Ein- und Ausgabedaten. So wird beispielsweise die Variable v_1 sowohl als Ausgabespeicher von *receive* als auch als Eingabespeicher für das nachfolgende *assign* referenziert.

Das schrittweise Ausführungsmodell sowie der variablenbasierte Datenfluss implizieren, dass alle Ein- und Ausgabedaten der einzelnen Aktivitäten als Ganzes in die Prozessinstanz geladen und zur Verarbeitung gehalten werden müssen. Übersteigt die Menge der zu ladenden Daten die der Prozessinstanz zur Verfügung stehenden Speicherressourcen, ist eine erfolgreiche Abarbeitung der Prozessaktivitäten nicht möglich und die Prozessinstanz terminiert vorzeitig. Durch die Berechnung der Datenabhängigkeiten zwischen den Aktivitäten können Variablen, auf die im weiteren Prozessverlauf nicht mehr zugegriffen wird, identifiziert werden und deren allokierter Speicher vorzeitig freigegeben werden [21]. Existieren in einer Prozessinstanz beispielsweise nur Datenabhängigkeiten zwischen aufeinanderfolgenden Aktivitäten, müssen lediglich die Ein- und Ausgabedaten der aktuell abzuarbeitenden Aktivität in den zugewiesenen Hauptspeicher der Prozessinstanz geladen werden. Ein Beispiel dafür bildet die Aktivitätenkette *invoke-assign-invoke* in Abbildung 3.1 mit den referenzierten Variablen v_6, v_7 und v_8. Für die erfolgreiche Ausführung dieser Aktivitätenkette müssen jeweils nur die Variablenmengen $\{v_6,v_7\}$ für *invoke*, $\{v_7,v_8\}$ für *assign* und $\{v_8\}$ für *invoke* gleichzeitig im Speicher gehalten werden. Existieren hingegen Datenabhängigkeiten zwischen nicht benachbarten Aktivitäten, müssen diese Variablen entsprechend bis zur endgültigen Verwendung vorgehalten werden. In Abbildung 3.1 trifft dies auf die Variablen v_3 und v_5 zu, welche erst in der Verbundoperation *assign* schlussendlich konsumiert werden. Betrachtet man diese Datenabhängigkeiten im Kontext des eigentlichen, unternehmensrelevanten DIA-Prozesses, so hängen die notwendigen Speicherressourcen stark von der Menge der zurückgegebenen Rechnungen (v_3) und Bestellungen (v_4) der einzelnen Kunden ab und die Variablen v_3, v_5 und v_6 müssen für den Verbund gleichzeitig im Speicher gehalten werden. Eine Verarbeitung beliebig großer Datenmengen (*A1*) und deren Austausch mit entfernten Komponenten (*A2*) können somit nicht gewährleistet werden. Dies disqualifiziert das kontrollfussbasierte Ausführungsmodell für die Umsetzung eines Gesamtkonzeptes zur skalierbaren und effizienten Ausführung datenintensiver Prozessanwendungen.

Zeitliches Ausführungsverhalten

Zur Bewertung der Anforderungen *A3 (Latenz)* und *A4 (Korrelation)*, d.h. der skalierbaren Verarbeitung hoher Eingangsnachrichtenraten sowie der Möglichkeit zur inhärenten und damit effizienten Korrelation von Eingangsnachrichten, wird nachfol-

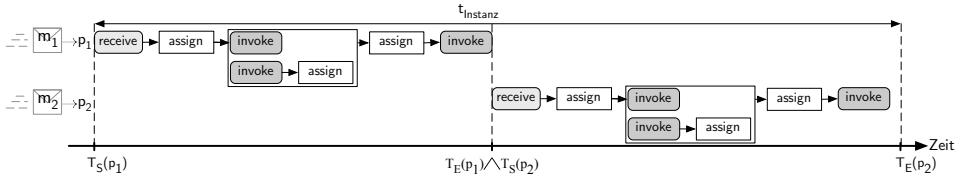


Abbildung 3.2: Serielle instanzbasierte Ausführung von Prozessinstanzen.

gend das zeitliche Ausführungsverhalten diskutiert. Existierende Prozessumgebungen führen Prozessbeschreibungen aus, indem sie den Kontrollfluss und damit die aufeinander folgenden Aktivitäten auf jede eintreffende Nachricht separat anwenden [22, 30, 174, 182, 100]. Für n Nachrichten m führt dies zu n isolierten Prozessinstanzen p_i. Dadurch kann Anforderung *A4* nicht erfüllt werden. Prozessinstanzen p_i eines Prozessplans P_j werden grundsätzlich seriell nacheinander in der Reihenfolge der eintreffenden Eingangsnachrichten m verarbeitet [28], um die Konsistenz der Daten in den involvierten externen Systemen zu gewährleisten. Werden diese Konsistenzanforderungen aufgrund ausschließlich lesender Prozesse nicht benötigt, kann die Einschränkung der seriellen Instanzausführungen aufgehoben werden. Im Rahmen dieser Arbeit wird jedoch davon abgesehen und die striktere Konsistenzanforderung angenommen.

Durch die serielle Ausführung von Prozessinstanzen eines Prozesstyps ist zunächst nur ein CPU-Thread für deren Ausführung notwendig. Wird in einem Prozessplan nur lesend auf die externen Systeme zugegriffen, ist die parallele Verarbeitung der Prozessinstanzen und damit die parallele Verarbeitung der jeweiligen Eingangsnachrichten mithilfe mehrerer CPU-Threads möglich. Das zeitliche Ausführungsverhalten für Prozessinstanzen des Beispielprozesses 2 ist in Abbildung 3.2 anhand von zwei Prozessinstanzen p_1 und p_2 dargestellt. Instanz p_1 startet mit dem Eintreffen der Nachricht m_1 zum Zeitpunkt $T_S(p_1)$. Die Aktivitäten *receive* (`getID`) und *assign* (`transform`) werden schrittweise verarbeitet. Durch die explizite Modellierung der Nebenläufigkeit von beiden darauffolgenden *invoke*-Aktivitäten (`getOrders` und `getInvoices`) werden deren Pfade parallel ausgeführt, bis alle Aktivitäten dieser Pfade abgearbeitet sind und der Kontrollfluss durch die assign-Aktivität (`joinIds`) synchronisiert wird. Die nachfolgenden Aktivitäten werden entsprechend seriell verarbeitet und der Prozess endet beim Endzeitpunkt $T_E(p_1)$. Dieser Zeitpunkt entspricht dem Startzeitpunkt $T_S(p_2)$ der nachfolgenden Prozessinstanz p_2. Somit ergibt sich die Gesamtausführungszeit aller Instanzen mit $t_{Instanz} = \sum_{i=1} n(T_E(p_i) - T_S(p_i))$, wobei durch die Nutzung nur eines Threads pro Prozesstyp im Rahmen von Multi-Core-Architekturen keine hohe Auslastung verfügbarer CPU-Ressourcen realisiert werden kann [23, 29].

Anforderung *A5*, d.h. die native Unterstützung hierarchischer Datenoperationen durch ein internes, hierarchisch aufgebautes Datenmodell, wird durch aktuelle Workflow-Systeme nativ unterstützt, da sie direkt auf den XML-basierten und damit hierarchisch strukturierten SOA-Technologien aufbauen.

3.1.2 Datenmanagementsysteme

Dieser Abschnitt diskutiert und bewertet die Konzepte der bereits vorgestellten Datenintegrations- und Datenstromsysteme aus den Abschnitten DIA und BAM im Kontext der sechs, in Abschnitt 2.5 Anforderungen. Im Gegensatz zum kontrollflussbasierten Verarbeitungsmodell der Workflow-Systeme arbeiten die Systemtypen dieser Klassen mit einem physischen Datenflussgraphen, der die Transformations- und Analyseschritte (DIA) bzw. die Anfrage an den Datenstrom (BAM) repräsentiert.

Beide Systemtypen verarbeiten die ihnen gegebene Datenmenge D mithilfe von Stromsemantik im Datenflussgraphen. Die Verarbeitung erfolgt in den Knoten des Graphen auf der Granularität von Datentupeln $d_i \in D$ als flache attributbasierte Datenstruktur [98, 119]. Durch die grundlegende Unabhängigkeit der Datentupel in einer Datenmenge D verarbeiten aktuelle DIA- und BAM-Systeme diese Datentupel mithilfe des nebenläufigen *Pipes-and-Filters*-Ausführungsmodells [3, 115]. Operatoren (*Filters*) sind darin durch Warteschlagen (*Pipes*) miteinander verbunden und verarbeiten Datentupel sobald sie als Eingangsdaten beim Operator vorliegen.

Da die Prozessinstanzen von DIA-Systemen Datenmengen D in beliebiger Größe verarbeiten müssen, arbeiten diese Systeme mit dem Konzept der Partitionierung bzw. Aufteilung der zu verarbeitenden Daten in Teilstücke. Dabei wird D auf Basis ihrer feingranularsten Datenelemente d_i unterteilt und verarbeitet. Abbildung 3.3 zeigt die Aufteilung anhand der Datenbankrelation für Kundenrechnungen aus Beispielprozess *Top n Kunden* (Abbildung 2.11), in welcher in dieser Darstellung die Rechnungsnummer (`invID`), die Kundennummer (`cID`), der Rechnungswert (`value`) sowie die Währung der Rechnung (`CUR`) gespeichert sind. Dabei werden die Tupel d_i der Relation D nacheinander aus der Relation geladen und in den Aktivitäten einzeln strombasiert verarbeitet. Dieses Konzept der DIA-Systeme ermöglicht die Unterstützung von Anforderung *A1*. Da jedoch im Bereich der BAM-Systeme D nur kleine Datenmengen umfasst bzw. D sogar meist nur ein Datenelement d_1 mit $D = \{d_1\}$ enthält, wird in BAM-Systemen keine explizite Partitionierung und damit auch nicht Anforderung *A1* unterstützt. Die proprietäre Kommunikation mit externen Datenbanksystemen wird bei DIA-Systemen, analog der Verarbeitung von D, strombasiert realisiert. Dabei werden bei der Nutzung von Punkt-zu-Punkt-Verbindungen zu Datenbanksystemen die einzelnen Datenelemente aller zu verarbeitenden Daten innerhalb eines Kommunikationsstroms in den Integrations- bzw. Analyseprozess gezogen [127]. Somit wird Anforderung *A2* lediglich mithilfe proprietärer Kommunikation und nicht mithilfe von Nachrichtenaustausch unterstützt.

Die Ausführung von DIA-Prozessen erfolgt jedoch analog zu den Workflow-Prozessen weiterhin instanzbasiert, d.h. eine Eingangsnachricht m_i instanziiert jeweils eine Prozessinstanz p_i mit $m_i \rightarrow p_i$, die D_i isoliert von anderen Instanzen und somit isoliert von anderen D verarbeitet. Instanzen des gleichen Prozesstyps P_j werden ebenfalls grundsätzlich seriell ausgeführt. Diese beiden Eigenschaften verhindern zum einen

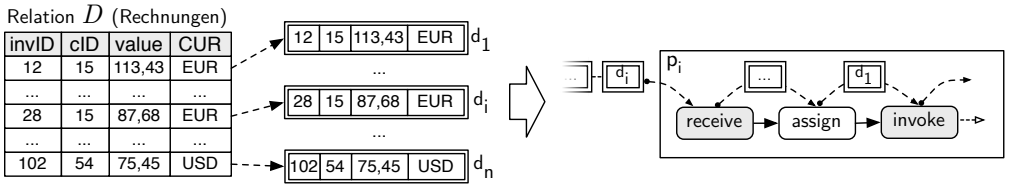


Abbildung 3.3: Partitionierte Verarbeitung von D im Beispielprozess *Top n Kunden*.

eine native, datenübergreifende Korrelation unterschiedlicher D und führen zum anderen zu einer ineffizienten Eingangsnachrichtenverarbeitung. Anforderungen *A4 (Korrelation)* und *A3 (Latenz)* werden folglich nicht unterstützt.

BAM-Systeme begegnen diesen beiden Anforderungen mit dem Konzept der stehenden Anfragen (engl. *standing queries*), welches für alle Eingangsnachrichten eines Prozesstyps P_j nur initial *eine gemeinsame* Prozessinstanz $p_{j,1}$ instanziiert und alle eintreffenden Nachrichten in der Reihenfolge ihrer Ankunft verarbeitet. Im Gegensatz zu DIA-Prozessen beinhaltet eine Eingangsnachricht m_i bei BAM-Prozessen jeweils nur ein Verarbeitungselement d_i (vgl. Abschnitt 2.4.3) und definiert damit die Verarbeitungsgranularität auf Ebene einer Eingangsnachricht m_i. Alle Eingangsnachrichten m_i werden jedoch nacheinander in *einer* Prozessinstanz verarbeitet, wodurch sich der interne Zustand eines Operators o_k einer stehenden Prozessinstanz $p_{j,1}$ inhärent über Menge der bereits verarbeiteten m_i erstreckt. Dieses Konzept realisiert eine effiziente Korrelationsmöglichkeit über alle Eingangsnachrichten m_i in den Operatoren von $p_{j,1}$ und erfüllt damit Anforderung *A4 (Korrelation)*.

Abbildung 3.4 zeigt abschließend die zeitliche Abfolge einer pipeline- bzw. strombasierten Instanzausführung von BAM-Systemen analog zur instanzbasierten Ausführung in Workflow-Systemen (vgl. Abbildung 3.2) am Beispielprozess *Top n Kunden*. Die Eingabedaten und damit die Verarbeitungsgranularität bilden die einzelnen Eingangsnachrichten m_i, welche in der Reihenfolge ihrer Ankunft an eine stehende Prozessinstanz p_j übergeben werden. Trifft eine Nachricht m_1 im System

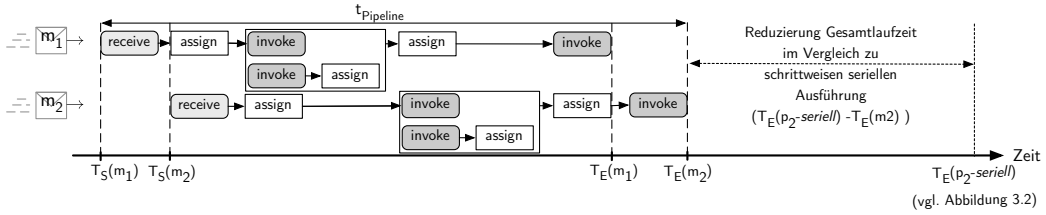


Abbildung 3.4: Pipeline-basierte Verarbeitung von Nachrichten im Beispielprozess *Top n Kunden*.

ein, wird ihr Inhalt im *receive*-Operator verarbeitet und an den nachfolgenden *assign*-Operator weitergegeben. Durch die Nebenläufigkeit aller Operatoren kann der *receive*-Operator bereits Eingangsnachricht m_2 abarbeiten und anschließend, falls m_1 den *assign*-Operator bereits wieder verlassen hat, m_2 an den *assign*-Operator weitergeben. Somit ergibt sich die dargestellte zeitliche Abfolge der Aktivitäten für jede eintreffende Eingangsnachricht m_i [22, 29] und im Vergleich zur seriellen Ausführung (vgl. Abbildung 3.2) reduziert sich die Gesamtverarbeitungszeit für beide Eingangsnachrichten [26, 29].

Das interne Datenformat basiert auf den datenbankspezifischen Datentupeln und deren flacher, attributbasierter Struktur. Dementsprechend sind XML-Daten und deren Operationen nicht nativ in das Verarbeitungsmodell integriert. Anforderung *A5 (Datenmodell)* wird somit nicht erfüllt.

3.1.3 Hybride Ansätze

Dieser Abschnitt stellt Ansätze vor, welche die kontroll- und datenflussbasierten Konzepte erweitern und Teilaspekte der aufgestellten Anforderungen adressieren:

[S-JOPERA] [23, 24] untersuchen die Laufzeitzustände der Aktivitäten traditioneller Workflow-Systeme und erweitern diese um Zustände für eine pipeline-basierte Ausführung auf Basis von Petrinetzsemantik und Aktualitätsmarkern. Der Fokus dieses Ansatzes liegt jedoch weder auf dem Einsatz in verteilten Dienstumgebungen noch werden die Probleme der Verarbeitung hoher Datenvolumen und die damit verbundene Kommunikation mit externen Systemen betrachtet. Des Weiteren verarbeiten die Aktivitäten des vorgestellten Ansatzes die einzelnen Dateneinheiten isoliert voneinander. Eine Korrelation der einzelnen Dateneinheiten und damit Anforderung *A4 (Korrelation)* werden nicht unterstützt.

[V-WFPE] [29, 30] adressieren den Durchsatz seriell ausgeführter Workflow-Instanzen und überführen die schrittweise kontrollflussbasierte Verarbeitung in eine pipelinebasierte Verarbeitung. Dabei wird jede Aktivität durch einen physischen Operator repräsentiert. Die einzelnen Operatoren werden durch Warteschlangen miteinander zu einem Datenflussgraphen verbunden. Die Verarbeitung der Eingangsnachrichten mit ihren dazugehörigen Prozesskontexten C_i erfolgt nebenläufig entlang des Operatorgraphen. Somit verarbeitet eine Instanz p_i alle Eingangsnachrichten für einen Prozessplan P_i. Die Überführung der als Kontrollfluss modellierten Workflow-Prozesse wird mithilfe kostenbasierter Umschreiberegeln transparent realisiert. Ähnlich zu [23] werden die Nachrichtenkontexte isoliert verarbeitet ($\neg A4$) und keine Optimierungen bzgl. der Nachrichtengrößen ($\neg A1$) und skalierbarer Kommunikation ($\neg A2$) vorgenommen.

[XPEDIA] In [21] wird das Problem der flachen internen Datenmodelle bei Datenintegrationsumgebungen wie beispielsweise ETL-Werkzeugen adressiert. Es wird ein Rahmenwerk präsentiert, welches ein natives Verarbeitungsmodell für XML-

Daten in das flache Datenmodell integriert. Damit wird die direkte und effiziente Anwendung der ETL-Techniken auf XML-Daten möglich. Des Weiteren erlaubt das Rahmenwerk, die XML-Verarbeitungen in die Quell- bzw. Zieldatenbank zu verlagern. Partitionierungstechniken innerhalb eines XML-Dokumentes eines Tupels ermöglichen die parallele und effiziente Verarbeitung dieser Dokumente auf moderner Multi-Core-Hardware. Nachteile des Ansatzes liegen in seinem starken Fokus auf ETL-Prozessen und dem vormals batchbasierten Ausführungsrythmus sowie der instanzbasierten Ausführung einzelner Prozesse. Zudem basiert die grundsätzliche Verarbeitungssemantik weiterhin auf dem datenbankzentrischen Tupelkonzept und der damit verbundenen flachen Datenstruktur (Anforderung *A5*). Schlussendlich wird die Kommunikation mit entfernten, dienstbasierten Komponenten und deren Integration in den Prozessfluss (Anforderung *A2*) adressiert.

Die Optimierung datenintensiver Geschäftsprozesse wird von den Autoren der Arbeiten in [109], [156, 157] und [71, 72] untersucht und im Folgenden beschrieben.

[BPEL4SQL] Während [109] die Erweiterung der Prozessbeschreibungssprache BPEL um dedizierte Datenbankaktivitäten in Form von SQL-Anweisungen vorschlägt, beschreiben [156, 157] die Optimierung solcher Prozesse. Das Optimierungsziel besteht dabei darin, möglichst viele Datenoperationen direkt in der Datenbank zu verarbeiten, ohne sie in den Prozess zu laden. Die Daten werden deshalb zwischen den Aktivitäten nur als Referenzen übergeben. Der Nachteil dieses Ansatzes liegt in der engen Kopplung zwischen Prozess und Datenbankendpunkt. Die lose Kopplung über Dienste wird umgangen.

[Data Greybox Services] In [71] bzw. [72] wird hingegen eine dienstbasierte Architektur für datenintensive Analyseprozesse auf Basis der Web Service Spezifikationen und BPEL präsentiert. Die Grundidee der Autoren besteht in einem dezentralen Datenfluss von Massendaten direkt zwischen den entsprechenden Diensten. Dazu werden auf der kontrollflussorientierten Modellierungsebene explizite Datenflüsse zwischen den Aktivitäten (und damit zwischen den Diensten) eines Prozesses definiert. Diese werden im Anschluss zur Laufzeit über effiziente Datentransfertechnologien zwischen den Diensten transformiert und übertragen. Die notwendige zentrale XML-Kommunikation entfällt. Die Nachteile dieses Ansatzes bestehen in der gestiegenen Anzahl an involvierten Komponenten und deren gegenseitiger Kommunikation. Zudem wird die serielle und instanzbasierte Ausführungssemantik auf Prozessebene beibehalten. Weitere Arbeiten zu Ansätzen des dezentralen Datenflusses bilden [103, 104] und [117], wobei diese auf proprietären Technologien und Kommunikationsmustern aufbauen.

3.1.4 Zusammenfassung

Tabelle 3.1 fasst die diskutierten Verarbeitungsmodelle sowie die hybriden Ansätze der unterschiedlichen Anwendungsklassen im Kontext der Anforderungen *A1 (Verarbeitung)*, *A2 (Datenaustausch*, *A3 (Latenz)*, *A4 (Korrelation)*, *A5 (Datenmodell)*

Ansatz/Anforderungen	A1	A2	A3	A4	A5	A6
Workflow-Systeme	nein	nein	nein	ja[1]	ja	nein
DIA	ja	ja	nein	nein	nein	nein
BAM	nein	nein	ja	ja	ja[2]	nein
S-JOPERA	nein	nein	ja	nein	ja	nein
V-WFPE	nein	nein	ja	nein	ja	nein
XPEDIA	ja	ja	nein	nein	ja	nein
BPEL4SQL	ja[3]	ja	nein	nein	ja	nein
Data Greybox Services	ja[3]	ja	nein	nein	ja	nein

[1] vereinzelt durch Korrelationsaktivitäten möglich [178]
[2] Hierarchisches Datenmodell bei XQuery-Stromsystemen [32] bzw. [57]
[3] über die Verarbeitung reiner Datenreferenzen zwischen Aktivitäten

Tabelle 3.1: Zusammenfassung der vorgestellten Prozessverarbeitungsmodelle.

und *A6 (Dienstsemantik)* zusammen (vgl. auch Abschnitt 2.5, Seite 37). Dabei erwies sich die kontrollflussbasierte Verarbeitung von Aktivitäten als kritisch für eine skalierbare Verarbeitung großer Datenmengen in einer Prozessinstanz. Außerdem bedingt eine kontrollflussbasierte Verarbeitung die ganzheitliche Kommunikation mit beteiligten Diensten und ist daher als kritisch einzuschätzen. Die weiteren hier diskutierten Arbeiten verbessern die Eigenschaften eines ganzheitlichen Ausführungssystems nur bedingt bzw. in Teilaspekten.

Jedoch lässt sich aus den vorgestellten Konzepten und deren diskutierter Eignung für die sechs formulierten Anforderungen erkennen, dass die Anwendung von Stromsemantik mithilfe des Pipes-and-Filters-Verarbeitungsmodells in Verbindung mit der Partitionierung der im Prozess zu verarbeitenden Daten die Anforderungen *A1* bis *A4* bereits erfüllt. Die Integration effizienter Dienstkommunikation (Anforderung *A6*), welche wiederum die Anforderungen *A2* bis *A4* erfüllt, fehlt dabei jedoch bei allen diskutierten Ansätzen.

3.2 Prozess- und Dienstkommunikation

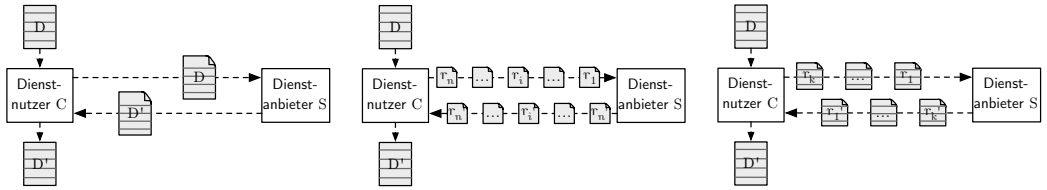
In diesem Abschnitt werden Ansätze zur Kommunikation zwischen den Anwendungsprozessen und den darin involvierten externen Komponenten diskutiert. Im Fokus steht dabei die XML-basierte Nachrichtenkommunikation zum Austausch strukturierter Informationen, da sie zum einen durch die Web Service-Spezifikationen standardisiert ist und eine lose und damit skalierbare Kopplung der verteilten Komponenten erlaubt. Zum anderen bildet sie die Grundlage der zentralen Anwendungsklasse der *dienstbasierten Prozesse*.

3.2.1 Nachrichtenbasierte Aufrufmethoden

Im Kontext nachrichtenbasierter Kommunikation lässt sich die Übertragung und Verarbeitung großer Datenmengen zwischen Dienstnutzer und Dienstanbieter in die drei Aufrufmodelle *Gesamtnachrichtentransfer*, *Einzelnachrichtentransfer* und *Teilnachrichtentransfer* unterteilen. Für die Bewertung dieser Ansätze im Kontext der aufgestellten Anforderungen sei ein Datensatz $D = (d_1, \ldots, d_i, \ldots, d_n)$ mit einer beliebigen Anzahl gleich strukturierter Datenelemente d_i gegeben. Dieser Datensatz D soll durch einen Dienstnutzer C von einer Dienstkomponente S verarbeitet werden. Im Allgemeinen hat ein Dienstnutzer dazu die Möglichkeit, D in k Dienstaufrufen (engl. *requests*) r_j an den Dienst S zu übertragen. Die Anzahl R aller notwendigen Anfragen wird mit $R = (r_1, \ldots, r_j, \ldots, r_k)$ beschrieben. Durch die Veränderung von k lassen sich die nachfolgenden Aufrufmodelle abbilden.

Gesamtnachrichtentransfer (GNT) Die direkte Abbildung der Verarbeitung eines Datensatzes D durch einen Dienst S auf eine dienstorientierte Architektur bildet der *Gesamtnachrichtentransfer* (engl. *bulk message transfer*, Abbildung 3.5(a)). Dabei wird D mit der Größe $s_D = \sum_1^n s_{d_i}$ in *einem* Dienstaufruf r_1 verpackt und an den Dienst gesandt. Somit ergibt sich die Anzahl der notwendigen Dienstaufrufe mit $|R| = 1$. Die Größe s_R dieser einzelnen Nachricht ergibt sich aus $s_R = s_{Header} + s_{Body}$ in Analogie zur SOAP-Spezifikation wobei s_{Header} die Größe des Nachrichtenkopfes und s_{Body} die Größe des Nachrichtenkörpers beschreibt. Die Größe des Nachrichtenkörpers s_{Body} kann dabei auch als Nutzdatengröße mit $s_{Body} = s_D = \sum_1^n s_{d_i}$ bezeichnet werden. Die Gesamtnachrichtengröße s_R ist somit stark von der Größe jedes einzelnen Datenelements s_{d_i} sowie deren Anzahl n in der Nachricht abhängig. Im Gegensatz dazu wird die Größe des Nachrichtenkopfes s_{header} als konstant und klein angenommen. Nachdem die gesamte Anfragenachricht r_1 und damit der gesamte Datensatz D beim Dienst S empfangen wurde, wird der Datensatz komplett verarbeitet. In Abhängigkeit der bereitgestellten Funktion wird dabei von S gegebenenfalls eine Antwortnachricht r'_1 generiert und an den Aufrufenden zurückgesandt.

Diese Transfermethode spiegelt das ursprüngliche Anfrage-Antwort-Paradigma eines Dienstaufrufes mit einem Datensatz D wider und korreliert somit direkt mit der schrittweisen Verarbeitung von Aktivitäten bei Workflow-Systemen. Jedoch setzt dieses Aufrufmodell voraus, dass D bei der Dienstkomponente in einem Schritt verarbeitet werden kann. XML bzw. dessen strukturierte Baumdarstellung im Hauptspeicher kann bis zu einem Faktor von zehn höher als die textuelle Repräsentation ausfallen [51, 108, 152]. Deshalb ist die zu verarbeitende Gesamtdatenmenge D bei diesem Aufrufmodell durch Speicherbegrenzungen im Zusammenhang mit parallelen Dienstinstanzen und verschiedenen Diensttypen auf einem Serverknoten stark eingeschränkt. Bewertet man dieses Aufrufmodell im Kontext der Anforderung *A2 (Datenaustausch)*, so wird die Übertragung von D in beliebiger Größe nicht unterstützt. Des Weiteren muss ein Dienstnutzer auf die vollständige Verarbeitung von D



(a) *Gesamtnachrichtentransfer* (GNT). (b) *Einzelnachrichtentransfer* (ENT). (c) *Teilnachrichtentransfer* (TNT).

Abbildung 3.5: Nachrichtenbasierte Aufrufmethoden.

bei der Dienstkomponente warten, ohne dass mögliche Teilergebnisse bereits während der Verarbeitung an den Nutzer zurück übertragen werden können. Dadurch erhöht sich die Latenz der einzelnen Datenelemente bei großen Datenmengen signifikant. Anforderung *A3 (Latenz)* wird folglich nicht erfüllt. Lediglich Anforderung *A4 (Korrelation)* wird durch die gesamtheitliche Verarbeitung Rechnung getragen.

Einzelnachrichtentransfer (ENT) Das entgegengesetzte Modell zur Verarbeitung eines Datensatzes D durch einen Dienst S bildet der *Einzelnachrichtentransfer* (engl. *single-item message transfer*, Abbildung 3.5(b)). Dabei wird jedes Datenelement $d_i \in D$ in einen separaten Dienstaufruf r_i verpackt, sodass $|R| = |D|$. Nachdem ein Element d_i durch S verarbeitet und dessen Antwort r'_i empfangen wurde, wird eine neue Anfrage mit dem Element d_{i+1} an S gesandt. Die Nachrichtengröße berechnet sich mit $s_{r_i} = s_{Header} + s_{Body}$ mit $s_{Body} = s_{d_i}$ für jedes Element d_i. Daraus ergibt sich ein Gesamtvolumentransfer von $s_R = \sum_1^n (s_{Header} + s_{d_i})$.

Durch die Partitionierung der Datenmenge D in Verarbeitungseinheiten wird die maximal zu verarbeitende Nachrichtengröße durch s_{d_i} definiert und eine Verarbeitung beliebig großer Datenmengen ermöglicht (*A2 (Datenaustausch)*). Dadurch werden zudem, anders als beim *Gesamtnachrichtentransfer*, die Übermittlung von Zwischenergebnissen der Verarbeitung von D unterstützt. Obwohl die Latenz einzelner Nachrichten bzw. einzelner Datenelemente gegenüber dem *Gesamtnachrichtentransfer* stark reduziert wird und somit Anforderung *A3 (Latenz)* erfüllt zu sein scheint, bedingen der Mehraufwand an Nachrichtengenerierung und deren Versand in Höhe der Anzahl n der Datenelemente in D einen höhere CPU-Last sowie ein höheres Transfervolumen. Aufgrund dessen steigen die Latenzen einzelner Nachrichten bzw. Datenelemente wieder stark an und Anforderung *A3* bleibt unerfüllt (siehe auch Experimente in Abschnitt 4.5). Des Weiteren wird bei diesem Verarbeitungsmodell die Unabhängigkeit der Datenelemente angenommen, da jedes Element in D einen separaten Dienstaufruf impliziert. Wird eine Korrelation der Datenelemente beim Dienstanbieter benötigt, so muss ein Zustand verwaltet werden, welcher die einzelnen Dienstaufrufe miteinander in Verbindung bringen kann. Anforderung *A4 (Korrelation)* wird dadurch nicht erfüllt.

Teilnachrichtentransfer (TNT) Das dritte Aufrufmodell, der *Teilnachrichtentransfer* (engl. *chunk-based message transfer*, Abbildung 3.5(c)), stellt eine Generalisierung der vorherigen Modelle des *Gesamtnachrichtentransfers* und des *Einzelnachrichtentransfers* dar und wurde erstmals in [146] formal definiert. Die grundlegende Idee ist es dabei, die zu übertragenden Datenelemente d_i des Datensatzes D auf k unterschiedliche Dienstaufrufe r_j zu verteilen, sodass $|R| = k$ mit $1 \leq k \leq |D|$, und damit die durchschnittliche Antwortzeit zu minimieren. Gleichverteilung angenommen, so enthält $r_j = \frac{|D|}{k}$ Datenelemente aus D. Das Aufrufmodell *Gesamtnachrichtentransfer* entspricht somit dem Modell *Teilnachrichtentransfer* mit $k = 1$. Analog dazu entspricht das Aufrufmodell *Einzelnachrichtentransfer* dem Modell *Teilnachrichtentransfer* mit $k = |D|$. Unter der Annahme der Unabhängigkeit zwischen den Datenelementen, können Teilnachrichten beliebiger Größe als Nachrichten versandt werden. Die Anzahl der Elemente in einer Teilnachricht bestimmt jedoch maßgeblich die Antwortzeit des Dienstes [62] bzw. dessen generelle Möglichkeiten zur Verarbeitung. Es wurden daher Ansätze beschrieben [62, 146], welche die optimale Nachrichtengröße reaktiv anhand der Antwortzeitentwicklung bestimmen. Dadurch können sich verändernde Arbeitslasten beim Dienst erkannt und die Nachrichtengröße entsprechend angepasst werden. Es hat sich jedoch gezeigt, das ein reaktiver Ansatz bei stark schwankender Arbeitslast eines Dienstes oder dessen Laufzeitumgebung schlechte Antwortzeiten generiert [63].

Das Modell des *Teilnachrichtentransfers*, als Generalisierung der beiden vorherigen Modelle, ähnelt in seinen Eigenschaften eher dem *Einzelnachrichtentransfer*. Durch die Partitionierung der Datenmenge D auf mehrere Anfragen r_j erlaubt auch dieses Modell die Verarbeitung beliebig vieler Datenelemente in D und erfüllt somit Anforderung *A2*. Auch Zwischenergebnisse werden dadurch unterstützt. Aufgrund der Bündelung von Datenelementen in Nachrichten reduziert es den Mehraufwand für Nachrichtengenerierung und -versand im Gegensatz zum *Einzelnachrichtentransfer* erheblich (siehe auch Experimente in Abschnitt 4.5). Die richtige Wahl des Parameters k ermöglicht somit auch die Erfüllung der Anforderung *A3 (Latenz)*. Eine inhärente und damit effiziente Möglichkeit zur Korrelation von Datenelementen aus D besteht zunächst nur zwischen Elementen einer Nachricht r_i. Zur Korrelation aller Elemente in D zueinander, beispielsweise zur Berechnung einer Summe, benötigt der Dienstanbieter die zusätzliche Verwaltung des Zustandes über Einzelnachrichten hinweg. Somit bleibt auch hier Anforderung *A4 (Korrelation)* unerfüllt.

3.2.2 Optimierungen der Nachrichtenkommunikation

Die XML-basierte Nachrichtenkommunikation in dienstorientierten Umgebungen zeichnet sich im Vergleich zu anderen Datenformaten sowohl durch einen höheren CPU- und Speichermehraufwand als auch durch ein höheres zu übertragendes Datenvolumen aus [45, 46, 71, 108]. Aus diesem Grund wurden zahlreiche Ansätze entwickelt, welche sich mit der effizienten XML-basierten Nachrichtenkommunikation befassen. Sie lassen sich in zwei Klassen unterteilen.

Die erste Klasse enthält Arbeiten zur Reduzierung des Mehraufwandes bei der Serialisierung, d.h. der Transformation von Hauptspeicherstrukturen in XML-Text, sowie der Deserialisierung, d.h. der Rücktransformation von XML-Text in die Hauptspeicherstrukturen der Ausführungsumgebung, durch die Anwendung von Puffermechanismen [4, 5, 50, 129, 148, 164]. Die Kernannahme der Ansätze dieser Klasse ist dabei, dass die Grundstruktur aufeinanderfolgender Dienstnachrichten eines Dienstes konstant bleibt und nicht die gesamte Nachricht notwendigerweise erneut transformiert werden muss. Deshalb wird bei diesen Ansätzen ein konstantes Skelett der Nachricht im Speicher vorgehalten und nur deren dynamische Teile, d.h. Werte der spezifischen Nachricht, werden der Transformation unterzogen.

Die zweite Klasse von Ansätzen optimiert den Netzwerktransfer durch die Nutzung von Kompressionsalgorithmen oder kompakteren Nachrichtenformaten. Bei den Kompressionsalgorithmen werden dabei allgemeine, nicht strukturerhaltende Algorithmen [61, 102, 169] und strukturerhaltende Algorithmen [110, 123, 163] unterschieden. Werden strukturerhaltende Algorithmen angewandt, ist ein direktes Arbeiten auf den komprimierten Daten möglich und eine vorherige Dekompression wird obsolet. Kompaktere Nachrichtenformate bauen hingegen auf dem XML zugrundeliegenden *XML Information Set* [202] auf, umgehen jedoch die expansive textbasierte Struktur von XML [207, 122, 167].

Alle hier diskutierten Optimierungen adressieren die effiziente und schnelle XML-Kommunikation nur, indem sie den Mehraufwand von Nachrichtenerstellung, Nachrichtentransfer und Nachrichtenempfang teilweise signifikant reduzieren. Dies führt zwar dazu, dass sich diese Optimierungen positiv auf die Anforderung *A3 (Latenz)* der Aufrufmodelle *Einzelnachrichtentransfer* und *Teilnachrichtentransfer* auswirken, weitere positive Effekte bleiben jedoch aufgrund methodischer Probleme der Aufrufmodelle *Gesamtnachrichtentransfer*, *Einzelnachrichtentransfer* und *Teilnachrichtentransfer* aus.

3.2.3 Zusammenfassung

Tabelle 3.2 fasst die Bewertung der Aufrufmodelle in Verbindung mit den vorgestellten Nachrichtenoptimierungen und den aufgestellten Anforderungen zusammen. Sie zeigt, dass keines der hier diskutierten Aufrufmodelle alle notwendigen Anforderungen *A2 (Datenaustausch)*, *A3 (Latenz)* und *A4 (Korrelation)* unterstützt. Es lässt sich ableiten, dass durch die Partitionierung der Daten, und damit die Verarbeitung von Teilmengen, Anforderung *A2* erfüllt wird. Werden zusätzlich der Mehraufwand für diese Partitionierung und der Mehraufwand der Kommunikation minimal gehalten, so ermöglicht die partitionierte Verarbeitung auch die Erfüllung von Anforderung *A3*. Anforderung *A4*, d.h. die inhärente Möglichkeit zur Korrelation von Datenelementen, kann durch die Modelle *Einzelnachrichtentransfer* und *Teilnachrichtentransfer* nicht ohne eine zusätzliche Verwaltung des Zustandes realisiert werden. Hingegen ermöglicht der gemeinsame Kontext der Datenelemente im Aufruf-

Modell/Anforderungen	A2	A3	A4	Zwischen-ergebnisse
GNT	nein	nein	ja	nein
ENT	ja	ja[1]	nein	ja
TNT	ja	ja[2]	nein	ja

[1] durch *Optimierung der Nachrichtenkommunikation*
[2] durch [1] oder, wenn Parameter k in Abhängigkeit der Dienstarbeitslast optimal gesetzt

Tabelle 3.2: Zusammenfassung der Aufrufmodelle bei der Dienstkommunikation.

modell *Gesamtnachrichtentransfer* eine inhärente und damit effiziente Korrelation und somit die Unterstützung von Anforderung *A4*.

3.3 Zielarchitektur

Nachdem die Systeme der hier betrachteten Anwendungsklassen und deren Verarbeitungs- und Kommunikationskonzepte im Kontext der in Abschnitt 2.5 formulierten Anforderungen diskutiert wurden, zeigt dieser Abschnitt nochmals die Anforderungen $A1$ - $A6$ und mögliche Konzepte für deren Erfüllung auf. Damit wird ein Überblick über die zu entwickelnde Zielarchitektur und die zu nutzenden Konzepte gegeben. Die Architektur unterstützt durch die definierten Anforderungen die Integration von Anwendungsteilen der Klassen Datenintegration und Ereignismonitoring in die Klasse der SOA-basierten Prozesse. Diese Prozesse zeichnen sich durch die Verteilung von Komponenten sowie deren nachrichtenbasierte Kommunikation aus.

Dienstaspekt

Zunächst kann zusammengefasst werden, dass bestehende Ansätze zur Optimierung der Dienstkommunikation lediglich die reine Kommunikation adressieren, während die eigentliche Verarbeitung beim Dienst nicht betrachtet wird. Somit tragen diese Ansätze nur bedingt zur Erfüllung der hier formulierten Anforderungen bei. Weiterhin hat sich auf Ebene der nachrichtenbasierten Dienstkommunikation gezeigt, dass die partitionierte Kommunikation, und damit die partitionierte Verarbeitung einer Datenmenge D bei Dienstkomponenten, eine skalierbare Kommunikation ermöglicht und somit Anforderung *A2 (Datenaustausch)* erfüllt. Bisherige partitionierende Aufrufmodelle müssen den dabei auftretenden Verlust des gemeinsamen Kontextes aller d_i beim Dienst durch einen künstlichen Zustand kompensieren. Einen inhärenten gemeinsamen Kontext bietet hingegen nur der *Gesamtnachrichtentransfer*, da dieser alle zusammengehörenden Datenelemente $d_i \in D$ in einer Nachricht und damit in einem Kontext zusammenfasst und von einer Dienstinstanz verarbeiten lässt (entspricht Anforderung *A4 (Korrelation)*.

Anforderung	Umsetzung auf Dienstebene
A2 (*Datenaustausch*)	Datenpartitionierung und strombasierte Verarbeitung von Nachrichteninhalten bei Dienstinstanzen.
A3 (*Latenz*)	Feingranulare Datenpartitionierung ohne Mehraufwand für zusätzliche Nachrichtengenerierung.
A4 (*Korrelation*)	Abstraktion von Datenelementen und Nachrichteninhalten zu allgemeinen Stromobjekten und Instanziierung eines gemeinsamen physischen Nachrichtenkontextes auf Basis *stehender* Dienstinstanzen.

Tabelle 3.3: Implikationen auf Dienstebene.

Implikation: *Für die Unterstützung datenintensiver und skalierbarer nachrichtenbasierter Dienstkommunikation muss ein neues Aufrufmodell entwickelt werden, welches den strombasierten Ansatz der partitionierenden Aufrufmodelle* Einzelnachrichtentransfer *und* Teilnachrichtentransfer *mit dem Konzept eines einheitlichen physischen Nachrichtenkontextes vereint. Im Einzelnen werden die Anforderungen durch die Konzepte in der rechten Spalte der Tabelle 3.3 erfüllt. Das dazugehörige Gesamtkonzept der Dienstkommunikation wird in Kapitel 4 detailliert beschrieben.*

Prozessaspekt

Auf Ebene der Prozessausführung wurden die bestehenden Verarbeitungskonzepte analysiert, ihr Beitrag zu den Anforderungen *A1 bis A6* diskutiert und in Tabelle 3.1 (Seite 47) zusammengefasst. Dabei zeigte sich, dass die kontrollflussbasierte Ausführung mit ihrer schrittweisen und atomaren Verarbeitung der Aktivitäten kritisch für große Datenmengen und damit kritisch für die Anforderungen *A1* und *A2* ist. Die Partitionierung und die strombasierte Verarbeitung der vom Prozess zu verarbeitenden Datenmenge unterstützen hingegen diese Anforderungen effizient und skalierbar. Die diskutierten, bereits bestehenden Ansätze adressieren nur Teilaspekte der hier benötigten Anforderungen. So wird beispielsweise in [72] und [156] durch die Nutzung von Datenreferenzen zwischen den Aktivitäten das Laden der eigentlichen Daten in den Prozess vermieden. Sind jedoch Entscheidungen anhand der entfernt referenzierten Daten zu treffen, muss dies entweder vom entfernten Datencontainer unterstützt oder die Daten traditionell in den Prozess geladen und dort verarbeitet werden.

Implikation: *Für die Unterstützung datenintensiver Anwendungsszenarien und damit einer skalierbaren Prozessausführung ist ein ganzheitliches Ausführungskonzept zu entwickeln, welches die Ansätze der Datenmanagementkonzepte mit deren Datenpartitionierung, pipeline-basierter Ausführung und stehenden Prozessinstanzen mit den Eigenschaften der dienstbasierten Prozessausführung und Kommunikation vereint.*

Anforderung	Umsetzung auf Prozessebene
A1 (*Verarbeitung*)	Partitionierung der vom Prozess zu verarbeitenden Daten sowie deren pipelinebasierte Abarbeitung entlang der Aktivitätenkette.
A2 (*Datenaustausch*)	Integration der Datenpartitionierung der Prozessinstanz in die Kommunikation zwischen Instanz und Dienstkomponente.
A3 (*Latenz*)	Reduzierung der Verarbeitungslatenz einzelner Prozessinstanzen durch pipelinebasierte Verarbeitung der Eingangsnachrichten sowie durch die Realisierung des Konzeptes *stehender* Prozessinstanzen.
A4 (*Korrelation*)	Abstraktion von Datenelementen einer Nachricht und ganzen Nachrichteninhalten zu allgemeinen Stromobjekten und Instanziierung eines gemeinsamen physischen Prozesskontextes auf Basis *stehender* Prozessinstanzen.
A5 (*Datenmodell*)	Nutzung eines XML-basierten Datenmodells SOA-basierter Workflow-Systeme auf Basis der *XML InfoSet*-Semantik [202].
A6 (*Dienstsemantik*)	Integration der strombasierten und traditionellen Kommunikation in die Prozessausführung.

Tabelle 3.4: Implikationen für Prozessebene.

Die Kommunikation mit orthogonalen Systemen der diskutierten Anwendungsklassen gilt es dabei zu vermeiden. Im Einzelnen werden die Anforderungen auf Prozessebene durch die Konzepte der rechten Spalte in Tabelle 3.4 erfüllt. Das dazugehörige Gesamtkonzept der Prozessausführung wird in Kapitel 5 detailliert beschrieben.

Infrastrukturaspekt

Da derzeitige dienstbasierte Umgebungen die Nutzung von XML zur Kommunikation implizieren und diese Kommunikationsform weiterhin die größten Performanceprobleme verursacht [71, 108], wird das Konzept der skalierbaren Prozessausführung auf die Infrastrukturebene erweitert. Die Grundidee besteht darin, durch eine gemeinsame Platzierung von Prozessen und Diensten auf lokalen Serverknoten die XML-basierte Nachrichtenkommunikation vollständig zu vermeiden sowie den damit einhergehenden signifikanten Mehraufwand zu reduzieren und die Bandbreite des Netzwerkes für andere Aufgaben freizuhalten.

Implikation: *Auf Ebene der Infrastruktur wird deshalb ein Advisor-Ansatz beschrieben, welcher auf Grundlage der Prozessarbeitslast und der Kommunikationskosten eine skalierbare Konfiguration von Prozess- und Dienstplatzierung auf gegebenen Serverknoten vorschlägt. Das dazugehörige Gesamtkonzept der arbeitslastbasierten*

Verteilung von Prozess- und Dienstkomponenten wird in Kapitel 6 detailliert beschrieben.

Abbildung 3.6 visualisiert auf Basis von Abbildung 2.1 zusammenfassend die Zielarchitektur auf Anwendungs- und Systemebene. Zudem werden darin die nachfolgenden Kapitel entsprechend ihres Beitrages zur Gesamtarchitektur eingeordnet. Im Fokus dieser Arbeit befinden sich dabei die technischen dienstbasierten Prozesse im Rahmen der Anwendungsklassenkonsolidierung.

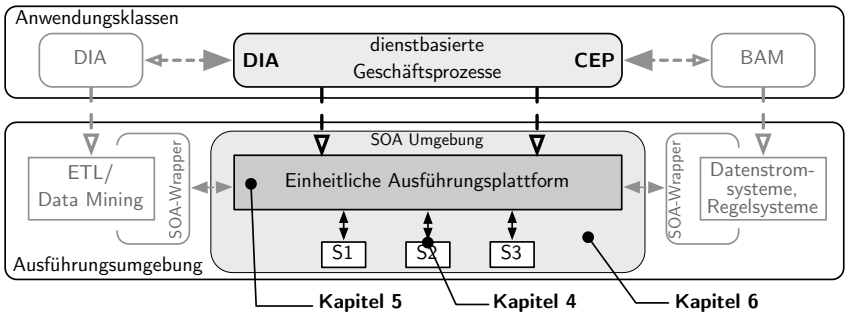


Abbildung 3.6: Überblick über die Zielarchitektur dieser Arbeit mit Kapitelreferenz.

4 Integration von Datenstromsemantik in die Dienstausführung

Auf Grundlage der Implikationen aus Kapitel 3 betrachtet dieses Kapitel die Ebene der Dienstkommunikation und stellt den Ansatz des *strombasierten Dienstaufrufes* vor. Dieser Ansatz beschreibt ein skalierbares und effizientes Aufrufmodell, welches den nativen gemeinsamen Kontext des *Gesamtnachrichtentransfer*s (Anforderung *A4* (Korrelation)) mit den Partitionierungskonzepten des *Einzelnachrichtentransfer*s (Anforderung *A2* (Datenaustausch)) und der Effizienz des *Teilnachrichtentransfer*s für einzelne Datenelemente (Anforderung *A3* (Latenz)) vereint und damit die dienstbasierte Komponentennutzung für weitere Anwendungsklassen öffnet. Der Ansatz des *strombasierten Dienstaufrufes* wird zum Zwecke der Evaluierung beispielhaft mithilfe der SOAP-Spezifikation implementiert. Die Allgemeingültigkeit dieses Ansatzes wird dadurch nicht eingeschränkt und erlaubt die Übertragung der Konzepte auf beliebige andere Nachrichtenformate und Datenrepräsentationen.

Grundlage des strombasierten Ansatzes bildet die Etablierung einer Datenstromsemantik zwischen Dienstanbieter und Dienstnutzer, welche

1. den nachrichtenbasierten Transfer beliebig großer Datenmengen bzw. kontinuierlicher Daten ermöglicht und dabei den Mehraufwand einzelner Nachrichtengenerierungen vermeidet (Anforderung *A2* und *A3*),
2. die Verarbeitungsmethodik bei Dienstanbieter und Dienstnutzer an einen strombasierten Datentransfer anpasst und
3. einen nativen gemeinsamen Kontext für alle transferierten Datenelemente anbietet (Anforderung *A4*).

Weiterhin wird beim Datentransfer von den zugrundeliegenden Daten abstrahiert, wodurch der Ansatz um Konzepte der Verarbeitungsoptimierung verfeinert werden kann. Entscheidend ist jedoch, dass die grundlegenden SOA-Eigenschaften der losen Kopplung durch standardisierten und plattformunabhängigen Nachrichtenaustausch, zustandslose Dienstinstanzen sowie eine plattformunabhängige Beschreibung der Dienste trotz der aufgelisteten Vorteile beibehalten werden.

Als formale Grundlage für dieses Kapitel sei ein Datensatz D gegeben, welcher aus einer Menge n an gleich strukturierten XML-basierten Datenelementen d_i mit $D = (d_1, \ldots, d_i, \ldots, d_n)$ besteht und die von einer Dienstinstanz s eines Dienstes S verarbeitet werden soll. Als Beispiel einer solchen Datenmenge sei der Prozessschritt

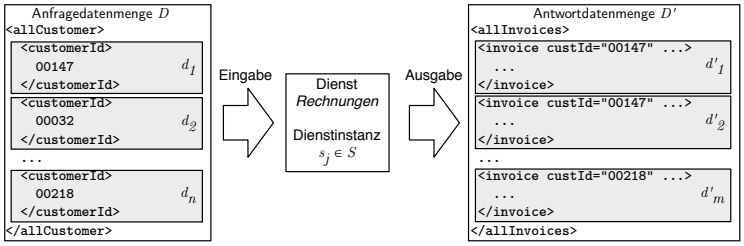


Abbildung 4.1: Beispieldaten der Prozessaktivität „lade Rechnungen“.

lade Rechnungen des Beispielprozesses *Top n Kunden* aus Abbildung 2.11 (Seite 25) gegeben, dessen Ein- und Ausgabedaten in Abbildung 4.1 dargestellt sind und die eine gleiche Struktur der einzelnen d_i aufweisen. Der Beispielprozess sendet dabei eine Menge an Kundennummern als XML-basierte Eingabedaten an eine Instanz s des entfernten Dienstes *Rechnungen*. Jede Kundennummer entspricht einem Datenelement $d_i \in D$. Für jede Kundennummer gibt die Dienstinstanz alle Rechnungen des letzten Quartals zurück.

4.1 Überblick

Die Kernidee des hier diskutierten *strombasierten Dienstaufrufes* ist es, die zu verarbeitenden Nutzdaten D der Größe n einer Anfragenachricht R als endlichen Nachrichtenstrom NS_I der Länge n zu beschreiben. Die Struktur einer klassischen Nachricht mit Kopf- und Nutzdatenbereich wird beibehalten und fungiert als umschließender Container des Stromes. Dadurch werden die Semantik einer nachrichtenbasierten Kommunikation bewahrt und ihre Eigenschaften der losen Kopplung und der Plattformunabhängigkeit beibehalten. Der Dienstnutzer kontrolliert das Einfügen der n Stromobjekte in den Strom und somit das Senden der Nutzdatenteile. Für den eigentlichen Transfer der Daten wird zudem eine Abstraktionsschicht eingeführt, die spätere Verarbeitungsoptimierungen sowie Funktionserweiterungen erlaubt. Außerdem kann der Dienstnutzer den Anfragestrom auf Wunsch schließen.

Ausgangspunkt des strombasierten Dienstaufrufes ist der bereits erläuterte *Gesamtnachrichtentransfer*, der eine Datenmenge D der Größe n in einer einzigen Anfrage R an den Dienst sendet, deren Verarbeitung initiiert und dadurch auch einen gemeinsamen Verarbeitungskontext für alle Datenelemente $d_i \in D$ bereitstellt. Für die Integration der partitionierten Kommunikation der Aufrufmethoden *Teilnachrichtentransfer* sowie *Einzelnachrichtentransfer* in die Methode des *strombasierten Dienstaufrufes* werden *drei* Modifikationen am ursprünglichen *Gesamtnachrichtentransfer* vorgenommen. Einen Überblick zu diesen Modifikationen gibt Abbildung 4.2. Diese stellt den Gesamtansatz des strombasierten Dienstaufrufes dar und enthält die Referenzen zu den weiterführenden Abschnitten dieses Kapitels.

Als erste Modifikation (vgl. Abschnitt 4.2) wird die Anfragenachricht R (vgl. Abbildung 2.4(a), Seite 12) eines Dienstnutzers C als Eingabestrom NS_I des Dienstes mit $R \rightarrow NS_I$ definiert. Dieser Eingabestrom transportiert den Datensatz D der Größe n in Übereinstimmung zu R als Nutzdaten der Nachricht. Analog dazu wird die Antwortnachricht R' als Ausgabestrom NS_O des Dienstes mit $R' \rightarrow NS_O$ definiert, welcher in Übereinstimmung zu R' die verarbeiteten Daten D' an den Dienstnutzer zurückgibt. Da der Eingabestrom NS_I analog zur Anfragenachricht R die Nutzdaten der Größe n besitzt, gilt der Eingabestrom nach dem Transport der n Datenelemente $d_i \in D$ als beendet und wird geschlossen. Somit repräsentiert NS_I eine Anfrage im klassischen Sinne. Jede Anfragenachricht R_j eines Dienstnutzers c_j mit einem separaten D_j generiert einen neuen Eingabestrom $NS_{I,j}$ und, in Abhängigkeit von der implementierten Dienstfunktion, einen Rückgabestrom $NS_{O,j}$ (siehe Abbildung 4.2). Dies impliziert, dass jedes $\{NS_{I,j}, NS_{O,j}\}$-Paar genau zu einer Anfrage R_j eines Nutzers c_j gehört und somit genau von einer Dienstinstanz s_j verarbeitet wird. Dadurch existieren zwei Datenströme für Ein- und Ausgabedaten, die analog zur Methode des *Gesamtnachrichtentransfer* einen nativen gemeinsamen Kontext über alle Daten in D bereitstellen.

Als zweite Modifikation (vgl. Abschnitt 4.3) gilt es, die Art des Datentransfers eines Datensatzes D_j anzupassen, um eine partitionierte Übertragung und damit die strombasierte Verarbeitung der Datenelemente $d_{i,j}$ zu ermöglichen. Deshalb wird der eigentliche Transfer der Daten auf das Konzept von *Verarbeitungseinheiten* (engl. *processing buckets*, kurz Buckets oder Verarbeitungsbuckets) b abgebildet, welches von den ursprünglich zu übertragenden Nutzdaten abstrahiert. Diese Abstraktion erlaubt eine spätere Verarbeitungsoptimierung sowie Funktionserweiterungen auf Transferebene. Somit wird D_j auf eine Menge von l Verarbeitungsbuckets B mit $B = (b_1, b_2, \ldots, b_l)$ aufgeteilt, wobei der Dienst zusichert, ein solches Bucket in einem Schritt verarbeiten zu können. Die Struktur der Anwendungsdaten in diesen Buckets wird vom Dienst festgelegt und in seiner Dienstbeschreibung hinterlegt.

Im Rahmen der dritten Modifikation (vgl. Abschnitt 4.4) ist das Verarbeitungsmodell der Dienstinstanzen s anzupassen, um die bereits partitioniert eintreffenden Datenelemente d_i in den l Verarbeitungsbuckets b strombasiert und damit skalierbar

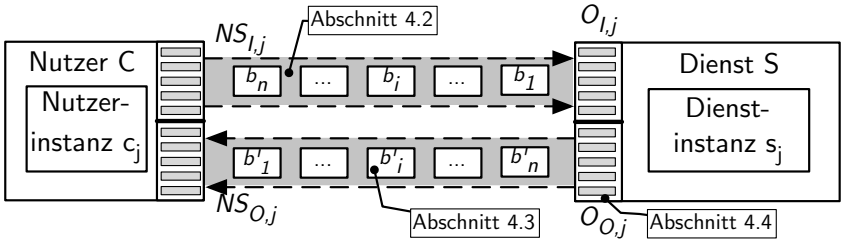


Abbildung 4.2: Überblicksarchitektur des *strombasierten Dienstaufrufes*.

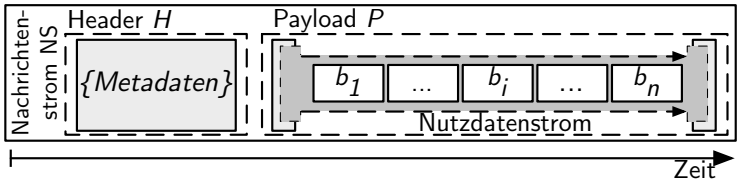


Abbildung 4.3: Aufbau des Nachrichtenstroms NS.

zu verarbeiten. Dazu wird der Empfang von Eingabebuckets und das Zurücksenden von Ausgabebuckets in den einzelnen Dienstinstanzen s_j entkoppelt, indem Ein- und Ausgabewarteschlangen ($Q_{I,j}$ bzw. $Q_{O,j}$) jeweils einer Instanz s_j zugewiesen werden. Diese Warteschlangen transferieren die Eingabe- und Ausgabebuckets zwischen Dienstnutzer und Dienstinstanz, analog zu den bereits diskutierten *Message Queuing*-Systemen, asynchron und entkoppeln somit die Kommunikation von der eigentlichen Verarbeitung. In den nachfolgenden Abschnitten werden die Modifikationen detailliert vorgestellt.

4.2 Kommunikationsprotokoll

Wie bereits im vorherigen Abschnitt angedeutet, wird als Grundlage des Kommunikationsprotokolls eine Anfragenachricht R in einen Anfragestrom NS transformiert, in dem die zu übertragenden Nutzdaten als abstrakte Stromobjekte b transferiert und letztendlich beim Dienst S verarbeitet werden. Die grundlegende Struktur eines Anfragestromes NS entspricht dabei dem Aufbau einer standardisierten Nachricht (vgl. Aufbau SOAP, Abschnitt 2.1.3, Abbildung 2.5, Seite 13). Eine solche Nachricht besteht neben dem eigentlichen Nachrichtenumschlag aus einem Kopfbereich H (engl. *header*) sowie einem Nutzdatenbereich P (engl. *payload*). Der Kopfbereich H enthält Metadaten über die Nachricht selbst, wie beispielsweise Zieladresse bzw. Endpunkt der Nachricht, transaktionale Kontexte für einen robusten Nachrichtentransfer oder Verschlüsselungsinformationen für die Verarbeitung der Nutzdaten. Der Nutzdatenbereich P enthält die eigentlichen Nutzdaten und somit die Datenmenge D.

Der erzeugte Anfragestrom NS des hier vorgestellten *strombasierten Dienstaufrufes* ist ähnlich strukturiert (vgl. Abbildung 4.3). Der Kopfbereich H des Stromes beinhaltet die gleichen Metadaten wie eine Anfragenachricht und wird somit zur Aushandlung der Verbindung zwischen Dienstnutzer und Dienstinstanz und zur Instantiierung des Stromes genutzt. Der Nutzdatenbereich P beinhaltet den Datensatz D, dessen Datenelemente d_i zur Abstraktion und späteren Optimierung über l Verarbeitungsbuckets b verteilt sind. Verglichen mit dem Aufbau einer klassischen Nachricht, entspricht ein Anfragestrom NS einer zeitlich gestreckten Nachricht R, zu welcher der Dienstnutzer der Anfragenachricht R auch nach dem Absenden des Nachrichtenkopfes H noch Nutzdaten in Form von Buckets b hinzufügen kann. Analog gelten diese Eigenschaften für den Rückgabestrom NS_O bzw. R'. Aufgrund der Tatsache,

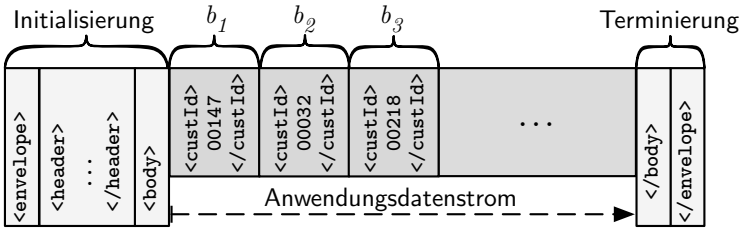


Abbildung 4.4: Beispielnachrichtenstrom NS_I auf Basis der SOAP-Spezifikation.

dass der Aufbau des Stromes dem einer traditionellen Nachricht entspricht, kann der Strom prinzipiell auch von klassischen Diensten, die den gesamten Nachrichtenstrom puffern und die entsprechende Datenstruktur der Nachricht im Speicher aufbauen, gelesen werden.

Damit ein Dienstnutzer eine Anfrage R als Anfragestrom NS_I versenden kann, sendet er zunächst den Kopfbereich H des Anfragestromes und die Startmarkierung des Nutzdatenbereiches an den Dienstanbieter. Die Nutzung von SOAP als Nachrichtenformat für die strombasierte Anfragenachricht zeigt Abbildung 4.4. Dabei umfasst der Kopfbereich des Anfragestromes das öffnende XML-Tag des *envelop*-Elementes sowie das gesamte *header*-Element. Die Startmarkierung, welche den Beginn des Nutzdatenbereichs anzeigt, bildet das öffnende XML-Tag des `<body>`-Elementes. Dadurch wird eine Dienstinstanz s beim Dienstanbieter initiiert und der Eingabestrom NS_I zur Dienstinstanz aufgebaut. Diese Dienstinstanz wartet nun auf die eigentlichen Nutzdaten und der Dienstnutzer ist in der Lage, die Datenmenge D mithilfe der Verarbeitungsbuckets b an den Dienst zu übermitteln. Im Beispiel wird ein Verarbeitungsbucket durch eine Kundennummer repäsentiert. Dabei verarbeitet die Dienstinstanz alle Verarbeitungsbuckets b im Kontext dieses einen Eingabestroms NS_I. Wurden durch den Dienstnutzer alle Daten in den Anfragestrom NS_I eingefügt, platziert dieser die Endmarkierung des Nutzdatenbereiches im Anfragestrom NS_I (schließendes XML-Tag `</body>` bei SOAP-Nachrichtenformat) sowie die Endmarkierung der gesamten Nachricht (schließendes XML-Tag `</envelope>` beim SOAP-Nachrichtenformat) und der Anfragestrom NS_I wird geschlossen.

Der Dienst baut den Ausgabe- bzw. Rückgabestrom NS_O zurück zum Dienstnutzer auf, sobald er das erste Bucket b'_1 für die Rückgabe an den Dienstnutzer erstellt hat. Die Methodik für den Aufbau des Ausgabestromes NS_O entspricht dabei der Methodik des Eingabestromes NS_I. Durch die nebenläufige Handhabung von Ein- und Ausgabestrom können Rückgabebuckets b'_i bereits an den Dienstnutzer zurückgegeben werden, während Anfragebuckets b_i noch vom Dienstnutzer zum zur Dienstinstanz übertragen werden. Sobald das letzte Rückgabebucket b'_l an den Dienstnutzer zurückgesandt wurde, schließt die Dienstinstanz den Ausgabestrom NS_O durch das Senden der Endmarkierung des Nutzdatenbereichs sowie der Endmarkiertung der gesamten Nachricht.

Auf Grundlage des eben beschriebenen Kommunikationsprotokolls lässt sich ein *strombasierter Dienst* zunächst wie folgt definieren:

Definition 7 Strombasierter Dienst: *Ein strombasierter Dienst S wird mit $S = (id, NS_I, NS_O, f)$ definiert. Dabei beschreibt id die eindeutige ID des Diensttyps, NS_I den Eingabestrom vom Dienstnutzer zum Dienst, NS_O den Ausgabestrom vom Dienst zum Dienstnutzer sowie f die vom Dienst implementierte Funktion.*

4.3 Datenmodell

Das Datenmodell des *strombasierten Dienstaufrufes* basiert wie auch das Nachrichtenformat SOAP im Grundsatz auf XML und beschreibt die Struktur und die Semantik, in welcher Nutzdaten über die Ein- und Ausgabeströme NS übertragen werden. Wie bereits in Abschnitt 4.1 kurz beschrieben, wird die Datenmenge D der Größe n zur Übertragung auf das Konzept der Verarbeitungsbuckets abgebildet. Dieses Konzept abstrahiert physisch von den eigentlichen Anwendungsdaten beim Datentransfer. Die Aufteilung von n Datenelementen $d_i \in D$ erfolgt dabei auf l physische Verarbeitungsbuckets b_i. Somit definieren Buckets die Verarbeitungsgranularität von D beim Dienst und ermöglichen so eine partitionierte, strombasierte Abarbeitung.

4.3.1 Definition

Um die Vorteile einer physischen Abstraktionsschicht bei der Datenübertragung von D an eine Dienstinstanz s aufzuzeigen, wird im Folgenden zunächst der strombasierte Transfer auf Basis einer logischen Abstraktion diskutiert. Da D aus n gleich strukturierten Datenelementen d_i besteht, kann zunächst die Partitionierung der Daten auf die Granularität von d_i erfolgen, wobei $d_i == b_i$ gilt (vgl. auch Abbildung 4.4, Buckets b_1, b_2 und b_3). Darauf aufbauend ist eine Zusammenfassung einer Anzahl von m Datenelementen d_i zu einem logischen b_i mit $m < n$ möglich, wodurch eine Dienstinstanz s_j die Verarbeitung eines d_i erst startet, wenn eine Anzahl von $m \cdot d_i$ in s eingetroffen ist. Dieses *logische Datenmodell* wird wie folgt definiert:

Definition 8 Logisches Datenmodell: *Das logische Datenmodell basiert auf dem Konzept der Verarbeitungsbuckets und definiert ein solches Bucket b_j als Menge von m XML-Elementen d_i aus D mit $b_j = (d_1, \ldots, d_m)$ und $1 \leq m \leq |D|$. Dabei bildet zunächst jedes Bucket b_j eine einfache Liste von Datenelementen d_i.*

Abbildung 4.5a zeigt das *logische Datenmodell* am Beispiel der Kundeninformationen aus Abbildung 4.1, die zum Rechnungsdienst *Rechnung* übertragen werden sollen. In diesem Beispiel entspricht jedes Datenelement d_i einem einzelnen Bucket b_i, wodurch die Anzahl n der Datenelemente der Anzahl der Verarbeitungsbuckets l mit $n = l$ entspricht. Dieses *logische Datenmodell* hat den Vorteil, dass alle Buckets b_i auf textueller Ebene der Konkatenation aller Datenelemente d_i entsprechen. Dadurch

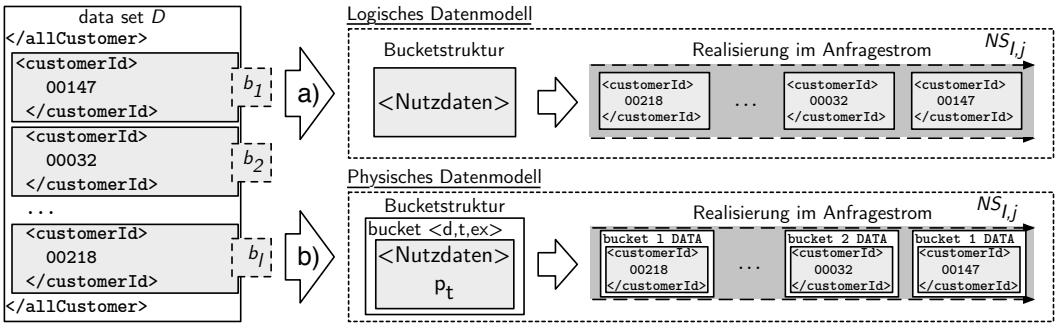


Abbildung 4.5: Einfaches und erweitertes Bucket-Datenmodell.

gleicht der Strom NS textuell der Nachricht R und aufgrund der Strukturäquivalenz können auch klassische Dienste diesen Nachrichtenstrom verarbeiten. Durch die direkte Abbildung von d_i auf b_i beschränkt sich die mögliche Struktur der übertragbaren Daten jedoch auf die Struktur von d_i. Da die bisherige Annahme gilt, dass alle d_i dem gleichen Schema entsprechen, wird die Übertragung andersartiger Anwendungs- bzw. Steuerdaten erschwert. Deshalb gestaltet sich beispielsweise die Implementierung parametrierbarer und damit allgemeingültiger Dienste als schwierig.

Datenmodell des strombasierten Dienstaufrufes

Aufgrund der beschriebenen Nachteile des *logischen Datenmodells* beschreibt dieser Abschnitt ein Datenmodell auf Basis von expliziten, physischen Verarbeitungsbuckets. Ein Verarbeitungsbucket b wird wie folgt definiert:

Definition 9 Physisches Datenmodell: *Das physische Datenmodell basiert, wie auch das logische Datenmodell, auf dem Konzept der Verarbeitungsbuckets, definiert jedoch ein Verarbeitungsbucket als XML-Element* $b = (d, t, ex, p_t)$ *mit* d *als eindeutiger Bucket-ID,* t *als Kennzeichnung des Buckettyps und* p_t *als XML-basierter Nutzlast in Abhängigkeit des Buckettyps* t*. Dabei beinhaltet* p_t *analog zum logischen Datenmodell die Nutzlast in Form von* m *XML-Elementen mit* $p_t = (d_1, \ldots, d_m)$*. Der Basis-Buckettyp* b_{data}*, der seine Nutzdaten als Anwendungsdaten für die Dienstinstanz definiert, wird in* `t` *mit* `data` *gekennzeichnet. Weiterhin beschreibt* ex *eine erweiterbare Liste an Bucketattributen in Form von Schlüssel-Wert-Paaren, die für die Annotation zusätzlicher Informationen verwendet werden kann.*

Den schematischen Aufbau sowie ein Beispiel eines solchen Buckets zeigt Abbildung 4.5b. Darin umschließt das Bucket die Nutzdaten und erlaubt somit die Annotation weiterer Informationen im Kopfbereich. Dies ermöglicht eine flexible Erweiterung des Datenmodells um weitere Buckettypen oder die Einführung priorisierbarer

Verarbeitungsbuckets. Somit können weiterführende Konzepte einer flexible Parametrierung von Dienstinstanzen (vgl. Abschnitt 4.3.2) oder der Transfer von Systemparametern der darunterliegenden Ausführungsumgebung realisiert werden.

Die Nachteile der hier vorgestellten physischen Abstraktion auf Ebene der Verarbeitungsbuckets liegen im Mehraufwand für die Generierung und den Transfer der expliziten Verarbeitungsbuckets. Außerdem entspricht die textuelle Repräsentation des Anfragestroms NS nicht mehr dem ursprünglichen Datensatz D. Dennoch überwiegen die Vorteile einer hohen Flexibilität und Erweiterbarkeit. Die explizite Dienstparametrierung als eine solche Erweiterung wird im folgenden Abschnitt erläutert.

4.3.2 Dienstinstanzparametrierung

Das physische Datenmodell definiert bisher lediglich einen Buckettyp t mit b_{data}, der die Nutzdaten des Buckets als Anwendungsdaten in Form von gleichartig strukturierten Datenelementen $d_i \in D$ deklariert (t =`data`). Eine Übertragung expliziter Parameter, welche die Dienstfunktion f (vgl. Definition 7) für eine Menge von Anwendungsdaten d_i konfiguriert, ist jedoch wünschenswert und ermöglicht die Generalisierung von Dienstimplementierungen analog zu den klassischen Dienstaufrufen.

Abbildung 4.6 stellt Anwendungs- und Parameterdaten für den Beispieldienst *Rechnungen* aus Abbildung 4.1 dar. Dabei repräsentieren die einzelnen Kundennummern Anwendungsdaten für die Dienstinstanz. Zusätzlich benötigt die Dienstinstanz für ihre korrekte Ausführung den Zeitraum der zu extrahierenden Rechnungen. Dadurch werden nur Rechnungen vom Dienst geladen und zurückgegeben, die in diesem Zeitraum erstellt wurden. Die Übertragung dieses Zeitraums als expliziter Parameter der Dienstanfrage wird jedoch im bisherigen Datenmodell nicht unterstützt. Ein fester Zeitraum müsste in der Dienstimplementierung hinterlegt werden.

Die Möglichkeiten zur Integration von Dienstparametern in den Anfragestrom lassen sich in *strukturelle* und *zeitliche* Aspekte unterteilen (vgl. Abbildung 4.7). Der *strukturelle* Aspekt beschreibt, in welcher Art und Weise die Parameter in den Strom eingefügt werden und wie sie zum Dienst gelangen. Der *zeitliche* Aspekt hingegen definiert, wann ein Parameter in den Strom eingefügt wird und mit welcher Semantik ihn ein Dienst verarbeitet. Dabei sind die Ausprägungen beider Aspekte unabhängig voneinander und können beliebig kombiniert werden.

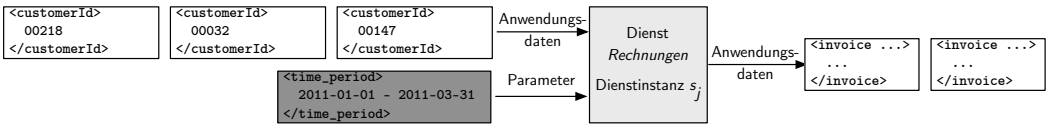


Abbildung 4.6: Anwendungs- und Parameterdaten für Dienst *Rechnungen*.

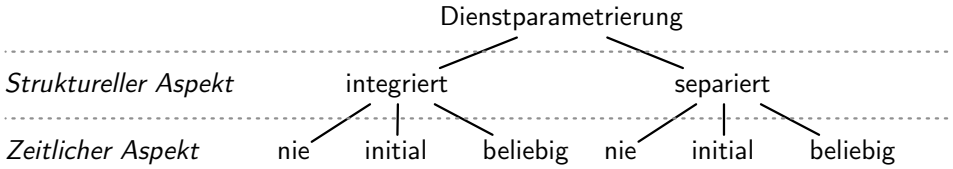


Abbildung 4.7: Aspekte und Ausprägungen der Dienstparametrierung.

Struktureller Aspekt

Der strukturelle Aspekt beschreibt die grundlegende Art und Weise, wie Parameter im Anfragestrom physisch repräsentiert werden. Eine Entscheidung für eine dieser Ausprägungen bildet die konzeptuelle Grundlage für das Gesamtsystem der Dienstkommunikation und damit auch für alle Ausprägungen von Dienstanbietern und Dienstnutzern. Eine Darstellung beider Ausprägungen des strukturellen Aspektes bietet Abbildung 4.8.

Die erste Ausprägung, der *integrierte* Ansatz, besteht darin, die Parameter p_i direkt mit den Anwendungsdaten b_i als Teil der eigentlichen Bucketnutzdaten p_t in einem Datenbucket zu platzieren (vgl. Abbildung 4.8a). Daraus ergeben sich folgende Nachteile: Neben der semantischen Vermischung von Anwendungsdaten und Parametern in einer XML-Datenstruktur ist es beim Erhalt des Buckets zunächst unklar, ob ein Bucket neben den eigentlichen Daten auch Parameter überträgt. Deshalb müssen die Nutzdaten jedes Buckets vollständig gescannt und daraufhin überprüft werden, ob Parameter in den Nutzdaten vorhanden sind bzw. ob sich die Parameterwerte im Vergleich zum vorherigen Bucket geändert haben.

Die zweite Ausprägung, der *separierte* und in dieser Arbeit genutzte Ansatz, besteht in der Einführung eines neuen Buckettyps b_{param} (t =`PARAM`), welcher die Nutzlast eines Buckets explizit als Parameter für eine Dienstinstanz deklariert (vgl. Abbildung 4.8a). Dies führt zu einer klaren Unterscheidung der Semantik einzelner Buckets und ihrer Nutzdaten bereits auf Bucketebene, ohne die zwingende Notwendigkeit, für diese Entscheidung die Nutzdaten zu interpretieren. Durch die Nutzung des *separierten* Ansatzes erweitert sich die Signatur des strombasierten Dienstes aus Definition 7 wie folgt:

Definition 10 Erweiterter strombasierter Dienst: *Ein erweiterter strombasierter Dienst S wird mit $S = (id, NS_I, NS_O, f, p_f)$ definiert. Dabei beschreibt id die ein-*

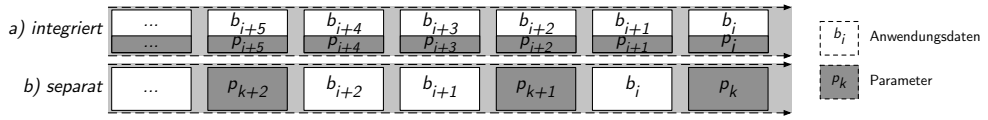


Abbildung 4.8: Strukturelle Ausprägungen der Dienstparametrierung.

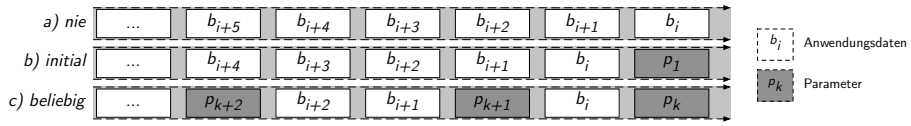


Abbildung 4.9: Zeitliche Ausprägungen der Dienstparametrierung.

deutige Id des Diensttyps, NS_I den Eingabestrom vom Dienstnutzer zum Dienst, NS_O den Ausgabestrom vom Dienst zum Dienstnutzer, f die vom Dienst implementierte Funktion und p_f den von f abhängigen Parameter, welcher f konfiguriert.

Zeitlicher Aspekt

Aufbauend auf diesem *strukturellen* Ansatz der *separierten* Parameterbuckets zeigt Abbildung 4.9 die möglichen *zeitlichen* Aspekte der Dienstparametrierung. Die Ausprägung dieses Aspektes ist, im Gegensatz zum *strukturellen* Aspekt, vom Dienstanbieter und seinen Diensten frei wählbar.

Abbildung 4.9a zeigt einen Anfragestrom, bei welchem die Parametrierung der Dienstinstanz nicht unterstützt wird. Abbildung 4.9b beschreibt die initiale Parametrierung einer Dienstinstanz zum Zeitpunkt ihrer Erstellung und entspricht im Beispiel des Rechnungsdienstes der Zeitspanne von drei Monaten für die Extraktion der Kundenrechnungen (vgl. Abbildung 4.6). Ein Anwendungsszenario für Abbildung 4.9c bildet die Rekonfigurierung einer Dienstinstanz, wie beispielsweise die Änderung der Zeitspanne für zu extrahierende Rechnungen, ohne den Anfragestrom zu unterbrechen und neu zu initialisieren. Dabei sind aktuelle Parameterausprägungen bis zum Eintreffen eines neuen Parameterbuckets gültig. Der Vorteil einer Rekonfiguration von Dienstinstanzen wird in Zusammenhang mit stehenden Dienstinstanzen in Abschnitt 4.4.3 verdeutlicht.

Zur Unterstützung der Variationen des zeitlichen Aspektes aus Abbildung 4.9, die abhängig von der Dienstimplementierung angeboten werden, unterscheidet die Ausführungsumgebung drei zeitliche Ausprägungen der Parameterübergabe, welche entsprechend von den Diensten angezeigt werden müssen:

Parametermodus	Bedeutung
NONE (nie)	Keine Parametrierung des Dienstes möglich.
INIT (initial)	Parametrierung nur beim Start der Dienstinstanz möglich. Gültig für *alle* nachfolgenden Verarbeitungseinheiten oder bis zur Terminierung des Stroms.
INTERMEDIATE (beliebig)	Parametrierungen an beliebigen Stellen im Strom möglich. Gültig für *nachfolgende* Verarbeitungseinheiten *bis* zum Eintreffen eines neuen Parameterbuckets oder der Stromterminierung.

Tabelle 4.1: Zeitliche Definition der Parameterübergabe in der Laufzeitumgebung.

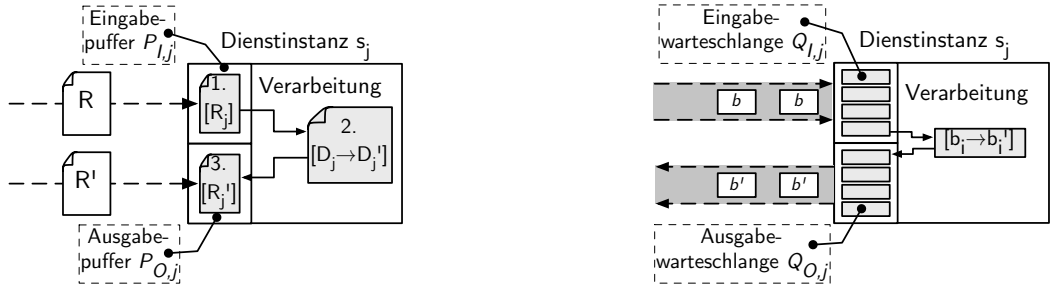
4.4 Dienstverarbeitungsmodell

Auf Grundlage des Bucketkonzeptes aus dem vorherigen Kapitel in Verbindung mit dem vorgestellten Stromtransfer können die einzelnen Datenelemente strombasiert übertragen werden. Um jedoch den strombasierten Datentransfer auch auf der Ebene der Dienstverarbeitung zu unterstützen und dessen Vorteile vollständig zu nutzen, muss das Verarbeitungsgranulat in der Dienstinstanz von derzeit einer gesamten Nachricht auf das Granulat einzelner Verarbeitungsbuckets verfeinert werden (vgl. *Modifikation 3*, Abschnitt 4.1).

Im folgenden Abschnitt werden zunächst die Kernpunkte des bisherigen, dienstbasierten Verarbeitungsmodells herausgestellt und darauf aufbauend ein Überblick über die in dieser Arbeit zugrundeliegende strombasierte Verarbeitung gegeben. Zur Beschreibung der Verarbeitungsmodelle wird angenommen, dass ein Datenelement $d_i \in D$ beim Dienst genau ein Antwortelement $d'_i \in D'$ und somit $|D| == |D'|$ bedingt. Andere Ein- und Ausgabebeziehungen werden im Anschluss diskutiert.

4.4.1 Überblick

Im traditionellen *Gesamtnachrichtentransfer* basiert das Verarbeitungsmodell auf einem klassischen Methodenaufruf, bei dem die Nutzdaten der Eingangsnachricht als Parameter an die implementierte Dienstfunktion übergeben werden. Dadurch kann die Dienstinstanz s_j beliebig auf die gesamte Datenmenge D_j in R_j zugreifen. Die Verarbeitung einer Eingangsnachricht erfolgt im klassischen Verarbeitungsmodell schrittweise, wie Abbildung 4.10(a) zeigt. Die Nachricht R_j trifft ein und initialisiert damit eine neue Dienstinstanz s_j. R_j und damit D_j werden im Eingabepuffer $P_{I,j}$ abgelegt und die interne Datenstruktur entsprechend D_j aufgebaut. Danach erfolgen in einem zweiten Schritt die Verarbeitung von D_j und die Generierung des Antwortdatensatzes D'_j. Erst nachdem alle Daten in D_j durch den Dienst verarbeitet



(a) Schrittweises Verarbeitungsmodell.

(b) Strombasiertes Verarbeitungsmodell.

Abbildung 4.10: Nachrichtenbasierte Dienstkommunikation.

wurden, wird D'_j in einem dritten Schritt in den Ausgabepuffer $P_{O,j}$ geschrieben und mithilfe der Antwortnachricht R' an den Dienstnutzer zurückgesandt. Diese Verarbeitungssemantik entspricht dem bereits diskutierten *Anfrage-Antwort*-Paradigma bei der Kommunikation von Dienstkomponenten.

Im Gegensatz zum klassischen Verarbeitungsmodell erfordert eine strombasierte Kommunikation eine verfeinerte Verarbeitung des Datensatzes D_j auf der Granularität der Verarbeitungsbuckets b_i. Dafür sind zwei Modifikationen am Verarbeitungsmodell und der Dienstarchitektur notwendig.

- Definition von Ein- und Ausgabewarteschlangen
- Anpassung des Programmiermodells an feingranulare Verarbeitung

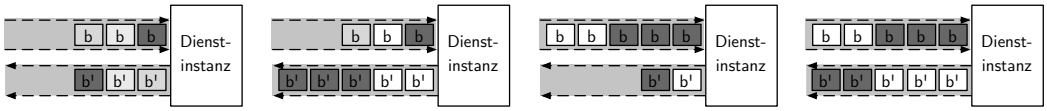
Diese Modifikationen werden im Folgenden anhand der Architektur einer Dienstinstanz (vgl. Abbildung 4.10(b)) beschrieben:

Als erste Modifikation werden Eingabe- und Ausgabewarteschlangen $Q_{I,j}$ und $Q_{O,j}$ pro Dienstinstanz s_j definiert. Diese basieren auf dem *FirstIn-FirstOut*-Prinzip (*FIFO*) und puffern eintreffende bzw. ausgehende Buckets. Durch das *FIFO*-Prinzip wird sichergestellt, dass die Reihenfolge der Verarbeitung der Reihenfolge der eintreffenden Buckets entspricht. Die Nutzung von Warteschlangen ermöglicht eine Entkopplung von Anfrage- und Antwortstrom und resultiert somit in einer asynchronen Kommunikation zwischen Dienstinstanz und Dienstnutzer. Die Dienstinstanz s_j greift auf die Eingabewarteschlange $Q_{I,j}$ und damit auf die Eingangsbuckets b_i zu und fügt entsprechend Antwortbuckets b'_i in die Ausgabewarteschlange $Q_{O,j}$ ein.

Im Rahmen der zweiten Modifikation wird das Programmiermodell der feingranularen Verarbeitung einzelner Eingangsbuckets angepasst. Da Anforderung *A2 (Datenaustausch)* den Transfer und die Verarbeitung beliebig großer Datenmengen fordert, kann im Allgemeinen nicht garantiert werden, dass die gesamte Datenmenge D_j eines Dienstaufrufes im zugewiesenen Hauptspeicher der Dienstinstanz verarbeitet werden kann. Deshalb muss der Zugriff auf eintreffende Buckets des Eingabestroms mithilfe einer vorwärtsgerichteten, cursorbasierten Methodik vollzogen werden. Dabei iteriert die Anwendungslogik über die Eingangsbuckets und sieht jedes Bucket damit prinzipiell nur einmal. Trotz dieses Verarbeitungsmodells ist es der Anwendungslogik natürlich freigestellt, die bereits bearbeiteten Buckets für spätere Informationen persistent oder transient zu speichern. Lediglich die Funktionsfähigkeit des Dienstes ist bei beliebig großen Datenmengen sicherzustellen.

4.4.2 Ein- und Ausgabebeziehungen

Die strombasierte Verarbeitung von Eingabebuckets in einer Dienstinstanz ermöglicht unterschiedliche Eingabe-Ausgabe-Beziehungen zwischen Eingabebuckets und daraus resultierenden Ausgabebuckets. Dabei hängt die Beziehung stark von der Art und Weise der Dienstimplementierung ab. Viele Dienstfunktionen lassen sich mit



(a) 1:1. (b) 1:N. (c) N:1. (d) N:M.

Abbildung 4.11: Ein-/Ausgabebeziehungen strombasierter Dienstimplementierungen.

mehr als einer Ein- und Ausgabebeziehung umsetzen. Somit stellen die im Folgenden genannten Ein- und Ausgabebeziehungen lediglich Verhaltensklassen strombasierter Dienste dar, ermöglichen dem Dienstnutzer jedoch eine entsprechend effiziente Nutzung und Einbindung in dessen Arbeitsfluss:

1:[0,1] *Ein Eingabebucket bedingt kein oder ein Ausgabebucket (Abbildung 4.11(a)).* Diese Eingabe-Ausgabe-Beziehung beschreibt die Funktionsklasse, in der jedes Eingabebucket b_i unabhängig von anderen Buckets in der Dienstinstanz verarbeitet werden kann. Dabei werden alle mit dem Eingabebucket b_i verbundenen Rückgabedaten d'_{b_i} in einem Antwortbucket b'_i platziert. Für Beispiel dieser Ein- und Ausgabe-Beziehungen sei auf Abbildung 4.1 verwiesen, in der der Dienst pro eintreffender Kundennummer genau ein Antwortbucket generiert und zurückgibt. In diesem Antwortbucket sind alle Rechnungen des Kunden enthalten. Handelt es sich bei der implementierten Dienstfunktion um einen Filteralgorithmus, so gilt das Verhältnis 1:0, da das Antwortbucket b'_i eines Eingabebuckets b_i durch den Algorithmus verworfen werden kann.

1:N *Ein Eingabebucket bedingt ein oder mehrere Ausgabebuckets (Abbildung 4.11(b)).* Diese Beziehung beschreibt die Funktionsimplementierung, in der die Rückgabedaten d_{b_i} eines Eingabebuckets b_i in einem oder mehreren Ausgabebuckets an den Dienstnutzer zurückgegeben werden. Die Aufteilung der Antwortdaten auf mehrere Ausgabebuckets wird meist aus Gründen der effizienten Verarbeitung sowohl beim Dienst als auch beim Dienstnutzer vollzogen, da auf diese Weise die Granularität der Verarbeitungseinheiten fein gehalten wird und die strombasierte Verarbeitung somit beim Dienstnutzer fortgeführt werden kann. Ein Beispiel dieser Ein- und Ausgabebeziehung sei wiederum Abbildung 4.1, bei der jede Rechnung einer eintreffenden Kundennummer, im Gegensatz zur Eingabe-Ausgabe-Beziehung 1:[0,1], als separates Anwortbucket zurückgegeben wird.

N:1 *Mehrere Eingabebuckets bedingen ein Ausgabebucket (Abbildung 4.11(c)).* Diese Beziehung beschreibt im Allgemeinen die Funktionsklasse der Aggregation. Je nach Algorithmus und Datenverteilung im Eingabestrom kann das Ergebnis des Algorithmus dabei erst nach einer Menge an Eingabebuckets eindeutig bestimmt werden. Natürlich ist es der Dienstimplementierung freigestellt, vorläufige Werte an den Dienstnutzer zurückzusenden. Diese können vom Dienst-

nutzer für vorläufige Abbruchbedingungen genutzt werden, wenn beispielsweise ein Maximalwert einer Aggregation bereits nach einer geringen Menge an Eingabebuckets überschritten wird und der exakte Maximalwert deshalb nicht mehr berechnet werden muss.

N:M *Mehrere Eingabebuckets bedingen mehrere Ausgabebuckets (Abbildung 4.11(d)).* Diese Beziehung beschreibt Funktionen, welche für eine zusammengehörende Menge N an Eingabebuckets eine zusammengehörende Menge M an Ausgabebuckets generieren. Sie stellt eine Generalisierung der anderen Eingabe- und Ausgabebeziehungen dar und beschreibt vor allem das Verhalten holistischer Funktionen, die eine Menge von N Eingangsbuckets zusammen verarbeiten, jedoch deren Ausgabemenge nicht direkt mit der Eingabemenge korreliert.

4.4.3 Variationen des Verarbeitungsmodells

Das in diesem Abschnitt vorgestellte strombasierte Verarbeitungsmodell erfüllt in Verbindung mit dem strombasierten Datentransfer aus Abschnitt 4.2 die der Dienstebene in Kapitel 3 zugewiesenen Anforderungen *A2 (Datenaustausch)*, *A3 (Latenz)* und *A4 (Korrelation)*. Durch die initiale Zuordnung eines Eingabestromes $NS_{I,j}$ zu einer Dienstinstanz s_j werden alle Eingabebuckets eines Dienstnutzers c_j in einer *gemeinsamen* Dienstinstanz s_j strombasiert verarbeitet, was eine direkte Korrelation dieser Datenermöglicht.

Kombination mit methodenbasiertem Verarbeitungsmodell

Wie bereits angedeutet, kann der strombasierte Datentransfer aus Abschnitt 4.2 auch als reines Datentransferkonzept genutzt werden, ohne das Verarbeitungsmodell dafür anzupassen. Daraus folgt, dass der strombasierte Datentransfer auf Dienstseite mit dem klassischen schrittweisen Dienstverarbeitungsmodell aus Abbildung 4.10(a) kombiniert wird. Die Semantik dieser Kombination aus strombasiertem Transfer und klassischem Verarbeitungsmodell liegt darin, dass jedes einzelne Eingabebucket b_i des Eingabestromes NS_I eine logische, separate Nachricht R bzw. deren Nutzlast d_i repräsentiert. Jedes Bucket b_i instanziiert eine eigene Dienstinstanz s_i, welche die Nutzlast von b_i isoliert von nachfolgenden Buckets verarbeitet. Dies impliziert, dass die Summe n aller Eingabebuckets b_i eine Reihe von n logischen Einzelnachrichten R_i zwischen einem Dienstnutzer c_k und einem Diensttyp S_k bildet, jedoch der Mehraufwand für die Nachrichtenerstellung jeder Einzelnachricht R_i entfällt.

Neben der Reduzierung der Nachrichtenerstellung zwischen einem Diensttyp und einem Dienstnutzer, können durch den Transfer dieser logischen Einzelnachrichten auch für jede Nachricht wiederkehrende Aktionen reduziert werden. Ein Beispiel dafür bildet die einmalige Authentifizierung und Authorisierung des Dienstnutzers im Kontext des Eingabestroms NS_I. Die Nutzdaten von vormals separaten Nachrichten R werden als Stromobjekte b in den Strom eingefügt und auf Dienstseite durch

separate Methodenaufrufe abgearbeitet, ohne jeweils für jede einzelne Nutzlast die Authentifizierungsdaten zu validieren.

Trotz dieser Vorteile und der Tatsache, dass die Nutzung des methodenbasierten Verarbeitungsmodells keine Änderungen an der ursprünglichen Dienstimplementierung bedingt, verhindert die isolierte Verarbeitung einzelner Buckets b die Unterstützung von Anforderung A4 (*Korrelation*) und damit die Eignung im Gesamtkonzept dieser Arbeit. Dennoch realisiert das methodenbasierte Verarbeitungsmodell in Kombination mit dem strombasierten Datentransfer eine effiziente Dienstkommunikation, wie sich auch in der Evaluierung in Abschnitt 4.5 zeigt.

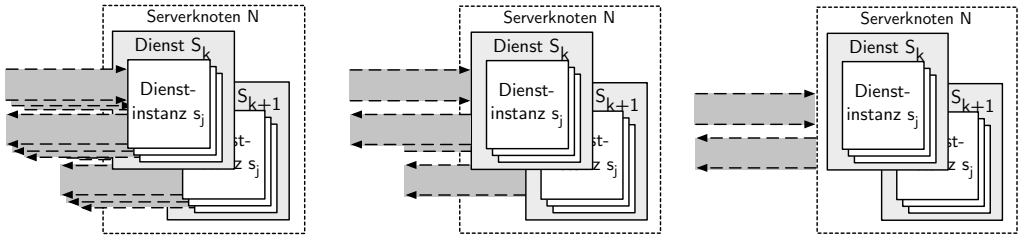
Zuordnung von Strompaaren zu Dienstinstanzen

Bisher wurde lediglich die Zuordnung *eines* Ein- und Ausgabestrompaares $\{NS_I, NS_O\}$ eines Dienstnutzers zu genau *einer* Dienstinstanz $s_{j|k}$ eines Diensttyps S_k und damit ein *Strompaar pro Dienstinstanz* betrachtet. Unabhängig vom gewählten Verarbeitungsmodell (methodenbasiert oder strombasiert) ermöglicht jedoch die differenzierte Zuordnung des Strompaares *eines* Dienstnutzers zu einzelnen Dienstinstanzen oder ganzen Dienstinstanzgruppen auf einem Serverknoten N weitere Optimierungen. Im Folgenden werden die drei Zuordnungsmöglichkeiten unterschieden:

1. ein Ein-/Ausgabestrompaar $\{NS_I, NS_O\}$ **pro Dienstinstanz** $s_{j|k}$,
2. ein Ein-/Ausgabestrompaar $\{NS_I, NS_O\}$ **pro Diensttyp** S_k und
3. ein Ein-/Ausgabestrompaar $\{NS_I, NS_O\}$ **pro Serverknoten** N

unterschieden, Diese drei Zuordnungsmöglichkeiten bedingen die Anzahl der insgesamt zu initialisierenden Strompaare und ergeben in Zusammenhang mit dem *strombasierten* Verarbeitungsmodell und dem in diesem Abschnitt bereits diskutierten, *methodenbasierten* Verarbeitungsmodell insgesamt *sechs* Kombinationsmöglichkeiten $K1$-$K6$. Deren Implikationen für den Inhalt des Anfragestromes sowie ihre Vor- und Nachteile werden im Weiteren näher analysiert.

Abbildungen 4.12(a) bis 4.12(c) zeigen die drei Zuordnungsmöglichkeiten im Überblick. Die **erste** Zuordnungsmöglichkeit (Abbildung 4.12(a)) repräsentiert die bisher in dieser Arbeit angenommene Zuordnung, bei der jede Dienstinstanz $s_{j|k}$ eines Diensttyps S_k genau ein Ein- und Ausgabestrompaar $\{NS_{I,j|k}, NS_{O,j|k}\}$ zugewiesen bekommt. Als Inhalt dieses Strompaares werden nur alle Datenelemente eines D_j bzw. D'_j der spezifischen Dienstinstanz übertragen und nur durch diese Instanz verarbeitet. Wird der Strom geschlossen, ist dies auch das Ende der Dienstinstanz und damit das Ende des gemeinsamen Kontexts über die Daten. Den Inhalt eines Eingabestromes pro Dienstinstanz $s_{j|k}$ (1. Zuordnungsmöglichkeit) zeigt Abbildung 4.13(a). Diese Zuordnung von Strompaar zu Dienstinstanzen entspricht einem klassischen Nachrichtenpaar pro Dienstinstanz und führt bei m Diensttypen mit durchschnittlich p Dienstinstanzen zu $m * p$ zu initialisierenden Strompaaren. Die



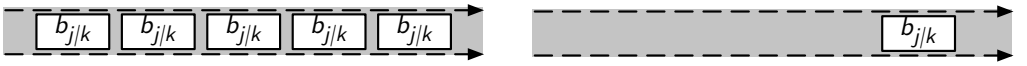
(a) 1. Ein Strompaar pro Dienstinstanz $s_{j|k}$.

(b) 2. Ein Strompaar pro Diensttyp S_k.

(c) 3. Ein Strompaar pro Serverknoten N ($S_{1\ldots m|N}$).

Abbildung 4.12: Zuordnungsmöglichkeiten von Strompaaren zu Diensten.

Ausführung aller p Dienstinstanzen eines Diensttyps S_k erfolgt seriell. Dies verhindert das Überholen einzelner Anfragen beim Dienst und damit potenzielle Dateninkonsistenzen beim Dienstnutzer [30]. Bei der Verwendung des *strombasierten Verarbeitungsmodells* werden alle drei Anforderungen *A2*, *A3* und *A4* unterstützt. Wird hingegen das *methodenbasierte Verarbeitungsmodell* mit dieser Zuordnungsmöglichkeit kombiniert, besteht die Nutzlast aus nur einem Verarbeitungsbucket pro Nachrichtenstrom, welches als Parameter an den Methodenaufruf übergeben wird (Abbildung 4.13(b)). Dies entspricht dem *Einzelnachrichtentransfer*, der Anforderung *A4* nicht unterstützt. Durch den signifikanten Mehraufwand der Strominitialisierung mit deren Warteschlangen gegenüber klassischer Nachrichtenverarbeitung wird zudem Anforderungen *A3* verletzt.



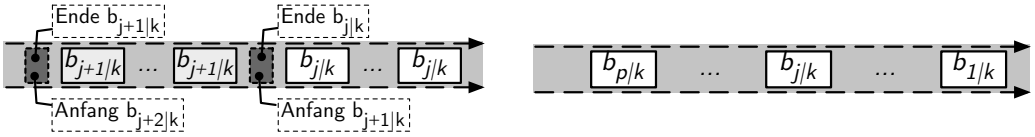
(a) Strombasiertes Verarbeitungsmodell ($K1$).

(b) Methodenbasiertes Verarbeitungsmodell ($K2$).

Abbildung 4.13: Eingabestrominhalt für $\{NS_{I,j|k}, NS_{O,j|k}\} \rightarrow s_{j|k}$.

Die **zweite** Möglichkeit der Zuordnung (Abbildung 4.12(b)) entspricht der Initialisierung eines Ein- und Ausgabestrompaares $\{NS_{I,k}, NS_{O,k}\}$ pro Diensttyp S_k. Dies führt bei m Diensttypen zu m zu initialisierenden Strompaaren zwischen einem Dienstnutzer und einem Serverknoten. Durch die bereits angesprochene serielle Ausführung von Instanzen gleichen Diensttyps pro Dienstnutzer, werden entsprechend auch alle Buckets unterschiedlicher D_j seriell transferiert und verarbeitet. Da bei der vorherigen Zuordnung aus Abbildung 4.12(a) das Schließen des Eingabestromes das Ende der Dienstinstanz s_j markierte und dieses Schließen bei der nun diskutierten Zuordnung entfällt, muss das Ende einer Dienstinstanz nun auf logischer Ebene im Strom signalisiert werden. Mithilfe einer solchen Signalisierung in Form eines *neuen* Buckettyps (`END`) gehören alle Verarbeitungsbuckets nach dieser Markierung zur

logisch nachfolgenden Instanz s_{j+1}. Den Inhalt eines Eingabestromes pro Diensttyp S_k zeigt Abbildung 4.14(a).



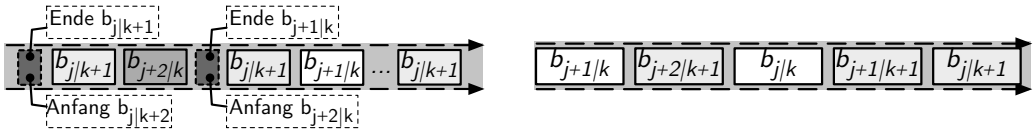
(a) Strombasiertes Verarbeitungsmodell ($K3$). (b) Methodenbasiertes Verarbeitungsmodell ($K4$).

Abbildung 4.14: Eingabestrominhalt für $\{NS_{I,k}, NS_{O,k}\} \rightarrow S_k$.

Diese Signalisierung auf Dienstinstanzebene kann dabei sowohl auf der Initialisierung einer neuen physischen Instanz als auch auf der Zurücksetzung des internen Zustand einer bereits laufenden Dienstinstanz basieren. Letzterer Fall reduziert den Mehraufwand nochmals und entspricht dem Konzept der *stehenden Dienstinstanz*. Da die Annahme gilt, dass alle Dienstaufrufe und damit auch alle Dienstinstanzen eines Dienstnutzers nacheinander ausgeführt werden, ergeben sich aus der Zuordnung nur eines Strompaares pro Dienstnutzer und Diensttyp S_k keine Nachteile in Bezug auf die Anforderungen *A2*, *A3* und *A4*. Bei der Nutzung des methodenbasierten Verarbeitungsmodells entspricht diese Zuordnung der bereits diskutierten Variation, bei der jedes Bucket von einer separaten Dienstinstanz verarbeitet wird (Abbildung 4.14(b)). Einzig Anforderung *A4* bleibt dabei unerfüllt.

Die **dritte** Möglichkeit der Zuordnung (Abbildung 4.12(c)) entspricht der Initialisierung eines einzigen Ein- und Ausgabestrompaares $\{NS_{I,1}, NS_{O,1}\}$ pro Dienstnutzer für alle Dienste $S_{1,\ldots,m|N}$ auf einem Serverknoten N. Bei der Verwendung des strombasierten Verarbeitungsmodells enthält der Strom neben den Datenelementen der seriell ausgeführten Dienstinstanzen eines Diensttyps potenziell auch die Datenelemente der seriell ausgeführten Dienstinstanzen der anderen Diensttypen. Somit muss neben der Signalisierung neuer Dienstinstanzen eines Diensttyps durch den neuen Buckettyp (`END`) auch die Zuordnung einzelner Buckets zu den unterschiedlichen Diensttypen möglich sein. Den Inhalt eines Eingabestromes pro Serverknoten N zeigt Abbildung 4.15(a).

Durch die Nutzung eines einzigen Strompaares kann es beim Transfer von Buckets zum Engpass kommen. Da die Eingabebuckets auf Dienstseite verarbeitet werden



(a) Strombasiertes Verarbeitungsmodell ($K5$). (b) Methodenbasiertes Verarbeitungsmodell ($K6$).

Abbildung 4.15: Eingabestrominhalt für $\{NS_{I,1}, NS_{O,1}\} \rightarrow S_{1\ldots m|N}$.

K	Zuordnung Strompaare zu Dienstinstanzen	Anzahl der Strompaare	Verarbeitungs-modell	Unterstützte Anforderungen
K1	$\{NS_{I,j\|k}, NS_{O,j\|k}\} \rightarrow s_{j\|k}$	m * p	strombasiert	*A2,A3,A4*
K2			methodenbasiert	*A2*
K3	$\{NS_{I,k}, NS_{O,k}\} \rightarrow S_k$	m	strombasiert	*A2,A3,A4*
K4			methodenbasiert	*A2,A3*
K5	$\{NS_{I,1}, NS_{O,1}\} \rightarrow S_{1...m\|N}$	1	strombasiert	*A2,A3,A4*
K6			methodenbasiert	*A2,A3*

m - Anzahl der Diensttypen S mit $S = (S_1, \ldots, S_k, \ldots, S_m)$
p - durchschnittliche Anzahl der Dienstinstanzen s_j pro Diensttyp S_k
mit $S_k = (s_1, \ldots, s_j, \ldots, s_p)$

Tabelle 4.2: Kombinationen K von Verarbeitungsmodellen und Zuordnungen von Strompaaren zu Dienstinstanzen.

müssen, um weitere Buckets durch den Strom zu transferieren, kann der problematische Konvoy-Effekts [25, 128] auftreten. Dabei diktiert die Dienstinstanz mit der längsten Verarbeitungszeit die Verarbeitungszeit aller anderen parallel ausgeführten Instanzen anderer Diensttypen. Bei der Nutzung des methodenbasierten Verarbeitungsmodells ist eine explizite Signalisierung neuer Dienstinstanzen durch den Buckettyp `END` im Strom nicht notwendig, da die implizite Semantik einer neuen Dienstinstanz pro Verarbeitungsbucket gilt (Abbildung 4.15(b)). Die Probleme der Zuordnung von Buckets zu unterschiedlichen Diensttypen sowie des Konvoy-Effektes bleiben bestehen.

Tabelle 4.2 fasst noch einmal die diskutierten Ergebnisse der sechs Kombinationen aus Zuordnungsmöglichkeit und Verarbeitungsmodell zusammen. Je nach Kombination ergibt sich eine unterschiedliche Anzahl an zu initialisierenden Strompaaren sowie eine differenzierte Unterstützung der für diese Arbeit notwendigen Anforderungen. Somit lässt sich festhalten, dass bei der Nutzung des *strombasierten Verarbeitungsmodells* die Kombinationen $K1$ und $K3$ alle notwendigen Anforderungen unterstützen und deshalb präferiert werden. Im Rahmen des *methodenbasierten Verarbeitungsmodells* unterstützt Kombination $K4$ die meisten Anforderungen. Die Kobinationen $K5$ und $K6$ werden aufgrund des auftretenden Konvoy-Effektes und der damit einhergehenden Anpassung aller Dienstinstanzen aller Diensttypen an den Takt der langsamten Dienstinstanz eines Diensttyps nicht weiter betrachtet.

4.5 Evaluierung

Der nachfolgende Abschnitt evaluiert das in diesem Kapitel vorgestellte Konzept des *strombasierten Dienstaufrufes*. Die prototypische Implementierung erfolgte im Rahmenwerk Axis2[1] auf Basis von Java 1.6[2]. Eine detaillierte Vorstellung der Implementierung erfolgt in Kapitel 7.

[1] http://axis.apache.org/axis2/java/core/
[2] http://java.oracle.com

Experimentaufbau

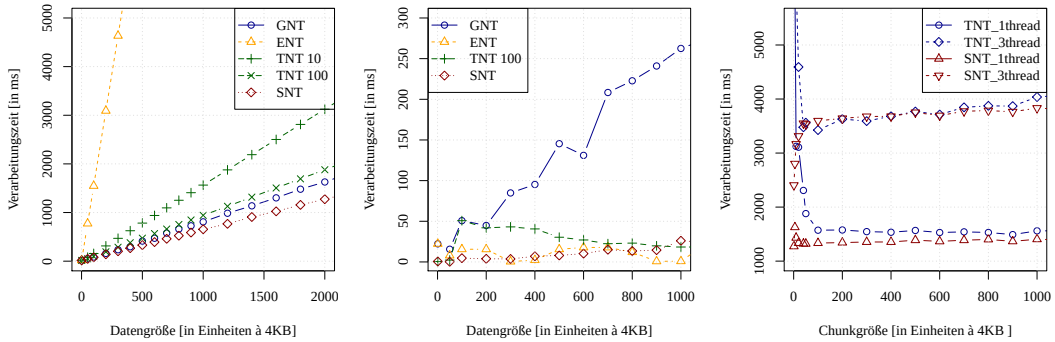
Alle Experimente dieser Arbeit wurden mithilfe von fünf heterogenen Serverknoten realisiert, deren jeweiligen Hard- und Softwareeigenschaften in Tabelle A.1 im Anhang nachgelesen werden können. Für die Experimente dieses Kapitels wurden Knoten A und B verwandt. Beide Rechner sind durch ein Local Area Network (LAN) miteinander verbunden. Jedem Java-Prozess wurde 1 GB RAM als Heapsize zugewiesen.

Alle Experimente basieren auf synthetisch generierten Daten und wurden jeweils 50 mal wiederholt. Die Datenmenge D, die ein Dienstnutzer zur Verarbeitung an die Dienstinstanzen sendet, wird durch eine Menge von Rechnungen d_i aus Abbildung 4.1 mit einer Größe von jeweils 4 KB repräsentiert. Der in den folgenden Experimenten aufgerufene Dienst nimmt die Anfrage bzw. D entgegen, iteriert über alle enthaltenen Rechnungen und gibt diese unverändert wieder an den Dienstnutzer zurück. Beim Aufrufmodell des *strombasierten Dienstaufrufes* beträgt die Kapazität *cap* der Warteschlangen bei Dienstnutzer und Dienstinstanz jeweils 1000 Verarbeitungsbuckets.

Performancemessungen

Abbildungen 4.16(a)-(c) und 4.17(a)-(c) zeigen die Performancemessungen für die zuvor in diesem Kapitel diskutierten Ansätze. Dabei beschreibt n die Anzahl an Datenelementen d_i, welche gesamtheitlich von einer Dienstinstanz verarbeitet werden sollen. Des Weiteren beschreibt c die Anzahl der Datenelemente in einer Teilnachricht (kurz Chunkgröße) bei der Nutzung des *Teilnachrichtentransfers* bzw. die Anzahl der Datenelemente in einem Verarbeitungsbucket bei der Nutzung des *strombasierten Nachrichtentransfers*.

Das erste Experiment in Abbildung 4.16(a) betrachtet die Skalierbarkeit der einzelnen Aufrufmodelle *Gesamtnachrichtentransfer* (`GNT`), *Einzelnachrichtentransfer* (`ENT`), *Teilnachrichtentransfer* (`TNT`), *strombasierter Nachrichtentransfer* (`SNT`) sowie *strombasierter Nachrichtentransfer mit methodenbasierter Ausführung* (`SNT MB`) über die Größe der Datenmenge n. Es werden zudem keine nebenläufigen Prozesse ausgeführt. Dabei lässt sich beobachten, dass die Ausführungszeiten aller Aufrufmodelle im Allgemeinen linear skalieren. `GNT` hat dabei im Vergleich zu den anderen klassischen Aufrufmodellen über alle n die geringsten Ausführungszeiten, ist jedoch nicht in der Lage, eine Datenmenge D von beliebiger Größe zu verarbeiten, da die Datenelemente dann nicht mehr in den zugewiesenen Hauptspeicher der Dienstinstanz passen. Im Gegensatz dazu skaliert `ENT` für unendliche Datenmengen, hat jedoch in Bezug auf die Gesamtausführungsdauer der Verarbeitung aufgrund des Mehraufwandes für die Nachrichtengenerierung und Dienstinstanzinitialisierung für jedes zusätzliche Datenelement d_i einen viel stärkeren Anstieg gegenüber `GNT`. Ähnlich zu `ENT` skaliert `TNT` für unendliche Datenmengen. Je nach Chunkgröße c (`TNT`



(a) Skalierbarkeit über n. (b) Standardabweichung über n. (c) Ausführungszeit über c.

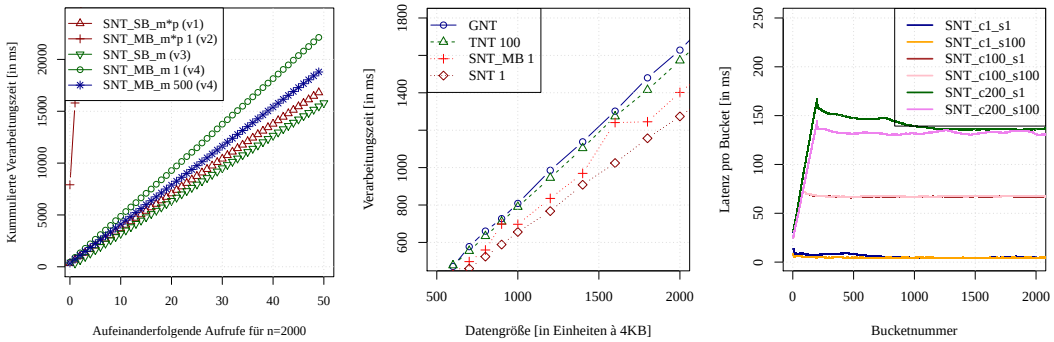
Abbildung 4.16: Performancemessungen I.

`10` mit $c = 10$ und `TNT 100` mit $c = 100$) liegt die Ausführungszeit für D deutlich unter der von `ENT`, da mit steigendem c die Anzahl der Einzelnachrichten an den Dienst sowie die damit verbundenen Dienstinstanziierungen abnehmen, jedoch die Nutzdaten der Teilnachrichten im Gegensatz zum `GNT` im Hauptspeicher einer Dienstinstanz gehalten werden können. Die Ausführungszeiten des in dieser Arbeit vorgestellten Ansatzes des *strombasierten Dienstaufrufes* (`SNT`) liegen für jede Datengröße unter den Ausführungszeiten der anderen hier vorgestellten Aufrufmodelle. Dies liegt zum einen daran, dass effektiv nur eine Nachricht und damit analog zum `GNT` nur eine Dienstinstanz generiert wird. Zum anderen bedingt das nebenläufige Senden und Empfangen, dass die Antwortdaten nicht erst dann zum Dienstnutzer zurücktransferiert werden, wenn alle Eingangsdaten empfangen wurden.

Abbildung 4.16(b) stellt die Standardabweichung der Ausführungszeiten für die einzelnen Aufrufmodelle in Abhängigkeit der Datengröße n dar. Die Standardabweichung von `GNT` erhöht sich mit steigendem n. Dies lässt sich damit erklären, dass die Verarbeitungszeit durch die schrittweise Abarbeitung von Empfang, Verarbeitung und Rücktransport stärker von auftretenden Transferverzögerungen betroffen ist. Im Gegensatz dazu weisen die zu verarbeitenden Pakete des `TNT 100` eine feinere Granularität als bei `GNT` auf und die Transferverzögerungen wirken sich nicht mehr so stark auf die Gesamtverarbeitungszeit aus. Dieser Sachverhalt bestätigt sich bei der Analyse der Kurve des `TNT 100`. Sie zeigt ab einer Datengröße von $n = 200$ eine nahezu konstante Standardabweichung von $50ms$. Jedoch steigt mit diesem Aufrufmodell die Anzahl der neu zu initialisierenden Dienstinstanzen, wodurch sich die diese Standardabweichung erklären lässt. `SNT` weist die geringste Standardabweichung auf. Dies liegt zum einen an der sehr feinen Granularität der Verarbeitungseinheiten und der Inititalisierung nur einer Dienstinstanz für alle n Daten und zum anderen an der Nutzung von Warteschlangen, die als Zwischenspeicher fungieren und Schwankungen in der Netzwerkübertragung einzelner Buckets ausgleichen. Dadurch wirken sich Verzögerungen im Netzwerktransfer nicht direkt auf die Verarbeitungszeit in der Dienstinstanz aus. Dies gilt für den Eingabe- und Ausgabestrom gleichermaßen.

Abbildung 4.16(c) betrachet die Ausführungszeit in Abhängigkeit der Chunkgröße c. Dafür wird n auf 2000 festgelegt und die Chunkgröße c bei `TNT` und `SNT` zwischen 1 und 1000 variiert. `TNT` bei $c = 1$ entspricht dabei `ENT` und `TNT` bei $c = 2000$ entsprechend `GNT`. Zusätzlich werden die Ausführungszeiten von `TNT` und `SNT` bei einer Dienstinstanz (`TNT_1thread`, `SNT_1thread`) und bei drei nebenläufigen Dienstinstanzen (`TNT_3thread` und `SNT_3thread`) gemessen. Dabei wird in Abbildung 4.16(c) deutlich, dass die Ausführungszeit von `TNT` bei einer ausgeführten Dienstinstanz (`TNT_1thread`) mit steigender Chunkgröße c ab $c = 100$ nahezu konstant bei $1600ms$ liegt bzw. bei $c = 900$ bis auf $1514ms$ fällt. Im Gegensatz dazu liegt die optimale Chunkgröße des *TNT* bei drei nebenläufigen Dienstinstanzen (`TNT_3thread`) bei $c = 30$ mit $3419ms$. Die optimale Chunkgröße ist damit stark abhängig von der aktuellen Arbeitslast des Dienstes kann somit nur durch weitere Informationen zum aktuellen Zustand der Dienstausführungsumgebung bestimmt werden. Diese Informationen liegen einem Dienstnutzer jedoch im Allgemeinen nicht vor. Beim strombasierten Nachrichtentransfer `SNT` zeigt sich die Verarbeitungszeit von $n = 2000$ Datenelementen ohne weitere nebenläufige Dienstaufrufe (`SNT_1thread`) nahezu konstant, hat jedoch sein Optimum bei $c = 1$ mit $1267ms$. Werden die Dienstanfragen bei drei nebenläufigen Dienstinstanzen beantwortet (`SNT_3thread`), ergibt die Chunkgröße bei $c = 1$ ebenfalls die kürzeste Ausführungszeit, die jedoch mit steigendem c schnell ansteigt und dann annähernd konstant bei $3700ms$ liegt. Dieses Verhalten lässt sich damit begründen, dass die Verarbeitungsgranularität in Form von einzelnen Datenelementen d_i sehr fein ausfällt, wodurch der Scheduler im Dienstframework die CPU-Ressourcen entsprechend feiner und gleichmäßiger auf die Dienstinstanzen verteilen kann und somit der Gesamtdurchsatz erhöht wird.

Im folgenden Experiment (Abbildung 4.17(a)) werden die Kombinationsmöglichkeiten $K1$ bis $K4$ aus Verarbeitungsmodell und Strompaarzuordnung (vgl. Tabelle 4.2) in ihrem Ausführungsverhalten untersucht und gegenübergestellt. Gemessen werden dabei die Zeiten für 50 aufeinanderfolgende Verarbeitungen von jeweils $n = 2000$ Datenelementen durch *einen* Dienstnutzer. Als Notation beschreibt `SNT_SB` dabei den strombasierten Datentransfer in Verbindung mit dem *strombasierten Verarbeitungsmodell* auf Dienstebene und entspricht damit dem `SNT` vorheriger Experimente. `SNT_MB` beschreibt hingegen den strombasierten Datentransfer in Kombination mit dem *methodenbasierten Verarbeitungsmodell*. $K1$ (`SNT_SB_m*p`) entspricht somit der Basismethode des in diesem Kapitel vorgestellten strombasierten Dienstaufrufes, bei dem ein Strompaar pro Dienstinstanz generiert wird und die zu verarbeitende Datenmenge auf eine Menge an Verarbeitungsbuckets verteilt wird. Das Ausführungsverhalten entspricht dem Verhalten des `SNT` aus Abbildung 4.16(a). Kombination $K2$ (`SNT_MB_m*p 1`) ordnet ebenfalls jeder Dienstinstanz ein Strompaar zu. Da die Ausführungssemantik jedoch eine Dienstinstanz pro Verarbeitungsbucket bedingt, wird pro Strompaar nur ein einziges Bucket transportiert. Dadurch hängt die Anzahl der Strompaare und folglich die Anzahl der Dienstinstanzen von der Aufteilung der $n = 2000$ Datenelemente auf einzelne Verarbeitungsbuckets ab. Es werden insgesamt $\frac{D}{c}$ Strompaare und damit verknüpfte Dienstinstanzen generiert. Die Initialisierung



(a) Skalierbarkeit bei Zuordnung von Strompaaren zu Dienstinstanzen.

(b) Skalierbarkeit des methodenbasierten Aufrufs.

(c) Latenz in Abhängigkeit der Warteschlangengröße q.

Abbildung 4.17: Performancemessungen II.

des Stromtransfers ist mit Mehraufwand für Strom- und Warteschlangeninitialisierungen verbunden, was sich im Experiment bestätigt (`SNT_MB_m*p 1`). Es hat sich weiterhin gezeigt, dass die Erhöhung der Chunkgröße c die Ausführungszeit aller 50 aufeinanderfolgenden Dienstaufrufe erheblich reduziert. $K3$ (`SNT_SB_m`) nutzt in Verbindung mit der strombasierten Verarbeitung ein Strompaar mit einer Dienstinstanz für alle Aufrufe eines Diensttyps. Entsprechend werden die einzelnen Aufrufe nur logisch durch Signalisierungsbuckets im Strom angezeigt und von einer Dienstinstanz als separate logische Dienstinstanzen umgesetzt. Durch die Reduzierung von Strompaaren und Dienstinstanzen zeigt sich eine verkürzte Verarbeitungszeit. Kombination $K4$ (`SNT_MB_m_100`) entspricht der intuitiven Kombination aus strombasiertem Transfer und methodenbasierter Dienstausführung und zeigt gute Performance-Eigenschaften, da nur ein Strompaar für alle Buckets generiert wird und auf Dienstseite jeweils nur eine neue Methode pro Bucket aufgerufen wird. Zudem wird damit das klassische, methodenbasierte Programmiermodell beibehalten.

Abbildung 4.17(b) bewertet die Ausführungszeiten für Kombination $K4$ der methodenbasierten Ausführung im Vergleich zu den bisherigen Aufrufmodellen und zeigt einen Ausschnitt des Graphen. Zum Vergleich der Ausführungszeiten wurden dafür nochmals `GNT` und `TNT 100` abgetragen. Wie zu sehen ist, ermöglicht bereits die Hinzunahme des reinen strombasierten Transfers in Kombination mit dem methodenbasierten Verarbeitungsmodell (`SNT MB 1`) bereits ein verbessertes Ausführungsverhalten gegenüber `GNT` und `TNT 100`. Dennoch bleibt der strombasierte Transfer in Kombination mit dem angepassten strombasierten Verarbeitungsmodell `SNT` weiterhin der effizienteste Ansatz, der zudem als einziger Ansatz alle drei Anforderungen *A2 (Datenaustausch)*, *A3 (Latenz)* sowie *A4 (Korrelation)* unterstützt.

Im letzten der hier dargestellten Experimente werden für den *strombasierten Dienstaufruf* (`SNT`) die Latenzen der einzelnen gesendeten Buckets in Millisekunden ms gemessen. Diese Latenz umfasst die Zeitdauer vom Einfügen eines Buckets in die Sen-

dewarteschlange beim Dienstnutzer bis zur Entnahme des jeweiligen Antwortbuckets aus der Empfangswarteschlange beim Dienstnutzer. Dazu werden die Datenmenge auf $n = 2000$ und die Chunkgröße auf $c = 1$ fixiert. Zudem werden die Kapazitäten *cap* der Sendewarteschlange Q_C beim Dienstnutzer (*client*) mit $c\{1, 100, 200\}$ und die Eingabewarteschlange Q_I der Dienstinstanz (*service*) mit $s\{1, 100, 200\}$ variiert. Dabei zeigt sich, dass eine von 1 (`SNT_c1_s1`) auf 200 (`SNT_c200_s1`) steigende Kapazität von Q_C die Latenz der Buckets maßgeblich negativ beeinflusst. Dies lässt sich damit begründen, dass das Einfügen von Buckets in die Warteschlange erwartungsgemäß schneller vollzogen wird, als der tatsächliche Transfer des Buckets zur Dienstinstanz. Dadurch blockiert die Sendewarteschlange Q_C des Dienstnutzers sobald ihre maximale Kapazität erreicht wird und lässt erst wieder Einfügungen zu, wenn ein Bucket aus der Warteschlange vollständig versendet wurde. Folglich passt sich das Einfügen von Buckets beim Dienstnutzer dem Takt des Netzwerktransfers zum Dienst an. Somit werden noch $cap - 1$ Buckets zur Dienstinstanz übertragen, bevor der physische Transfer des zuletzt in die Warteschlange eingefügten Buckets beginnt. Die Spitzen der Latenzkurven beim Einfügen des *cap*ten Buckets und das nachfolgende Absinken lassen sich mit dem Mehraufwand der Initialisierung der Strominfrastruktur auf Dienstseite beim Versenden des *ersten* Buckets erklären. Diese erhöhte Latenz, die sich auch bei `SNT_c1_s1` ablesen lässt, hat Auswirkungen auf alle nachfolgenden $cap - 1$ in die Warteschlange Q_C eingefügten Buckets, da sich deren Latenzen aufaddieren. Erst beim $cap + 1$ten Bucket übt die erhöhte Latenz des ersten Bucket keinen direkten Einfluss mehr aus. Die Erhöhung der Warteschlangenkapazität *cap* der Eingabewarteschlange Q_I der Dienstinstanz verbessert die Latenz geringfügig. Bei der Festlegung einer Warteschlangengröße gilt es somit zwischen der Latenz einzelner Buckets, dem vorzeitigen Beenden des Einfügens von Buckets in die Warteschlange Q_C und einer Verminderung der Standardabweichung der Verarbeitungszeit der Gesamtdatenmenge (vgl. Abbildung 4.16(b)) abzuwägen.

4.6 Zusammenfassung

In diesem Kapitel wurde der Ansatz des *strombasierten Dienstaufrufes* vorgestellt und evaluiert. Dazu wurden drei Modifikation am traditionellen Aufrufmodell vorgenommen, um die formulierten Anforderungen *A2 (Datenaustausch)*, *A3 (Latenz)* und *A4 (Korrelation)* während der Dienstausführung inhärent zu unterstützen.

Die erste Modifikation definiert eine Anfragenachricht als Datenstrom, in welchem der Nutzdatenteil einer klassischen Nachricht in Form von voneinander unabhängigen Stromobjekten an den Dienst gesendet werden kann. Dabei obliegt es dem Dienstnutzer, die Stromobjekte im Strom zu platzieren und den Strom nach Beendigung des Transfers zu schließen. Darauf aufbauend definierte die zweite Modifikation ein Datenmodell auf dem Nutzdatenteil der strombasierten Nachricht und führte die Abstraktionsschicht der Verarbeitungsbuckets ein. Schlussendlich ermöglicht es die Anpassung des Verarbeitungsmodells innerhalb der Dienstinstanz (Modifikation 3),

die Anwendungsdaten des Anfragestromes als *stehende Dienstinstanz* strombasiert zu verarbeiten.

Alle drei Modifikationen bilden in Kombination den Ansatz des *strombasierten Dienstaufrufes*. Im Rahmen der durchgeführten Experimente konnte gezeigt werden, dass diese Modifikationen

1. zu einem niedrigeren Speicherverbrauch in der Dienstinstanz zur Verarbeitung der Datenelemente und zu einem höheren Durchsatz der einzeln zu verarbeitenden Datenelemente führen (Anforderungen *A2* und *A3*),
2. einen gemeinsamen inhärenten Kontext für alle Datenelemente bei der Dienstinstanz zur Verfügung stellen (Anforderung *A4*) und
3. das Weiterverarbeiten erster Resultate von bereits beim Dienstanbieter verarbeiteten Datenelemente beim Dienstnutzer ermöglichen.

Die explizite Typisierung unterschiedlicher Arten von Verarbeitungsbuckets im Strom ermöglicht die semantische Trennung verschiedenartiger Anwendungsdaten durch die Ausführungsumgebung. Dadurch wurde im Gegensatz zum traditionellen nachrichtenbasierten Aufruf von Diensten die explizite Trennung von Nutzdaten D und Dienstparametern p möglich. Zudem gestattet die explizite Typisierung auch die Zuordnung eines Nachrichtenstrompaares pro Diensttyp und damit die Realisierung des Konzeptes *stehender Dienstinstanzen*.

Neben den aufgezeigten Möglichkeiten zur Dienstparametrierung und zu stehenden Dienstinstanzen, ermöglicht das Konzept der typisierten Verarbeitungsbuckets auch die Erweiterung des Ansatzes um weiterführende Funktionalitäten auf Transferebene, da unterschiedlich typisierte Verarbeitungsbuckets mit ihren Daten im Ausführungssystem entsprechend unterschiedliche Semantiken repräsentieren. Eine Erweiterung um Mechanismen zur Ausfallsicherheit wie beispielsweise in [14, 34, 77, 76] oder die zeitnahe Übertragung von Auslastungsinformationen vom Serverknoten zum Dienstnutzer für eine Analyse der aktuellen Verarbeitungskapazität gestaltet sich dadurch sehr einfach. Diese Mechanismen werden jedoch von den in dieser Arbeit definierten Anforderungen nicht berührt und deshalb nicht weiter betrachtet.

5 Integration von Datenstromsemantik in die Prozessausführung

Das folgende Kapitel betrachtet die Ebene der Prozessausführung und damit die Ebene der kompositen Dienstaktivitäten im Rahmen einer SOA. Bisher können datenintensive Anwendungen anderer Anwendungsklassen wie DIA oder BAM nicht direkt mit SOA-basierten Ausführungsplattformen derzeitiger BPM-Systeme umgesetzt werden. Dieses Grundproblem liegt in der bereits diskutierten schrittweisen, kontrollflussbasierten Ausführung der definierten Prozesse und der damit verbundenen Problematik der Verarbeitung großer Datenmengen innerhalb der Prozessinstanz und bei den externen Dienstpartnern begründet.

Deshalb wird nachfolgend der Ansatz zur *strombasierten Prozessausführung* vorgestellt, der genau dieses Problem überwindet, damit eine effiziente und skalierbare Verarbeitung datenintensiver Prozessanwendungen ermöglicht und die in Kapitel 2 formulierten Anforderungen erfüllt. Der beschriebene Ansatz erlaubt die Konsolidierung der miteinander interagierenden Anwendungsklassen BPM, DIA und BAM auf einer Ausführungsplattform und die Definition und Ausführung von ebenso konsolidierten Anwendungen im Rahmen dienstbasierter Architekturen. Dabei werden die in Abschnitt 3.4 zusammengefassten Konzepte in die Ebene der Prozessausführung integriert.

Das hier vorgestellte Konzept der *strombasierten Prozessausführung* basiert im Grundsatz auf einer partitionierten, pipelinebasierten Verarbeitung von Prozessdaten, ähnlich der Ausführungssemantik der Systeme in den Klassen DIA und BAM. Dies gestattet die Verarbeitung beliebig großer Prozessdaten auf Basis feingranularer Einheiten (Anforderung *A1 (Verarbeitung)*). Die starke Integration dieser Partitionierung in die XML-basierte Dienstkommunikation auf Basis des bereits vorgestellten Konzeptes des *strombasierten Dienstaufrufes* (Kapitel 4) ermöglicht es, beliebige Datenmengen mit externen Diensten auszutauschen (Anforderung *A2 (Datenaustausch)*, *A6 (Dienstsemantik)*) und diese Dienste somit als entfernte Operatoren mit beliebiger Funktionalität zu nutzen. Des Weiteren baut das Konzept auf einem XML-basierten Datenmodell auf, welches die direkte Manipulation dieser Daten mithilfe existierender Sprachen ermöglicht (Anforderung *A5 (Datenmodell)*). Schlussendlich reduziert die Integration der Semantik *stehender Prozessinstanzen* den Mehraufwand zur Initialisierung neuer Prozessinstanzen (Anforderung *A3 (Latenz)*) und ermöglicht einen nachrichtenübergreifenden Kontext in den einzelnen Aktivitäten des Prozessgraphs (Anforderung *A4 (Korrelation)*).

Die folgenden Abschnitte stellen die einzelnen Aspekte der strombasierten Prozessausführung detailliert vor. Dabei wird zunächst das zugrundeliegende Datenmodell als eine Erweiterung des strombasierten Datenmodells aus Kapitel 4.3 eingeführt, auf dem das im Anschluss beschriebene Prozessmodell und dessen Operatoren basieren. Des Weiteren gilt es, die Parametrierung der Operatoren in einer Prozessinstanz zu diskutieren und die Integration des strombasierten Dienstaufrufes in das Prozessmodell zu beschreiben.

5.1 Datenmodellerweiterung

Bei der Verarbeitung großer Datenmengen in einer Prozessinstanz begrenzt der einer Instanz zur Verfügung stehende Hauptspeicher die maximale Größe der verarbeitbaren Daten. Im Kontext strukturierter, baumbasierter Datenformate wie XML verstärkt sich dieser Effekt aufgrund der speicherinternen Darstellung, sodass der Speicherbedarf eines XML-Dokumentes je nach enthaltenen Datentypen bis zu zehn mal größer ist als dessen textuelle XML-Repräsentation [108, 152].

Aus diesem Grund bestehen die folgenden beiden Anforderungen an das zu konzipierende Prozessdatenmodell, um eine konsolidierte Ausführungsumgebung für die drei beschriebenen Anwendungsklassen zu ermöglichen:

1. Unterstützung der nativen Repräsentation strukturierter Daten auf Basis der XML-Semantik von *XML Infoset* (Anforderung *A5 (Datenmodell)*).
2. Unterstützung grundlegender Datenoperationen für die strombasierte Verarbeitung von Datenmengen auf Basis von Datenpartitionierung und pipelinebasierter Verarbeitungsausführung in den Prozessaktivitäten (Anforderung *A1 (Verarbeitung)*).

Grundlage des Prozessdatenmodells bildet die Erweiterung des *strombasierten Datenmodells* der Dienstebene (vgl. Kapitel 4.3) und seine Eigenschaften. Für das Prozessdatenmodell werden analog zum *strombasierte Datenmodell* Verarbeitungseinheiten b_p (engl. *processing buckets*), kurz Prozessbuckets bzw. Buckets, definiert.

Diese Prozessbuckets existieren innerhalb einer Prozessinstanz und abstrahieren von der zu verarbeitenden und aus n gleichstrukturierten Datenelementen bestehenden Datenmenge D. Sie untergliedern D damit in n physische, verarbeitbare Teilmengen, welche zwischen den Operatoren weitergegeben werden. Dabei wird vorausgesetzt, dass die Größe eines Verarbeitungsbuckets die Hauptspeicherkapazität eines Operators nicht übersteigt und somit die Verarbeitung eines solchen Buckets in jeder Aktivität fehlerfrei möglich ist.

Definition 11 Prozessdatenmodell: *Ein Verarbeitungsbucket b_p ist definiert als ein XML-Element b_p mit $b_p = (d, t, ex, pt)$, wobei d einen eindeutigen Indentifikator für dieses Bucket beschreibt, t den Buckettyp definiert, ex eine erweiterbare Liste an*

Attributen darstellt und p_t die eigentliche Nutzlast in Abhängigkeit des Buckettyps als Kinderelemente eines Buckets beschreibt. Der Basisbuckettyp $b_{p,data}$ annotiert die Nutzdaten des Buckets als Anwendungsdaten des Prozesses analog zum Basisbuckettyp des strombasierten Dienstaufrufes und wird im Weiteren mit `data` *gekennzeichnet.*

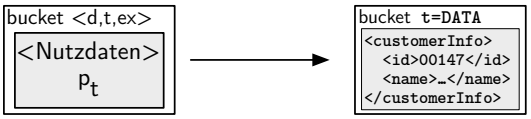
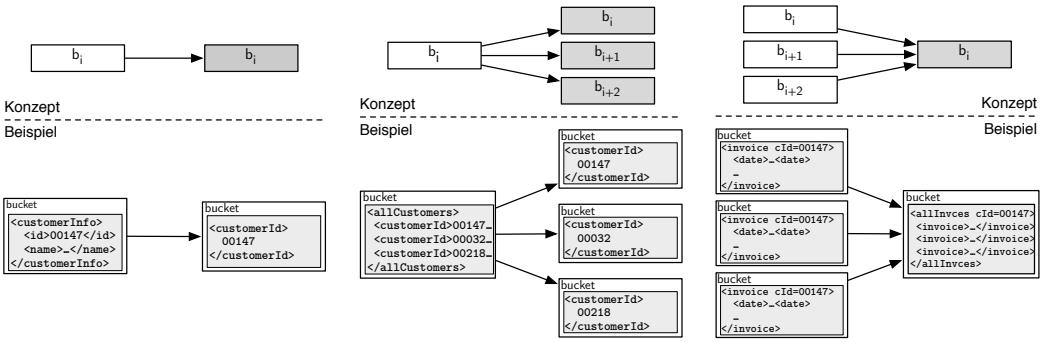


Abbildung 5.1: Verarbeitungsbucket b_p des Prozessdatenmodells.

Abbildung 5.1 zeigt ein solches Verarbeitungsbucket mit einer Kundeninformation als Beispiel. Ein Verarbeitungsbucket b_p kann dabei entweder den vollständigen Datensatz D_m einer Eingangsnachricht m mit $D_m == p_t$ beinhalten, der dann entlang der Operatorkette des Prozessplans verarbeitet wird, oder nur jeweilige Untermengen von D_m mit $p_t \subseteq D_m$ transportieren. Ersteres ist der Fall, wenn D_m dem Prozess bei dessen Start initial übergeben wird und die Operatoren die Größe der Datenmenge D_m jeweils fehlerfrei verarbeiten können. Übersteigt die Datengröße von D_m jedoch die von einer Aktivität verarbeitbare Größe, muss das Datenmodell die Partitionierung des Datensatzes auf eine feingranularere Menge von Verarbeitungseinheiten $b_{p,i}$ und damit eine Untermenge von D_m mit $p_{t,i} \subseteq D_m$ unterstützen.

Abbildung 5.2 zeigt die drei grundlegenden *Datenoperationen*, die für die feingranulare Verarbeitung beliebiger Datenmengen zur Unterstützung von Anforderung *A1 (Verarbeitung)* auf Ebene des Datenmodells unterstützt werden. Diese Datenoperationen spiegeln sich später in den Operatoren des Prozessmodells wider und werden in Abbildung 5.2 jeweils *konzeptionell* dargestellt sowie mit *Beispieldaten* des Prozesses „*Top n Kunden*“ (Abbildung 2.11, Seite 25) visualisiert. Im Gegensatz zu den Eingabe- / Ausgabebeziehungen der Dienste in Kapitel 4 (vgl. Abbildung 4.11, Seite 69), welche nur das von außen beobachtbare Dienstverhalten beschreiben, widmen sich die nun vorgestellten Operationen konkreten Datenoperationen auf den Eingabedaten. Dabei lassen sich die drei Operationen direkt auf die Eingabe- / Ausgabebeziehungen der Dienste in Abbildung 4.11 abbilden. Eine Schlussfolgerung von Ein- Ausgabebeziehungen der Dienste auf eine konkrete, beim Dienst implementierte Datenoperation ist nicht möglich, da aus Sicht des Dienstnutzers ein Dienst einen Blackbox-Charakter besitzt, der die interne Anwendungslogik versteckt.

Die erste Operation (Abbildung 5.2(a)) bildet die *Modifikation* der Anwendungsdaten p_t eines Prozessbuckets b_i. Das Bucket und die Granularität der enthaltenen Datenelemente werden dabei nicht verändert. Abbildung 5.2(a) zeigt als Beispiel die Aktivität *a1 (transform)* des DIA-Prozesses, bei der die Kundennummer aus der Kundeninformation extrahiert wird, um diese als Nutzdateninhalt im Bucket zu ersetzen.



(a) Modifikation (1:1). (b) Aufteilung (1:N). (c) Aggregation (N:1).

Abbildung 5.2: Semantische Datenoperationen des Datenmodells.

Abbildung 5.2(b) zeigt die zweite Operation und beschreibt die *Aufteilung* der Nutzdaten p_t eines Buckets b_i auf eine Menge B neuer Buckets. Im Beispiel wird eine Menge an Kundennummern auf neue Buckets aufgeteilt, was wiederum die feingranulare Verarbeitung der einzelnen Kundenummern in nachfolgenden Operatoren erlaubt. Diese Operation stellt somit die eigentliche Partitionierung der Anwendungsdaten für eine skalierbare Datenverarbeitung dar.

Die dritte Operation (Abbildung 5.2(c)) realisiert die Aggregation von Verarbeitungsbuckets, in dem die Nutzdaten aus einer Menge an Eingangsbuckets B in den Nutzdatenbereich p_t eines Buckets b_i integriert werden. Diese Operation wird von Aggregations- und Gruppierungsfunktionen in zustandsbehafteten Operatoren realisiert. Das Beispiel in Abbildung 5.2(c) zeigt die Gruppierung bzw. den Verbund aller Einzelrechnungen einer Kundennummer in eine gemeinsame Verarbeitungseinheit in Aktivität *a5 (Verbund der Daten)*. Aus Gründen der Übersichtlichkeit wurden die Bestellungen in der Abbildung nicht dargestellt.

Da die Nutzdaten p_t eines Verarbeitungsbuckets b auf *XML InfoSet* basieren und somit inhärent Anforderung *A5 (Datenmodell)* unterstützen, könnten XPath-, XQuery- und XSLT-Ausdrücke genutzt werden, um Anwendungsdaten in p_t zu modifizieren, neue Daten in p_t zu erstellen oder eine Datenmenge zu aggregieren. Deshalb erfolgt die Abbildung der Datenoperationen des Datenmodells auf das XQuery Data Model (XDM) [211] und damit auf die Anfrage-Modifikationssprache XQuery. Ein Hauptmerkmal des XDM bildet neben der Definition von validen Datentypen, wie beispielsweise Elementknoten, Textknoten, Datumswerten oder Integerwerten, die Definition der Ergebnismenge als Sequenz von Objekten valider Datentypen. Diese Sequenz enumeriert die Ergebnisobjekte und kann wiederum als Eingabe für nachfolgende XPath/XQuery-Ausdrücke dienen.

Die konkrete Abbildung des Datenmodells auf das XDM besteht darin, dass die Anzahl der Resultate einer Ergebnissequenz die Anzahl der daraus zu generieren-

den Buckets definiert. Bei der Operation der *Modifikation* (Abbildung 5.2(a)) wird alternativ die Nutzung von XML-Transformationsausdrücken wie XSLT oder STX unterstützt.

Das hier eingeführte Datenmodell bildet die Grundlage für das nachfolgende Prozessmodell, indem es von den eigentlich zu verarbeitenden Daten abstrahiert und auf Ebene von Prozessbuckets notwendige Basisoperationen (vgl. Abbildung 5.2), Datenformate sowie Sprachkonstrukte für die Bucketverarbeitung festlegt.

5.2 Prozessmodell

Nach der Definition des Datenmodells beschreibt dieser Abschnitt das darauf aufbauende Prozessmodell mit seiner zugrundeliegenden Verarbeitungssemantik, vordefinierten Operatortypen und der damit einhergehenden Integration des *strombasierten Dienstaufrufes* (Kapitel 4).

5.2.1 Ausführungssemantik

Zur Unterstützung der sechs in dieser Arbeit definierten Anforderungen für eine skalierbare Ausführungsumgebung wird die kontrollflussbasierte Ausführungssemantik bisheriger dienstbasierter Prozesse durch ein datenflusszentrisches Verarbeitungsmodell ersetzt. Das datenflusszentrische Verarbeitungsmodell bildet in Zusammenhang mit dem bereits vorgestellten Datenmodell die Grundlage für die Umsetzung der in dieser Arbeit definierten Anforderungen *A1* bis *A6*. Ein *strombasierter Prozessplan* zur Ausführung konsolidierter Anwendungen auf Basis einer datenflusszentrischen Ausführungssemantik wird wie folgt definiert:

Definition 12 *Ein strombasierter Prozessplan P_S ist definiert als gerichteter, azyklischer Graph (DAG) mit $P_S = (C, O, Q, S)$. Dabei beschreibt C den Prozesskontext, $O = (o_1, \ldots, o_i, \ldots, o_l)$ die Menge an Operatoren als Knoten im Graphen, $Q = (q_1, \ldots, q_j, \ldots, q_m)$ die Menge an Warteschlangen zwischen den Operatoren und somit die Kanten im Graphen und S die Menge an Diensten, mit der der Prozess interagiert. Weiterhin wird ein Operator o definiert als $o = (q_I, q_O, f, p)$ mit q_I als Menge an eingehenden Warteschlagen und q_O als Menge an ausgehenden Warteschlangen. f beschreibt die Funktion des Operators (bzw. des Aktivitätstyps in Anlehnung an traditionelle Workflow-Sprachen), die auf alle eintreffenden Daten angewandt wird, und p die Menge an Parametern, die f bzw. den Operator (re-)konfiguriert. Zunächst bedingt eine Eingangsnachricht m_i genau eine Prozessinstanz p_i mit $m_i \rightarrow p_i$ und $p_i = (C_i, O_i, Q_i, S_i)$.*

Alle Operatoren $o_j \in O_i$ einer Prozessinstanz p_i werden unabhängig voneinander und nebenläufig ausgeführt. Abbildung 5.3 zeigt zwei aufeinanderfolgende Operatoren o_j

und o_{j+1}, die durch eine Warteschlange miteinander verbunden sind und durch ihre Parameter p_j und p_{j+1} konfiguriert werden. Da der Operator o_{j+1} die Daten seines Vorgängers o_j verarbeitet, muss die Struktur bzw. das XML-Schema der Nutzdaten p_t eines Verarbeitungsbuckets b in der Warteschlange q_i der erwarteten Struktur des Operators o_{j+1} entsprechen. Warteschlangen sind dabei konzeptionell und zur Ausführungszeit an keine operatorspezifischen Datenschemata gebunden. Die daraus resultierenden multiplen Ausgabestrukturen (pro Operator) erhöhen die Flexibilität der Datenflüsse zwischen den Operatoren. In der Phase der Prozessmodellierung (Designzeit) kann jedoch pro Warteschlange eine Menge an Datenschemata annotiert werden, welche die Validierung der Ein- und Ausgabestrukturen der einzelnen Operatoren während der Modellierungsphase ermöglicht und eine fehlerfreie Prozessausführung unterstützt.

Für die Parametermenge p werden die Anfrage- und Modifikationssprachen XPath und XQuery bzw. XSLT und STX-Skripte unterstützt, die bereits für das Datenmodell definiert wurden. Die spezifische Art der Parametermenge und deren Ausprägungen werden durch den spezifischen Operatortyp bzw. dessen Funktion f definiert.

Analog zu Abbildung 3.1 (Seite 40), welche die *kontrollflussbasierte* Architektur einer Prozessinstanz des Prozesses *Top n Kunden* darstellt, zeigt Abbildung 5.4 die Architektur der pipelinebasierten Version dieser Prozessinstanz. Da der Datenfluss zwischen den Operatoren explizit modelliert wird, wurde der implizite, variablenbasierte Datenfluss entfernt. In Abhängigkeit der Komplexität des Prozesses ergeben sich dabei Änderungen zwischen den Operatorgraphen der beiden Verarbeitungsmodelle und den darin verwendeten Operatoren/Aktivitäten. Im Beispiel des Prozesses *Top n Kunden* wird beispielsweise zur parallelen Verarbeitung der Pfade der zusätzliche Operator *copy* benötigt, der jede Verarbeitungseinheit für den zweiten Operatorpfad dupliziert.

5.2.2 Operatortypen

Wie bereits im vorherigen Abschnitt beschrieben, besteht ein Operator aus Eingabewarteschlangen q_I, Ausgabewarteschlangen q_O, einer definierten Funktion f, welche auf die Nutzdaten der eintreffenden Verarbeitungsbuckets angewandt wird, sowie einer dazugehörigen Parametermenge p, welche f konfiguriert. Zur Modellierung und Ausführung von grundlegenden Prozessfunktionen der drei Anwendungsklassen

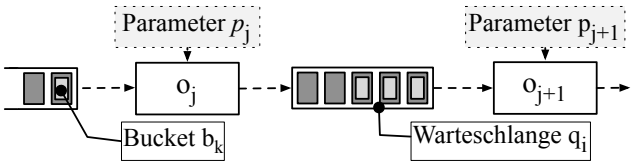


Abbildung 5.3: Operatoren und Warteschlange mit Prozessbuckets.

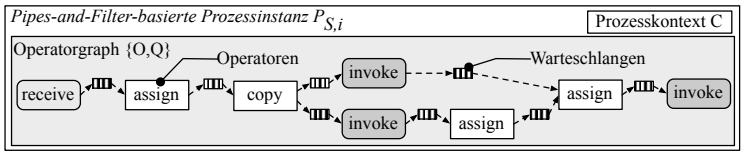


Abbildung 5.4: Pipelinebasierter Operatorgraph einer Prozessinstanz $P_{S,i}$.

werden unterschiedliche Funktionen f als Operatortypen vordefiniert. Diese stehen einer Prozessinstanz zunächst als lokale Operatoren zur Verfügung. Durch die direkte Integration von Dienstaufrufen in die Prozessausführung über einen dedizierten Operatortyp kann der Prozess zudem um beliebige Funktionen erweitert werden.

Auf Grundlage der Spezifikation der Prozessbeschreibungssprache BPEL sowie der Operatorklassifikation für datenzentrische Integrationsprozesse im Kontext von Enterprise Application Integration (EAI) [26, 28] werden vier Klassen von Operatortypen definiert. Neben den Klassen der *interaktionsorientierten* Operatoren, *kontrollflussorientierten* Operatoren und *datenflussorientierten* Operatoren wird als vierte Klasse zusätzlich die Klasse der *ereignisorientierten* Operatoren definiert. Die Operatoren aller Klassen arbeiten auf der Granularität der Verarbeitungsbuckets b.

Weiterhin können die Operatortypen nach [153] anhand der Anzahl ihrer Eingangswarteschlangen klassifiziert werden. Diese Klassifikation wird für die spätere Integration von *strombasierten Dienstaufrufen* in das Prozessmodell benötigt. Dabei wird zwischen *unären* Operatoren, d.h. Operatoren mit *einer* Eingabewarteschlange und einer Ausgabewarteschlange, und *binären* Operatoren mit *zwei* Eingabewarteschlangen und einer Ausgabewarteschlange unterschieden. Im Folgenden werden die einzelnen Klassen kurz beschrieben und die Vertreter dieser Klassen tabellarisch zusammengefasst. Dabei wird jeder Operatortyp durch seinen Namen, die mögliche Anzahl von Ein- und Ausgabewarteschlangen q_I und q_O, durch eine kurze Funktionsbeschreibung sowie durch die notwendigen Parameter beschrieben.

Interaktionsorientierte Operatoren

Die Operatoren dieser Klasse (vgl. Tabelle 5.2) beinhalten Funktionen zur Interaktion eines Prozessplans P_j bzw. seiner Prozessinstanzen p_i mit der umgebenden Infrastruktur. Dabei beschreibt der Operatortyp `invoke` die Integration beliebiger Funktionen über synchrone sowie asynchrone Dienstaufrufe [114, 205]. Die Nutzung der Operatoren `receive` und `reply` bestimmt das Kommunikationsmuster (synchron oder asynchron), das eine Prozessinstanz von P nach außen anbietet.

Beispielhaft wird im Folgenden die Funktionalität des `receive`-Operators beschrieben und in Abbildung 5.5 dargestellt. Eine Eingangsnachricht m_j eines Dienstnutzers

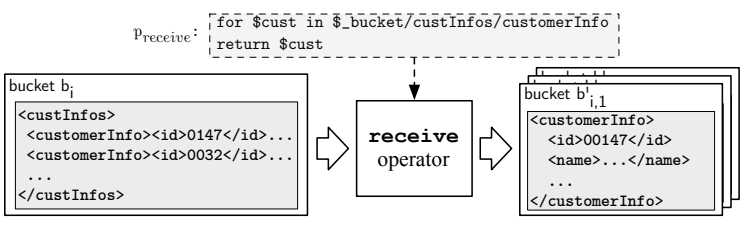


Abbildung 5.5: Semantik des `receive`-Operators.

Operatortyp	q_I	q_O	Funktion f	Parameter p
Receive	0	1	Startpunkt des Prozessgraphen und Verbindung zu den Eingangsdaten des Prozessaufrufes. Ermöglicht Partitionierung der eintreffenden Daten über XQuery-Ausdruck.	$expr_{xquery}$
Reply	1	0	Endpunkt des Prozessgraphen. Sendet Antwortbuckets an aufrufende Komponente zurück.	–
TInvoke	1	1	Realisiert aktive, traditionelle Kommunikation mit entfernter Dienstkomponente S_j (vgl. Abschnitt 5.2.4).	Endpunkt E_{S_j} Parameter $p_{f_{S_j}}$
SInvoke	1	1	Realisiert aktive, strombasierte Kommunikation mit entfernter Dienstkomponente S_j (vgl. Abschnitt 5.2.4).	Endpunkt E_{S_j} Parameter $p_{f_{S_j}}$

Tabelle 5.2: Interaktionsorientierte Operatoren.

Operatortyp	q_I	q_O	Funktion f	Parameter p
Copy	1	2...n	Kopiert Eingabebuckets physisch und verteilt sie auf die ausgehenden Warteschlangen. Realisiert damit parallele Prozessflüsse.	–
Route	1	2...n	Realisiert inhaltsbasierte Verteilung von Buckets auf die Ausgabewarteschlangen entsprechend einer definierten Bedingung.	$expr_{xpath}$
Signal	1	0	Möglichkeit, einen kritischen Zustand innerhalb der Prozessinstanz zu signalisieren. Pausiert Prozessausführung.	$expr_{xpath}$

Tabelle 5.3: Kontrollflussorientierte Operatoren.

mit einer Menge an Kundeninformationen initialisiert eine neue Prozessinstanz p_j. Die Nutzdaten dieser Nachricht werden als Verarbeitungsbucket b_i an den `receive`-Operator gegeben, welcher den Prozess startet und die Daten an die nachfolgenden Operatoren weiterleitet. Durch die Parametrisierung $p_{receive}$ in Form eines XQuery-Ausdrucks, wird eine XDM-Sequenz generiert, die jede Kundeninformation als separates Element beinhaltet. Folglich wird jede Kundeninformation in einem neuen Bucket $b_{i,l}$, mit l als der Anzahl der in der XDM-Sequenz enthaltenen Elemente, platziert und an die nachfolgenden Operatoren weitergegeben. Damit wird die Datenoperation der *Aufteilung* (vgl. Abbildung 5.2(b)) unterstützt.

Kontrollflussorientierte Operatoren

Die Operatoren dieser Klasse (vgl. Tabelle 5.3) beinhalten Funktionen, die den zeitlichen Ablauf und die Struktur des strombasierten Prozesses bestimmen. Dazu gehören die Realisierung paralleler Ausführungsblöcke, die bedingte, inhaltsbasierte

Operatortyp	q_I	q_O	Funktion f	Parameter p
Filter	1	1	Verwirft Eingabebuckets, die bei der Auswertung des angegebenen XPath-Ausdrucks eine leere Elementsequenz zurückgeben.	$expr_{xpath}$
Groub By	1	1	Aggregiert Werte der Eingangsbuckets. Entspricht Datenoperation der *Aggregation*.	$expr_{xpathToGrp}$, $expr_{xpathToAgg}$, $expr_{AggFunc}$, $expr_{having} \cup expr_{xquery}$
Join	2	1	Verbundoperator für parallele Datenflüsse. Entspricht Datenoperation der *Aggregation*.	$expr_{xquery}$
Order By	1	1	Sortiert Verarbeitungsbuckets anhand eines definierten Attributes.	$expr_{attr}$, $expr_{asc\|desc}$
Split	1	1	Realisiert Datenoperation der *Aufteilung*. Partitioniert Eingabebuckets anhand der Ergebnissequenz des angegebenen XQuery-Ausdrucks. Entspricht der Semantik des `receive`-Operators.	$expr_{xquery}$
Transform	1	1	Stellt die Datenoperation der *Modifikation* bereit, indem die Nutzdaten eines Buckets durch einen XQuery-Ausdruck oder eine XSLT-Anweisung verändert werden.	$expr_{xquery} \cup script_{xslt\|stx}$
Union	2	1	Führt parallele Datenflüsse zusammen. Entspricht UNION-ALL Semantik aus SQL [113].	–

Tabelle 5.4: Datenflussorientierte Operatoren.

Weiterleitung von Verarbeitungseinheiten sowie die Möglichkeit zur Fehlerbehandlung.

Datenflussorientierte Operatoren

Die Operatortypen dieser Klasse (vgl. Tabelle 5.4) ermöglichen die lokale Ausführung datenzentrischer Funktionen. Dabei wurden grundlegende Funktionen der DIA-Anwendungsklasse sowie der dienstbasierten Prozesse im Kontext von EAI-Anwendungen [26] definiert. Funktionen dieser Klasse ermöglichen die effiziente Verarbeitung großer Datenmengen und beinhalten damit auch Vertreter der im Datenmodell bereits definierten Operationen *Modifikation*, *Aufteilung* und *Aggregation*. Die Verarbeitung von unendlichen Datenmengen in den zustandsbehafteten Operatoren `Join`, `GroupBy` und `OrderBy` wird zunächst nicht zugesichert, da sie in den ursprünglichen Anwendungsklassen nicht vorgesehen war. Diese Verarbeitung kann jedoch durch die Ergänzung der Operatorparameter um Zeitfensterdefinitionen erweitert werden werden.

Operatortyp	q_I	q_O	Funktion f	Parameter p
Arithmethic	2	1	Stellt dem Datenstrom die Grundrechenarten bereit. Notwendig für die Berechnung von KPIs.	$expr_{xpathToId}$, $expr_{xpathToAgg}$, $expr_{arithm}$, $expr_{relSymbol}$
Sequence	1	1	Überprüft die Reihenfolge der Buckets nach einem festgelegten Muster.	$expr_{sequence}$ $expr_{xpathToId}$
Window	1	1	Gibt jedem Bucket ein Gültigkeitsintervall und beschneidet somit zeitlich und damit mengenmäßig den zu analysierenden Nachrichtenstrom. Es existieren unterschiedliche Arten von Fensteroperatoren [1, 10, 96].	$expr_{xpathTmstp}$, $expr_{duration}$, $expr_{timeunit}$, $expr_{starttime}$

Tabelle 5.5: Ereignisorientierte Operatoren.

Ereignisorientierte Operatoren

Die Operatoren dieser Klasse (vgl. Tabelle 5.5) beschreiben die Funktionen für die Analyse von Nachrichtenströmen und der darin enthaltenen Ereignismuster aus der Anwendungsklasse BAM. Im Speziellen ermöglichen die Operatoren dieser Klasse die Kontrolle von Ereignismustern, die Berechnung von Kennzahlen (KPI) und deren Vergleich. Der Operatortyp `Window` begrenzt dabei den Blick auf den durchfließenden Daten- bzw. Nachrichtenstrom zeitlich oder inhaltlich und erlaubt es somit, Operatoren anderer Operatorklassen auf diesen Strom anzuwenden.

Die Summe aller hier vorgestellten Operatoren ermöglicht die konsolidierte Ausführung von Anwendungen der einzelnen Anwendungsklassen BPM, DIA und BAM sowie die gemeinsame Modellierung konsolidierter Anwendungen der Klassen BPM und DIA.

Nachdem nun alle vordefinierten Operatortypen in diesem Kapitel kurz beschrieben wurden, beschäftigt sich der nachfolgende Abschnitt mit der Umsetzung der Parametrierung dieser Operatoren zur Modellierungs- und Ausführungszeit eines Prozessplanes und seiner Prozessinstanzen.

5.2.3 Prozessparametrierung

Wird ein Prozess in seiner Entwicklungsphase definiert, erfolgt dies auf Ebene des Prozessplans. Erst in der Phase der Ausführung werden die eigentlichen Prozessinstanzen initialisiert und abgearbeitet. Somit müssen auch die Parameter bereits im Rahmen der Entwicklungsphase definiert werden und zur Ausführungszeit entsprechend gültig sein. In diesem Kontext werden folglich zwei Arten der Parametrierung unterschieden (vgl. Abbildung 5.6):

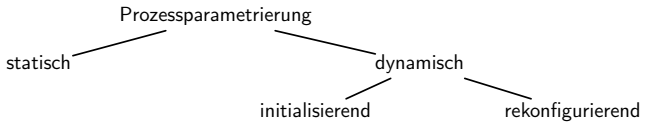


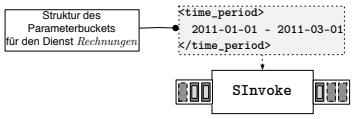
Abbildung 5.6: Arten der Prozessinstanzparametrierung.

Statische Parametrierung

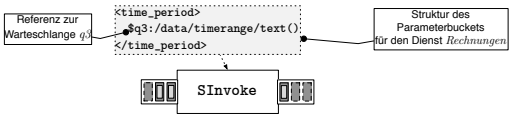
Diese Form der Parametrierung beschreibt den Fall, dass die Parameter der einzelnen Operatoren bereits in der Prozessbeschreibung während der Entwicklungsphase mit statischen Werten hinterlegt werden. Folglich ändern sich die Parameter der Operatoren während der Gesamtlaufzeit aller Instanzen nicht. Eine solche Parametrierung benötigt zur Laufzeit keine weitere Unterstützung auf Ebene der Prozessausführung. Abbildung 5.7(a) stellt ein Beispiel einer statischen Parameterdefinition im Kontext des Szenariodienstes *Rechnungen* dar. Dabei wird die Struktur des beim Dienst definierten Parameterbuckets b_{param} (`t=PARAM`) mit dem Zeitraum der gültigen Rechnungen als statischer Wert hinterlegt (in der Beispielabbildung das erste Quartal 2011). Beim Start der Prozessinstanz wird daraufhin der Anfragestrom zum strombasierten Dienst *Rechnungen* etabliert, der hinterlegte Parameter als Parameterbucket in den Strom platziert und der Dienst damit konfiguriert.

Dynamische Parametrierung

Diese Art der Parametrierung beschreibt den Fall, dass sich Werte des Parameters erst zur Laufzeit des Prozesses aus den Daten ergeben und damit in der Entwicklungsphase nur referenziert werden können. Betrachtet man die kontrollflussbasierte Prozessausführung klassischer BPM-Prozesse, so wird diese Referenzierung über globale Prozessinstanzvariablen, auf die jeder Operator der Prozessinstanz beliebig zugreifen kann, realisiert. Die Notwendigkeit, dass die von einem Operator o_j benötigten Daten im direkt vorangegangenen Prozessschritt o_{j-1} erstellt werden müssen und damit im Eingangsdatenstrom von o_j anliegen, entfällt damit. Jedoch muss die Prozessmodellierung bei der Nutzung globaler Prozessvariablen sicherstellen, dass



(a) Statische Operatorparametrierung.



(b) Dynamische Operatorparametrierung.

Abbildung 5.7: Operatorparametrierungen.

nur solche Daten von einem Operator referenziert werden, welche bereits initialisiert bzw. erstellt wurden. Die Möglichkeit dynamischer, initialer Parameter bieten auch existierende Systeme der DIA- und BAM-Anwendungsklassen [177]. Dabei ist es möglich, jede Prozessinstanz mit spezifischen Parametern für ihre Initialisierung aufzurufen. Aus diese Parameter kann daraufhin während der gesamten Laufzeit der Instanz nur lesend zugegriffen werden. Alle anderen dynamischen Werte von Operatorparametern müssen im Eingabedatenstrom des jeweiligen Operators anliegen.

Im Kontext der hier vorgestellten *strombasierten Prozessausführung* existieren keine Prozessvariablen einer kontrollflussbasierten Ausführung im klassischen Sinne, da der Datenfluss explizit zwischen den Operatoren stattfindet. Die genannte Parametrierungsmöglichkeit der DIA- und BAM-Systeme ist möglich, beschränkt jedoch die Flexibilität und Effizienz des hier vorgestellten Ansatzes. Deshalb erfolgt die Referenzierung von Laufzeitdaten in Operatorparametern über die Spezifikation der Warteschlange, aus welcher die Daten benötigt werden (vgl. Abbildung 5.7(b)). Dabei wird vorausgesetzt, dass Operatoren nur Daten aus Warteschlangen referenzieren, die im vorangehenden Prozessfluss liegen. Auch wird davon ausgegangen, dass die XML-Strukturen der Nutzdaten einzelner Prozessbuckets innerhalb *einer* Warteschlange gleich sind. Referenziert ein Operator in seiner Parameterdefinition ein Datum in einer vorangehenden Warteschlange, wird diese Referenz mit der entsprechenden Operator-ID in der Warteschlange registriert. Abbildung 5.8 zeigt das Beispiel aus Abbildung 5.7(b) mit zwei Operatoren und zwei Warteschlangen im Detail. Dabei referenziert Operator $o5$ Daten aus Warteschlange $q3$, weshalb er in $q3$ registriert wurde. Warteschlange $q4$ wird hingegen von keinem Operator referenziert.

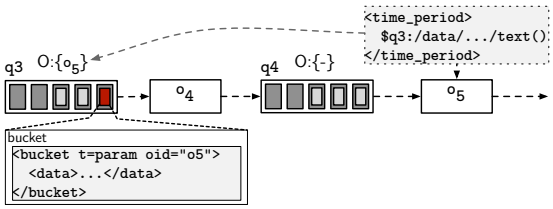


Abbildung 5.8: Referenzierung und Registrierung.

Initiale Parametrierung Zur Realisierung dieser dynamischen Parametrierung auf Prozessebene wird das Datenmodell analog zur Dienstebene um den Buckettyp der Parameterbuckets b_{param} (`t=param`) erweitert. Für eine initiale Parametrierung der Operatoren einer Prozessinstanz kopiert eine Warteschlange für jede bei ihr registrierte Operator-ID den Inhalt des *ersten* Datenbuckets, welches in die Warteschlange eingefügt wird, in den Nutzdatenbereich eines Parameterbuckets, vermerkt die Operator-ID im Parameterbucket und stellt das Parameterbucket dem eigentlichen Datenbucket in der Warteschlange voran (vgl. Abbildung 5.8). Der referenzierte Operator entnimmt das Parameterbucket anhand der Operator-ID aus dem Strom

und setzt den Wert entsprechend. Alle anderen Operatoren ignorieren das spezifische Parameterbucket und leiten es ohne Verarbeitung direkt an die nachfolgenden Warteschlangen weiter.

Referenziert ein Operator in seinem Parameter mehr als eine Warteschlange oder befinden sich referenzierte Warteschlangen vor *binären* Operatoren wie beispielsweise **Join** oder **Union**, so warten *binäre* Operatoren im Rahmen der Prozessinitialisierung darauf, dass an beiden eingehenden Warteschlangen mindestens ein Parameterbucket anliegt, bevor sie mit der Verarbeitung der eigentlichen Datenbuckets beginnen. Dadurch gewährleisten sie die priorisierte Propagierung der Parameterbuckets entlang der Operatorkette für die Initialisierung aller Operatoren. Referenziert kein Operator Daten in einer spezifischen Warteschlange, wird für diese Warteschlange dennoch ein initiales, aber leeres Parameterbucket generiert, um die Funktionsweise der binären Operatoren zu gewährleisten. In diesem leeren Parameterbucket belegt die Operator-ID einen negativen Wert.

Rekonfigurierbare Parametrierung Im Rahmen von Rekonfigurationen der Operatorparameter zur Laufzeit einer Prozessinstanz gelten zunächst die Sachverhalte der *initialen Parametrierung*. Die Semantik der Parameterbucket-Generierung wird in den Warteschlangen jedoch dahingehend erweitert, dass Parameterbuckets in erneut generiert werden, wenn sich der entsprechende, in den Operatorparametern referenzierte Wert eines Buckets im Vergleich zum vorherigen Bucket ändert. Durch die FIFO-basierte Propagierung erreicht das neue Parameterbucket den referenzierten Operator und überschreibt den entsprechenden Wert. Alle nachfolgenden Datenbuckets werden daraufhin mit den aktualisierten Operatorparametern verarbeitet. Dies erhöht die Flexibilität einzelner Prozessinstanzen und reduziert zudem die Notwendigkeit, laufende Prozessinstanzen aufgrund falscher Operatorparametrierungen zu terminieren und durch neue Prozessinstanzen zu ersetzen

Die Analyse der Beispielprozesse zeigt jedoch, dass die Operatoren in den Prozessdefinitionen eine Mischung aus statischen und dynamischen Parametern beinhalten. Schlussendlich liegt es im Ermessen des Prozessdesigners, die verschiedenen Arten der Parametrierung und ihre Vorteile für die individuellen Prozessdefinition anzuwenden.

5.2.4 Kommunikation mit externen Diensten

Die bereits vordefinierten Operatortypen in Abschnitt 5.2.2 bieten nur grundlegende Funktionen der Datenverarbeitung und Flusskontrolle. Die eigentlichen, anwendungsspezifischen Funktionen sollen über externe Dienstkomponenten in die Prozesspläne integriert werden. Zu diesen Funktionalitäten zählen beispielsweise die Bereitstellung von Datenquellen und Datensenken oder auch spezifische Analysealgorithmen, anhand deren Ergebnissen automatisierte Entscheidungen getroffen werden sollen. Die Interaktion mit diesen externen Dienstkomponenten soll dabei dienst-

orientiert auf Basis einer XML-basierten Nachrichtenkommunikation erfolgen. Der folgende Abschnitt beschreibt daher die Integration von klassischen und strombasierten Dienstaufrufen in die Prozessausführung mithilfe der den beiden Operatortypen `TInvoke` und `SInvoke` zugrundeliegenden Konzepten.

Integration klassischer Dienstaufrufe

Die Integration klassischer Dienstaufrufe erfolgt über den Operatortyp `TInvoke`. Die zugrundeliegende Semantik besteht darin, für jedes in den Operator eintreffende Bucket b_i eine Anfragenachricht R mit der Nutzlast p_t zu erstellen und den im Operator hinterlegten Dienst aufzurufen. Die Antwortnachricht R' wird daraufhin erwartet, ihre Nutzlast in einem neuen Ausgabebucket b'_i verpackt und in die dem `TInvoke`-Operator folgende Warteschlange eingefügt. Somit ergibt sich eine Eingabe-Ausgabebeziehung der Verarbeitungsbuckets von 1 : 1. Abbildung 5.9 zeigt den `TInvoke`-Operator am Beispiel des Dienstes *Rechnungen*, in dem für jede Kundennummer `customerId` eine Anfrage R_i an den Dienst gesandt wird und alle Rechnungen des Kunden in einer Antwortnachricht R_i zurückgesandt werden. Dabei muss das Eingabebucket b_i der vom Dienst erwarteten Struktur entsprechen und im Falle des Rechnungsdienstes auch die Zeitspanne `time_period` beinhalten, für welche die Rechnungen zurückgegeben werden sollen.

Die zurückgegebene Datenmenge darf dabei die maximal zulässige Größe für ein Bucket nicht überschreiten, da die Datenmenge im Ausgabebucket als ein Datum integriert wird . Eine Ausnahme bildet die Modellierung des `TInvoke`-Operators im Zusammenhang mit einem darauf folgenden `Split`-Operator. Dabei wird die `Split`-Funktionalität zur Ausführungszeit direkt im `TInvoke`-Operator auf die Textrepräsentation von R' angewandt. Hierdurch entstehen feingranulare Dateneinheiten in Form von Buckets zur nachfolgenden Verarbeitung (vgl. Abbildung 5.10).

Entgegen dem Vorteil der Integration bereits existierender Dienstimplementierungen, bedingt die Nutzung klassischer Dienste im Kontext der strombasierten Prozessausführung die folgenden Nachteile: Zum einen führt die in der Prozessinstanz p_i präferierte feingranulare Partitionierung der Verarbeitungsdaten zu einer Vielzahl von Buckets b_i und damit auch zu einer Vielzahl von einzelnen Dienstaufrufen bei der Nutzung des `TInvoke`-Operators (vgl. Abbildung 5.10) Der dadurch entstehende signifikante Mehraufwand bei *Einzelnachrichtentransfer* wurde bereits im

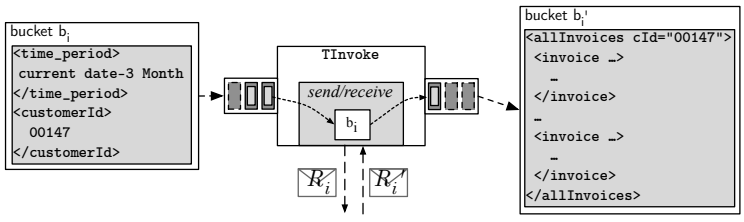


Abbildung 5.9: Funktionsweise des `TInvoke`-Operators.

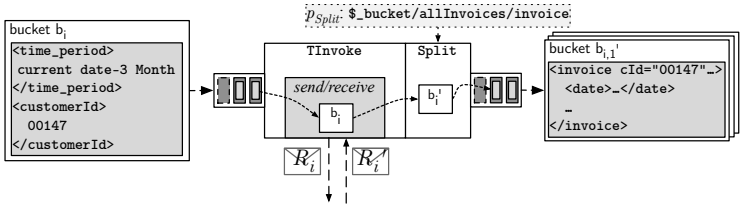


Abbildung 5.10: Funktionsweise des `TInvoke`-Operators in Kombination mit `Split`.

vorherigen Kapitel diskutiert und steht im Gegensatz zum Vorteil der effizienten partitionierten Verarbeitung innerhalb der Prozessinstanz. Zum anderen kann die Notwendigkeit bestehen, zum Zwecke der Korrelation alle auf eine Menge an Prozessbuckets verteilten Daten in einer Dienstinstanz gemeinsam zu verarbeiten. Eine vorherige Bündelung bzw. *Aggregation* ist dafür unabdingbar, was sowohl die Datenmenge der Anfragenachricht R als auch die Datenmenge in der Antwortnachricht R' weiter erhöht und somit gegebenenfalls schnell zur Überschreitung der maximal verarbeitbaren Größe eines Prozessbuckets führt.

Integration strombasierter Dienstaufrufe

Die theoretische Herleitung und die experimentelle Evaluierung in Kapitel 4 haben gezeigt, dass das Konzept des strombasierten Dienstaufrufes die Anforderungen *A2 (Datenaustausch)*, *A3 (Latenz)* und *A4 (Korrelation)* auf Dienstebene erfüllt.

Über den Operatortyp `SInvoke` erfolgt die Integration des strombasierten Dienstaufrufes in die Prozessausführung. Dazu instanziiert der Operatortyp in einer Prozessinstanz einen Anfragestrom NS_I und einen Antwortstrom NS_O zu dem entsprechenden strombasierten Dienst und platziert alle ankommenden Buckets direkt im Anfragestrom NS_I. Zudem entnimmt der Operator die Antwortbuckets direkt aus dem Antwortstrom NS_O und platziert sie in der ausgehenden Warteschlange.

Die Parametrierung der Dienste erfolgt über die Nutzung des bereits eingeführten Buckettyps b_{param} (`param`), welcher ebenfalls direkt im Anfragestrom NS_I platziert wird und die Daten für die Konfiguration der Funktion f enthält. Dabei werden analog zur Prozessebene nur diejenigen Parameterbuckets verarbeitet, die der ID des Diensttyps S entsprechen. Parameterbuckets mit einer anderen Ziel-ID sowie Buckettypen, welchen nicht dem Buckettypen b_{data} (`data`) angehören, werden bei ihrer Entnahme aus dem Anfragestrom direkt wieder in den Ausgabestrom des Dienstes eingefügt. Auf Seiten des `SInvoke`-Operators werden diese „fremden“ Buckets mit den normalen Antwortbuckets des Dienstes entnommen und somit in korrekter zeitlicher Reihenfolge an die nachfolgenden Operatoren propagiert.

Abbildung 5.11 zeigt den Operatortyp `SInvoke` mit den Beispieldaten des bereits bekannten, nun strombasierten Dienstes *Rechnungen*. Dabei platziert der Operator

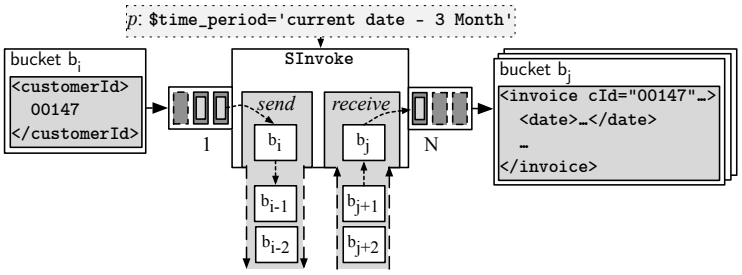


Abbildung 5.11: Funktionsweise des `SInvoke`-Operators.

zunächst ein Parameterbucket p mit `time_period='current date-3 Month'` in den Anfragestrom und konfiguriert somit f. Nachfolgend fügt der Operator jede eintreffende Kundennummer in den Anfragestrom ein. Der Parameter p garantiert, dass pro Kundennummer nur Rechnungen der letzten drei Monate zurückgegeben werden. Je nach Dienstimplementierung werden die zur Kundennummer gefundenen Rechnungen entweder in einem gemeinsamen Bucket oder jede Rechnung in einem separaten Bucket zum `SInvoke`-Operator transportiert. Letzteres entspricht Abbildung 5.11 und spiegelt die in Abschnitt 4.4.2 beschriebenen Ein-/ Ausgabebeziehung $1 : N$ auf Ebene der Prozessoperatoren wider.

Ein Vergleich der Signatur strombasierter Dienste aus Definition 10 (Seite 65) mit der Signatur der Operatoren aus Definition 12 (Seite 85) offenbart, dass diese sich in ihrer Grundstruktur ähneln. Da sowohl der Anfragestrom NS_I als auch der Antwortstrom NS_O auf beiden Seiten durch FIFO-Warteschlangen mit der Ausführungslogik verbunden werden, ist die Semantik dieser Ströme äquivalent zur Semantik der im Prozessmodell definierten Warteschlangen $q_i \in Q$. Dadurch ändert sich die Signatur eines strombasierten Dienstes zu $S = \{id, q_I, q_O, f, p\}$ und entspricht somit der Signatur von lokalen Operatoren o im Prozessmodell. Somit gilt

$$S(id, q_I, q_O, f, p) == o(id, q_I, q_O, f, p).$$

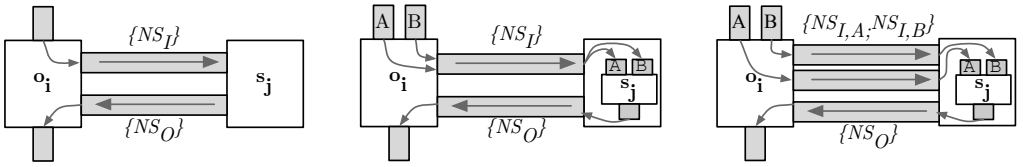
Ein strombasierter Dienst S entspricht damit einem entfernt ausgeführten Operator o. Beide, Dienst S und Operator o, werden durch einen Identifikator id, eine eingehende Warteschlange q_I, eine ausgehende Warteschlange q_O, eine Funktion f und deren Parameter p beschrieben. Die strikte Unterscheidung zwischen Dienst und Operator wird dadurch in der Phase der Prozessmodellierung vernachlässigbar. Im Sinne der effizienten Verarbeitung ist jedoch eine lokale Ausführung eines Operators dem entfernten Aufruf im Allgemeinen vorzuziehen. Für speicher- und CPU-Intensive Aufgaben ermöglicht jedoch der entfernte Aufruf die Verteilung von Operatoren auf weitere Serverknoten und führt damit zur Erweiterung der zur verfügung stehenden Hardwareressourcen.

Klassifikation und Anwendbarkeit entfernter Operatoren

Um beliebige Operatortypen mithilfe strombasierter Dienste in das Prozessmodell zu integrieren, wird im Folgenden die Abbildung von unären und binären Operatoren auf das strombasierte Aufrufmodell diskutiert (Abbildung 5.12(a)-(c)).

Abbildung unärer Operatoren: Unäre Operatoren lassen sich direkt auf das strombasierte Aufrufmodell und damit auf einen strombasierten Dienst abbilden, da ein solcher Dienst jeweils genau einen Eingabe- und Ausgabestrom besitzt (vgl. Abbildung 5.12(a)). Durch die Entkopplung des Sendens und Empfangens der jeweiligen Anfrage- und Antwortbuckets werden die Eingabe- und Ausgabebeziehungen $1:1$, $1:N$ und $N:1$ direkt unterstützt, wie bereits in Abbildung 5.11 zur Funktionsweise des `SInvoke`-Operators dargestellt wurde.

Abbildung binärer Operatoren: Da ein strombasierter Dienst nur einen Eingabestrom NS_I pro Dienstinstanz anbietet, ist eine direkte Abbildung auf binäre Operatoren nicht möglich. Ein erster, naiver Ansatz einen binären Operator auf einen strombasierten Dienst abzubilden besteht darin, alle Buckets beider Eingabewarteschlangen des Operators in den einen existierenden Anfragestrom NS_I einzufügen und auf Seiten der Dienstinstanz den jeweiligen Eingabewarteschlangen des Operators zuzuordnen (vgl. Abbildung 5.12(b)). Obwohl dieser Ansatz keine Änderung im eigentlichen Aufrufmodell impliziert, können damit Verbund-Operatoren, wie beispielsweise Sort-Merge-Joins [53, 66, 133] oder Pipelined-Hash-Joins [59, 74, 79, 87], die eine strombasierte Verarbeitung erlauben, nur bedingt abgebildet werden. So kann es bei ungleicher Ankunftsrate zweier zu verbindender Bucketströme A und B in den beiden Eingabewarteschlangen beim Operator passieren, dass der Verbundalgorithmus in der entfernten Dienstinstanz aufgrund der FIFO-Semantik des Anfragestromes NS_I keinen Verbundpartner aus B findet, da alle Buckets von A den Anfragestrom und die damit verbundenen Warteschlangen blockieren. Der Ansatz zur Vermeidung dieses Problems liegt in der Initialisierung zweier separater Anfrageströme $NS_{I,A}$ und $NS_{I,B}$ pro Dienstinstanz s_j (vgl. Abbildung 5.12(c)). Dies verhindert das Problem des Verhungerns (engl. *starvation*) bei binären Opera-



(a) Abbildung von unären Operatoren auf S.

(b) Abbildung von binären Operatoren auf S.

(c) Abbildung von binären Operatoren durch Erweiterung der Aufrufsemantik.

Abbildung 5.12: Abbildung der Operatortypen auf strombasierte Dienstinstanz.

toren, führt jedoch zur Notwendigkeit einer zustandsbehafteten Korrelation beider Anfrageströme $NS_{I,1}$ und $NS_{I,2}$ pro Dienstinstanz und zu einem komplexeren Kommunikationsprotokoll.

Zusammenfassend lässt sich festhalten, dass die Integration traditioneller und strombasierter Dienste mit den vorgestellten Konzepten effizient vollzogen wird. Strombasierte Dienste ermöglichen zudem die Erweiterung des Prozessplans um beliebige Funktionen als entfernte Operatoren. Dadurch können verteilte Hardwareressourcen erschlossen werden. Klassische Dienstfunktionen im Rahmen des strombasierten Dienstaufrufes können als unäre Operatoren aufgefasst und mit den derzeitigen Konzepten direkt in den Prozessfluss integriert werden. Dies bildet die Grundlage für die Unterstützung der Anforderungen *A2 (Datenaustausch)* sowie *A6 (Dienstsemantik)*.

5.3 Ausführungsoptimierungen

Der folgende Abschnitt analysiert und diskutiert Implikationen und Optimierungsmöglichkeiten der bisher vorgestellten strombasierten Prozessausführung, um den sechs in Kapitel 2 formulierten Anforderungen noch besser gerecht zu werden. Die Optimierungsansätze gliedern sich dabei in zwei Abschnitte. Im Abschnitt der *Intra-Prozess-Optimierungen* werden Möglichkeiten zur Performance-Verbesserung *innerhalb* einer Prozessinstanz analysiert. Im Gegensatz dazu betrachet der Abschnitt der *Inter-Prozess-Optimierungen* Möglichkeiten zu Performance-Verbesserungen zwischen *aufeinanderfolgenden* Prozessausführungen.

5.3.1 Intra-Prozess-Optimierungen

Da das Datenmodell der strombasierten Prozessausführung die feingranulare Partitionierung von Nutzdaten erlaubt, skalieren die damit realisierten Prozessanwendungen theoretisch für beliebig große Datenmengen. Allerdings hängt die Skalierbarkeit der Prozesse von der Größe der Nutzdaten in den einzelnen Prozessbuckets sowie von der Summe aller Prozessbuckets in einer Prozessinstanz ab. Da die Warteschlangen zwischen den Operatoren das Einfügen neuer Buckets blockieren, solange sie vollständig befüllt sind, wird die maximale Anzahl p von Buckets in der Prozessinstanz durch die Summe der Kapazitäten cap aller Warteschlangen $q_i \in Q$ definiert. Bei gleicher Warteschlangenkapazität aller $q_i \in Q$ ergibt sich p mit

$$p = cap * |Q|.$$

Da die Nutzlastschemata der Buckets in den einzelnen Warteschlangen q_i zur Modellierungszeit des Prozessplans bekannt sind, kann der durch die Buckets entstehende, maximale Speicherverbrauch pro Prozessinstanz errechnet werden.

Aufgrund der extensiven Nutzung der Warteschlangen, wird die Veränderung der Kapazität *cap* der einzelnen Warteschlangen als Optimierungsmöglichkeit betrachtet. So halten Prozessinstanzen mit kleinen Warteschlangenkapazitäten weniger Prozessbuckets im Hauptspeicher. Der von der Instanz benötigte maximale Hauptspeicher wird reduziert. Allerdings sind die Instanzen damit anfälliger gegen Kommunikationslatenzen einzelner Buckets bei der Nutzung von `Invoke`-Operatoren. Im Gegensatz dazu benötigen Prozessinstanzen mit größeren Operatorwarteschlangen mehr Hauptspeicher. Dies erhöht die Anzahl der Buckets, die zwischen den Operatoren in den Warteschlangen gepuffert werden. Bei der Verarbeitung endlicher Datenmengen ermöglicht es schnellen Operatoren am Anfang der Prozesskette ihre Verarbeitung zeitiger zu abzuschließen und ihre Ressourcen freizugeben, während nachfolgende langsame Operatoren die Daten noch verarbeiten. Dies reduziert den Konvoy-Effekt für einzelne Operatoren. Bei abschließender Verarbeitung eines Buckets am Ende der Operatorkette wird es verworfen und gibt den eingenommenen Hauptspeicherplatz frei. Dieses einfache Verhalten kann durch existierende Ansätze wie einer intelligenten Speicherverwaltung mit Möglichkeiten zur Wiederverwendung [68] verbessert werden.

Eine weitere Implikation für eine skalierbare Verarbeitung großer Eingangsdatenmengen D bildet der verzögerte Aufbau (engl. *deferred building*) der dazugehörigen Hauptspeicherstruktur im `receive`-Operator einer Prozessinstanz. Dabei wird die eintreffende Nutzdatenmenge D zunächst im Rohformat auf einem internen, persistenten Speicher abgelegt und nur schrittweise auf Granularität des im `receive`-Operator angegebenen XQuery-Ausdrucks gelesen und im Speicher aufgebaut. Den Nachteil dieses Ansatzes bildet die Einschränkung möglicher XQuery-Ausdrücke, da die Daten lediglich vorwärtsgerichtet gelesen und verarbeitet werden können.

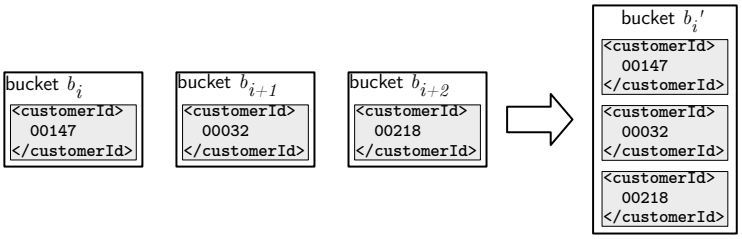


Abbildung 5.13: Intra-Prozess-Optimierung durch Bucketreduzierung.

Des Weiteren ist die Größe der Nutzdaten in einem Bucket stark vom XQuery-Ausdruck und von der Struktur der Eingabedatenmenge D abhängig. Bisher gilt, dass ein Prozessbucket b_i genau ein Daten- bzw. Verarbeitungselement d_i als Nutzdatum $p_{t,i}$ von b_i enthält und somit vereinfacht

$$p_{t,i} == d_i \text{ bzw. } b_i == d_i$$

angenommen wird. Bei Aufhebung dieser strikten Beschränkung und bei Trennung der logischen Verarbeitungseinheiten d_i von deren physischen Repräsentation b_i mit

$$b_i = \{d_k, \ldots, d_l\},$$

kann die Zusammenfassung mehrerer Bucketnutzdaten p_t und somit mehrerer d in nur ein physisches Bucket b_i die Prozessausführung verbessern (vgl. Abbildung 5.13 am Beispiel der logischen Verarbeitungseinheiten *Kundennummern*). Während dabei die logische Trennung der einzelnen Nutzdaten in den Operatoren beibehalten wird, befinden sich weniger physische Buckets in der Prozessinstanz, was zu einer Reduzierung des erforderlichen Speichers und der notwendigen Synchronisation zwischen den Threads der Operatoren führt. Es gilt zu analysieren, ob diese Art der Optimierung eine Laufzeitverbesserung ermöglicht und wenn ja, wie viele Nutzdaten in einem Bucket zusammengefasst werden sollten. Nähere Ausführungen dazu finden sich in Abschnitt 5.5.

5.3.2 Inter-Prozess-Optimierungen

Derzeit wird die strombasierte Ausführung nur auf Ebene einer Prozessinstanz (engl. *intra-process-level*) angewandt. Somit werden alle Prozessbuckets, die durch eine Prozessinstanz fließen, in den Operatoren mit einem gemeinsamen Kontext verarbeitet. Dies ermöglicht die effiziente Ausführung der bereits genannten DIA- und BAM-Anwendungen.

Aufeinanderfolgende Datenmengen, die in getrennten Kontexten verarbeitet werden sollen, werden bisher auch in separaten, aufeinanderfolgenden Prozessinstanzen abgearbeitet. Durch die Instrumentalisierung von Punktuationen [17, 151], die ursprünglich dazu entwickelt wurden, blockierende Algorithmen auch in Datenstromsystemen zu verwenden, können aufeinanderfolgende, separate Prozesskontexte auf logischer Ebene im Datenfluss einer Prozessinstanz angezeigt werden. Diese Punktuationen ermöglichen die Verarbeitung der Datenmengen *aller* aufeinander folgenden Prozessaufrufe in *einer* physischen Prozessinstanz, deren Operatoren jedoch die angestrebte Kontexttrennung realisiert (vgl. Abbildung 5.14). Eine solche Prozessinstanz wird im Weiteren analog zu *stehenden Anfragen* im Bereich der Datenstromsysteme auch *stehende Prozessinstanz* bezeichnet.

Die Realisierung der Kontextpunktuationen aus Abbildung 5.14 führt zum einen dazu, dass eine nachfolgende Datenmenge D_{j+1} durch die Instanz der vorherigen Datenmenge D_j verarbeitet wird und damit der Mehraufwand einer separaten Instanzgenerierung für D_{j+1} entfällt. Zum anderen kann die Verarbeitung von D_{j+1} bereits begonnen werden, während die Prozessinstanz noch D_j verarbeitet. Diese Tatsache kann sich jedoch auch negativ auf die Laufzeit der einzelnen (nun logischen) Prozesse auswirken.

Um die Trennung einzelner Verarbeitungskontexte im Datenfluss der Prozessinstanz anzuzeigen, wird das *Datenmodell* um den Prozessbucketyp `SEPCTX` erweitert. Ein solches Punktuationsbucket wird zwischen dem letzten Bucket von D_j und dem ersten Bucket von D_{j+1} in den Datenstrom eingefügt. Da sich die Metadaten zwischen aufeinanderfolgenden Prozessaufrufen und damit zwischen den aufeinanderfolgenden Prozesskontexten unterscheiden, besitzt das Punktuationsbucket `SEPCTX` eine vordefinierte Nutzdatenstruktur, in der die Metadaten zum Prozessaufruf der Datenmenge D_{j+1} gespeichert werden. Zu den Metadaten gehören der gesamte Inhalt des SOAP-Nachrichtenkopfes und damit beispielsweise die Anfrage-ID des Aufrufes bzw. der Rückgabeendpunkt bei einem synchronen Prozessaufruf.

Neben der Erweiterung des *Datenmodells* um den neuen Buckettyp, muss das *Prozessmodell* angepasst werden, damit die Operatoren die Kontextpunktuationen verstehen und die Kontexte von aufeinanderfolgenden Datenmengen auf logischer Ebene trennen. Im Allgemeinen entnimmt ein Operator ein Punktuationsbucket aus seiner Eingabewarteschlange, setzt seinen internen Zustand zurück und propagiert das Punktuationsbucket an jede ausgehende Warteschlange. Dies impliziert auch, dass der `copy`-Operator das Punktuationsbucket aus der Warteschlange löst, es entsprechend der Anzahl der ausgehenden Warteschlangen kopiert und an diese weiterleitet.

Binäre Operatoren wie beispielsweise `Join` und `Union` sowie der strombasierte`SInvoke`-Operator benötigen weiterführende Anpassungen. Da ein binärer Operator nebenläufige Datenflüsse zusammenführt, ist dieser Operator gezwungen, den Konsum aus einer Eingabewarteschlange A anzuhalten, wenn er ein Punktuationsbucket darin entdeckt. Der Operator muss daraufhin solange Buckets aus der Eingabewarteschlange B konsumieren, bis er auch darin auf ein Punktuationsbucket stößt. Daraufhin entfernt der Operator beide Punktuationen aus den Eingabewarteschlangen A und B und propagiert eines der beiden Buckets an seine ausgehende Warteschlange.

Für den Operator `SInvoke` sind zwei Möglichkeiten denkbar, auf eine Kontextpunktuation zu reagieren. Wenn der entfernte Dienst keine Kontextpunktuationen unterstützt, muss der Operator das Strompaar $\{NS_I, NS_O\}$ beenden, als ob alle Daten verarbeitet wurden und der Operator herunterfährt. Wurden alle verbliebenen Buckets aus dem Antwortstrom NS_O des Dienstes empfangen und an die ausgehende Warteschlange des `SInvoke`-Operators gegeben, wird auch das Punktuationsbucket in der ausgehenden Warteschlange platziert. Daraufhin gilt es, einen neuen Anfra-

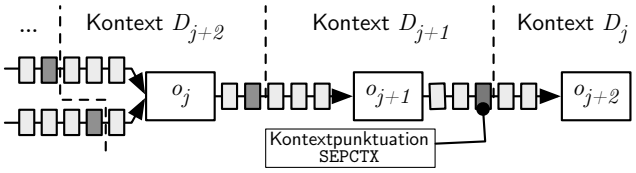


Abbildung 5.14: Inter-Prozess-Optimierung durch Kontextpunktuationen.

gestrom zum Dienst zu instanziieren und damit einen neuen Verarbeitungskontext beim Dienst zu initialisieren.

Wenn die Dienstimplementierung hingegen Kontextpunktuationen über den Buckettyp `SEPCTX` unterstützt, kann das Strompaar beim Eintreffen eines Punktuationsbuckets geöffnet bleiben, sodass das Punktuationsbucket gemeinsam mit allen anderen Prozessbuckets in den Anfragestrom zum Dienst platziert werden kann. Dabei setzt der Dienst beim Empfang des Punktuationsbuckets seinen inneren Zustand analog zu den lokalen Prozessoperatoren selbstständig zurück und gibt die Punktuation im Antwortstrom an den `SInvoke`-Operator zurück. Beide Reaktionen auf den Empfang des Punktuationsbuckets im `SInvoke`-Operator werden im Rahmen der anwendungsinvarianten Evaluation in Abschnitt 5.5 miteinander verglichen.

Die in diesem Abschnitt vorgestellten prozessinternen und prozessübergreifenden Optimierungsansätze zeigen die allgemeinen Optimierungsrichtungen für eine erhöhte Performance und Effizienz der Ausführungsumgebung und lassen sich im Detail weiter verfeinern. So wird beispielsweise das Konzept der *stehenden Prozessinstanz* auch im nachfolgenden Kapitel 6 als Grundlage der Prozessausführung genutzt.

5.4 Modellierung und Abbildung

In diesem Abschnitt werden die Definition neuer Prozesse sowie die Abbildung bereits vorhandener Prozessdefinitionen der drei Anwendungsklassen BPM, DIA und BAM auf die hier vorgestellte Prozessausführungsumgebung kurz diskutiert.

Datenintegration und Datenanalyse (DIA)

Die Abbildung der Anwendungsklasse (DIA) auf das hier vorgestellte strombasierte Verarbeitungsmodell gestaltet sich zunächst einfach, da DIA-Prozessdefinitionen die Modifikation der Daten innerhalb eines Datenflussgraphen inhärent beschreiben. Die Prozessdefinition kann dabei sowohl auf Grundlage einer graphischen Modellierung als auch auf Basis der Übersetzung deklarativer Datenmanipulationen wie beispielsweise SQL erfolgen. Letzteres setzt jedoch eine Abbildung des mengenorientierten, flachen Datenmodells auf das XML-basierte hierarchische Datenmodell des hier vorgestellten Ansatzes voraus. Sollen strukturierte Daten in vollem Umfang verarbeitet werden, so bedingt dies auch die Nutzung adäquater Anfragesprachen wie beispielsweise XQuery. Zudem müssen die partizipierenden Datenquellen in Form von Dienstendpunkten spezifiziert werden.

Bereits definierte DIA-Prozesse herstellerspezifischer Systeme besitzen die Eigenschaft, dass sie nicht standardisierte Operatordefinitionen aufweisen [186, 140, 153]. Zudem basieren das grundlegende Datenmodell und somit auch die Operatorparameter meist auf SQL [21, 141]. Dadurch lassen sich diese bereits existierenden Prozesse nicht direkt auf die neue Ausführungsumgebung übersetzen. Weiterhin

können Datenquellen und Datensenken nur über Dienstaufrufe in den Prozessfluss der neuen Prozessausführung integriert werden. Dies bedingt die durchgängige Verwendung von XML als Datenformat der Wahl und setzt voraus, dass die zu integrierenden Datenquellen und -senken mithilfe von Dienstimplementierungen in der SOA-Infrastruktur bereitgestellt werden.

Nachrichtenstromanalyse (BAM)

Auch die Anwendungsklasse (BAM) lässt sich zunächst einfach auf das neue Prozessmodell abbilden. Dies liegt, wie bereits bei der Anwendungsklasse DIA, an der inhärenten Datenflusssemantik dieser Anwendungsklasse. Auch in dieser Klasse kann die Definition neuer Prozesse auf Basis einer graphischen Modellierung ähnlich zu [97, 199] vollzogen werden. Außerdem lässt sich die Definition von BAM-Prozessen über die Nutzung von deklarativen Anfragen wie XQuery [32, 57] realisieren. Eine weitere Möglichkeit der Prozessdefinition bilden Regelsprachen auf Grundlage von Kennzahlen und deren Zielvorgaben in einem Zeithorizont [89]. Teile dieser regelbasierten Abbildung wurden in den Prototypen integriert [52], der in Kapitel 7 näher vorgestellt wird.

Dienstbasierte Geschäftsprozesse (BPM)

Die Abbildung dienstbasierter Prozesse auf die strombasierte Prozesssemantik ist abhängig von den jeweiligen Prozesstypen. Dienstbasierte Prozesse, welche vornehmlich Anwendungs- und Datenintegrationsaufgaben im Kontext von Enterprise Application Integration (EAI) realisieren [27], können durch ihre vermehrte Nutzung von datenorientierten Aktivitätstypen sowie dem reduzierten Einsatz kontrollflussbasierter Aktivitätstypen im Rahmen einer datenflussbasierten Prozessausführung modelliert werden [30].

Die Überführung bereits existierender, kontrollflussbasierter Prozessdefinitionen auf die strombasierte Prozessausführung gestaltet sich hingegen schwieriger. [26, 29, 30] beschreiben die Abbildungsmöglichkeiten kontrollflussbasierter Prozesse auf ein pipelinebasiertes Verarbeitungsmodell und präsentieren dazu einen kostenbasierten Ansatz, der über die Bestimmung der Datenabhängigkeiten zwischen Aktivitäten die aufeinanderfolgende Ausführung von Prozessinstanzen in eine pipelinebasierte Ausführung überführt. Dabei bildet die einzelne Prozessinstanz jedoch das zu verarbeitende Granulat in einem Operator, indem alle Prozessvariablen in einem Container gehalten werden. Weiterführende Datenpartitionierungen für die Daten innerhalb einer Prozessinstanz betrachtet der kostenbasierte Ansatz nicht. Eine Überführung existierender Prozesse ist mithilfe dieses Ansatzes zunächst problemlos möglich. Für die partitionierte, skalierbare Verarbeitung beliebig großer Datenmengen in einer Prozessinstanz auf Basis der definierten Datenoperationen *Aufteilung* und *Aggregation* ist zum einen die Analyse der Operationen der Prozessinstanzen auf den Daten

sowie die dazugehörigen Parameter notwendig. Dadurch können die Verarbeitung von möglichen Wiederholgruppen in den Prozessdaten erkannt, die Operatoren entsprechend angepasst und die partitionierte Verarbeitung durchgeführt werden. Zum anderen beschränkt sich bei einer solchen automatisierten Abbildung die Dienstnutzung auf die bereits in der existierenden Prozessdefinition referenzierten klassischen Dienste. Die automatische Integration von strombasierten Diensten ist nicht möglich.

5.5 Evaluierung

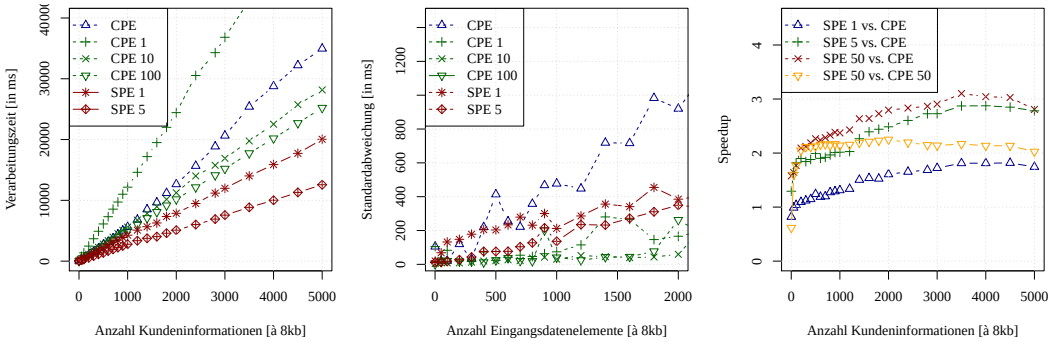
Der nachfolgende Abschnitt evaluiert die in diesem Kapitel vorgestellten Konzepte der strombasierten Prozessausführung und deren Einflussgrößen.

Experimentaufbau

Die prototypische Umsetzung der Prozessausführungsumgebung erfolgte analog zur *strombasierten Dienstkommunikation* in Java 1.6. Zur Kommunikation mit entfernten klassischen und strombasierten Diensten kam daher das modifizierte Rahmenwerk Axis2 aus Kapitel 4 zum Einsatz. Alle Experimente bzw. die dazugehörigen Prozessinstanzen dieses Kapitels wurden auf Knoten A (vgl. Tabelle A.1, Seite 195) ausgeführt. Die entfernten Dienste wurden auf Knoten B platziert und beide Serverknoten über ein LAN miteinander verbunden.

Im Allgemeinen werden die sechs Performance-Aspekte der Eingangsdatengröße n (Anzahl der Eingangsdatenelemente), der Anzahl der Operatoren in einer Prozessinstanz o, der Chunkgröße der Buckets c, der Warteschlangengröße q, der Zeitintervall zwischen zwei Prozessaufrufen t und die Anzahl an m vordefinierten Prozessaufrufen betrachtet. Alle Experimente basieren auf synthetisch generierten Daten und wurden jeweils 50 mal wiederholt.

Mithilfe der hier vorgenommenen Experimente wird die klassische, kontrollflussbasierte Prozessausführung (*Control-flow-based Process Execution*, `CPE`) mit der in diesem Kapitel vorgestellten strombasierten Prozessausführung (*Stream-based Process Execution*, `SPE`) verglichen. Als Basisprozess dient eine Sequenz von sechs Operatoren. Dabei werden eine Eingangsnachricht m_i mit Kundeninformationen empfangen (`receive`), die Kundennummern extrahiert ($o1$, `transform`) und als Eingabe für die Extraktion der dazugehörigen Rechnungen ($o2$, `invoke`) genutzt. Die zurückgegebenen Rechnungen werden wiederum lokal transformiert ($o3$, `transform`) und über einen Dienstaufruf gefiltert ($o4$, `invoke`). Abschließend werden die Kundennummern aus den zurückgegebenen Rechnungen extrahiert und um die Kundeninformationen angereichert ($o5$, `transform`), sodass die Ausgabedatenmenge der Eingabedatenmenge von $o1$ entspricht. Diese Sequenz wird im Folgenden als $o = 5$ referenziert.



(a) Skalierbarkeit über n. (b) Standardabweichung über n. (c) Speedup SPE vs. CPE.

Abbildung 5.15: Intra-Prozess-Performancemessungen I.

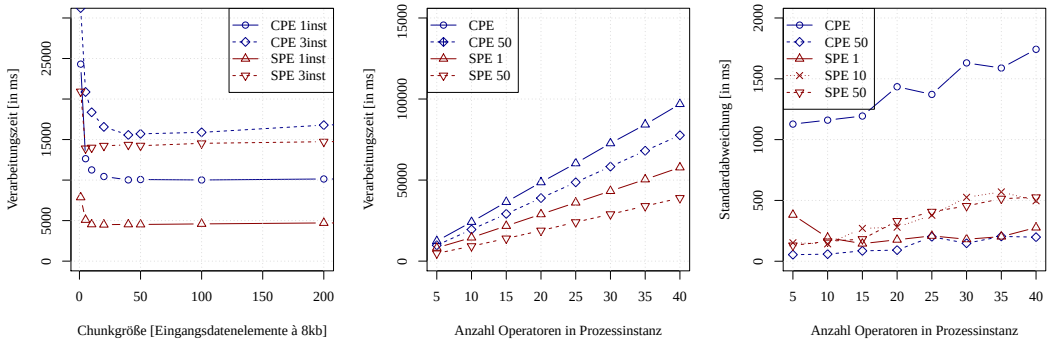
Bei einer Skalierung von o bis $o = 40$ werden diese fünf Operatoren der Basissequenz kopiert und an die bisherige Operatorenkette angehangen.

Performancemessungen

Zunächst werden Experimente zu den Performance-Aspekten *innerhalb* einer Prozessinstanz (Intra-Prozess) vorgestellt. Die Abbildungen 5.15(a)-5.17(c) zeigen deren Ergebnisse.

In Abbildung 5.15(a) wird zunächst die Eingangsdatengröße n variiert und die Verarbeitungszeit gemessen. Dabei zeigt sich ein deutlich besseres Laufzeitverhalten der datenflussbasierten `SPE` gegenüber der kontrollflussbasierten `CPE`. `SPE` partitioniert die eintreffenden Kundeninformationen mit $c = 1$, wodurch ein Datenelement pro Bucket transportiert wird. Die Ausführungsvariationen `CPE 1`, `CPE 10` und `CPE 100` beschreiben den partitionierten, seriellen Aufruf von separaten CPE-Instanzen mit je $c = 1$, $c = 10$ oder $c = 100$ Kundeninformationen pro Prozessaufruf, bis die Summe der in allen Prozessinstanzen zu verarbeitenden Kundeninformationen der Eingabedatenmenge n entspricht. Diese chunkbasierte Ausführungssemantik erlaubt zwar die Verarbeitung beliebig großer Datenmengen mit `CPE`, jedoch existiert nur für jeweils c Kundeninformationen ein gemeinsamer Kontext und nicht für alle n. `SPE 1` bzw. `SPE 5` entsprechen dabei `SPE` mit $c = 1$ bzw. $c = 5$, d.h. dass bei $c = 1$ jedes Datenelement in einem separaten Bucket durch den Prozessplan geleitet wird, während bei $c = 5$ die aus einem Operator ausgehenden Datenelemente zu Buckets mit je fünf Datenelementen zusammengefasst werden. Diese Zusammenfassung entspricht der *Intra-Prozess-Optimierung durch Bucketreduzierung* (vgl. Abbildung 5.13, Seite 99).

Ein Vergleich der Standardabweichungen (Abbildung 5.15(b)) der Ausführungsvarianten ergibt für `SPE 1` eine konstant höhere Standardabweichung als für `SPE 5`. Dies lässt sich mit dem hohen Synchronisierungsaufwand zwischen den Warteschlangen und den Operatoren sowie der damit einhergehenden sehr großen Anzahl von Pro-



(a) Ausführungszeit über c.

(b) Skalierbarkeit über o.

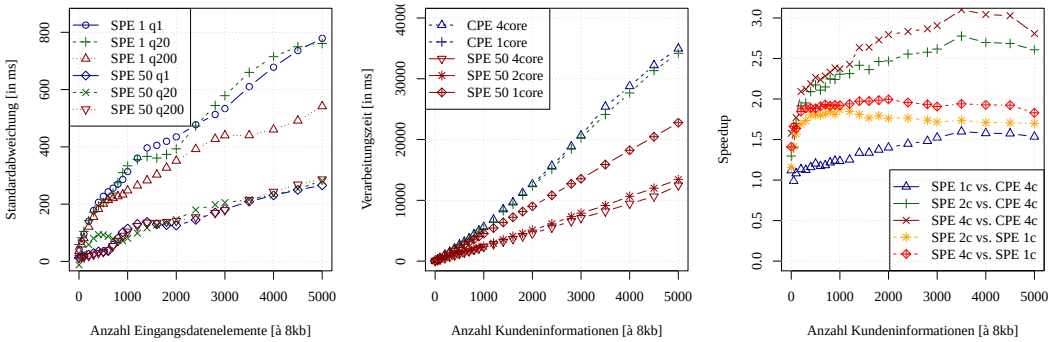
(c) Skalierbarkeit über n für unterschiedliche q.

Abbildung 5.16: Intra-Prozess-Performancemessungen II.

zessbuckets bei $c = 1$ begründen. Da $o = 5$, arbeiten zudem fünf Operatoren nebenläufig in eigenständigen Threads, wobei das Testsystem nur über vier CPU-Kerne verfügt und damit zur Abarbeitung der fünf Operatoren zwischen diesen umgeschalten muss. Die Standardabweichung von *CPE* steigt mit zunehmendem n. Diese Beobachtung entspricht dem Ergebnis der Analyse der Standardabweichung des *Gesamtnachrichtentransfers* aus Kapitel 4 (Abbildung 4.16(b), Seite 76) und lässt sich ähnlich erklären. Dabei steigt der Einfluss von Transfer- und Verarbeitungsverzögerungen mit steigendem n durch die strikte, schrittweise Abarbeitung der Aktivitäten und der groben Granulariät der Verarbeitunspakete. Bei der Verarbeitung der Daten auf einer feineren Granularitätsstufe (`CPE{1,10,100}`) bleibt die Standardabweichung gering.

Für die Darstellung des erreichten Speedups von `SPE` gegenüber `CPE` auf dem Serverknoten mit vier CPU-Kernen, wird die Chunkgröße c für alle `CPE`-Ausführungen zunächst auf $c = n$ gesetzt (`CPE`) und deren Ausführungzeit mit der Ausführungszeit bei unterschiedlichen c mit $c = \{1, 5, 50\}$ der `SPE` verglichen (`SPE{1,5,50}`). Abbildung 5.15(c)) zeigt dazu den jeweils erreichten Speedup von `SPE` gegenüber `CPE`. Dabei nimmt der Speedup zunächst jeweils mit steigendem n zu, stagniert dann jedoch bei $n = 3500$. Analysiert man zudem die partitionierte Ausführung von `CPE` (`CPE 50`) mit der von `SPE 50`, so bleibt der Speedup konstant.

Abbildung 5.16(a) vergleicht die Ausführungszeiten für $n = 2000$ mit unterschiedlichen Chunkgrößen c und keiner (`1inst`) bzw. zwei (`3inst`) zusätzlichen nebenläufigen Prozessinstanzen. Im Allgemeinen haben `CPE` und `SPE` eine von der Anzahl der nebenläufigen Instanzen annähernd unabhängige optimale Chunkgröße bei $c = 40$ für `CPE` und $c = 5$ bzw. $c = 10$ bei `SPE`. Für `SPE` scheint somit die Bündelung von jeweils 5 bzw. 10 Datenelementen in einem Verarbeitungsbucket das optimale Verhältnis zwischen Synchronisierungsaufwand von Operatoren und Warteschlangen auf der einen Seite und der möglichen nebenläufigen Verarbeitung dieser Elemente auf der anderen Seite zu definieren.



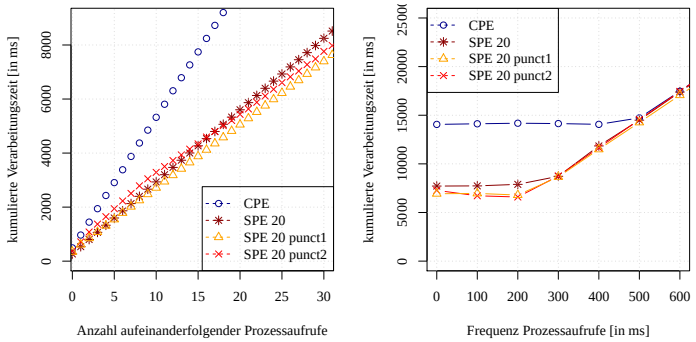
(a) Standardabweichung für q. (b) Einfluss Anzahl genutzter Prozessorkerne. (c) Speedup über Anzahl Prozessorkerne.

Abbildung 5.17: Intra-Prozess-Performancemessungen III.

Abbildung 5.16(b) zeigt die Ausführungszeiten in Abhängigkeit der Anzahl aufeinanderfolgender Operatoren o von $o = 5$ bis $o = 40$ in einer Prozessinstanz für $n = 2000$ und zwischen 2 ($o = 5$) und 16 ($o = 8$) Dienstaufrufen. Dabei lässt sich sowohl im Falle von `CPE` als auch von `SPE` eine lineare Abhängigkeit der Ausführungszeit bei steigender Operatorenzahl feststellen. Dies ist bei `CPE` damit zu erklären, dass alle Operatoren seriell ausgeführt werden und sich deren Ausführungszeiten somit addieren. `SPE` führt hingegen alle o Operatoren nebenläufig aus, wodurch zwar der Synchronisierungsaufwand zwischen Operatoren und Warteschlangen steigt, jedoch auch Wartezeiten bei der entfernten Kommunikation durch die Ausführung anderer Operatoren versteckt werden (engl. *latency hiding*). Der Anstieg der Ausführungszeit fällt somit bei `SPE` geringer aus als bei `CPE`.

Ein Vergleich der Standardabweichungen der Ausführungszeiten bei unterschiedlichen Anzahlen an Operatoren pro Prozessinstanz (vgl. Abbildung 5.16(c)) offenbart für `SPE` im Allgemeinen eine höhere Standardabweichung als `CPE`. Zudem liegt die Standardabweichung im Falle von `SPE` mit $c = 50$ (`SPE 50`) bei $o = 5$ Operatoren pro Prozessinstanz unter der der Standardabweichung von `SPE` mit $c = 1$ (`SPE 1`), was bereits in Abbildung 5.15(b) zu erkennen war und mit dem sehr hohen Synchronisationsaufwand für Warteschlangen und Operatoren erklärt wurde. Ab einer Operatorenzahl von 10 wirkt sich jedoch die grobgranulare Verarbeitung von 50 Datenelementen pro Bucket negativ auf die Standardabweichung aus, da die Verarbeitung der Daten in der Prozessinstanz zunehmend von der externen Kommunikation und deren Latenzen für Chunks der Größe 50 bestimmt wird. Dabei sei darauf verwiesen, dass $o = 40$ insgesamt 16 nebenläufige Dienstaufrufe und damit 16 nebenläufige Dienstinstanzen auf *einem* externen Server bedingt.

In einem weiteren Experiment wird das Laufzeitverhalten von `SPE` bei unterschiedlichen Warteschlangengrößen der Operatoren mit Kapazitäten von $q = \{1, 20, 200\}$ untersucht. Die Experimente ergeben, dass die durchschnittliche Laufzeit einer `SPE`-Prozessinstanz unabhängig von der gewählten Warteschlangengröße q ist. Die da-



(a) Kumulierte Ausführung für $t = 0$.

(b) Skalierbarkeit über t für 30 Prozessaufrufe.

Abbildung 5.18: Inter-Prozess-Performancemessungen.

zugehörige Standardabweichung (vgl. Abbildung 5.17(a)) steigt hingegen bei allen Warteschlangenkapazitäten mit steigendem n. Dabei entwickeln sich die Standardabweichungen für $c = 1$ bei $q = 1$ (`SPE 1 q1`) und $q = 20$ (`SPE 1 q20`) sehr ähnlich, nur für $q = 200$ (`SPE 1 q200`) fällt der Anstieg der Standardabweichung mit steigendem n geringer aus. Dies lässt sich auf die Puffereffekte zurückführen, mit denen beispielsweise Latenzen bei der Kommunikation durch die nebenläufige Verarbeitung anderer Operatoren versteckt werden. Bei $c = 1$ erfolgt die Verarbeitung in den Operatoren sehr feingranular mit der höchsten Anzahl an Prozessbuckets in einer Prozessinstanz und ihren Warteschlangen. Bei $c = 50$ verhält sich die Standardabweichung durch die stark verringerte Anzahl an Prozessbuckets in der Prozessinstanz bei allen drei Warteschlangenkapazitäten $q = 1$ (`SPE 50 q1`), $q = 20$ (`SPE 50 20`) und $q = 200$ (`SPE 50 q200`) gleich.

Abbildung 5.17(b) und 5.17(c) zeigen den Einfluss der Anzahl der Prozessorkerne auf die Gesamtperformance des Beispielprozesses. Dabei werden die absoluten Ausführungszeiten (Abbildung 5.17(b)) sowie der erreichte Speedup (Abbildung 5.17(c)) dargestellt. Für die Durchführung des Experiments wurde c bei `CPE` auf $c = n$ und c bei `SPE` auf $c = 50$ gesetzt. Zudem beinhalten alle Prozessinstanzen jeweils fünf Operatoren ($o = 5$). Die Messungen ergeben, dass `CPE` durch die strikte serielle Ausführung der Operatoren bei steigender Anzahl verfügbarer Prozessorkerne keine Verbesserung der Ausführungszeit erhält. Hingegen führt die Hinzunahme eines zweiten Prozessorkerns zu signifikanten Ausführungszeitverbesserungen bei `SPE`. Dieser Beschleunigungsfaktor wird durch eine Erhöhung der Operatorenanzahl o begünstigt.

Abschließend werden die in Abschnitt 5.3 diskutierten Inter-Prozess-Optimierungen experimentell untersucht, welche die effiziente Ausführung aufeinanderfolgender Prozessaufrufe der `SPE` zum Ziel haben. Abbildung 5.18(a) zeigt dazu die kumulierte Ausführungszeit aufeinanderfolgender Prozessaufrufe für `CPE` mit $c = n$ und `SPE` mit $c = 20$ mit den Ausprägungen *keine Optimierung* (`SPE 20`), *Kontextpunktuationen*

mit separaten Dienstaufrufen pro Prozesskontext (`SPE 20 punct1`) und *Kontextpunktuationen mit einem Dienstaufruf pro Prozessinstanz* (`SPE 20 punct2`). Dabei wurde die Wartezeit t zwischen zwei aufeinanderfolgenden Prozessaufrufen auf $t = 0$ gesetzt und die Eingabedatenmenge n auf $n = 100$. Somit werden pro Dienstkontext fünf ($\frac{n}{c}$) physische Eingangsbuckets an $o1$ übergeben. Es zeigt sich zunächst, dass die allgemeine Optimierung mithilfe von Kontextpunktuationen die Gesamtausführungdauer reduziert. Beim Vergleich der beiden vorgestellten Varianten dieser Punktuationen lässt sich hingegen feststellen, dass `SPE 20 punct1` in Summe eine schnellere Verarbeitungszeit besitzt als `SPE 20 punct2`. Dies wurde zunächst nicht erwartet, jedoch scheint sich der Mehraufwand des erneuten Verbindungsaufbaus bei `SPE 20 punct1` durch Puffereffekte stark zu minimieren und die entstehenden Latenzen bzw. Ruhezeiten während des Aufbaus stellen CPU-Ressourcen für die Abarbeitung anderer Operatoren zur Verfügung. Zudem werden beim Abschluss eines Prozesskontextes bereits nachfolgende Kontexte verarbeitet, wodurch sich die Verarbeitungszeit der einzelnen Prozesskontexte erhöht. Das Verhalten von `SPE 20 punct1` im Vergleich zu `SPE 20 punct2` hat sich in Experimenten mit unterschiedlichen n ebenfalls bestätigt.

Den Einfluss der Frequenz t auf die Gesamtausführungsdauer von 30 Prozessen zeigt Abbildung 5.18(b). Dabei wurde t von $t = 0$ auf $t = 600\ ms$ erhöht und die kumulierte Ausführungsdauer gemessen. Es zeigt sich, dass durch `SPE` im Allgemeinen sowie durch die Inter-Prozess-Optimierungen eine höhere Frequenz an aufeinanderfolgenden Prozessaufrufen unterstützt wird. Vergleicht man `SPE 20 punct1` und `SPE 20 punct2` in diesem Zusammenhang genauer, so benötigen alle 30 Prozessaufrufe für `SPE 20 punct2` bei $t \neq 0$ eine geringere Gesamtausführungszeit. Dies stützt die Annahme, dass die Existenz von Ruhephasen während der Verarbeitung sich positiv auf `SPE 20 punct2` auswirkt.

5.6 Zusammenfassung

Dieses Kapitel stellte den Ansatz der *strombasierten Prozessausführung* vor, welcher die in Kapitel 2 formulierten sechs Anforderungen *A1 (Verarbeitung)*, *A2 (Datenaustausch)*, *A3 (Latenz)*, *A4 (Korrelation)*, *A5 (Datenmodell)* und *A6 (Dienstintegration)* auf Ebene der Prozessausführung unterstützt und somit die konsolidierte Ausführung von Anwendungen aller drei Anwendungsklassen erlaubt. Auf dieser Ebene wurden die notwendigen Anforderungen in zwei Bereichen adressiert.

Im Bereich des Datenmodells wurde das Konzept der Prozessbuckets vorgestellt, welches von den eigentlich zu verarbeitenden Daten abstrahiert und deren prinzipielle strombasierte Verarbeitung ermöglicht. Zur Abbildung der notwendigen Datenverarbeitungsanforderungen der Anwendungsklassen DIA, BAM und BPM wurden auf Grundlage der Prozessbuckets die *Datenoperationen* der *Modifikation*, *Aufteilung* und *Aggregation* definiert. Die interne Datenrepräsentation des Datenmodells ba-

siert auf *XML Infoset* und unterstützt damit nativ Anforderung A5. Die Umsetzung der Datenoperationen erfolgt dabei mithilfe der XML-basierten Sprachen XPath, XQuery und XSLT.

In einem auf dem Datenmodell aufbauenden Schritt wurde das Prozessmodell beschrieben, das eine Prozessinstanz durch einen direkten, azyklischen Graphen definiert, dessen Knoten die eigentlichen Operatoren und dessen Kanten FIFO-Warteschlangen zwischen den Operatoren darstellen. Als grundlegende Menge an vordefinierten Operatoren wurden Operatortypen eingeführt, die neben der Umsetzung der *Datenoperationen* des Datenmodells auch grundlegende Funktionalitäten der drei in dieser Arbeit zu konsolidierenden Anwendungsklassen bereitstellen. Die tiefe Integration *strombasierter Dienstaufrufe* in das Prozessmodell erlaubt es zudem, strombasierte Dienste als entfernte Operatoren beliebiger Funktionalität im Prozess zu nutzen.

Die Kombination der beiden Ebenen des Datenmodells und des Prozessmodells sowie die Ausführungsoptimierungen auf Prozessmodellebene definieren den Ansatz des *strombasierten Prozessaufrufes*. Im Rahmen der durchgeführten Experimente wurde gezeigt, dass

1. eine strombasierte Verarbeitung von Daten vollzogen werden kann, die den Speicherverbrauch in einer Prozessinstanz reduziert und deren Ausführung für beliebige Datengrößen skaliert (*A1*).
2. die Abstraktion von den zu verarbeitenden Anwendungsdaten mithilfe des Konzeptes der XML-basierten Prozessbuckets (*A5*) weiterführende Optimierungen, wie beispielsweise das Konzept der stehenden Prozessinstanzen für gemeinsame Prozesskontexte (*A4*) oder rein logisch separierte Prozesskontexte in einer gemeinsamen Prozessinstanz (*A3*), erlaubt.
3. die zusätzliche Integration des strombasierten Dienstaufrufes aus Kapitel 4, im Gegensatz zur traditionellen Dienstkommunikation, die Erweiterung des Prozesses um beliebige, auch datenintensive Operationen ermöglicht (*A2*, *A6*).

Die Abstraktion von den eigentlich zu verarbeitenden Anwendungsdaten ermöglicht weitere Optimierungen, wie beispielsweise die Integration von Mechanismen für transaktionale Sicherheit, Robustheit und Fehlertoleranz [17, 34, 35, 75, 77]. Des Weiteren ermöglichen schemafreie und nur auf das Konzept der Prozessbuckets fixierte Warteschlangen auch die Übertragung multipler, nicht auf eine Datenstruktur fixierter Anwendungsdaten zwischen zwei Operatoren. In Verbindung mit der Möglichkeit, mehrfache Parameterdefinitionen in einem Operator, beispielsweise im `route`-Operator, für unterschiedliche Anwendungsdatenstrukturen zu definieren, eliminiert diese Schemafreiheit die einschränkende Annahme, dass nur gleichstrukturierte Daten zwischen zwei Operatoren fließen dürfen. Eine flexiblere Modellierung und Definition von Prozessflüssen, in denen beispielsweise der `receive`-Operator komplexe Datenstrukturen partitioniert und gemeinschaftlich an die nachfolgenden Operatoren weitergibt, ist die Folge.

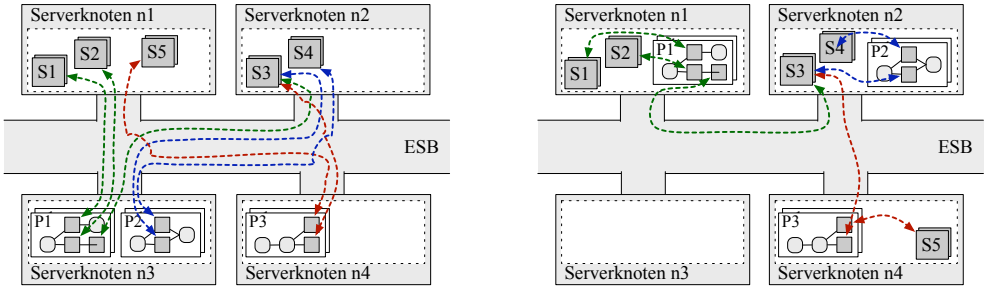
6 Kommunikations- und workloadbasierte Verteilung von Prozessen

Die vorherigen Kapitel widmeten sich der skalierbaren Ausführung geschäftsrelevanter IT-Prozesse auf den Ebenen der *atomaren Dienste* (Kapitel 4) und der *kompositen Prozesse* (Kapitel 5). Beide Ebenen bzw. ihre Softwarekomponenten bilden die grundlegenden Akteure einer SOA bzw. einer dienstorientierten Unternehmenslandschaft und interagieren über Nachrichtenkommunikation miteinander [114, 159]. Die dadurch erreichte lose Kopplung ermöglicht es, die Komponenten dieser beiden Ebenen zunächst beliebig auf die Ebene der physischen Serverknoten abzubilden [99, 159].

Die tatsächliche physische Verteilung von SOA-Komponenten auf die vorhandenen Serverknoten in einer IT-Infrastruktur wird dabei oft durch betriebliche Randbedingungen und weniger anhand performance-spezifischer Gesichtspunkte definiert [112, 147]. Historisch gewachsene Server- und Anwendungslandschaften oder unterschiedliche Unternehmensabteilungen und damit verschiedene Zuständigkeitsbereiche sind maßgeblich für die vorherrschende Verteilung von Komponenten auf den Serverknoten der SOA-Infrastruktur verantwortlich [147].

Der stark verteilte Charakter einer solchen IT-Landschaft bedingt jedoch weiterhin eine intensive Kommunikation zwischen Komponenten verschiedener Serverknoten auf Basis standardisierter, XML-basierter Nachrichten. Trotz vielfältiger Ansätze, den Mehraufwand einer solchen XML-basierten Kommunikation in verschiedenen Teilaspekten zu reduzieren (vgl. Kapitel 3.2.2), zeichnet sich die XML-basierte Kommunikation durch einen hohen Ressourcenverbrauch in den Bereichen CPU [46, 50, 64], Hauptspeicher [152] und Netzwerkkommunikation [47, 55, 93] aus und definiert damit den performance-kritischen Faktor datenintensiver Prozessanwendungen [73] in SOA-Umgebungen. Dies gilt unabhängig vom gewählten Prozessausführungsmodell und trifft damit auch auf das in dieser Arbeit vorgestellte Konzept der strombasierten Prozessausführung zu.

Abbildung 6.1(a) zeigt dazu eine typische Netzwerktopologie mit den gerade beschriebenen Eigenschaften. Serverknoten ($n1 \ldots n4$) führen dabei die eigentlichen Dienste ($S1 \ldots S4$) und Prozesspläne ($P1 \ldots P3$) aus. Diese Serverknoten sind über einen Enterprise Service Bus (ESB) (vgl. Abschnitt 2.1) verbunden, welcher die Nachrichtenzustellung zwischen entfernten Komponenten übernimmt. Die Verteilung der Dienste auf die Serverknoten basiert, wie zuvor beschrieben, nicht auf performance-orientierten Gesichtspunkten. Prozessflüsse werden zudem meist in zen-



(a) Historisch bzw. fachlich bedingte Komponentenverteilung.

(b) Kommunikationsoptimierte Komponentenverteilung.

Abbildung 6.1: SOA Infrastruktur-Topologie.

tralen Ausführungsumgebungen auf dedizierten Servern ausgeführt [112] (vgl. Prozesse $P1$, $P2$ und $P3$ auf Knoten $n3$ und $n4$). Dabei interagiert jede Prozessinstanz eines Prozessplans typischerweise mit einer Menge an Diensten, um auf Daten zuzugreifen oder sie zu modifizieren (vgl. Abbildung 6.1(a): $P1 = \{S1, S2, S3\}$, $P2 = \{S3, S4\}$ und $P3 = \{S3, S5\}$).

Eine intelligente und arbeitslastbasierte Verteilung von SOA-Komponenten auf verfügbare Serverknoten einer Infrastruktur, bei der miteinander interagierende Komponenten physisch nah zueinander platziert werden, bietet ein hohes Optimierungspotenzial, um die dabei auftretende Kommunikationen zwischen diesen Komponenten zu reduzieren bzw. gegebenenfalls (bei Platzierung auf demselben Serverknoten) zu eliminieren [19, 90, 112, 139]. Abbildung 6.1(b) zeigt beispielhaft eine kommunikationsoptimierte Verteilung, bei der die Prozesse und die von ihnen benötigten Daten (Dienste) zusammengebracht werden. Dies reduziert die notwendigen CPU- und Speicherressourcen einer XML-basierten Kommunikation mit entfernten Komponenten und eliminiert sowohl die Latenzen der Kommunikation zwischen Prozessen und Diensten als auch das Netzwerktransfervolumen und somit Latenzen für andere Aufgaben auf dem ESB.

Aus diesem Grund betrachtet dieses Kapitel die kommunikationsoptimierte Abbildung der SOA-Komponenten auf vorhandene Serverknoten und erweitert damit das bisherige Gesamtkonzept aus *atomarer Dienstebene* (vgl. Kapitel 4) und *kompositer Prozessebene* (vgl. Kapitel 5) um die zu diesen Ebenen orthogonale *Infrastrukturebene*. Abbildung 6.2 zeigt die Einordnung dieser Erweiterung im Kontext dieser Arbeit.

Um die kommunikationsoptimierte Abbildung der SOA-Komponenten auf die Infrastrukturebene zu realisieren, wird das aus dem Gebiet der physischen Datenbankoptimierung bekannte Konzept des *Design-Advisors* [6, 43, 168] adaptiert. Dieses trifft mit einer *what-if* Semantik [42] Annahmen über das vorhandene System und

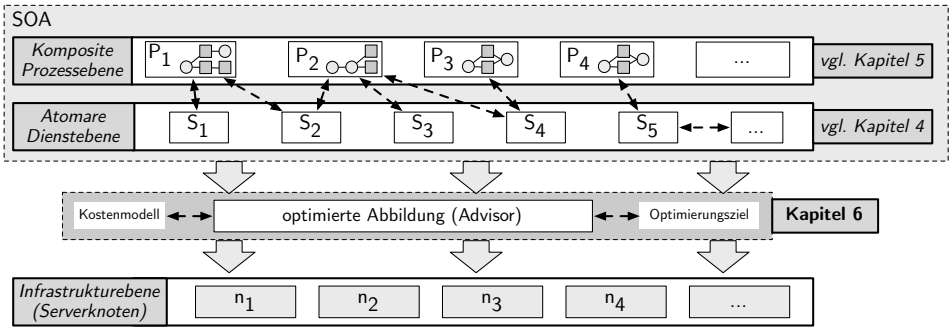


Abbildung 6.2: Die drei Ebenen einer SOA-basierten IT-Infrastruktur.

schlägt mithilfe eines Kostenmodells und einer jeweiligen Zielfunktion eine optimale Systemkonfigurationen vor.

Deshalb wird in diesem Kapitel der *Infrastruktur-Design-Advisor* [125] vorgestellt, welcher

- ein System- und Kostenmodell für den strombasierten Dienstaufruf und die strombasierte Prozessverarbeitung einführt,
- die Arbeitslast der zu verteilenden Prozesse und Dienste in Form von eintreffenden Prozess- und Dienstaufrufen sowie deren Interaktionen miteinander berücksichtigt und
- eine kommunikationsoptimierte Abbildung der kompositen Prozesse und atomaren Dienste auf die vorhandenen Serverknoten vorschlägt.

Obwohl die Abbildung der Komponenten auf die Serverknoten auf unterschiedlichsten Optimierungszielen beruhen kann (vgl. Abschnitt 6.2.3), liegt der Fokus dieser Arbeit auf der Reduktion der Kommunikation zwischen den beteiligten Prozessen und Diensten, da die Kommunikation den performance-kritischen Faktor einer Prozessausführung in Form von Netzwerklatenz und Ressourcenverbrauch darstellt.

6.1 Vorbetrachtungen

Dieser Abschnitt legt die Grundlagen für den *Infrastruktur-Design-Advisor* und diskutiert zunächst verwandte Arbeiten. Da das in dieser Arbeit vorgestellte Konzept der *strombasierten Prozessausführung* sowohl Eigenschaften klassischer, workflowbasierter Systeme als auch Eigenschaften von Datenstromsystemen besitzt, werden zunächst verwandte Arbeiten dieser beiden Klassen diskutiert (Abschnitt 6.1.1) und daraufhin Implikationen (Abschnitt 6.1.2) für das Kostenmodell des *Infrastruktur-Design-Advisors* in Abschnitt 6.3 abgeleitet.

6.1.1 Verteilung und Fragmentierung von Prozessen

Die Verteilung von Prozessgraphen und ihrer Kommunikationspartner (im Weiteren allgemein Komponenten genannt) auf verfügbare Serverknoten wurde in der bisherigen Literatur für verschiedene Anwendungsbereiche und Systemklassen untersucht. Ziel dieser Analysen war dabei zumeist die optimale Verteilung im Rahmen einer Lastbalancierung (engl. *load balancing*). Im Folgenden werden die wesentlichen Arbeiten dazu kurz vorgestellt.

Im Kontext von *dokumentbasierten Workflowsystemen* wurde bereits die örtlich naheliegende Ausführung von Aufgaben und ihrer dazugehörigen Daten im Rahmen physisch verteilter Workflow-Aktivitäten betrachtet [7, 19, 120]. Während [7] Kontroll- und Dokumentenfluss explizit separiert und damit entfernte Aktivitäten vorab mit den benötigten Dokumenten versorgt, unterteilt [19] das Servernetzwerk in sogenannte Domains bzw. Subnetze, zwischen denen die Kommunikation minimiert werden soll. Hashbasierte Funktionen unterteilen die Workflowprozesse in Prozessfragmente und verteilen diese auf die vorhandenen Domains. Innerhalb einer Domain werden die Prozessfragmente dabei jedoch zufällig auf die Serverknoten verteilt. Die Kommunikation zwischen diesen Servern wird nicht betrachtet.

In den vergangenen Jahren wurde die Optimierung von *SOA-basierten Workflowsystemen* bzw. Integrationsprozessen im Allgemeinen vermehrt untersucht [23, 29, 31, 101, 146, 156]. Ansätzen einer prozessübergreifenden Optimierung als Ausgangspunkt einer intelligenten Komponentenplatzierung wurde dabei allerdings wenig Aufmerksamkeit zuteil. So existieren Ansätze, welche die Fragmentierung vorhandener Prozessbeschreibungen und deren Verteilung auf Serverknoten adressieren. Dabei werden vormals zentrale Prozessbeschreibungen entweder durch manuelle Annotation [16] oder durch automatische Analyse [11, 39, 41, 90] in Prozessfragmente unterteilt und diese auf Serverknoten platziert. Ein solcher Vorgang ähnelt der Anfrageplanpartitionierung in Datenstrommanagementsystemen. Die kostenbasierte Bewertung der Prozessfragmente sowie die ganzheitliche Optimierung von Fragmenten unterschiedlicher Prozesspläne wurde in bestehenden Ansätzen jedoch nicht betrachtet.

Weiterhin existieren in der Klasse der *SOA-basierten Workflowsysteme* Arbeiten, welche die physische Realisierung der Verteilung und der Kommunikation von Prozessfragmenten [83, 112, 135], die Migration von laufenden Prozessinstanzen [166] oder die Zuordnung von Kommunikationspartnern zu Prozessinstanzen [54] adressieren. Allen Arbeiten ist jedoch gemein, dass sie keine konkreten Kostenmodelle, keine kostenbasierten Platzierungsstrategien und damit keine kostenbasierte Optimierung einer gesamtheitlichen Komponentenverteilung betrachten.

Auf dem Gebiet der *Datenstrommanagementsysteme* (DSMS) basiert Lastbalancierung über verteilte Serverknoten entweder auf der *Partitionierung der Anfragepläne* [15, 14, 76, 136, 158] oder auf der *Partitionierung des Datenstroms* [40, 81, 137]. Während [15] Anfragepläne auf unterschiedlichen Serverknoten ausführt und die Da-

tenstromlast über ein vertragsbasiertes Konzept operatorweise auf die Server verteilt, versuchen [136] über Multi-Query-Optimization (MQO) die Anfrageteilpläne so auf die vorhandenen Serverknoten zu verteilen, dass möglichst viele Teilpläne von Zwischenergebnissen eines Knotens profitieren. Des Weiteren existieren Ansätze, welche mehrere Instanzen einer Anfrage aus Gründen der Ausfallsicherheit auf unterschiedlichen Serverknoten platzieren [14, 76]. Im Rahmen von Datenstrompartitionierung beschreiben [40, 137] einen dynamisch anpassbaren FLUX-Operator, welcher zwischen zwei Datenstromoperatoren platziert wird und den Datenstrom zwischen unterschiedlichen Instanzen des nachfolgenden Operators ausführungsabhängig auf unterschiedliche Serverknoten verteilt. In [81] wird ebenfalls Datenstrompartitionierung angewandt, wobei eine Strategie zur Aufteilung von Zeitfenstern genutzt wird, um Eingangsdaten so zu partitionieren, dass die Ergebnisse dieser Partitionen effizient wieder vereint werden können. Weiterhin kombiniert [85] Anfrageplanpartitionierung und Datenstrompartitionierung, sodass Daten- und Anfragepartitionen verschiedener Anfragen so auf Serverknoten vereinigt werden, dass zum einen Zwischenergebnisse zwischen den Anfragepartitionen eines Serverknotens geteilt werden und zum anderen der involvierte Datentransfer zwischen den Knoten minimiert wird.

Alle genannten Ansätze auf dem Gebiet der DSMS arbeiten auf Basis klassischer, relationaler DSMS und damit auf CQL-Anfragen wie beispielsweise Aggregationsanfragen oder einfachen Selection-Projection-Join-Anfragen (SPJ). Die Komplexität der einzelnen Operatoren und Dienste kompositer Prozessgraphen in SOA-Umgebungen wird dabei nicht betrachtet. Auch das spezifische Ausführungsverhalten eines zentralen Prozesses, bei dem der Datenstrom jeweils nach einem entfernten Knoten an die zentrale Prozessinstanz zurückfließen muss, findet in bisherigen Arbeiten keine Beachtung.

6.1.2 Arten von Kostenmodellen und Implikationen

Die in Kapitel 5 vorgestellte *strombasierte Prozessausführung* unterstützt die einheitliche Ausführung klassischer Integrationsprozesse (*BPM/DIA*) mit *endlichen Eingabedatenmengen* pro Prozessinstanz sowie kontinuierliche Datenstromanalyseprozesse (*BAM*) mit einem *unendlichen Eingabedatenstrom* pro Prozessinstanz.

Für die Kostenbewertung beider Prozessarten gibt es in der Literatur jeweils unterschiedliche Ansätze. So existieren im Kontext klassischer Integrationsprozesse mit endlichen Datenmengen pro Prozessinstanz zahlreiche Arbeiten, deren Kostenmodelle auf Relationen und ihren Datenkardinalitäten und Werteverteilungen basieren [2, 28, 111]. Werden hingegen kontinuierliche Datenstromanalyseprozesse ausgeführt, sind diese Datenkardinalitäten und Werteverteilungen der Quelldaten nicht zum Zeitpunkt der Verarbeitung bzw. niemals bekannt. Deshalb existieren auf dem Gebiet der kontinuierlichen Anfrageverarbeitung Ansätze zur ratenbasierten Kostenabschätzung, bei denen die Anzahl der eintreffenden Datenelemente pro Zeiteinheit im Fokus steht [12, 87, 105, 106, 155]. Neben den Eingangsdatenraten betrachten

letztgenannte Kostenmodelle auch den Einfluss der Operatorfunktion auf die Ausgabedatenrate sowie die Verarbeitungszeit pro Datenelement in einem Operator.

Um beide Prozessarten auf ein gemeinsames Kostenmodell abzubilden und damit eine einheitliche Kostenabschätzung aller Prozesse zu gewährleisten, wird die Ausführung klassischer Integrationsprozesse mit endlichen Daten pro Prozessinstanz in eine kontinuierliche Ausführung mit unendlichen Eingangsnachrichten pro Prozessinstanz überführt. Das bereits vorgestellte Konzept der *Stehenden Prozessinstanz* (vgl. Abschnitt 5.3) ermöglicht diese Überführung.

$$m_j \rightarrow p_{i,j} \in P_i \Longrightarrow m_j \rightarrow p_i \in P_i$$

Dabei werden alle eintreffenden Nachrichten m_j, die vorher jeweils eine neue physische Prozessinstanz $p_{i,j}$ eines Prozessplans P_i generierten, an eine bereits laufende Instanz p_i übergeben und eine kontinuierliche Nachrichtenverarbeitung in einer *Stehenden Prozessinstanz* realisiert. Kontextpunktuationen zur Annotation einzelner Nachrichtenkontexte im Datenstrom des Prozesses ermöglichen die isolierte Ausführung einzelner Eingangsnachrichten.

Da Kardinalitäten bzw. Selektivitäten beliebiger, orchestrierter Dienste im Vorfeld nicht bekannt sind, müssen deren Charakteristika mithilfe von Monitoring, Simulation bzw. darauf aufbauenden analytischen Modellen erfasst und für die spätere Nutzung geschätzt werden [80, 138, 162].

Im Folgenden werden zunächst das dem *Infrastruktur-Design-Advisor* zugrundeliegende *Systemmodell* beschrieben und die Beziehungen der Komponenten darin erläutert. Danach gilt es das eigentliche Kostenmodell zu erarbeiten, indem Einflussgrößen der Operatoren und ihre Wirkung auf die Verarbeitungszeit untersucht werden und die Kostenaggregation auf Prozess- und Systemebene beschrieben wird. Abschließend werden die Verteilungsalgorithmen präsentiert und ihre Anwendbarkeit evaluiert.

6.2 Systemmodell

Das Systemmodell abstrahiert von einer konkreten SOA-Infrastruktur und definiert die grundlegenden Entitäten mit denen der *Infrastruktur-Design-Advisor* arbeitet sowie deren Beziehungen zueinander. Um das Systemmodell zu formulieren, werden die Komponenten in eine abstrahierte *Systemarchitektur* eingeordnet und das darauf aufbauende *Advisor-Metamodell* beschrieben.

6.2.1 Systemarchitektur

Als Ausgangspunkt der SOA-Komponenten und ihrer Beziehungen untereinander dient die Architektur in Abbildung 6.3. Darin werden die Softwareebenen der *Kom-*

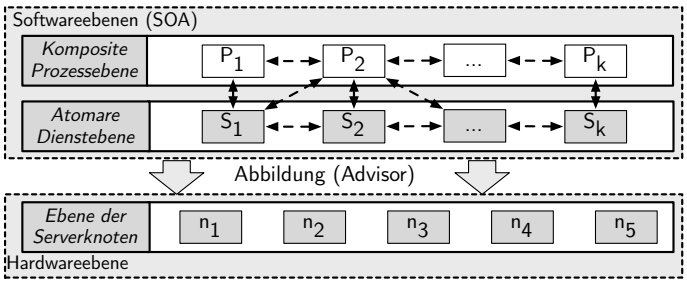


Abbildung 6.3: SOA Komponentenebenen.

positen Prozessebene und der *Atomaren Dienstebene* sowie die Hardwareebene der *Serverknoten* unterschieden. Beide Softwareebenen bilden die Grundbausteine einer Unternehmensarchitektur (SOA) und kommunizieren miteinander.

Die *Atomare Dienstebene* beinhaltet die Kernkomponenten der Anwendungslogik, welche in traditionellen Programmiersprachen verfasst und als Dienstschnittstelle gekapselt wurden. Diese Dienstimplementierungen interagieren mit Datenspeichern wie beispielsweise relationalen Datenbanksystemen [2], Key-Value-Stores [173, 171] oder einfachen Textdateien außerhalb der SOA-Schicht über eng gekoppelte Verbindungen mit meist proprietären Protokollen und Formaten. Die *Komposite Prozessebene* als zweite Softwareebene umfasst Komponenten, welche die Dienste der *Atomaren Dienstebene* zu höherwertigen Prozessen orchestrieren.

Orthogonal zu den beiden beschriebenen *Softwareebenen* befindet sich die physische Hardwareebene der *Serverknoten*, welche aus einer Menge an Serverknoten besteht und alle Softwarekomponenten darauf ausführt. Dabei wird angenommen, dass diese Serverknoten heterogene Hardwareeigenschaften besitzen.

Die Kommunikation zwischen den Softwarekomponenten, d.h. die *vertikale* Kommunikation zwischen der *Atomaren Dienstebene* und der *Kompositen Prozessebene* sowie die *horizontale* Kommunikation mit anderen Komponenten auf der selben Ebene erfolgt dabei grundsätzlich lose gekoppelt über Nachrichtenaustausch in standardisierten Protokollen und Formaten. Dies trifft vor allem auf Komponenten beider Ebenen zu, die entfernt voneinander auf verschiedenen Serverknoten ausgeführt werden. Befinden sich hingegen Softwarekomponenten beider Ebenen auf dem selben physischen Serverknoten, kommunizieren diese Komponenten meist über effizientere Protokolle ohne XML-basierten Nachrichtenaustausch [193].

Ziel ist es im Folgenden, die vorhandenen Prozesspläne und Dienste einer SOA so auf die vorhandenen Serverknoten abzubilden, dass sich das System hinsichtlich des vorher definierten Optimierungsziels optimal verhält. Diese Arbeit konzentriert sich dabei auf das Konzept eines Design-Advisors [6, 43, 168], der mit den gegebenen Informationen zu den einzelnen Softwarekomponenten und Serverknoten *vor* deren Ausführungszeit eine Menge an möglichen Varianten der Komponentenverteilungen (Systemkonfigurationen) erstellt. Diese Systemkonfigurationen werden aufgrund der

Komplexität der heterogenen Softwarekomponenten und ihrer Beziehungen untereinander sowie wegen möglicher unternehmensrechtlicher Beschränkungen in einem zweiten Schritt ähnlich wie im Bereich der physischen Datenbankoptimierung von einem Nutzer bewertet entsprechend vollständig oder in Teilen umgesetzt.

Der Schwerpunkt der Optimierung liegt im Rahmen dieser Arbeit auf der Reduktion der entfernten, nachrichtenbasierten Kommunikation, sodass die *horizontale* Kommunikation mit Komponenten derselben Ebene sowie die *vertikale* Kommunikation mit Komponenten der jeweils anderen Ebene, wenn möglich, auf einem Serverknoten stattfindet.

6.2.2 Advisor Metamodell

Das Metamodell des *Infrastruktur-Design-Advisors* beschreibt die Komponenten und deren Beziehungen zueinander. Es besteht grundlegend aus einer Menge kompositer Prozesse P, einer Menge von Diensten S, einer Menge an Dienstpaketen B (engl. *bundles*), einer Menge an Serverknoten N sowie einer Menge an Systemkonfigurationen K.

Dabei entspricht die Definition von einzelnen Prozessplänen P_i einer reduzierten Version der *strombasierten Prozesse* (vgl. Definition 12, Seite 85) mit $P_i = (O_i, S_i)$, bei denen O_i mit $O_i = (o_1, \ldots, o_k)$ die Menge an Operatoren darstellt, welche als gerichteter, azyklischer Graph miteinander verbunden sind, und S_i mit $S_i = (s_1, \ldots, s_l)$ die Menge an Diensten beschreibt, mit denen P_i interagiert. S_i bildet eine Untermenge aller verfügbaren Dienste S mit $S_i \subseteq S$. Da atomare Dienste untereinander Abhängigkeiten haben können, die eine physische Trennung auf unterschiedliche Serverknoten verhindern, dient das Konzept der Dienstpakete B als Abstraktionsschicht zwischen Diensten und Serverknoten. Dienste in einem Dienstpaket b sind logisch gruppiert und können nur gemeinsam auf einzelnen Serverknoten platziert werden. Dabei enthält ein Dienstpaket b alle Dienste eines Knotens oder nur eine Untermenge davon mit $b = (s_1, \ldots, s_k)$. Außerdem kann die Platzierbarkeit auf ausgewählte Serverknoten eingeschränkt werden, da beispielsweise Dienste in einem Dienstpaket spezielle Hard- oder Softwareanforderungen an einen Serverknoten stellen können.

Weiterhin bildet N die Menge an physischen Knoten mit $N = (n_1, \ldots, n_m)$ ab, welche die Ausführungsumgebungen für Dienste und Prozesse umfassen. Es werden heterogene Hardwareeigenschaften in Bezug auf CPU und Hauptspeicherressourcen sowie homogene Netzwerkverbindungen angenommen.

Abschließend beschreibt K die Menge aller Systemkonfigurationen mit $K = (k_1, ..., k_o)$, welche P und S auf N abbilden. Als Untermenge davon beschreibt K_v mit $K_v \subseteq K$ alle *gültigen* Systemkonfigurationen. Als gültige Systemkonfiguration $k_{v,i}$ wird jedes k_i bezeichnet, welches die durch die Zielfunktion definierten Nebenbedingungen erfüllt. Ziel des *Infrastruktur-Design-Advisors* ist es, die Systemkonfiguration $k_{v,i}$ zu finden, welche sich optimal zur vorher definierten Zielfunktion verhält.

6.2.3 Optimierungsziele

Auf Grundlage des Systemmodells gibt dieser Abschnitt einen kurzen Überblick über mögliche Optimierungsziele bei der Abbildung der SOA-Komponenten auf die vorhandenen Serverknoten. Diese Optimierungsziele gelten für eine *Systemkonfiguration* k auf der Menge aller P, S und N. Zwischen den Optimierungszielen existieren vereinzelt Zielkonflikte, weshalb zunächst eine isolierte Betrachtung jedes Zieles und der dafür jeweils optimalen Verteilung stattfindet.

Im Folgenden werden vier mögliche Optimierungsziele kurz diskutiert.

Minimales Datenvolumen Entsprechend dieses Zieles gilt es, die Summe des Datentransfers über das Netzwerk im Gesamtsystem, d.h. zwischen allen betrachteten Komponenten der SOA-Infrastruktur, zu minimieren. Dies hat mehrere Vorteile. Zum einen wird so der CPU- und Speichermehraufwand für die XML-basierte Nachrichtenkommunikation reduziert. Zum anderen reduziert ein verringertes Transfervolumen die Auslastung des Netzwerkes und erhöht damit die Kapazität für andere netzwerkintensive Aufgaben. Wird der Netzwerktransfer zudem mit monetären Kosten bewertet, impliziert eine transfervolumenminimierende Platzierung der Komponenten eine reale Kostenersparnis.

Minimale Verarbeitungszeit Ziel der minimalen Verarbeitungszeit ist es, Prozesse und Dienste so zu verteilen, dass die summierten Verarbeitungszeiten der einzelnen Prozessinstanzen minimiert werden. Da sich im Kontext SOA-basierter Prozesse die Kommunikation stark auf die Verarbeitungszeiten auswirkt, sollten sich dieses Ziel das zuvor Beschriebene positiv beeinflussen.

Minimale Knotenanzahl Die Zielkonfiguration des Systems soll unter der Voraussetzung, dass kein Knoten bei der Ausführung von Prozessinstanzen überlastet wird, auf der minimalen Anzahl an Knoten ausgeführt werden. Dadurch wird es beispielsweise möglich, existierende Knoten abzuschalten und damit effektiv Kosten zu sparen oder die nicht benutzten Knoten anderweitig dediziert zu nutzen.

Begrenzte Prozessorlast Dieses Optimierungsziel soll die notwendigen Dienste und Prozesse so auf die vorhandenen Knoten verteilen, dass eine vorgegebene Prozessorlast bei Ausführung der Instanzen nicht überschritten wird.

Neben den hier beschriebenen Optimierungszielen lassen sich natürlich weitere Optimierungsziele wie beispielsweise Lastspitzen-Eliminierung oder Lastausgleich zwischen Serverknoten (engl. *load balancing*) ableiten. Diese werden in dieser Arbeit nicht weiter betrachtet, da sie für das Ziel einer effizienten, skalierbaren Ausführung von kompositen Prozessanwendungen eine geringe Relevanz besitzen. Der Fokus dieser Arbeit liegt auf den Zielen des *minimalen Datenvolumens* sowie der *minimalen*

Verarbeitungszeit, da das *Datenvolumen* und damit die Kommunikation auch direkten Einfluss auf die *Verarbeitungszeit* besitzt [55, 50, 73].

6.3 Kostenmodell

Die Besonderheit der *strombasierten Prozessausführung* liegt in der Ausführung sowohl klassischer Integrationsprozesse (*BPM/DIA*) mit *endlichen Eingabedatenmengen* pro Prozessinstanz als auch kontinuierlicher Datenstromanalyseprozesse (*BAM*) mit *unbegrenzten Eingabedatenmengen*. Durch die Überführung der klassischen Integrationsprozesse in eine kontinuierliche Datenstromverarbeitung mithilfe des Konzeptes der *Stehenden Prozessinstanz* (vgl. Abschnitt 6.1.2), wird im Rahmen des *Infrastruktur-Design-Advisors* ein einheitliches, ratenbasiertes Kostenmodell für alle hier betrachteten Prozessarten angewandt. Dabei stehen die Ein- und Ausgaberaten der Operatoren und Prozesse sowie die Verarbeitungszeit einzelner Verarbeitungseinheiten im Fokus. Zusätzlich berücksichtigt das hier vorgestellte Kostenmodell den verteilten Charakter SOA-basierter Prozesse, indem neben den eigentlichen Verarbeitungszeiten auch die Kommunikationkosten eines Prozesses mit den an ihm beteiligten Diensten in die Optimierung der Infrastruktur einfließen.

Aufgrund der Annahme, dass die in der Infrastruktur vorhandenen Serverknoten N heterogene Hardwareeigenschaften besitzen, variiert das Ausführungsverhalten einzelner Operatoren, Prozesse und Dienste pro Serverknoten [28, 80]. Die hardwarespezifische Bewertung dieses Verhaltens wird deshalb durch empirische Messungen vorgenommen. Auf Basis der so gewonnenen Daten kann über eine Kostenfunktion das Verhalten der Operatoren für weitere Serverknoten mit anderen Hardwareeigenschaften geschätzt werden [80, 138, 162, 161].

Das Kostenmodell basiert auf dem Advisor-Metamodell (vgl. Abschnitt 6.2.2) und definiert die Kosten der einzelnen Operatoren, die Kosten eines Prozesses, die Kosten der beteiligten Dienste sowie die Kosten einer Systemkonfiguration. Im Folgenden werden die Kernpunkte des Kostenmodells vorgestellt.

6.3.1 Operatorkosten

Zunächst werden die Kosten für die unterschiedlichen Operatortypen aus Kapitel 5 untersucht. Jeder Prozessplan P_i nutzt dabei eine Teilmenge O_i dieser Operatoren. Neben den Operatoren und ihren Beziehungen zueinander sind nach der vollständigen Prozessdefinition zudem die XML-Schemata der zu verarbeitenden Daten sowie die Kommunikationspartner S_i bekannt. Auf Grundlage der ratenbasierten Kostenmodelle aus 6.1.2 werden die Basismetriken in Tabelle 6.1 zur Leistungsbestimmung von Operatoren verwendet. Ähnlich zu [26, 28] sind einige dieser Metriken hardwareabhängig (a) bzw. hardwareunabhängig (u).

Metrik	Beschreibung	Hardware[1]
r_{in}	Eingabedatenrate ($\frac{\lvert b_i \rvert}{\Delta t}$)	a/u
r_{out}	Ausgabedatenrate ($\frac{\lvert b_j \rvert}{\Delta t}$)	a/u
g_{in}	Größe Eingabebucket (in Bytes)	u
g_{out}	Größe Ausgabebucket (in Bytes)	u
s_o	Operatorselektivität	u
o_{load}	vom Operator verursachte CPU-Last (in %)	a
o_{time}	Verarbeitungszeit pro Eingabeelement (in ms)	a
r_p	hardwareabhängige Verarbeitungsrate mit $r_p = \frac{1}{o_{time}}$	a

[1] a - hardware<u>a</u>bhängig; u - hardware<u>u</u>nabhängig

Tabelle 6.1: Basismetriken der Operatoren.

Die Ausgaberate r_{out} eines Operators o_j wird grundsätzlich durch dessen Eingaberate r_{in} oder dessen hardwareabhängige Verarbeitungszeit o_{time} pro Bucket und damit durch dessen hardwareabhängige Verarbeitungsrate r_p bestimmt. r_{out} entspricht damit allgemein dem Minimum (der langsameren Rate) aus r_{in} und r_p mit

$$r_{out} = min(r_{in}, r_p). \tag{6.1}$$

Somit gestaltet sich r_{out} entweder hardwareabhängig bei $r_{in} > r_p$ bzw. hardwareunabhängig bei $r_{in} < r_p$. Bei der Betrachtung von r_{in} im Kontext eines isoliert laufenden Operators wird r_{in} als hardwareunabhängig angenommen. Ist der Operator jedoch Bestandteil einer Operatorkette, so entspricht seine Eingaberate der Ausgaberate des vorgelagerten Operators, dessen Ausgaberate wiederum hardwareabhängig sein kann.

Die Größen für Eingabe- und Ausgabedaten g_{in} bzw. g_{out} werden im Rahmen dieser Arbeit in Bytes ihrer XML-Repräsentation angegeben und zur Abschätzung potenzieller Kommunikationskosten verwandt. Diese Metriken sind hardwareunabhängig und lassen sich durch andere Bewertungsmodelle ersetzen.

Einfluss der Operatorspezifika auf die Basismetriken

Bevor das hardwareabhängige Verhalten der Operatortypen in Abhängigkeit unterschiedlicher Einflussgrößen ermittelt wird, beinhalten die Tabellen 6.2 bis 6.5 implementierungsnahe, jedoch hardwareunabhängige Aussagen über die Funktionsweise der Operatortypen und ihren Einfluss auf einen Teil der Basismetriken. Im Speziellen werden der Einfluss der Funktion f des Operators und der Einfluss seiner Selektivität s_o auf die Ausgaberate r_{out} sowie die Auswirkungen von f auf die ausgehenden Bucketgrößen g_{out} untersucht. Die Auswirkungen eines Operators o_i auf

seine Ausgaberate r_{out} hat direkten Einfluss auf die Eingaberate r_{in} des nachfolgenden Operators o_{i+1}.

Operator	Ausgaberate r_{out}	Ausgabegröße g_{out}
TInvoke	$r_{out} = min(r_{in}, r_p)$	$g_{out} = f_{service}(g_{in})$
SInvoke	$r_{out} = s_{service} \cdot min(r_{in}, r_p)$	$g_{out} = f_{service}(g_{in})$
Receive	$r_{out} = n \cdot min(r_{src}, r_p)$	$g_{out} = \frac{g_{src} - g_c}{n} + g_c$
Reply	–	–

Tabelle 6.2: Interaktionsorientierte Operatoren.

Aus der Klasse der *interaktionsorientierten* Operatoren beeinflussen die Operatoren `TInvoke`, `SInvoke` und `Receive` die Ausgaberate und die Ausgabegröße sehr unterschiedlich. Während `TInvoke` die synchrone Kommunikation mit traditionellen Diensten repräsentiert und somit die Ausgaberate nicht verändert, ist die Ausgaberate des `SInvoke`-Operators abhängig von den Ein- / Ausgabebeziehungen des *strombasierten Dienstes* und damit von dessen Selektivität $s_{service}$. Die Ausgabegröße g_{out} der Buckets ist dabei direkt an die Funktion f des Dienstes gekoppelt und steht über die bekannten Schema-Informationen der Dienste zur Verfügung. Der `Receive`-Operator beeinflusst die Eingangsdatenrate des Prozesses r_{src} dahingehend, dass die eingehenden Buckets entsprechend der Partitionierungsvorschrift für jede Partition innerhalb eines Datenelementes in r_{src} ein separates Ausgabebucket erstellen und sich somit r_{out} um die Anzahl der Partitionen n in r_{src} vervielfacht. Somit entsprechen die Selektiviät s_o des Operators der Anzahl der Partitionen n mit $s_o = n$ und r_{out} der Eingabedatenrate für einen klassischen DIA-Prozess. Die Ausgabegröße g_{out} steigt dabei nur um die neu generierten Bucket-Container g_c pro Partition.

Operator	Ausgaberate r_{out}	Ausgabegröße g_{out}
Copy	$r_{out} = 2 \cdot min(r_{in}, r_p)$	$g_{out} = g_{in}$
End	–	–
Route	$r_{out} = r_{out_1} + r_{out_2} = min(r_{in}, r_p)$ mit $r_{out_1} = s \cdot min(r_{in}, r_p)$ und $r_{out_2} = (1 - s) \cdot min(r_{in}, r_p)$	$g_{out} = g_{in}$
Signal	–	–

Tabelle 6.3: Kontrollflussorientierte Operatoren.

Die Klasse der *kontrollflussorientierten* Operatoren beinhaltet die Operatoren `Copy`, `End`, `Route` und `Signal`. Der `Copy`-Operator kopiert eingehende Buckets und verdoppelt damit seine Ausgabedatenrate mit $2 \cdot r_{in}$. Die Ausgabedatenrate wird dabei auf zwei parallele ausgehende Datenflüsse aufgeteilt, wodurch pro Datenfluss die Ausgaberate r_{out} der Eingaberate r_{in} entspricht. Der `Route`-Operator verändert die Ausgaberate r_{out} nicht, verteilt jedoch die Ausgabebuckets r_{out} auf die zwei ausgehenden, parallelen Datenflüsse entsprechend der Selektivität s des Prädikates. Beide Operatoren verändern die Größe g_{out} der Ausgabebuckets nicht.

Operator	Ausgaberate r_{out}	Ausgabegröße g_{out}
Empty	$r_{out} = min(r_{in}, r_p)$	$g_{out} = g_{in}$
Filter	$r_{out} = s \cdot min(r_{in}, r_p)$	$g_{out} = g_{in}$
Groupby	$r_{out} = \frac{groups}{window}$	const
Join	$r_{out} = min(r_{in_1} \cdot r_{in_2} \cdot t_{window}, r_p) \cdot s$	$g_{out} = g_{in_1} + g_{in_2} - g_c$
Orderby	$r_{out} = \frac{r_{in}}{t_{window}}$	$g_{out} = g_{in}$
Split	$r_{out} = g \cdot min(r_{in}, r_p)$	$g_{out} = \frac{g_{in} - g_c}{n} + g_c$
Transform	$r_{out} = min(r_{in}, r_p)$	$g_{out} = f_T(g_{in})$
Union	$r_{out} = min(r_{in_1} + r_{in_2}, r_p)$	$g_{out} = \frac{r_{in_1} \cdot g_{in_1} + r_{in_2} \cdot g_{in_2}}{r_{in_1} + r_{in_2}}$

Tabelle 6.4: Datenflussorientierte Operatoren.

In der Klasse der *datenflussorientierten* Operatoren richtet sich die Ausgaberate des binären `Join`-Operators nach der Rate der eintreffenden Elemente beider Eingangsströme r_{in_1} und r_{in_2} sowie den bereits im Zeitfenster t_{window} gesammelten Elementen. Dies entspricht dem kartesischen Produkt aus r_{in_1} und r_{in_2}, da jedes eintreffende Element aus r_{in_1} mit jedem bereits eingetroffenen Element aus r_{in_e} verglichen werden muss. Durch die Selektivität s des Verbundausdrucks wird die Ausgaberate r_{out} entsprechend reduziert. Die Art und Dauer des Zeitfensters ist abhängig von der Operatorkonfiguration. Die Ausgabegröße g_{out} wird durch die Parameter sowie des angegebene XML-Schema definiert. Für eine erste Abschätzung der Ausgabegröße wird angenommen, dass die Ausgabegröße aus den Daten aus g_{in_1} und g_{in_2} abzüglich der Größe eines Bucketcontainers g_c besteht. Die Ausgaberate des `Groupby`-Operators richtet sich nach den Aggregationsgruppen *groups*, welche für die Dauer eines Zeitfensters angereichert werden und jeweils pro Zeitfenster ausgegeben werden. Da die Ausgabebuckets der Aggregation ein festes XML-Schema besitzen, ist deren Ausgabegröße g_{out} konstant.

Operator	Ausgaberate r_{out}	Ausgabegröße g_{out}
Arithmetic	$r_{out} = min(min(r_{in_1}, r_{in_2}), r_p)$	const
Sequence	$r_{out} \leq \frac{seq\_inst}{window}$	$\sum_{i=1}^{n\_seq} g_{seq_i} \cdot g_{in} + g_c$
Window	$r_{out} = min(r_{in}, r_p)$	$g_{out} = g_{in} + g_a$

Tabelle 6.5: Ereignisorientierte Operatoren.

Die Klasse der *ereignissorientierten* Operatoren beinhaltet die Operatortypen `Arithmetic`, `Sequence` und `Window`. Der binäre Operator `Arithmetic` berechnet das Ergebnis von zwei eintreffenden Buckets mit der definierten Funktion und generiert ein Ausgabebucket mit festem XML-Schema. Hingegen richtet sich die Ausgaberate des `Sequence`-Operators nach den aktuell im Operator vorliegenden Sequenzinstanzen (pro Sequenzschlüssel) pro Zeitfenster. Die Ausgabe erfolgt nur, wenn entweder die für eine Sequenzinstanz erforderlichen Ereignissbuckets eingetroffen sind oder das Zeitfenster für die Gültigkeit dieser abgelaufen ist. Die Ausgabebuckets beinhalten

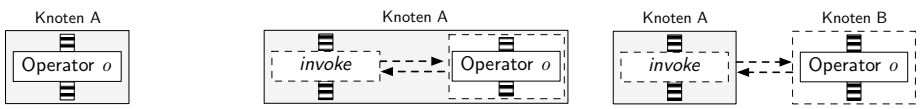
alle eingetroffenen Buckets, die zu einer Sequenznummer gehören. Die Anzahl dieser Buckets ist von der jeweiligen Sequenz abhängig.

Einflussgrößen der Operatorausführung

Dieser Abschnitt beschreibt **acht** Einflussgrößen auf die Verarbeitungszeit der Operatoren in Anlehnung an [28] und analysiert diese experimentell. Das Ziel dieser experimentellen Analyse besteht in der korrekten Schätzung von Verarbeitungszeiten und Datenvolumen der auszuführenden Prozesse. Sie bilden die Grundlage für das spätere Kostenmodell. Die acht in dieser Arbeit betrachteten Einflussgrößen auf die Verarbeitungszeit der Operatoren sind:

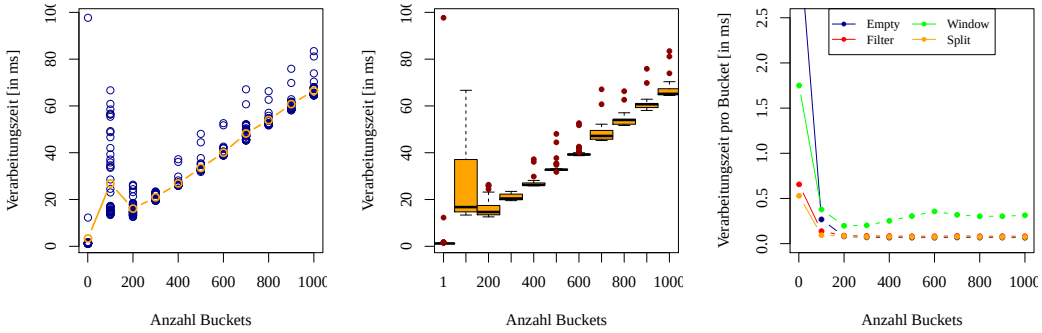
1. Anzahl Verarbeitungsbuckets,
2. Operatortyp,
3. Operatorselektivität und Bucketgröße,
4. Netzwerkkommunikation,
5. Eingabedatenrate,
6. Nebenläufige Operatorausführung (Parallelität),
7. Prozessorauslastung und
8. Hardwarekonfiguration Serverknoten.

Die experimentelle Analyse der Einflussgrößen auf die Operatoren unterscheidet *lokale* und *entfernte* Testläufe. Die Testdaten für die experimentelle Analyse der Einflussgrößen auf die Operatoren werden jeweils vor den Testläufen, d.h. bevor sie in die Eingabewarteschlange des Testoperators eingefügt werden, vollständig generiert. Im Rahmen der Analyse werden die einzelnen Operatoren zunächst lokal auf unterschiedlichen Serverknoten ausgeführt und die Verarbeitungszeiten bei den sich ändernden Einflussgrößen gemessen (vgl. Abbildung 6.4(a)). Zusätzlich werden diese Operatoren entfernt als *strombasierter Dienst* über einen lokalen `SInvoke`-Operator aufgerufen. Dabei unterscheiden die Testfälle den Aufruf eines lokalen Dienstes über XML-basierte Nachrichtenkommunikation (vgl. Abbildung 6.4(b)) bzw. den Aufruf eines entfernten Dienstes auf anderen Serverknoten (vgl. Abbildung 6.4(c)). Diese Dienste kapseln jeweils nur diesen Operator. Damit lassen sich entsprechend der Eigenschaften der Operatorfunktionalität Aussagen zu Netzwerktransfer und Seria-



(a) Lokale Operatoranalyse. (b) Operatoranalyse über lokalen Dienst. (c) Operatoranalyse über entfernten Dienst.

Abbildung 6.4: Aufbau Testumgebung der Operatoranalyse.



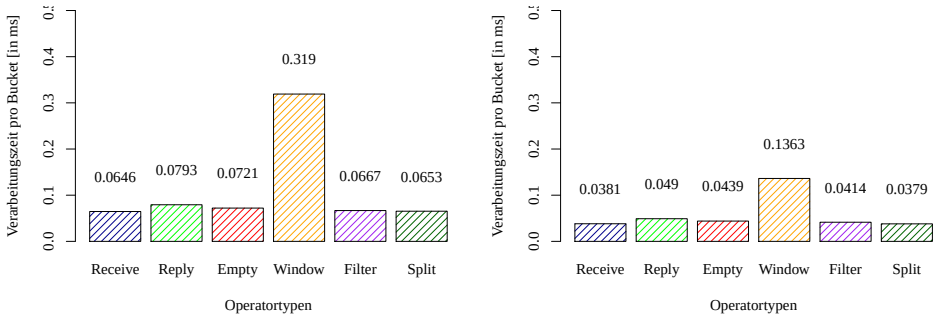
(a) Rohdaten und Mittelwert. (b) Boxplot der Rohdaten. (c) Verarbeitungszeit pro Bucket.

Abbildung 6.5: Einfluss Bucketanzahl auf Verarbeitungszeit (Knoten A).

lisierungskosten ableiten. Als Testsystem stehen fünf Knoten A bis E zur Verfügung, deren Hardwareeigenschaften im Anhang auf Seite 195 beschrieben sind.

Anzahl Verarbeitungsbuckets Zunächst wird die Verarbeitungszeit eines Operatortyps in Abhängigkeit der zu verarbeitenden Eingangsdaten und somit anhand der Anzahl eintreffender Buckets untersucht. Dazu wird die Eingabemenge schrittweise von 1 auf 1000 Buckets erhöht und die jeweilige Verarbeitungszeit gemessen. Die Messergebnisse werden am Beispiel des `Empty`-Operators exemplarisch dargestellt. Abbildung 6.5(a) zeigt die Gesamtausführungszeiten in Abhängigkeit der Eingangsdatengröße für jeweils 50 Tests sowie die entsprechende mittlere Ausführungszeit. Es ist ersichtlich, dass die Verarbeitungszeit mit der Anzahl der Buckets linear steigt. Auch die Analyse aller weiteren Operatortypen zeigt, dass die gemessenen Ausführungszeiten unabhängig von Operatortyp und ausführendem Serverknoten in Abhängigkeit der Eingabemenge proportional ansteigen. Dieses Verhalten ist in die spätere Schätzung der Verarbeitungszeiten der Operatoren einzubeziehen. Wie Abbildung 6.5(b) in Form eines Box-Whistler-Plots zeigt, fällt die Varianz der Messwerte klein aus. Abschließend stellt Abbildung 6.5(c) die mittlere Verarbeitungszeit pro Eingabebucket dar. Dabei lässt sich ablesen, dass sich die mittlere Verarbeitungszeit pro Bucket unabhängig vom Operatortyp nach einer Initialisierungsmenge von ca. 100 Buckets nahezu konstant verhält. Unter der Annahme, dass es sich um langlaufende, *stehende Prozessinstanzen* handelt, wird diese Initialisierungszeit in den weiteren Betrachtungen vernachlässigt und die Bestimmung der Verarbeitungszeiten pro Bucket erfolgt im Weiteren jeweils auf Grundlage von $n = 1000$ Eingabebuckets.

Operatortyp Bei der Analyse dieser Einflussgröße werden die einzelnen Operatortypen und deren Verarbeitungszeiten gegenübergestellt. Abbildung 6.6 zeigt die durchschnittlichen Verarbeitungszeiten pro Bucket für ausgewählte Operatortypen bei $n = 1000$ und stellt diese für Serverknoten A (6.6(a)) und B (6.6(b)) gegen-



(a) Lokale Ausführung Knoten A. (b) Lokale Ausführung auf Knoten B.

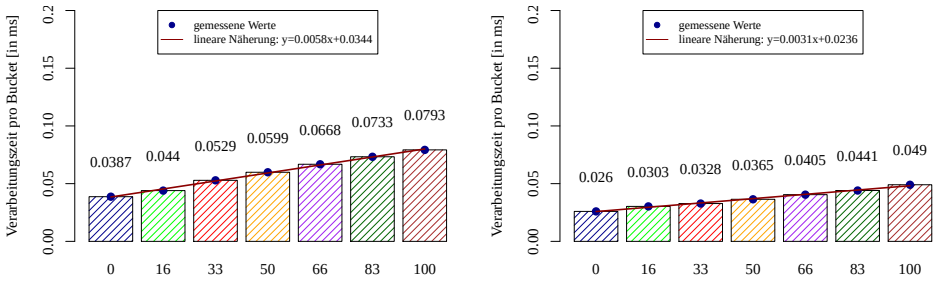
Abbildung 6.6: Einfluss Operatortyp auf Verarbeitungszeit.

über. Dabei wird die Selektivität s_o aller Operatortypen jeweils auf 1.0 gesetzt. Dies bedeutet beispielsweise 1) im Falle des `Split`-Operators, dass ein eintreffendes Bucket nicht weiter partitioniert wird und 2) im Falle des `Filter`-Operators, dass kein eintreffendes Bucket verworfen wird. Der Einfluss der Selektivität wird im nachfolgenden Abschnitt diskutiert.

Es lässt sich erwartungsgemäß erkennen, dass sich die Verarbeitungszeiten der einzelnen Operatortypen sowohl innerhalb eines Knotens als auch zwischen unterschiedlichen Knoten unterscheiden. Dabei bleiben die Verarbeitungszeiten jedoch im Verhältnis zueinander knotenübergreifend ähnlich. Es lassen sich allerdings keine Faktoren bestimmen, anhand derer sich beispielsweise die Verarbeitungszeit eines `Split`-Operators aus der Verarbeitungszeit eines `Empty`-Operators auf dem gleichen Knoten herleiten lässt. Die explizite Messung der Verarbeitungszeiten jedes einzelnen Operators auf einem Knoten ist somit unabdingbar. Die Verarbeitungszeiten einzelner Operatortypen auf einem Serverknoten lassen sich aufgrund annähernd gleicher Werte in Gruppen zusammenfassen.

Operatorselektivität und Bucketgröße Die Analyse des Einflusses der Operatorselektivität s_o bezieht sich nur auf Operatortypen, deren Selektivität ungleich 1.0 beträgt. Dies trifft in Abhängigkeit von der Operatorparametrierung beispielsweise auf die Operatortypen `Filter`, `Join`, `GroupBy` oder `Split` zu (vgl. Tabellen 6.2 bis 6.5). Beispielhaft sei der Einfluss der Selektivität auf die Verarbeitungszeit anhand der Operatortypen `Filter` und `Split` gezeigt, da diese beiden Operatortypen die Klassen der Selektivität ≤ 1.0 (Filter) sowie ≥ 1.0 repräsentieren.

Abbildungen 6.7(a) und (b) zeigen zunächst die durchschnittliche Verarbeitungszeit des `Filter`-Operators mit $s_o = \{0.0, 0.16, 0.33, 0.5, 0.66, 0.83, 1.0\}$ für die Serverknoten A und B. Diese Messergebnisse belegen, dass sich die durchschnittliche Verarbeitungszeit mit steigender Selektivität erhöht, da entsprechend mehr Buckets an die nachfolgenden Operatoren weitergegeben werden müssen. Die Verarbeitungszeit

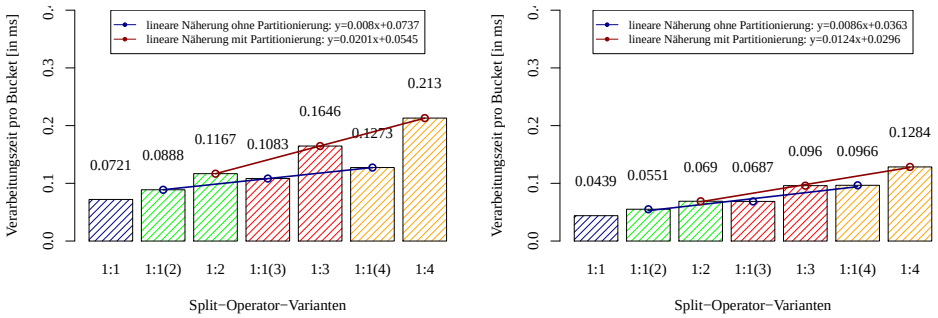


(a) Lokale Ausführung auf Knoten A. (b) Lokale Ausführung auf Knoten B.

Abbildung 6.7: Einfluss Operatorselektivität auf Verarbeitungszeit (`Filter`).

ist dabei linear von s_o abhängig und kann, wie in den Abbildungen dargestellt, sehr gut durch eine Näherungsfunktion approximiert werden.

Zusätzlich wurde der Einfluss der Eingabebucketgröße g_{in} im Zusammenhang mit dem `Split`-Operator untersucht, welcher die Datenpartitionierung der Eingangsbuckets vornimmt und dessen Selektivität $s_o \geq 1.0$ beträgt. Dazu bildet Abbildung 6.8 die mittleren Verarbeitungszeiten pro Bucket für unterschiedliche Parameterkonfigurationen des `Split`-Operators ab. Der `Split`-Operator wertet seinen Partitionierungsausdruck über dem gesamten Inhalt eines Eingabebuckets aus, wodurch die Verarbeitungszeit linear mit der in einem Eingabebucket enthaltenen Anzahl n der Partitionen, und damit mit der Bucketgröße g_{in}, steigt. Dies geschieht unabhängig davon, ob tatsächlich eine physische Partitionierung erfolgt. Der Mehraufwand der physischen Partitionierung, welcher lediglich in der Generierung der Ausgabebucketcontainer und dem Umhängen des Wurzelknotens der einzelnen Partitionen besteht, scheint hingegen gering auszufallen. In Messungen anderer Operatortypen bestätigte sich eine lineare Abhängigkeit der Verarbeitungszeit o_{time} von der Bucketgröße g_{in}, wobei sich deren Stärke zwischen verschiedenen Operatortypen unterscheidet.

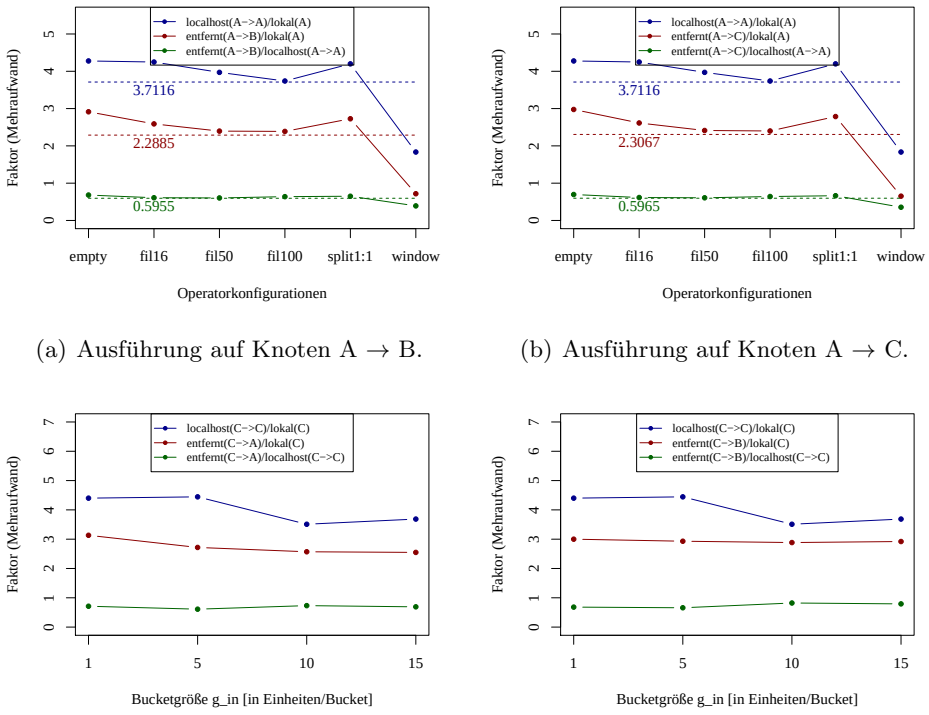


(a) Lokale Ausführung auf Knoten A. (b) Lokale Ausführung auf Knoten B.

Abbildung 6.8: Einfluss Operatorselektivität auf Verarbeitungszeit (`Split`).

Netzwerkkommunikation Für die Kommunikation eines Prozesses mit einem entfernten Dienst müssen die Daten zunächst aus der internen Darstellung in XML-Text transformiert (serialisiert), über das Netzwerk transportiert und schließlich beim Empfänger wieder in die interne Darstellung umgewandelt (deserialisiert) werden. Die dabei entstehenden Kosten sind abhängig von Menge und Struktur der zu übermittelnden Daten und unabhängig vom aufgerufenen Dienst [80].

Mithilfe des Konzeptes der *strombasierten Prozessausführung* lassen sich neben *atomaren Diensten* auch einzelne Operatortypen direkt als entfernte Dienste in den Prozessfluss integrieren (vgl. Abschnitt 5.2.4). Um den Einfluss der Serialisierungs- und Deserialisierungskosten auf die Verarbeitungszeiten der einzelnen Operatoren zu ermitteln, werden die zuvor lokal ausgeführten Operatoren (vgl. Abbildung 6.4(a)) als Dienst gekapselt und zunächst als lokaler, nachrichtenbasierter Dienst auf demselben Knoten betrieben und über einen `SInvoke`-Operator aufgerufen (vgl. Abbildung 6.4(b)). Dadurch werden zwar einerseits Transferzeiten für den Netzwerktransport vermieden, andererseits erhöht sich jedoch dadurch die Prozessorauslastung des Knotens und der `SInvoke`-Operator sowie der Dienst beeinflussen sich gegenseitig. Als dritte Testreihe werden die als Dienste gekapselten Operatoren auf entfernten Serverknoten aufgerufen (vgl. Abbildung 6.4(c)).



(a) Ausführung auf Knoten A → B.

(b) Ausführung auf Knoten A → C.

(c) Ausführung Split auf Knoten C → A.

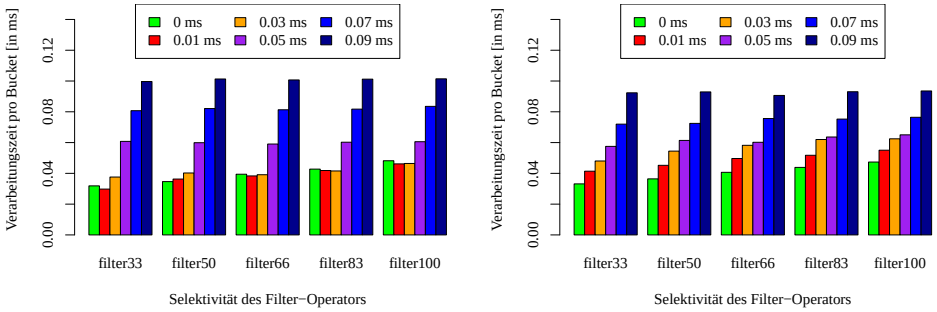
(d) Ausführung Split auf Knoten C → B.

Abbildung 6.9: Einfluss Netzwerkkommunikation auf Operatortypen.

Mit diesen drei Testreihen lässt sich der Mehraufwand der XML-basierten Kommunikation empirisch messen und als Faktor auf die lokalen Verarbeitungszeiten dazurechnen. Abbildungen 6.9(a) und 6.9(b) zeigen die errechneten Faktoren für verschiedene Operatortypen und Selektivitäten. Darin wird deutlich, dass der Mehraufwand für viele Operatortypen und ihre Parametrierungen gleich ausfällt. Der Mehraufwand einer XML-basierten Kommunikation für einen lokalen Dienstaufruf (`localhost`) liegt dabei im Vergleich zur lokalen Operatorausführung (`lokal`) in Abbildung 6.9(a) im Durchschnitt bei einem Faktor von 3.71. Der Aufruf eines entfernten Operators auf anderen Serverknoten bedingt hingegen weniger Mehraufwand (Faktor 2.89). Dies scheint zunächst überraschend, da der Datentransfer über das Netzwerk bei lokalem Dienstaufruf (`localhost`) gegenüber dem entfernten Dienstaufruf (`entfernt`) wegfällt. Jedoch werden die vorhandenen CPU-Ressourcen auf dem Knoten durch den `SInvoke`-Operator und den lokalen Dienst so stark beansprucht, dass durch Verteilung des Dienstes auf einen anderen Serverknoten ein Teil der Deserialisierung bereits nebenläufig auf diesem entfernten Serverknoten stattfindet und dessen CPU-Ressourcen nutzt. Somit erweist sich die lokale XML-basierte Kommunikation mit den gegebenen Hardwareressourcen als ineffizienteste Methode des Datenaustauschs.

Steigt das Datenvolumen pro Verarbeitungsbucket (vgl. Abbildung 6.9(c) und 6.9(d) am Beispiel des `Split`-Operators), bleiben jedoch alle drei Faktoren der Dienstkommunikation konstant und es lässt sich keine Abhängigkeit dieser Faktoren von der Bucketgröße g_{in} erkennen.

Eingangsdatenrate Die Eingangsdatenrate r_{in} wirkt sowohl auf die Verarbeitungszeit des Operators als auch auf die Prozessorauslastung des Serverknotens im selben Zeitraum. Dabei gilt Formel 6.1, wonach die Ausgaberate entweder von der Eingaberate r_{in} oder der Verarbeitungsrate r_p abhängt. Dazu wurden am Beispiel des `Filter`-Operators die Eingabedatenrate von $t = \{0.0ms, \ldots, 0.09ms\}$ variiert und die Verarbeitungszeit auf Knoten A gemessen (vgl. Abbildung 6.10(a)). Diese Einga-

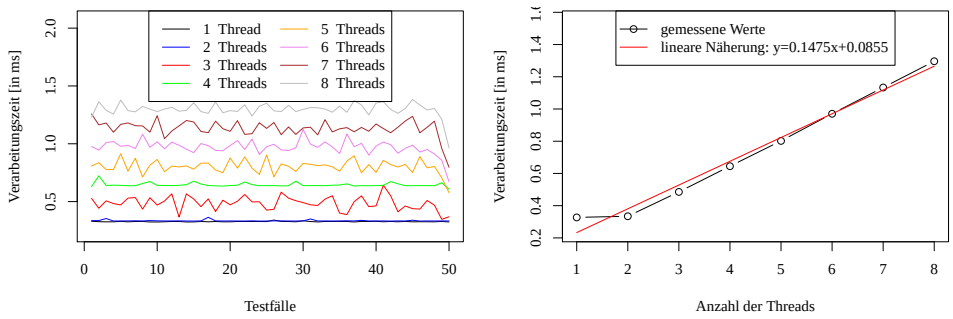


(a) Lokale Ausführung auf Knoten A. (b) Lokale Ausführung auf Knoten B.

Abbildung 6.10: Einfluss der Eingangsdatenrate auf Verarbeitungszeit (`Filter`).

bedatenraten orientieren sich an der tatsächlichen Verarbeitungszeit des Operators auf dem Knoten. Es ist zu erkennen, dass die Ausgangsdatenrate r_{out} bis zu einer Eingangsdatenrate r_{in} von $t = 0.04ms$ von der Verarbeitungszeit o_{time} des Operators abhängt und erst mit $r_{in} < 0.04ms$ der Eingabedatenrate mit $r_{out} == r_{in}$ entspricht.

Nebenläufige Operatorausführung (Parallelität) Des Weiteren wird die nebenläufige Ausführung von Operatorinstanzen und ihr Einfluss auf die einzelnen Verarbeitungszeiten untersucht. Dabei wird angenommen, dass ein Operator die ihm übergebenen Daten schnellstmöglich verarbeitet und somit jeweils eine CPU-Kernauslastung von 100% generiert. Für diesen Test (vgl. Abbildung 6.11(a)) werden die Anzahl der nebenläufigen Operatorinstanzen (`Threads`) des `Window`-Operators schrittweise von 1 auf 8 inkrementiert und die durchschnittlichen Verarbeitungszeiten der Operatoren für jeden der 50 Testfälle gemessen. Wie sich erkennen lässt, steigt die durchschnittliche Verarbeitungszeit mit jedem zusätzlichen nebenläufigen Aufruf. Abbildung 6.11(b) zeigt den Anstieg der Verarbeitungszeit für jeden weiteren nebenläufigen Aufruf sowie dessen Approximation durch eine linearen Näherungsgleichung.



(a) Lokale Ausführung Window-Operator (Knoten A).

(b) Anstieg Verarbeitungszeit nebenläufiger Ausführungen (Knoten A).

Abbildung 6.11: Einfluss nebenläufiger Operatorausführung auf Verarbeitungszeit (`Window`).

Prozessorauslastung Weiterhin wird der Einfluss der Datenraten auf die verursachte Prozessorlast untersucht. Dazu werden die Datenraten mit $t = \{0ms, \ldots, 1ms\}$ variiert und die CPU-Last gemessen. Abbildung 6.12 beinhalten die Messergebnisse für den `Filter`-Operator auf Knoten B. Dabei zeigt sich, dass sich die Prozessorlast mit sinkender Datenrate entsprechend verringert (6.12). Ab einer Datenrate von $0.2ms$ steht die Prozessorlast in linearer Abhängigkeit von der Datenrate und lässt sich mithilfe einer linearen Näherungsgleichung schätzen. Die rote Linie sowie die Zahlenwerte geben die Mittelwerte der jeweiligen Datenrate in Prozent an.

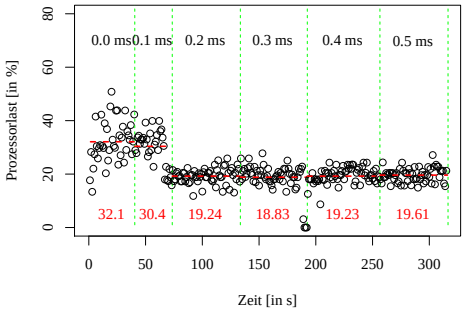


Abbildung 6.12: Prozessorlast (Filter) bei lokaler Ausführung auf Knoten B.

Hard- und Softwarekonfiguration des Serverknoten Im Allgemeinen bedingen die unterschiedlichen Hard- und Softwarekonfigurationen der Serverknotens auch unterschiedliche Verarbeitungszeiten der einzelnen Operatortypen. Allerdings zeigen die Messungen zu den Verarbeitungszeiten in Abbildung 6.6 eine ähnliche Ausführungscharakteristik der einzelnen Operatortypen auf den unterschiedlichen Serverknoten. Diese Beobachtungen können genutzt werden, um gegebenenfalls unvollständige Statistiken über Verarbeitungszeiten von Operatortypen O oder Verarbeitungszeiten von atomaren Diensten S auf spezifischen Serverknoten abzuleiten. Für eine solche Statistikschätzung werden Hardwarekoeffizienten in Form eines Faktors $hw_{n_1 \to n_2}$ berechnet, welcher dazu dienen kann, die Verarbeitungszeiten von Operatoren eines Knotens n_2 aus den entsprechenden Verarbeitungszeiten eines Knotens n_1 zu schätzen. Die so berechneten Werte können später während der Ausführung des Prozesses auf Knoten n_2 durch die tatsächlichen Statistiken ersetzt und der Koeffizient entsprechend für diesen Operatortyp angepasst werden. Die Formel zur Berechnung eines Hardwarekoeffizienten lautet:

$$hw_{n_1 \to n_2} = \frac{\sum_{i=1}^{k} \frac{o_{time,i}(n_2)}{o_{time,i}(n_1)}}{k} \tag{6.2}$$

Der Hardwarekoeffizient errechnet sich aus der Gesamtmenge aller k vorhandenen Verarbeitungszeiten der Operatortypen. Jede Verarbeitungszeit $o_{time,i}$ entspricht einem Operatortyp mit jeweils einer Ausprägung von Eingangsdatengröße g_{in} und Operatorselektivität s_o. Dabei werden nur diejenigen Verarbeitungszeiten bzw. Operatorkonfigurationen einbezogen, welche auf beiden Systemen existieren.

6.3.2 Prozesskosten

Anhand der ermittelten Statistiken der Operatoren aus dem vorherigen Abschnitt, werden im Anschluss die Kosten für komposite, strombasierte Prozesse berechnet. Zunächst wird jedoch jeder Prozess isoliert betrachtet und die minimalen Prozesskosten bezüglich Kommunikationsvolumen p_{com} sowie Verarbeitungszeit bzw. -kosten p_{time} berechnet.

Das Kommunikationsvolumen p_{com} eines Prozesses ergibt sich aus der Summe der Interaktionen des Prozesses mit externen Diensten, deren Funktionalität über `SInvoke`-Operatoren in den Prozessgraphen integriert wurde. Algorithmus 1 (*CalculateMinimumProcessDataVolume*) berechnet das minimale Kommunikationsvolumen p_{com} eines Prozesses bei einer gegebenen Menge N an Serverknoten. Hierbei wird für jeden Knoten berechnet, wie groß das Datenvolumen des Prozesses ausfällt, wenn der Prozess auf diesem Knoten ausgeführt wird.

Algorithmus 1 CalculateMinimumProcessDataVolume

Benötigt: Prozess P, Knotenmenge N, Datenquelle D_{src}

1: $min_{com} \leftarrow \infty, min_n \leftarrow \oslash, result_P \leftarrow \oslash$
2: **for** $n = 1$ to $|N|$ **do** *// ∀ Knoten n in der Knotenmenge N*
3: $p_{com} \leftarrow 0, result_{temp} \leftarrow \oslash$
4: $O \leftarrow list(P.getOperators())$ order ASC *//sortiere in Reihenfolge der Benutzung*
5: **for** $j = 1$ to $|O|$ **do** *// ∀ Operatoren o_j in O*
6: $o_{j_{com}} \leftarrow 0$
7: $o_j.r_{in} \leftarrow P.getPreceedingOutputRate(o_j)$
8: $o_j.g_{in} \leftarrow P.getPreceedingOutputSize(o_j)$
9: **if** $o_j.type == invoke$ **then** *// Wenn Invoke-Operator*
10: $S \leftarrow o_j.getDestinationService()$
11: $serviceNodes \leftarrow getAvailableNodes(S)$
12: **if** $\neg(n \in serviceNodes)$ **then** *// S nicht auf aktuellem Knoten verfügbar?*
13: *// Berechne o_{com} für diesen Operator*
14: $o_{j_{com}} \leftarrow o_j.g_{in} \cdot o_j.r_{in}$ *//Berechne o_{jcom} für Dienstanfrage*
15: $o_j.r_{out} \leftarrow calculateServiceOutputRate(o_j, S)$
16: $o_j.g_{out} \leftarrow calculateServiceOutputSize(o_j, S)$
17: $o_{j_{com}} \leftarrow o_{j_{com}} + (o_j.g_{out} \cdot o_j.r_{out})$
18: $p_{com} \leftarrow p_{com} + o_{j_{com}}$
19: **else** *// Lokale Ausführung von S möglich*
20: *// Aktualisiere nur Statistiken zur Nutzung in nachfolgenden Operatoren*
21: $o_j.r_{out} \leftarrow calculateOutputRate(o_j)$
22: $o_j.g_{out} \leftarrow calculateOutputSize(o_j)$
23: **end if**
24: **else** *// kein Invoke-Operator*
25: *// Aktualisiere nur Statistiken zur Nutzung in nachfolgenden Operatoren*
26: $o_j.r_{out} \leftarrow calculateOutputRate(o_j)$
27: $o_j.g_{out} \leftarrow calculateOutputSize(o_j)$
28: **end if**
29: $result_{o_j} \leftarrow (o_j.g_{out}, o_j.r_{out}, o_{j_{com}})$
30: $result_{tmp} \leftarrow result_{tmp} \cup (result_{o_j})$
31: **end for**
32: **if** $p_{com} < min_{com}$ **then** *// p_{com} minimal?*
33: $min_{com} \leftarrow p_{com}$
34: $min_n \leftarrow n$
35: $result_P \leftarrow result_{tmp}$
36: **end if**
37: **end for**
38: **return** $result_P, n$

Algorithmus 2 CalculateMinimalProcessingTimes

Benötigt: Prozess P, Knotenmenge N

```
1:  min_costs ← ∞; min_node ← ⊘
2:  for n = 1 to |N| do                          // ∀ Knoten n in der Knotenmenge N
3:    result_temp ← ⊘
4:    p_time ← 0
5:    O ← list(P.getOperators()) order ASC   //sortiere in Reihenfolge der Benutzung
6:    for j = 1 to |O| do                        // ∀ Operator o im Prozessgraphen P
7:      localCosts ← 0, connection ← ⊘
8:      o_j.r_in ← P.getPreceedingOutputRate(o_j)
9:      o_j.g_in ← P.getPreceedingOutputSize(o_j)
10:     if o_j.type == invoke then                 // Operator vom Typ Invoke?
11:       S ← o_j.getDestinationService()
12:       connection ← getMinConnection(n, S)
13:       o_j,time ← connection.getProcessingTime()
14:     else                                       // Wenn kein Invoke-Operator
15:       o_j.r_out ← calculateOutputRate(o_j)
16:       o_j.g_out ← calculateOutputSize(o_j)
17:       o_j,time = getLocalProcessingTime(o_j, n)
18:       o_j,time ← linearFit(o_j,time, o_j.sel) // Kostenanpassung für Selektivität
19:     end if
20:     o_j,time ← linearFit(o_j,time, o_j.g_out) //Kostenanpassung für Bucketgröße
21:     p_time ← p_time + localCosts
22:     result_o_j ← (o_j.r_out, localCosts, connection)
23:     result_tmp ← result_tmp ∪ result_p_j
24:   end for
25:   if p_time < min_costs then          // Sind die berechneten Kosten p_time minimal?
26:     min_costs ← p_time
27:     min_node ← n
28:     result_P ← result_tmp
29:   end if
30: end for
31: return result_P, min_node
```

Dazu iteriert der Algorithmus über alle Operatoren (Zeile 5). Handelt es sich bei dem aktuellen Operator um einen `Invoke`-Operator (Zeile 9), wird der aufgerufene Dienst extrahiert. Steht der Dienst nur auf entfernten Knoten zur Verfügung (Zeile 12), wird das entstehende Datenvolumen mithilfe der aktuell am Operator anliegenden Eingangsdatenrate r_{in} sowie dessen Eingangsbucketgröße g_{in} geschätzt. Existiert der Dienst auf demselben Knoten (Zeile 19) oder handelt es sich bei dem aktuellen Operator nicht um einen `Invoke`-Operator (Zeile 24), fällt kein Datenvolumen an. Jedoch müssen die Ausgabedatenraten dieser lokalen Operatoren geschätzt werden, um die Modifikationen der Quelldaten und Bucketgröße entlang des Prozessgraphs zu propagieren, damit realistische Werte für die Datenvolumenberechnung an den `Invoke`-Operatoren anliegen.

Operator	$o_{time}(A)$	$o_{time}(A \rightarrow B)$	$o_{time}(A \rightarrow C)$
Filter ($s = 0,16$)	0,0615	0,1420 (+2.31)	0,1469 (+2.39)
Filter ($s = 0,50$)	0,0768	0,1751 (+2.28)	0,1766 (+2.30)
Window	0,3273	0,7527 (+2.30)	0,7953 (+2.32)
Split ($s = 4$)	0,2442	0,5641 (+2.31)	0,5616 (+2.30)

Tabelle 6.6: Verarbeitungszeiten für lokale und entfernte Operatoraufrufe.

Auch die minimale Verarbeitungszeit p_{time} eines Prozesses wird zunächst isoliert betrachtet. Algorithmus 2 (*CalculateMinimalProcessingTimes*) iteriert dazu über alle verfügbaren Knoten und berechnet die Verarbeitungskosten des Prozesses auf diesem Knoten. Dazu werden ähnlich zu Algorithmus 1 der Prozessgraph traversiert (Zeile 6) und für jeden Operator dessen Verarbeitungszeiten $o_{j,time}$ entweder aus vorhandenen Statistiken ausgelesen oder über den im vorherigen Abschnitt eingeführten Hardwarekoeffizienten $hw_{n_1 \rightarrow n_2}$ geschätzt (Zeilen 13 und 17). Zusätzlich werden diese Verarbeitungszeiten in Abhängigkeit der Eingangsdatenrate r_{in} und der Eingabebucketgröße g_{in} durch eine Näherungsgleichung entsprechend der Ergebnisse des vorherigen Abschnittes angepasst und die Ausgaberaten r_{out} und Ausgabegrößen g_{out} der entsprechenden Operatoren aktualisiert. Tabelle 6.6 zeigt die Basisverarbeitungszeiten ausgewählter Operatoren für die lokale Ausführung auf den beiden Knoten A und B sowie für die entfernte Ausführung von Knoten A zu Knoten B ($A \rightarrow B$) und von Knoten A zu C ($A \rightarrow C$). Liegen einzelne Werte nicht vor, lassen sich die Verarbeitungszeiten der entfernten Aufrufe über die Faktoren bzw. Näherungsgleichungen schätzen.

6.3.3 Kosten einer Systemkonfiguration

Nachdem im vorherigen Abschnitt die Kostenaggregation einzelner Operatoren zu Prozessen betrachtet wurde, beschreibt Algorithmus 3 (*CalculateSystemCosts*) die Aggregation der Kosten von Prozessen und Diensten für eine gegebene Systemkonfiguration k. Die dem Algorithmus übergebene Systemkonfiguration k beinhaltet sowohl konkrete Dienste und Prozesse als auch deren Zuordnung auf konkrete Serverknoten der Infrastruktur. Dabei wird zunächst die CPU-Last für jeden Knoten mit den ihm zugewiesenen Komponenten geschätzt (Zeile 5). Liegt die geschätzte CPU-Last bei einem Knoten der Konfiguration über 80 Prozent, werden die gesamte Systemkonfiguration als ungültig markiert und alle weiteren Berechnungen abgebrochen (Zeile 8). Für den Fall, dass kein Knoten durch die ihm zugewiesenen Komponenten überlastet wird, werden Kommunikationsvolumen und Verarbeitungszeiten für alle Prozesse berechnet (Zeilen 13 und 14) und als Kosten für die Systemkonfiguration aggregiert (Zeilen 15 und 16).

Algorithmus 3 CalculateSystemCosts

Benötigt: Systemkonfiguration k, Knotenmenge N

```
1: k_com, k_time ← 0
2: result_k ← ∅
3: // Berechne CPU-Load für jeden Knoten in aktueller Systemkonfiguration
4: for n = 1 to |N| do                          // ∀ Knoten n in Systemkonfiguration k
5:    nodeCpuLoad ← predictCpuLoad(k, N)
6:    if nodeCpuLoad > 80 then                  // Ist Knoten ausgelastet?
7:       // Ungültige Konfiguration
8:       return ∅
9:    end if
10: end for
11: // Bestimme Datenvolumen und Verarbeitungszeit für aktuelle Systemkonfiguration
12: for P = 1 to |P| do                         // ∀ Prozesse P in Konfiguration k
13:    P_com ← Berechne das Datenvolumen von Prozess P
14:    P_time ← Berechne die Verarbeitungszeit von Prozess P
15:    k_com ← Sys_com + P_com
16:    k_time ← Sys_time + P_time
17:    result_tmp ← result_k ∪ (P, P_com, P_time)
18: end for
19: return result_k
```

6.3.4 Zusammenfassung

In den vorherigen Abschnitten zum Kostenmodell des *Infrastruktur-Design-Advisors* erfolgte zum einen die theoretische Betrachtung der hardwareunabhängigen, aber operatorspezifischen Metriken der Ausgaberate r_{out} und des Datenvolumens o_{com} jedes einzelnen Operatortyps. Des Weiteren wurden die beiden hardwareabhängigen Operatormetriken *Verarbeitungszeit* o_{time} und *Prozessorauslastung* o_{load} vorgestellt und acht Einflussgrößen darauf untersucht. Auf Grundlage der drei Metriken o_{com}, o_{time} sowie o_{load}, welche die Kosten der Operatoren (o) beschreiben, wurden Algorithmen vorgestellt, welche diese Kosten auf Prozess- (p) und Systemebene (k) aggregieren und in einem naiven Ansatz minimieren.

Im Folgenden wird das dem *Infrastruktur-Design-Advisor* zugrundeliegende Kostenmodell vollständig definiert. Dabei bildet die Summe der Operatorkosten die entsprechenden Prozesskosten und die Summe der Prozesskosten die jeweiligen Systemkosten. Die Kosten für Operatoren (o), Prozesse (p), Dienste (s) und Systemkonfigurationen (k) werden allgemein in Form von $c_{<Metrik>}(< Komponente >)$ definiert.

$$c_{com}(s) = s_{com} \tag{6.3}$$

$$c_{time}(s) = s_{time} \tag{6.4}$$

$$c_{load}(s) = s_{load} \tag{6.5}$$

$$c_{com}(o) = \begin{cases} fit_s \cdot fit_g \cdot \{o_{r,com} | s_{com}\}, & \text{wenn } o \texttt{ SInvoke}\text{-Operator} \\ 0, & \text{sonst} \end{cases} \tag{6.6}$$

$$c_{time}(o) = \begin{cases} fit_s \cdot fit_g \cdot o_{time}, & \text{wenn } \frac{1}{r} \leq fit_s \cdot fit_g \cdot o_{time} \\ \frac{1}{r}, & \text{sonst} \end{cases} \tag{6.7}$$

$$c_{load}(o) = \begin{cases} fit_r \cdot o_{load}, & \text{wenn } \frac{1}{r} \geq 1 \\ 95, & \text{sonst} \end{cases} \tag{6.8}$$

$$c_{com}(p) = \sum_{i=1}^{|O|} c_{com}(o_i) \tag{6.9}$$

$$c_{time}(p) = \sum_{i=1}^{|O|} c_{time}(o_i) \tag{6.10}$$

$$c_{load}(p) = \sum_{i=1}^{|O|} c_{load}(o_i) \tag{6.11}$$

$$c_{load}(n) = \sum_{j=1}^{|P|} c_{load}(s_j) + \sum_{m=1}^{|S|} c_{load}(s_m) = \sum_{j=1}^{|P|} \sum_{l=1}^{|O|} C_{load}(o_l) + \sum_{m=1}^{|S|} c_{load}(s_m) \tag{6.12}$$

$$c_{com}(k) = \sum_{j=1}^{|P|} c_{com}(p_j) = \sum_{j=1}^{|P|} \sum_{i=1}^{|O|} c_{com}(o_i) \tag{6.13}$$

$$c_{time}(k) = \sum_{j=1}^{|P|} c_{time}(p_j) + \sum_{m=1}^{|S|} c_{time}(s_m) = \sum_{j=1}^{|P|} \sum_{i=1}^{|O|} c_{time}(o_i) + \sum_{m=1}^{|S|} c_{time}(s_m) \tag{6.14}$$

$$c_{load}(k) = \sum_{j=1}^{|N|} c_{load}(n_j) \tag{6.15}$$

Die drei beschriebenen Metriken *Datenvolumen*, *Verarbeitungszeit* und *Prozessorlast* liegen auch für atomare Dienste s vor. Bei der Berechnung des generierten Datenvolumens eines Operators $c_{com}(o)$ werden nur `Invoke` bzw. `SInvoke`-Operatoren betrachtet, da diese einen entfernten Dienst aufrufen und Netzwerktransfer verursachen. Die dabei entstehenden Kosten richten sich nach dem aufgerufenen Dienst s bzw. dem entfernten Operator o_r, dessen tatsächliche Kommunikationskosten über die Näherungsgleichungen fit_s und fit_g für die arbeitslastabhängige Selektivität und Bucketgröße angepasst werden. Die Verarbeitungskosten $c_{time}(o)$ eines Operators werden entweder durch die Eingaberate oder durch die hardwarespezifische Verarbeitungszeit mithilfe der Näherungsgleichungen fit_s und fit_g bestimmt. Ebenso ist die von einem Operator generierte Prozessorlast $c_{load}(o)$ abhängig von der Datenrate r_{in} und der Verarbeitungszeit $c_{time}(o)$ des Operators und wird gleichermaßen über eine Näherungsgleichung fit_r an die Eingaberate angepasst. Für die Verarbeitung eines Buckets wird dabei eine Prozessorauslastung von 95% angenommen. Die Aggregation der Kosten für die jeweiligen Metriken auf Prozess- und Systemebene geschieht additiv.

6.4 Verteilung ganzheitlicher Prozesse

Nachdem im vorherigen Abschnitt das dem *Infrastruktur-Design-Advisor* zugrundeliegende Kostenmodell abgeleitet und aufgestellt wurde, beschäftigt sich dieser Abschnitt mit der eigentlichen Systemoptimierung in Form der dem Optimierungsziel nach optimalen Verteilung von Diensten S und Prozessen P auf Serverknoten N. Zur Einschränkung des Suchraumes sowie zur Vereinfachung der Algorithmen gelten deshalb folgende drei Annahmen:

1. Ein Prozessplan P_i wird in der Infrastruktur nur von *einer* Prozessinstanz p_i repräsentiert. Aufeinanderfolgende Aufrufe des Prozessplans durch Prozessnutzer werden mithilfe des Konzeptes der *stehenden Prozessinstanz* (vgl. Abschnitt 6.1.2) realisiert. Somit ergibt sich mit $|P|$ die Anzahl der in der Infrastruktur vorhandenen unterschiedlichen Prozessinstanzen.
2. Die Platzierung von Prozessen und Diensten auf einem Serverknoten darf diesen nicht überlasten. Eine einzelne Prozessinstanz p_i mit ihrer Eingangsdatenrate kann einen Serverknoten n nicht überlasten. Nur in Verbindung mit weiteren Prozessinstanzen auf demselben Serverknoten ist eine Überlastung möglich. Liegt die Verarbeitungsdatenrate des Prozesses auf einem Serverknoten unter der Eingangsdatenrate des Prozesses, so kann der Prozess dennoch auf dem Serverknoten platziert werden, wobei das ungünstige Verhältnis von Eingabe- und Verarbeitungsrate durch eine Warnung in der Systemkonfiguration angezeigt wird.
3. Alle atomaren Dienste in der Infrastruktur werden über mindestens einen `Invoke`-Operator in der Prozessmenge aufgerufen. Existieren Dienstaufrufe für *atomare Dienste* außerhalb von P, wird der Prozessmenge P für jeden dieser Dienste S_i *eine* neue virtuelle Prozessinstanz p_{s_i} hinzugefügt. Diese speziellen Prozesse enthalten nur einen `SInvoke`-Operator, der genau diesen atomaren Dienst mit der Menge seiner externen Aufrufe anfragt.

S, P und N, spannen als Dimensionen den Suchraum für mögliche Systemkonfigurationen in Form von Zuordnungskombinationen, welche auf Basis der Bewertungen des Kostenmodells bei einer vollständigen Suche analysiert werden müssen, auf. Die tatsächliche Größe des Suchraumes wird dabei durch die Menge der Elemente in S, P, und N sowie die Verteilungsbeschränkungen von S auf N bestimmt. Über diese Verteilungsbeschränkungen von Diensten auf die verfügbaren Knoten lassen sich Abstufungen bei der Anzahl möglicher Systemkonfigurationen definieren:

Keine Verteilungsbeschränkung, jeder Dienst mehrfach. Im ersten Szenario existiert keine Verteilungsbeschränkung. Dienste können mehrfach auf $n \geq 1$ Knoten der Infrastruktur angeboten werden. Jedoch wird davon ausgegangen, dass ein Dienst jeweils nur einmal auf einem Serverknoten angeboten wird. Die potenziell mögliche Ausführung aller Dienste auf allen Knoten ist gleichbedeutend damit, dass alle Dienste potenziell immer auch lokal bei den zu verteilenden

Prozessen ausgeführt werden könnten. Dieser Spezialfall, der den Suchraum signifikant reduziert, da Prozesse und ihre Dienstpartner immer gemeinsam auf einem Knoten platziert werden würden, wird jedoch nicht weiter betrachtet. Unter Vernachlässigung dieses Spezialfalls, gilt das beschriebene Szenario als obere Schranke an Kombinationsmöglichkeiten für die Verteilung von P und S auf N. Die Anzahl möglicher Systemkonfigurationen ergibt sich somit

$$|K| = |N|^{|P|} \cdot |N|^{|S|\cdot|N|} = |N|^{|P|+||S|\cdot|N||}. \tag{6.16}$$

Sie beschreibt die Anzahl der Kombinationen möglicher Zuordnungen, welche bei einer vollständigen Suche überprüft werden müssen. Bei genauerer Analyse der Interaktion von Prozessen und Diensten zeigt sich, dass vielmehr die Gesamtanzahl aller `Invoke`-Operatoren $|I|$ die Ausprägungen der Dienste im System, und damit die Größe des Suchraumes, beeinflusst. Der Grund dafür liegt in der Annahme, dass ein Dienst pro Prozess mehrmals aufgerufen und dabei in Abhängigkeit von Datenrate und Bucketgröße jeweils ein anderer Serverknoten für die Dienstausführung gewählt werden kann. Dies reduziert gegebenenfalls die Anzahl an möglichen Kombinationen in diesem Szenario mit $|I| \leq |N| \cdot |S|$, wodurch sich auf Basis der Gleichung 6.16 die obere Schranke wie folgt ändert:

$$|K| = |N|^{|P|} \cdot |N|^{|I|} = |N|^{|P|+||I||}. \tag{6.17}$$

Keine Verteilungsbeschränkung, jeder Dienst einmal. Das zweite Szenario beschreibt jeden Dienst als einmalige Ausprägung innerhalb der Infrastruktur. Gegenüber dem ersten Szenario kann ein Dienst jedoch potenziell auf allen Knoten platziert werden. Dies reduziert die Anzahl möglicher Kombinationen und damit den Suchraum stark. Die Anzahl an Kombinationen hängt dabei nicht mehr von der Anzahl der `Invoke`-Operatoren im System, sondern lediglich von der Anzahl der Dienste $|S|$ ab:

$$|K| = |N|^{|P|} \cdot |N|^{|S|} = |N|^{|P|+|S|}. \tag{6.18}$$

Feste Zuordnung der Dienste. Das dritte Szenario beschreibt jeden Dienst mit vordefinierter Zuordnung. Wird jeder Dienst im System zudem nur einmalig angeboten, reduziert sich der Suchraum auf

$$|K| = |N|^{|P|}. \tag{6.19}$$

Sofern vereinzelt mehrere Ausprägungen der fest zugeordneten Dienste auf unterschiedlichen Serverknoten existieren, erhöht sich der Suchraum um die Dienste mit ≥ 1 Ausprägungen. Dieser hier vorgestellte Suchraum ergibt sich auch im Spezialfall des ersten Szenarios, wenn man davon ausgehen kann, dass die Dienste lokal mit jedem Prozess ausgeführt werden können.

Die tatsächlichen Verteilungsbeschränkungen der Dienste sowie deren eventuell mehrfache Ausprägung werden durch eine Vielzahl an rechtlichen oder technischen Ursachen bedingt, welche den tatsächlichen Suchraum zwischen den genannten Schranken aufspannen.

6.4.1 Heuristiken zur Reduzierung des Suchraumes

Um den exponentiell großen Suchraum bei einer vollständigen Suche zu reduzieren, werden im Folgenden heuristische Funktionen (kurz: Heuristiken) $h(x)$ beschrieben. Heuristiken bewerten einen Zwischenzustand bei der vollständigen Suche mithilfe von Wissen über das System und ermöglichen es, vorzeitig invalide Systemkonfigurationen von der Suche auszuschließen. Die Anforderung an $h(x)$ ist es dabei, dass sie die tatsächlichen Kosten niemals überschätzt [131, 160]. Die Unterschätzung der Kosten wird durch die Wahl der Kostenfunktionen, die der Heuristik zugrundeliegt, begründet. Je mehr Wissen über das System vorliegt, desto hochwertigere Heuristiken können definiert werden. Das Ziel von Heuristikdefinitionen ist es, die Funktion zu finden, bei der der Wert b mit $b = c(x) - h(x)$ zwischen den tatsächlichen Kosten $c(x)$ und den geschätzten Kosten $h(x)$ minimal ist. Im Folgenden werden Heuristiken diskutiert, welche den Suchraum für gültige Systemkonfigurationen einschränken.

Prozessorlast eines Knotens Sobald die Summe der Prozessorlast eines Serverknotens im Laufe der Prozesszuordnung über einen festgelegten Schwellenwert steigt, kann dieser Knoten für die weitere Suche aus dem Suchraum entfernt werden, weil ihn weitere Komponentenzuordnungen überlasten würden. Eine Systemkonfiguration ist jedoch nur gültig, wenn kein Knoten überlastet ist. Somit müssen die Kosten der Systemkonfiguration nach jedem Zuordnungsschritt aktualisiert werden, um die aktuelle Systemkonfiguration gegebenenfalls vorzeitig verwerfen zu können.

Zuordnungsbeschränkungen der Dienste Wie bereits bei der Analyse des Suchraumes im vorherigen Abschnitt erläutert, haben Verteilungsbeschränkungen und die Anzahl möglicher Dienstausprägungen starken Einfluss auf die Größe des Suchraumes. Dieses Wissen ermöglicht es während der Suche, ganze Teilbereiche des Suchraumes nicht weiter in Betracht zu ziehen. Wenn beispielsweise ein Dienst mit einer einmaligen Ausprägung durch den Suchalgorithmus bereits als platziert angenommen wurde, ist die Beachtunge dieses Dienstes auf weiteren Knoten überflüssig.

Hardwarekonfiguration Die Analyse der Verarbeitungszeiten und ihrer Abhängigkeit von der Hardwarekonfiguration im vorherigen Abschnitt ergab, dass Hardwarekoeffizienten berechnet werden können, mit denen sich nicht vorhandene Statistiken für Serverknoten vorhersagen lassen. Mithilfe dieser Hardwarekoeffizienten ist es zudem möglich, die Knoten entsprechend ihrer Leistungsfähigkeit zu bewerten und zu sortieren. Es kann davon ausgegangen werden, dass ein schlechterer Knoten mit einer Zuordnung überlastet wäre, wenn bereits ein

bessere Knoten mit den ihm zugeordneten Prozessen und Diensten überlastet ist. Diese Heuristik reduziert den Suchraum, da aufgrund ihrer Nutzung eine Vielzahl an Konfigurationen wegfallen.

Prozessgröße/-kosten Ähnlich der Sortierung der Serverknoten können Prozesse nach folgenden Kriterien geordnet werden:

1. Anzahl der Operatoren im Prozess mit $c(p) = |O|$,
2. Anzahl der `Invoke`-Operatoren im Prozess mit $c(p) = |I|$,
3. Minimales Datenvolumen des Prozesses (vgl. Algorithmus 1) und
4. Minimale Ausführungszeit des Prozesses (vgl. Algorithmus 2).

Die Sortierungskriterien 1 und 2 lassen sich einfach berechnen. Die Sortierungskriterien 3 und 4 lassen sich, wenn auch aufwendiger, mit den jeweiligen Algorithmen bestimmen. Die Idee einer sortierten Prozessliste besteht darin, bei der Zuordnung eines Prozesses zu einem Serverknoten Rückschlüsse auf die Zuordenbarkeit anderer Prozesse in der Liste zu ziehen und damit gegebenenfalls Serverknoten bei der weiteren Suche auslassen zu können.

Eingangsdatenrate Wenn die Eingangsdatenrate eines Prozesses geringer ausfällt als die Verarbeitungszeit auf dem langsamsten Serverknoten, kann ein Knoten aus der Liste gewählt werden, welcher gegenüber anderen Serverknoten einen schlechteren Leistungswert besitzt, da auch auf diesem langsameren Knoten die Gesamtverarbeitungszeit des Prozesses einzig von der Eingaberate abhängt. Dadurch können leistungsfähigere Knoten für anspruchsvollere Prozesse freigehalten werden.

Bezogen auf die vorliegende Problemstellung der Verteilung optimalen Verteilung von Prozessen und Diensten auf Serverknoten werden drei Heuristiken h für das System definiert.

$$h_{sum}(k) = c(p_1) + c(p_2) + ... + c(p_q) \tag{6.20}$$

$$h_{max}(k) = max(c(p_1), c(p_2), ..., c(p_q)) \tag{6.21}$$

$$h_{comb}(k) = w \cdot h_{sum}(k) + (1 - w) \cdot h_{max}(k) \tag{6.22}$$

Die Heuristik h_{sum} beschreibt zunächst die Addition beliebiger Prozesskosten einer Systemkonfiguration. In den weiteren Betrachtungen entsprechen die Kosten den drei bereits in Abschnitt 6.3.4 genannten Kostentypen Verarbeitungszeit $c_{time}(p)$, Kommunikationskosten $c_{com}(p)$ bzw. Prozessorlast $c_{load}(p)$. Die Heuristik h_{max} nimmt den Wert des kostenintensivsten Prozesses als Suchkriterium und versucht, zunächst diesen zu platzieren. Die dritte Heuristik h_{comb} kombiniert die beiden Heuristiken h_{sum} und h_{max} über ein Gewicht w. Vor- und Nachteile dieser drei Heuristiken werden im Abschnitt 6.5 diskutiert.

6.4.2 Allgemeine Suchalgorithmen

Der folgende Abschnitt stellt existierende Suchalgorithmen, welche den aufgespannten Suchraum durchsuchen, vor. Diese Suchalgorithmen unterscheiden sich sowohl in der Nutzung von Systemwissen mithilfe von Heuristiken, als auch in ihrer Komplexität $\mathcal{O}$. Zusätzlich unterscheiden sie sich in ihrer Eigenschaft, eine Lösung zu finden, wenn eine existiert (Vollständigkeit) bzw. die optimale Lösung zu finden (Optimalität). Grundlage dieser Algorithmen bildet die Abbildung des Suchraumes auf Baum- oder Graphstrukturen.

Deshalb wird der Suchraum der Prozess- und Dienstverteilung zunächst auf eine Baumstruktur abgebildet. Abbildung 6.13 zeigt einen beispielhaften Suchbaum mit zwei Knoten n_1 und n_2 und drei Prozessen p_1, p_2 und p_3. Dienste werden in diesem Beispiel nicht dargestellt.

Der Wurzelknoten des Baumes enthält keinerlei Zuordnung eines Dienstes oder Prozesses zu einem Knoten. Jede Ebene l im Suchbaum entspricht einer neuen zu platzierenden Komponente. Somit ergibt sich die Anzahl der Ebenen m des Baumes und damit seine maximale Tiefe mit $m = |P| + |S| - 1$. Der Verzweigungsfaktor (engl. *branching factor*) b des Baumes bei jedem Baumknoten entspricht mit $b = |N|$ der Anzahl an Knoten im System. Die potenziellen Lösungen in Form von vollständigen Zuordnungen aller Prozesse und Dienste auf Knoten und damit die eigentlichen Systemkonfigurationen k befinden sich ausschließlich in den Blattknoten des Baumes. Als gültige Lösungen k_v gelten nur diejenigen Systemkonfigurationen, welche die Prozessorlast jedes Knoten nicht überschreiten.

Zunächst werden *einfache Suchstrategien* in Anlehnung an [131] kurz vorgestellt, die kein Systemwissen voraussetzen und damit zur Klasse der uninformierten Suchen gehören. Auf diesen grundlegenden Suchstrategien bauen die danach beschriebenen informierten Suchstrategien auf.

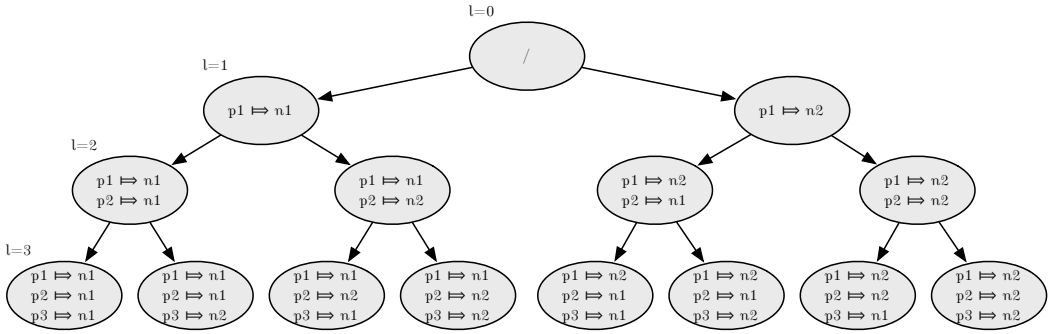


Abbildung 6.13: Aufbau des beispielhaften Suchbaumes.

Breitensuche Die Breitensuche (engl. *breadth-first-search*) beginnt bei Ebene $l = 0$ und expandiert zunächst alle Kindknoten. Danach werden die Kindknoten der Ebenen l = l+1 iterativ expandiert, bis eine Lösung gefunden oder der Suchbaum vollständig durchsucht wurde. Entsprechend werden im Beispiel des Suchbaumes der Prozess- und Dienstplatzierung nur in den Blattknoten des Baumes und damit erst nach einer vollständigen Expansion des Suchbaumes Lösungen gefunden. Speicher- $\mathcal{O}_S$ und Zeitkomplexität $\mathcal{O}_T$ betragen $\mathcal{O}(b^m)$ und sind damit exponentiell mit $\mathcal{O}(2^n)$.

Tiefensuche Die Tiefensuche (engl. *depth-first-search*) beginnt ebenfalls bei Ebene $l = 0$, expandiert jedoch zunächst entlang der jeweils ersten Kindknoten. Die Suche bricht ab, wenn eine Lösung gefunden wurde. Dies erfolgt im genannten Suchbaum nach frühestens m Schritten, im ersten Blattknoten des Baumes. Eine Optimalität der Lösung kann damit nicht gewährleistet werden. Die Zeitkomplexität der Tiefensuche entspricht mit $\mathcal{O}_T(b^m)$ der Komplexität der Breitensuche, die Speicherkomplexität entspricht jedoch $\mathcal{O}_T(b \cdot m)$ und verhält sich mit $\mathcal{O}_S(n)$ linear. Es existieren weitere Modifikationen der Tiefensuche (*begrenzte Tiefensuche* bzw. *iterative begrenzte Tiefensuche*), welche jedoch hinsichtlich der Problemstellung dieses Kapitels keine verbesserten Eigenschaften gegenüber der klassischen Tiefensuche aufweisen.

Im Folgenden werden einige, auf diesen beiden Strategien aufbauende Suchstrategien zur Durchführung informierter Suchen vorgestellt:

Greedy Search Die gierige Suche (engl. *greedy search*) nutzt Systemwissen in Form einer Heuristik, um die einzelnen Knoten des Suchbaumes zu bewerten. Im Rahmen der Prozess- und Dienstverteilung bietet sich dabei die Heuristik h_{sum} an, da diese differenzierte Kosten des Systems widerspiegelt. Die gierige Suche startet am Wurzelknoten und expandiert immer den Kindknoten mit den geringsten Kosten. Nach jeder Expansion muss die Prozessorlast der beteiligten Serverknoten überprüft werden. Überlastet eine gewählten Zuordnung einen Serverknoten, wird der entsprechende Teilbaum abgeschnitten und mit dem nächstteureren Knoten im Baum fortgefahren. Die gierige Suche entspricht einer informierten Tiefensuche und ist im Problembereich dieser Arbeit vollständig (wenn eine Lösung exisitert, wird sie gefunden), jedoch nicht optimal. Der Suchraum wird bereits nach dem ersten Schritt um $\frac{n-1}{n}$ beschnitten. Diese Menge kann jedoch optimale Lösungen enthalten. Die Reihenfolge der Anordnung von Prozessen und Diensten im Baum übt deshalb einen großen Einfluss auf die gefundene Lösungsmenge aus. Die Speicher- und Zeitkomplexität liegt zwischen $\mathcal{O}(2^n)$ im schlechtesten und $\mathcal{O}(n)$ im günstigsten Fall.

Greedy Search mit aktualisierenden Baumknotenstatistiken Im Gegensatz zur vorher beschriebenen gierigen Suche wird ein Serverknoten nicht von der weiteren Suche ausgeschlossen, sobald seine kumulierte Prozessorlast den definierten Schwellenwert überschreitet. Vielmehr werden bei Volllast eines Knotens die

Verarbeitungszeiten aller im Folgenden zuzuordnenden Dienste und Prozesse über die lineare Näherungsgleichung erhöht (vgl. Parallelität, Seite 130). Dadurch kann ein Serverknoten auch dann genutzt werden, wenn er trotz Volllast schnellere Verarbeitungszeiten von Diensten oder Prozessen ermöglicht als noch unausgelastete Knoten. Ein Nachteil dieser Suchvariante ist zum einen, dass sich die Parallelität auch auf bereits verteilte Prozesse auswirkt und somit die Entscheidungsgrundlage nachträglich modifizieren kann. Zum anderen bedingt Parallelität eine starke Varianz der Ausführungszeiten und damit wenig Stabilität. Zudem bleiben die Nachteile der gierigen Suche ohne Statistiken, d.h. die fehlende Garantie, eine optimale Lösung zu finden und die starke initiale Beschneidung des Suchraumes, bestehen.

Hill Climbing Die Grundidee des Bergsteigeralgorithmus (engl. *hill climbing*) besteht darin, eine Startkonfiguration durch geringfügige Veränderungen so lange iterativ anzupassen, bis ein Optimum gefunden wurde und der Algorithmus terminiert. Die Auswahl der Startkonfiguration geschieht dabei zufällig aus der Menge aller möglichen Dienst- und Prozessverteilungen und damit aus der Menge aller Blattknoten des bisher betrachteten Suchbaumes. Die notwendigen geringfügigen Veränderungen der Konfiguration bestehen in der veränderten Zuweisung einzelner Dienste oder Prozesse zu anderen Serverknoten. Da es sich bei einem Endergebnis um ein lokales Optimum handeln kann, werden nach der Identifikation eines Optimums weitere zufällige Systemkonfigurationen ausgewählt und der Suchalgorithmus erneut durchlaufen. Auch der Bergsteigeralgorithmus kann keine Optimalität garantieren, jedoch wird in jedem Fall eine lokal optimale Lösung gefunden.

A* Suche Auch die A*-Suche nutzt Heuristiken sowie eine Baumstruktur als Grundlagen der Suchstrategie. Bei einer monoton steigenden oder monoton fallenden Kostenfunktion der Heuristik pro Baumebene l wurden für diesen Suchalgorithmus dessen Vollständigkeit und Optimalität nachgewiesen [131]. Der A*-Algorithmus besitzt eine exponentielle Speicherkomplexität sowie, bei ungünstiger Heuristik, zudem eine exponentielle Zeitkomplexität [94, 124]. Abbildung 6.14 stellt die Funktionsweise dieses Algorithmus am Beispiel von drei Prozessen $p1$, $p2$, und $p3$ und zwei Serverknoten $n1$ und $n2$ analog zum Suchbaum in Abbildung 6.13 dar. Die A*-Suche startet in der Baumwurzel und expandiert zunächst die erste Ebene des Baumes vollständig. Die Baumknoten werden daraufhin nach den sich aus der Heuristikfunktion ergebenden Kosten sortiert in eine Warteschlange eingefügt. Abbildung 6.14 zeigt für jeden Baumknoten die entsprechenden Kosten der drei unterschiedlichen Heuristiken, aufgrund derer der Algorithmus die Knoten sortiert. Der erste Knoten der Warteschlange, welcher folglich auch die geringsten Kosten verursacht (zunächst Knoten 3 für alle drei Heuristiken), wird entnommen und wieder expandiert. Die neuen Knoten werden daraufhin der Warteschlange hinzugefügt und erneut sortiert. Der Algorithmus endet, wenn der erste Blattknoten aus der Warteschlange entnommen und damit eine Lösung gefunden wurde. Nach

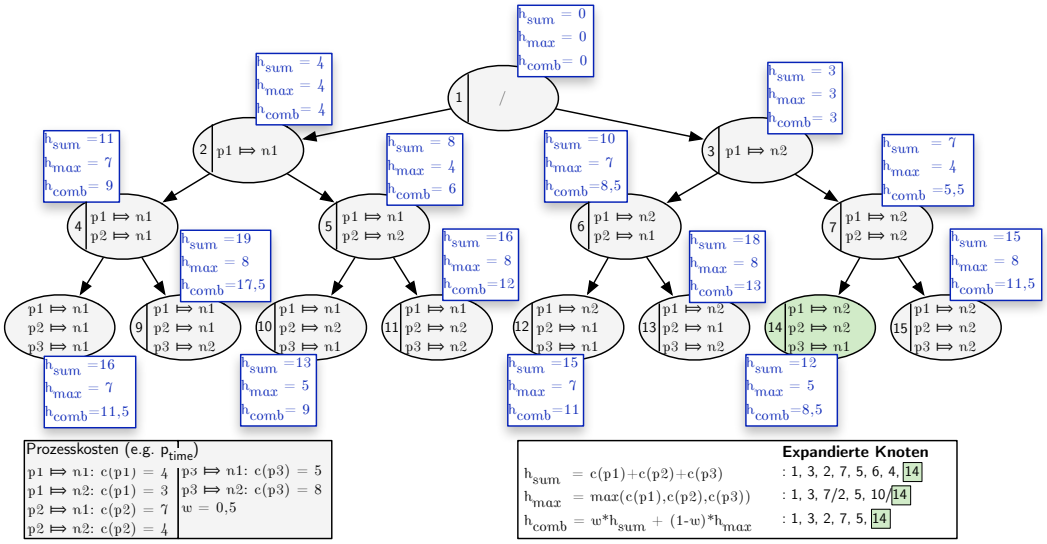


Abbildung 6.14: Beispiel A*-Suche im Suchbaum.

jedem Schritt wird die verursachte Prozessorlast auf den Serverknoten neu berechnet und gegebenenfalls Baumknoten mit überlasteten Serverknoten aus der Warteschlange entfernt. Das Verhalten des Algorithmus wird stark von der verwendeten Heuristik beeinflusst. Während die Kosten der Heuristik h_{max} ausschließlich die maximalen Kosten einer der bisher verteilten Komponente annimmt und der Algorithmus deshalb schnell eine Lösung findet und gegebenenfalls die optimale Lösung verfehlt (vgl. Knoten 10), verändert die Heuristik h_{sum} den Algorithmus zu einer Breitensuche, welche durch einen Großteil des Suchbaumes navigiert, jedoch schließlich auch die optimale Lösung (Knoten 14) findet.

Tabelle 6.7 fasst die diskutierten Suchalgorithmen zusammen und weist deren Nutzbarkeit im Rahmen des *Infrastruktur-Design-Advisors* aus. Obwohl dabei fast alle vorgestellten Algorithmen im schlechtesten Fall eine exponentielle Komplexität besitzen, existieren im Durchschnittsfall wesentliche unterschiede in der Laufzeit der einzelnen Algorithmen. Durch die Art des Aufbaus des Suchbaumes bzw. durch die Menge an Lösungen im Fall des Bergsteigeralgorithmus besitzen alle Algorithmen die Eigenschaft der Vollständigkeit und finden somit eine Lösung, wenn eine solche existiert. Optimalität kann hingegen einzig der A*-Algorithmus jedoch nur bei guter Heuristik gewährleisten. Der Bergsteigeralgorithmus findet dem gegenüber in jedem Fall ein lokales Optimum, das globale Optimum kann jedoch nur bei guter Startkonfiguration gewährleistet werden.

Algorithmus	Komplexität	Opt.-ziele	Komp.	h(x)	Vollst.	Opt.
Tiefensuche	$\mathcal{O}_S(\|N\| \cdot (\|S\| + \|P\|))$ $\mathcal{O}_T(\|N\|^{\|S\|+\|P\|})$	$c_{time} \mid c_{com}$	P+S	nein	ja	nein
Breitensuche	$\mathcal{O}_S(\|N\|^{\|S\|+\|P\|})$ $\mathcal{O}_T(\|N\|^{\|S\|+\|P\|})$	$c_{time} \mid c_{com}$	P+S	nein	ja	nein
Greedy Search	$\mathcal{O}_S(\|N\|^{\|S\|+\|P\|})$ $\mathcal{O}_T(\|N\|^{\|S\|+\|P\|})$	$c_{time} \mid c_{com}$	P+S	ja	ja	nein
Greedy Search mit Statistiken	$\mathcal{O}_S(\|N\| \cdot (\|S\| + \|P\|))$ $\mathcal{O}_T(\|N\| \cdot (\|S\| + \|P\|))$	c_{time}	P	ja	ja	nein
Hill Climbing	$\mathcal{O}_S(\|S\| + \|P\|)$ $\mathcal{O}_T(steps \cdot restarts \cdot (\|S\| + \|P\|)$	$c_{time} \mid c_{com}$	P+S	nein	ja	ja[1]
A*-Suche	$\mathcal{O}_S(\|N\|^{\|S\|+\|P\|})$ $\mathcal{O}_T(\|N\|^{\|S\|+\|P\|})$	$c_{time} \mid c_{com}$	P+S	ja	ja	ja[2]

[1] Abhängig von Qualität der Startkonfiguration.
[2] nur mit monoton steigender bzw. monoton fallender Heuristikfunktion.[131]

Tabelle 6.7: Worst-Case-Komplexität der Suchalgorithmen im Advisor.

6.4.3 Verteilungsalgorithmus

Auf Grundlage des vorherigen Abschnitts beschreibt dieser Abschnitt die spezifischen Suchalgorithmen für eine optimale Verteilung der Prozesse und Dienste auf die Serverknoten. Dafür werden sowohl ein Algorithmus für eine vollständige Suche vorgestellt als auch ein Algorithmus auf Basis der A*-Suche diskutiert. Ohne die allgemeine Gültigkeit einzuschränken, wird zur Illustration der Algorithmen angenommen, dass jeder Dienst zwar beliebig in der Infrastruktur verteilt werden kann, jedoch nur auf einem Serverknoten existiert. Somit beschränkt sich die Anzahl der Kombinationen gemäß Gleichung 6.18 auf $|N|^{|P|+|S|}$.

Vollständige Suche

Die vollständige Suche analysiert jede einzelne der $|N|^{|P|+|S|}$ Systemkonfigurationen anhand der Kostenfunktionen 6.3 - 6.15 (vgl. Abschnitt 6.3.4, Seite 135). Dabei wird am Ende jene Systemkonfiguration zurückgegeben, welche die Kosten entsprechend des Optimierungsziels minimiert.

Die vollständige Suche besitzt eine exponentielle Komplexität, da alle Kombinationen durchsucht werden. Tabelle 6.8 zeigt eine mögliche Repräsentation der einzelnen Systemkonfigurationen in einer Datenstruktur. Dazu werden zunächst alle möglichen Systemkonfigurationen k enumeriert. Die Werte im Hauptteil der Tabelle repräsentieren die Nummer Serverknotens, auf dem die Komponente ausgeführt wird. Die Berechnung dieser Zuordnungen erfolgt durch die Darstellung von k im Zahlensystem zur Basis der Anzahl an verfügbaren Serverknoten. Im Beispiel der Tabelle gilt für Systemkonfiguration mit 3 Knoten $k = 11_{10} = 00102_3$.

$P \cap S \backslash k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\cdots$
p_1	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	$\cdots$
p_2	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2	$\cdots$
p_3	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	$\cdots$
s_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\cdots$
s_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\cdots$

Tabelle 6.8: Zuordnungstabelle mit $|P| = 3$, $|S| = 2$ sowie $|N| = 3$ und damit $3^{3+2} = 243$ möglichen Systemkonfigurationen.

Die Vorgehensweise der vollständigen Suche auf dieser Zuordnungstabelle zeigt Algorithmus 4 (*CompleteSearch*). Dieser durchläuft zunächst alle k Systemkonfigurationen entsprechend ihrer Enumeration (Zeile 2, Tabelle 6.8 *horizontal*). Mit jedem Schritt wird über alle vorhandenen Prozesse und Dienste iteriert (Zeile 7, Tabelle 6.8 *vertikal*), der jeweilige Knotenindex berechnet (Zeile 9) und der entsprechende Knoten extrahiert (Zeilen 12 und 14). Die daraus resultierende Konfiguration wird auf ihre Gültigkeit überprüft mit der aktuell minimale Systemkonfiguration verglichen (Zeile 19). Ist diese Konfiguration gültig und der Wert der Kostenfunktion kleiner als die bisher minimale Konfiguration k_{min} wird die neue Konfiguration als

Algorithmus 4 CompleteSearch

Benötigt: Prozessmenge P, Knotenmenge N, Dienstmenge S, Kostenfunktion $c()$

```
 1: k_cur, k_min ← ∅
 2: for i = 1 to |N|^(|S|+|P|) do                    // ∀ Systemkonfigurationen
 3:    // Bestimme Zuordnung zu Knoten
 4:    d, m ← 0
 5:    set_P, set_S ← ∅
 6:    value ← i
 7:    for j = 1 to |P| + |S| do
 8:       d ← value div |N|
 9:       m ← value mod |N|
10:       value ← d
11:       if j < |P| then
12:          set_P ← setP ∪ (P[j], N[m])
13:       else
14:          set_S ← setS ∪ (S[j − |P|], N[m])
15:       end if
16:    end for
17:    k_cur ← (setP ∪ setS)
18:    // Bestimme Validität und Kosten von k und aktuelles Minimum
19:    if isValid(k_cur) & c(k_cur) < c(k_min) then    // überlastet k keinen Knoten?
20:       k_min ← k_cur
21:    end if
22: end for
23: return k_min
```

k_{min} gespeichert. Wurden alle Konfigurationen durchsucht, wird k_{min} abschließend zurückgegeben.

Unabhängig davon, welches Optimierungsziel der Algorithmus verfolgt, werden für jede Konfiguration die Verarbeitungszeit, das Datenvolumen der Kommunikation sowie die auftretende Prozessorlast geschätzt. Dies ermöglicht die Analyse der beiden Optimierungsziele *Verarbeitungszeit* und *Kommunikationsvolumen* während der Laufzeit des Algorithmus und damit eine priorisierte Ausgabe der Ergebnisse.

A*-Suche

Die A*-Suche nutzt die bereits in Abbildung 6.14 vorgestellte Datenstruktur des Suchbaumes und traversiert diesen gemäß Algorithmus 5 (*StarSearch*). Dabei startet der Algorithmus zunächst mit einer leeren Systemkonfiguration im Blattknoten des Baumes und fügt der bisherigen Systemkonfiguration pro Baumebene je einen weiteren Prozess oder Dienst hinzu. Um bei der Zuordnung der Dienste eine genauere Kostenschätzung bezüglich entfernter Kommunikation und tatsächlicher CPU-Last vorzunehmen, werden zunächst die Dienste den Knoten zugeordnet. Allerdings kann die Verarbeitungszeit der Dienste in Form von lokalen oder entfernten Verarbeitungskosten in diesem Schritt noch nicht endgültig bewertet werden.

Algorithmus 5 StarSearch

Benötigt: Suchbaum T, Kostenfunktion c

```
 1: validNodes, invalidNodes ← ⊘
 2: availableNodes ← T.getRootNode()
 3: while true do                                  // Solange kein Blattknoten gefunden
 4:   bestValidNode ← validNodes.getNextNode()
 5:   if bestValidNode == ⊘ then                   // Gibt es noch offene Knoten?
 6:     // Kein gültiges Ergebnis gefunden
 7:     return ⊘
 8:   end if
 9:   childNodes ← bestValidNode.getChildren()
10:   if childNodes == ⊘ then                     // Ist bestValidNode Blattknoten?
11:     return bestValidNode
12:   else
13:     // Gültige Kindknoten zur Menge der offenen Knoten validNodes hinzufügen
14:     for k = 1 to |childNodes| do                          // ∀ k in childNodes
15:       if isValid(k) then
16:         validNodes ← validNodes ∪ k
17:       end if
18:     end for
19:     inValidNodes ← invalidNodes ∪ bestValidNode
20:   end if
21: end while
```

Die Liste *availableNodes* enthält sowohl gültige als auch eventuell noch nicht expandierte Baumknoten, die absteigend nach der gegebenen Kostenfunktion bzw. Heuristik sortiert werden. Zunächst wird diese Liste mit dem Wurzelknoten des Baumes initialisiert (Zeile 2). Der Wurzelknoten enthält als Startkonfiguration eine leere Menge. Mithilfe einer Schleife wird jeweils der anhand der Kostenfunktion aktuell am besten bewertete Baumknoten aus der Liste extrahiert (Zeile 4). Wird kein Baumknoten zurückgegeben mehr, konnte keine gültige Systemkonfiguration gefunden werden und der Algorithmus terminiert. Andernfalls expandiert der Algorithmus den entnommenen Baumknoten (Zeile 9). Besitzt der Baumknoten keine Kindknoten, handelt es sich um einen Blattknoten und damit um eine vollständige und valide Systemkonfiguration, die der Algorithmus als Lösung zurückgibt und terminiert (Zeile 11). Verfügt der aktuelle Baumknoten jedoch über Kindknoten, werden diese auf ihre Gültigkeit geprüft, der Liste *availableNodes* hinzugefügt und der aktuelle Baumknoten von weiteren Betrachtungen ausgeschlossen (Zeile 19).

6.5 Evaluation

Dieser Abschnitt evaluiert die Kernkonzepte des in diesem Kapitel vorgestellten *Infrastruktur-Design-Advisor*. Dabei wird zunächst die Güte des in Abschnitt 6.3 abgeleiteten Kostenmodells in Bezug auf die Kostenschätzung der Verarbeitungszeit, der Datenkommunikation sowie der Prozessorlast untersucht. Im zweiten Teil werden die beiden vorgestellten Suchstrategien *A*-Suche* und *vollständige Suche* auf das im Rahmen dieses Kapitels definierte Problem der Prozess- und Dienstzuordnung verglichen und deren Güte aufgezeigt.

Die Testumgebung für dieses Evaluationskapitel besteht aus den bereits bekannten Serverknoten in Tabelle A.1 (vgl. Anhang Seite 195), die über ein LAN miteinander verbunden sind. Als Testprozesse werden zwei Prozesspläne p_1 und p_2 definiert. p_1 beinhaltet als Operatorgraphen die fünf Operatoren `Receive` (*o1*), `Split` (*o2*), `Invoke` (*o3*), `Invoke` (*o4*) und `Reply` (*o5*), wobei *o3* den entfernten Operator `Filter` und (*o4*) den entfernten Operator `Window` aufruft. p_2 besteht hingegen aus vier Operatoren `Receive` (*o1*), `Split` (*o2*), `Invoke` (*o3*) und `Reply` (*o4*). In diesem Prozess ruft (*o3*) ebenfalls den entfernten Operator `Filter` auf.

Güte des Kostenmodells

Zunächst wird die Berechnung der Prozesskosten evaluiert. Grundlage dafür bilden die Messungen und das Kostenmodell aus Abschnitt 6.3. Von der Genauigkeit der Kostenschätzung hängt sowohl die Schätzung der Prozesskosten als auch später die Güte der vorgeschlagenen optimalen Systemkonfiguration ab. Die Prozesskosten einzelner Prozesspläne ergeben sich aus der Eingaberate der Datenquelle, den am

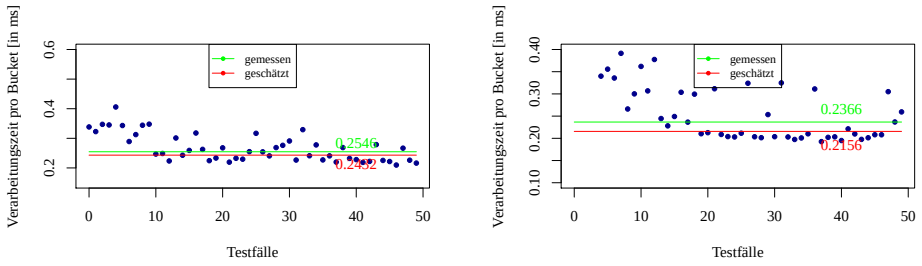
	Prozess p_1 (Knoten A)		Prozess p_2 (Knoten A)	
Metrik	**Gemessen**	**Geschätzt**	**Gemessen**	**Geschätzt**
$c_{time}(p)$ $(\frac{ms}{b})$	0.2546	0.2432	0.2366	0.2156
$c_{com}(p)$ $(\frac{Byte}{b})$	643.11	608.20	838.83	822.00
$c_{load}(p)$ (in %)	64.91	64.0	75.33	69.0
$r_{out}(p)$ $(\frac{b}{s})$	16.0	16.8	100.0	100.0

Tabelle 6.9: Vergleich der Ergebnisse der Prozesse p_1 und p_2.

Prozess beteiligten Operatoren, deren Selektivitäten und Bucketgrößen sowie der Hardwareleistung der Serverknoten.

Bewertung der Prozesskostenbestimmung Zunächst werden die Prozesse p_1 und p_2 und ihre Metriken Verarbeitungszeit (c_{time}), Datenvolumen (c_{com}), Prozessorlast (c_{load}) und Ausgabedatenrate (r_{out}) vom Advisor zur Ausführung auf Knoten A geschätzt und im Anschluss über tatsächliche Experimente überprüft. Alle `SInvoke`-Operatoren ($o3$ und $o4$ bei p_1; $o3$ bei p_2) rufen ihre Dienste auf Knoten B auf. Die Eingabedatenrate r_{src} für beide Prozesse beträgt jeweils 100 $\frac{b}{s}$. Die Eingabebucketgrößen werden für p_1 mit $g_{in} = 433\frac{Byte}{b}$ und für p_2 mit $g_{in} = 703\frac{Byte}{b}$ festgelegt.

Tabelle 6.9 vergleicht die erzielten Ergebnisse, die jeweils Durchschnittswerte pro Eingabebucket b darstellen. Dabei lässt sich erkennen, dass die geschätzten Werte, auf Basis derer die spätere Verteilung durch den *Infrastuktur-Design-Advisor* vorgenommen wird, die gemessenen Werte nicht übertreffen und damit die Voraussetzung einer validen Heuristik erfüllen. Dabei lässt sich beispielsweise der Schätzfehler des generierten Datenvolumens auf unterschiedliche Zeichenkettenlängen innerhalb der Anwendungsdaten zurückführen.



(a) Gemessene und geschätzte Verarbeitungszeit p_1 (Knoten A).

(b) Gemessene und geschätzte Verarbeitungszeit p_2 (Knoten A).

Abbildung 6.15: Verarbeitungszeiten p_1 und p_2 (Knoten A).

Bei genauerer Betrachtung der gemessenen und geschätzten Verarbeitungszeiten (c_{time}) der beiden Prozesse (vgl. Abbildung 6.15) zeigt sich, dass die gemessenen Verarbeitungszeiten auf der feingranularen Ebene $\leq 1.0ms$ starken Schwankungen unterliegen. Dadurch werden die Kosten für einzelne Prozessausführungen zwar in Einzelfällen überschätzt, jedoch liegt der geschätzte Durchschnittswert jeweils unter dem tatsächlichen Durchschnittswert.

Bewertung der Systemkostenbestimmung Nachdem gezeigt werden konnte, dass das Kostenmodell hinreichend genaue Aussagen zur Bestimmung einzelner Prozesskosten trifft, wird im Folgenden die korrekte Bewertung von Systemkonfigurationen untersucht. Als Grundlage gelten erneut die beiden Prozesse p_1 und p_2, die für eine umfangreichere Systemkonfiguration repliziert werden. Die Prozesse und interagierende Dienste wurden auf drei Serverknoten platziert, deren Metriken analog zur Prozesskostenbestimmung geschätzt und die Systemkonfiguration letztendlich ausgeführt. Dabei zeigte sich, dass die Berechnung der Datenvolumina ähnlich präzise wie die Prozesskostenberechnung ausfällt. Dies ist damit zu erklären, dass das kommunizierte Datenvolumen unabhängig von zusätzlichen, nebenläufigen Prozessen und Diensten in der Infrastruktur bleibt. Auch die Verarbeitungskosten der einzelnen Prozesse entsprachen den vorherigen Schätzungen. Einzig die von den in der Systemkonfiguration enthaltenen Prozessen und Diensten verursachten Prozessorlasten wurden zum Teil stark überschätzt. Eine solche Überschätzung ist zwar in Bezug auf die Optimalität des Algorithmus unkritisch, führt jedoch dazu, dass eventuell bessere Systemkonfigurationen frühzeitig verworfen werden. Insbesondere das Optimierungsziel einer minimalen Knotenanzahl kann damit nicht erreicht werden, da auf den verwendeten Knoten durchaus mehr Prozesse oder Dienste zugeordnet werden könnten.

Die Gründe für die Überschätzung der Prozessorlast sind vielfältig. Einerseits wird für die Operatoren beim Überwachen und Messen der notwendigen Metriken durch das Abgreifen von Informationen und die Speicherung dieser Informationen ein Mehraufwand generiert. Dieser Mehraufwand fließt mit in die Beobachtungen und damit in das Schätzmodell ein. Andererseits verkörpern die gemessenen Prozessorlasten nur Durchschnittswerte entsprechend der Abtastrate des Systemmonitors. Für eine präzisere Vorhersage der Prozessorlast wird deshalb entweder ein komplexeres Modell benötigt, welches verschiedene Einflussgrößen ausschließlich im Zusammenhang mit der Vorhersage der Prozessorlast berücksichtigt, oder, man verwendet Monitoring-Schnittstellen auf Betriebssystem- oder Hardwareebene für eine feingranulare Abschätzung.

Vergleich der Suchstrategien

Abschließend werden die im Rahmen dieser Arbeit betrachteten Suchalgorithmen evaluiert. Da die Evaluierung des Kostenmodells gezeigt hat, dass das Verhalten von

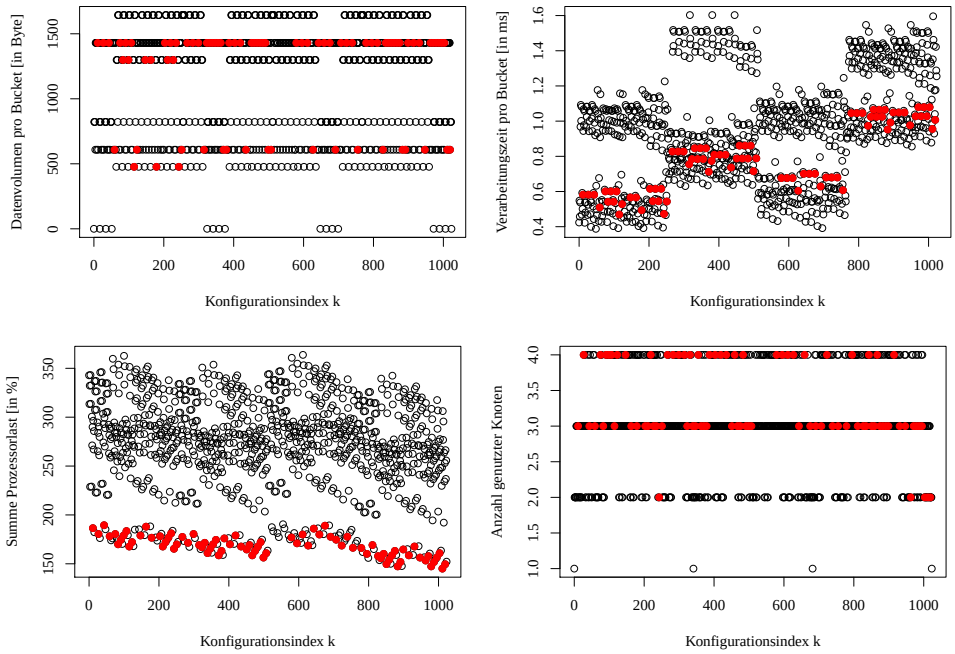


Abbildung 6.16: Systemkonfigurationen k der vollständigen Suche bei $|P| = 2$, $|S| = 3$, $|N| = 4$.

Prozessen und Systemkonfigurationen hinreichend genau vorhergesagt wird, sollte der Optimerungsalgorithmus auf Grundlage des Kostenmodells möglichst effizient und schnell eine optimale Lösung finden.

Die Abbildungen 6.16 und 6.17 visualisieren 1024 bzw. 4096 möglichen Systemkonfigurationen und tragen dagegen jeweils das generierte Datenvolumen (6.16(a) und 6.17(a)), die Verarbeitungszeit (6.16(b) und 6.17(b)), die verursachte Prozessorlast (6.16(c) und 6.17(c)) sowie die Anzahl der belegten Knoten (6.16(d) und 6.17(d)) ab. Die rot hervorgehobenen Konfigurationen repräsentieren gültige Konfigurationen, da sie die zugewiesenen Knoten nicht überlasten. Die Messungen hinsichtlich aller vier betrachteten abhängigen Variablen unterstreichen, dass die vollständige Suche aufgrund der hohen Komplexität des Suchraumes nur für kleine Problemräume geeignet erscheint.

Tabelle 6.10 stellt die Laufzeiteigenschaften der vollständigen Suche und der A*-Suche gegenüber. Dabei lässt sich erkennen, dass der A*-Algorithmus ähnlich der vollständigen Suche in allen Testfällen die optimale Lösung gefunden hat. Des Weiteren nimmt das Verhältnis von Anzahl der Kombinationen im Suchraum zur reduzierten Anzahl der Optimierungsschritte beim A*-Algorithmus stetig zu. Somit eignet sich dieser Algorithmus besser für die Lösung des Verteilungsproblems bei großen Suchräumen.

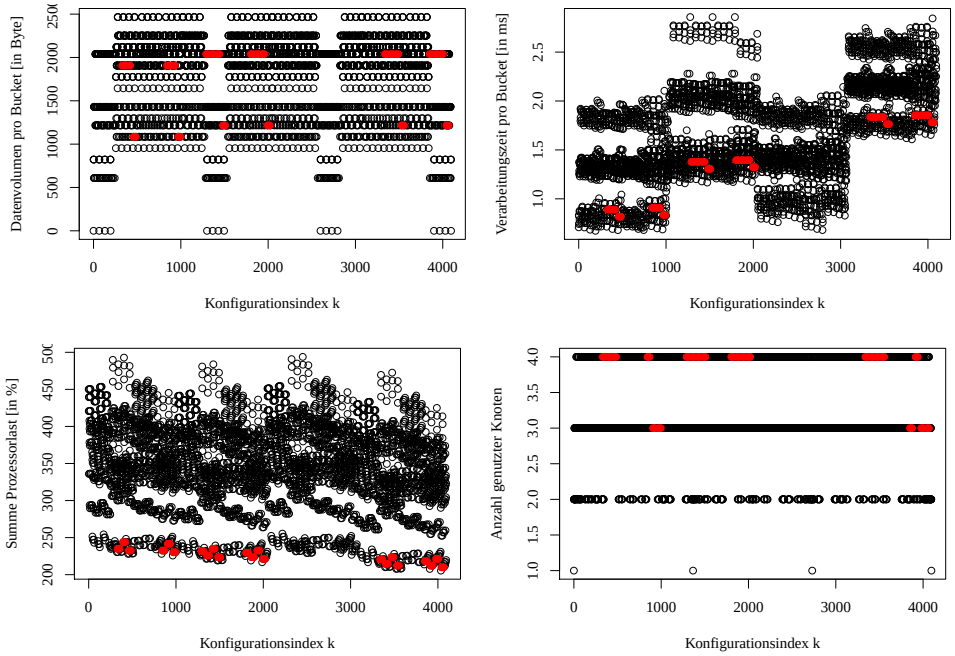


Abbildung 6.17: Systemkonfigurationen k der vollständigen Suche bei $|P| = 3$, $|S| = 3$, $|N| = 4$.

Such-algorithmus	Such-raum	Besuchte Knoten	Ungültig	Opt.-schritte	opt. Lsg?	Dauer
Vollständig	64	64	60	64	ja	0:00.130
A*-Suche	64	52	0	14	ja	0:00.077
Vollständig	1.024	1.024	943	1.024	ja	0:01.473
A*-Suche	1.024	1.364	0	342	ja	0:01.960
Vollständig	4.096	4.096	4.052	4.096	ja	0:04.424
A*-Suche	4.096	2.328	0	583	ja	0:03.118
Vollständig	16.384	16.384	16.354	16.384	nicht	0:16.042
A*-Suche	16.384	4.904	0	1.227	existent	0:05.755

Tabelle 6.10: Laufzeiteigenschaften der Suchstrategien.

6.6 Zusammenfassung

Dieses Kapitel stellte den Ansatz des *Infrastruktur-Design-Advisors* vor, der durch eine intelligente Platzierung von Prozessen und Diensten auf Serverknoten der Infrastruktur die Effizienz und damit die Skalierbarkeit der gesamten SOA-basierten Infrastruktur erhöht. Dieser Ansatz verhält sich damit sowohl zu dem Konzept der skalierbaren Dienstkommunikation (Kapitel 4) als auch zu dem Konzept der skalier-

baren Prozessausführung (Kapitel 5) orthogonal, indem er zwar auf den Eigenschaften der Dienste und Prozesse dieser Konzepte aufbaut, jedoch dabei die gesamte Infrastruktur betrachtet.

Im Speziellen nimmt der *Infrastruktur-Design-Advisor* eine Menge an Diensten, Prozessen und heterogenen Serverknoten und schlägt gemäß eines Optimierungszieles eine optimale Systemkonfiguration, d.h. eine optimale Verteilung der Dienste und Prozesse auf die Serverknoten, vor. So reduziert beispielsweise eine Systemkonfiguration mit dem Ziel des minimalen Datenvolumens die knotenübergreifende Kommunikation durch eine gemeinsame Platzierung der Dienste und Prozesse auf denselben Serverknoten.

Zunächst befasste sich das Kapitel mit bereits verwandten Arbeiten, die sich Verteilungsstrategien und möglichen Kostenmodellen widmen. Darauf aufbauend wurde ein Systemmodell definiert, welches von der im Rahmen dieser Arbeit betrachteten Systemwelt abstrahiert und diese für den Advisor aufbereitet. Des Weiteren wurden mögliche Optimierungsziele hinsichtlich dieses Systemmodells beschrieben und eine Nebenbedingung dieser Ziele aufgestellt, welche besagt, dass die Verteilung der Prozesse und Dienste keinen Serverknoten überlasten darf.

Ein Kernstück des *Infrastruktur-Design-Advisors* bildet das Kostenmodell, welches einerseits die Kosten der möglichen Operatortypen eines Prozesses beschreibt und diese Kosten andererseits zu Prozess- und Systemkosten aggregiert. Dazu wurden *acht* mögliche Einflussgrößen auf die Verarbeitungszeit, wie beispielsweise Eingabedatenrate, Kommunikationskosten oder auch die Selektivität auf die Verarbeitungszeit der Operatortypen, experimentell gemessen und Kostenfunktionen abgeleitet.

Die Analyse des Problems einer optimalen Verteilung von Diensten und Prozessen ergab einen exponentiellen Suchraum, der jedoch bei vorhandenen Verteilungsbeschränkungen der Dienste und Prozesse stark eingeschränkt werden kann. Zudem wurden weitere Heuristiken erläutert, die eine zusätzliche Verkleinerung des Suchraums ermöglichen, und drei Heuristikfunktionen h_{sum}, h_{max} und h_{comb} aufgestellt.

Auf der Suche nach der optimalen Systemkonfiguration wurden existierende Suchalgorithmen analysiert und deren Eigenschaften im Kontext des Verteilungsproblems gegenübergestellt. Aufgrund dessen wurde schließlich der A*-Suchalgorithmus gewählt, da er die Eigenschaften der Vollständigkeit und, bei monoton steigender oder fallender Heuristik, die Eigenschaft der Optimalität erfüllt.

Der Vergleich der Suchstrategien *Vollständige Suche* und *A*-Suche* im Zuge der Evaluation ergab, dass der A*-Algorithmus mit der Heuristik h_{sum} jeweils die optimale Systemkonfiguration findet, ohne dabei den gesamten Suchraum der *vollständigen Suche* zu durchlaufen. Bei der Analyse der Güte des Kostenmodells zeigte sich, dass die Kosten für einen Prozess durch den *Infrastruktur-Design-Advisor* mit nur geringer Abweichung geschätzt werden. Für die optimale Systemkonfiguration ergeben sich jedoch starke Abweichungen bei der Schätzung der verursachten Prozessorlast,

da die Prozessorlast bei einer geringen Eingangsdatenrate nicht mehr der zuvor berechneten linearen Näherungsgleichung entspricht und die Prozessorlast überschätzt wird. Dadurch werden bei der Optimierung auch Systemkonfigurationen verworfen, die bei einer tatsächlichen Ausführung die Serverknoten nicht überlasten würden.

7 Prozessframework SOA-XS

Dieses Kapitel gibt einen Überblick über den den im Rahmen dieser Arbeit entwickelten Prototypen *SOA-XS* (*SOA for XML Streaming*), welcher die in dieser Arbeit vorgestellten strombasierten Verarbeitungskonzepte auf Dienst- und Prozessebene integriert. Des Weiteren wird die Komponente *SOA-XS-DA* (*SOA-XS-Design Advisor*) kurz dargestellt, welche auf SOA-XS aufbaut und die Konzepte der Infrastrukturebene implementiert. Die Implementierunggrundlage bildet die Laufzeitumgebung *Java 1.6*.

7.1 Ausführungsframework SOA-XS

Das Ausführungsframework SOA-XS beinhaltet die grundlegenden Funktionen und Komponenten zur effizienten und skalierbaren Ausführung dienstbasierter Geschäftsprozesse, datenintensiver Datenintegrations- und -analyseprozesse sowie Prozesse zur Nachrichtenstromanalyse. Abbildung 7.1 zeigt die Gesamtarchitektur des Frameworks. Diese besteht aus dem *Frontend*, welches die Komponenten für die Nutzerinteraktionen bereitstellt, und dem *Backend*, welches die Komponenten der Ausführungsumgebung beinhaltet.

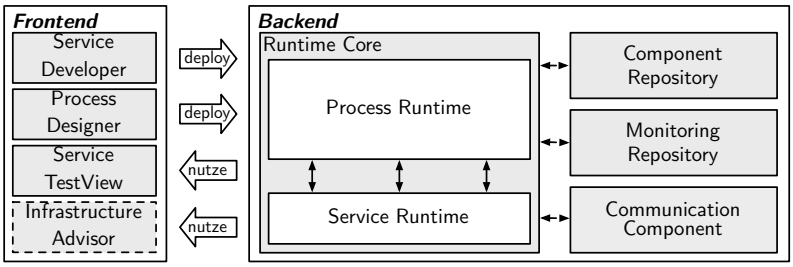


Abbildung 7.1: SOA-XS Architektur.

Frontend

Das Frontend realisiert die Interaktion mit dem Prozess- oder Dienstentwickler und fasst die Komponenten *AtomicServiceDevelopment*, *ProcessDesigner*, *ServiceTestView* sowie *Infrastructure Advisor* zusammen. Alle vier Komponenten basieren auf der Eclipse-Java-IDE[1] und stellen ihre Funktionalitäten im Rahmen dieses Rahmen-

[1] http://eclipse.org/

werkes zur Verfügung. Im Folgenden werden die ersten drei Komponente detaillierter beschrieben. Die Komponente *Infrastructure Advisor* wird hingegen in Abschnitt 7.2 separat beshrieben.

Die Komponente *AtomicServiceDevelopment* bildet die Grundlage für die Entwicklung atomarer strombasierter Dienste und unterstützt dies durch spezielle Objektklassen und Annotationen. Listing 7.1 zeigt einen Ausschnitt der Javaklasse des atomaren Dienstes *Rechnungen* (vgl. Prozessaktivität *a2* des DIA-Prozesses *Top n Kunden*, Seite 25), deren Methode `getInvoiceStream()` als strombasierter Dienst publiziert wurde.

```
1  @StreambasedService(dataInSchema = "in1.xsd", dataOutSchema = "out1.xsd",
2                      paramMode    = ParamMode.INIT, paramSchema  = "par.xsd"
3                      multiplicity = Multiplicity.ONE2MANY)
4  public void getInvoicesStream(SOAPInQueue sc, SOAPOutQueue sp, ParBucket par){
5    OMStreamBucket element;
6    while (sc.hasNext()) {
7      if(par.hasChanged()){
8        ...
9      }
10     element = sc.consumeElement();
11     ...
12     sp.put(element);
13   }
14   sp.closeStream();
15 }
```

Listing 7.1: Methodenkonstrukt eines atomaren strombasierten Dienstes.

Mithilfe der Annotation *@StreambasedService* (Zeile 1) definiert der Dienstentwickler die gültigen Eigenschaften des strombasierten Dienstes, welche später in das WSDL-Dokument integriert werden und damit von jedem potentiellen Dienstnutzer ausgelesen werden können. Auf diese Weise lassen sich die Schemainformationen der Ein- und Ausgabedaten als XML Schema-Dokument referenzieren (Zeile 1, `dataInSchema`, `dataOutSchema`), die Parametrierungsart festlegen (Zeile 2, `paramMode`, `paramSchema`, vgl. Abschnitt 4.3.2, Seite 64) sowie die Klasse der Ein- und Ausgabebeziehungen anzeigen (Zeile 3, `multiplicity`, vgl. Abschnitt 4.4.2).

```
1  <wsdl:import namespace="..." location="../schema/in1.xsd"/>
2  <wsdl:import namespace="..." location="../schema/out1.xsd"/>
3  <wsdl:import namespace="..." location="../schema/par.xsd"/>
4   ...
5  <wsdl:operation name="getInvoiceStream">
6    <sws:streamws streamable="true" multiplicity="ONE2MANY" paramMode="INIT"/>
7    <wsdl:input  message="..." wsaw:Action="urn:getInvoice"/>
8    <wsdl:output message="..." wsaw:Action="urn:getInvoiceResponse"/>
9  </wsdl:operation>
```

Listing 7.2: Ausschnitt des WSDL-Dokumentes für Listing 7.1.

Listing 7.2 zeigt die um die Eigenschaften des strombasierten Dienstes modifizierten Einträge des WSDL-Dokumentes. Dabei wurden zunächst die XML Schema-Dokumente der Eingabe-, Ausgabe- und Parameterdefinitionen direkt in das WSDL-

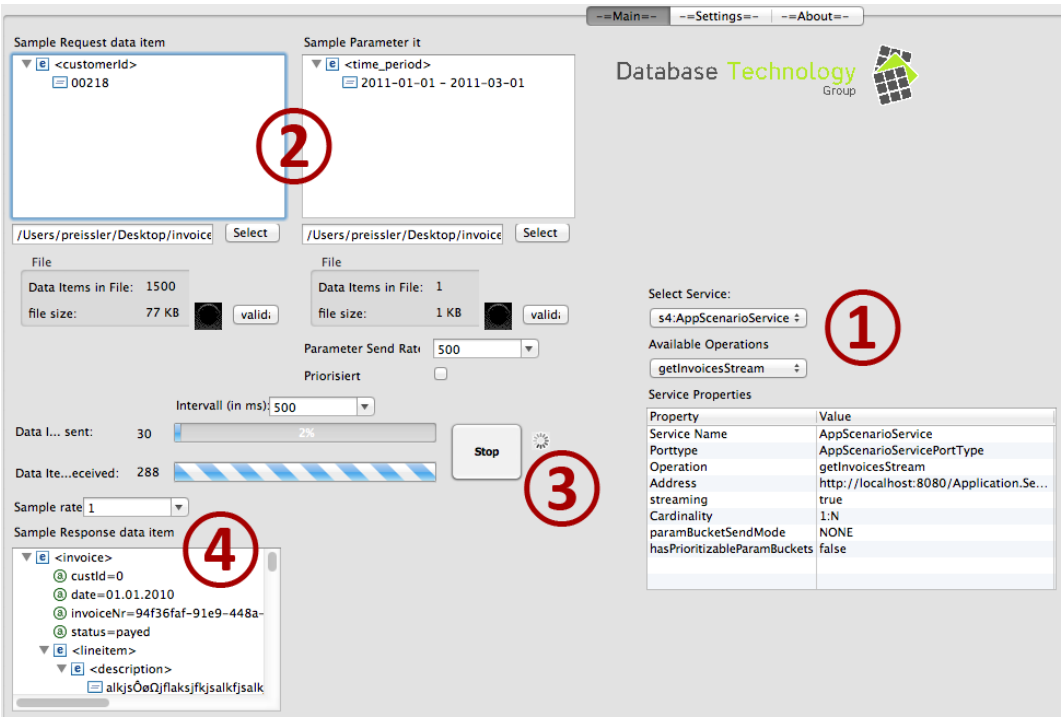


Abbildung 7.2: SOA-XS Service Test View.

Dokument importiert (Zeilen 1-3). Das Element `operation` (Zeilen 5-9) repräsentiert die Operation `getInvoiceStream` und umschließt deren abstrakte Beschreibung. Es beinhaltet das Element `streamws` mit den in Listing 7.1 definierten Eigenschaften des strombasierten Dienstes (Zeile 6) sowie die klassischen Elemente `input` und `output` (Zeilen 7-8), welche über das Attribut `messages` die importierten XML Schemata referenzieren.

Der Zugriff auf die Daten innerhalb der Java-Methode in Listing 7.1 erfolgt mittels einer Schleife über alle in der Eingabewarteschlange enthaltenen Buckets (Zeile 6). Dabei blockiert die Methode `sc.hasNext()` solange keine Buckets in der Warteschlange liegen und der Eingabestrom noch nicht geschlossen wurde. Der Methodenparameter `par` enthält das jeweils aktuelle Parameterbucket zur Parametrierung der Dienstinstanz. Die Abfrage `par.hasChanged()` (Zeile 7) gibt an, ob das Parameterbucket seit dem vorherigen Schleifendurchlauf inhaltlich geändert wurde. Dies gilt nur für den Parametermodus `INTERMEDIATE` (vgl. Abschnitt 4.3.2), bei dem eine Dienstinstanz während ihrer Laufzeit neue Parameter zur Rekonfiguration empfangen kann. Nach dieser Abfrage und der eventuellen Rekonfiguration kann die Dienstinstanz Verarbeitungsbuckets aus der Eingabewarteschlange `SOAPInQueue` entnehmen (Zeile 10) und gegebenenfalls in den Rückgabestrom zurückgeben (Zeile 14). Wurden alle Buckets verarbeitet, wird der Rückgabestrom vom Dienst zum Dienstnutzer geschlossen (Zeile 14).

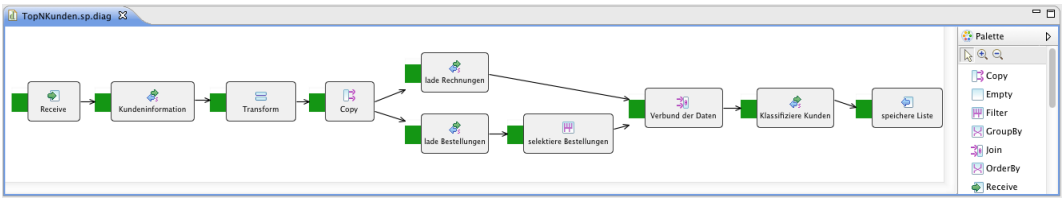


Abbildung 7.3: SOA-XS Process Designer.

Nach der Entwicklung des atomaren Dienstes und dessen Deployment auf einem Server, ermöglicht es die Komponente *ServiceTestView* (vgl. Abbildung 7.2), diesen Dienst über eine grafische Oberfläche aufzurufen. Dabei ist es im ersten Schritt möglich, die Eigenschaften der in der Anwendung registrierten Dienste und deren jeweilige Operationen aufzulisten (1). In einem zweiten Schritt können Testdaten und Parameterdaten geladen und gegen die Schemadefinitionen des Dienstes validiert werden (2). Im dritten Schritt wird der Dienst aufgerufen und die Menge der aktuell gesendeten und empfangenen Elemente während der Übertragung angezeigt (3). Dabei kann die Geschwindigkeit während des Sendens der Elemente modifiziert werden. Der Bildschirmbereich (4) stellt die bereits empfangenen Antwortdaten dar.

Wurde der Dienst entwickelt und getestet, kann er in die Ausführung strombasierter Prozesse integriert werden. Für die Modellierung strombasierter Prozesse wurde dazu die Komponente *Process Designer* prototypisch realisiert, die auf dem Eclipse-GMF-Framework[2] aufsetzt. Abbildung 7.3 zeigt beispielhaft den Beispielprozesse *Top n Kunden* aus Kapitel 2. Neben der eigentlichen Modellierung strombasierter Prozesse ermöglicht der *Process Designer* den direkten Test des Prozesses mithilfe von Testdaten innerhalb der Modellierungsansicht. Im abschließenden Schritt werden der Prozess für die Ausführungsumgebung kompiliert und dessen Dienstbeschreibung und Dienstkonfiguration erstellt. Die dadurch generierten Artefakte werden anschließend ins Backend importiert und für die eigentliche Nutzung aufbereitet.

Backend

Das Backend fasst die Komponenten *Runtime Core*, *Component Repository*, *Monitoring Repository* sowie *Communication Component* zusammen und realisiert die eigentliche Ausführung der Dienst- und Prozessinstanzen.

Die Komponente *Runtime Core* beinhaltet die Kernkomponenten *Process Runtime* und *Service Runtime* und realisiert die in dieser Arbeit notwendigen Eigenschaften an eine effiziente, strombasierte Prozessausführung auf Basis von XML-basierter Nachrichtenkommunikation. Die Komponente *Process Runtime* bildet dabei den zentralen Punkt der Prozessausführung und bietet alle notwendigen Klassen an, um

[2] www.eclipse.org/gmf/

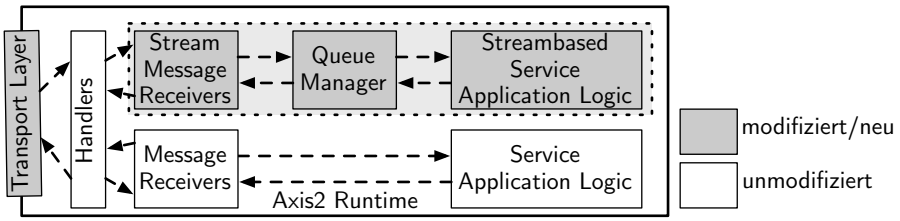


Abbildung 7.4: SOA-XS Communication Component.

die im *Process Designer* modellierten oder manuell programmierten Prozesse zu instanziieren, auszuführen und zu terminieren. Dabei interagiert die *Process Runtime* sowohl mit der *Service Runtime*-Komponente als auch mit den Komponenten *Global Operator and Process Repository* und *Monitoring Repository*. Die *Service Runtime*-Komponente realisiert die Ausführung traditioneller und strombasierter *atomarer Dienste*, die mit dem *Atomic Service Development* entwickelt wurden. Die beiden Komponenten der *Process Runtime* und *Service Runtime* sind eng miteinander gekoppelt, da der Zugriff von Prozessen auf lokale Dienste direkt in der Ausführungsumgebung vollzogen wird und keine externe Nachrichtenkommunikation stattfindet.

Die Komponenten *Process Runtime* und *Service Runtime* nutzen zudem die Komponente *Communication Component* zur Kommunikation mit externen Diensten und Prozessen, welche auf dem Rahmenwerk Axis2[3] aufsetzt. Abbildung 7.4 stellt die Architektur und die darin vorgenommenen Modifikationen dar. Grundsätzlich besteht die Architektur aus der Transportebene (*Transport Layer*), welche die eigentliche Kommunikation über TCP/IP übernimmt, und den *Handlern*, welche eine Nachricht direkt nach dem Empfang, jedoch vor der Anwendungslogik verarbeiten. Ein Beispiel für die Aufgaben eines solchen *Handler*s bildet die Entschlüsselung von chiffrierten Nachrichten vor der eigentlichen Verarbeitung durch die Anwendungslogik.

Die *Message Receiver*-Komponente in der Axis2-Architektur bewerkstelligt die Interaktion der Nachricht mit der Anwendungslogik. Sie extrahiert die Nutzdaten, ruft damit die Anwendungslogik auf und generiert gegebenenfalls eine Antwortnachricht. Innerhalb dieser Unterkomponente des *Message Receiver*s wurden, zusätzlich zum *Transport Layer*, Modifikationen durchgeführt, um eine strombasierte Verarbeitung der Nutzdaten zu realisieren. So wurde neben einer Warteschlangenverwaltung (*Queue Manager*) auch das Rahmenwerk der Anwendungslogik (*Streambased Service Application Logic*) gemäß Listing 7.1 angepasst.

Das *Component Repository* speichert die auf dem lokalen und auf den entfernten Knoten als Operatoren verfügbaren Prozesse und Dienste. Handelt es sich bei einem aufgerufenen Operator in einem lokalen Prozessgraphen um lokal verfügbare Prozesse oder Dienste, so interagiert der Prozessgraph innerhalb der lokalen Ausfürungsum-

[3] http://axis.apache.org/axis2/java/core/

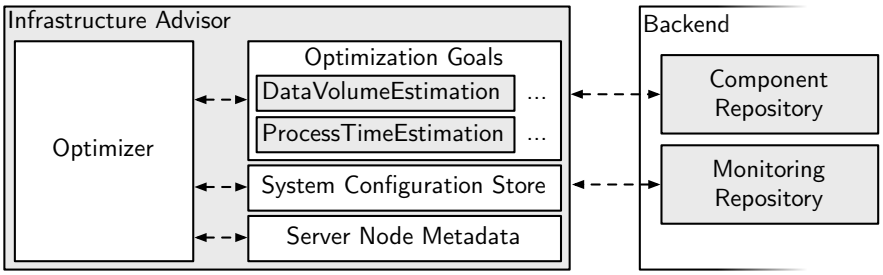


Abbildung 7.5: SOA-XS Infrastucture Advisor Architektur.

gebung direkt und effizient mit diesen. Ist hingegen der referenzierte Operator oder Dienst nur auf entfernten Knoten verfügbar, realisiert die *Communication Component* die traditionelle oder strombasierte Kommunikation. Abschließend speichert das *Monitoring-Repository* die Ausführungsstatistiken der Operatoren und Dienste und stellt die Werte dem Infrastruktur-Design-Advisor zur Verfügung.

7.2 Infrastruktur-Design-Advisor SOA-XS-DA

Der *SOA-XS Design Advisor* (DA) bildet einen Teil des SOA-XS Frontends und kommuniziert mit den Komponenten *Monitoring Repository* und *Component Repository* des SOA-XS Backends, um alle notwendigen Prozesse, Dienste und deren Laufzeitstatistiken abzufragen. Den Eingabeparameter des Advisors bildet die jeweilige Arbeitslast in Form der separaten Eingabedatenraten der einzelnen Prozesse. Die verfügbaren Serverknoten sowie mögliche Verteilungsbeschränkungen von Diensten und Prozessen auf diesen Knoten sind in der Komponente *Server Node Metadata* hinterlegt. Die Komponente *Optimization Goals* beinhaltet die Algorithmen und Komponenten zum Erreichen der einzelnen in Abschnitt 6.2.3 definierten Optimierungsziele (vgl. Algorithmen 1 und 2 auf den Seiten 132 und 133) und lässt sich um zusätzliche Optimierungsziele erweitern. Die einzelnen Optimierungszielkomponenten fordern dazu über den Optimierer alle für die spezifische Optimierung notwendigen Daten vom *Monitoring Repository* an. Die berechneten Systemkonfigurationen werden während der Optimierung in der Komponente *System Configuration Store* abgelegt. Dabei werden die Dienst- und Prozesszuordnungen auf die beteiligten Knoten, die geschätzten Verarbeitungszeiten, die geschätzte generierte Prozessorlast pro Knoten sowie das anfallende Datenvolumen gespeichert. Dies ermöglicht die Ausgabe von detaillierten Informationen zu jeder Systemkonfiguration sowie die Sortierung der Konfigurationen nach bestimmten Eigenschaften.

7.3 Zusammenfassung

Dieses Kapitel bot einen Überblick über das im Rahmen dieser Arbeit entwickelte prototypische Rahmenwerk *SOA for XML Streaming* (SOA-XS). Dabei wurden die einzelnen Kernkomponenten des Rahmenwerkes vorgestellt und an ausgewählten Beispielen der in Kapitel 2 formulierten Anwendungsprozesse und ihren jeweilgen Dienste beschrieben. Teile dieses Prototyps wurden im Rahmen des Projektes THESEUS.TEXO[4] auf dem Demonstrator *SPACEflight*[5] verwertet und befinden sich zum Zeitpunkt dieser Arbeit im THESEUS Innovationszentrum (TIZ)[6].

[4] http://www.theseus-programm.de/de/texo.php
[5] http://texo.inf.tu-dresden.de/wiki/Main_Page
[6] http://theseus-programm.de/de/tiz.php

8 Zusammenfassung und Ausblick

Moderne IT-basierte Unternehmensinfrastrukturen haben sich in den letzten Jahren von monolithischen Systemanwendungen hin zu verteilten, komponentenbasierten Systemen entwickelt. Darin stellen die geschäftsprozessorientierten Abläufe des Unternehmens auf Basis von orchestrierten, verteilten Dienstkomponenten die zentrale Anwendungsklasse der Geschäftsprozesse (*BPM*) dar. Sie bilden zusammen mit der Anwendungsklasse der Datenintegration und Datenanalyse (*DIA*) sowie mit der Anwendungsklasse des Geschäftsprozessmonitorings (*BAM*) die Voraussetzung, um sowohl kurzfristig Probleme des Geschäftsablaufes zu identifizieren als auch um mittel- und langfristige Veränderungen im Markt zu erkennen und die Geschäftsprozesse des Unternehmens flexibel darauf anzupassen.

Aufgrund der geschichtlich bedingten, voneinander unabhängigen Entwicklung der Systeme der Anwendungsklassen *BPM*, *DIA* und *BAM*, werden die jeweiligen Anwendungsprozesse gegenwärtig in eigenständigen Systemen modelliert und ausgeführt. Daraus resultiert jedoch eine Reihe an Nachteilen, welche diese Arbeit aufzeigt und ausführlich diskutiert. Vor diesem Hintergrund beschäftigte sich die vorliegende Arbeit mit der Ableitung einer konsolidierten Ausführungsplattform, die es ermöglicht, Prozesse aller drei Anwendungsklassen gemeinsam zu modellieren und in einer SOA-basierten Infrastruktur effizient auszuführen.

Dazu wurden zunächst jede der drei Anwendungsklassen hinsichtlich ihrer Datencharakteristika untersucht und existierende Ausführungskonzepte sowie verwandte Arbeiten im Bereich der Anwendungsklassen analysiert. Die durchgeführte Analyse ergab, dass eine strombasierte und partitionierte Verarbeitung der Daten für eine konsolidierte Ausführungsumgebung, in welcher neben den Anforderungen der Anwendungsklasse *BPM* auch die Anforderungen der Anwendungsklassen *DIA* und *BAM* erfüllt werden sollen, unabdingbar ist.

Um eine effiziente und skalierbare Ausführung derartiger Prozessanwendungen zu erreichen, betrachtet die vorliegende Arbeit drei Ebenen einer SOA-Infrastruktur und legt dafür ein Gesamtkonzept vor.

Auf der Ebene der Dienstausführung wurde ein strombasiertes Aufrufmodell entwickelt, welches die bisherige nachrichtenbasierte Kommunikation modifiziert und einen skalierbaren, strombasierten Transfer beliebig großer Anwendungsdaten ermöglicht. Zudem wurde die Verarbeitungssemantik innerhalb der Dienstinstanzen angepasst, wodurch einerseits die transferierten Datenmengen entsprechend skalierbar verarbeitet werden können und andererseits der Empfang der Daten sowie deren

Rückgabe nebenläufig stattfinden. Die experimentelle Bewertung dieses Ansatzes zeigte dessen Skalierbarkeit und Effizienz gegenüber bisherigen Aufrufmodellen bei gleichzeitiger Erfüllung der auf Ebene der Dienstausführung erforderlichen Anforderungen des Gesamtkonzeptes.

Auf der zweiten Ebene, der Ebene der SOA-basierten Prozessausführung, wurde ebenfalls ein skalierbares Verarbeitungsmodell vorgestellt. Die Grundlage dieses Verarbeitungsmodells bildet, analog zur Ebene der Dienstausführung, die strombasierte, nebenläufige Verarbeitung der Prozessdaten. Zudem ermöglicht es das entwickelte Prozessmodell, die von einem Prozess zu verarbeitenden Daten für die Verarbeitung in den einzelnen Aktivitäten zu partitionieren, zu aggregieren und zu modifizieren. Durch die starke Integration der Konzepte der Ebene der Dienstausführung wird darüber hinaus eine effiziente und skalierbare Nutzung verteilter Dienste und Funktionalitäten realisierbar. Dies ermöglicht die Erfüllung aller der drei Anwendungsklassen gestellten Anforderungen. In den zur Evaluierung des entwickelten Ausführungskonzeptes durchgeführten Experimenten konnte nachgewiesen werden, dass die entwickelte Ausführungsumgebung für beliebig große Datenmengen skaliert, solange sich diese Datenmengen partitioniert verarbeiten lassen. Zudem ermöglicht die konzipierte, strombasierte Prozessverarbeitung die Ausnutzung von Multicore-Architekturen, indem sie in Abhängigkeit der Anzahl der in einem Prozess definierten Operatoren mit den zur Verfügung stehenden CPU-Kernen skaliert.

Schließlich wurde auf der dritten Ebene, der Ebene der Infrastruktur, ein Advisor vorgestellt, der eine effiziente Verteilung der Prozess- und Dienstkomponenten auf verfügbare Serverknoten vorschlägt. Diese Verteilung basiert auf vorgegebenen Optimierungszielen. Im Rahmen dieser Arbeit wurden die beiden Optimierungsziele der minimalen Verarbeitungszeit der Prozesse sowie des minimalen Datenverkehrs zwischen einzelnen Serverknoten betrachtet. Die Grundlage des entwickelten Advisors bildet das Kostenmodell, welches acht Einflussgrößen auf die Verarbeitungszeit der Operatoren berücksichtigt, um präzise Kostenschätzungen vorzunehmen. Die Verteilung der Prozess- und Dienstkomponenten erfolgt gemäß der Optimierungsziele mithilfe des A*-Algorithmus. Die Evaluierung des Advisor-Konzeptes wies einerseits nach, dass einerseits die Kostenschätzungen für die Verarbeitungszeit und das Datenvolumen hinreichend genau arbeiten und andererseits die eingesetzten Heuristikfunktionen für eine effiziente Suche nach der optimalen Systemkonfiguration genutzt werden können.

Ausblick

Zwar gibt es weiterhin spezielle Anwendungen der beiden Anwendungsklassen *DIA* und *BAM*, die aus Gründen der Effizienz nicht mit SOA-basierten Konzepten realisiert werden können, jedoch bietet eine einheitliche Modellierungs- und Ausführungsumgebung zahlreiche, in dieser Arbeit ausführlich diskutierte Vorteile gegenüber voneinander losgelösten Systemen. Die in dieser Arbeit vorgestellten Konzepte

stellen einen ersten Schritt hin zu einer konsolidierten Ausführungsumgebung für alle drei Anwendungsklassen dar. Aus diesem Grund existieren auf allen drei Ebenen weiterhin offene Forschungsfragen, deren Lösung zukünftigen Arbeiten vorbehalten bleibt.

So stellt sich auf Ebene der Dienstausführung die Frage nach der möglichen Erweiterung zusätzlicher Buckettypen im Anfrage- und Antwortstrom. Diese Erweiterung könnte einerseits dazu genutzt werden, um nicht-funktionale Eigenschaften wie die Ausfallsicherheit zu gewährleisten oder die priorisierte Verarbeitung von speziellen Datenelementen zu ermöglichen. Andererseits lassen sich damit Systeminformationen über den Antwortstrom zurückzugeben, welche beispielsweise die aktuelle Auslastung des Serverknotens beinhalten und dadurch weitere Optimierungen auf Prozessseite, wie beispielsweise die Lastbalancierung der Dienstaufrufe, realisieren.

Auf Ebene der SOA-basierten Prozessausführung sollte untersucht werden, inwieweit klassische Optimierungsansätze für Datenflussgraphen von Datenbankmanagementsystemen (DBMS) oder Datenstrommanagementsystemen (DSMS) sowie für kontrollflussbasierte Integrationsprozesse [26] auf die von einem Nutzer modellierten Prozessgraphen angewandt werden können, um ihre Effizienz und Skalierbarkeit zu steigern. Weiterhin sollte untersucht werden, ob die explizite Modellierung von Invoke-Operatoren durch die direkte Modellierung der Dienstfunktionalität als lokaler Operator ersetzt werden kann. Dabei sollte die Ausführungsumgebung zur Ausführungszeit entscheiden, ob der Operator, sofern lokal vorhanden, über effiziente Protokolle mit dem Prozess interagiert oder mithilfe externer Kommunikation auf einem entfernten Serverknoten aufgerufen werden muss.

Das Konzept des *Infrastruktur-Design-Advisors* bietet Forschungsfragen in den unterschiedlichen Teilgebieten. So erwies sich im Rahmen des Kostenmodells die Schätzung der verursachten Prozessorlast von Prozessen bei geringen Eingabedatenraten und nebenläufigen Instanzen als ungenau. Eine detailliertere Analyse der Prozessorlast in Abhängigkeit der Eingabedatenrate, des Operatortyps und der nebenläufigen Ausführung von Prozessen scheint notwendig. Auch ist es möglich, existierende Kostenmodelle für die Spezifika der XML-Kommunikation zu integrieren, um beispielsweise die Schätzung der Kommunikationskosten als Mehraufwand zur Verarbeitungszeit eines Operators zu verbessern. Auf Ebene der Optimierungsalgorithmen sollten im Rahmen der spezifischen Prozess- und Dienstverteilung zusätzliche Optimierungsalgorithmen analysiert werden, um gegebenenfalls spezielle Optimierungsalgorithmen zur Problemlösung zu konzipieren.

Derzeit verteilt der *Infrastruktur-Design-Advisor* ganzheitliche Prozesse. In einem zweiten Schritt ist es jedoch möglich, die definierten Prozesse zu fragmentieren und die einzelnen Prozessfragmente feingranular auf die Serverknoten zu verteilen. Dazu muss untersucht werden, inwieweit das derzeitige Kostenmodell sowie die Suchalgorithmen eine solche fragmentierte Verteilung ermöglichen und ob nicht die performance-kritische XML-basierte Kommunikation eine feingranulare Verteilung von Prozessfragmenten verhindert.

Literaturverzeichnis

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005.

[2] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in db2 udb. In *Proceedings of the Thirtieth international conference on very large data bases - Volume 30*, pages 1158–1169. VLDB Endowment, 2004.

[3] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4:319–364, 1995.

[4] N. Abu-Ghazaleh and M. Lewis. Differential deserialization for optimized soap performance. In *Proceedings of the 2005 Conference on Supercomputing*, page 21, Seattle, WA, USA, 2005.

[5] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential serialization for optimized soap performance. In *HPDC04*, pages 55–64, 2004.

[6] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005: demo. In *Proceedings of the 2005 ACM SIGMOD international conference on management of data*, pages 930–932, New York, NY, USA, 2005.

[7] G. Alonso, B. Reinwald, and C. Mohan. Distributed data management in workflow environments. In *Proceedings of the 7th International Workshop on Research Issues in Data Engineering (RIDE '97) High Performance Database Management for Large-Scale Applications*, pages 82–, Washington, DC, USA, 1997.

[8] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. Daml-s: Web service description for the semantic web. In *Second International Semantic Web Conference (ISWC 02), USA*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2002.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[10] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, June 2006.

[11] V. Atluri, S. Chun, R. Mukkamala, and P. Mazzoleni. A decentralized execution model for inter-organizational workflows. *Distributed and Parallel Databases*, 22:55–83, 2007. 10.1007/s10619-007-7012-1.

[12] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of the 2004 ACM SIGMOD international conference on management of data*, SIGMOD '04, pages 419–430, New York, NY, USA, 2004.

[13] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, Sept. 2001.

[14] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3:1–3:44, Mar. 2008.

[15] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 15–15, Berkeley, CA, USA, 2004.

[16] L. Baresi, A. Maurino, and S. Modafferi. Towards distributed bpel orchestrations. *Electronic Communications of the EASST*, 3, 2006.

[17] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, Online Proceedings*, pages 363–374. www.cidrdb.org, 2007.

[18] A. Bauer and H. Günzel. *Data Warehouse Systeme - Architektur, Entwicklung, Anwendung.* dpunkt-Verlag, Heidelberg, 3 edition, 2009.

[19] T. Bauer, M. Reichert, and P. Dadam. Intra-subnet load balancing in distributed workflow management systems. *International Journal of Cooperative Information Systems*, 12(3):295–324, 2003.

[20] O. Becker. *Serielle Transformationen von XML.* PhD thesis, Humboldt-Universität zu Berlin, 2004.
`http://edoc.hu-berlin.de/docviews/abstract.php?id=20895`.

[21] M. Bhide, M. Agarwal, A. Bar-Or, S. Padmanabhan, S. Mittapalli, and G. Venkatachaliah. Xpedia: Xml processing for data integration. *The Proceedings of*

the Very Large Data Base Endowment, 2(2):1330–1341, 2009.

[22] B. Bioernstad. *A Workflow Approach to Stream Processing.* Phd thesis, ETH Zurich, Computer Science Department, 2008.

[23] B. Bioernstad, C. Pautasso, and G. Alonso. Control the flow: How to safely compose streaming services into business processes. In *Proceedings of the 2006 IEEE International Conference on Services Computing (SCC 2006), Chicago, Illinois, USA, 18-22 September*, pages 206–213, 2006.

[24] B. Biörnstad and C. Pautasso. Let it flow: Building mashups with data processing pipelines. In *ICSOC Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2007.

[25] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *SIGOPS Operating Systems Review*, 13:20–25, April 1979.

[26] M. Boehm. *Cost-Based Optimization of Integration Flows.* PhD thesis, Technische Universität Dresden, 2011.

[27] M. Böhm, D. Habich, and W. Lehner. Multi-flow optimization via horizontal message queue partitioning. In *Proceedings of the 2010 International Conference on Enterprise Information Systems (ICEIS 2010), Funchal, Madeira, Portugal, June 8 - 12*, volume 73 of *Lecture Notes in Business Information Processing*, pages 31–47, 2010.

[28] M. Böhm, D. Habich, W. Lehner, and U. Wloka. Systemübergreifende kostennormalisierung für integrationsprozesse. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, Münster, 2.–6. März*, pages 67–86, 2009.

[29] M. Böhm, D. Habich, S. Preissler, W. Lehner, and U. Wloka. Cost-based vectorization of instance-based integration processes. In *Proceedings of the 2009 East European Conference on Advances in Databases and Information Systems (ADBIS 2009), Riga, Latvia, September 7-10*, pages 253–269. Springer, 2009.

[30] M. Böhm, D. Habich, S. Preissler, W. Lehner, and U. Wloka. Vectorizing instance-based integration processes. In *Proceedings of the 2009 International Conference on Enterprise Information Systems (ICEIS 2009), Milan, Italy, May 6-10*, pages 40–52, 2009.

[31] M. Böhm, U. Wloka, D. Habich, and W. Lehner. Workload-based optimization of integration processes. In *Proceedings of the 2008 Conference on Information and Knowledge Management, Napa Valley, USA*, pages 1479–1480, 2008.

[32] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending xquery with window functions. In *Proceedings of the 2007 international conference on Very large data bases (VLDB 2007)*, pages 75–86, Vienna, Austria, 2007.

[33] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102, Bejing, China, 2007.

[34] A. Brito, C. Fetzer, and P. Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *Proceedings of the 2009 International Conference on Distributed Computing Systems (ICDCS 2009)*, 2009.

[35] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the 2008 International Conference on Distributed Event-Based Systems (DEBS 2008), Rome, Italy, July 1-4*, pages 265–275, 2008.

[36] F. Buytendijk and D. Flint. Bam architecture: More building blocks than you think. *Gartner Research Note ID Number AV-15-5070*, 2002. `http://www.gartner.com/DisplayDocument?doc_cd=105560&ref=g_fromdoc`.

[37] F. Buytendijk and D. Flint. How bam can turn a business into a real-time enterprise. *Gartner Research Note ID Number AV-15-4650*, 2002. `http://www.gartner.com/DisplayDocument?doc_cd=105278&ref=g_fromdoc`.

[38] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81:463–480, April 2008.

[39] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *Proceedings of the 2004 international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004.

[40] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, Online Proceedings*. www.cidrdb.org, 2003.

[41] S. Chandrasekaran, J. A. Miller, G. A. Silver, I. B. Arpinar, and A. P. Sheth. Performance analysis and simulation of composite web services. *Electronic Markets*, 13(2), 2003.

[42] S. Chaudhuri and V. Narasayya. Autoadmin ‘what-if’ index analysis utility. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 367–378, New York, NY, USA, 1998.

[43] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 2007 international conference on Very large data*

bases (VLDB 2007), pages 3–14, Vienna, Austria, 2007. VLDB Endowment.

[44] S. Chen, P. B. Gibbons, and S. Nath. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *Proceedings of the 2010 SIGMOD International Conference on Management of data*, pages 147–158, New York, NY, USA, 2010.

[45] S. Chen, B. Yan, J. Zic, R. Liu, and A. Ng. Evaluation and modeling of web services performance. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 437–444. IEEE Computer Society, 2006.

[46] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC02*, pages 246–254, 2002.

[47] D. Davis and M. P. Parashar. Latency performance of soap implementations. In *Proceedings of the 2002 International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, pages 407–, 2002.

[48] U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.

[49] A. Demers, J. Gehrke, and P. Biswanath. Cayuga: A general purpose event monitoring system. In *Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, Online Proceedings*, pages 412–422. www.cidrdb.org, 2007.

[50] K. Devaram and D. Andresen. Soap optimization via parameterized client-side caching. In *Proceedings of the 2003 International Conference on Web Services, ICWS '03, June 23 - 26, 2003, Las Vegas, Nevada, USA*, pages 520–. CSREA Press, 2003.

[51] K. Devaram and D. Andresen. Soap optimization via parameterized client-side caching. In *Parallel and Distributed Computing and Systems*, Calgary, Canada, 2003.

[52] F. Dittmar. Prozessbasierte Datenstromverarbeitung für Prozess- und Dienst-Monitoring. Belegarbeit. Technische Universität Dresden, 2010.

[53] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *Proceedings of the 2002 international conference on Very Large Data Bases (VLDB 2002)*, pages 299–310. VLDB Endowment, 2002.

[54] T. Dornemann, E. Juhnke, and B. Freisleben. On-demand resource provisioning for bpel workflows using amazon's elastic compute cloud. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 140–147, Washington, DC, USA, 2009. IEEE

Computer Society.

[55] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of soap in web service environment compared to corba. In *Proceedings of the 2002 Asia-Pacific Software Engineering Conference*, pages 84–, Washington, DC, USA, 2002.

[56] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[57] P. M. Fischer, A. Garg, and K. S. Esmaili. Extending xquery with a pattern matching facility. In *Proceedings of the 2010 International XML Database Symposium (XSym 2010), Singapore, September 17*, pages 48–57, 2010.

[58] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[59] P. Garcia and H. F. Korth. Pipelined hash-join on multithreaded architectures. In *Proceedings of the 2007 international workshop on Data management on new hardware (DaMoN 2007)*, pages 1:1–1:8, Beijing, China, 2007.

[60] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[61] M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of xml over the web. *Computer Networks*, 33(1-6):747–765, 2000.

[62] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. Self-optimizing block transfer in web service grids. In *Proceedings of the 2007 ACM International Workshop on Web Information and Data Management (WIDM 2007), Lisbon, Portugal, November 9*, pages 49–56, 2007.

[63] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *Proceedings of the 2008 International Conference on Data Engineering (ICDE 2008), Cancún, México, April 7-12*, pages 596–605, 2008.

[64] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. v. Engelen, and M. J. Lewis. Toward characterizing the performance of soap toolkits. In *Proceedings of the 2004 International Workshop on Grid Computing (GRID 2004), November 8*, pages 365–372, Pittsburgh, USA, 2004.

[65] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on management of data*, pages 102–111, Atlantic City, NJ, USA, 1990.

[66] G. Graefe. Sort-merge-join: an idea whose time has(h) passed? In *Proceedings of the 1994 International Conference on Data Engineering (ICDE 1994), Houston, Texas, USA, February 14-18*, pages 406 –417, 1994.

[67] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[68] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml,Volume 1.* John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[69] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 287–298, New York, NY, USA, 1999.

[70] D. Habich. *Komplexe Datenanalyseprozesse in serviceorientierten Umgebungen.* PhD thesis, Technische Universität Dresden, 2008.

[71] D. Habich, S. Preissler, W. Lehner, S. Richly, U. Aßmann, M. Grasselt, and A. Maier. Data-grey-boxweb services in data-centric environments. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 976–983. IEEE Computer Society, 2007.

[72] D. Habich, S. Preissler, H. Voigt, and W. Lehner. Innovative process execution in service-oriented environments. In *Proceedings of the 2009 International Conference on Enterprise Information Systems (ICEIS 2009), Milan, Italy, May 6-10*, pages 299–302, 2009.

[73] D. Habich, S. Richly, A. Ruempel, W. Buecke, and S. Preissler. Open service process platform 2.0. In *Proceedings of the 2008 IEEE Congress on Services - Part I (SERVICES 2008), July 6-11*, pages 152 –159, Haway, USA, 2008.

[74] H.-I. Hsiao, M.-S. Chen, and P. S. Yu. On parallel execution of multiple pipelined hash joins. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 185–196, New York, NY, USA, 1994.

[75] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 2005 International Conference on Data Engineering (ICDE 2005), Tokyo, Japan, 5-8 April*, pages 779–790, 2005.

[76] J.-H. Hwang, S. Cha, U. Cetintemel, and S. Zdonik. Borealis-r: a replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1303–1306, New York, NY, USA, 2008.

[77] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proceedings of the 2007 International Conference on Data Engineering (ICDE 2007), Istan-*

bul, Turkey, April 15-20, pages 176–185, 2007.

[78] IDC. Soa in Deutschland: Beratungskompetenz als Erfolgsfaktor. *Report #GY08P*, 2007.

[79] K. Imasaki, H. Nguyen, and S. Dandamudi. Performance comparison of pipelined hash joins on workstation clusters. In S. Sahni, V. Prasanna, and U. Shukla, editors, *Proceedings of the 2002 International Conference on High Performance Computing*, volume 2552 of *Lecture Notes in Computer Science*, pages 264–275. Springer Berlin / Heidelberg, 2002.

[80] G. Imre, M. Kaszó, T. Levendovszky, and H. Charaf. A novel cost model of xml serialization. *Electronic Notes in Theoretical Computer Science*, 261:147–162, 2010.

[81] M. Ivanova and T. Risch. Customizable parallel execution of scientific stream queries. In *Proceedings of the 2005 International Conference on Very large data bases (VLDB 2005)*, pages 157–168. VLDB Endowment, 2005.

[82] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql standard. *The Proceedings of the Very Large Data Base Endowment*, 1:1379–1390, August 2008.

[83] R. Jiménez-Peris, M. Patiño-Martínez, and E. Martel-Jordán. Decentralized web service orchestration: a reflective approach. In *Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, March 16 - 20*, pages 494–498, 2008.

[84] D. Jobst and G. Preissler. Mapping clouds of soa- and business-related events for an enterprise cockpit in a java-based environment. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 230–236, New York, NY, USA, 2006. ACM.

[85] T. Johnson, M. S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1135–1146, New York, NY, USA, 2008.

[86] N. M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2 edition, 2008.

[87] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 2003 International Conference on Data Engineering (ICDE 2003), Bangalore, India, March 5-8*, pages 341 – 352, 2003.

[88] C. Kecher. *UML 2.0 - Das umfassende Handbuch*. Galileo Press, 2006.

[89] I. Kellner and L. Fiege. Viewpoints in complex event processing: industrial experience report. In *Proceedings of the 2009 International Conference on Distributed Event-Based Systems (DEBS 2009), Nashville, Tennessee, USA, July 6-9*, pages 1–8, 2009.

[90] R. Khalaf and F. Leymann. E role-based decomposition of business processes using bpel. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006)*, pages 770–780, Washington, DC, USA, 2006.

[91] B. Kiepuszewski, A. ter Hofstede, and W. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39:143–209, 2003.

[92] C. Kleiner and A. Koschel. Praxisfallbeispiel: Modernisierung einer Mainframe-Anwendung durch eine verteilte SOA. *Electronic Communications of the EASST*, 17(0), 2009.

[93] C. Kohlhoff and R. Steele. Evaluating soap for high performance applications in capital markets. *Computer Systems Science and Engineering*, 19(4), 2004.

[94] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-a*. *Artificial Intelligence*, 129(1-2):199 – 218, 2001.

[95] R. Kouzes, G. Anderson, S. Elbert, I. Gorton, and D. Gracio. The changing paradigm of data-intensive computing. *Computer*, 42(1):26 –34, January 2009.

[96] J. Kraemer and B. Seeger. Pipes - a public infrastructure for processing and exploring streams. In *Proceedings of the 2004 ACM SIGMOD international conference on management of data*, pages 925–926, Paris, France, 2004.

[97] J. Krämer and B. Seeger. A temporal foundation for continuous queries over data streams. In *Proceedings of the 2005 International Conference on Management of Data, Goa, India, January 6 - 8*, pages 70–82, 2005.

[98] W. Lehner. *Datenbanktechnologie für Data-Warehouse-Systeme - Konzepte und Methoden.* dpunkt.verlag Heidelberg, 2003.

[99] F. Leymann. Web services: Distributed applications without limits. In *10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web , Leipzig, 26.-28. Februar*, pages 2–23, 2003.

[100] F. Leymann and D. Roller. *Production workflow: concepts and techniques.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[101] H. Li and D. Zhan. Workflow timed critical path optimization. *Nature and Science*, 3(2), 2005.

[102] H. Liefke and D. Suciu. Xmill: An efficient compressor for xml data. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data*, pages 153–164, Dallas, Texas, USA, 2000.

[103] D. Liu, K. H. Law, and G. Wiederhold. Analysis of integration models for service composition. In *Proceedings of the 2002 International Workshop on Software and Performance (IWOSP 2002)*, pages 158–165, Rome, Italy, 2002.

[104] W. D. Liu. *A distributed data flow model for composing software services.* PhD thesis, Stanford University, Stanford, CA, USA, 2003. Adviser-Gio Wiederhold.

[105] Y. Liu and B. Plale. Multi-model based optimization for stream query processing. In K. Zhang, G. Spanoudakis, and G. Visaggio, editors, *Proceedings of the 2006 International Conference on Software Engineering & Knowledge Engineering (SEKE 2006), San Francisco, CA, USA, July 5-7*, pages 150–155, 2006.

[106] Y. Liu and B. Plale. Query optimization for distributed data streams. In *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), Los Angeles, CA, USA July 6-8*, pages 259–266, 2006.

[107] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[108] A. C. C. Machado and C. A. G. Ferraz. Guidelines for performance evaluation of web services. In *WebMedia '05: Proceedings of the 11th Brazilian Symposium on Multimedia and the web*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[109] A. Maier, B. Mitschang, F. Leymann, and D. Wolfson. On combining business process integration and etl technologies. In *11. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, Karlsruhe, 2.-4. März*, pages 533–546, 2005.

[110] S. Maneth, N. Mihaylov, and S. Sakr. Xml tree structure compression. In *Proceedings of the 2008 International Conference on Database and Expert Systems Application (DEXA 2008)*, pages 243–247, 2008.

[111] V. Markl and G. Lohman. Learning table access cardinalities with leo. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 613–613, Madison, Wisconsin, USA, 2002.

[112] D. Martin, D. Wutke, and F. Leymann. A novel approach to decentralized workflow enactment. *IEEE International Conference on Enterprise Distributed Object Computing*, 0:127–136, 2008.

[113] G. Matthiessen and M. Unterstein. *Relationale Datenbanken und Standard-SQL: Konzepte der Entwicklung und Anwendung.* Addison Wesley Verlag, 2007.

[114] I. Melzer. *Service-orientierte Architekturen mit Web Services*. Spektrum Akademischer Verlag, Elsevier GmbH, München, 4 edition, 2010.

[115] R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[116] B. M. Michelson. Event-driven architecture overview. Technical report, Patricia Seybold Group, 2006.

[117] L. mircea Patcas, J. Murphy, and G. miro Muntean. Middleware support for data-flow distribution in web services composition. In *In The PhDOOS Workshop and Doctoral Symposium*, 2005.

[118] A. Mohan, G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A. E. Abbadi, and M. Kamath. Exotica/fmqm: A persistent message-based architecture for distributed workflow management. In *Proceedings IFIP Working Conference on Information Systems for Decentralized Organizations*, volume 8, pages 1–18, Trondheim, 1995.

[119] H. Mucksch and W. Behme. *Das Data Warehouse Konzept. Architektur-Datenmodelle - Anwendungen*. Gabler Verlag, Wiesbaden, 4 edition, 2000.

[120] P. Muth, D. Wodtke, J. Weissenfels, A. K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflowexecution. *Journal of Intelligent Information Systems*, 10:159–184, March 1998.

[121] Y. V. Natis. Service-oriented architecture scenario. *Gartner Research Note ID Number AV-19-6751*, 2003.
`http://www.gartner.com/DisplayDocument?id=391595`.

[122] A. Ng. Optimising web services performance with table driven xml. In *Proceedings of the 2006 Australian Software Engineering Conference ,Sydney, Australia, April 18-21*, pages 100–112, 2006.

[123] W. Ng, W. Y. Lam, P. T. Wood, and M. Levene. Xcq: A queriable xml compression system. *Knowledge and Information Systems*, 10(4):421–452, 2006.

[124] J. Pearl and R. E. Korf. Search techniques. *Annual Review of Computer Science*, 2:451–467, 1987.

[125] S. Preissler, D. Habich, and W. Lehner. Cost-based business process deployment advisor. In *Proceedings of the 2011 International Conference on Enterprise Information Systems (ICEIS 2011), Beijing, China, 8-11 June*, pages 211–216, 2011.

[126] P. Rajasekaran, J. Miller, K. Verma, and A. Sheth. Enhancing web services description and discovery to facilitate composition. In *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 55–68. Springer Berlin / Heidelberg, 2005.

[127] G. Reese. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2000.

[128] K. A. Ross. Optimizing read convoys in main-memory query processing. In *Proceedings of the 2010 International Workshop on Data Management on New Hardware*, pages 27–33, New York, NY, USA, 2010.

[129] M.-C. Rosu. A-soap: Adaptive soap message processing and compression. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 200–207. IEEE Computer Society, 2007.

[130] W. A. Ruh, W. J. Brown, and F. X. Maginnis. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[131] S. Russell and P. Norvig. *Künstliche Intelligenz: Ein moderner Ansatz*, volume 2. Auflage. Pearson Studium, 2004.

[132] S. Schmidt, T. Legler, S. Schär, and W. Lehner. Robust real-time query processing with qstream. In *Proceedings of the 2005 International Conference on Very large data bases (VLDB 2005)*, pages 1299–1301. VLDB Endowment, 2005.

[133] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 110–121, New York, NY, USA, 1989.

[134] W. R. Schulte and Y. V. Natis. Most composite applications will need an integration layer. *Gartner Research Note ID Number DF-19-5175*, 2003. `http://www.gartner.com/DisplayDocument?doc_cd=114284`.

[135] R. Sen, G.-C. Roman, and C. Gill. Cian: a workflow engine for manets. In *Proceedings of the 2008 international conference on Coordination models and languages*, pages 280–295, Oslo, Norway, 2008.

[136] S. Seshadri, V. Kumar, B. F. Cooper, and L. Liu. Optimizing multiple distributed stream queries using hierarchical network partitions. In *IPDPS*, pages 1–10. IEEE, 2007.

[137] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings of the 2003 International Conference on Data Engineering (ICDE 2003), Bangalore, India, March 5-8*, pages 25 – 36, 2003.

[138] P. Shivam, S. Babu, and J. S. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *Proceedings of the 2006 international conference on Very large data bases (VLDB 2006)*, pages 535–546, Seoul, Korea, 2006.

[139] J. Shneidman, P. R. Pietzuch, M. Welsh, M. I. Seltzer, and M. Roussopoulos. A cost-space approach to distributed query optimization in stream based overlays. In *Proceedings of the 2005 International Conference on Data Engineering (ICDE 2005), Tokyo, Japan, 5-8 April*, page 1182, 2005.

[140] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziovara. Benchmarking etl workflows. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 199–220. Springer Berlin / Heidelberg, 2009.

[141] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing etl processes in data warehouses. In *Proceedings of the 2005 International Conference on Data Engineering (ICDE 2005), Tokyo, Japan, 5-8 April*, pages 564–575, 2005.

[142] T.-X. Song, P.-J. Tian, Y.-H. Liu, and B.-Q. Huang. Web services' semantic annotation and auto-matching based on sawsdl. *International Symposium on Information Science and Engieering*, 2:577–580, 2008.

[143] J. Spillner. *Methodik und Referenzarchitektur zur inkrementellen Verbesserung der Metaqualität einer vertragsgebundenen, heterogenen und verteilten Dienstausführung.* PhD thesis, Technische Universität Dresden, 2010.

[144] J. Spillner, A. Kümpel, I. Braun, and A. Schill. Infrastruktur zur experimentellen evaluierung von konzepten im internet der dienste. In *Informatik 2010: Service Science - Neue Perspektiven für die Informatik, GI Fachtagung, 27.09. - 1.10.*, pages 509–514, Leipzig, 2010.

[145] J. Spillner, M. Winkler, S. Reichert, J. Cardoso, and A. Schill. Distributed contracting and monitoring in the internet of services. In *Distributed Applications and Interoperable Systems*, volume 5523 of *Lecture Notes in Computer Science*, pages 129–142. Springer Berlin / Heidelberg, 2009.

[146] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proceedings of the 2006 international conference on Very large data bases (VLDB 2006)*, pages 355–366, Seoul, Korea, 2006.

[147] V. Stiehl. *Composite Application Systems - Systematisches Konstruieren von Verbundanwendungen unter Verwendung von BPMN.* PhD thesis, TU Darmstadt, September 2011.

[148] T. Suzumura, T. Takase, and M. Tatsubori. Optimizing web services performance by differential deserialization. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA*, pages 185–192. IEEE Computer Society, 2005.

[149] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, New York, NY, USA, 1992.

[150] M. Thiele. *Qualitätsgetriebene Datenproduktionssteuerung in Echtzeit-Data-Warehouse-Systemen*. PhD thesis, Technische Universität Dresden, 2010.

[151] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[152] R. van Engelen. Pushing the soap envelope with web services for scientific computing. In *Proceedings of the 2003 International Conference on Web Services, ICWS '03, June 23 - 26, 2003, Las Vegas, Nevada, USA*, pages 346–352. CSREA Press, 2003.

[153] P. Vassiliadis, A. Simitsis, and E. Baikousi. A taxonomy of etl activities. In *Proceedings of the2009 International Workshop on Data Warehousing and OLAP (DOLAP 2009)*, pages 25–32, Hong Kong, China, 2009.

[154] K. Vidackovic, I. Kellner, and J. Donald. Business-oriented development methodology for complex event processing: demonstration of an integrated approach for process monitoring. In *Proceedings of the 2010 International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 111–112, 2010.

[155] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 37–48, Madison, Wisconsin, USA, 2002.

[156] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. In *Proceedings of the 2007 international conference on Very large data bases (VLDB 2007)*, pages 615–626, Vienna, Austria, 2007.

[157] M. Vrhovnik, O. Suhre, S. Ewen, and H. Schwarz. Pgm/f: A framework for the optimization of data processing in business processes. In *Proceedings of the 2008 International Conference on Data Engineering (ICDE 2008), Cancún, México, April 7-12*, pages 1584–1587. IEEE, 2008.

[158] S. Wang, Z. Tan, and X. Gao. Query optimization over distributed data stream. In G. Yu, M. Köppen, S.-M. Chen, and X. Niu, editors, *HIS (2)*, pages 415–418. IEEE Computer Society, 2009.

[159] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[160] M. Weinard. *Analyse von Heuristiken*. PhD thesis, Johann-Wolfgang-Goethe-Universität Frankfurt, 2006.

[161] A. M. Weiner. Advanced cardinality estimation in the xml query graph model. In *14. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web,*

Kaiserslautern, 28.Februar - 4.März, pages 207–226, 2011.

[162] A. M. Weiner and T. Härder. Using structural joins and holistic twig joins for native xml query optimization. In *Proceedings of the 2009 East European Conference on Advances in Databases and Information Systems (ADBIS 2009), Riga, Latvia, September 7-10*, pages 149–163. Springer, 2009.

[163] C. Werner, C. Buschmann, Y. Brandt, and S. Fischer. Compressing soap messages by using pushdown automata. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 19–28. IEEE Computer Society, 2006.

[164] C. Werner, C. Buschmann, and S. Fischer. Compressing soap messages by using differential encoding. In *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*, pages 540–. IEEE Computer Society, 2004.

[165] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Commununication of the ACM*, 35:89–99, November 1992.

[166] S. Zaplata, K. Hamann, K. Kottke, and W. Lamersdorf. Flexible execution of distributed business processes based on process instance migration. *Journal of Systems Integration (JSI)*, 1(3):3–16, 7 2010.

[167] W. Zhang and R. A. van Engelen. A table-driven streaming xml parsing methodology for high-performance web services. In *Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 197–204. IEEE Computer Society, 2006.

[168] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: integrated automatic physical database design. In *Proceedings of the 2004 international conference on Very large data bases (VLDB 2004)*, pages 1087–1097, Toronto, Canada, 2004. VLDB Endowment.

[169] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337 – 343, 1977.

Online-Quellenverzeichnis

[170] D. Almaer. Json vs. xml: The debate.
http://ajaxian.com/archives/json-vs-xml-the-debate
Zuletzt besucht: 06.07.2012.

[171] Amazon Web Services. Amazon simpledb, 2012.
http://aws.amazon.com/de/simpledb/
Zuletzt besucht: 06.07.2012.

[172] Apache Software Foundation. Activemq, 2012.
http://activemq.apache.org/
Zuletzt besucht: 06.07.2012.

[173] Apache Software Foundation. Apache couchdb, 2012.
http://couchdb.apache.org/
Zuletzt besucht: 06.07.2012.

[174] Apache Software Foundation. Apache ODE, 2012.
http://ode.apache.org/
Zuletzt besucht: 06.07.2012.

[175] The Internet Engineering Task Force. Javascript object notation (json). RFC 4627, 2006.
http://tools.ietf.org/html/rfc4627
Zuletzt besucht: 06.07.2012.

[176] R. R. M. GmbH. Rtm analyzer, 2012.
http://www.realtime-monitoring.de/
Zuletzt besucht: 06.07.2012.

[177] IBM Corporation. Infosphere DataStage.
http://www-01.ibm.com/software/data/infosphere/datastage/
Zuletzt besucht: 06.07.2012.

[178] IBM Corporation. Websphere Business Events.
http://www-01.ibm.com/software/integration/wbe/
Zuletzt besucht: 06.07.2012.

[179] IBM Corporation. BPELJ: BPEL for Java, 2004.
http://www-128.ibm.com/developerworks/library/specification/ws-bpelj/
Zuletzt besucht: 06.07.2012.

[180] IBM Corporation. Ibm global ceo study 2008, 2008.
http://www-935.ibm.com/services/de/bcs/html/ceostudy.html
Zuletzt besucht: 06.07.2012.

[181] IBM Corporation. A new way of working - insights from global leaders, 2010.
http://www-01.ibm.com/software/solutions/smartwork/study/?re=ussph3.1.1
Zuletzt besucht: 06.07.2012.

[182] IBM Corporation. IBM Business Process Manager, 2012.
http://www-01.ibm.com/software/integration/business-process-manager/
Zuletzt besucht: 06.07.2012.

[183] IBM Corporation. IBM Websphere MQ, 2012.
http://www-01.ibm.com/software/integration/wmq/
Zuletzt besucht: 06.07.2012.

[184] JBoss Community. Jboss enterprise service bus.
http://jboss.org/jbossesb
Zuletzt besucht: 06.07.2012.

[185] JBoss Community. Drools - the business logic integration platform, 2012.
http://www.jboss.org/drools
Zuletzt besucht: 06.07.2012.

[186] MANAPPS. Etl benchmarks v 1.1, 2012.
http://www.manapps.tm.fr/index.php?/benchmark-etl.html
Zuletzt besucht: 06.07.2012.

[187] OASIS. Soa reference model, 2006.
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm
Zuletzt besucht: 06.07.2012.

[188] OASIS. Web services business process execution language 2.0 (ws-bpel), 2007.
http://www.oasis-open.org/committees/wsbpel/
Zuletzt besucht: 06.07.2012.

[189] OMG. Business process modeling language 1.2, 2009.
http://www.omg.org/spec/BPMN/1.2/PDF/
Zuletzt besucht: 06.07.2012.

[190] Pentaho Corporation. Pentaho business analytics, 2012.
http://www.pentaho.com/explore/pentaho-business-analytics/
Zuletzt besucht: 06.07.2012.

[191] Pentaho Corporation. Pentaho data integration, 2012.
http://www.pentaho.com/explore/pentaho-data-integration/
Zuletzt besucht: 06.07.2012.

[192] rapid-i GmbH. Rapid miner, 2012.
http://rapid-i.com/content/view/181/190/lang,de/
Zuletzt besucht: 06.07.2012.

[193] Red Hat Inc. Jboss web services, 2012.
http://www.jboss.org/jbossws
Zuletzt besucht: 06.07.2012.

[194] SAP AG. Universal description language 3.0, 2010.
http://www.internet-of-services.com/index.php?id=382
Zuletzt besucht: 06.07.2012.

[195] T. Schmale. Bpm 2.0 ist mehr als soa, 2008.
http://www.ap-verlag.de/Online-Artikel/20080304/20080304m%20inubit%20BPM%2020%20SOA%20Geschaeftsprozesse.htm
Zuletzt besucht: 06.07.2012.

[196] Software AG. Aris platform, 2012.
http://www.softwareag.com/corporate/products/aris_platform/default.asp
Zuletzt besucht: 06.07.2012.

[197] H. Stiel. Ganzheitliche it-lösungen gefragt, 2010.
http://www.ap-verlag.de/Online-Artikel/20100708/20100708h%20Ganzheitliche%20IT-Loesungen.htm
Zuletzt besucht: 06.07.2012.

[198] Streambase Systems. Streamsql online documentation, 2010. http://streambase.com/developers/docs/latest/streamsql/index.html
Zuletzt besucht: 06.07.2012.

[199] StreamBase Systems. Streambase event processing platform, 2012. http://www.streambase.com/products/streambasecep/
Zuletzt besucht: 06.07.2012.

[200] W3C. Extensible markup language (xml).
http://www.w3.org/XML/
Zuletzt besucht: 06.07.2012.

[201] W3C. Owl-s: Semantic markup for web services, 2004.
http://www.w3.org/Submission/OWL-S/
Zuletzt besucht: 06.07.2012.

[202] W3C. Xml information set, 2004.
http://www.w3.org/TR/xml-infoset/
Zuletzt besucht: 06.07.2012.

[203] W3C. Xml schema, 2004.
http://www.w3.org/XML/Schema
Zuletzt besucht: 06.07.2012.

[204] W3C. Soap version 1.2, 2007.
`http://www.w3.org/TR/soap/`
Zuletzt besucht: 06.07.2012.

[205] W3C. Web service description language 2.0, 2007.
`http://www.w3.org/TR/wsdl20/`
Zuletzt besucht: 06.07.2012.

[206] W3C. Xsl transformations (xslt) version 2.0, 2007.
`http://www.w3.org/TR/xslt20/`
Zuletzt besucht: 06.07.2012.

[207] W3C. Efficient xml interchange format, 2009.
`http://www.w3.org/TR/exi/`
Zuletzt besucht: 06.07.2012.

[208] W3C. Namespaces in xml 1.0 (third edition), 2009.
`http://www.w3.org/TR/xml-names/`
Zuletzt besucht: 06.07.2012.

[209] W3C. Web services activity, 2009.
`http://www.w3.org/2002/ws/`
Zuletzt besucht: 06.07.2012.

[210] W3C. Xml linking language (xlink) version 1.1, 2010.
`http://www.w3.org/TR/xlink11/`
Zuletzt besucht: 06.07.2012.

[211] W3C. Xml path language (xpath) 2.0, 2010.
`http://www.w3.org/TR/xpath`
Zuletzt besucht: 06.07.2012.

[212] W3C. Xquery 1.0, 2010.
`http://www.w3.org/TR/xquery/`
Zuletzt besucht: 06.07.2012.

[213] W3C. Web service activity, 2012.
`http://www.w3.org/2002/ws/`
Zuletzt besucht: 06.07.2012.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BAM	Business Activity Monitoring
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMN	Business Process Modeling Notation
CEP	Complex Event Processing
CQL	Continuous Query Language
DBMS	Database Management System
DIA	Datenintegration und Datenanalyse
DSMS	Data Stream Management System
EAI	Enterprise Application Integration
EDA	Event-Driven Architecture
ESB	Enterprise Service Bus
ETL	Extract, Transform, Load
FIFO	First In First Out
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAN	Local Area Network
REST	Representational State Transfer
SLA	Service Level Agreement
SOA	service-orientierte Architektur
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UDDI	Universal Description, Discovery and Integration

URL	Uniform Resource Locator
USDL	Universal Service Description Language
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
XDM	XQuery Data Model
XML	Extensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language

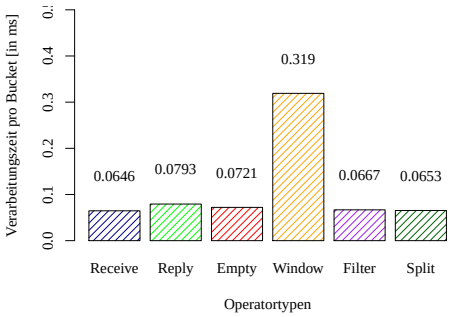
A Anhang

A.1 Testumgebung dieser Arbeit

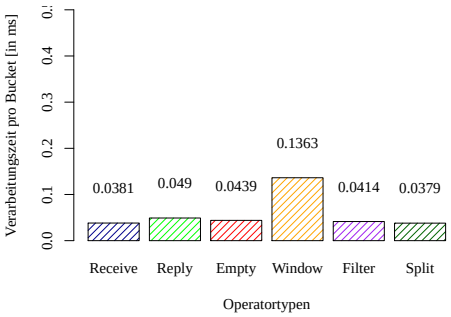
Knotenname	Hardwareeigenschaften
A	Intel Xeon CPU, 4x2.80GHz, 4 GB RAM, Fedora Linux 14, 32bit
B	Intel Core 2 Duo, 2x2,8 GHz, 3 GB RAM, Windows 7 Professional, 64bit
C	Intel Core 2 Duo, 2x2,8 GHz, 8 GB RAM, OS X 10.7, 64 bit
D	Intel Core Duo, 2x1,83 GHz, 2 GB RAM, Windows 7 Professional, 32bit
E	Intel Pentium D, 2x3,0 GHz, 2 GB RAM, Windows 7 Enterprise, 32bit

Tabelle A.1: Serverknoten des Testaufbaus.

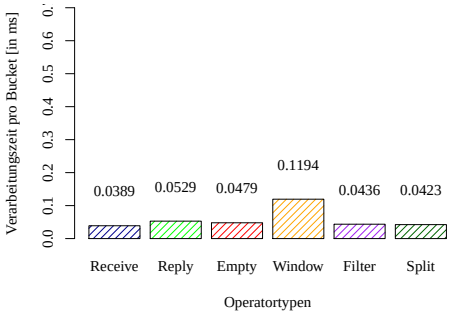
A.2 Weitere Messwerte Kostenmodell Kapitel 6



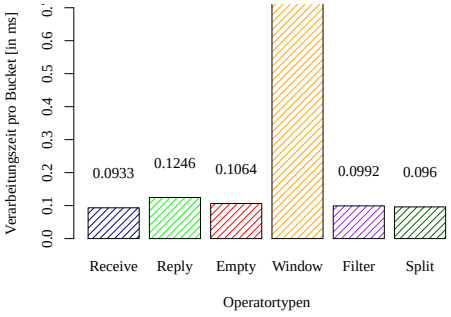
(a) Operatortypen auf Knoten A.



(b) Operatortypen auf Knoten B.

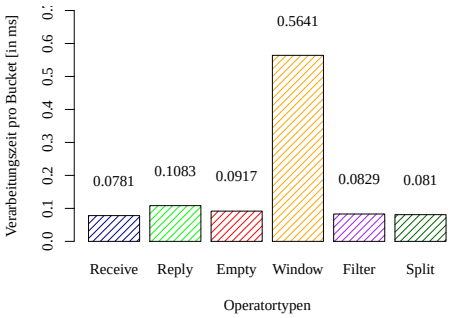


(c) Operatortypen auf Knoten C.

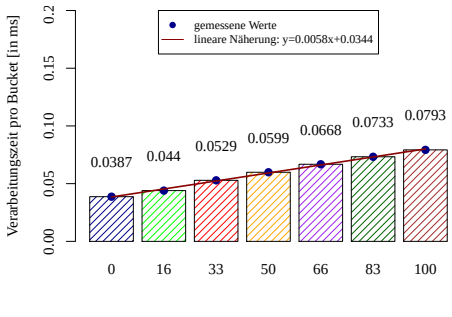


(d) Operatortypen auf Knoten D.

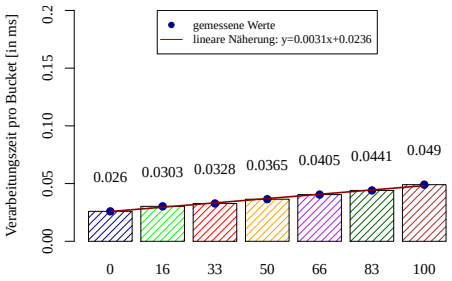
Abbildung A.1: Messungen für Kostenmodell Kapitel 6 (I).



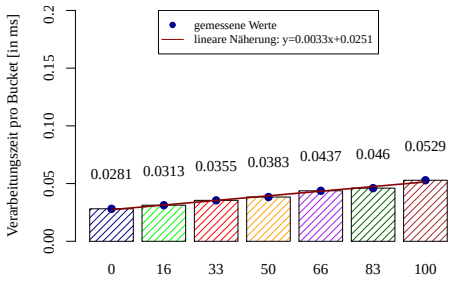
(a) Operatortypen auf Knoten E.



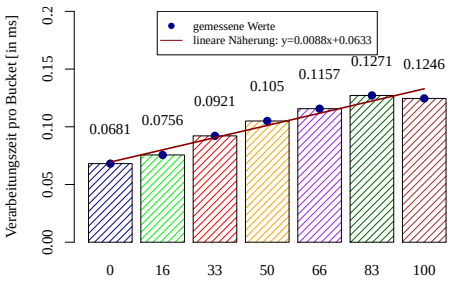
(b) Operatorselektivität (`filter`) Knoten A.



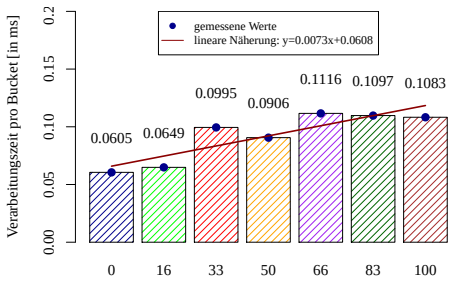
(c) Operatorselektivität (`filter`) Knoten B.



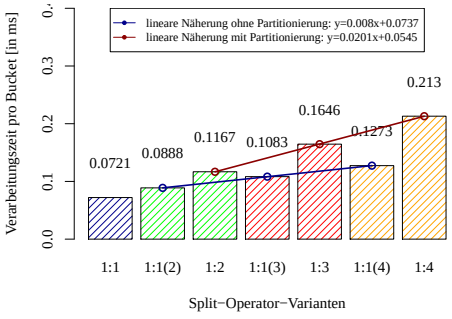
(d) Operatorselektivität (`filter`) Knoten C.



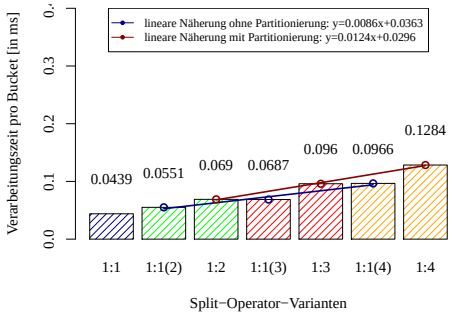
(e) Operatorselektivität (`filter`) Knoten D.



(f) Operatorselektivität (`filter`) Knoten E.

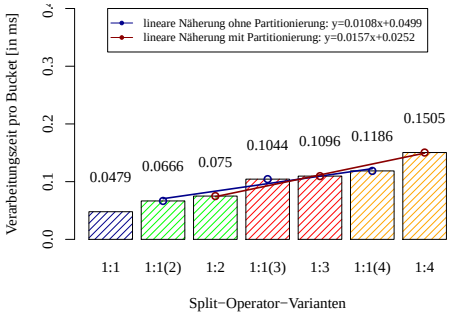


(g) Operatorselektivität (`split`) Knoten A.

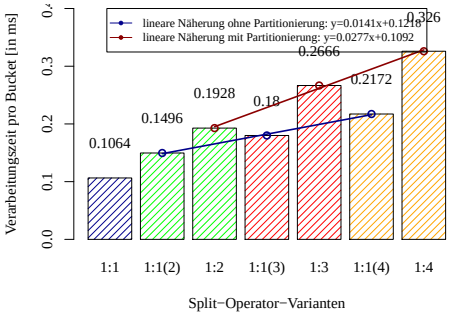


(h) Operatorselektivität (`split`) Knoten B.

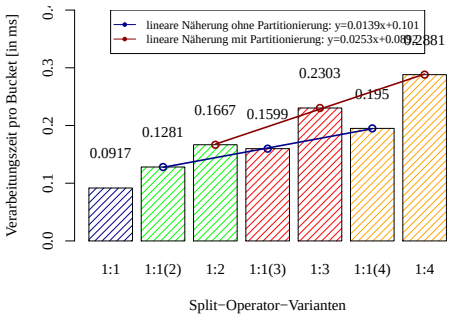
Abbildung A.2: Messungen für Kostenmodell Kapitel 6 (II).



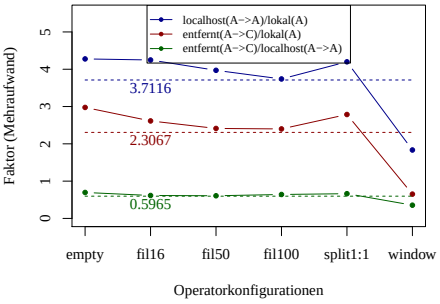
(a) Operatorselektivität (`split`) Knoten C.



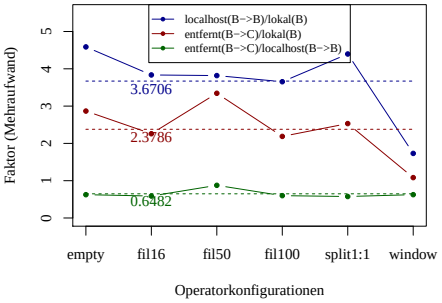
(b) Operatorselektivität (`split`) Knoten D.



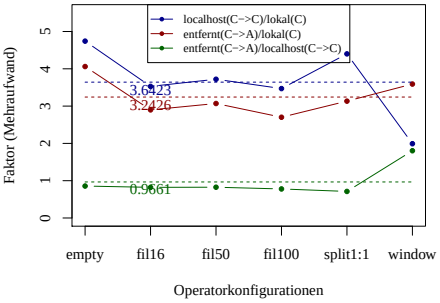
(c) Operatorselektivität (`split`) Knoten E.



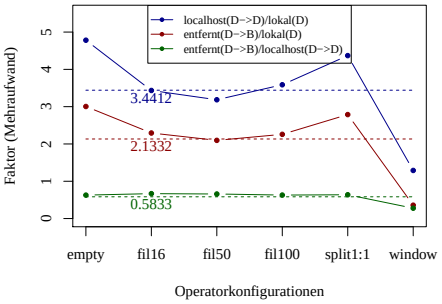
(d) Netzwerkkommunikation A→C.



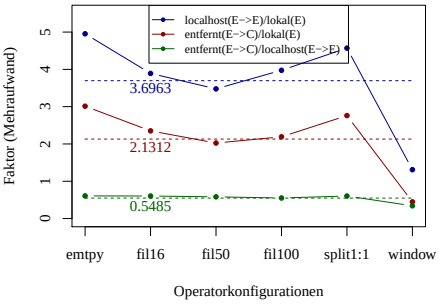
(e) Netzwerkkommunikation B→C.



(f) Netzwerkkommunikation C→A.

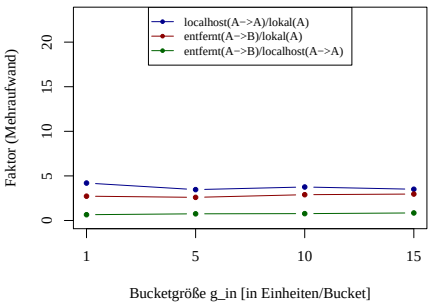


(g) Netzwerkkommunikation D→B.

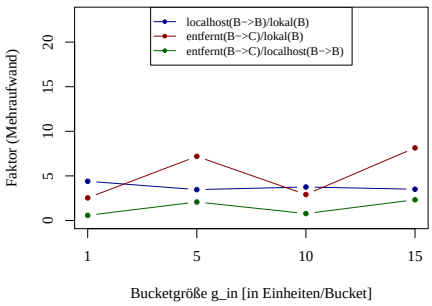


(h) Netzwerkkommunikation E→C.

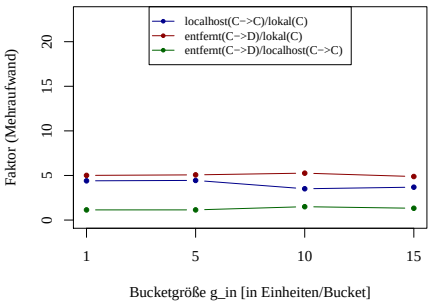
Abbildung A.3: Messungen für Kostenmodell Kapitel 6 (III).



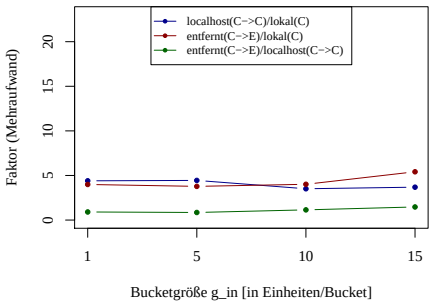
(a) Einfluss Bucketgröße (split) A→B.



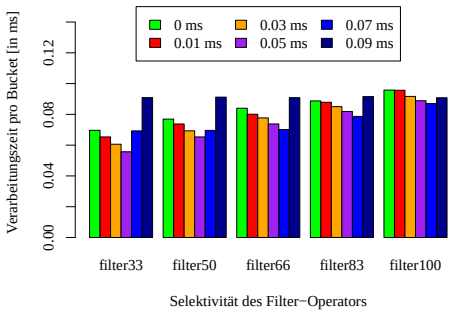
(b) Einfluss Bucketgröße (split) B→C.



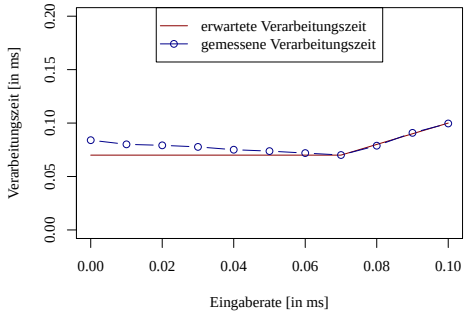
(c) Einfluss Bucketgröße (split) C→D.



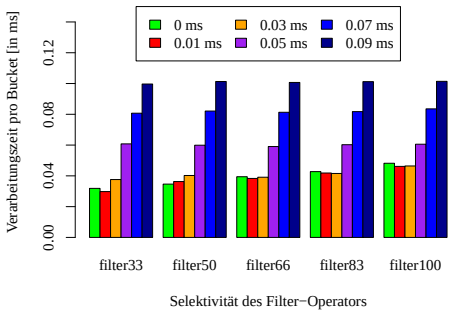
(d) Einfluss Bucketgröße (split) C→E.



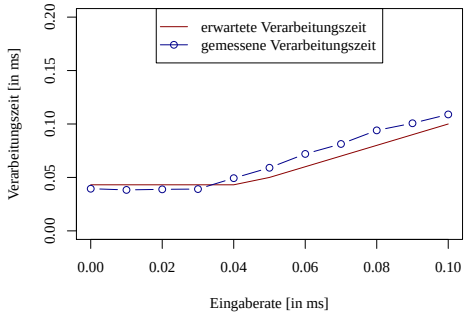
(e) Einfluss Datenrate Knoten A.



(f) Einfluss Datenrate Knoten A.

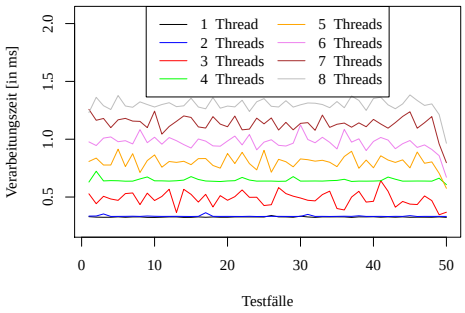


(g) Einfluss Datenrate Knoten C.

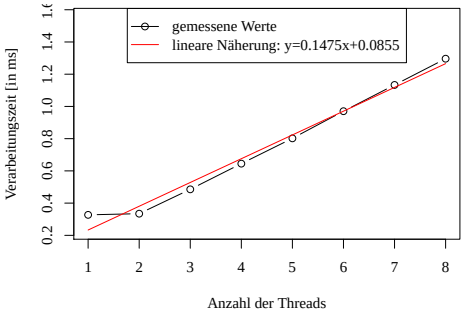


(h) Einfluss Datenrate Knoten C.

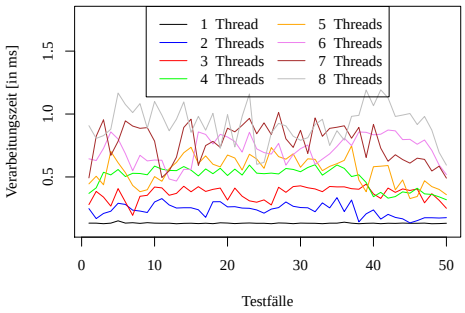
Abbildung A.4: Messungen für Kostenmodell Kapitel 6 (IV).



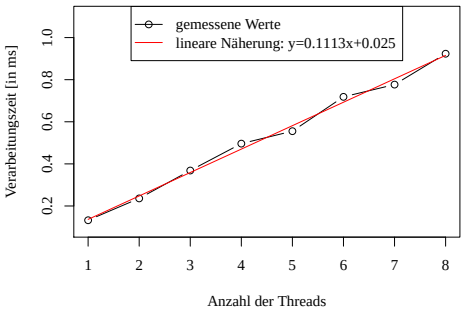
(a) Einfluss Parallelität Knoten A.



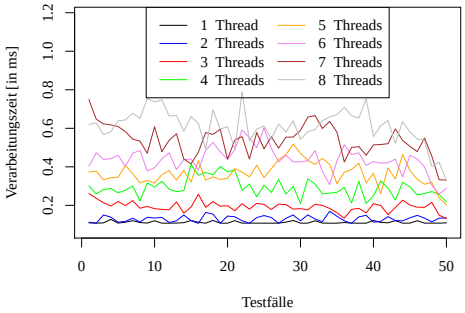
(b) Einfluss Parallelität Knoten A.



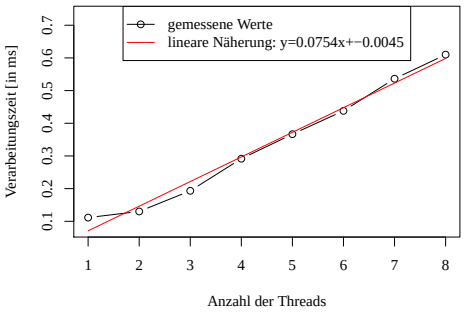
(c) Einfluss Parallelität Knoten B.



(d) Einfluss Parallelität Knoten B.



(e) Einfluss Parallelität Knoten C.



(f) Einfluss Parallelität Knoten C.

Abbildung A.5: Messungen für Kostenmodell Kapitel 6 (V).