



FAKULTÄT FÜR **INFORMATIK**

Ein regelbasierter Dispatcher für serviceorientierte Architekturen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Mag. Hans-Joachim Bobik

Matrikelnummer 8460291

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Univ.Ass. Dr. Uwe Zdun

Wien, 29.11.2008

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Ehrenwörtliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und außer den im Literaturverzeichnis angeführten Werken keine Hilfen in Anspruch genommen habe.

Die Arbeit wurde bisher noch keiner Prüfungskommission vorgelegt.

Wien, im November 2008

Kurzfassung

Serviceorientierte Architekturen haben in den letzten Jahren eine weite Verbreitung erfahren. Sie basieren auf der Annahme, dass komplexe Geschäftsprozesse durch die Komposition von verteilten Services aufgebaut werden können. Dabei kann ein Partner, der seine Anwendungen durch das Einbinden fremder Services komponiert, gleichzeitig Anbieter von Services für andere und somit Empfänger von Service-Requests sein.

Aus Sicht des Empfängers von Service-Requests können in einer solchen Architektur Softwarekomponenten für das Dispatching von Anfragen sinnvoll sein - insbesondere in all jenen Fällen, in welchen mehr als ein Service existiert, welches eine eintreffende Anfrage abarbeiten kann. In diesen Fällen muss die Entscheidung getroffen werden, welches dieser Services für die Bearbeitung einer konkreten Anfrage herangezogen wird.

Im Rahmen dieser Diplomarbeit soll ein Dispatcher entworfen werden, welcher in der Lage ist, eintreffende Service-Requests basierend auf zuvor festgelegten Regeln an das am besten geeignete Service zur Verarbeitung weiterzuleiten sowie die Antwort des gewählten Service entgegenzunehmen und an den ursprünglichen Aufrufer zurückzugeben. Dabei können unter anderem Load-Balancing- oder Performance-Überlegungen für die Entscheidungsfindung eine Rolle spielen. Generell sollen die Entscheidungen auf Basis der Informationen über den Service-Request und über die Eigenschaften der zur Verfügung stehenden Services getroffen werden. Weiters sollen bei der Verarbeitung von Anfragen durch den Dispatcher statistische Informationen ermittelt werden, welche in der Folge ebenfalls Basis für weitere Entscheidungen sein können.

Die Zuordnung von eingehenden Service-Requests zu potenziellen Zielservices sowie die Definition der Regelwerke für die Auswahl der optimalen Services sollen konfigurierbar implementiert werden. Für den Test der Applikation sollen Programme geschaffen werden, mit denen die Verarbeitung simuliert sowie Aussagen zur Performance getroffen werden können.

Abstract

Service-oriented architectures have become an important architectural style in software industry in the last couple of years. One of the basic assumptions is that complex business processes can be constructed by composing distributed services. A partner who composes his applications by embedding external services can in such a scenario himself be provider of services to others and hence be receiver of service requests.

From the viewpoint of a receiver of service requests it can be helpful to have software components available which are able to dispatch these service requests - this will be particularly helpful in situations where more than one client service is able to process an incoming request. In this case it must be decided which of the client services should be invoked.

The intent of this diploma thesis is to develop a dispatcher which is able to route incoming service requests, based on predefined rules, to the most suitable client service. The dispatcher should also be able to receive the response of the invoked client service and route it back to the caller. Load balancing or performance issues can play a role when selecting the client service. The decisions made by the dispatcher will be based on information about the incoming request as well as information about the available client services. Additionally, the dispatcher shall collect statistic data during the processing of requests which can later also be used in the decision making process.

The mapping of incoming service requests to applicable client services as well as the definition of the rules for selecting the optimal service are to be implemented in a configurable way. For testing the component an environment has to be created where the dispatching process can be simulated and information about its performance can be obtained.

Inhaltsverzeichnis

INHALTSVERZEICHNIS	4
1 EINLEITUNG	6
1.1 DEFINITIONEN	7
2 ANFORDERUNGEN UND ABGRENZUNG	10
2.1 ZU IMPLEMENTIERENDE ANFORDERUNGEN	10
2.2 ABGRENZUNG	12
2.3 IMPLEMENTIERUNG	14
3 SYSTEMARCHITEKTUR	16
3.1 KONFIGURATION UND STATISTIK	16
3.2 LAUFZEIT	22
4 DISPATCHER KONFIGURATION UND STATISTIKEN	28
4.1 INCOMING REQUESTS	28
4.2 CLIENT SERVICES	30
4.3 RULE ENGINES	36
4.4 RULE CONFIGURATION	38
4.5 STATISTICS CLIENT	45
5 DISPATCHER LAUFZEIT	47
5.1 SYNCHRONE VERARBEITUNG	47
5.2 ASYNCHRONE VERARBEITUNG	48
5.3 AUSWAHL DER CLIENT SERVICES	49
5.4 TRACING AM APPLIKATIONSSERVER	51
6 REGELDEFINITION	52
6.1 REGELDEFINITION DURCH RULE ENGINES	52
6.2 REGELDEFINITION MIT FRAG	55
6.3 DATE/TIME MANIPULATIONEN MITTELS <i>RBDTIME</i>	56
6.4 ZUGRIFF AUF CLIENT SERVICES MITTELS <i>RBDSERVICE</i>	59
6.5 ZUGRIFF AUF INCOMING REQUESTS MITTELS <i>RBDINCOMING</i>	69
6.6 SPEZIFIKATION DES FRAG-ERGEBNISSES MITTELS <i>RBDRESULT</i>	78
7 DATENBANKSCHEMA	82
7.1 INCOMING REQUESTS	83
7.2 CLIENT SERVICE	84
7.3 CLIENT SERVICE GROUP	85
7.4 REQUEST SERVICE LINK	86
7.5 SERVICE CALLBACK	87
7.6 RULE ENGINE	89
7.7 RULE SCRIPT	90
7.8 SCRIPT TEMPLATE	91
7.9 RUNTIME STATISTICS	92
8 PROJEKTSTRUKTUR UND DEPLOYMENT	94
8.1 PROJEKTSTRUKTUR	94
8.2 DEPLOYMENT	98
9 TESTUMGEBUNG	99

Inhaltsverzeichnis

9.1	MACROFLOW ENGINE SIMULATOR	99
9.2	INCOMING ADAPTER	100
9.3	OUTGOING ADAPTER	101
9.4	MICROFLOW ENGINE	102
10	VERGLEICH MIT BESTEHENDEN APPLIKATIONEN	103
10.1	IBM WEBSHERE	103
10.2	MULE	105
10.3	FUSE MEDIATION ROUTER	109
10.4	MÖGLICHE ERWEITERUNGEN DER DISPATCHER FUNKTIONALITÄT	110
11	ABBILDUNGEN UND TABELLEN	112
11.1	ABBILDUNGSVERZEICHNIS	112
11.2	TABELLENVERZEICHNIS	112
12	LITERATURVERZEICHNIS	115

1 Einleitung

Serviceorientierte Architekturen haben in den letzten Jahren eine große Verbreitung erfahren. Das Konzept geht davon aus, dass sich komplexe Geschäftsprozesse durch das Einbinden von verteilten Services komponieren lassen.

Eine solche Komposition ist vorstellbar, solange man sich in einem technischen Kontext geringer Komplexität bewegt. IT-Experten haben im Allgemeinen kaum Probleme, fremde Services für die eigenen Applikationen nutzbar zu machen (unter der Annahme, dass kompliziertere Themen wie Security-Überlegungen, QoS-Vereinbarungen usw. nur eine untergeordnete Rolle spielen). Bei komplexeren Anforderungen - insbesondere dort, wo unternehmensweite Geschäftsprozesse durch Business Process Engines abgearbeitet werden und die technischen Services nur einen kleinen Bestandteil der Gesamtarchitektur darstellen - stößt man aber in der Praxis schnell auf organisatorische Probleme.

In Unternehmen mit komplexen Geschäftsprozessen werden fachliche Prozessabläufe und IT-Services normalerweise von unterschiedlichen Personen oder Abteilungen verantwortet. Dabei setzen beide Partner ganz unterschiedliche Schwerpunkte. Geschäftsprozess-Spezialisten haben die Aufgaben, die Unternehmensabläufe zu definieren, die Ressourcenzuordnung zu den einzelnen Prozessschritten zu optimieren, Prozesse zu simulieren und anhand der Ergebnisse weitere Optimierungen vorzunehmen. Einige dieser Prozessschritte werden den Aufruf technischer Services nötig machen, andere können beispielsweise manuelle Tätigkeiten einzelner Abteilungen beinhalten.

IT-Spezialisten sind dafür verantwortlich, die benötigten technischen Services zur Verfügung zu stellen. Das kann die Implementierung im eigenen Unternehmen einschließen, unterschiedliche Sourcing-Strategien nötig machen oder auch die Einbindung externer Services bedingen. Das Hauptaugenmerk von IT-Spezialisten liegt in der kostenoptimalen Implementierung bzw. Integration dieser Services. Die umfassende Gesamtsicht auf die unternehmensweiten Geschäftsprozesse ist im Allgemeinen den Prozess-Spezialisten vorbehalten.

Wie [Hentrich et al. 2007] ausführen, sind Prozess-Spezialisten nicht dazu ausgebildet, technische Details von IT-Services zu verstehen. Umgekehrt ist es Technikern nicht zumutbar, sich zusätzlich zu den technischen Services auch noch um die unternehmensweite fachliche Definition von Geschäftsprozessen zu kümmern.

Es ist daher sinnvoll, das Konzept der "Separation of Concerns" auch auf dem Feld der Geschäftsprozessmodellierung umzusetzen, wobei das nahtlose Zusammenspiel der genannten Tätigkeiten durch eine eigene Softwarekomponente garantiert werden soll - den *regelbasierten Dispatcher*, welcher im Rahmen dieser Arbeit erstellt werden soll.

Dabei werden wir noch folgende zusätzliche Dimension in unsere Überlegungen einbeziehen:

Wie ändern sich die Anforderungen, wenn die Zuordnung von fachlichem Prozessschritt und ausführendem IT-Service nicht 1:1 ist, sondern wenn ein Schritt in einem Geschäftsprozess von einer größeren Anzahl von fachlich gleichartigen IT-Services ausgeführt werden kann? In diesen Fällen ist es nötig, sowohl die möglichen Zuordnungen zwischen Prozessschritt und IT-Services zu definieren, als auch die Regeln, aufgrund welcher eines der zur Verfügung stehenden Services ausgewählt und für die Verarbeitung herangezogen werden soll.

1.1 Definitionen

Die folgende Grafik wurde übernommen aus [Hentrich et al. 2007] und zeigt das Zusammenspiel der Komponenten in der dort definierten *Process-Based Integration Architecture*. Dabei treten *Macroflow Engines* über *Process Integration Adapter* mit einem *Rule Based Dispatcher* in Verbindung. Dieser nimmt die einlangenden Anfragen entgegen und entscheidet aufgrund eines internen Regelwerkes, an welche der bekannten *Microflow Engines* die Anfrage weitergeleitet werden soll.

Anmerkung: Bei [Hentrich et al. 2007] nicht explizit erwähnt sind die Process Integration Adapter auf Seiten der Microflow Engines. Diese sind aber sinnvoll, um die Interfaces dieser Microflow Engines vom Dispatcher abzuschirmen.

Der im Zuge dieser Arbeit entwickelte regelbasierte Dispatcher fügt sich in diese Architektur ein, wobei der Dispatcher auf der Request-Seite nur mit den Process Integration Adaptern (in unserem Kontext '*Incoming Adapter*') und auf der Service-Seite nur mit '*Outgoing Adapter*' kommuniziert. Diese Adapter sind in Abbildung 1 jeweils orange dargestellt.

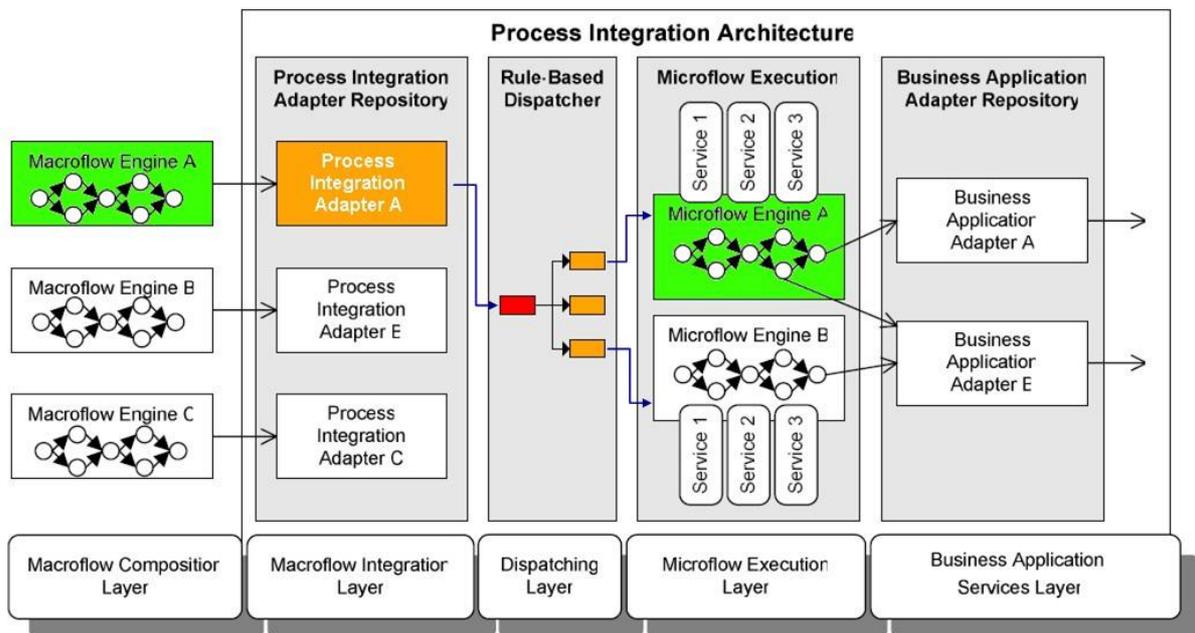


Abbildung 1 - Process Integration Architecture

Bevor die Architektur und Funktionsweise dieses Dispatchers im Detail vorgestellt werden, sollen hier noch die wichtigsten im Dokument verwendeten Komponenten und Ausdrücke definiert und überblicksartig dargestellt werden.

Komponente	Beschreibung
Client Service	Sind jene Services, welche eine einlangende Anfrage abarbeiten können. Sie werden im Zuge der Konfiguration den Incoming Requests zugeordnet, der Dispatcher wählt für jede einlangende Anfrage das am besten passende Service aus.
Client Service Binding	Bezeichnet die Zuordnung von Client Services zu Incoming Requests, d.h. die Festlegung, welche Client Services einen spezifischen Incoming Request verarbeiten können.
Client Service Gruppe	Client Services können zu Gruppen zusammengefasst werden. Die Client Service Gruppe eines Service ist eines der möglichen Kriterien für die Rule Engines.
Incoming Adapter	Entspricht dem in Abbildung 1 (orange) dargestellten Process Integration Adapter und stellt dem Dispatcher ein einheitliches Interface für die Request-Seite zur Verfügung. Der Incoming Adapter nimmt die Anfragen einer externen Anwendung (bzw. in unserer Testumgebung des <i>Macroflow Engine Simulators</i>) entgegen, konvertiert sie in ein für den Dispatcher verständliches, einheitliches Format und leitet sie an den Dispatcher weiter. Der Adapter nimmt im Fall von asynchroner Verarbeitung den Callback entgegen und konvertiert die erhaltenen Daten in ein für die externe Anwendung verständliches Format zurück.
Incoming Request	ein einlangender Auftrag zur Ausführung eines Service
(Incoming) Request Gruppe	Incoming Requests können zu Gruppen zusammengefasst werden - beispielsweise, um die gemeinsame Herkunft (z.B. aus derselben Business Process Engine) zu dokumentieren. Die Request Gruppe ist ein mögliches Kriterium für die Rule Engines - beispielsweise können alle Anfragen einer bestimmten Gruppe an ein bestimmtes Client Service weitergeleitet werden.
(regelbasierter) Dispatcher	Der regelbasierte Dispatcher ist das Kernstück dieser Arbeit; der Dispatcher nimmt Anfragen vom Incoming Adapter entgegen, verarbeitet sie und stößt den entsprechenden Outgoing Adapter an. Bei synchroner Verarbeitung oder im Fall von <i>managed Callbacks</i> nimmt er auch die Ergebnisse der Verarbeitung entgegen und leitet diese an den Aufrufer weiter.
Domain Specific Language, DSL	Laut [Wiki-DSL] "eine formale Sprache, die speziell für ein bestimmtes Problemfeld (die Domäne) entworfen und implementiert wird". In diesem Projekt wird die Sprache FRAG (siehe [Zdun 2006]) um einige Elemente erweitert, um die besonderen Anforderungen des Dispatchers abdecken zu können.
FRAG, FRAG Rule Engine	FRAG ist eine von Uwe Zdun entwickelte, objektorientierte Erweiterung der Sprache TCL. FRAG wird in diesem Projekt zur Definition der Regeln verwendet, die die Basis für die Entscheidungen des Dispatcher bilden. Die FRAG Rule Engine ist jene Komponente, welche die für den Dispatcher zusätzlich

	benötigten FRAG Objekte definiert, die Regeln auswertet und die Ergebnisse an den Dispatcher liefert. Siehe [Zdun 2006] für eine detaillierte Beschreibung von FRAG.
Macroflow Engine Simulator	Der Macroflow Engine Simulator ist eine Komponente der Testumgebung, welche die in Abbildung 1 (grün) dargestellte Rolle der Macroflow Engines übernimmt. Der Simulator ermöglicht es, die Funktionsweise des Dispatchers umfassend zu testen, ohne schwergewichtige Business Rule Engines zu verwenden. Siehe Kapitel 9.1
managed Callback	Ein Callback heißt "managed", wenn er nicht vom Client Service bzw. dessen Outgoing Adapter direkt an den Requester adressiert wird, sondern vom Dispatcher abgefangen wird, um statistische Informationen über die Verarbeitung zu sammeln.
Microflow Engine	Microflow Engines sind Komponenten der Testumgebung und simulieren die Client Services. Sie ermöglichen es, sowohl erfolgreiche als auch nicht erfolgreiche Aufrufe von Client Services zu testen. Siehe Kapitel 9.4.
Outgoing Adapter	Das sind jene Komponenten, welche vom Dispatcher angestoßen werden, um ein Client Service zu starten. Sie bieten dem Dispatcher ein standardisiertes Interface und kapseln die Spezifika der jeweiligen Services.
(externe) Rule Engine, Rule Service	Eine Enterprise Java Bean oder ein Webservice, welche(s) die Regeln zur Auswahl des passenden Client Service für einen Incoming Request enthält. Rule Engines können anstelle von oder zusätzlich zu FRAG Rule Scripts definiert werden.
Rule Script, FRAG Rule Script	eine in der Sprache FRAG definierte Menge von Regeln, anhand der von der FRAG Rule Engine das zur Verarbeitung eines Incoming Request am besten geeignete Client Service bestimmt werden kann.
Script Template	Eine Vorlage für die Definition von FRAG Rule Scripts. Öfters verwendete Teile solcher Scripts können als Template abgelegt und dann als Basis für die Definition neuer FRAG Rule Scripts verwendet werden.

Tabelle 1 - Beschreibung der verwendeten Ausdrücke

In der Folge werden Anforderungen an den Dispatcher sowie Abgrenzungen definiert und die genannten Komponenten sowie ihr Zusammenspiel im Detail beschrieben. Weitere programmtechnische Informationen sind in [Dispatcher-Project] als javadoc-Dokumente verfügbar.

2 Anforderungen und Abgrenzung

Bevor die Arbeitsweise des Dispatchers erklärt wird, welcher im Zuge dieser Arbeit erstellt wurde, sollen in Abschnitt 2.1 die Anforderungen an diese Komponente überblicksartig dargestellt werden. Zur Abgrenzung werden in Kapitel 2.2 jene Punkte angeführt, welche bewusst nicht in das Design des Dispatchers einfließen sollten.

2.1 Zu implementierende Anforderungen

Der Rule Based Dispatcher soll auf ähnlichen Konzepten wie der in [Hohpe et al. 2004] dargestellte Message Router basieren (anstelle von *InQueue* und *outQueue* treten in unserem Fall der Incoming und Outgoing Adapter). Für den Rule Based Dispatcher werden folgende Anforderungen definiert.

Anforderung 1 - Separation of Concerns

Der Dispatcher stellt die Schnittstelle zwischen den fachlichen Geschäftsprozess-Anforderungen und der technischen Implementierung dar. Er muss so einfach zu bedienen sein, dass ein Integrator, der weder ausgebildeter Prozessspezialist noch Techniker ist, die Zuordnung von Prozessschritten zu Services mit geringem Aufwand durchführen kann.

Anforderung 2 - Konfiguration und grafische Benutzeroberfläche

Die wesentlichen Objekte und ihr Zusammenspiel sollen nicht hardcodiert werden, was Anforderung 1 widersprechen würde. Stattdessen muss die Konfiguration mittels eines grafischen User Interface einfach und schnell möglich sein. Das Ziel ist, Client Services durch die Verbindung mit einlangenden Anfragen für die Bearbeitung "freizuschalten" und für den Dispatcher nutzbar zu machen. Daraus folgt Anforderung 3:

Anforderung 3 - Persistenz

Alle vorgenommenen Konfigurationen sollen persistent abgelegt werden. In unserem Fall soll die Persistenzschicht durch eine MySQL-Datenbank realisiert werden, allerdings ist ein Datenbank-Abstraktionslayer in einer Form zu implementieren, die auch die Einbindung anderer Datenbanken zulässt. Überdies soll die Implementierung von Cachingmechanismen für die Datenbankzugriffe leicht hinzugefügt werden können, ohne Eingriffe in die übrigen Komponenten des Dispatchers vornehmen zu müssen.

Anforderung 4 - Erweiterbarkeit um neuen Client Services

Neue Client Services sollen ohne Programmieraufwand eingebunden werden können. Daraus folgt, dass ein allgemein gültiges Interface definiert werden muss, dem diese Services genügen müssen, um vom Dispatcher erkannt und verwendet werden zu können.

Anforderung 5 - DSL für Regeln

Der Dispatcher benötigt zur Ermittlung des am besten passenden Client Service Logik, welche in Form von Regeln in einer Domain Specific Language (DSL) definiert werden kann. Eine solche DSL ist zu entwerfen bzw. ist eine vorhandene Sprache so zu adaptieren, dass die für Auswertungen nötigen Objekte (Informationen über Incoming Requests, Client Services sowie Runtime Statistics) darin verwendet werden können.

Anforderungen und Abgrenzung

Siehe auch [Hohpe et al. 2004] für die Darstellung des *Content-Based Router* Patterns.

Anforderung 6 - Validierung von Regeln

Parallel zur Möglichkeit der Definition von Regeln mittels DSL muss eine Möglichkeit vorgesehen werden, diese Regeln zum Zeitpunkt der Definition zu validieren. Da die Verarbeitung später auf einem entfernten Applikationsserver durchgeführt werden wird, ist das Debuggen dieser Regeln zur Laufzeit sehr aufwändig oder sogar unmöglich. Für die Simulation der Regeln ist es auch nötig, die möglichen Parameter der Incoming Requests vorgeben zu können, falls diese in den Regeldefinitionen eine Rolle spielen.

Anforderung 7 - Rule Engines

Eine performantere Alternative zur Definition von Regeln mittels einer DSL stellt die Codierung dieser Regeln in eigenen Modulen dar. Diese Module werden als externe Rule Engines bezeichnet, sollen "hot-deployable" sein (siehe Anforderung 10) und ohne Programmieraufwand eingebunden werden können. Analog zu Anforderung 5 sollen Regeln auf Basis der Daten der einlangenden Anfragen, der Informationen über die verfügbaren Client Services sowie der statistischen Informationen des Dispatchers definiert werden können.

Anforderung 8 - statistische Informationen

Die wichtigsten statistischen Informationen, welche der Dispatcher seinen Entscheidungen zugrunde legt, sollen jederzeit - auch parallel zur Dispatcher-Verarbeitung - abfragbar sein. Dazu ist eine eigene Applikation zu schaffen, in welcher diese Informationen visualisiert werden können.

Anforderung 9 - Suspend/Resume

Zur Laufzeit soll es möglich sein, die Verarbeitung von einlangenden Anfragen "auf Knopfdruck" zu unterbinden und ebenso wieder freizuschalten. Analog dazu sollen Client Services als "für die Verarbeitung gesperrt" gekennzeichnet werden können, ohne dass über das Konfigurationssystem die Client Service Bindings des betroffenen Client Service zu allen Incoming Requests gelöst werden müssen.

Anforderung 10 - Environment und Hot Deployment

Die Realisierung des Dispatchers soll mittels J2EE in Form von Enterprise Java Beans erfolgen, welche auf einem JBoss-Applikationsserver laufen. Die Anforderung 7 soll mittels Hot Deployment gelöst werden, d.h. neue Rule Engines können jederzeit hinzugefügt werden, ohne den Dispatcher oder den Applikationsserver unterbrechen zu müssen. Rule Engines und Client Services sollen ebenfalls mittels Enterprise Java Beans implementiert werden. Die Verwendung von WebServices ist anzudenken.

2.2 Abgrenzung

Bei der Erstellung des regelbasierten Dispatchers werden einige Themen bewusst aus den Designvorgaben ausgeklammert. Die wesentlichen Punkte sind hier angeführt.

Abgrenzung 1 - Workflows

Es liegt nicht in der Verantwortung des Dispatchers, Workflows abzubilden. Viele aktuelle Middleware-Systeme bieten diese Möglichkeit, beispielsweise werden in IBM WebSphere diese Regeln in *message flow nodes* abgebildet, welche ihrerseits in Workflows eingebunden sind. Siehe [WebSphere P1] und [WebSphere P3a] für eine Übersicht über mögliche Workflow Features.

Abgrenzung 2 - einfache Struktur, Queueing

Der Dispatcher soll in dieser ersten Version vollkommen zustandslos implementiert werden und dient ausschließlich zum Verteilen von Aufträgen nach den definierten Regeln. Da der Dispatcher immer in ein komplexeres Umfeld eingebunden sein wird, kann vorausgesetzt werden, dass Queueing-Funktionalitäten in den umliegenden Komponenten und nicht im Dispatcher selbst implementiert werden. Beispielsweise besitzen die in Abgrenzung 1 angesprochenen *message flow nodes* in IBM WebSphere *Input-Terminals* und *Output-Terminals*, welche das Queueing übernehmen, der Node selbst aber ist von diesen Aufgaben befreit.

Abgrenzung 3 - Multiple Endpoints

Der Dispatcher soll einen Incoming Request immer an genau ein Client Service zur Durchführung übermitteln und *Exception Based Routing* ([Hohpe et al. 2004]) auf folgende Weise unterstützen: nur wenn der gewählte Client die Anfrage aufgrund eines technischen (nicht fachlichen!) Fehlers nicht annehmen kann, soll der nächste mögliche Client für die Verarbeitung herangezogen werden.

Alternativ dazu wäre es möglich, den Request parallel an mehrere Client Services weiterzuleiten (siehe auch das Pattern *Recipient List* in [Hohpe et al. 2004]). Dann müssten aber die Antworten dieser Services durch ein aufwändiges Verfahren gemanaged werden. Die als erstes einlangende Antwort würde an den Requester übermittelt, die übrigen Services müssten informiert werden, dass sie die Verarbeitung abbrechen können. Bei asynchroner Verarbeitung ergibt sich ein hoher Aufwand in der Callback-Komponente. Bei synchroner Verarbeitung in der Prozessor-Komponente des Dispatchers, da hier ein Multithreading-Verfahren unterstützt werden müsste.

Abgrenzung 4 - Request splitting und result aggregation

In Fortführung von Abgrenzung 3 sollen die einlangenden Requests vom Dispatcher nicht manipuliert werden - beispielsweise muss das in Messaging-Systemen verbreitete Splitten von Anfragen und Weiterleiten der Einzelteile an verschiedene Empfänger sowie die Aggregation der Resultate nicht unterstützt werden. Falls nötig, müssen diese Tätigkeiten in einem (komplexen) Incoming Adapter durchgeführt und der Dispatcher für jeden Teil der Nachricht eigens aufgerufen werden.

Abgrenzung 5 - Quality-of-Service

Der Dispatcher sollte im Endausbau in der Lage sein, Quality-of-Service Informationen der bekannten Client Services zu lesen und seine Entscheidungen auch anhand dieser

Anforderungen und Abgrenzung

Informationen treffen zu können. Im aktuellen Datenmodell ist zwar eine Entität für die Aufnahme solcher Informationen vorzusehen, die (zyklische) Aktualisierung dieser Informationen sowie deren Abfrage zum Zeitpunkt der Regelauswertung werden vorläufig aber nicht implementiert. Für eine weitergehende Diskussion dieser Möglichkeiten siehe die Ausführungen zum *Dynamic Router Pattern* in [Hohpe et al. 2004].

Abgrenzung 6 - Datenkonvertierung

Der Dispatcher kann bei der Auswertung der Regeln auf Daten des Incoming Request sowie der zugeordneten Client Services zugreifen. Um diese Zugriffe zu ermöglichen, müssen die Daten in einem für den Dispatcher verständlichen Format vorliegen. Diese Konvertierung wird an die beiden "Kommunikationspartner" des Dispatchers übertragen - den Incoming Adapter des Request sowie den Outgoing Adapter des Client Service. Diese kennen sowohl ihren Aufrufer bzw. ihr Client Service als auch die Struktur des Dispatcher-Interface und können diese Konvertierung somit leicht durchführen. Würde der Dispatcher diese Aufgabe übernehmen, müsste für jede mögliche Interaktion ein komplexes Datenmapping definiert werden. In [WebSphere P4] ist ein solches alternatives Verfahren dargestellt.

Abgrenzung 7 - Caching

Der Dispatcher wird in der ersten Version als Menge von zustandslosen Services definiert. Das hat gegenüber einer Implementierung als zustandsbehaftete Komponente den Vorteil der geringeren Komplexität. Ein Nachteil liegt in der verminderten Performance, da das Caching von Lesezugriffen auf diese Weise nicht möglich ist. Caching kann aber durch Einführung einer Service Bean (siehe [JBoss Ext 2008]) leicht implementiert werden, der deutliche Performancegewinn wird erkaufte durch eine geringfügig höhere Komplexität der Applikation.

Abgrenzung 8 - Sicherheit

Der Dispatcher muss keine Sicherheitsmechanismen für die Kommunikation zu Incoming Adapter, Outgoing Adapter und externen Rule Engines implementieren. Es wird davon ausgegangen, dass alle diese Komponenten in Umgebungen laufen, zwischen denen externe Maßnahmen den gewünschten Sicherheitslevel garantieren.

Einige mögliche alternative Designentscheidungen und Erweiterungen für den Dispatcher werden in Kapitel 10.4 dargestellt.

2.3 Implementierung

In Abbildung 2 ist eine grobe Übersicht über die Zielarchitektur des Dispatchers in Form eines Klassendiagrammes zu sehen. Eine mögliche Deployment-Struktur wird in Kapitel 8.2 beschrieben.

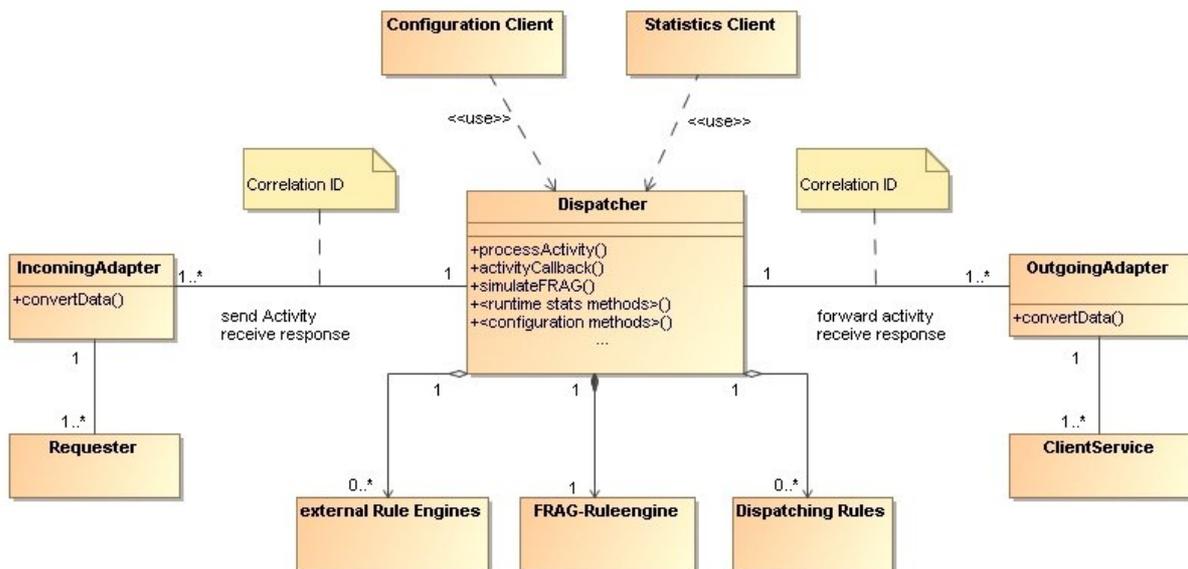


Abbildung 2 - Dispatcher Übersicht

Der Dispatcher soll seinen drei Typen von Clients - den Requestern, dem Configuration Client und dem Statistics Client - unterschiedliche Funktionalitäten zur Verfügung stellen:

Dispatching Services für die Requester

Der Dispatcher verteilt Anfragen der Requester an die Client Services und liefert deren Ergebnisse an die Requester zurück. Dabei steht er nicht direkt mit diesen in Verbindung, sondern ist von den Requestern durch *Incoming Adapter* und von den Client Services durch *Outgoing Adapter* abgeschirmt. Diese Adapter übernehmen die Datenkonvertierung zwischen externen Systemen und dem Dispatcher.

Ein Requester stellt eine Anfrage an seinen Incoming Adapter, dieser konvertiert die Daten und leitet die Anfrage - versehen mit einer eindeutigen Correlation-ID ([Hohpe et al. 2004]) - an den Dispatcher weiter. Der Dispatcher identifiziert den Incoming Request in der Datenbank, ermittelt die Liste der möglichen Client Services und die für diesen Request definierten (in Form eines FRAG Rule Scripts vorliegenden) Dispatching Regeln sowie eine eventuell vorhandene externe Rule Engine.

Die Daten des Incoming Request, der verfügbaren Client Services sowie die vorliegenden Statistikdaten aus vorangegangenen Aufrufen werden an die FRAG Rule Engine und/oder die externe Rule Engine übergeben. Diese liefern als Ergebnis eine priorisierte Liste mit Client Services, welche in der Folge vom Dispatcher über deren Outgoing Adapter - versehen mit eigens generierten Correlation-IDs - angestoßen werden.

Anforderungen und Abgrenzung

Die Ergebnisse der Client Services werden an den Dispatcher zurückgeliefert (im Fall von synchroner Verarbeitung bzw. managed Callbacks), dieser legt statistische Daten des Aufrufes in der Datenbank ab und leitet die Antwort unter Verwendung der ursprünglichen Correlation-ID - an den Incoming Adapter des Requesters weiter. Dieser konvertiert die erhaltenen Daten zurück in das für den Requester verständliche Format und retourniert sie an diesen.

Diese Abläufe sind in Kapitel 5, die beteiligten Komponenten und ihre Services in Kapitel 3.2 beschrieben.

Configuration Services für den Configuration Client

Um seine Verarbeitungen wie oben beschrieben durchführen zu können, ist der Dispatcher darauf angewiesen, dass die Informationen über Incoming Requests, verfügbare Client Services und Rule Engines, sowie die zur Entscheidung nötigen Regeln in der Datenbank vorhanden sind. Die Aufgabe, diese zu pflegen übernimmt der Configuration Client.

Die genaue Funktionsweise dieser Komponente ist in Kapitel 4 beschrieben. Der Configuration Client nutzt verschiedene, vom Dispatcher bereitgestellte, Services zum Lesen, Speichern und Ändern von Konfigurationsdaten (siehe Kapitel 3.1.2). Darüber hinaus stellt der Dispatcher eine Methode zur Verfügung, mit der FRAG Rule Scripts zum Zeitpunkt der Konfiguration validiert werden können (Siehe Kapitel 3.1.3).

Statistic Services für den Statistics Client

Der Statistics Client hat einerseits die Aufgabe, die vom Dispatcher angelegten statistischen Daten aus der Datenbank zu lesen, in einem grafischen User Interface anzuzeigen und dem Benutzer zu ermöglichen, die angezeigten Informationen über Incoming Requests und Client Services auf ihren Ausgangswert zurückzusetzen.

Andererseits können über diese Komponente auch Incoming Requests und Client Services temporär von der Verarbeitung durch den Dispatcher ausgenommen werden (suspend) bzw. wieder in Produktion gestellt werden (resume).

Für alle diese Verarbeitungen benutzt der Statistics Client die in Kapitel 3.2.7 beschriebenen Services des Dispatchers, die genaue Arbeitsweise ist in Kapitel 4.5 beschrieben.

3 Systemarchitektur

Basierend auf den in Kapitel 2 definierten Anforderungen und Abgrenzungen, wurde der Dispatcher unter J2EE als Menge von Enterprise Java Beans implementiert, welche auf einem JBOSS 4.0-Applikationsserver ausgeführt werden. Er bietet alle Funktionalitäten, um zu einlangenden Anfragen das am besten geeignete Client Service zu ermitteln und die Anfragen an dieses weiterzuleiten. Darüber hinaus stellt der Dispatcher alle für den Configuration Client und den Statistics Client benötigten Methoden zur Verfügung, sodass diese Komponenten sich insbesondere um die Speicherung der Daten am Server nicht zu kümmern brauchen. Im Unterschied zu der in [Hentrich et al. 2007] vorgeschlagenen Implementierung als Configurable Component im Sinne von [Schmidt et al. 2000], besitzt der Dispatcher aber nicht die Möglichkeit initialisiert oder finalisiert zu werden, da es keine permanent laufende Instanz der Komponente gibt.

Die Methoden zur Initialisierung sowie für Suspend und Resume werden aber auf Ebene der einzelnen Client Services unterstützt. Dabei wird die Initialisierung durch die Zuordnung eines Client Service zu einem Incoming Request abgebildet, das Finalisieren durch die Lösung dieser Bindung.

Die Softwaretopologie der Anwendung wird in Kapitel 8 näher beschrieben. Im weiteren Verlauf dieses Kapitels werden nun alle im Zuge der Diplomarbeit erstellten Komponenten des Dispatchers mit ihren Services im Detail vorgestellt.

3.1 Konfiguration und Statistik

Die Komponentenarchitektur für die Konfigurationsfunktionalität ist in Abbildung 3 dargestellt. Als DBMS kommen in unserer Implementierung MySQL 5.0, als grafisches User Interface für das DBMS die MySQL 5.0 GUI Tools zur Anwendung. Deren Funktionalität und Verwendung ist in [MySQL 2008] nachzulesen und wird hier nicht näher ausgeführt.

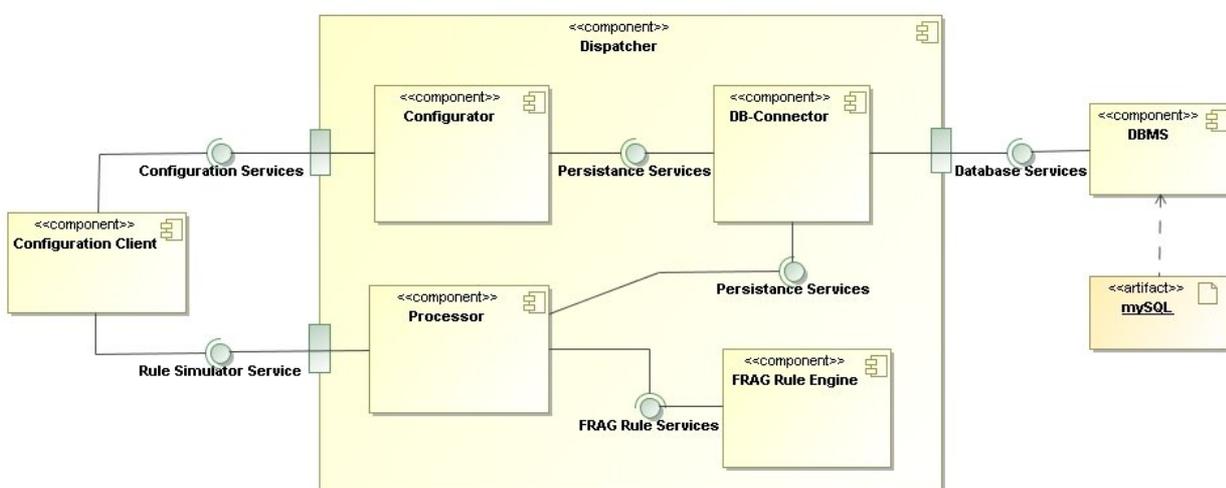


Abbildung 3 - Komponenten Konfiguration

3.1.1 Configuration Client

Diese Java-Applikation dient einerseits dazu, die verarbeitbaren Requests sowie die zur Verfügung stehenden Client-Services und externen Rule Engines festzulegen. Darüber hinaus werden die Zuordnungen von Requests zu Services (die Client Service Bindings) vorgenommen, sowie die Regeln definiert, anhand der die zur Laufzeit tatsächlich zu verwendenden Client Service bestimmt werden.

Die Möglichkeiten der Datenpflege sowie die Kommunikationsabläufe werden in Kapitel 4 genau beschrieben.

3.1.2 Die Configurator Komponente

Folgende Services bietet die Configurator Komponente des Dispatchers dem Configuration Client an (siehe *Configuration Services* in Abbildung 3).

Interface-Methode	Beschreibung
<code>getIncomingRequests()</code>	Liefert eine Liste aller aktuell konfigurierten Incoming Request-Objekte. Für jeden dieser Requests wird auch die Liste der aktuell vorhandenen Client Service Bindings geliefert, d.h. die Liste jener Client Services, die die Verarbeitung für diesen Request übernehmen können.
<code>setIncomingRequests()</code>	Die im Zuge der Konfiguration vorgenommenen Änderungen werden in der Datenbank abgelegt. Diese Änderungen können sein: Hinzufügen oder Löschen eines Incoming Requests oder Ändern seiner Beschreibung, Hinzufügen, Löschen oder Ändern der Reihenfolge von Client Service Bindings für Requests. Die IDs von Request-Gruppe und Request können nicht geändert werden.
<code>getClientServiceGroups()</code>	Liefert die Liste aller vorhandenen Client Service Gruppen. Zu jeder Gruppe wird auch die Liste der zugehörigen Client Services ermittelt.
<code>setClientServiceGroups()</code>	Die im Zuge der Konfiguration vorgenommenen Änderungen werden in der Datenbank abgelegt. Diese Änderungen können sein: Hinzufügen oder Löschen einer Client Service Gruppe, Hinzufügen oder Löschen von Client-Services der Gruppe. Das Ändern von Gruppe oder Client Services ist nicht möglich.
<code>getRuleEngines()</code>	Liefert die Liste der bekannten Rule Engines.
<code>setRuleEngines()</code>	Die im Zuge der Konfiguration vorgenommenen Änderungen werden in der Datenbank abgelegt. Diese Änderungen können sein: Hinzufügen oder Löschen von externen Rule Engines.
<code>getRuleScript()</code>	Liefert das zu einem Incoming Request definierte FRAG Rule Script bzw. ein Script Template (Templates werden ebenfalls als FRAG Rule Scripts in der Datenbank abgelegt)

Interface-Methode	Beschreibung
<code>setRuleScript()</code>	Legt ein neues FRAG Rule Script oder Script Template in der Datenbank ab oder ändert ein bestehendes Script oder Template.
<code>getTemplates()</code>	Liest die Bezeichnungen aller aktuell definierten Templates
<code>setTemplate()</code>	legt die Bezeichnung eines Script Template in der Datenbank ab
<code>delTemplate()</code>	löscht die Bezeichnung eines Script Template sowie das Template selbst aus der Datenbank
<code>getBeanServices()</code>	Fragt einen Applikationsserver nach allen Enterprise Java Beans, welche ein definiertes Reflection-Interface unterstützen und ermittelt aus den gefundenen Beans die Interface-Methoden.

Tabelle 2 - Configuration Services

Die Configurator-Komponente benötigt ihrerseits die in Tabelle 4 angeführten Services des Persistenzlayers.

3.1.3 Die Processor Komponente

Die Processor Komponente des Dispatchers bietet dem Configuration Client zusätzlich folgendes Service an (siehe *Rule Simulator Service* in Abbildung 3).

Interface-Methode	Beschreibung
<code>simulateFRAG()</code>	Interpretiert ein FRAG Rule Script und liefert das Ergebnis zurück. Siehe Kapitel 6.2 für eine detaillierte Beschreibung der FRAG-Syntax und Kapitel 4 für Konfigurationsbeispiele.

Tabelle 3 - Processing Services für den Configuration Client

Die in Tabelle 2 und Tabelle 3 beschriebenen Services bilden die Basis für die Konfiguration des Dispatchers - siehe Kapitel 4 für die Beschreibung des Configurator-GUI.

3.1.4 Der Persistenzlayer *DB-Connector*

Die Komponente *DB-Connector* übernimmt für die übrigen Komponenten des Dispatchers die Datenhaltung. Dazu werden die J2EE-Entity Beans in `RBDSERVER.SERVER.ENTITIES` verwendet. Für jede Datenbanktabelle existiert eine solche Entity Bean, diese regelt den Zugriff auf die Datenobjekte der Entität.

Im Augenblick sind keine Caching-Mechanismen implementiert, diese können aber sehr leicht - beispielsweise durch die Einführung einer Service Bean ([JBoss Ext 2008]) - hinzugefügt werden.

Folgende Services bietet die Komponente DB-Connector dem Dispatcher Configurator an (siehe *Persistence Services* in Abbildung 3).

Interface-Methode	verwendete Entity-Beans und Beschreibung
<code>cfgGetRequests()</code>	<code>IncomingRequestEBean</code> , <code>RequestServiceEBean</code> Liest alle derzeit bekannten Incoming Requests und deren Client Service Bindings.
<code>cfgAddRequest()</code>	<code>IncomingRequestEBean</code> Fügt einen neuen Incoming Request in die Datenbank ein.
<code>cfgDelRequest()</code>	<code>IncomingRequestEBean</code> Löscht einen Incoming Requests aus der Datenbank. Besitzt der Incoming Request zum Zeitpunkt des Löschversuches noch Client Service Bindings, so ist ein Löschen nicht möglich - die referentielle Integrität wird durch diese Operation gewährleistet.
<code>cfgUpdRequest()</code>	<code>IncomingRequestEBean</code> Ändert die Daten eines Incoming Request in der Datenbank. Schlüsselattribute können nicht verändert werden.
<code>cfgGetClientServiceGroups()</code>	<code>ClientServiceGroupEBean</code> Liest eine über ihre ID spezifizierte Client Service Gruppe aus der Datenbank. Wird keine ID angegeben, so werden alle aktuell konfigurierten Client Service Gruppen ermittelt.
<code>cfgAddClientServiceGroup()</code>	<code>ClientServiceGroupEBean</code> Speichert eine Client Service Gruppe in der Datenbank.
<code>cfgDelClientServiceGroup()</code>	<code>ClientServiceGroupEBean</code> Löscht eine Client Service Gruppe aus der Datenbank.
<code>cfgGetClientServices()</code>	<code>ClientServiceEBean</code> , <code>RequestServiceEBean</code> Liest die Menge Client Services einer Client Service Gruppe. Die Gruppe wird über ihre ID spezifiziert. Zu jedem gefundenen Client Service wird auch die Information geliefert, ob das Service im Augenblick einem der Incoming Requests zugeordnet ist (d.h. ein Client Service Binding zu einem Incoming Request existiert).
<code>cfgAddClientService()</code>	<code>ClientServiceEBean</code> Legt ein Client Service in der Datenbank ab.
<code>cfgDelClientService()</code>	<code>ClientServiceEBean</code> Löscht ein Client Service aus der Datenbank. Existiert zum Zeitpunkt des Löschversuches noch ein Client Service Binding zu einem Incoming Request, so ist ein Löschen nicht möglich - die referentielle Integrität wird durch diese Operation gewährleistet.
<code>cfgAddReqServLink()</code>	<code>RequestServiceEBean</code>

Systemarchitektur

Interface-Methode	verwendete Entity-Beans und Beschreibung
	Legt ein neues Client Service Binding in der Datenbank ab.
<code>cfgDelReqServLink()</code>	RequestServiceEBean Löscht ein Client Service Binding aus der Datenbank.
<code>cfgUpdReqServLink()</code>	RequestServiceEBean Ändert die Daten eines Client Service Binding in der Datenbank. Die Schlüsselattribute werden dabei nicht verändert.
<code>cfgGetRuleEngines()</code>	RuleEngineEBean, IncomingRequestEBean Liest alle aktuell bekannten Rule Engines aus der Datenbank. Zu jeder Rule Engine wird auch die Information geliefert, ob sie im Augenblick bei einem Incoming Request Verwendung findet.
<code>cfgAddRuleEngine()</code>	RuleEngineEBean, IncomingRequestEBean Legt eine neue Rule Engine in der Datenbank ab.
<code>cfgDelRuleEngine()</code>	RuleEngineEBean Löscht eine Rule Engine aus der Datenbank. Ist die Rule Engine zum Zeitpunkt des Löschversuches noch bei einem Incoming Request zugeordnet, so ist ein Löschen nicht möglich - die referentielle Integrität wird durch diese Operation gewährleistet.
<code>getRuleScript()</code>	RuleScriptEBean Liest ein Rule Script aus der Datenbank. Dieses wird über seine ID spezifiziert. Es werden alle Zeilen des Rule Scripts gelesen und in einem String zusammengefügt.
<code>cfgSetRuleScript()</code>	RuleScriptEBean Legt ein neues Rule Script in der Datenbank ab oder ändert ein bestehendes Rule Script. Rule Scripts werden über ihre ID spezifiziert. Da sie beliebig lange sein können, wird folgender Algorithmus angewendet: Ist noch kein Rule Script mit der angegebenen ID vorhanden, so wird es neu angelegt, wobei das übergebene Rule Script in Teile mit vorgegebener Länge gesplittet wird. Diese Teile werden jeweils als einzelne Rows in der Datenbank abgelegt. Ist ein Rule Script mit der angegebenen ID bereits vorhanden, und hat das übergebene Rule Script die Länge 0, so wird das bestehende Rule Script aus der Datenbank gelöscht. Ist ein Rule Script mit der angegebenen ID vorhanden und hat das übergebene Rule Script eine Länge > 0, so wird das bestehende Rule Script durch das neue ersetzt, wobei sich die Anzahl der gespeicherten Rows in der Datenbank abhängig von der neuen Länge auch vergrößern oder verkleinern kann.

Interface-Methode	verwendete Entity-Beans und Beschreibung
<code>cfgGetTemplates()</code>	ScriptTemplateEBean Liest die Menge der aktuell definierten Script Templates aus der Datenbank.
<code>cfgSetTemplate()</code>	ScriptTemplateEBean Legt ein neues Script Template in der Datenbank an.
<code>cfgDelTemplate()</code>	ScriptTemplateEBean Löscht ein Script Template aus der Datenbank.

Tabelle 4 - Persistenz Services für den Configurator

Folgendes Service bietet die Komponente *DB-Connector* dem Dispatcher Processor für die Simulation von FRAG Regeln an (siehe *Persistence Services* in Abbildung 3).

Interface-Methode	verwendete Entity-Bean und Beschreibung
<code>getRuleScript()</code>	RuleScriptEBean diese Methode wird auch von der Configurator Komponente verwendet - siehe Tabelle 4

Tabelle 5 - Persistenz Services für den Processor

Der DB-Connector benötigt seinerseits die in Tabelle 74 unter `RBDServer.server.entities` angeführten Entity Beans für die Kommunikation mit der Datenbank.

3.1.5 Die FRAG Rule Engine

Die Komponente *FRAG Rule Engine* ist für die Auswertung der FRAG Rule Scripts verantwortlich. In dieser Komponente sind die FRAG-Erweiterungen für den Dispatcher definiert. Siehe Kapitel 6 für die Beschreibung dieser Objekte und ihrer Methoden. Die Komponente bietet dem Dispatcher Processor das in der Tabelle unten angeführte Service an (siehe *FRAG Rule Services* in Abbildung 3).

Interface-Methode	Beschreibung
<code>invokeFRAG()</code>	initialisiert die FRAG Erweiterungen für den Dispatcher, ruft den FRAG Interpreter auf und retourniert die Ergebnisse der Auswertung.

Tabelle 6 - FRAG Rule Engine Services für den Processor (Simulation)

3.2 Laufzeit

Die Komponentenarchitektur für die Laufzeitfunktionalität sowie die Visualisierung von statistischen Informationen zur Laufzeit sind in Abbildung 4 dargestellt.

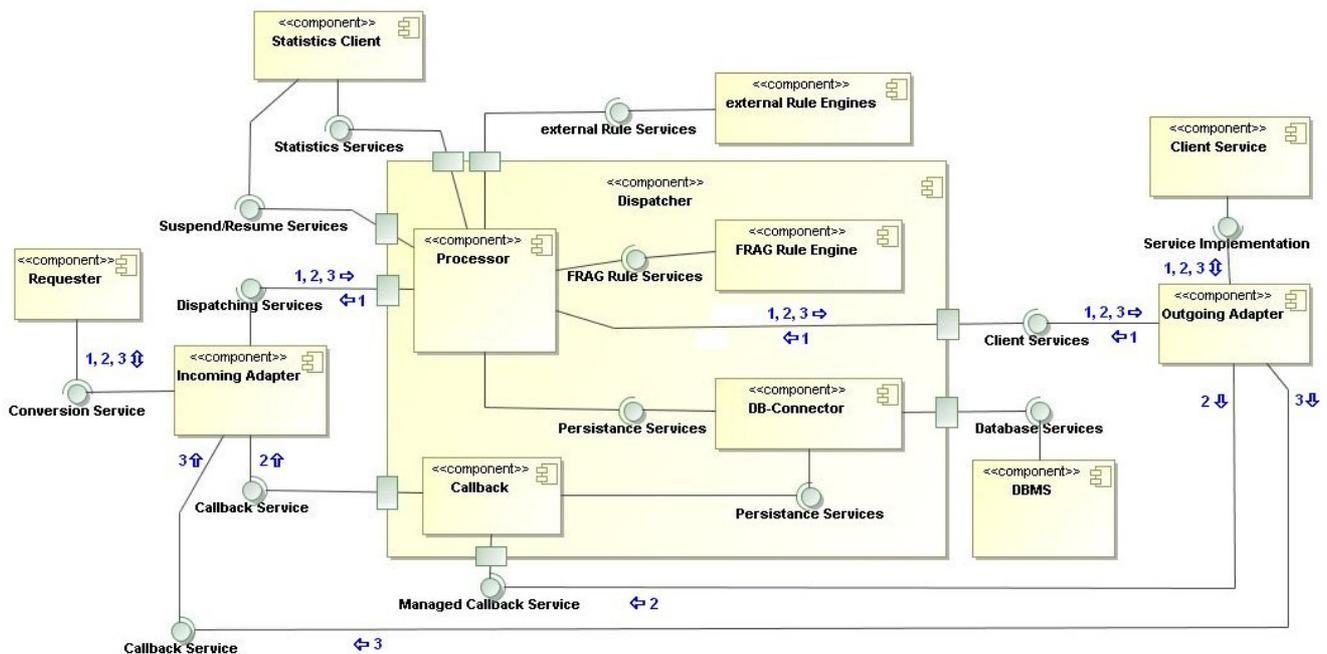


Abbildung 4 - Komponenten Laufzeit

Die Funktionsweise der Komponenten sowie ihre Interfaces werden nun im Einzelnen beschrieben. Die verschiedenen Arten der synchronen und asynchronen Kommunikation sind durch blaue Ziffern und Pfeile dargestellt und werden in Kapitel 5 erläutert.

3.2.1 Requester- und Client Service-Komponenten

Diese Komponenten gehören nicht zum Umfang des Dispatchers und werden hier nur der Vollständigkeit wegen angeführt.

Requester-Komponenten sind die klassischen "Benutzer" des Dispatchers. Dabei kann es sich um beliebige Applikationen handeln, beispielsweise um eine Macroflow Engine aus [Hentrich et al. 2007]. Der Macroflow Engine Simulator aus Kapitel 9.1, welcher im Zuge dieses Projektes entwickelt worden ist, um den Dispatcher zu testen, ist ein weiteres Beispiel.

Requester-Komponenten sind durch ihre Incoming Adapter vom Dispatcher abgeschirmt und für diesen immer unsichtbar.

Client Service-Komponenten implementieren die eigentliche (technische) Funktionalität der Client Services und entsprechen den Microflow Engines aus [Hentrich et al. 2007]. Im Rahmen dieses Projektes wurde eine solche Microflow Engine entwickelt, welche in der Lage ist, die Anfragen des Macroflow Engine Simulators aus Kapitel 9.1 zu verarbeiten. Siehe Kapitel 9.4 für eine Beschreibung dieser Engine und des Zusammenspiels mit dem Dispatcher.

Client Service Komponenten sind durch ihren Outgoing Adapter vom Dispatcher abgeschirmt und für diesen immer unsichtbar.

3.2.2 Incoming Adapter

Incoming Adapter stellen die Verbindung zwischen Requester und Dispatcher dar. Ihre Aufgabe ist es, die Anfragen ihres Requesters entgegen zu nehmen, die Parameter der Anfrage in ein für den Dispatcher verständliches Format zu konvertieren und sodann die Processor-Komponente des Dispatchers aufzurufen. Da der Incoming Adapter den Requester vom Dispatcher und den dahinter liegenden Client Services abschirmt, ist es naheliegend (aber nicht zwingend), dass dieser Adapter auch den Einsprungspunkt für Callbacks im Falle von asynchronen Verarbeitungen zur Verfügung stellt. In diesem Fall sowie im Falle von synchronen Aufrufen, in denen das Ergebnis ja direkt an den Incoming Adapter zurückgeliefert wird, hat dieser zusätzlich die Aufgabe, die erhaltenen Returnwerte in ein für den Requester verständliches Format zu konvertieren und an diesen zurückzuliefern.

Die Processor-Komponente des Dispatchers bietet den Incoming Adaptern folgendes Service an.

Interface-Methode	Beschreibung
<code>processActivity()</code>	Führt einen Request des Incoming Adapter aus und retourniert das Ergebnis im Falle eines synchronen Aufrufes oder - im Falle eines asynchronen Aufrufes - einen Statuscode, welcher den Erfolg oder Misserfolg der Weiterleitung des Aufrufes an ein vom Dispatcher ermitteltes Client Service dokumentiert.

Tabelle 7 - Processing Services für Incoming Adapter

Die Processor-Komponente des Dispatchers erwartet im Falle von asynchronen Verarbeitungen (bei managed Callbacks) vom Incoming Adapter die Information über ein nach Beendigung der Verarbeitung aufzurufendes Callback Service.

Interface-Methode	Beschreibung
<code><callback-method></code>	Jene Methode, welche vom Dispatcher im Falle eines managed Callback angesprochen wird. Der Name der Methode sowie die URL-Adresse und Beiname werden nicht vom Dispatcher vorgegeben, sondern werden vom Incoming Adapter an den Dispatcher übermittelt.

Tabelle 8 - Callback Service für den Dispatcher

In Kapitel 9.2 wird ein Beispiel für einen Incoming Adapter vorgestellt. Dort wird auch die Verwendung der Interfaces und Callbacks erläutert.

3.2.3 Outgoing Adapter

Outgoing Adapter stellen die Verbindung zwischen Dispatcher und Client Services dar. Ihre Aufgabe ist es, die Anfragen des Dispatchers entgegen zu nehmen, die Parameter der Anfrage in ein für ihr Client Service verständliches Format zu konvertieren und sodann das Client Service aufzurufen. Da der Outgoing Adapter den Dispatcher von "seinen" Client Services abschirmt, ist es naheliegend (aber nicht zwingend), dass dieser Adapter auch die Callbacks im Falle von asynchronen Verarbeitungen aufruft. In diesem Fall sowie im Falle von synchronen Aufrufen, in denen das Ergebnis ja vom Client Service direkt an den Outgoing Adapter zurückgeliefert wird, hat dieser zusätzlich die Aufgabe, die erhaltenen Returnwerte in ein für den Empfänger verständliches Format zu konvertieren und an diesen zurückzuliefern. Empfänger sind entweder das Callback Service des Dispatchers im Falle von managed Callbacks oder das Callback Service des Requesters (beispielsweise dessen Incoming Adapter; siehe oben) im Falle von non-managed Callbacks oder synchroner Verarbeitung.

Die Processor-Komponente des Dispatchers erwartet von Outgoing Adapters keine fix vorgegebene Interfacemethode, sondern diese Information wird vom Adapter selbst geliefert - siehe Kapitel 4.2.2 und 4.2.3 für die Beschreibung der Interface Reflection. Die Parameter sind allerdings fix definiert.

In Kapitel 9.3 wird ein Beispiel für einen Outgoing Adapter vorgestellt. Dort ist auch die Verwendung der Interfaces erläutert.

3.2.4 Callback Service

Das Callback Service des Dispatchers stellt dem Outgoing Adapter für asynchrone Verarbeitungen (bei managed Callbacks) folgendes Interface zur Verfügung.

Interface-Methode	Beschreibung
<code>activityCallback()</code>	jene Methode, welche vom Dispatcher als Einsprungspunkt für Callbacks von Client Services bzw. deren Outgoing Adapter bereitgestellt wird.

Tabelle 9 - Callback Service für den Outgoing Adapter

Bei asynchronen Verarbeitungen hat die Processor Komponente zuvor die vom Requester spezifizierte, ursprüngliche Callback-Methode sowie den Namen und die URL ihrer Enterprise Java Bean in der Datenbank abgelegt und durch diese Methode ersetzt. Darüber hinaus wurde auch die Correlation-ID des Requesters durch eine neue Correlation-ID für die Kommunikation mit dem Client Service ersetzt.

Das Callback Service liest nun die ursprüngliche Callback Methode sowie die ursprüngliche Correlation-ID aus der Datenbank, legt die ermittelten Runtime-Statistics in der Datenbank ab und ruft den Callback des Requesters auf.

3.2.5 Die Processor Komponente

Der Processor ist die zentrale Komponente für die Verarbeitung von Incoming Requests. Er liest die dem Incoming Request zugeordneten Client Services aus der Datenbank, ermittelt FRAG Rule Script und/oder Rule Engine, wertet diese aus und stößt die Client Services an.

Der Processor bietet den Incoming Adaptern das in Tabelle 7 beschriebene Service an. Darüber hinaus stellt er dem Statistics Client die in Tabelle 13 angeführten Methoden zur Verfügung.

Die Processor Komponente benötigt Ihrerseits die Services der FRAG Rule Engine (Tabelle 6) oder - falls für einen Incoming Request verwendet - die Services der externen Rule Engines.

Folgende Services bietet die Komponente DB-Connector dem Dispatcher Processor zur Laufzeit an (siehe *Persistence Services* in Abbildung 4).

Interface-Methode	verwendete Entity-Beans und Beschreibung
<code>getRuleScript()</code>	RuleScriptEBean liest ein FRAG Rule Script aus der Datenbank
<code>procSetRuntimeStats()</code>	RuntimeStatsEBean Setzt die Runtime Statistics für Incoming Requests und Client Services auf die Ausgangswerte zurück (Init).
<code>procUpdRuntimeStats()</code>	RuntimeStatsEBean Ändert die Runtime Statistics für Incoming Requests und Client Services auf die aktuell gültigen Werte.
<code>procGetAllReqStats()</code>	IncomingRequestEBean, RuntimeStatsEBean Ermittelt alle statistischen Daten für alle Incoming Request Objekte
<code>procGetAllSvcStats()</code>	ClientServiceEBean, RuntimeStatsEBean Ermittelt alle statistischen Daten für alle Client Service Objekte
<code>procGetRequest()</code>	IncomingRequestEBean, RequestServiceEBean ermittelt die Konfigurationsdaten eines Incoming Request, dessen aktuell definierte Client Service Bindings sowie die Information, ob der Request im Moment von der Verarbeitung ausgeschlossen (suspended) ist oder nicht
<code>procGetClientServiceGroup()</code>	ClientServiceGroupEBean ermittelt Informationen über Client Service Gruppen
<code>procGetClientService()</code>	ClientServiceEBean ermittelt Informationen über Client Services
<code>procGetRuleEngine()</code>	RuleEngineEBean ermittelt Informationen über die dem Incoming Request zugeordnete Rule Engine
<code>procGetCSInvocationContext()</code>	RequestServiceEBean, ClientServiceEBean ermittelt alle Informationen, welche benötigt werden, um einen Client Service Adapter aufzurufen

Interface-Methode	verwendete Entity-Beans und Beschreibung
setCallbackInfo	ServiceCallbackEBean legt vor dem asynchronen Aufruf von Client Services Informationen über Callbacks in der Datenbank ab
getCallbackInfo	ServiceCallbackEBean liest diese Callback-Information aus der Datenbank
deleteCallbackInfo	ServiceCallbackEBean löscht diese Callback-Information aus der Datenbank

Tabelle 10 - Persistenz Services für den Processor

Die Verarbeitung zur Laufzeit und die Reihenfolge der Aktivierung der Komponenten bei den unterschiedlichen Verarbeitungen wird in Kapitel 5 im Detail erläutert.

3.2.6 FRAG Rule Engine und externe Rule Engines

Die Komponente FRAG Rule Engine ist für die Auswertung der FRAG Rule Scripts verantwortlich - diese Komponente wird auch bei der Simulation von FRAG Rule Scripts im Zuge der Konfiguration verwendet (siehe Kapitel 3.1.5).

Interface-Methode	Beschreibung
invokeFRAG()	initialisiert die FRAG Erweiterungen für den Dispatcher, ruft den FRAG Interpreter auf und retourniert die Ergebnisse der Auswertung.

Tabelle 11 - FRAG Rule Engine Services für den Processor (Laufzeit)

Regeln für den Dispatcher können nicht nur in Form von FRAG Rule Scripts definiert, sondern auch als externe Rule Engines implementiert werden. In diesem Fall benötigt der Processor von den externen Rule Engines die folgenden Services.

Interface-Methode	Beschreibung
getEngineDescription()	liefert eine Beschreibung der externen Rules Engine
getRBDRuleServices()	liefert Informationen über die Services der externen Rules Engine.

Tabelle 12 - Interface externer Rule Engine Services

Siehe Kapitel 4.3 für eine detailliertere Diskussion dieser Methoden.

3.2.7 Statistics Client

Der Client zur Visualisierung von statistischen Informationen der vorangegangenen Verarbeitungen wird durch die Processor-Komponente des Dispatchers bedient. Diese bietet dem Statistics Client die folgenden Services an (siehe *Statistics Services* in Abbildung 4). Überdies können über ein Interface dieser Komponente Incoming Requests und Client Services temporär aktiviert und inaktiviert werden (siehe *Suspend/Resume Services* in Abbildung 4).

Interface-Methode	Beschreibung
<code>procGetAllReqStats()</code>	Liefert zu jedem bekannten Incoming Request die aktuell verfügbaren statistischen Informationen.
<code>procGetAllSvcStats()</code>	Liefert zu jedem bekannten Client Service die aktuell verfügbaren statistischen Informationen.
<code>procSetRuntimeStats()</code>	Initialisiert die statistischen Informationen für einen Incoming Request oder ein Client Service. Kennzeichnet einen Incoming Request oder ein Client Service als inaktiv (suspended) oder aktiviert einen Incoming Request oder ein Client Service

Tabelle 13 - Processing Services für Statistics Client

Das grafische User Interface des Statistics Client wird in Kapitel 4.5 erläutert.

4 Dispatcher Konfiguration und Statistiken

Der Configuration Client bietet die Möglichkeit, *Incoming Requests*, *Client Services* und *Rule Engines* zu definieren. Beim Start der Applikation werden Serveradresse und Port des Dispatchers als Startup-Parameter übergeben und in der Folge in der Kopfzeile des Hauptdialoges angezeigt.

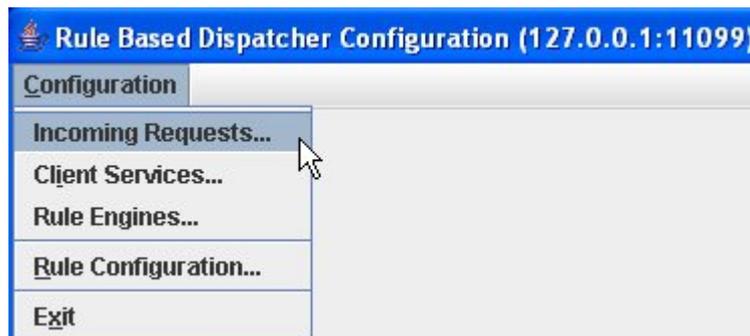


Abbildung 5 - Configuration Client Hauptmenü

Die von der Komponente gebotenen Konfigurationsmöglichkeiten werden in den folgenden Kapiteln dargestellt.

4.1 Incoming Requests

4.1.1 Dialog

Über diesen Dialog werden dem Dispatcher jene Anfragen bekannt gemacht, welche er später verarbeiten können soll. Der Dialog ist in Abbildung 6 dargestellt.

Jeder Incoming Request kann 2 Klassifikationskriterien aufweisen: die Request Gruppe (Request Group) - beispielsweise die ID der Business Process Engine, von der aus die Anfrage abgesetzt worden ist - sowie die Identifikation der Anfrage selbst. Beide Kriterien zusammen kennzeichnen eine Anfrage eindeutig.

Die beiden Kriterien werden im *Incoming Request*-Dialog manuell erfasst, optional können auch Beschreibungen in freier Textform für Gruppe und Anfrage erfasst werden. Aus den eingegebenen Identifikationen werden die intern für diese Anfrage verwendeten Schlüssel generiert - diese dienen als technische Identifikationen in diversen Entitäten (siehe Kapitel 7 Datenbankschema). Die generierten IDs werden jeweils rechts neben den Eingabefeldern grau hinterlegt angezeigt.

Durch Betätigen der Schaltflächen Insert , Modify  und Remove  werden die definierten Services in das Repository des Dispatchers aufgenommen, modifiziert bzw. aus dem Repository entfernt.

Dispatcher Konfiguration und Statistiken

Für Fälle, in denen die IDs der einlangenden Requests im Vorhinein nicht bekannt sind, wird beim Hinzufügen einer neuen Request Gruppe auch ein Incoming Request mit Request-ID *"-unspecified-"* angelegt. Dieser wird für die Verarbeitung all jener Anfragen herangezogen, welche keine Request-ID liefern. Ein Incoming Request mit zwei *"-unspecified-"* IDs (für die Anfrage-Gruppe und die Anfrage selbst) ist immer vorhanden. Auf diese Weise können auch anonyme Anfragen verarbeitet werden, welche überhaupt keine Kennung liefern.

Davon zu unterscheiden wären *"-other-"*-Einträge für die IDs. Über diese könnten all jene Anfragen verarbeitet werden, auf welche keine der dezidierten IDs "passen". Das bedingt aber eine mehrstufige Abfrage bei der Auswertung und wurde vorerst aus Performancegründen nicht implementiert.

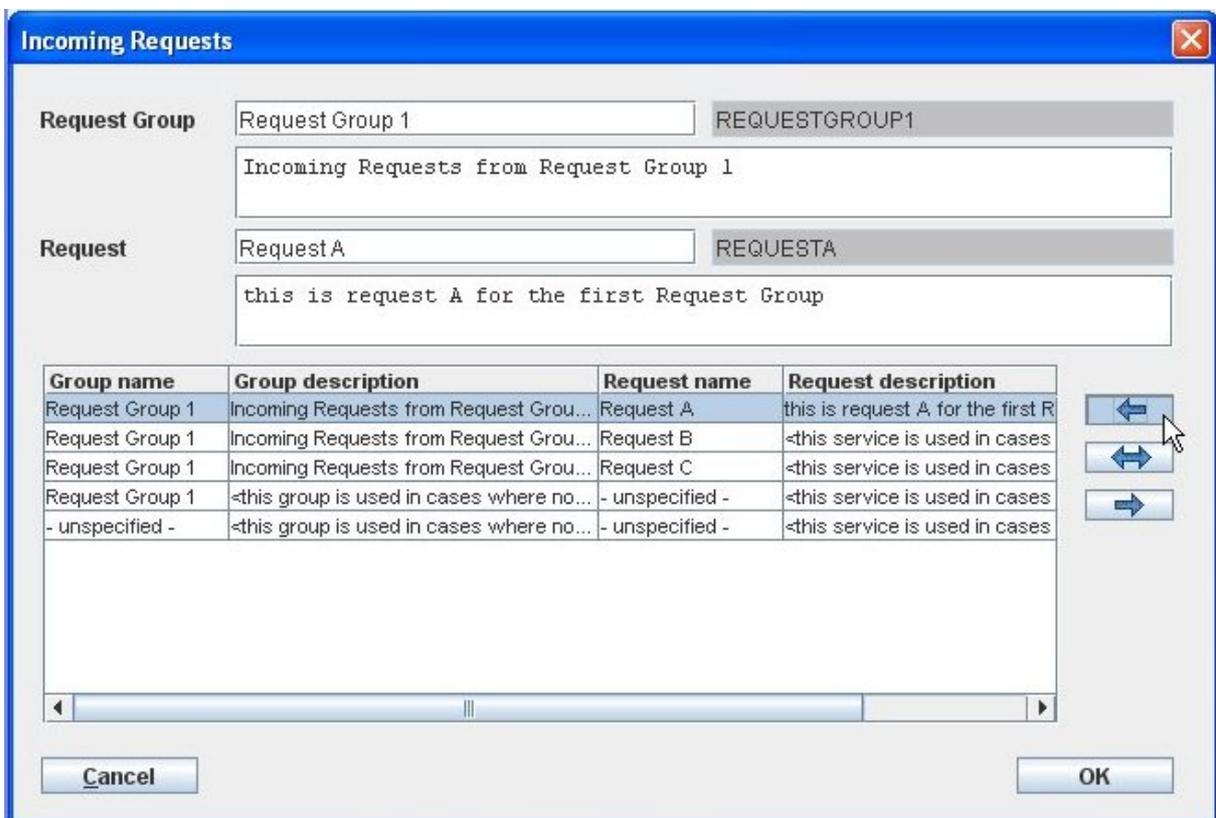


Abbildung 6 - Dialog Incoming Requests

Die in der Tabbox dargestellten Einträge bilden das Request Repository des Dispatchers. Hier ist auf einen Blick sichtbar, welche Anfragen vom Dispatcher verarbeitet werden können. Wird eine Anfrage aus der Tabbox entfernt, kann sie nicht mehr entgegengenommen werden - der Dispatcher liefert in diesem Fall einen Fehler an den Aufrufer. Ein Incoming Request kann nicht aus der Tabbox entfernt werden, solange noch Client Services zu diesem Request zugeordnet sind.

4.2 Client Services

4.2.1 Hauptdialog

Über diesen Dialog werden dem Dispatcher die verfügbaren Client Services durch die Definition Ihrer Outgoing Adapter bekannt gemacht. Der Dialog ist in Abbildung 7 dargestellt.

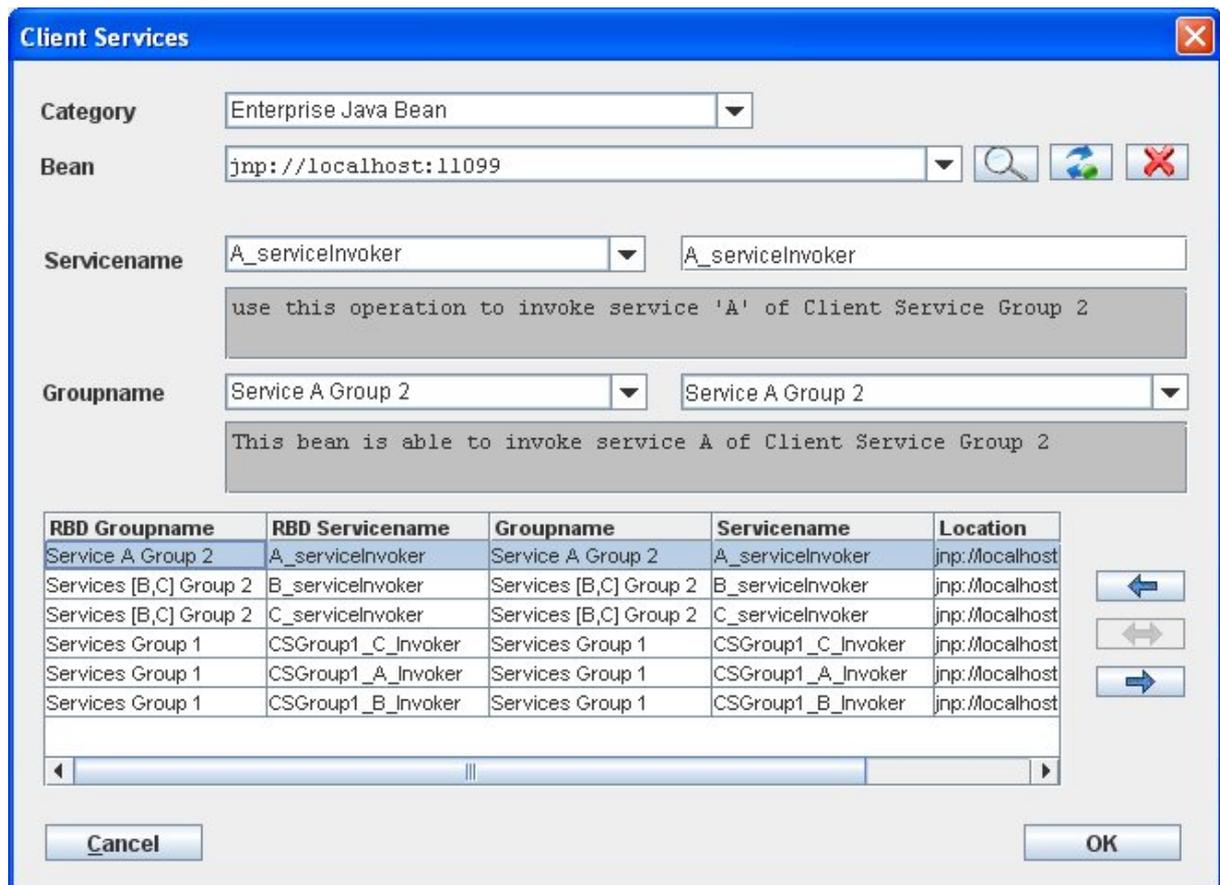


Abbildung 7 - Dialog Client Services

In der Groupbox *Category* wird gewählt, ob das Service über eine Enterprise Java Bean oder als Web Service implementiert ist. Im Falle einer Enterprise Java Bean wird die Combobox *Bean* mit der Adresse des Applikationsservers vorbelegt - diese kann aber jederzeit geändert werden.

4.2.2 Client Service Adapter als Enterprise Java Beans

Durch Betätigen der Schaltfläche Suche  wird am gewählten Applikationsserver nach allen Enterprise Java Beans gesucht, welche als Outgoing Adapter für Client Services geeignet sind. Das sind all jene Beans, welche das unten dargestellte Reflection-Interface implementieren.

Dispatcher Konfiguration und Statistiken

```
// provides a description of the beans services
public NameDscPair getServiceDescription();

// provides the operations which can be invoked by this bean
public Hashtable<String, String> getRBDServices();
```

Im Package `RBDEngines.connectors` wurden einige solche Adapter-Beans testweise implementiert (siehe Tabelle 74 in Kapitel 8.1). Im Beispiel unten bietet eine gedachte "Microflow Engine 2" drei Client Services A, B, C an, wobei Service A über den Adapter `MIFE2ServiceABean`, die Services B und C über den Adapter `MIFE2ServiceBCBean` angesprochen werden.

Das Remote Interface der Adapter Bean `MIFE2ServiceBCBean` sieht aus wie folgt:

```
package connectors;

import java.util.Hashtable;
import javax.ejb.Remote;

import server.common.Callback;
import server.common.NameDscPair;
import server.common.RBDStatus;
import server.sessions.IncomingMessage;

// this is the adapter for services B and C on microflow engine 2
@Remote
public interface MIFE2ServiceBCRemote
{
    public RBDStatus B_serviceInvoker(String corrID, IncomingMessage
                                        incomingMsg, Callback callback);
    public RBDStatus C_serviceInvoker(String corrID, IncomingMessage
                                        incomingMsg, Callback callback);

    // reflection interface used by the configuration system
    public NameDscPair getServiceDescription(); // description
    public Hashtable<String, String> getRBDServices(); // operations
}
```

Die Implementierung der Methode `getRBDServices()` ist in folgendem Abschnitt dargestellt:

```
public Hashtable<String, String> getRBDServices ()
{
    // this method provides information about the operations
    // provided by the bean
    Hashtable<String,String> htServices= new Hashtable<String,String> ();

    htServices.put ("B_serviceInvoker",
                    "use this operation to invoke ...");
    htServices.put ("C_serviceInvoker",
                    "use this operation to invoke ...");
    return htServices;
}
```

Es werden also die beiden Services `B_serviceInvoker` und `C_serviceInvoker` als Adapter-Methoden an den Caller zurückgeliefert.

Dispatcher Konfiguration und Statistiken

Die so ermittelten Beans und ihre Services werden im Dialog Bean Selector angezeigt, wie in Abbildung 8 dargestellt.

Wird von einem Adapter mehr als ein Service unterstützt, so wird das durch ein dem Service-Namen nachgestelltes ", ..." angezeigt (siehe Spalte Service in der Abbildung unten).



Abbildung 8 - Dialog zur Auswahl von Client Service Adapter Beans

Im Dialog wird nun der gewünschte Adapter gewählt und mit OK in den Dialog *Client Services* übernommen.



Abbildung 9 - Client Service Adapter Bean

Der Name des Adapter wird in der Combobox Bean angezeigt (Abbildung 9) und steht nun für die weiteren Verarbeitungen zur Verfügung.

Dispatcher Konfiguration und Statistiken

4.2.3 Client Service Adapter als Web Service

Im Dialog *Client Services* (Abbildung 7) kann als *Category* auch der Eintrag "Web Service" gewählt werden.

In diesem Fall wird bei Betätigen der Schaltfläche Suche  in einen Dateiauswahldialog verzweigt, in dem ein WSDL-File mit der Beschreibung der Web Services gewählt werden kann.

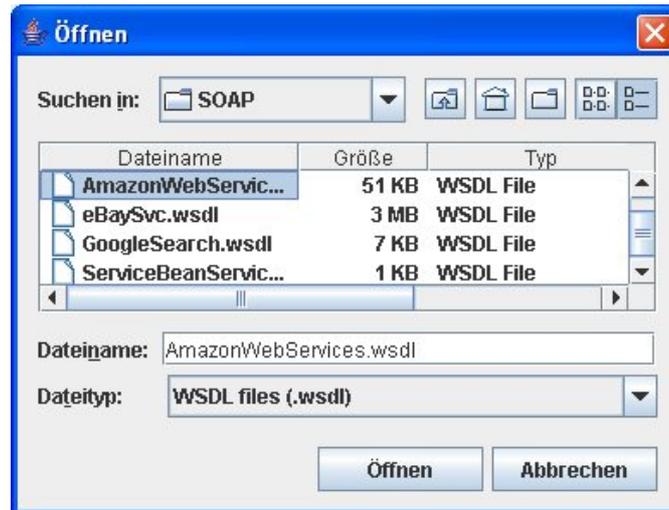


Abbildung 10 - Dialog zur Auswahl von Client Service Web Services

Durch Betätigen der Schaltfläche Öffnen wird der Dateiname in den Dialog *Client Services* übernommen und steht hier für die weitere Verarbeitung zur Verfügung.

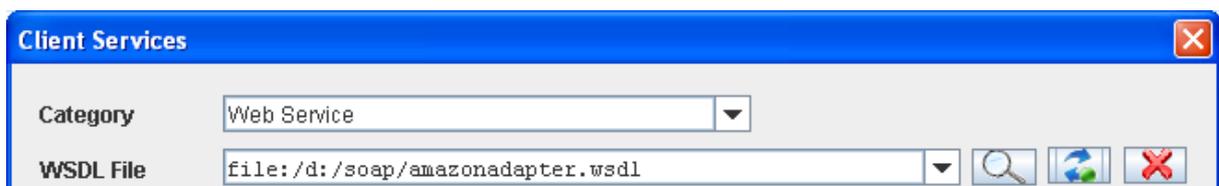


Abbildung 11 - Client Service Web Service

Alternativ kann im Dialog *Client Services* auch der Name der WSDL-Datei direkt eingegeben werden (Abbildung 11).

Dispatcher Konfiguration und Statistiken

Der WSDL-Parser des Dispatchers liest die WSDL-Datei und extrahiert <service>- und <operation>-Namen. Der Servicename wird in der Groupbox *Groupname* angezeigt, die Namen der einzelnen Operationen in der Groupbox *Servicename*.

Als Beispiel sei hier ein Adapter für Amazon-Webservices angeführt:

```
[...]
<operation name="ActorSearchRequest">
  <documentation>
    This service is used to search for author names
  </documentation>
  <input message="typens:ActorSearchRequest"/>
  <output message="typens:ActorSearchResponse"/>
</operation>

<operation name="AddShoppingCartItemsRequest">
  <documentation>
    no documentation available
  </documentation>
  <input message="typens:AddShoppingCartItemsRequest"/>
  <output message="typens:ShoppingCartResponse"/>
</operation>

<operation name="ArtistSearchRequest">
  <documentation>
    This service is used to search for artist names
  </documentation>
  <input message="typens:ArtistSearchRequest"/>
  <output message="typens:ArtistSearchResponse"/>
</operation>
[...]

<service name="AmazonSearchAdapter">
  <!-- This is the adapter for accessing Amazon Web APIs -->
  <documentation>
    This is the adapter for accessing amazon search services. Only a
    limited number of amazon services is supported at the moment.
  </documentation>
  <port name="AmazonSearchPort" binding="typens:AmazonSearchBinding">
    <soap:address location="http://soap.amazon.com/onca/soap2"/>
  </port>
</service>
[...]
```

Sind innerhalb von <service>- oder <operation>-Definitionen Beschreibungen angeführt, so werden diese in die entsprechenden Textfelder des Dialoges übernommen. Siehe Abbildung 12.

Dispatcher Konfiguration und Statistiken

4.2.4 Parsing und Zuordnung der Client Services

Die weitere Verarbeitung läuft für beide Arten von Client Service Adaptern gleich ab.

Durch Betätigen der Schaltfläche Parse  werden die von der gewählten Enterprise Java Bean bzw. vom gewählten Web Service gebotenen Services in den Groupboxen *Service*name und *Group*name angezeigt. In Abbildung 12 wurde die oben angeführte WSDL-Datei geparsed.

WSDL File: file:/d:/soap/amazonadapter.wsdl

Servicename: ActorSearchRequest
This service is used to search for author names

Groupname: AmazonSearchAdapter
This is the adapter for accessing amazon search services. Only a limited number of amazon services is supported at the moment.

Abbildung 12 - Informationen aus Web Service Adapter

Abbildung 13 zeigt die analoge Darstellung für die in Kapitel 4.2.2 besprochene Enterprise Java Bean.

Bean: jnp://localhost:11099/MIFE2ServiceBCBean

Servicename: B_serviceInvoker
es 'B' of Client Service Group 2

Groupname: Services [B,C] Group 2
This Bean is able to invoke services B and C of Client Service Group 2

Abbildung 13 - Informationen aus Bean Adapter

Es ist möglich, zu jedem externen (d.h. durch die Bean oder das Web Service vorgegebenen) Service- oder Gruppennamen auch jeweils einen internen Namen zu vergeben. Diese internen Namen können später in den Regeldefinitionen verwendet werden.

Durch Betätigen der Schaltfläche Insert  wird das Client Service in die Tabbox eingefügt, mittels Remove  wird ein zuvor definiertes Client Service aus der Tabbox entfernt. Sobald der Dialog mit OK verlassen wird, werden die durchgeführten Änderungen in der Datenbank festgeschrieben.

4.3 Rule Engines

Über diesen Dialog werden dem Dispatcher die verfügbaren externen Rule Engines bekanntgemacht. Rule Engines sind entweder als Enterprise Java Beans oder als Web Services implementiert. Der Dialog zur Konfiguration ist in Abbildung 14 dargestellt.

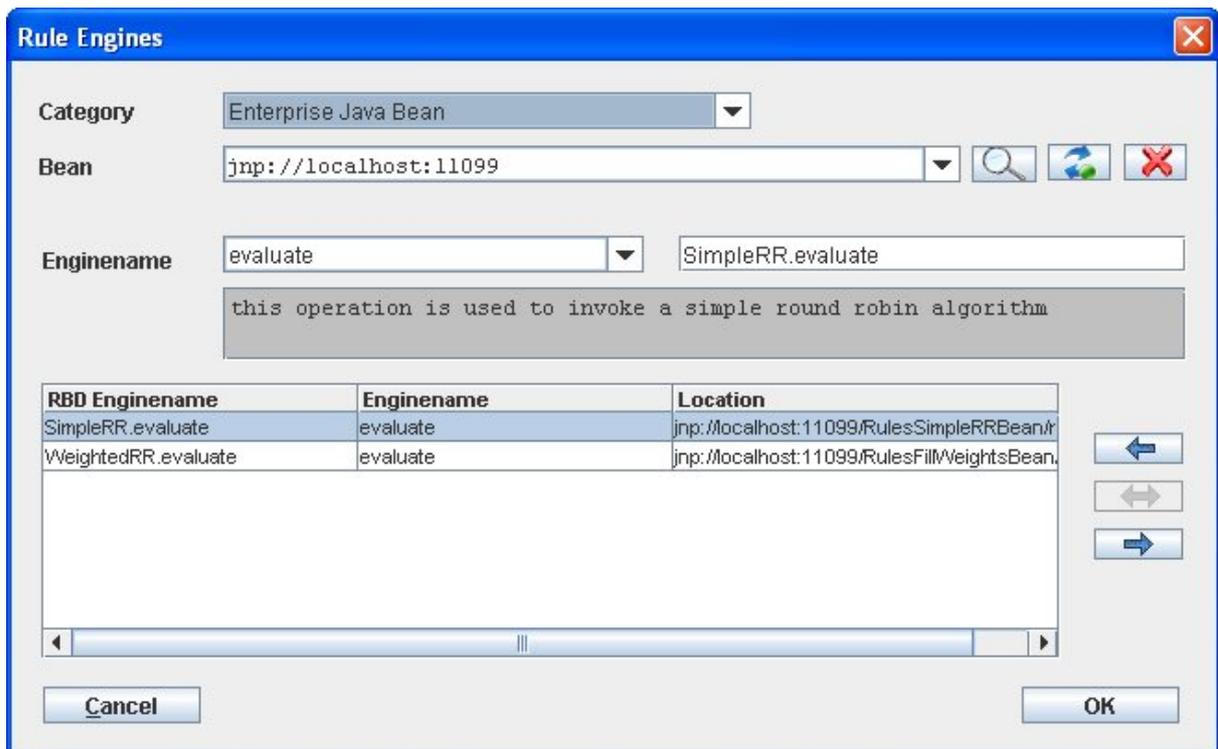


Abbildung 14 - Dialog Rule Engines

Das Handling des Dialoges ist völlig analog dem des Dialoges *Client Services* und wird hier nicht nochmals beschrieben.

Ein Unterschied besteht nur im Reflection Interface bei Enterprise Java Beans. Dieses muss in diesem Fall folgende Methoden implementieren.

```
// provides a description of the beans services
public NameDscPair getEngineDescription();

// provides the operations which can be invoked by this bean
public Hashtable<String, String> getRBDRuleServices();
```

Im Package `RBDServer.server.rules` wurden solche Rule Engine Beans testweise implementiert (siehe Tabelle 74 in Kapitel 8.1). Im Beispiel unten bietet eine `RulesSimpleRRBean` zwei einfache Round-Robin Algorithmen an.

Dispatcher Konfiguration und Statistiken

Das Remote Interface der Rule Engine Bean RulesSimpleRRBean sieht aus wie folgt:

```
package server.rules;

import java.util.Hashtable;
import java.util.Vector;
import javax.ejb.Remote;
import server.RuleEngineResult;
import server.common.NameDscPair;
import server.common.RBDEException;
import server.sessions.IncomingMessage;

@Remote
public interface RulesSimpleRRRemote
{
    public RuleEngineResult evaluate(Hashtable<String, Object>
        statsIncoming, IncomingMessage paramIncoming,
        Vector<Hashtable<String, Object>> paramCliService)
        throws RBDEException;

    // reflection interface used by the configuration system
    public NameDscPair getEngineDescription(); // description
    public Hashtable<String, String> getRBDRuleServices(); // operations
}
```

Die Implementierung der Methode getRBDRuleServices() macht die beiden Round Robin Algorithmen nach außen hin bekannt.

```
public Hashtable<String, String> getRBDRuleServices()
{
    // this method provides information about the operations
    // provided by the bean
    Hashtable<String, String> htServices = new Hashtable<String, String>();

    htServices.put("evaluate", "this operation is used to invoke " +
        "a simple round robin algorithm");
    htServices.put("simpleRR", "another simple round robin algorithm");
    return htServices;
}
```

Diese beiden Algorithmen können im Dialog ausgewählt werden. Die Steuerung der Tabbox erfolgt wie gewohnt. Wird versucht, eine Rule Engine zu entfernen, welche noch einem oder mehreren Incoming Requests zugeordnet ist, so reagiert das System mit einer Fehlermeldung. Der Löschvorgang kann in diesem Fall nicht durchgeführt werden und die referentielle Integrität der Daten bleibt gewahrt.

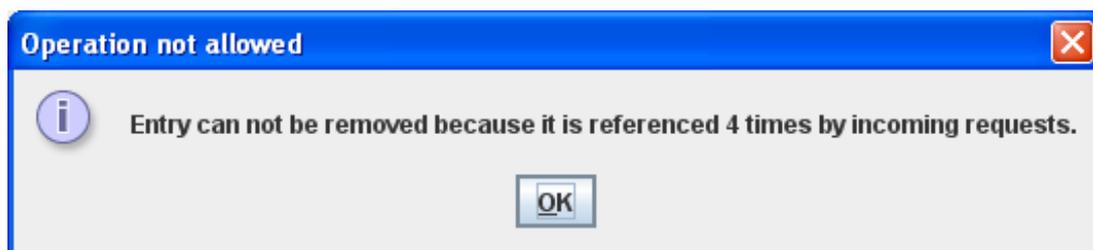


Abbildung 15 - Fehlermeldung bei Löschversuch von Rule Engines

4.4 Rule Configuration

Die Rule Configuration ist jene Stelle im Pflegesystem, in dem die bisher definierten Elemente Incoming Requests, Client Services, Rule Engines und Rule Scripts verknüpft werden. Die beteiligten Dialoge und Konfigurationsmöglichkeiten werden in den folgenden Kapiteln dargestellt.

4.4.1 Rule Configuration - Hauptdialog

Über diesen Dialog (Abbildung 16) werden den Incoming Requests die möglichen Client Services sowie Rule Engines zugeordnet. Von hier aus kann auch in den Dialog zur Definition von Rule Scripts verzweigt werden.

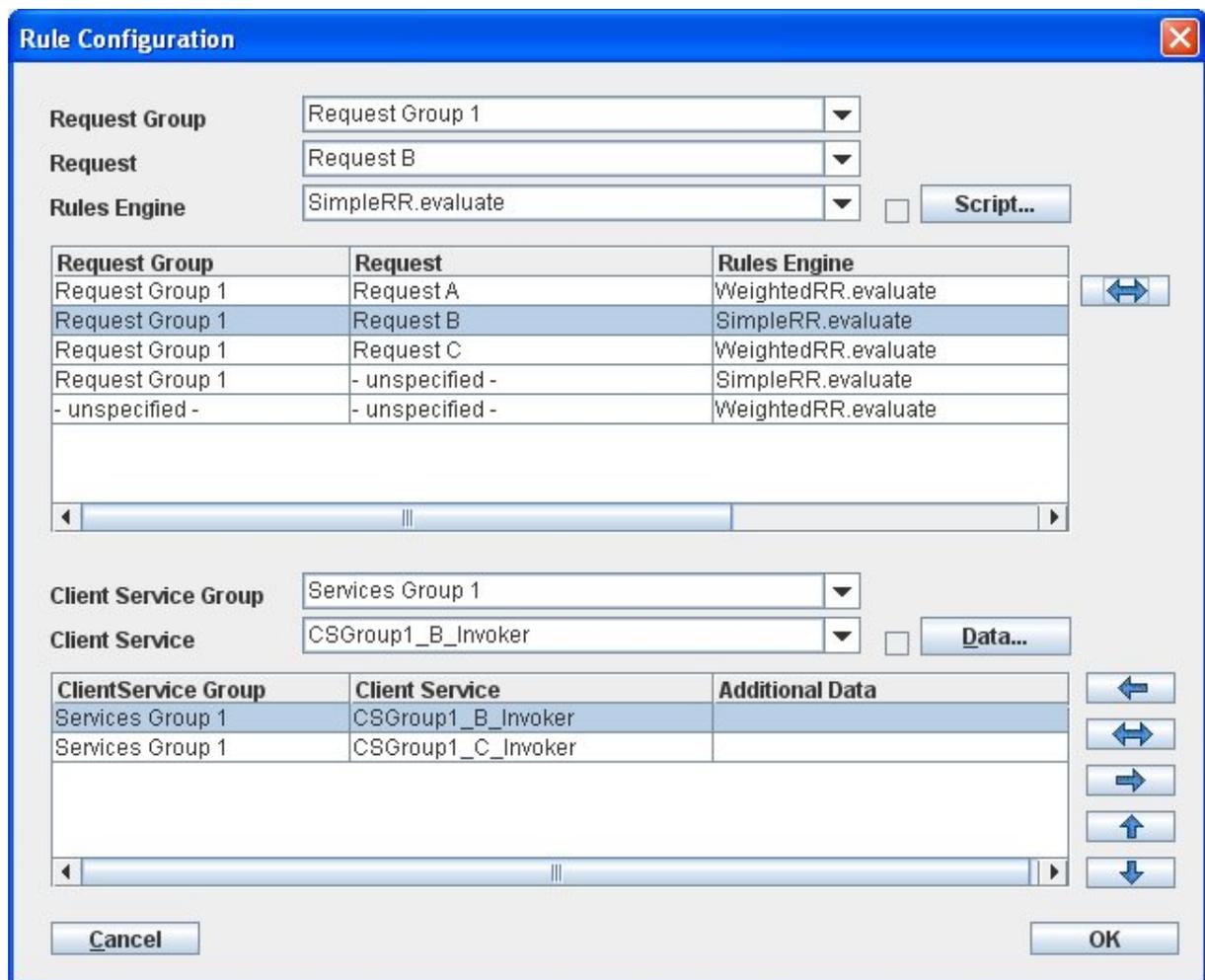


Abbildung 16 - Dialog Rule Configuration

Bei Aufruf des Dialoges wird die obere Tabbox ("Request-Tabbox") mit allen bekannten Incoming Requests vorbefüllt, ein Hinzufügen oder Löschen von Requests ist hier nicht möglich. In den Comboboxen *Request Group* und *Request* sind die bekannten Incoming Requests enthalten. Die Auswahl eines Items in einer dieser Boxen führt zur Auswahl des entsprechenden Eintrages in der Request-Tabbox.

Dispatcher Konfiguration und Statistiken

Anmerkung: Der Request mit der Kennung "- *unspecified* -" wird für jede Request Gruppe unterstützt. Über diesen Eintrag können all jene Incoming Requests verarbeitet werden, welche zwar die Kennung ihrer Request Gruppe, aber keine eigene Request-Kennung liefern. Über die Request Gruppe "- *unspecified* -" werden all jene Incoming Requests abgearbeitet, welche überhaupt keine Kennung liefern.

Zu jedem gewählten Incoming Request kann ein Rule Script erfasst werden (siehe Kapitel 4.4.3). Die Angabe einer Rule Engine ist optional ebenfalls möglich. Diese Rule Engine bestimmt in Fällen, in denen kein Rule Script spezifiziert wird, die Client Services, an welche der Request weiterzuleiten ist. Eine Rule Engine kann aber auch zusätzlich zu einem Rule Script definiert und im Ergebnis der FRAG-Auswertung referenziert werden - siehe Kapitel 6.6 für eine Beschreibung des Ergebnisses der Rule Script Auswertung.

Werden für einen Incoming Request weder ein FRAG Rule Script noch eine externe Rule Engine definiert, so werden die Client Services in der konfigurierten Reihenfolge für die Verarbeitung herangezogen.

Zuordnung von Client Services:

Jede Auswahl einer Zeile in der Request-Tabbox führt dazu, dass die Tabbox in der unteren Dialoghälfte ("Service-Tabbox") mit all jenen Client Services befüllt wird, die dem Eintrag in der Request-Tabbox im Moment zugeordnet sind. Diese Menge an Client Services kann nun durch weitere Services ergänzt werden. Ebenso ist es möglich, zugeordnete Services zu entfernen oder ihre Reihenfolge über die Schaltflächen Up  und Down  zu ändern.

Anmerkung: Die hier definierte Reihenfolge entspricht der Reihenfolge, in welcher die Services später beim Einlangen eines Incoming Request aufgerufen werden, falls weder durch ein FRAG Rule Script noch durch eine Rule Engine eine andere Reihenfolge vorgegeben oder die Menge der Services eingeschränkt wird.

Anmerkung: In der Regel wird nur das jeweils erste der definierten Services vom Dispatcher angestoßen werden. Weitere Services kommen nur zum Zug, wenn alle vorangegangenen Serviceaufrufe mit einem technischen Fehler beendet wurden.

Die hier definierten Services stellen alle theoretisch möglichen Implementierungen für einen Incoming Request dar. Die Informationen über diese Services werden zum Zeitpunkt der Verarbeitung eines Incoming Request an die Rule Engine bzw. das FRAG Rule Script als Inputparameter übergeben (siehe Kapitel 6.1 und 6.4). Rule Engine und Rule Script können die Auswahl beliebig einschränken sowie die Reihenfolge beliebig ändern.

Dispatcher Konfiguration und Statistiken

4.4.2 Dialog zur Definition von Zuordnungsattributen

Für alle Fälle, in denen die Reihenfolge der Zuordnung als Kriterium in der Regelauswertung nicht ausreicht, besteht die Möglichkeit, eine beliebige Anzahl von frei definierbaren Attributen für jede Zuordnung festzulegen. Dies geschieht über den Dialog Service Assignment Parameters (Abbildung 17), in welchen über die Schaltfläche **Data...** im Dialog Rule Configuration verzweigt werden kann.

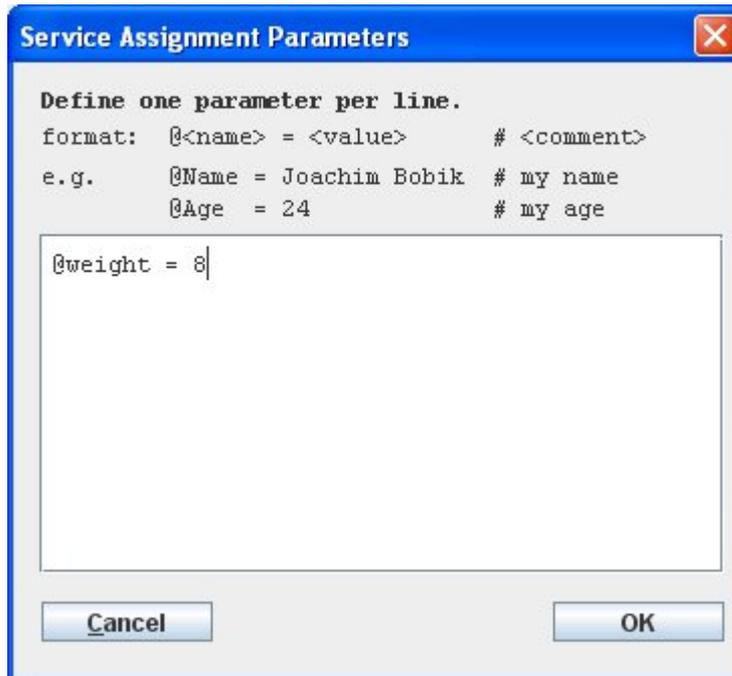


Abbildung 17 - Dialog Service Assignment Parameters

Zuordnungsattribute werden in der Form `@<name> = <value>` definiert, Zeichen hinter `#` bis zum Ende der jeweiligen Zeile werden ignoriert.

Beispielsweise könnte für einen Incoming Request ein Round Robin Algorithmus definiert werden, der die zugeordneten Services unterschiedlich gewichtet, d.h. die Services nicht gleich oft aufruft, sondern in einem Verhältnis, welches durch eine Attribut `@weight` festgelegt wird. In Abbildung 18 ist dieses Beispiel dargestellt.

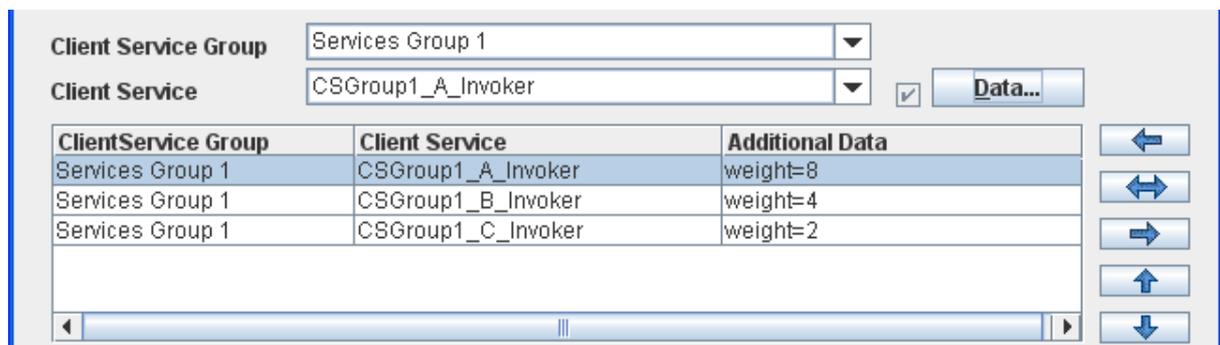


Abbildung 18 - Client Services mit Zuordnungsattributen

Dispatcher Konfiguration und Statistiken

Zuordnungsattribute werden zur Laufzeit vom Dispatcher an die externe Rule Engine und die FRAG Rule Engine übergeben und können dort zur Entscheidungsfindung verwendet werden. Siehe auch Kapitel 6.4, Methode RBDService featureList, feature ".<additional>".

4.4.3 Dialog Rule Script Definition

In diesem Dialog werden Rule Scripts und Rule Templates definiert. Er bietet überdies die Möglichkeit, die Scriptauswertung zu simulieren. Der Dialog ist in Abbildung 19 dargestellt und gliedert sich in drei Teile: die Textfelder für das FRAG Rule Script selbst, ein Textfeld für die Eingabe von simulierten Incoming Request Parametern sowie dem Ausgabefeld, in dem die Ergebnisse von Script-Simulation angezeigt werden.

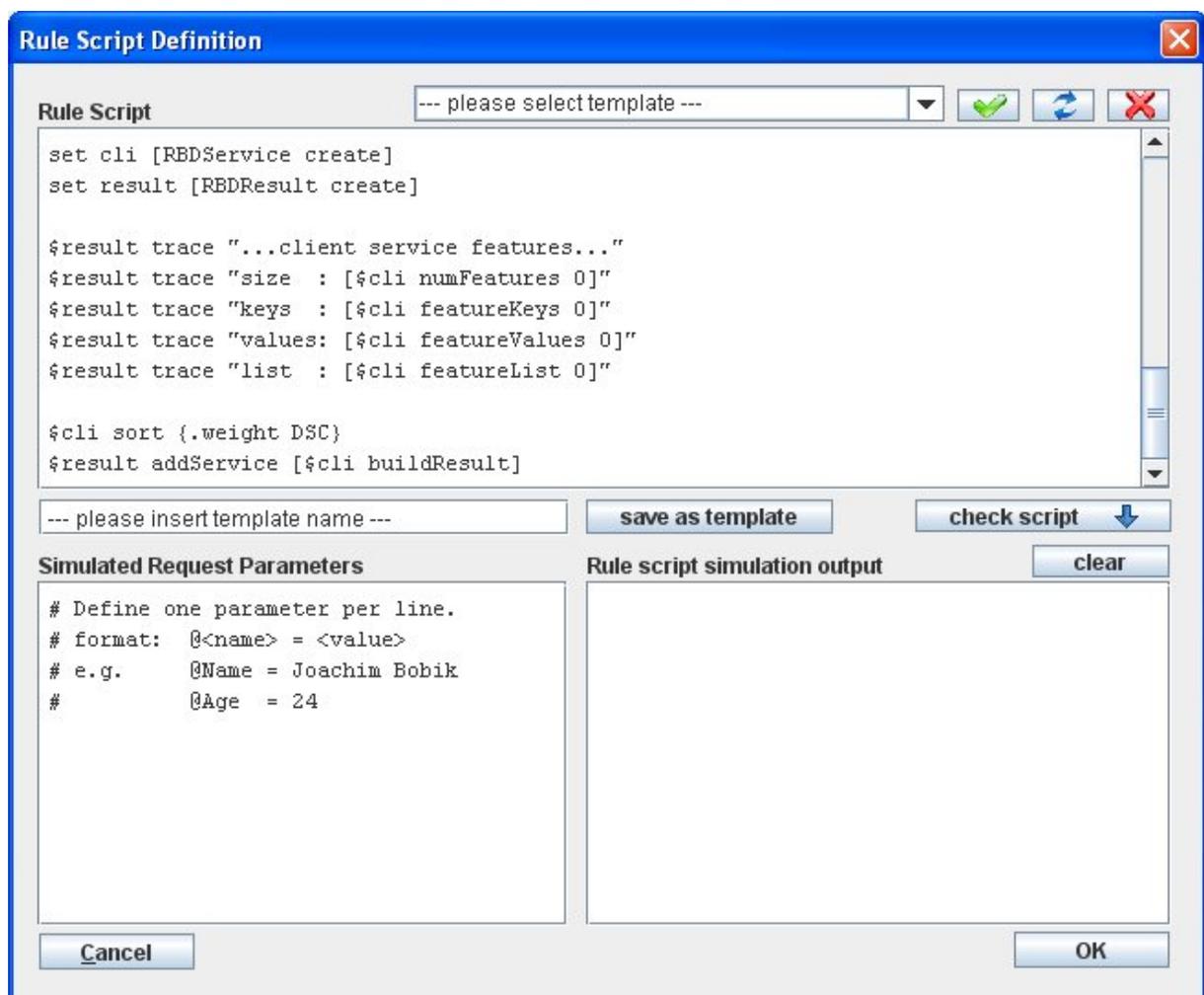


Abbildung 19 - Dialog Rule Script Definition

Im oberen Textfeld wird das FRAG Rule Script eingegeben. In Abbildung 19 ist ein einfaches Script dargestellt, in dem die Menge der für den Incoming Request definierten Client Services nach dem Zuordnungsattribut `.weight` absteigend sortiert wird.

Dispatcher Konfiguration und Statistiken

Anmerkung: Zuordnungsattribute können in der Rule Engine bzw. im Rule Script durch einen vorangegangenen Character '.' (dot) referenziert werden.

Durch Betätigen der Schaltfläche  wird das Script aus dem Eingabefeld gelesen und an den Dispatcher zur Auswertung übergeben. Am Dispatcher wird für die Auswertung exakt dieselbe Verarbeitung durchgeführt, wie später zur Laufzeit. Daher ist das Ergebnis aussagekräftig - unerwartete Ergebnisse können so schnell identifiziert und das Script richtiggestellt werden.

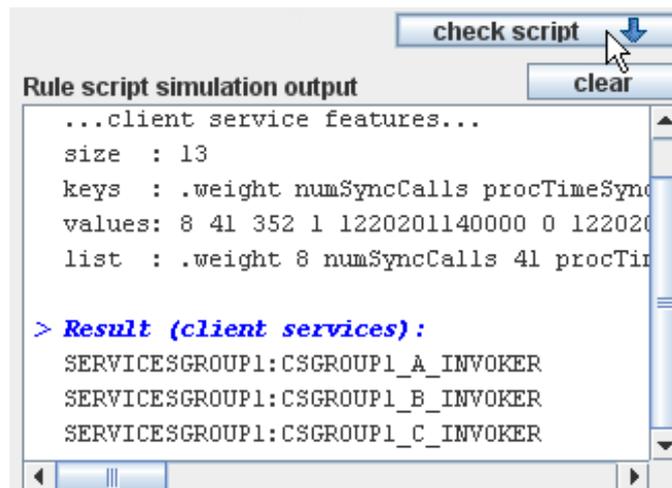


Abbildung 20 - Rule Script Auswertung

In Abbildung 20 ist der Inhalt des Outputfeldes nach Betätigen von  dargestellt. Es werden alle Informationen im Outputbereich dargestellt, welche im Script mit der Methode trace in das Ergebnis aufgenommen worden sind, zusammen mit dem eigentlichen Resultat des Scripts (in unserem Beispiel die 3 Services, absteigend sortiert).

Hier noch einmal das Script ...

```

set cli [RBDService create]
set result [RBDResult create]

$result trace "...client service features..."
$result trace "size : [$cli numFeatures 0]"
$result trace "keys : [$cli featureKeys 0]"
$result trace "values: [$cli featureValues 0]"
$result trace "list : [$cli featureList 0]"

$cli sort {.weight DSC}
$result addService [$cli buildResult]

```

...sowie das Ergebnis der Simulation

```

> Traced messages:
...client service features...
size : 13
keys : .weight numSyncCalls procTimeSync callOrder lastInit
numFiFoCalls lastCall numPending numCalls
numAsyncCalls isActive numFails procTimeAsync

```

Dispatcher Konfiguration und Statistiken

```

values: 8 41 352 1 1220201140000 0 1220201618000 1 59 18
      true 0 95512
list  : .weight 8 numSyncCalls 41 procTimeSync 352 callOrder
      1 lastInit 1220201140000 numFiFoCalls 0 lastCall
      1220201618000 numPending 1 numCalls 59 numAsyncCalls
      18 isActive true numFails 0 procTimeAsync 95512

> Result (client services):
SERVICESGROUP1:CSGROUP1_A_INVOKER
SERVICESGROUP1:CSGROUP1_B_INVOKER
SERVICESGROUP1:CSGROUP1_C_INVOKER

```

Das Textfeld *Simulated Request Parameters* kann verwendet werden, um Parameter zu simulieren, welche später vom Incoming Request an den Dispatcher übermittelt werden. Da das Rule Script seine Entscheidung auch auf Basis dieser Parameter treffen können soll, müssen sie für die Simulation ebenfalls berücksichtigt werden.

In Abbildung 21 ist dargestellt, wie solche Parameter spezifiziert werden können und wie die Ergebnisse im Output dargestellt werden.

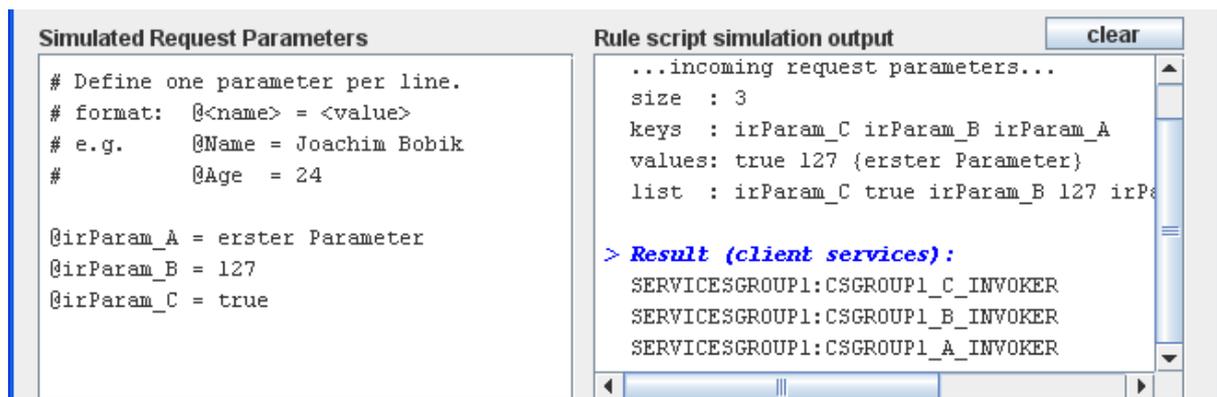


Abbildung 21 - Rule Script mit simulierten Request Parametern

Hier wieder das verwendete Script ...

```

set cli [RBDService create]
set inc [RBDIncoming create]
set result [RBDResult create]

$result trace "...incoming request parameters..."
$result trace "size : [$inc numParams]"
$result trace "keys  : [$inc paramKeys]"
$result trace "values: [$inc paramValues]"
$result trace "list  : [$inc paramList]"

$cli sort {.weight ASC}
$result addService [$cli buildResult]

```

Dispatcher Konfiguration und Statistiken

...sowie das Ergebnis der Simulation

```
> Traced messages:
...incoming request parameters...
size : 3
keys : irParam_C irParam_B irParam_A
values: true 127 {erster Parameter}
list : irParam_C true irParam_B 127 irParam_A {erster
Parameter}

> Result (client services):
SERVICESGROUP1:CSGROUP1_C_INVOKER
SERVICESGROUP1:CSGROUP1_B_INVOKER
SERVICESGROUP1:CSGROUP1_A_INVOKER
```

Wie zu sehen ist, wurden die Parameter übernommen und können im Rule Script verwendet werden. (Die Client Services wurden in diesem Beispiel nach dem Zuordnungsattribute `.weight` aufsteigend sortiert.)

Verwendung von Templates:

Um Rule Scripts nicht für jeden Incoming Request von Grund auf neu eingeben zu müssen, wurde die Möglichkeit geschaffen, Templates zu definieren, welche später bei der Definition von Rule Scripts verwendet werden können.

Ein Template wird angelegt, indem es im Textfeld für Rule Scripts eingegeben wird. Danach wird im Eingabefeld unter dem Textfeld ein Name für das Template angegeben und das Template mittels Schaltfläche **save as template** in der Datenbank abgelegt.



Abbildung 22 - Anlage eines Script Template

Ab diesem Zeitpunkt steht es in der Combobox am oberen Ende des Script-Eingabefeldes als Template zur Verfügung (Abbildung 23).

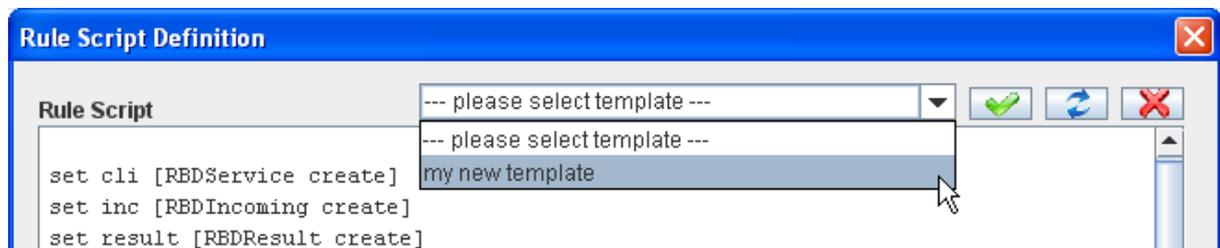


Abbildung 23 - Verwendung eines Script Template

Durch die Schaltfläche Use  wird das gewählte Template in das Scriptfeld kopiert. Durch die Operation Update  wird das gewählte Template mit dem Inhalt des Scriptfeldes überschrieben. Delete  löscht das gewählte Template aus der Datenbank.

Dispatcher Konfiguration und Statistiken

4.5 Statistics Client

Der Statistics Client dient der Anzeige der vom Dispatcher geführten Statistiken für Incoming Requests und Client Services. Der Dialog ist in Abbildung 24 dargestellt. Beim Start der Applikation werden Serveradresse und Port des Dispatchers als Startup-Parameter übergeben. Diese werden in der Kopfzeile des Dialoges angezeigt.

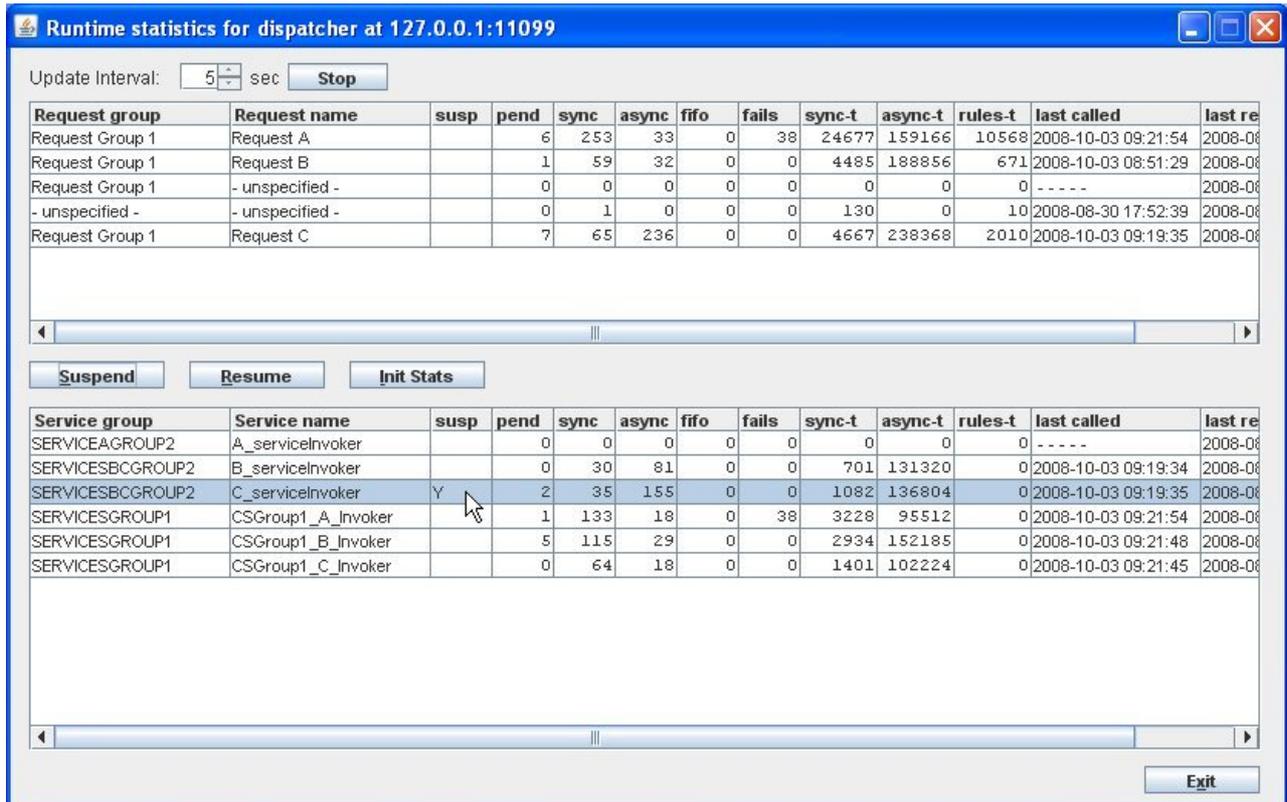


Abbildung 24 - Dialog Statistics Client

In der oberen Tabbox werden die Incoming Requests dargestellt, in der unteren Tabbox alle bekannten Client Services. Die Tabboxspalten haben die in Tabelle 14 beschriebenen Bedeutungen, diese sind für Incoming Requests und Client Services ident.

Tabbox-Spalte	Beschreibung
susp	gibt an ob das Objekt derzeit von der Verarbeitung ausgeschlossen ist (Eintrag 'Y') oder nicht (kein Eintrag) - siehe Beispiel in unterer Tabbox
pend	gibt die Anzahl der jener Objekte an, deren Verarbeitung asynchron läuft, aber noch nicht beendet ist
sync	gibt die Anzahl der synchronen Verarbeitungen seit der letzten Initialisierung an
async	wie oben für asynchrone Verarbeitungen
fifo	wie oben für fire&forget Verarbeitungen (analog non-managed asynchronous calls)

Dispatcher Konfiguration und Statistiken

Tabbox-Spalte	Beschreibung
fails	gibt an, wie viele Verarbeitungen seit der letzten Initialisierung nicht erfolgreich abgeschlossen werden konnten
sync-t	gibt die Gesamtzeit für alle synchronen Verarbeitungen seit der letzten Initialisierung in Millisekunden an.
async-t	wie oben für asynchrone Verarbeitungen
rules-t	gibt die Gesamtzeit seit der letzten Initialisierung an, die für die Auswertung der FRAG Rule Scripts und in externen Rule Engines verbraucht worden ist
last called	der Timestamp des letzten Einganges eines Incoming Request bzw. des letzten Aufrufes eines Client Service
last reset	der Zeitpunkt der letzten Initialisierung eines Objektes

Tabelle 14 - Statistics Client - Tabboxspalten

Im oberen Teil des Dialoges kann ein Intervall in Sekunden eingetragen werden, in dem der Client die jeweils aktuellen Werte vom Dispatcher ermittelt, Drücken der Schaltfläche

Start startet die Verarbeitung und die Schaltfläche verwandelt sich in die Schaltfläche **Stop**. Drücken von Stop beendet das zyklische Abfragen des Dispatchers.

Incoming Requests und Client Services werden durch Drücken von **Suspend** von der Verarbeitung zurückgezogen und durch **Resume** wieder freigegeben. Über **Init Stats** werden alle Werte für einen Incoming Request oder ein Client Service initialisiert.

Performancemessungen

Aus den Daten über die Anzahl der seit der letzten Initialisierung durchgeführten Aufrufe je Client Service sowie über die Gesamtzeit, die im jeweiligen Client Service für die Verarbeitung in Summe benötigt worden ist, können Performanceaussagen je Client Service getroffen werden - in den FRAG Rule Scripts oder den externen Rule Engines kann beispielsweise entschieden werden, dass Incoming Requests bevorzugt an jene Client Services weitergeleitet werden, welche die schnellste durchschnittliche Verarbeitungszeit aufweisen. Diese Daten können natürlich mit anderen statistischen Informationen gekoppelt werden - beispielsweise mit der Anzahl der im Moment je Client Service offenen (pending) Anfragen.

Aus der gemessenen Zeit, welche zur Auswertung der FRAG Rule Scripts sowie in den externen Rule Engines verbraucht wird, können Optimierungsoptionen für die Regelerstellung abgeleitet werden. Beispielsweise kann es Fälle geben, in denen nicht das gesamte Regelwerk in einem FRAG Rule Script abgelegt werden muss, sondern performanceintensive Berechnungen in externe Rule Engines verlagert werden können. Diese externen Rule Engines haben den Vorteil, dass sie zur Laufzeit nicht interpretiert werden müssen und die Auswertung daher deutlich schneller ist.

Diese Auslagerung ist in der vorliegenden Version des Dispatchers bereits eingeschränkt möglich. Als Ergebnis einer Script-Auswertung kann angegeben werden, dass die weitere Verarbeitung von einer externen Rule Engine übernommen werden soll. Für eine professionelle Umsetzung ist aber der in Kapitel 10.4 diskutierte Einsatz von Rule Script Plugins nötig.

5 Dispatcher Laufzeit

In diesem Abschnitt werden die Laufzeitumgebung des Dispatchers und der Client für die Anzeige von statistischen Informationen beschrieben.

Die für die Verarbeitung zur Laufzeit verantwortlichen Komponenten wurden bereits in Kapitel 3.2 dargestellt. Hier wird nun auf die möglichen Abläufe eingegangen.

Es ist prinzipiell zu unterscheiden, ob die Verarbeitung eines Incoming Request synchron oder asynchron erfolgen soll. Der Dispatcher kann beide Arten von Anfragen verarbeiten. Derzeit wird die Entscheidung synchron/asynchron vom Requester getroffen, es wäre aber natürlich denkbar, diese Information bei der Pflege von Incoming Requests vorzugeben bzw. die Entscheidung durch die Rule Engines treffen zu lassen.

In den folgenden (vereinfachten) Sequenzdiagrammen ist die Arbeitsweise des Dispatchers dargestellt.

5.1 Synchrone Verarbeitung

Im synchronen Fall wartet der Requester, bis das Client Service seine Verarbeitung beendet hat.

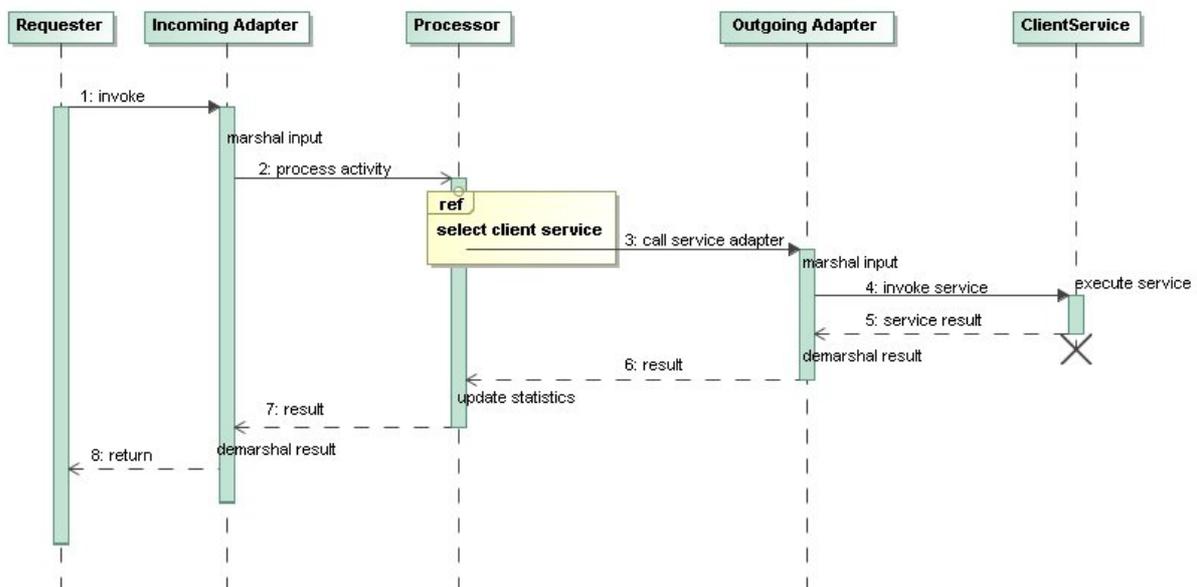


Abbildung 25 - Laufzeit synchron

Der Requester stößt die Verarbeitung durch Aufruf seines Incoming Adapter an. Dieser wandelt die erhaltenen Inputdaten in das für den Requester verständliche Format um und startet den Dispatcher (genauer: die Processor Komponente des Dispatchers).

Der Dispatcher ermittelt die passenden Client Services (siehe Kapitel 5.3) und ruft den Outgoing Adapter des bevorzugten Client Service auf. (Nur in dem Fall, dass ein Client Service nicht aufgerufen werden kann, wird die Verarbeitung mit dem nächsten möglichen Client Service versucht.) Der Outgoing Adapter wandelt die erhaltenen Daten in das für sein

Client Service verständliche Format um und startet die Verarbeitung synchron. Nach Beendigung der Verarbeitung werden die Ergebnisse an den Outgoing Adapter übermittelt, dieser formatiert und retourniert sie an den Dispatcher.

Der Dispatcher ermittelt verschiedene statistische Informationen (beispielsweise die Laufzeit des Service) und retourniert die Ergebnisse des Client Service seinerseits an den Incoming Adapter des Requesters. Hier werden die Daten in das für den Requester verständliche Format konvertiert und sodann an diesen übergeben.

Die beschrittenen Kommunikationswege für synchrone Verarbeitungen und die zum Einsatz kommenden Komponenten sind in Abbildung 4 des Kapitels 3.2 durch eine blaue 1 markiert.

5.2 Asynchrone Verarbeitung

Im asynchronen Fall wird an den Requester im ersten Schritt nur der Status übermittelt, ob die Anfrage an ein passendes Client Service erfolgreich weitergegeben werden konnte. Diese Information wird über alle oben beschriebenen Zwischenstufen an den Requester retourniert. Das (de)marshalling von Daten ist hier nicht mehr dargestellt.

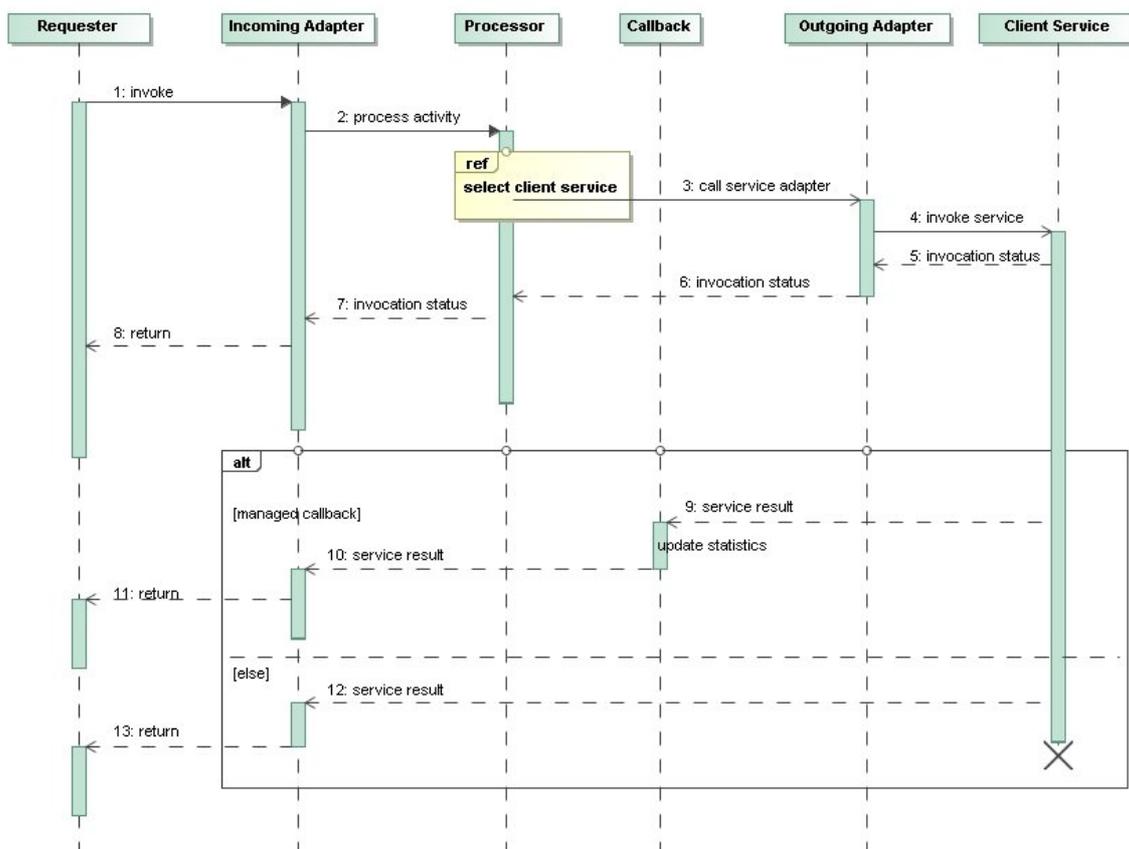


Abbildung 26 - Laufzeit asynchron

Beim Aufruf des Dispatchers wird vom Requester zusätzlich zu den normalen Inputparametern eine Callback-Methode spezifiziert, welche vom Client Service (bzw. dem Outgoing Adapter) nach Beendigung seiner Verarbeitung aufgerufen werden soll.

Hier sind nun zwei Arten von Asynchronität zu unterscheiden:

5.2.1 Managed Callbacks

In diesem Fall definiert der Requester, dass der Dispatcher statistische Daten über den Aufruf Request sammeln soll. Der vom Requester spezifizierte Callback wird in diesem Fall vom Dispatcher durch einen eigenen Callback ersetzt und dieser als aufzurufende Methode an das Client Service übermittelt.

Ist das Client Service beendet, wird diese Callbackmethode angestoßen und der Dispatcher ist in der Lage, die relevanten statistischen Daten über den Aufruf zu speichern. Diese können in weiteren Verarbeitungen als Basis für die Entscheidungen der Rule Engines dienen.

Danach wird der ursprüngliche Callback des Requesters mit den Ergebnissen des Client Service aufgerufen. Die beschriebenen Kommunikationswege und die zum Einsatz kommenden Komponenten sind in Abbildung 4 des Kapitels 3.2 durch eine blaue 2 markiert.

5.2.2 Non-managed Callbacks

In diesem Fall verzichtet der Requester darauf, dass der Dispatcher statistische Daten über den Aufruf sammelt. Der vom Requester spezifizierte Callback wird vom Dispatcher unverändert an das Client Service übergeben, der Dispatcher ist nicht in der Lage zu erkennen, ob bzw. wann das angestoßene Client Service seine Verarbeitung beendet hat und ob diese erfolgreich war. Siehe auch den durch eine blaue 3 markierten Kommunikationsweg in Abbildung 4 in Kapitel 3.2. Der erzielte Performancegewinn wird dadurch erkauft, dass diese Informationen in weiterer Folge nicht als Kriterien für die Auswertung von Regeln zur Verfügung stehen - beispielsweise kann das Client Service mit der kürzesten durchschnittlichen Verarbeitungszeit so verständlicherweise nicht ermittelt werden.

5.3 Auswahl der Client Services

Sowohl bei synchroner als auch bei asynchroner Verarbeitung läuft die Auswahl des am besten geeigneten Client Service gleich ab. Der Prozess ist in Abbildung 27 in Form eines vereinfachten Sequenzdiagramms dargestellt.

Nach Aufruf des Dispatchers durch den Incoming Adapter werden die konfigurierten Daten für den Incoming Request und die mit dem Request assoziierten Client Services aus der Datenbank ermittelt. Im nächsten Schritt werden die statistischen Daten für den Incoming Request und jedes der Client Services gelesen.

Wurde beim Request ein FRAG Rule Script definiert, so wird dieses aus der Datenbank ermittelt und an die FRAG Rule Engine zur Auswertung übergeben. Die Engine retourniert entweder eine Liste von Client Services (in diesem Fall ist das Auswahlverfahren hier beendet) oder die ID einer externen Rule Engine, welche für die Auswahl der geeigneten Client Services zuständig ist.

In letzterem Fall - und auch in Fällen, in denen kein FRAG Rule Script und nur eine externe Rules Engine definiert ist - wird die Aufrufinformation aus der Datenbank ermittelt und die Engine angestoßen um die Menge der passenden Client Services zu ermitteln.

Dispatcher Laufzeit

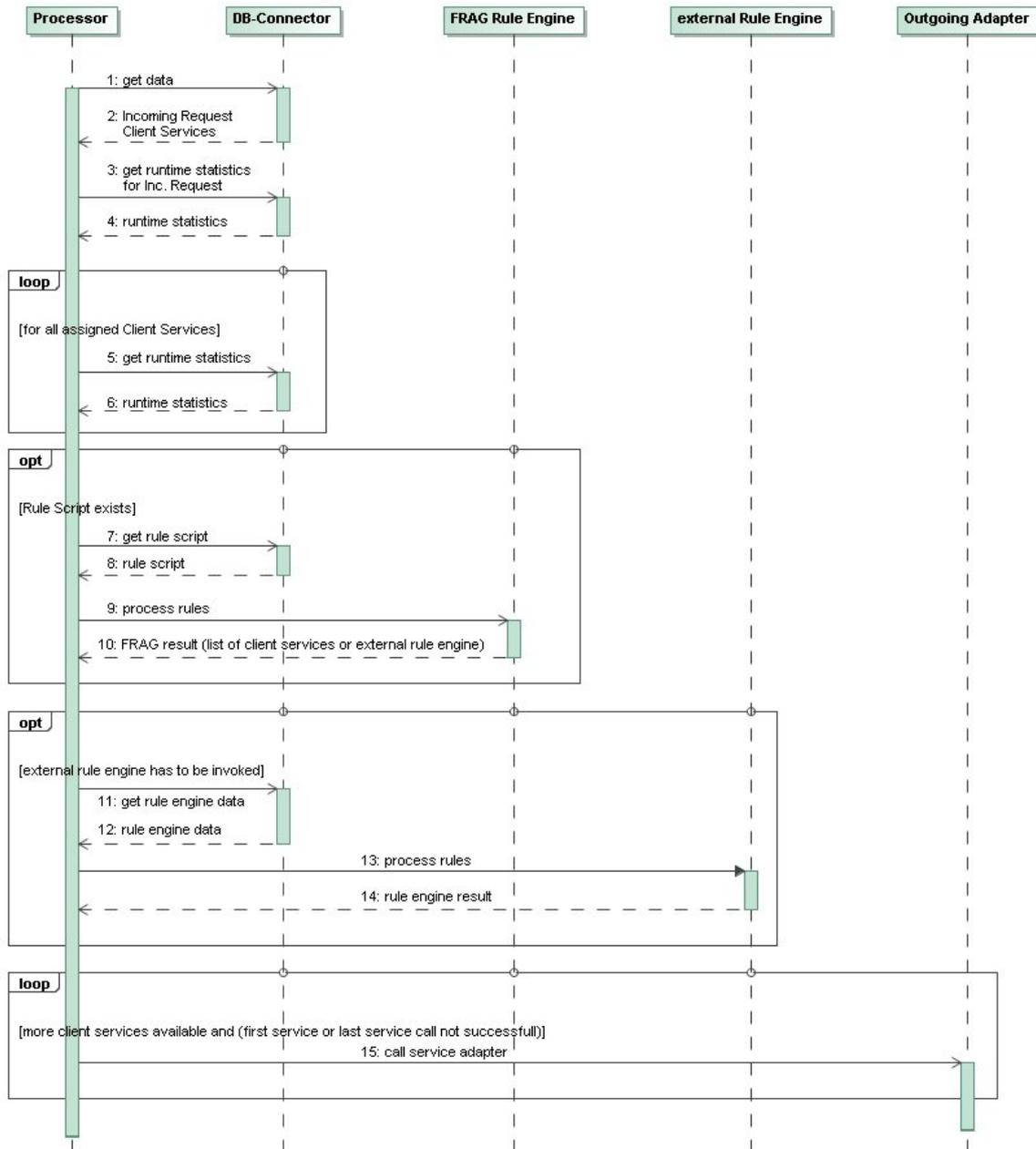


Abbildung 27 - Auswahl Client Service

Wurde für einen Incoming Request weder ein FRAG Rule Script noch eine externe Rule Engine definiert, so wird die bei der Konfiguration definierte Reihenfolge der zugeordneten Client Services herangezogen, um die Aufrufpriorität der Services zu bestimmen.

Client Services werden der Reihe nach aufgerufen, allerdings nur bis das erste Service einen erfolgreichen Aufruf meldet - ein Service auf der Liste kommt also nur zum Einsatz, wenn es entweder das Service mit der höchsten Priorität ist oder alle vorangegangenen Serviceaufrufe aufgrund eines technischen Fehlers fehlgeschlagen sind.

5.4 Tracing am Applikationsserver

Die Ausgabe von Tracingmessages am Applikationsserver wird über die Klasse `RBDServer.server.common.Config` gesteuert. Es wird hier definiert, ob nur Fehlermeldungen, Fehler und Warnungen oder zusätzlich auch Informationsmeldungen ausgegeben werden sollen.

Im Beispiel in Abbildung 28 ist ein Teil der Tracing-Ausgaben dargestellt, die bei einem einfachen, synchronen Aufruf des Dispatchers mit dem höchsten Tracelevel angezeigt werden.

```

PB> processActivity(403): ---RBD processor starting ----
PB> processActivity(404): processing request [REQUESTGROUP1-REQUESTA]
BI> <init>(52): looking up service [RBDDBConnectorBean/remote] at jnp://localhost:11099
BI> method to invoke = [procGetRequest]
DB> procGetRequest(1833): increg created: Request Group 1
DB> getRuntimeStats(1544): trying to read runtime stats for link_row_id 1 linktype [REQ]
PB> got request [REQUESTGROUP1;REQUESTA]
BI> method to invoke = [procGetClientService]
DB> getRuntimeStats(1544): trying to read runtime stats for link_row_id 1545006101 linktype [CLS]
BI> method to invoke = [procGetClientServiceGroup]
DB> procGetClientServiceGroup(1903): reading CS group [SERVICESGROUP1]
BI> method to invoke = [procGetClientService]
DB> getRuntimeStats(1544): trying to read runtime stats for link_row_id 1121079295 linktype [CLS]
BI> method to invoke = [procGetClientServiceGroup]
DB> procGetClientServiceGroup(1903): reading CS group [SERVICESGROUP1]
BI> method to invoke = [procGetClientService]
DB> getRuntimeStats(1544): trying to read runtime stats for link_row_id 1993079846 linktype [CLS]
BI> method to invoke = [procGetClientServiceGroup]
DB> procGetClientServiceGroup(1903): reading CS group [SERVICESGROUP1]
BI> method to invoke = [getRuleScript]
DB> getRuleScriptRows(1515): reading rule script for ID [47LB58CESNDJXCR5]
DB> getRuleScript(1486): getting rule script: 2 lines read.
PB> processActivity(503): after FRAG: Services: 0 Engine: 1 Errors: 0
PB> processActivity(561): after FRAG: rule engine = WEIGHTEDRREVALUATE
PB> Invoking rules engine WEIGHTEDRREVALUATE
BI> method to invoke = [procGetRuleEngine]
DB> procGetRuleEngine(2017): reading rule engine for ID [WEIGHTEDRREVALUATE]
BI> <init>(52): looking up service [RulesFillWeightsBean/remote] at jnp://localhost:11099
PB> calling rule engine [WeightedRR.evaluate] at jnp://localhost:11099, Invoker = RulesFillWeights
BI> method to invoke = [evaluate]
PB> Call order determined by the rule engine:
PB> 1. [SERVICESGROUP1]-[CSGROUP1_A_INVOKER]
PB> Starting invocations...
PB> 1. SERVICESGROUP1-CSGROUP1_A_INVOKER
BI> method to invoke = [procGetCSInvocationContext]
DB> procGetCSInvocationContext(2047): reading invocation context for service [CSGROUP1_A_INVOKER]
BI> <init>(52): looking up service [MIFE1UniversalBean/remote] at jnp://localhost:11099
PB> no callback info found for activity [REQUESTGROUP1-REQUESTA]
PB> Client service is invoked synchronously
PB> processActivity(823): I'm invoking service CSGroup1_A_Invoker of Bean MIFE1UniversalBean/remot
BI> method to invoke = [CSGroup1_A_Invoker]
OutAdapter> MIFE1UniversalBean.CSGroup1_A_Invoker invoked for service [1]
OutAdapter> sessionId= 1223048567436; activities= 1
OutAdapter> MIFE1UniversalBean.CSGroup1_A_Invoker: no callback info found for correlationID REQUES
OutAdapter> -processing synchronous call
OutAdapter> calling //127.0.0.1:1099/MIFE0001; params contains 2 elements.
PB> success.
PB> updating runtime statistics (service)...
BI> method to invoke = [procUpdRuntimeStats]
DB> procUpdRuntimeStats(1594): trying to read (for update) runtime stats (row 1777910101)
PB> updating runtime statistics (request)...
BI> method to invoke = [procUpdRuntimeStats]
DB> procUpdRuntimeStats(1594): trying to read (for update) runtime stats (row 281474998)

```

Abbildung 28 - Tracing

Mit diesen Informationen können eventuelle Fehler leicht nachvollzogen und schnell lokalisiert werden.

6 Regeldefinition

Wie bereits in den vorangegangenen Abschnitten beschrieben, werden im Zuge der Konfiguration für jeden Incoming Request ein oder mehrere Client Services definiert, welche die Anfrage abarbeiten können. Der Dispatcher muss nun entscheiden, welches der potenziell möglichen Services er auswählt sowie die Prioritäten dieser Services festlegen.

Der Dispatcher trifft diese Entscheidung nicht selbst, sondern delegiert sie entweder an eine externe Rule Engine oder an die FRAG Rule Engine, welche das am besten passende Client Service durch Auswertung eines FRAG Rule Scripts ermitteln.

Diese beiden Alternativen werden in der Folge beschrieben.

6.1 Regeldefinition durch Rule Engines

Um für den Dispatcher als Rule Engine nutzbar zu sein, muss eine Enterprise Java Bean oder ein Webservice das in Tabelle 15 beschriebene Interface unterstützen. Siehe Kapitel 4.3 für eine Beschreibung des Reflection Interface.

Interface-Methode	Beschreibung
<code>getEngineDescription()</code>	Liefert die Beschreibung der Rule Engine in der Form [Engine-Name, Engine-Beschreibung]. Diese Information wird im Zuge der Konfiguration im Dialog zur Definition von Rule Engines angezeigt. Siehe Kapitel 4.3.
<code>getRBDRuleServices()</code>	Liefert eine Liste von Namen und Beschreibungen jener Methoden, welche die Regellogik implementieren. Beispiel: [eval_RRob; Round Robin Processing] [eval_Date; Date/Time based Processing]
<interface-method 1> : : <interface-method n>	die mittels <code>getRBDRuleServices()</code> gelieferten Methoden Im obigen Beispiel müssen die Methoden <code>eval_RRob()</code> und <code>eval_Date()</code> für den Dispatcher zugreifbar sein.

Tabelle 15 - Rule Engine Interface

Die Methoden, welche vom Dispatcher für die Regelauswertung angesprochen werden, besitzen das im folgenden Codesegment dargestellte Interface.

```
public RuleEngineResult evaluate(Hashtable<String, Object>
                                statsIncoming, IncomingMessage paramIncoming,
                                Vector<Hashtable<String, Object>> paramCliService)
    throws RBDEException;
```

Die Parameter haben folgende Bedeutung:

Regeldefinition

Input-Parameter Key / .Attribut	Beschreibung
statsIncoming	die statistischen Werte für den Incoming Request
RBDCrit.LAST_REQUEST	der Timestamp des letzten Einganges eines solchen Incoming Request
RBDCrit.REQPENDING	die Anzahl der noch nicht fertig verarbeiteten Requests
RBDCrit.NUM_REQUESTS	die Anzahl der bisherigen Eingänge des Request seit der letzten Initialisierung der Statistikdaten
RBDCrit.REQS_SYNC	wie oben für synchrone Requests
RBDCrit.REQS_ASYNC	wie oben für asynchrone Requests
RBDCrit.REQS_FIFO	wie oben für fire & forget Requests
RBDCrit.REQS_FAILS	die Anzahl der bislang nicht erfolgreich verarbeiteten Requests
RBDCrit.REQPROCT_SYNC	die bisherige Verarbeitungsdauer aller synchronen Verarbeitungen in Millisekunden
RBDCrit.REQPROCT_ASYNC	wie oben für asynchrone Requests
RBDCrit.REQLAST_INIT	der Timestamp der letzten Initialisierung aller Werte für den aktuellen Incoming Request
RBDCrit.REQACTIVE	gibt an, ob der Request im Moment verarbeitet werden kann
paramIncoming	die Parameter des Incoming Request
.soapHeader	liefert den Header der SOAP-Request Message als String
.soapBody	liefert den Body der SOAP-Request Message als String
.reqGroupID	die eindeutige ID der Request Gruppe
.reqGroupName	die eindeutige externe Bezeichnung der Request Gruppe des Incoming Request
.requestID	die eindeutige ID des Request
.requestName	die eindeutige externe Bezeichnung des Incoming Request
.Parameters	die ursprünglichen Parameter des Incoming Request
paramCliService[i]	die Daten der Client Services (Vector)
RBDCrit.SGROUP_ID	die ID der Client Service Gruppe
RBDCrit.SGROUP_NAME	der bei der Konfiguration vergebene Gruppenname
RBDCrit.SGROUP_NAMEEX	der originale Gruppenname, wie vom Client Service spezifiziert
RBDCrit.SERVICE_ID	die ID des Client Service
RBDCrit.SERVICE_NAME	der bei der Konfiguration vergebene Servicename
RBDCrit.SERVICE_NAMEEX	der originale Servicename, wie vom Client Service spezifiziert
RBDCrit.CALLORDER	die konfigurierte Reihenfolge des Client Service für den aktuellen Incoming Request
RBDCrit.LASTCALL	der Timestamp des letzten Aufrufes des Service

Regeldefinition

Input-Parameter Key / .Attribut	Beschreibung
RBDCrit.PENDING	die Anzahl der noch nicht fertig verarbeiteten Serviceaufrufe
RBDCrit.SERVICE_CALLS	die Anzahl der bisher getätigten Aufrufe des Service seit der letzten Initialisierung der Statistikdaten
RBDCrit.CALLS_SYNC	wie oben für synchrone Aufrufe
RBDCrit.CALLS_ASYNC	wie oben für asynchrone Aufrufe
RBDCrit.CALLS_FIFO	wie oben für fire & forget Aufrufe
RBDCrit.SERVICE_FAILS	die Anzahl der bislang nicht erfolgreich verarbeiteten Serviceaufrufe
RBDCrit.PROCT_SYNC	bisherige Verarbeitungsdauer aller synchronen Verarbeitungen in Millisekunden
RBDCrit.PROCT_ASYNC	wie oben für asynchrone Verarbeitungen
RBDCrit.LAST_INIT	der Timestamp der letzten Initialisierung aller Werte für das aktuelle Client Service
RBDCrit.ACTIVE	gibt an, ob das Client Service im Moment des Aufrufes aktiv ist
.<additional>	weitere Zuordnungsparameter, auf welche durch den Parameternamen mit vorangestelltem '.' (dot) zugegriffen werden kann.

Tabelle 16 - Rule Engine Parameter

Anhand dieser Parameter kann die Rule Engine ihre Entscheidung für eines der im Vector paramCliServices zur Verfügung stehenden Services treffen. Werden mehrere Services ausgewählt, so wird durch die Reihenfolge, in welcher diese an den Dispatcher geliefert werden, die Priorität der Aufrufe festgelegt.

Die Methoden, mit denen die Ergebnisse von der Rule Engine definiert werden können, sind in Tabelle 17 erklärt.

Ergebnis-Parameter .Methode	Beschreibung
RuleEngineResult	die Ergebnisse der Rule Engine
.setTrace()	fügt einen neuen Trace-String zu den vorhandenen Traces hinzu
.setService()	fügt ein neues Client Service in die Liste der vorhandenen Client Services ein
.setEngine()	definiert die Rule Engine, welche die weitere Verarbeitung übernimmt
.setError()	setzt Fehlercode und Fehlerbezeichnung

Tabelle 17 - Rule Engine Ergebnisdefinition

6.2 Regeldefinition mit FRAG

FRAG ist eine dynamische Programmiersprache, mit der u.a. Domain Specific Languages einfach erstellt werden können. Siehe [Zdun 2006] für eine Beschreibung der Sprache. FRAG ist in Java geschrieben und leicht durch eigene Objekte zu erweitern. Dieser Ansatz wird hier genutzt, um einige Objekte in FRAG einzuführen, welche der Dispatcher für seine Entscheidungen nutzen kann.

Dabei handelt es sich um diese, in den folgenden Unterkapiteln im Detail beschriebenen, Objekte:

Objekt	Beschreibung
RBDTime	verwendet für Date/Time-Manipulationen
RBDIncoming	erlaubt Zugriff auf die Daten von Incoming Requests
RBDService	erlaubt Zugriff auf die Daten von Client Services
RBDResult	dient der Definition von Ergebnissen der Regelauswertung

Tabelle 18 - FRAG Erweiterung

6.3 Date/Time Manipulationen mittels *RBDTime*

Das Objekt *RBDTime* wird für Date/Time-Manipulationen verwendet. Die folgenden Methoden werden angeboten:

<i>format</i>	wandelt einen Timestamp in einen Date/Time-String um
<i>parse</i>	wandelt einen Date/Time-String in einen Timestamp um
<i>now</i>	liefert den aktuellen Timestamp in Millisekunden seit 01-Jan-1970

Tabelle 19 - *RBDTime* Methoden

Die Verwendung dieser Methoden wird nun beschrieben.

RBDTime format

Die Methode *RBDTime format* wandelt einen als Anzahl Millisekunden seit 01-Jan-1970 00:00:00:000 vorliegenden Timestamp in einen String um, wobei das Format durch die aus der Java-Klasse `SimpleDateFormat` bekannten Patterns vorgegeben werden kann. Wird der Timestamp nicht angegeben, so wird die aktuelle Uhrzeit zum Zeitpunkt der Durchführung verwendet.

Syntax	<code>RBDTime format <date/time pattern> ?date/time in milliseconds?</code>
Beispiel	<code>puts "time: [RBDTime format "yyyy.MM.dd HH:mm:ss" 1008846420000]"</code>
Ergebnis	<code>"time: 2001.12.20 12:07:00"</code>

Tabelle 20 - *RBDTime format*

Am Ende dieses Kapitels finden sich weitere Beispiele für die Verwendung der *RBDTime*-Methoden.

RBDTime parse

Die Methode *RBDTime parse* wandelt eine als String vorliegende Datums/Zeitangabe in die Anzahl der Millisekunden seit 01-Jan-1970 00:00:00:000 um. Ergebnis ist ein Long Value. Für nicht spezifizierte Jahres/Monats/Tages-Teile des Patterns wird immer der entsprechende Wert des aktuellen Datums eingesetzt.

Syntax	<code>RBDTime parse <date/time string> <date/time pattern></code>
Beispiel	<code>puts "ms: [RBDTime parse "22.07 12:47:13" "dd.MM HH:mm:ss"]"</code>
Ergebnis	<code>"ms: 1216723633000"</code>

Tabelle 21 - *RBDTime parse*

RBDTime now

Die Methode *RBDTime now* liefert den aktuellen Timestamp als Anzahl der Millisekunden seit 01-Jan-1970 00:00:00:000 in einem Long Value.

Syntax	<code>RBDTime now</code>
Beispiel	<code>puts "now in ms: [RBDTime now]"</code>
Ergebnis	<code>"now in ms: 1216723633000"</code>

Tabelle 22 - RBDTime now

RBDTime Beispiel - Date/Time Formatierung

Im folgenden FRAG Rule Script werden einige Beispiele für die Formatierung von Datums- und Zeitangaben angeführt

```
set mytime [RBDTime format "HH:mm:ss:SSS"]
puts "time is $mytime"

puts "1: [RBDTime format "yyyy.MM.dd HH:mm:ss" 1008846420000]"
puts "2: [RBDTime format "yyyy.MM.dd HH:mm:ss"] \ (now\)"

set mytime [RBDTime parse "22.07 12:47:13" "dd.MM HH:mm:ss"]
puts "\nparsed time \ (long\ ) is $mytime"
puts "3: [RBDTime format "yyyy.MM.dd HH:mm:ss:SSS" $mytime]"

set mytime ($mytime - 1000000000)
puts "\nnew time \ (long\ ) is $mytime"
puts "4: [RBDTime format "yyyy.MM.dd HH:mm:ss:SSS" $mytime]"
puts "5: [RBDTime format "yyyy.MM.dd HH:mm:ss:SSS"
      [RBDTime now]]"

set date [RBDTime format "EEE, MMM d, 'yy"]
puts "\n6: date is $date"
```

Das dargestellte FRAG Rule Script liefert den folgenden Output

```
time is 11:47:52:710
1: 2001.12.20 12:07:00
2: 2008.03.25 11:47:52 (now)

parsed time (long) is 1216723633000
3: 2008.07.22 12:47:13:000

new time (long) is 1215723633000
4: 2008.07.10 23:00:33:000
5: 2008.03.25 11:47:52:710

6: date is Di, Mrz 25, '08
```

RBDTime Beispiel - Entscheidungen aufgrund Datum und Uhrzeit

Im folgenden FRAG Rule Script werden einige weitere Beispiele für die Verwendung von *RBDTime* dargestellt.

```
set minstr "10:00"
set maxstr "12:00"
set mintime [RBDTime parse $minstr "HH:mm"]
set maxtime [RBDTime parse $maxstr "HH:mm"]

set acttime [RBDTime now]
puts "current time is [RBDTime format "HH:mm:ss:SSS" $acttime]"

if ($acttime < $maxtime) {
  puts "time is < $maxstr" }

if ($acttime > $mintime) {
  puts "time is > $minstr" }

if ( $acttime <= $maxtime) {
  if ( $acttime >= $mintime ) {
    puts "time is in >$minstr; $maxstr<"
  } else {
    puts "time is < $minstr" }
} else {
  puts "time is > $maxstr" }
```

Das obige FRAG Rule Script liefert den folgenden Output:

```
current time is 10:55:41:947
time is < 12:00
time is > 10:00
time is in >10:00; 12:00<
```

6.4 Zugriff auf Client Services mittels *RBDService*

Die konfigurierten und für den Dispatcher nutzbaren Client Services sind über das Objekt *RBDService* zugreifbar. Jedes Service besitzt neben seinen 'Parametern' eine Liste von 'Features', welche über die entsprechenden Methoden abgefragt werden können. Dabei handelt es sich zum einen um statistische Daten (Anzahl der bisher getätigten asynchronen Aufrufe, Timestamp des letzten Aufrufes,...) und andererseits um jene Daten, welche im Zuge der Konfiguration als Zuordnungsattribute für die Client Service Bindings eingepflegt worden sind.

Das Objekt *RBDService* bietet die folgenden Methoden an:

<i>paramList</i>	liefert eine Liste der Werte des angegebenen Parameters aller Client Services
<i>contains</i>	liefert die Information, ob ein bestimmtes Service existiert
<i>search</i>	sucht nach einem bestimmten Service
<i>getService</i>	liefert ein spezielles Service-Objekt
<i>restrict</i>	schränkt die Menge der über <i>RBDService</i> verarbeitbaren Service-Objekte ein
<i>numServices</i>	liefert die Anzahl der verfügbaren Services
<i>featureList</i>	liefert eine Liste aller Features eines Service zusammen mit ihren Ausprägungen
<i>featureKeys</i>	liefert eine Liste aller Features eines Service
<i>featureValues</i>	liefert eine Liste aller Feature-Ausprägungen eines Service
<i>numFeatures</i>	gibt die Anzahl der Features eines Service an
<i>hasFeature</i>	gibt an, ob ein spezielles Feature bei einem Service vorhanden ist
<i>getFeature</i>	ermittelt den Werte eines speziellen Feature
<i>sort</i>	sortiert die Liste der Client Services
<i>buildResult</i>	erzeugt Einträge für <i>RBDResult</i>

Tabelle 23 - *RBDService* Methoden

Die Verwendung dieser Methoden wird in der Folge beschrieben.

RBDSERVICE paramList

Bei Aufruf der Methode *RBDSERVICE paramList* wird die Liste der für den aktuellen Incoming Request verfügbaren Client Services durchlaufen und jeweils der Wert des spezifizierten Parameters ermittelt. Das Ergebnis wird in einer Liste zurückgeliefert.

Folgende Parameterwerte werden unterstützt:

groupID	die ID der Client Service Gruppe
groupName	der bei der Konfiguration vergebene Gruppenname
groupNameExt	der originale Gruppenname, wie vom Client Service spezifiziert
serviceID	die ID des Client Service
serviceName	der bei der Konfiguration vergebene Servicename
serviceNameExt	der originale Servicename, wie vom Client Service spezifiziert
Syntax	<code>\$<var> paramList <parameter></code>
Beispiel	<pre>set cli [RBDSERVICE create] puts "group-names: [\$cli paramList serviceName]"</pre>
Ergebnis	<pre>{service 1.1-name} {service 2.1-name} {service 2.2-name}</pre>

Tabelle 24 - RBDSERVICE paramList

Im Beispiel für *RBDSERVICE contains* wird eine weitere Anwendung der Methode gezeigt.

RBDSservice contains

Über die Methode *RBDSservice contains* kann ermittelt werden, ob in der Liste der Client Services ein Service mit bestimmten Ausprägungen existiert. Die gesuchten Ausprägungen werden in einer Liste von *key/value*-Paaren übergeben, wobei die *values* auch die aus Java bekannten *regular expressions* enthalten können - siehe auch `String.matches()` in Java.

Es kann nach allen Parametern gesucht werden, welche auch in der Methode *RBDSservice.paramList* verwendet werden können und zusätzlich nach dem Flag

`isActive` gibt an, ob das Client Service im Moment aktiv ist (`true`) und zur Verarbeitung eines Incoming Request herangezogen werden kann oder nicht (`false`).

Syntax	<code><var> contains {<parameter> <value>, ...}</code>
Beispiel	<p>es soll festgestellt werden, ob in der Menge der verfügbaren Client Services ein aktives Service existiert, dessen Gruppenname mit '2' endet.</p> <pre> set cli [RBDSservice create] puts "groups: [\$cli paramList groupName]" puts "active: [\$cli paramList isActive]" set ok [\$cli contains {groupName ".*?2" isActive "true"}] if {\$ok == true } { puts "active group found !" } </pre>
Ergebnis	<pre> groups: {name group 1} {name group 2} {name group 2} active: false false true active group found ! </pre>

Tabelle 25 - RBDSservice contains

RBDService search

Die Methode *RBDService search* arbeitet analog *RBDService contains*, nur wird als Ergebnis der Index des ersten Client Service geliefert, welches den spezifizierten Suchkriterien genügt. Die Kriterien entsprechen jenen von *RBDService contains*.

Syntax	<code>\$<var> search {<parameter> <value>, ...}</code>
Beispiel	<p>aus der Liste der Client Services soll der erste Eintrag gesucht werden, welcher aktiv sind und dessen Gruppenname mit '2' endet.</p> <pre> set cli [RBDService create] puts "groups: [\$cli paramList groupName]" puts "active: [\$cli paramList isActive]" set ok [\$cli contains {svcGroupName ".*?2" isActive "true"}] if {\$ok == true } { set idx [\$cli search {groupName ".*?2" isActive "true"}] puts "active group found, index is \$idx" puts "group data: [\$cli getService \$idx]" } </pre>
Ergebnis	<pre> groups: {name group 1} {name group 2} {name group 2} active: false false true active group found, index is 2" group data: [...]svcGroupName {name group 2} [...]isActive true </pre>

Tabelle 26 - RBDService search

RBDService getService

Die Methode *RBDService getService* liefert ein spezielles Client Service Objekt. Das Objekt wird über seinen Index adressiert, wird kein Index angegeben, so werden alle verfügbaren Client Service Objekte als Antwort geliefert.

Syntax	<code>\$<var> getService [<index>]</code>
Beispiel	siehe RBDService search

Tabelle 27 - RBDService getService

RBDService restrict

Die Methode *RBDService restrict* schränkt die Menge der verfügbaren Client Services auf jene Services ein, welche ein definiertes Set an Parameter-Ausprägungen aufweisen. Die gewünschten Ausprägungen werden wie bei der Methode *RBDService contains* als Liste von *key/value*-Paaren angegeben.

Syntax	<code>\$<var> restrict {<parameter> <value>, ...}</code>
Beispiel	<p>die Liste der Client Services soll auf jene Einträge eingeschränkt werden, welche inaktiviert wurden und deren Gruppenname mit '2' endet.</p> <pre>set cli [RBDService create] puts "groups: [\$cli paramList svcGroupName]" puts "active: [\$cli paramList isActive]" \$cli restrict {svcGroupName ".*?2" isActive "false"} puts "groups: [\$cli paramList svcGroupName]"</pre>
Ergebnis	<pre>groups: {name group 1} {name group 2} {name group 2} active: false false true groups: {name group 2}</pre>

Tabelle 28 - RBDService restrict

RBDService numServices

Die Methode *RBDService numServices* liefert die Anzahl der vorhandenen Client Services.

Syntax	<code>\$<var> numServices</code>
Beispiel	<p>Die Anzahl der inaktiven Client Services wird ausgegeben</p> <pre>set cli [RBDService create] \$cli restrict {isActive "false"} puts "number of inactive services: [\$cli numServices]"</pre>
Ergebnis	<pre>number of inactive services: 2</pre>

Tabelle 29 - RBDService numServices

RBDSservice featureList

Über die Methode *RBDSservice featureList* erhält man eine Liste aller Features und ihrer Ausprägungen eines Client Service. Das Service wird über seinen Index (beginnend mit 0 für das erste definierte Service) spezifiziert, folgende Features werden unterstützt:

callOrder	die konfigurierte Reihenfolge des Client Service für den aktuellen Incoming Request
lastCall	der Timestamp des letzten Aufrufes des Service
numCalls	die Anzahl der bisher getätigten Aufrufe des Service seit der letzten Initialisierung der Statistikdaten
numSyncCalls	wie oben für synchrone Aufrufe
numAsyncCalls	wie oben für asynchrone Aufrufe
numFiFoCalls	wie oben für fire & forget Aufrufe
numPending	die Anzahl der noch nicht fertig verarbeiteten Serviceaufrufe
numFails	die Anzahl der bislang nicht erfolgreich verarbeiteten Serviceaufrufe
procTimeSync	die bisherige Verarbeitungsdauer aller synchronen Verarbeitungen in Millisekunden
procTimeAsync	die bisherige Verarbeitungsdauer aller asynchronen Verarbeitungen in Millisekunden
lastInit	der Timestamp der letzten Initialisierung aller Werte für das aktuelle Client Service
isActive	gibt an, ob das Client Service im Moment des Aufrufes aktiv (d.h. nicht suspended) ist
.<additional>	weitere Features, welche im Zuge der Konfiguration als Zuordnungsattribute des Client Service zum aktuellen Incoming Requests definiert worden sind. Auf diese kann über den konfigurierten Featurenamen mit vorangestelltem '.' (dot) zugegriffen werden. Beispiel: das im Zuge der Konfiguration definierte Attribut <i>weight</i> ist hier über <i>.weight</i> zugreifbar.

Syntax	<code>\$<var> featureList <index></code>
Beispiel	Die Features des ersten definierten Service sollen angezeigt werden <pre>set cli [RBDSservice create] if ([\$cli numServices] > 0) { puts "features: [\$cli featureList 0]" } </pre>
Ergebnis	<pre>features: numSyncCalls 200 procTimeSync 12220 callOrder 1 lastInit 1218192452727 numFiFoCalls 20 lastCall 1218193652727 numPending 312 numCalls 233 numAsyncCalls 13 isActive false numFails 7 procTimeAsync 2000 .weight 8 </pre>

Tabelle 30 - RBDSservice featureList

RBDService featureKeys

Über die Methode *RBDService featureKeys* erhält man eine Liste aller Features eines Client Service. Für die Liste der unterstützten Features siehe *RBDService featureList*.

Syntax	<code>\$<var> featureKeys <index></code>
Beispiel	<p>Die Feature-Keys des ersten Client Services werden ausgegeben.</p> <pre>set cli [RBDService create] if ([\$cli numServices] > 0) { puts "feature keys: [\$cli featureKeys 0]" }</pre>
Ergebnis	<pre>feature keys: numSyncCalls procTimeSync callOrder lastInit numFiFoCalls lastCall numPending numCalls numAsyncCalls isActive numFails procTimeAsync .weight</pre>

Tabelle 31 - RBDService featureKeys

RBDService featureValues

Über die Methode *RBDService featureValues* erhält man eine Liste aller Feature-Ausprägungen eines Client Service. Für die Liste der unterstützten Features siehe *RBDService featureList*.

Syntax	<code>\$<var> featureValues <index></code>
Beispiel	<p>Die Feature-Values des ersten Client Services werden ausgegeben.</p> <pre>set cli [RBDService create] if ([\$cli numServices] > 0) { puts "feature values: [\$cli featureValues 0]" }</pre>
Ergebnis	<pre>feature values: 200 12220 2 1218192452727 20 218193652727 312 233 13 false 7 2000 8</pre>

Tabelle 32 - RBDService featureValues

RBDService numFeatures

Die Methode *RBDService numFeatures* liefert die Anzahl der bei einem Client Service vorhandenen Features. Das Service wird über seinen Index definiert.

Syntax	<code>\$(var) numFeatures <index></code>
Beispiel	Die Anzahl der Features des ersten Client Services wird ausgegeben. <pre>set cli [RBDService create] puts "number of features: [\$cli numFeatures 0]"</pre>
Ergebnis	number of features: 13

Tabelle 33 - RBDService numFeatures

RBDService hasFeature

Über die Methode *RBDService hasFeature* kann ermittelt werden, ob das spezifizierte Feature beim aktuellen Client Service vorhanden ist. Für die Liste der unterstützten Features siehe *RBDService.featureList*.

Syntax	<code>\$(var) hasFeature <index> <feature></code>
Beispiel	siehe RBDService getFeature

Tabelle 34 - RBDService hasFeature

RBDService getFeature

Über die Methode *RBDService getFeature* kann der Wert des spezifizierten Feature ermittelt werden. Für die Liste der unterstützten Features siehe *RBDService.featureList*.

Syntax	<code>\$(var) getFeature <index> <feature></code>
Beispiel	die bisherige Verarbeitungszeit bei synchronen Aufrufen des ersten Client Service soll ermittelt werden. <pre>set cli [RBDService create] if ([\$cli numServices] > 0) { if {[\$cli hasFeature 0 procTimeSync] == true} { puts "feature procTimeSync found. value is [\$cli getFeature 0 procTimeSync]." } } </pre>
Ergebnis	feature procTimeSync found. value is 12000.

Tabelle 35 - RBDService getFeature

RBDService sort

Die Methode *RBDService sort* dient dazu, die vorhandenen Client Services nach einem oder mehreren der vorhandenen Features zu sortieren. Die Sortierkriterien werden in einer Liste übergeben, bei jedem Kriterium wird spezifiziert, ob die Sortierung auf- oder absteigend erfolgen soll. Für die Liste der unterstützten Features siehe *RBDService featureList*.

Syntax	<code>\$<var> sort { <feature> [asc dsc] ... }</code>
Beispiel	<p>Die Menge der Client Services soll nach <i>numCalls</i> absteigend und bei gleicher Anzahl von <i>numCalls</i> nach <i>procTimeSync</i> aufsteigend sortiert werden.</p> <pre> set cli [RBDService create] set i 0 while {\$i < [\$cli numServices]} { puts "data of group \$i" puts " > [\$cli getService \$i]" set i (\$i + 1) } puts "\nsorting numCalls DSC, procTimeSync ASC\n" \$cli sort {numCalls dsc procTimeSync asc} <print services> </pre>
Ergebnis	<pre> data of group 1 > numCalls 133 serviceID {serviceid 1.1} procTime 1200 ... data of group 2 > numCalls 133 serviceID {serviceid 2.1} procTime 5500 ... data of group 3 > numCalls 1133 serviceID {serviceid 2.2} procTime 9900 ... "sorting numCalls DSC, procTime ASC" data of group 1 > numCalls 1133 serviceID {serviceid 2.2} procTime 9900 ... data of group 2 > numCalls 133 serviceID {serviceid 1.1} procTime 1200 ... data of group 3 > numCalls 133 serviceID {serviceid 2.1} procTime 5500 ... </pre>

Tabelle 36 - RBDService sort

RBDService buildResult

Die Methode *RBDService buildResult* liefert als Ergebnis eine Liste mit jenen Informationen, welche durch das Objekt *RBDResult* als Ergebnis der Auswertung des FRAG Rule Scripts an den Dispatcher geliefert werden. Wird der Parameter *index* nicht angegeben, so wird für jedes Client Service ein Eintrag geliefert, anderenfalls nur für jenes, welches durch den Index spezifiziert wird.

Syntax	<code>\$<var> buildResult [<index>]</code>
Beispiel	<p>Als Ergebnis soll jenes Service an den Dispatcher geliefert werden, welches die kürzeste durchschnittliche (synchrone) Laufzeit aufweist.</p> <pre>set cli [RBDService create] set result [RBDResult create] \$cli sort {procTimeSync asc} \$result addService [\$cli buildResult 0]</pre>
Ergebnis	RBDResult enthält das Service mit der kürzesten synchronen Processing Time

Tabelle 37 - RBDService buildResult

6.5 Zugriff auf Incoming Requests mittels *RBDIncoming*

Die bekannten und für den Dispatcher nutzbaren Incoming Requests sind über das Objekt *RBDIncoming* zugreifbar. Jeder Request besitzt neben seinen Stammdaten eine Menge von 'Features', welche über die entsprechenden Methoden abgefragt werden können. Dabei handelt es sich um statistische Daten (Anzahl der bisher getätigten asynchronen Aufrufe, Timestamp des letzten Aufrufes,...), die im Zuge der bisherigen Verarbeitungen des Request gesammelt worden sind.

Auf die Parameter des Incoming Request kann ebenfalls zugegriffen werden, um zu ermöglichen, dass das optimale Client Service anhand der operativen Daten des Incoming Request gesucht werden kann.

Das Objekt *RBDINCOMING* bietet die folgenden Methoden an:

<i>groupName</i>	die eindeutige externe Bezeichnung der Request Gruppe des Incoming Request
<i>groupID</i>	die eindeutige ID der Request Gruppe
<i>requestName</i>	die eindeutige externe Bezeichnung des Incoming Request
<i>requestID</i>	die eindeutige ID des Request
<i>soapHeader</i>	liefert den Header der SOAP-Request Message als String
<i>soapBody</i>	liefert den Body der SOAP-Request Message als String
<i>paramList</i>	liefert eine Liste aller Parameters inklusive deren Ausprägungen für den Incoming Request
<i>paramKeys</i>	liefert eine Liste der Parameter des Incoming Request
<i>paramValues</i>	liefert eine Liste der Parameterwerte des Incoming Request
<i>numParams</i>	die Anzahl der Parameter
<i>hasParam</i>	gibt an, ob ein spezifizierter Parameter existiert
<i>getParam</i>	liefert den Wert des angegebenen Parameters
<i>setParam</i>	ändert den Wert des angegebenen Parameters
<i>clearParams</i>	löscht alle Parameter des Request
<i>featureList</i>	liefert die Liste aller Features inklusive deren Ausprägungen eines Incoming Request
<i>featureKeys</i>	liefert eine Liste aller Features eines Incoming Request
<i>featureValues</i>	liefert eine Liste aller Feature-Ausprägungen eines Incoming Request
<i>numFeatures</i>	gibt die Anzahl der Features eines Incoming Request an
<i>hasFeature</i>	gibt an, ob ein spezielles Feature bei einem Incoming Request vorhanden ist
<i>getFeature</i>	ermittelt den Werte eines speziellen Feature

Tabelle 38 - RBDIncoming Methoden

Die Verwendung dieser Methoden wird in der Folge beschrieben.

RBDIncoming groupName

Die Methode *RBDIncoming groupName* liefert den Namen der Request Gruppe des aktuellen Request.

Syntax	<code>\$<var> groupName</code>
Beispiel	siehe <i>RBDIncoming soapBody</i>

Tabelle 39 - RBDIncoming groupName***RBDIncoming groupID***

Die Methode *RBDIncoming groupID* liefert die ID der Request Gruppe des aktuellen Incoming Request.

Syntax	<code>\$<var> groupID</code>
Beispiel	siehe <i>RBDIncoming soapBody</i>

Tabelle 40 - RBDIncoming groupID***RBDIncoming requestName***

Die Methode *RBDIncoming requestName* liefert den Namen des aktuellen Request.

Syntax	<code>\$<var> requestName</code>
Beispiel	siehe <i>RBDIncoming soapBody</i>

Tabelle 41 - RBDIncoming requestName

RBDIncoming requestID

Die Methode *RBDIncoming requestID* liefert die ID des aktuellen Incoming Request.

Syntax	<code>\$<var> requestID</code>
Beispiel	siehe <i>RBDIncoming soapBody</i>

Tabelle 42 - RBDIncoming requestID

RBDIncoming soapHeader

Die Methode *RBDIncoming soapHeader* liefert den SOAP-Header des aktuellen Incoming Request.

Syntax	<code>\$<var> soapHeader</code>
Beispiel	siehe <i>RBDIncoming soapBody</i>

Tabelle 43 - RBDIncoming soapHeader

RBDIncoming soapBody

Die Methode *RBDIncoming soapBody* liefert den SOAP-Header des aktuellen Incoming Request.

Syntax	<code>\$<var> soapBody</code>
Beispiel	Zugriff auf die Basisdaten des Incoming request
	<pre> set msg [RBDIncoming create] puts "groupName: [\$msg groupName]" puts "group-ID : [\$msg groupID]" puts "reqName : [\$msg requestName]" puts "req-ID : [\$msg requestID]" puts "SOAP-Hdr : [\$msg soapHeader]" puts "SOAP-Body: [\$msg soapBody]" </pre>
Ergebnis	<pre> groupName: name of request group group-ID : NAMEOFREQUESTGROUP reqName : name of request req-ID : NAMEOFREQUEST SOAP-Hdr : <SOAP header> SOAP-Hdr : <SOAP body> </pre>

Tabelle 44 - RBDIncoming soapBody

RBDIncoming paramList

Die Methode *RBDIncoming paramList* liefert eine Liste von *key/value*-Paaren für alle Parameter des aktuellen Incoming Request

Syntax	<code>\$<var> paramList</code>
Beispiel	siehe <i>RBDIncoming numParams</i>

Tabelle 45 - RBDIncoming paramList***RBDIncoming paramKeys***

Die Methode *RBDIncoming paramKeys* liefert eine Liste aller Parameternamen des aktuellen Incoming Request

Syntax	<code>\$<var> paramKeys</code>
Beispiel	siehe <i>RBDIncoming numParams</i>

Tabelle 46 - RBDIncoming paramKeys***RBDIncoming paramValues***

Die Methode *RBDIncoming paramValues* liefert eine Liste aller Parameterwerte des aktuellen Incoming Request

Syntax	<code>\$<var> paramValues</code>
Beispiel	siehe <i>RBDIncoming numParams</i>

Tabelle 47 - RBDIncoming paramValues

RBDIncoming numParams

Die Methode *RBDIncoming numParams* liefert die Anzahl der beim Incoming Request vorhandenen Parameter.

Syntax	<code>\$<var> numParams</code>
Beispiel	Anzeige der vorhandenen Parameter des aktuellen Incoming Request <pre>set msg [RBDIncoming create] puts "size : [\$msg numParams]" puts "keys : [\$msg paramKeys]" puts "values: [\$msg paramValues]" puts "list : [\$msg paramList]"</pre>
Ergebnis	<pre>size : 4 keys : Male Surname Forename Age values: true Bobik {Hans Joachim} 42 list : Male true Surname Bobik Forename Joachim Age 42</pre>

Tabelle 48 - *RBDIncoming numParams****RBDIncoming hasParam***

Über die Methode *RBDIncoming hasParam* kann festgestellt werden, ob ein spezifizierter Parameter beim aktuellen Request vorhanden ist.

Syntax	<code>\$<var> hasParam <parameter></code>
Beispiel	siehe <i>RBDIncoming setParam</i>

Tabelle 49 - *RBDIncoming hasParam****RBDIncoming clearParam***

Die Methode *RBDIncoming clearParam* löscht alle Parameter des aktuellen Incoming Request.

Syntax	<code>\$<var> clearParams</code>
--------	--

Tabelle 50 - *RBDIncoming clearParam*

RBDIncoming getParam

Die Methode *RBDIncoming getParam* liefert den Wert des spezifizierten Parameters.

Syntax	<code>\$<var> getParam <parameter></code>
Beispiel	siehe <i>RBDIncoming setParam</i>

Tabelle 51 - RBDIncoming getParam

RBDIncoming setParam

Die Methode *RBDIncoming setParam* setzt den Wert des spezifizierten Parameters auf einen neuen Wert.

Syntax	<code>\$<var> setValue <parameter> <value></code>
Beispiel	<p>das Vorhandensein eines Parameters wird abgefragt. Ist er nicht vorhanden, so wird er angelegt. Dann wird sein Wert abgefragt.</p> <pre> set msg [RBDIncoming create] puts "keys: [\$msg paramKeys]" if { ![\$msg hasParam newParam] } { puts "msg does not contain parameter newParam" \$msg setParam newParam "this is the new one" puts "parList: [\$msg paramKeys]" } puts "value of newParam is: '[\$msg getParam newParam]'" </pre>
Ergebnis	<pre> keys: Male Surname Forename Age msg does not contain parameter newParam parList: Male true Surname Bobik Forename {Hans Joachim} Age 42 newParam {this is the new one} value of newParam is 'this is the new one' </pre>

Tabelle 52 - RBDIncoming setParam

RBDIncoming featureList

Über die Methode *RBDIncoming featureList* erhält man eine Liste aller Features und ihrer Ausprägungen eines Incoming request.

Folgende Features werden unterstützt:

lastRequest	der Timestamp des letzten Einganges eines solchen Incoming Request
numRequests	die Anzahl der bisherigen Eingänge des Incoming Request seit dem letzten Init der Statistikdaten
numSyncReqs	wie oben für synchrone Requests
numAsyncReqs	wie oben für asynchrone Requests
numFiFoReqs	wie oben für fire & forget Requests
pendingReqs	die Anzahl der noch nicht fertig verarbeiteten Requests
numFails	die Anzahl der bislang nicht erfolgreich verarbeiteten Incoming Requests
procTimeSync	die bisherige Verarbeitungsdauer aller synchronen Verarbeitungen in Millisekunden
procTimeAsync	die bisherige Verarbeitungsdauer aller asynchronen Verarbeitungen in Millisekunden
lastInit	der Timestamp der letzten Initialisierung aller Werte für den aktuellen Incoming Request
isActive	gibt an, ob der Incoming Request im Moment verarbeitet werden kann

Syntax	<code><var> featureList</code>
Beispiel	Die Features des Incoming Request sollen angezeigt werden <pre>set req [RBDIncoming create] puts "features: [\$req featureList]"</pre>
Ergebnis	<pre>features: numFails 9 procTimeSync 2222 numRequests 14702 lastRequest 1218201154569 numSyncReqs 12345 lastInit 1218203376769 procTimeAsync 44444 numFiFoReqs 12 pendingReqs 4 isActive true numAsyncReqs 2345</pre>

Tabelle 53 - RBDIncoming featureList

RBDIncoming featureKeys

Über die Methode *RBDIncoming featureKeys* erhält man eine Liste aller Features eines Incoming Request. Für die Liste der unterstützten Features siehe *RBDIncoming featureList*.

Syntax	<code>\$<var> featureKeys</code>
Beispiel	Die Feature-Keys des Incoming Request werden ausgegeben. <pre>set req [RBDIncoming create] puts "feature keys: [\$req featureKeys]"</pre>
Ergebnis	<code>feature keys: numFails procTimeSync numRequests lastRequest numSyncReqs lastInit procTimeAsync numFiFoReqs pendingReqs isActive numAsyncReqs</code>

Tabelle 54 - RBDIncoming featureKeys

RBDIncoming featureValues

Über die Methode *RBDIncoming featureValues* erhält man eine Liste aller Feature-Ausprägungen eines Incoming Request. Für die Liste der unterstützten Features siehe *RBDIncoming featureList*.

Syntax	<code>\$<var> featureValues</code>
Beispiel	Die Feature-Values des Incoming Request werden ausgegeben. <pre>set req [RBDIncoming create] puts "feature values: [\$req featureValues]"</pre>
Ergebnis	<code>feature values: 200 12220 260 1218192452727 20 218193652727 312 233 13 false 7 2000 8</code>

Tabelle 55 - RBDIncoming featureValues

RBDIncoming numFeatures

Die Methode *RBDIncoming numFeatures* liefert die Anzahl der bei einem Incoming Request vorhandenen Features. Für die Liste der unterstützten Features siehe *RBDIncoming featureList*.

Syntax	<code>\$<var> numFeatures</code>
Beispiel	Die Anzahl der Features des ersten Incoming Request wird ausgegeben. <pre>set req [RBDIncoming create] puts "number of features: [\$req numFeatures]"</pre>
Ergebnis	number of features: 11

Tabelle 56 - RBDIncoming numFeatures

RBDIncoming hasFeature

Über die Methode *RBDIncoming hasFeature* kann ermittelt werden, ob das spezifizierte Feature beim aktuellen Incoming Request vorhanden ist. Für die Liste der unterstützten Features siehe *RBDIncoming featureList*.

Syntax	<code>\$<var> hasFeature <feature></code>
Beispiel	siehe RBDIncoming getFeature

Tabelle 57 - RBDIncoming hasFeature

RBDIncoming getFeature

Über die Methode *RBDIncoming getFeature* kann der Wert des spezifizierten Feature ermittelt werden. Für die Liste der unterstützten Features siehe *RBDIncoming featureList*.

Syntax	<code>\$<var> getFeature <feature></code>
Beispiel	die bisherige Verarbeitungszeit bei synchronen Verarbeitungen des Incoming Request soll ermittelt werden. <pre>set req [RBDIncoming create] if {[\$req hasFeature procTimeSync] == true} { puts "feature procTimeSync found. value is [\$req getFeature procTimeSync]."</pre>
Ergebnis	feature procTimeSync found. value is 12000.

Tabelle 58 - RBDIncoming getFeature

6.6 Spezifikation des FRAG-Ergebnisses mittels *RBDResult*

Über dieses Objekt werden die Ergebnisse der Auswertung eines FRAG Rule Scripts an den Dispatcher retourniert. Diese Ergebnisse können jene Client Services sein, welche vom Dispatcher zu starten sind. Alternativ ist aber auch die Definition einer Rule Engine möglich, welche die weitere Auswertung übernehmen soll.

Das Objekt *RBDRESULT* bietet die folgenden Methoden an:

<i>addService</i>	ein oder mehrere Client Services werden zum bisherigen Ergebnis hinzugefügt
<i>addEngine</i>	es wird die externe Rule Engine angegeben, welche die Menge der passenden Client Services ermittelt
<i>trace</i>	Trace-Meldungen werden im Ergebnis abgelegt
<i>clearTrace</i>	die im Ergebnis vorhandenen Trace-Meldungen werden gelöscht

Tabelle 59 - RBDResult Methoden

Anmerkung: Das Objekt *RBDResult* ist als Singleton konzipiert. Der Versuch, in einem FRAG Rule Script ein zweites Objekt dieses Typs anzulegen, führt zu einer Exception, wie in folgendem Beispiel zu sehen ist:

Beispiel	Versuch der Anlage zweier Objekte vom Typ <i>RBDResult</i> <pre>set result_A [RBDResult create] set result_B [RBDResult create]</pre>
Ergebnis	eine Exception der Form <pre>A RBDResult object was already instanciated in: 'RBDResult create' at: line = 2</pre>

Tabelle 60 - RBDResult Exception

Auf den nächsten Seiten wird die Verwendung dieser Methoden erläutert.

RBDResult addService

Über die Methode *RBDResult addService* werden ein oder mehrere Client Services zum Ergebnis hinzugefügt. Bisher bereits angegebene Services werden nicht verändert, zuvor mittels *addEngine* spezifizierte Rule Engines werden gelöscht. Wird die als Parameter an das Service übergebene Liste nicht von der Methode *RBDService buildResult* übernommen sondern manuell angegeben, so ist eine Liste bestehend aus Paaren {client service group name, client service name} anzugeben, wobei jeweils die bei der Konfiguration der Client Services definierten (internen) Namen der Objekte zu verwenden sind.

Syntax	<pre>\$<var> addService <client service result> oder \$<var> addService {<cs group> <client service> ...}</pre>
Beispiel	<p>Das Client Services mit der kürzesten durchschnittlichen Laufzeit, gefolgt von den beiden Services der Client Service Gruppe CSGROUP_1, soll an den Dispatcher übergeben werden.</p> <pre>set cli [RBDService create] set result [RBDResult create] \$cli sort {procTime asc} \$result addService [\$cli buildResult 0] \$result addService {CSGROUP_1 SVC_A CSGROUP_1 SVC_B}</pre>
Ergebnis	RBDResult enthält die drei Client Services - siehe auch <i>RBDResult trace</i>

Tabelle 61 - *RBDResult addService*

Anmerkungen: Die Reihenfolge, in der die Client Services zum Ergebnis hinzugefügt werden, entspricht der Reihenfolge, in der die Services später vom Dispatcher angestoßen werden.

Wird das Ergebnis mittels *RBDService create* ohne Angabe eines Index erstellt, so entspricht die Reihenfolge der Client Services der aktuellen Reihenfolge in *RBDService*.

Ein Client Service, welches bereits im Ergebnis vorhanden ist, kann nicht neuerlich hinzugefügt werden, der entsprechende Versuch wird ignoriert.

RBDResult addEngine

Über die Methode *RBDResult addEngine* kann eine externe Rule Engine angegeben werden, welche die zu verwendenden Client Services ermittelt. Die Angabe einer Rule Engine überschreibt eine zuvor definierte Rule Engine und löscht alle zuvor mittels *addService* spezifizierten Services. Ein Engine Name * bedeutet, dass die konfigurierte Rule Engine zu verwenden ist.

Syntax	<code>\$<var> addEngine [<rule engine> *]</code>
Beispiel	Die Rule Engine mit der Kennung SIMPLERROBIN soll an den Dispatcher übergeben werden. <pre>set result [RBDResult create] \$result addEngine SIMPLERROBIN</pre>
Ergebnis	RBDResult enthält die Rule Engine mit der Kennung SIMPLERROBIN

Tabelle 62 - RBDResult addEngine

RBDResult trace

Die Methode *RBDResult trace* dient analog dem FRAG command *puts* dazu, Informationen, welche im Zuge der Auswertung eines FRAG Rule Scripts anfallen, in der Ergebnisstruktur zu speichern. Diese Informationen können im Konfigurationssystem beim Test des FRAG Rule Scripts angezeigt werden.

Syntax	<code>\$<var> trace <trace string></code>
Beispiel	Speichern von Trace Messages im Ergebnisobjekt <pre>set cli [RBDService create] set result [RBDResult create] \$result trace "starting traces..." \$result trace "Result 0: [\$cli buildResult 0]" \$result trace "Result 1: [\$cli buildResult 1]" \$result addService [\$cli buildResult]</pre>
Ergebnis	Im Konfigurationssystem werden bei Betätigen der Schaltfläche <i>CHECK SCRIPT</i> die folgenden Ergebnisse angezeigt: <pre>> Traced messages: starting traces... Result 0: SERVICESGROUP1 CSGROUP1_B_INVOKER Result 1: SERVICESGROUP1 CSGROUP1_C_INVOKER > Result (client services): SERVICESGROUP1:CSGROUP1_B_INVOKER SERVICESGROUP1:CSGROUP1_C_INVOKER</pre>

Tabelle 63 - RBDResult trace

RBDResult clearTrace

Über die Methode *RBDResult clearTrace* werden die bisher angelegten Trace Messages in *RBDResult* gelöscht.

Syntax	<code>\$<var> clearTrace</code>
Beispiel	<p>Die Trace Messages im Ergebnisobjekt sollen gelöscht werden.</p> <pre>set result [RBDResult create] \$result trace "This is a wonderful trace message" \$result clearTrace \$result trace "starting traces..."</pre>
Ergebnis	<p>Im Konfigurationssystem werden bei Betätigen der Schaltfläche <i>CHECK SCRIPT</i> die folgenden Ergebnisse angezeigt:</p> <pre>> Traced messages: starting traces...</pre>

Tabelle 64 - RBDResult clearTrace

7 Datenbankschema

In diesem Kapitel wird der Persistenz-Layer des Dispatchers beschrieben. Das Entity-Relationship Modell für den Dispatcher ist in Abbildung 29 dargestellt.

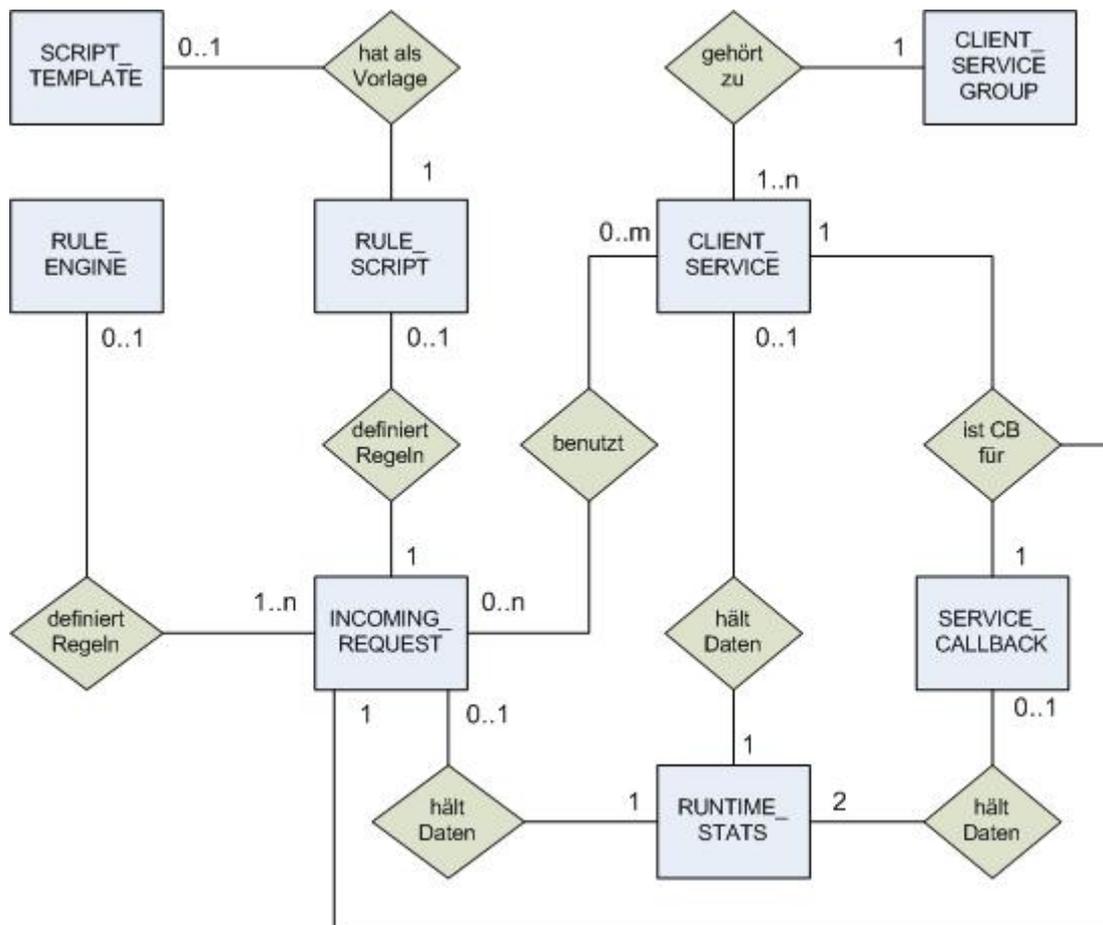


Abbildung 29 - Entity Relationship Diagramm

Ein INCOMING REQUEST kann von 0 - m CLIENT SERVICES verarbeitet werden. Welches Service für eine Anfrage konkret herangezogen wird, wird entweder durch eine RULE ENGINE bestimmt oder durch die Auswertung eines RULE SCRIPT. Beide können dem INCOMING REQUEST zugeordnet werden, wobei Vorlagen für RULE SCRIPTS als SCRIPT TEMPLATES abgelegt werden können. Jedes CLIENT SERVICE ist Member genau einer CLIENT SERVICEGROUP. Wird ein Request asynchron verarbeitet, wird dafür ein SERVICE CALLBACK Objekt angelegt, in dem die ursprünglich vom Request definierte Callback Info solange abgelegt wird, bis der Callback des CLIENT SERVICE einlangt. RUNTIME STATS werden für INCOMING REQUESTS und CLIENT SERVICES geführt.

Im Rest dieses Kapitels werden nun die in der aktuellen Implementierung des Dispatchers verwendeten Entitäten mit ihren Attributen und Indizes beschrieben.

7.1 Incoming Requests

In dieser Entität werden die im Zuge der Konfiguration definierten Daten über eingehende Anfragen gespeichert. Zu jeder Anfrage können die Links zu einer Rule Engine und/oder zu einem FRAG Rule Script abgelegt werden.

7.1.1 Create Statement

```
drop table if exists DISPATCHER.INCOMING_REQUEST;
create table DISPATCHER.INCOMING_REQUEST (
  row_id          INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  reqgroup_id    VARCHAR(254) NOT NULL,
  reqgroup_name  VARCHAR(254) NOT NULL,
  reqgroup_dsc   VARCHAR(254) NOT NULL,
  request_id     VARCHAR(254) NOT NULL,
  request_name   VARCHAR(254) NOT NULL,
  request_dsc    VARCHAR(254) NOT NULL,
  call_type      CHAR(12) NOT NULL,
  rule_engine    VARCHAR(254),
  rulescript_id  CHAR(16)
);
create unique index INREQIDIDX on DISPATCHER.INCOMING_REQUEST
(request_id, reqgroup_id);
create index RULENGIDX on DISPATCHER.INCOMING_REQUEST (rule_engine);
commit;
```

7.1.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
reqgroup_id	die ID der Request-Gruppe; als ID dient der normalisierte Gruppename (d.h. der Name in Großbuchstaben, aus dem alle nicht-alphanumerischen Zeichen entfernt wurden); kennzeichnet zusammen mit der request_id einen Datensatz eindeutig
reqgroup_name	der Name der Request-Gruppe
reqgroup_dsc	die Beschreibung der Request-Gruppe
request_id	die ID des Request; als ID dient der normalisierte Requestname; kennzeichnet zusammen mit der reqgroup_id einen Datensatz eindeutig
request_name	der Name des Request - eindeutig innerhalb seiner Gruppe
request_dsc	die Beschreibung des Request
call_type	gibt an, ob der Request synchron, asynchron oder als fire&forget-call ausgeführt werden soll; derzeit fix mit dem Wert "RUNTIME" belegt, d.h. der Aufrufer legt die Verarbeitungsoption fest.
rule_engine	die ID der dem Incoming Request (optional) zugeordneten Rule Engine, welche die für diesen Request passenden Client Services festlegt

rulescript_id	die ID des dem Incoming Request (optional) zugeordneten FRAG Rule Scripts, welches die für diesen Request passenden Client Services festlegt.
---------------	---

Tabelle 65 - Entität INCOMING_REQUEST

Anmerkung: In einigen Entitäten werden *normalisierte* Namen verwendet. Diese werden aus dem ursprünglichen Namen generiert, indem alle nicht-alphanumerischen Zeichen entfernt und die verbleibenden Zeichen in Großbuchstaben umgewandelt werden.

Anmerkung: Da unter MySQL in der vorliegenden Version wiederholt Probleme mit AUTO-INCREMENT Attributen aufgetreten sind, werden in einigen Entitäten zusätzlich zum Primärschlüssel noch weitere IDs generiert, welche einen Datensatz eindeutig identifizieren.

7.2 Client Service

In dieser Entität werden Informationen über die verfügbaren Client Services abgelegt. Neben ID, Name und Typ des Service liegen hier auch jene Informationen, welche für den Aufruf des Service durch den Dispatcher benötigt werden.

7.2.1 Create Statement

```
drop table if exists DISPATCHER.CLIENT_SERVICE;
create table DISPATCHER.CLIENT_SERVICE (
  row_id          INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  group_namenorm  VARCHAR(254) NOT NULL,
  service_type    CHAR(4) NOT NULL,
  service_name    VARCHAR(254) NOT NULL,
  service_namenorm VARCHAR(254) NOT NULL,
  service_nameext VARCHAR(254) NOT NULL,
  service_dscext  VARCHAR(254) NOT NULL,
  service_invoker VARCHAR(254) NOT NULL,
  provider_url    VARCHAR(254) NOT NULL,
  qos_service     BOOLEAN NOT NULL
);
create unique index CLISIDX_NN on DISPATCHER.CLIENT_SERVICE
(group_namenorm, service_namenorm);
commit;
```

7.2.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
group_namenorm	die ID der übergeordneten Client Service Gruppe
service_type	gibt an, ob das Service als Webservice ("WSDL") oder Enterprise Java Bean ("BEAN") implementiert ist
service_name	der intern verwendete Servicename, welcher zum Aufruf des

	Service verwendet wird
service_namenorm	der normalisierte interne Servicename, verwendet für Prüfungen auf Eindeutigkeit
service_nameext	der extern definierte Name des Service
service_dscext	die externe Beschreibung des Service
service_invoker	gibt den Namen des Webservice an bzw. den Namen der Enterprise Java Bean, welche das Service implementiert.
provider_url	die URL des Providers des Client Service
qos_service	derzeit nicht verwendet, gibt an, ob es sich bei dem Eintrag um ein Service handelt, welches zur Ermittlung der QoS-Informationen der Gruppe dient

Tabelle 66 - Entität CLIENT_SERVICE

7.3 Client Service Group

In dieser Entität werden Informationen über die verfügbaren Client Services Gruppen abgelegt.

7.3.1 Create Statement

```
drop table if exists DISPATCHER.CLIENT_SERVICEGROUP;
create table DISPATCHER.CLIENT_SERVICEGROUP (
  row_id          INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  group_id        CHAR(32) NOT NULL,
  group_name      VARCHAR(254) NOT NULL,
  group_namenorm  VARCHAR(254) NOT NULL,
  group_nameext   VARCHAR(254) NOT NULL,
  group_dscext    VARCHAR(254),
  qos_service     VARCHAR(254),
  qos_valid_till  TIMESTAMP,
  version         INTEGER
);
create unique index CLISGIDX_ID on DISPATCHER.CLIENT_SERVICEGROUP
(group_id);
create unique index CLISGIDX_NN on DISPATCHER.CLIENT_SERVICEGROUP
(group_namenorm);
commit;
```

7.3.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
group_id	die interne eindeutige ID der Client Service Gruppe
group_name	der im Zuge der Konfiguration intern vergebene Gruppenname
group_namenorm	der normalisierte interne Gruppenname
group_nameext	der extern definierte Name der Client Service Gruppe

group_dsctxt	die externe Beschreibung der Client Service Gruppe
qos_service	derzeit nicht verwendet; zukünftig die ID jenes Service, über welches die QoS-Attribute für die Gruppe ermittelt werden können
qos_valid_till	derzeit nicht verwendet; der Zeitpunkt, an dem die aktuellen QoS-Informationen ungültig werden
version	derzeit nicht verwendet; die aktuelle Version des Datensatzes

Tabelle 67 - Entität CLIENT_SERVICEGROUP

7.4 Request Service Link

Diese Entität enthält die Client Service Bindings der Incoming Requests, d.h. Informationen darüber, welche Client Services die Verarbeitung für einen Incoming Request übernehmen können.

7.4.1 Create Statement

```
drop table if exists DISPATCHER.REQUEST_SERVICE;
create table DISPATCHER.REQUEST_SERVICE (
  row_id          INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  reqgroup_id    VARCHAR(254) NOT NULL,
  request_id     VARCHAR(254) NOT NULL,
  csgroup_id    VARCHAR(254) NOT NULL,
  service_id     VARCHAR(254) NOT NULL,
  call_order    INTEGER NOT NULL,
  version        INTEGER,
  assn_params    VARCHAR(1024)
);
create index INREQIDX on DISPATCHER.REQUEST_SERVICE (reqgroup_id,
request_id);
create index INCSGIDX on DISPATCHER.REQUEST_SERVICE (csgroup_id,
service_id);
create unique index REQSRVIDX on DISPATCHER.REQUEST_SERVICE
(reqgroup_id, request_id, csgroup_id, service_id);
commit;
```

7.4.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
reqgroup_id	die ID der Request-Gruppe; als ID dient der normalisierte Gruppenname; kennzeichnet zusammen mit der request_id einen Incoming Request eindeutig
request_id	die ID des Request; als ID dient der normalisierte Requestname; kennzeichnet zusammen mit der reqgroup_id einen Incoming Request eindeutig
csgroup_id	die ID der Client Service Gruppe, entspricht dem Attribut group_namenorm der Entität

	DISPATCHER.CLIENT_SERVICEGROUP und kennzeichnet zusammen mit der service_id ein Client Service eindeutig.
service_id	die ID des Client Service, entspricht dem Attribut service_namenorm der Entität DISPATCHER.CLIENT_SERVICE und kennzeichnet zusammen mit der csgroup_id ein Client Service eindeutig.
call_order	die im Zuge der Konfiguration definierte Reihenfolge der Client Services.
version	derzeit nicht verwendet; die aktuelle Version des Datensatzes
assn_params	die Zuordnungsattribute, welche zu jedem Client Service Binding angegeben werden können - siehe Beschreibung des Pflegesystems in Kapitel 4.4.2.

Tabelle 68 - Entität REQUEST_SERVICE

7.5 Service Callback

Diese Entität hält alle nötigen Informationen, damit ein vom Outgoing Adapter eines Client Service angestoßener Callback die an der Transaktion beteiligten Objekte identifizieren und den Incoming Adapter des ursprünglichen Request aufrufen kann.

7.5.1 Create Statement

```
drop table if exists DISPATCHER.SERVICE_CALLBACK;
create table DISPATCHER.SERVICE_CALLBACK (
  client_corr_id          VARCHAR(64) NOT NULL PRIMARY KEY,
  request_corr_id        VARCHAR(64) NOT NULL,
  invoc_req_ts           TIMESTAMP NOT NULL,
  invoc_req_pk           INTEGER NOT NULL,
  invoc_svc_ts           TIMESTAMP NOT NULL,
  invoc_svc_pk           INTEGER NOT NULL,
  reqgroup_id            VARCHAR(254) NOT NULL,
  request_id             VARCHAR(254) NOT NULL,
  csgroup_id             VARCHAR(254) NOT NULL,
  service_id            VARCHAR(254) NOT NULL,
  provider_url           VARCHAR(254) NOT NULL,
  callback_invoker       VARCHAR(254) NOT NULL,
  callback_method        VARCHAR(254) NOT NULL,
  invoc_error_ts         TIMESTAMP,
  invoc_error_msg        VARCHAR(254)
);
create index SVCCBIDX on DISPATCHER.SERVICE_CALLBACK (reqgroup_id,
request_id, csgroup_id, service_id);
create unique index RCIDIDX on DISPATCHER.SERVICE_CALLBACK
(request_corr_id);
commit;
```

7.5.2 Beschreibung der Attribute

Attribut	Beschreibung
----------	--------------

Datenbankschema

client_corr_id	die Correlation-ID für die Verbindung zwischen Dispatcher und Client Service
request_corr_id	die Correlation-ID für die Verbindung zwischen Dispatcher und Incoming Request Engine
invoc_req_ts	der Timestamp des Zeitpunktes, zu dem der Incoming Request beim Dispatcher eingelangt ist. Wird benötigt, um bei Einlangen eines managed Callbacks die "Durchlaufzeit" eines Request ermitteln zu können. Siehe auch Entität <code>DISPATCHER.RUNTIME_STATS</code>
invoc_req_pk	der Primary Key des mit dem Incoming Request assoziierten Datensatzes in der Entität <code>DISPATCHER.RUNTIME_STATS</code>
invoc_svc_ts	der Timestamp des Zeitpunktes, zu dem der Dispatcher das passende Client Service aktiviert hat. Wird benötigt, um die Verarbeitungszeit des Client Service messen zu können. Siehe auch Entität <code>DISPATCHER.RUNTIME_STATS</code>
invoc_svc_pk	der Primary Key des mit dem Client Service assoziierten Datensatzes in der Entität <code>DISPATCHER.RUNTIME_STATS</code>
reqgroup_id	die ID der Request-Gruppe; als ID dient der normalisierte Gruppenname; kennzeichnet zusammen mit der request_id einen Incoming Request eindeutig
request_id	die ID des Request; als ID dient der normalisierte Requestname; kennzeichnet zusammen mit der reqgroup_id einen Incoming Request eindeutig
csgroup_id	die ID der Client Service Gruppe, entspricht dem Attribut <code>group_namenorm</code> der Entität <code>DISPATCHER.CLIENT_SERVICEGROUP</code> und kennzeichnet zusammen mit der service_id ein Client Service eindeutig.
service_id	die ID des Client Service, entspricht dem Attribut <code>service_namenorm</code> der Entität <code>DISPATCHER.CLIENT_SERVICE</code> und kennzeichnet zusammen mit der csgroup_id ein Client Service eindeutig.
provider_url	die URL des Providers des Callback-Service
callback_invoker	gibt den Namen der Enterprise Java Bean an, welche das Callback-Service implementiert.
callback_method	die Callback-Methode
invoc_error_ts	der Timestamp des Zeitpunktes, an dem der letzte Fehler aufgetreten ist
invoc_error_msg	der Fehlertext des letzten aufgetretenen Fehlers

Tabelle 69 - Entität SERVICE_CALLBACK

7.6 Rule Engine

Diese Entität hält die Informationen über die verfügbaren Rule Engines.

7.6.1 Create Statement

```
drop table if exists DISPATCHER.RULE_ENGINE;
create table DISPATCHER.RULE_ENGINE (
    row_id            INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    engine_type       CHAR(4)  NOT NULL,
    engine_name       VARCHAR(254) NOT NULL,
    engine_namenorm   VARCHAR(254) NOT NULL,
    engine_nameext    VARCHAR(254) NOT NULL,
    engine_dscext     VARCHAR(254) NOT NULL,
    engine_invoker    VARCHAR(254) NOT NULL,
    provider_url      VARCHAR(254) NOT NULL # url where engine can be
found
);
create unique index RULENGIDX on DISPATCHER.RULE_ENGINE
(engine_namenorm);
commit;
```

7.6.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
engine_type	gibt an, ob die Rule Engine als Webservice ("WSDL") oder Enterprise Java Bean ("BEAN") implementiert ist
engine_name	der im Zuge der Pflege vergebene interne Name der Rule Engine
engine_namenorm	der normalisierte interne engine_name
engine_nameext	der externe Name der Rule Engine
engine_dscext	die externe Beschreibung der Rule Engine
engine_invoker	gibt den Namen des Webservices an bzw. den Namen der Enterprise Java Bean, welche die Rule Engine implementiert.
provider_url	die URL des Providers der externen Rule Engine

Tabelle 70 - Entität RULE_ENGINE

7.7 Rule Script

Diese Entität enthält die definierten FRAG Rule Scripts. Da Rule Scripts beliebig lange sein können, werden sie in Teile zu jeweils maximal 1024 Zeichen gesplittet und als Menge von Rows mit fortschreitender Sequenznummer abgelegt.

7.7.1 Create Statement

```
drop table if exists DISPATCHER.RULE_SCRIPT;
create table DISPATCHER.RULE_SCRIPT (
    row_id            INTEGER NOT NULL PRIMARY KEY,
    rulescript_id    CHAR(16) NOT NULL,
    seq_number       INTEGER NOT NULL,
    rule_size        INTEGER NOT NULL,
    rule_text        VARCHAR(1024) NOT NULL
);
create unique index RULEIDX on DISPATCHER.RULE_SCRIPT (rulescript_id,
seq_number);
commit;
```

7.7.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
rulescript_id	eine intern generierte eindeutige ID für das Script
seq_number	die Sequenznummer des Script-Teiles
rule_size	gibt die Länge des aktuellen Script-Teiles in Bytes an
rule_text	der Text des aktuellen Rule Script Teiles

Tabelle 71 - Entität RULE_SCRIPT

7.8 Script Template

Diese Entität enthält die Informationen über die definierten Script Templates. Diese Templates werden als normale Rule Scripts in der Entität `DISPATCHER.RULE_SCRIPT` abgelegt, in der Entität `DISPATCHER.SCRIPT_TEMPLATE` wird nur der Templatename gespeichert.

7.8.1 Create Statement

```
drop table if exists DISPATCHER.SCRIPT_TEMPLATE;
create table DISPATCHER.SCRIPT_TEMPLATE (
  row_id          INTEGER NOT NULL PRIMARY KEY,
  rulescript_id   CHAR(16) NOT NULL,
  template_name   VARCHAR(254) NOT NULL,
  template_namenorm VARCHAR(254) NOT NULL
);
create unique index NAMEIDX on DISPATCHER.SCRIPT_TEMPLATE
(template_namenorm);
commit;
```

7.8.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
rulescript_id	der Verweis auf den Datensatz in der Entität <code>DISPATCHER.RULE_SCRIPT</code> , welcher das Template enthält
template_name	der im Zuge der Konfiguration vergebene Name des Script Templates
template_namenorm	der normalisierte Name des Script Template

Tabelle 72 - Entität `SCRIPT_TEMPLATE`

7.9 Runtime Statistics

Diese Entität enthält die vom Dispatcher gesammelten statistischen Informationen über alle Incoming Requests und Client Services.

7.9.1 Create Statement

```
drop table if exists DISPATCHER.RUNTIME_STATS;
create table DISPATCHER.RUNTIME_STATS (
    row_id            INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
    link_row_id       INTEGER NOT NULL,
    link_type         CHAR(3) NOT NULL,
    last_called       TIMESTAMP DEFAULT '1970-01-02 00:00:00',
    pending_calls     INTEGER NOT NULL,
    num_calls_sync    INTEGER NOT NULL,
    num_calls_async   INTEGER NOT NULL,
    num_calls_fifo    INTEGER NOT NULL,
    num_fails         INTEGER NOT NULL,
    proctime_sync     INTEGER NOT NULL,
    proctime_async    INTEGER NOT NULL,
    proctime_rules    INTEGER NOT NULL,
    last_init         TIMESTAMP DEFAULT '1970-01-02 00:00:00',
    suspended         BOOLEAN NOT NULL,
    resume_at        TIMESTAMP DEFAULT '1970-01-02 00:00:00'
);
create unique index STATDX on DISPATCHER.RUNTIME_STATS (link_row_id,
link_type);
commit;
```

7.9.2 Beschreibung der Attribute

Attribut	Beschreibung
row_id	der Primärschlüssel der Entität; wird beim Einfügen eines neuen Datensatzes generiert.
link_row_id	der Fremdschlüssel des referenzierten Objektes (Incoming Request oder Client Service)
link_type	gibt an, ob die Statistikdaten einen Incoming Request ('REQ') oder ein Client Service ('CLS') betreffen
last_called	der Timestamp des Zeitpunktes des letzten Eintreffens eines Incoming Request bzw. des letzten Aufrufes eines Client Service
pending_calls	die Anzahl jener Incoming Requests bzw. Client Services, deren Verarbeitung gestartet, aber noch nicht beendet worden ist
num_calls_sync	die Anzahl der seit der letzten Initialisierung der Daten durchgeführten synchronen Verarbeitungen für den Request bzw. das Client Service
num_calls_async	wie oben für asynchrone Verarbeitungen mit managed Callbacks
num_calls_fifo	wie oben für Fire&Forget-Verarbeitungen

Datenbankschema

num_fails	die Anzahl der fehlerhaften Durchführungen eines Incoming Request oder Client Service
proctime_sync	die in Summe für alle synchronen Verarbeitungen eines Incoming Request oder Client Service seit der letzten Dateninitialisierung benötigte Zeit in Millisekunden
proctime_async	wie oben für asynchrone Verarbeitungen mit managed Callbacks
proctime_rules	die in Summe für alle Regelauswertungen (FRAG Rule Scripts und Rule Engines) eines Incoming Request seit der letzten Initialisierung der Daten benötigte Zeit in Millisekunden
last_init	der Zeitpunkt der letzten Initialisierung aller Daten
suspended	gibt an, ob ein Incoming Request oder ein Client Service im Moment inaktiviert und die entsprechende Verarbeitung somit nicht möglich ist
resume_at	derzeit nicht verwendet; der Timestamp des nächsten automatischen <i>resume</i> eines Request oder Client Service

Tabelle 73 - Entität **RUNTIME_STATS**

8 Projektstruktur und Deployment

8.1 Projektstruktur

Die Projektstruktur ist in Abbildung 30 dargestellt. Das Package `RBDConfigure` enthält das Konfigurationssystem sowie den Client zur Anzeige der statistischen Informationen, `RBDEngines` den Macroflow Engine Simulator und die Microflow Engines für den Test (siehe Kapitel 9) und das Package `RBDServer` den Dispatcher inklusive aller Session Beans und Entity Beans sowie die Rule Engines.

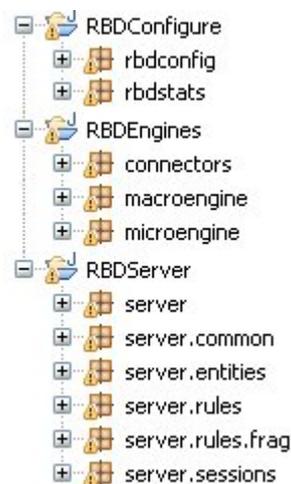


Abbildung 30 - Projektstruktur

In der folgenden Tabelle werden die Inhalte der einzelnen Packages kurz beschrieben.

Package .Java File	Beschreibung
RBDConfigure.rbdconfig	dieses Package enthält den Configuration Client
.RBDAssnParamsGUI	der Dialog für die Definition der Zuordnungsattribute
.RBDBeanSelector	der Dialog zur Auswahl von Enterprise Java Beans auf einem Applikationsserver. Verwendet bei der Auswahl von Client Service Adaptern und Rule Engines, wenn diese als Beans implementiert sind.
.RBDChatter	übernimmt die Kommunikation zwischen Configuration Client und Dispatcher
.RBDClientGUI	der Dialog zur Konfiguration von Client Services
.RBDConfig	die Basisklasse des Configuration Client
.RBDErrorPane	der Dialog zur Anzeige von Fehlerbedingungen
.RBDFileFilter	ein File-Filter für den Datei-Auswahldialog
.RBDFrameGUI	de GUI-Frame für den Configuration Client
.RBDIncomingGUI	der Dialog zur Konfiguration von Incoming Requests
.RBDRuleEnginesGUI	der Dialog zur Konfiguration von Rule Engines
.RBDRulesGUI	der Dialog zur Zuordnung von Client Services und Rule

Projektstruktur und Deployment

Package .Java File	Beschreibung
.RBDScriptGUI .TableColumnManager .WSDLParser	Engines/Rule Scripts zu Incoming Requests de Dialog zur Konfiguration von Rule Scripts eine Hilfsklasse für die Gestaltung von Tabboxen der WSDL-Parser; extrahiert Services und Operationen aus WSDL-Dateien
RBDConfigure.rbdstats .MyCellRenderer .RBDStatistics .RBDStatisticsGUI .RBDStatsReader	dieses Package enthält den Statistics Client ein CellRenderer für die Anzeige von Statistikdaten die Basisklasse des Statistics Client der Dialog des Statistics Client liest Statistikdaten vom Dispatcher
RBDEngines.connectors .IncomingAdapterBean .IncomingAdapterRemote .MIFE1UniversalBean .MIFE1UniversalRemote .MIFE2ServiceABean .MIFE2ServiceARemote .MIFE2ServiceBCBean .MIFE2ServiceBCRemote .OutgoingAdapter	dieses Package enthält die Incoming/Outgoing Adapter der Incoming Adapter für den Macroflow Engine Simulator; liest die SOAP-Message, extrahiert die Parameter, bereitet die Inputstrukturen für den Dispatcher auf und ruft diesen auf. Enthält auch die Callbackmethode für asynchrone Verarbeitungen - konvertiert in diesem Fall die einlangenden Daten und gibt sie an den Macroflow Engine Simulator zurück das Remote Interface des Incoming Adapters der Outgoing Adapter für die Microflow Engine 1 das Remote Interface für diesen Adapter der Outgoing Adapter für Service A auf Microflow Engine 2 das Remote Interface für diesen Adapter der Outgoing Adapter für Services B und C auf Microflow Engine 2 das Remote Interface für diesen Adapter die Superklasse für alle angeführten Outgoing Adapter
RBDEngines.macroengine .MacroEngine .MacroEngineGenerator .MacroEngineGUI .MacroEngineInterface .MacroEngineServer	dieses Package enthält den Macroflow Engine Simulator die Basisklasse des Macroflow Engine Simulator (siehe Kap. 9.1) generiert Aufrufe und leitet diese an den Incoming Adapter weiter das GUI des Macroflow Engine Simulator das Remote Interface für den MacroEngineServer der Server Thread für die Entgegennahme von einlangenden Antworten des Dispatchers
RBDEngines.microengine .MicroEngine .MicroEngineGUI .MicroEngineInterface .MicroEngineServer	dieses Package enthält die Microflow Engine die Basisklasse der Microflow Engine (siehe Kapitel 9.4) das GUI der Microflow Engine das Remote Interface für den MicroEngineServer der Server Thread für die Entgegennahme von einlangenden Requests des Dispatchers

Projektstruktur und Deployment

RBDServer.server	dieses Package enthält den Dispatcher
.CallbackInternal	eine Hilfsklasse für das Handling von Callbacks
.ClientService	Handling von Client Service Objekten
.ClientServiceGroup	Handling von Client Service Group Objekten
.ClientServiceQoS	Handling von QoS-Attributen für Client Services und Client Service Gruppen (derzeit nicht unterstützt)
.IncomingRequest	Handling von Incoming Requests
.RequestService	Handling der Zuordnungen von Client Services zu Incoming Requests
.RuleEngine	Handling von Rule Engines
.RuleEngineResult	Handling des Ergebnisses von Rule Engines bzw. der Auswertung von Rule Scripts
.RuntimeStats	Handling von Runtime Statistics
.ScriptTemplate	Handling von Script Templates
.ServiceBeanInfo	Hilfsklasse für das Handling von Client Service Adaptern und Rule Engines
.ServiceInvoContext	hält Informationen über den Aufrufkontext für Client Services
RBDServer.server.common	dieses Package enthält diverse Hilfsklassen
.AttTypeVal	implementiert ein Attribute/Type/Value - Tripel
.Callback	hält die für die Durchführung von Callbacks nötigen Informationen
.Config	hält diverse Basis-Konfigurationseinstellungen
.ConfigState	die Statusinformationen für die Konfiguration
.NameDscPair	implementiert ein Name/Description - Paar
.RBDException	die von diversen Methoden geworfene Exception
.RBDStatus	das Statusobjekt - enthält Statuscodes und Fehlermeldungen; ist Returnwert vieler Methoden des Dispatchers
.Util	implementiert diverse Hilfsfunktionen
RBDServer.server.entities	dieses Package enthält alle benötigten Entity Beans ...
.ClientServiceEBean	...für Objekte der Entität DISPATCHER.CLIENT_SERVICE
.ClientServiceGroupEBean	...für Objekte der Entität DISPATCHER.CLIENT_SERVICEGROUP
.IncomingRequestEBean	...für Objekte der Entität DISPATCHER.INCOMING_REQUEST
.RequestServiceEBean	...für Objekte der Entität DISPATCHER.REQUEST_SERVICE
.RuleEngineEBean	...für Objekte der Entität DISPATCHER.RULE_ENGINE
.RuleScriptEBean	...für Objekte der Entität DISPATCHER.RULE_SCRIPT
.RuntimeStatsEBean	...für Objekte der Entität DISPATCHER.RUNTIME_STATS
.ScriptTemplateEBean	...für Objekte der Entität DISPATCHER.SCRIPT_TEMPLATE
.ServiceCallbackEBean	...für Objekte der Entität DISPATCHER.SERVICE_CALLBACK
RBDServer.server.rules	dieses Package enthält Beispiele für Rule Engines
.RulesFillWeightsBean	ein Round Robin Algorithmus, der die Gewichte der Client Service Bindings berücksichtigt

Projektstruktur und Deployment

.RulesFillWeightsRemote .RulesSimpleRRBean .RulesSimpleRRRemote	das Remote Interface zu obiger Bean ein einfacher Round Robin Algorithmus das Remote Interface zu obiger Bean
RBDServer.server.rules.frag	enthält die FRAG-Erweiterungen für den Dispatcher
.FRAGParser .RBDCrit .RBDIncoming .RBDResult .RBDService .RBDTime	initialisiert die FRAG-Erweiterungen und ruft FRAG auf Hilfsklasse für diverse Definitionen das FRAG Objekt zum Handling von Incoming Requests das Ergebnis von FRAG Auswertungen das FRAG Objekt für die Verarbeitung von Client Services das FRAG Objekt für Date/Time-Manipulationen
RBDServer.server.sessions	dieses Package enthält alle benötigten Session Beans
.BeanInvoker .IncomingMessage .RBDConfiguratorBean .RBDConfiguratorRemote .RBDDBConnectorBean .RBDDBConnectorRemote .RBDProcessorBean .RBDProcessorRemote .RBDServiceCallbackBean .RBDServiceCallbackRemote	Hilfsklasse für den Aufruf von Enterprise- und Session Beans die Incoming Message, welche vom Incoming Adapter an den Dispatcher gesendet wird die Configurator Komponente des Dispatchers das Remote Interface der Configurator Komponente die Komponente DB-Connector das Remote Interface des DB-Connectors die Prozessor Komponente des Dispatchers das Remote Interface der Prozessor Komponente der Einsprungspunkt des Dispatchers für Callbacks der Client Services im Falle von managed Callbacks das Remote Interface für obige Bean

Tabelle 74 - Projektstruktur

8.2 Deployment

In Abbildung 31 sind die für das Funktionieren des Dispatchers nötigen Komponenten sowie die Struktur der Deployments dargestellt.

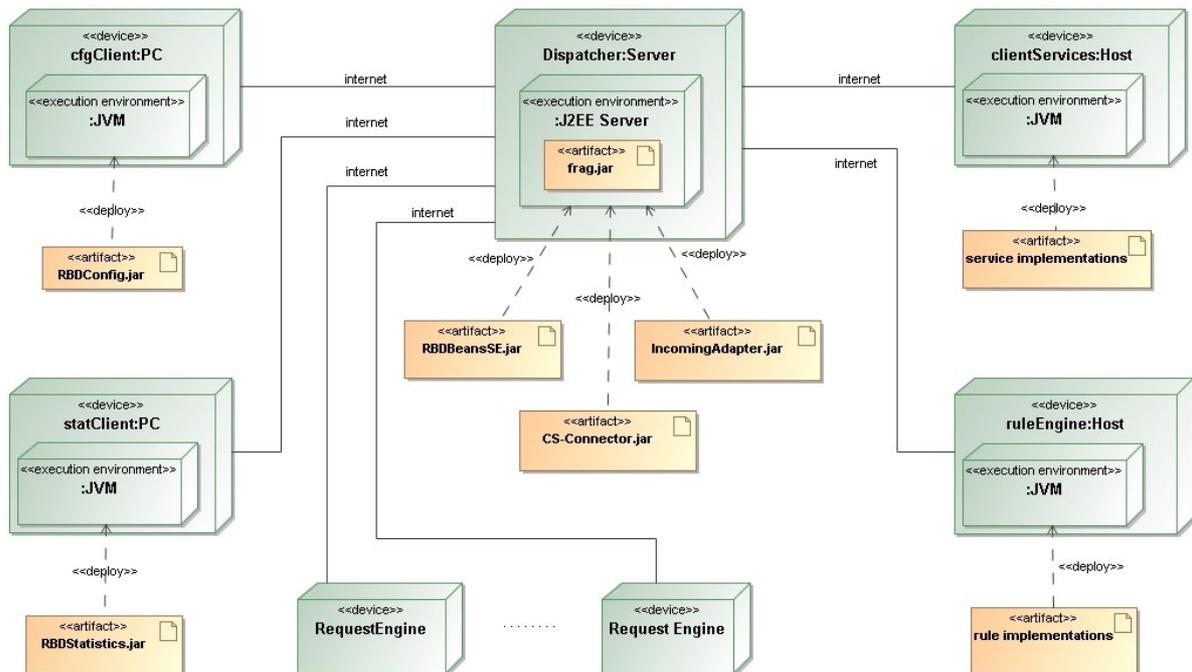


Abbildung 31 - Deployment

Der Configuration Client läuft in einer Java Virtual Machine auf einem beliebigen, möglicherweise entfernten Device *cfgClient*. Die Funktionalität ist im Package *RBDConfig.jar* enthalten, dieses Package muss auf *cfgClient* deployed worden sein um die Konfiguration durchführen zu können.

Analoges gilt für das Package *RBDStatistics.jar*, welches auf einem Device *statClient* installiert worden sein muss, um die Funktionalität des Statistics Client nutzen zu können.

Der Dispatcher selbst läuft auf einem Dispatcher:Server in einem J2EE Applikationsserver. Die gesamte Funktionalität ist im Package *RDBBeansSE.jar* enthalten, dieses Package muss auf den Applikationsserver deployed werden. Für jeden Incoming Requests muss ein Incoming Adapter definiert sein. In Abbildung 31 ist ein solcher Adapter beispielhaft im Package *IncomingAdapter.jar* enthalten. Gleiches gilt für die Outgoing Adapter, welche in der Abbildung oben beispielhaft im Package *CS-Connectors.jar* enthalten sind.

Anmerkung: Die Namen dieser Packages können beliebig gewählt werden, in der Testapplikation (siehe Kapitel 9) befinden sich der Incoming Adapter sowie alle Outgoing Adapter im Package *MIFConnectors.jar*.

Anmerkung: Da alle Methoden des Dispatchers über das remote Interface der jeweiligen Java Bean angesprochen werden, können Incoming/Outgoing Adapter auch auf entfernten Devices installiert werden.

9 Testumgebung

In diesem Kapitel werden die Testapplikationen Macroflow Engine Simulator und Microflow Engine sowie die verwendeten Incoming und Outgoing Adapter beschrieben.

9.1 Macroflow Engine Simulator

Um den Dispatcher testen zu können, wurde im Rahmen dieses Projektes ein Requester entwickelt (der *Macroflow Engine Simulator*), welcher eine Reihe von Anfragen absetzt und die Ergebnisse dieser am Bildschirm anzeigt.

In Abbildung 32 ist diese Applikation dargestellt.

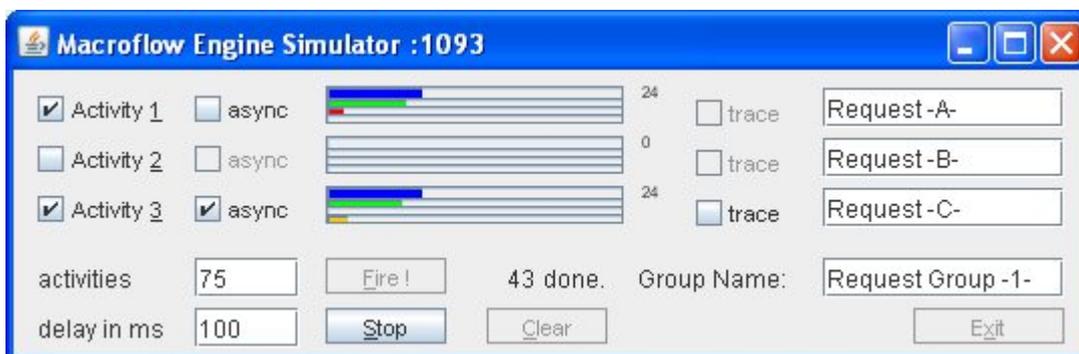


Abbildung 32 - Macroflow Engine Simulator

Der Simulator ist in der Lage, drei unterschiedliche Requests zu generieren (*Activity 1* bis *Activity 3* in der Abbildung oben) - die IDs dieser Aktivitäten werden in den Textfeldern definiert ("*Request -A-*", ...), ebenso die ID der Request Gruppe.

Im unteren Teil des Dialoges werden die Anzahl der in einem Testlauf insgesamt zu generierenden Requests im Eingabefeld *activities* sowie die Verzögerung zwischen den einzelnen Aufrufen in Millisekunden im Eingabefeld *delay in ms* definiert.

Für jeden Request ist durch Behaken der entsprechenden Checkboxes steuerbar, ob der Aufruf des Dispatchers synchron oder asynchron erfolgen soll. Im asynchronen Fall kann durch Behaken der Checkbox *trace* zusätzlich gesteuert werden, ob es sich um einen managed Callback (behakt) handelt oder nicht.

Die Anzahl der in einem Testlauf bereits gesendeten Anfragen wird bei jedem Request durch den obersten (blauen) Balken angezeigt, die Anzahl der erfolgreich erhaltenen Antworten durch den zweiten (grünen) Balken. Konnten Requests nicht erfolgreich verarbeitet werden, wird das im dritten (roten) Balken dargestellt. Der unterste (braune) Balken zeigt die Anzahl jener asynchronen Requests, die zwar abgesetzt worden sind, für die die Antwort des Dispatchers aber noch nicht eingetroffen ist.

Mit der Schaltfläche **Fire!** wird die Verarbeitung gestartet, **Stop** beendet die Verarbeitung und **Clear** setzt den Dialog wieder in seinen Ausgangszustand zurück.

9.2 Incoming Adapter

Der Macroflow Engine Simulator verwendet Incoming Adapter, um mit dem Dispatcher zu kommunizieren. Ein Beispiel für einen solchen Adapter liegt im Package RBDENGINES.CONNECTORS - siehe Tabelle 74 in Kapitel 8.1.

Wie in Kapitel 3.2.2 dargestellt, hat der Adapter die Aufgabe, die Daten zwischen Requester und Dispatcher zu konvertieren sowie einen Einsprungspunkt für Callbacks des Dispatchers zur Verfügung zu stellen.

Im folgenden Codesegment ist das Remote Interface des Incoming Adapters dargestellt.

```
package connectors;

import java.util.Hashtable;
import javax.ejb.Remote;

import server.common.Callback;
import server.common.RBDStatus;

// This is the invoker for the Rule Based Dispatcher used by
// the Macroflow Engine Simulator
@Remote

public interface IncomingAdapterRemote
{
    // Interface exposed to Requesters
    public RBDStatus invokeDispatcher(String reqGroupName, String
        requestName, Hashtable<String, Object> params);

    // Interface exposed to the Dispatcher
    public RBDStatus callbackService(String correlationID, Callback
        callback, Hashtable<String, Object> params);
}
```

Im Beispiel erhält der Incoming Adapter Anfragen seiner Requester über die Methode `invokeDispatcher()`, der Dispatcher ruft den Adapter über das `callbackService()` auf.

Der Adapter erhält vom Requester Daten in Form einer `Hashtable<String, Object>` und konvertiert diese in das für den Dispatcher verständliche Format Incoming Message. Er erstellt auch ein Callback-Objekt, dann wird der Dispatcher aufgerufen, wie in folgendem Codesegment dargestellt:

```
// now invoke the dispatcher with the input data just created
try {
    BeanInvoker procBean = new BeanInvoker(Config.PROCBEAN_URL,
        Config.PROCBEAN_NAME);
    status = (RBDStatus)procBean.invoke("processActivity", myIncoming,
        myCallback);
} catch (Exception x) {
    status.setStatus(RBDStatus.ERR_BEAN);
    status.setMessage(Util.status("Error invoking RBDProcessorBean: " +
        x.getMessage()));
}
return status;
```

Wird der Adapter per Callback aufgerufen, werden die Daten des Dispatchers konvertiert und das Ergebnis an den Requester zurückgegeben.

```

// otherwise everything should be OK - we try to connect to the
// Macroflow Engine Simulator
try {
    MacroEngineInterface remoteMAFE =
        (MacroEngineInterface)Naming.lookup(myEngine);
    if (remoteMAFE != null)
    {
        // the Macroflow Engine Simulator was found - invoke it
        remoteMAFE.invoke(correlationID, status.isOK(),
            sessID.longValue());
    }
} catch (Exception x)
{
    status.setStatus(RBDStatus.ERR_REQUEST);
    status.setMessage(Util.status("Can not connect to Macroflow " +
        "Engine Simulator" + x.getMessage()));
}
return status;

```

9.3 Outgoing Adapter

Der Dispatcher verwendet Outgoing Adapter, um mit den Client Services zu kommunizieren. Beispiele für solche Adapter liegen im Package RBDENGINES.CONNECTORS - siehe Tabelle 74 in Kapitel 8.1.

Wie in Kapitel 3.2.3 dargestellt, haben diese Adapter die Aufgabe, die Daten zwischen Dispatcher und Client Services zu konvertieren, ihre Services aufzurufen, die Ergebnisse dieser Aufrufe entgegenzunehmen, zu konvertieren sowie eventuell ein Callback-Service (des Dispatchers bei managed Callbacks bzw. der Incoming Adapter bei non-managed Callbacks) anzustoßen.

Ein Beispiel für einen solchen Outgoing Adapter wurde bereits in Kapitel 4.2.2 vorgestellt. Im folgenden Codesegment ist das Remote Interface des Callback Service des Dispatchers dargestellt.

```

package server.sessions;

import java.util.Hashtable;
import javax.ejb.Remote;

import server.common.Callback;
import server.common.RBDStatus;

@Remote
public interface RBDServiceCallbackRemote
{
    // the activity which fired the callback is identified by its
    // correlation-ID
    public RBDStatus activityCallback(String correlationID, Callback
        callback, Hashtable<String, Object> params);
}

```

9.4 Microflow Engine

Microflow Engines simulieren in unserer Testumgebung die Client Services. Es stehen 2 Microflow Engines mit jeweils drei Client Services zur Verfügung, welche in der Lage sind, die Requests des Macroflow Engine Simulators zu bedienen.

In unserem Beispiel (Kapitel 9.1, Abbildung 32) wurden die Requests auf die beiden Microflow Engines durch einen gewichteten Round Robin Algorithmus verteilt. Das Ergebnis des Testlaufes ist in Abbildung 33 zu sehen.

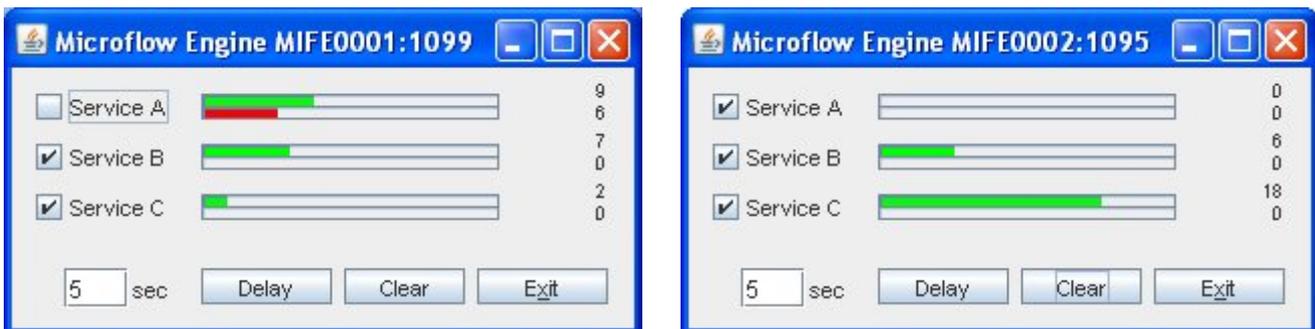


Abbildung 33 - Microflow Engines

Die Checkboxen vor den Services dienen dazu, Fehlersituationen zu simulieren. Ist ein Service enthakt, so bedeutet das "Service nicht verfügbar", der Outgoing Adapter meldet an den Dispatcher einen technischen Fehler. Sind für den gerade verarbeiteten Incoming Request weitere Services definiert, welche in einem solchen Fall angestoßen werden können, so werden diese der Reihe nach verwendet, bis ein "funktionierendes" Service gefunden worden ist.

In diesem Beispiel ist der Round Robin Algorithmus aber so implementiert, dass von der Rule Engine immer nur genau ein mögliches Service für einen Incoming Request geliefert wird. Somit führt das Enthaken von Service A auf Microflow Engine 1 zu der in Abbildung 32 durch den roten Balken bei Activity 1 dargestellten Fehlersituation beim Requester.

Durch die Schaltfläche wird die im davor liegenden Eingabefeld definierte Anzahl an Sekunden als Verzögerungszeitraum für asynchrone Verarbeitungen gesetzt. Das ist jener simulierte Zeitraum, den das Service für die Verarbeitung benötigt und nach der die Antwort an das Callback-Service geliefert wird.

Die Schaltfläche setzt den Dialog wieder in seinen Ausgangszustand zurück.

10 Vergleich mit bestehenden Applikationen

In diesem Abschnitt werden die Features des Dispatchers mit denen von marktgängigen ähnlichen Applikationen verglichen und die Unterschiede - soweit aus der Dokumentation dieser Systeme ableitbar - dargestellt. Wichtig ist, dass nur jene Funktionalitäten betrachtet werden, welche nicht unter die in Kapitel 2.2 dargestellten Abgrenzungen fallen.

10.1 IBM WebSphere

IBM WebSphere ist ein Enterprise Service Bus, welche den Austausch von Nachrichten zwischen Applikationen in heterogenen Netzwerken erlaubt. Dazu können in WebSphere *message flows* erzeugt werden, durch welche die beteiligten Applikationen verbunden werden - siehe [WebSphere P1] und [WebSphere P3a] für eine Übersicht über mögliche Workflow Features. Diese *message flows* werden über die Verbindung von *message nodes* und *message queues* implementiert.

Im Zuge der Verarbeitung einer Anfrage können neue Nachrichten erzeugt oder bestehende Nachrichten um zusätzliche Daten angereichert werden. WebSphere stellt dazu eine eigene Sprache ESQL (SQL mit "message specific extensions") zur Verfügung. Mit ESQL können überdies zur Laufzeit Daten aus Datenbanken ermittelt bzw. in diesen abgelegt werden. In Abbildung 34 ist ein Teil eines Message Flow zur Verarbeitung einer Kundenanfrage dargestellt (Grafik übernommen aus [WebSphere P1]).

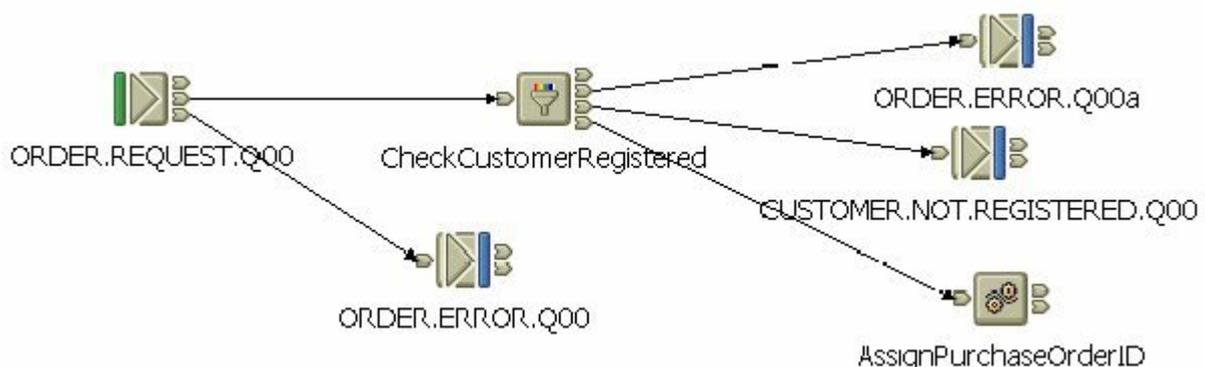


Abbildung 34 - WebSphere: ProcessPurchaseOrder

Jeder Knoten in einem solchen Flow besitzt *Input-Terminals* und *Output-Terminals*, über welche das Queueing abgewickelt wird. Im Knoten selbst können die Entscheidungen über das Routing eines Request mittels Regeldefinition mit ESQL vorgenommen werden. In Abbildung 35 ist ein einfaches ESQL-Skript dargestellt (Skript übernommen aus [WebSphere P1]; siehe [ESQL-Ref] für eine Übersicht über die von ESQL gebotenen Sprachelemente).

Im Beispiel wird die Customer-ID aus den Inputdaten ermittelt, dann wird getestet, ob diese in der Datenbank vorhanden ist und abhängig davon TRUE oder FALSE geliefert. Aufgrund des Ergebnisses wird entschieden, an welchen der drei Output-Terminals die Message weitergeleitet wird.

Vergleich mit bestehenden Applikationen

```

CREATE FILTER MODULE ProcessPurchaseOrderFlow_CheckCustomerRegistered
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN

    DECLARE customerID INTEGER;

    -- extract the customer ID from the message
    SET customerID = Root.MRM.customerDetails.customerID;

    --if the customerID cannot be found in the message then we return unknown
    IF (customerID is NULL) THEN
      RETURN UNKNOWN;
    END IF;

    -- search the customer table for the customerID
    SET Environment.Variables = THE (SELECT T.* FROM Database.SAMPLE.CUSTOMER AS T WHERE
    T.CUSTOMERID = customerID);

    --if the customerID is found return true
    IF (CARDINALITY(Environment.Variables[])=0) THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;

  END;
END MODULE;

```

Abbildung 35 - WebSphere: using ESQL for message routing

An dieser Stelle soll keine Untersuchung durchgeführt werden, in welchen Details die Möglichkeiten, welche die von uns verwendete Regelsprache FRAG bietet, von jenen abweichen, die durch ESQL implementiert werden können. Generell gilt, dass FRAG in seinem Funktionsumfang an objektorientierte Sprachen angelehnt ist, wohingegen ESQL eine Erweiterung der Datenbank-Abfragesprache SQL darstellt.

Abgesehen von den unterschiedlichen Sprachkonstrukten bleibt festzuhalten, dass ein Datenbankzugriff zum Zeitpunkt der Auswertung von FRAG Rule Scripts mit den in Kapitel 6.2 vorgestellten FRAG-Erweiterungen nicht möglich ist. Dazu müsste ein eigenes RBDDatabase-Objekt geschaffen werden beziehungsweise die Möglichkeit, eine beliebige, als Enterprise JavaBean oder Webservice vorliegende Funktion, im Zuge der FRAG-Auswertung aufzurufen. Siehe dazu auch Kapitel 10.4, Rule Script Plugins.

Anmerkung: Werden die Dispatching-Regeln nicht über ein FRAG Rule Script sondern über eine externe Rule Engine definiert, so stehen dieser selbstverständlich alle Möglichkeiten - auch die Durchführung von Datenbankzugriffen - offen.

Durch die Möglichkeit des Datenbankzuges können in WebSphere auch statistische Informationen in der Datenbank abgelegt und für die Auswertung von Regeln wieder herangezogen werden. Hier liegt ein weiterer Vorteil von WebSphere, denn die für die Auswertung von Regeln verfügbaren statistischen Informationen werden in unserem Fall vom Dispatcher vorgegeben und können nicht im Zuge der Regelauswertung frei erzeugt werden. Eine entsprechende Erweiterung des Dispatchers wäre aber leicht machbar.

Vergleich mit bestehenden Applikationen

10.2 Mule

Ähnlich wie WebSphere ist Mule ein messaging framework zum Austausch von Nachrichten zwischen Applikationen in unterschiedlichen Umgebungen. Mule ist herstellerneutral, setzt keine bestimmten Applikationsserver oder Messagesserver voraus und kann laut Eigendefinition unterschiedlichste Komponenten integrieren (siehe [Mule 2008]).

In Abbildung 36 ist ein vereinfachter Ausschnitt aus der Mule-Komponentenarchitektur dargestellt. (Grafik übernommen aus [Mule 2008])

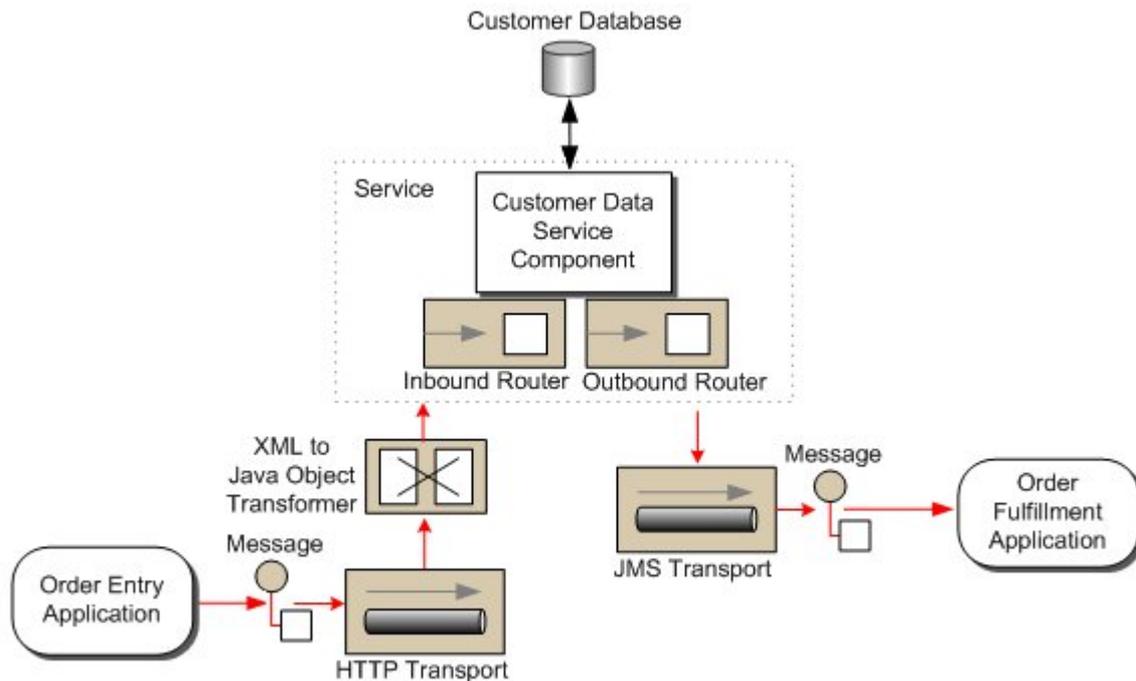


Abbildung 36 - Mule: Order Message Flow

In Mule gibt es eine strenge Trennung zwischen Message-Transport und der Business-Logik, welche nötig ist, um einlangende Messages zu verarbeiten. Mule ist für den Transport zuständig, die Verarbeitung erfolgt in *Customer Service Components*. Diese können jede Art von Logik enthalten, unter anderem auch Zugriffe auf Datenbanken, und sind vom Message Transport komplett abgeschottet.

Um sicherzustellen, dass die *Customer Service Component* die richtigen Messages erhält und diese nach der Verarbeitung korrekt weiter routet, werden *Inbound* und *Outbound Router* spezifiziert.

Inbound Router filtern, aggregieren und passen die Reihenfolge von einlangenden Messages an, bevor diese an die *Service Component* zur Verarbeitung weitergegeben werden. Nach der Verarbeitung bestimmt der *Outbound Router*, an wen die verarbeitete Nachricht weitergeleitet wird.

Falls nötig, werden die einlangenden Daten vor der Weiterleitung an das *Customer Service* durch Transformers in ein für das Service verständliches Format umgewandelt.

Vergleich mit bestehenden Applikationen

10.2.1 Inbound Router

Die Dispatchinglogik wird also von den Routern implementiert. Mule bietet eine ganze Reihe solcher vorgefertigter Router an, welche stark an die Enterprise Integration Patterns in [Hohpe et al. 2004] angelehnt sind. In folgender Tabelle werden einige dieser Router vorgestellt und erklärt, ob und wie die von diesen gebotene Funktionalität in unserem Dispatcher abgebildet werden kann.

Inbound Router	Beschreibung	Abbildung im Dispatcher
No Router	alle Messages werden an die Service Component weitergeleitet	✔ durch Implementierung von "other"-IDs, um auch Requests mit unbekanntem IDs verarbeiten zu können - siehe Kapitel 4.1.1
Selective Consumer	nur Messages, die bestimmten Kriterien genügen, werden an das Service weitergeleitet	✔ Durch Request-ID und Request-Gruppe implementiert ("Specifying Producer" im Sinne von [Hohpe et al. 2004]); Messages, welche den Anforderungen nicht genügen, können abgelehnt oder an ein spezielles Client Service weitergeleitet werden. Siehe Forwarding in Kapitel 10.2.3
Idempotent Receiver	Stellt sicher, dass Messages nur einmalig angenommen werden (Check gegen Message-ID)	✘ nicht direkt implementiert, ist aber durch die Zuordnung externer Rule Engines abbildbar. Diese sind in ihrer Funktionalität nicht beschränkt und können die nötigen Prüfungen durchführen.
Collection Aggregator und Message Chunking Aggregator	Gruppiert Messages mit derselben Correlation-ID innerhalb eines vorgegebenen Timeouts und liefert die gesamte Gruppe an das Service	✘ nicht implementiert; der Dispatcher ist stateless implementiert und hat somit keine Komponente, welche aktiv auf einlangende Anfragen wartet
Forwarding Router	leitet eine einlangende Message direkt an einen Outbound Router weiter, ohne die Service Component aufzurufen	✔ sind für einen Incoming Request weder ein FRAG Rule Script noch eine externe Rule Engine spezifiziert, so wird der Request direkt an das zugeordnete Client Service geroutet
Custom	die einlangende Message wird nur an das Service weitergeleitet, wenn sie bestimmten Regeln genügt	✔ analog Selective Consumer oben

Tabelle 75 - Mule Inbound Routers

Vergleich mit bestehenden Applikationen

10.2.2 Outbound Router

In folgender Tabelle werden die wichtigsten von Mule unterstützten Outbound Router vorgestellt und erklärt, ob und wie die von diesen gebotene Funktionalität in unserem Dispatcher abgebildet werden kann.

Outbound Router	Beschreibung	Abbildung im Dispatcher
Pass-through	alle Messages werden an den einzigen möglichen Endpoint weitergeleitet	✔ durch Zuordnung des Endpoint (Client Service) ohne Rule Script oder externe Rule Engine anzugeben
Filtering	nur Messages, die bestimmten Kriterien genügen, werden an den Endpoint weitergeleitet	✔ durch die Regeldefinitionen werden die Bedingungen festgelegt, unter denen eine Liste von relevanten Client Services ermittelt wird
Multicasting	Messages werden gleichzeitig an mehrere Endpoints versendet	✘ im Dispatcher nicht standardmäßig abgebildet (Abgrenzung 3) kann aber im Incoming Adapter des Request implementiert werden - siehe Chaining und Kapitel 10.4.
Chaining	Messages werden an eine Reihe von Endpoints versendet, wobei immer das Ergebnis eines Aufrufes den Input des nächsten Aufrufes darstellt	✘ kann im Incoming Adapter eines Request abgebildet werden, indem die Message nach dem Ende der Verarbeitung durch ein Client Service als "von diesem Service verarbeitet" markiert und in den Dispatcher-Regeln auf diese Markierung eingegangen wird.
List Message Splitter	teilt eine Message in logische Einheiten und sendet die Einzelteile - versehen mit Correlation-ID und Sequenznummern an verschiedene Endpoints	✘ siehe Abgrenzung 4; kann durch den Incoming Adapter implementiert werden, der in der Folge auch für das Aggregieren der Einzelergebnisse verantwortlich ist
Message Chunking Router	teilt eine Message in Teile vorgegebener Größe und leitet diese Teile an denselben Endpoint weiter	✘ rein technisches Splitting; ist keine Anforderung an den Dispatcher und wird nicht unterstützt
Exception Based Router	sendet eine Message an den ersten aus einer Liste von Endpoints, der die Message entgegennehmen kann	✔ Standardverfahren des Dispatchers bei Zuordnung mehrerer Client Services zu einem Incoming Request
Endpoint Template	erlaubt es, eine Message abhängig von deren Inhalt an unterschiedliche Endpoints weiterzuleiten	✔ Standardverfahren des Dispatchers - die Liste der in Frage kommenden Client Services wird zur Laufzeit anhand der Regeln oder Rule Engines ermittelt
Custom	der passende Endpoint für eine Message wird über definierbare Regeln festgelegt	✔ Standardverfahren des Dispatchers

Tabelle 76 - Mule Outbound Routers

Vergleich mit bestehenden Applikationen

10.2.3 Asynchronous Reply Routers und Catch-all Strategie

Neben den oben genannten Inbound und Outbound Routers stellt Mule noch einige sogenannte Asynchronous Reply Routers zur Verfügung. Diese werden verwendet, um die Antworten auf eine Menge von asynchron (und somit parallel) verarbeiteten Requests zu sammeln und konsolidiert an den Aufrufer zurückzuliefern. Da der Dispatcher diese Art der Verarbeitung nicht standardmäßig unterstützt (Abgrenzung 4), werden sie hier auch nicht näher ausgeführt.

Interessanter sind Catch-all Strategien. Diese werden eingesetzt, wenn für das Routing einer Message kein Endpoint ermittelt werden konnte. Für diese Fälle kann in Mule ein eigener Endpoint angegeben werden, an den alle diese "nicht zustellbaren" Messages geroutet werden. Die in Tabelle 77 dargestellten Strategien werden unterstützt.

Catch-all Strategie	Beschreibung	Abbildung im Dispatcher
Forwarding	alle unzustellbaren Messages werden an diesen Endpoint weitergeleitet	✔ durch die Vergabe von Zuordnungsattributen im Dialog Service Assignment Parameters können bestimmte Client Services speziell gekennzeichnet werden. In den FRAG Regeln oder den externen Rule Engines kann auf diese Marker abgefragt und der Request entsprechend geroutet werden.
Custom Forwarding	analog Forwarding, allerdings können zusätzliche Regeln für das Routing angegeben werden	✔ im Dispatcher abbildbar - siehe Forwarding
Logging	es erfolgt kein Routing, aber eine Warnung wird in den Routing Log geschrieben	✔ Logging wird standardmäßig im Dispatcher unterstützt, allerdings erfolgt die Ausgabe derzeit nur auf die Konsole des Applikationsservers. Ein maßgeschneidertes Logging ist durch externe Rule Engines ebenfalls möglich - siehe Custom unten.
Custom	Im Falle eines Fehlers erfolgt kein Routing, es wird aber eine vorab definierte Funktion ausgeführt	✔ Durch die Möglichkeit der Zuordnung von externen Rule Engines im Dispatcher möglich. Kann durch ein FRAG Rule Script kein passendes Client Service ermittelt werden, so besteht die Möglichkeit, die Verarbeitung an eine externe Rule Engine zu übergeben. Diese ist in Ihrer Funktionalität nicht eingeschränkt, und kann jede beliebige Funktionalität ausführen

Tabelle 77 - Mule Catch-all Strategien

Vergleich mit bestehenden Applikationen

10.3 FUSE Mediation Router

Das Produkt FUSE Mediation Router ist eine Open Source Distribution von Apache Camel, die von IONA Technologies vertrieben wird. Für die Beschreibung von Funktionalität und Einsatzmöglichkeiten siehe [Fuse 2008] und [Apache 2008].

Auch FUSE bezieht sich in den Routing Strategien auf die in [Hohpe et al. 2004] dargestellten Patterns. Zusätzlich zu den in Kapitel 10.2 angeführten Patterns werden noch die Throttler- und Delayer-Strategie (in FUSE *processors* genannt) unterstützt.

Processor	Beschreibung	Abbildung im Dispatcher
Throttler	gewährleistet, dass keine zu hohe Last auf die Endpoints geleitet wird. Die maximale Anzahl an weitergeleiteten Messages pro Sekunde kann vorab angegeben werden, ist diese erreicht, werden Messages gebuffert bis der Endpoint wieder in der Lage ist, weitere Messages entgegenzunehmen.	 da die Last an jedem Client Service dem Dispatcher bekannt ist und auf die statistischen Informationen in der Regelauswertung zugegriffen werden kann, ist ein analoges Verfahren im Dispatcher abbildbar. Die Maximalanzahl an Requests kann in den Regeln hardcodiert oder durch die Vergabe von Zuordnungsattributen im Dialog Service Assignment Parameters festgelegt werden. Die Vorgabe einer Zeiteinheit wird nicht unterstützt.
Delayer	verzögert die Weiterleitung von Messages. Die Verzögerung kann entweder relativ sein (Weiterleitung nach 2 Sekunden) oder erst erfolgen, wenn ein absoluter Zeitpunkt erreicht ist (Weiterleitung um 11:30:00)	 im Dispatcher über external Rule Engines abbildbar; da der Dispatcher stateless implementiert ist und jede Session bis zum Ende der Verarbeitung erhalten bleibt, kann es bei einer größeren Anzahl solcher Requests zu Ressourcenproblemen am Applikationsserver kommen.
Load Balancer	die Message wird an jenen aus einer Gruppe von Endpoints weitergeleitet, der im Moment die geringste Last zu verarbeiten hat	 die Last an jedem Client Service ist dem Dispatcher bekannt, die Dispatching-Regeln können darauf Rücksicht nehmen - siehe Throttler. Beliebige Balancing-Policies (RoundRobin, Random, ...) können frei definiert werden.

Tabelle 78 - FUSE Routing Strategien

10.4 Mögliche Erweiterungen der Dispatcher Funktionalität

In diesem Kapitel werden einige mögliche technische und fachliche Erweiterungen für den Dispatcher besprochen.

Übergang von Stateless zu Statefull

Durch die Implementierung des Dispatchers als *statefull component* können einige der in den letzten Kapiteln besprochenen Einschränkungen gemildert werden. Das kann unter JBoss durch Einführen einer Service Bean (Annotations `@Service` und `@Management`) geschehen. Service Beans sind Singletons - es gibt also nur genau eine Instanz der Bean am Applikationsserver, diese "lebt" während der gesamten Laufzeit des Applikationsservers. Diese Bean kann folgende Aufgaben übernehmen:

- Caching Die in der Datenbank abgelegten Konfigurationsdaten werden nur beim Hochfahren des Applikationsservers bzw. zu definierten Zeitpunkten (möglicherweise timergesteuert oder auf Basis lazy acquisition) aus der Datenbank gelesen und im Speicher durch die Service Bean vorgehalten. Der Persistenzlayer des Dispatchers verwendet die Daten im Speicher, was zu erheblichen Performancesteigerungen führt. Der Nachteil liegt in der höheren Komplexität des Dispatchers.
- Queueing Incoming Requests könnten durch eine von der Service Bean verwaltete Queue zwischengespeichert werden, bis sie verarbeitet werden können - siehe die Delayer-Policy in Kapitel 10.3. Noch naheliegender ist die analoge Anwendung bei Client Services, wo durch Queueing von Aufrufen Throttler-Policies leicht implementiert werden können. Prioritäten für Incoming Requests und Client Services könnten in der Queueing-Logik zusätzlich berücksichtigt werden.

Multicasting

Bei diesem Konzept wird ein Incoming Request gleichzeitig an mehr als ein Client Service zur Verarbeitung übergeben. Diese Erweiterung ist relativ einfach implementierbar - im Konfigurationssystem müsste nur die zusätzliche Information beim Incoming Request abgelegt werden, dass die Liste der zugeordneten Client Services nicht im Sinne der standardmäßig implementierten Exception-Based Policy durchlaufen wird, sondern jedes dieser Client Services vom Dispatcher mit derselben Anfrage aufgerufen wird. Das ist natürlich nur bei asynchronen Verarbeitungen sinnvoll.

Komplex wird in diesem Fall jedoch die Callback-Komponente - sie muss erkennen, dass die einlangende Antwort eines Client Service entweder die erste aus der Anfragemenge ist - in diesem Fall müssen die Dispatcher-Statistiken angepasst und die Antwort an den Incoming Adapter des Requesters zurückgeliefert werden. Ist die Antwort nicht die erste, ist sie zu verwerfen - auch hier sind allerdings unterschiedliche Policies denkbar. Die Callback-Komponente benötigt jedenfalls entweder Zugriff auf eine Datenbank oder auf einen Memory-Cache (siehe Thema Caching oben).

Callback Plugins

Wie im Abschnitt über Multicasting angedeutet, kann die Verarbeitung in der Callback-Komponente komplex werden. Für diese Fälle wäre es unter Umständen wünschenswert, dass zu einem Incoming Request ein oder mehrere Plugins definiert werden könnten, welche beim Eintreffen der Antwort eines Client Service spezielle, Request-spezifische Aufgaben wahrnehmen. Es ist auch denkbar, dem Callback-Objekt ebenfalls Zugriff auf die

Vergleich mit bestehenden Applikationen

Rule Engines zu gewähren, um die Weiterleitung der Antworten an die Requester gezielt steuern zu können.

Rule Script Plugins

Einige der in Kapitel 10.2 dargestellten Router (Idempotent Receiver, Delayer) und Strategien (Custom Catch-all, Logging) können derzeit nur über externe Rule Engines abgebildet werden. Würde die Möglichkeit geschaffen, aus einem FRAG Rule Script auf Java Beans oder Webservices zugreifen zu können, könnte dieselbe Verarbeitung auch durch die FRAG Rule Scripts sichergestellt werden.

Dieser Zugriff ist sehr einfach durch ein weiteres FRAG-Objekt RBDINVOKER zu gewährleisten, dem die Adresse einer Enterprise Java Bean oder eines Webservice sowie die zur Ausführung des Service nötigen Parameter übergeben werden. Dieses Verfahren würde auch ermöglichen, Teile der Regelverarbeitung aus den Rule Scripts über wesentlich schnellere Enterprise Java Beans abzuwickeln, was zu deutlichen Performancesteigerungen bei der Regelauswertung führen würde.

Zeitreihenverarbeitung

Alle Änderungen, welche im Zuge der Konfiguration vorgenommen werden, werden derzeit wirksam, sobald sie in der Datenbank abgelegt sind. Das ist in der Praxis aber nicht immer wünschenswert, vor allem dann nicht, wenn Änderungen umfangreich sind und während der Konfiguration inkonsistente Zustände auftreten können.

Ein Problem tritt ebenfalls auf, wenn Änderungen genau zu einem bestimmten Zeitpunkt wirksam werden sollen. Normalerweise ist es nicht sinnvoll zu verlangen, dass die Konfiguration zu genau diesem Zeitpunkt vorgenommen werden muss. Vor allem bei umfangreichen Änderungen ist das generell unmöglich.

In der Praxis hat sich eine Strategie bewährt, bei der zu allen Änderungen auch der Zeitpunkt definiert wird, ab dem sie wirksam werden. Das kann einfach durch Angabe eines Gilt-ab Timestamp während der Konfiguration erreicht werden, ein optionaler Gilt-bis Timestamp kann die Gültigkeit einer Konfigurationseinstellung wieder aufheben. Somit können Konfigurationen, welche erst zu einem definierten Zeitpunkt in der Zukunft gültig werden, schon vorab angelegt werden.

Zur Laufzeit werden nur jene Datensätze aus der Datenbank ermittelt, deren Gilt-ab Timestamp kleiner gleich und deren Gilt-bis Timestamp größer als der aktuelle System-Timestamp sind.

Diverses

Ob ein Request synchron oder asynchron verarbeitet werden soll, könnte nicht wie bisher vom Aufrufer gesteuert werden, sondern vom Dispatcher durch Konfiguration beim Incoming Request oder auf Ebene der Client Services. Ebenfalls denkbar wäre, die Steuerung den Rule Engines zu überlassen und den Typ der weiteren Verarbeitung als Ergebnis der Regelauswertung an den Dispatcher zu liefern.

Sinnvoll ist ein solches Vorgehen vor allem bei der Entscheidung managed oder non-managed Callback - diese Unterscheidung ist dem Requester egal, der Dispatcher kann durch die Festlegung auf non-managed in jenen Fällen Performance gewinnen, in denen die Verarbeitungszeiten nicht Kriterien für die Regelauswertung darstellen.

11 Abbildungen und Tabellen

11.1 Abbildungsverzeichnis

Abbildung 1 - Process Integration Architecture	7
Abbildung 2 - Dispatcher Übersicht	14
Abbildung 3 - Komponenten Konfiguration	16
Abbildung 4 - Komponenten Laufzeit	22
Abbildung 5 - Configuration Client Hauptmenü	28
Abbildung 6 - Dialog Incoming Requests	29
Abbildung 7 - Dialog Client Services	30
Abbildung 8 - Dialog zur Auswahl von Client Service Adapter Beans	32
Abbildung 9 - Client Service Adapter Bean	32
Abbildung 10 - Dialog zur Auswahl von Client Service Web Services	33
Abbildung 11 - Client Service Web Service	33
Abbildung 12 - Informationen aus Web Service Adapter	35
Abbildung 13 - Informationen aus Bean Adapter	35
Abbildung 14 - Dialog Rule Engines	36
Abbildung 15 - Fehlermeldung bei Löschversuch von Rule Engines	37
Abbildung 16 - Dialog Rule Configuration	38
Abbildung 17 - Dialog Service Assignment Parameters	40
Abbildung 18 - Client Services mit Zuordnungsattributen	40
Abbildung 19 - Dialog Rule Script Definition	41
Abbildung 20 - Rule Script Auswertung	42
Abbildung 21 - Rule Script mit simulierten Request Parametern	43
Abbildung 22 - Anlage eines Script Template	44
Abbildung 23 - Verwendung eines Script Template	44
Abbildung 24 - Dialog Statistics Client	45
Abbildung 25 - Laufzeit synchron	47
Abbildung 26 - Laufzeit asynchron	48
Abbildung 27 - Auswahl Client Service	50
Abbildung 28 - Tracing	51
Abbildung 29 - Entity Relationship Diagramm	82
Abbildung 30 - Projektstruktur	94
Abbildung 31 - Deployment	98
Abbildung 32 - Macroflow Engine Simulator	99
Abbildung 33 - Microflow Engines	102
Abbildung 34 - WebSphere: ProcessPurchaseOrder	103
Abbildung 35 - WebSphere: using ESQL for message routing	104
Abbildung 36 - Mule: Order Message Flow	105

11.2 Tabellenverzeichnis

Tabelle 1 - Beschreibung der verwendeten Ausdrücke	9
Tabelle 2 - Configuration Services	18
Tabelle 3 - Processing Services für den Configuration Client	18
Tabelle 4 - Persistenz Services für den Configurator	21
Tabelle 5 - Persistenz Services für den Processor	21
Tabelle 6 - FRAG Rule Engine Services für den Processor (Simulation)	21
Tabelle 7 - Processing Services für Incoming Adapter	23
Tabelle 8 - Callback Service für den Dispatcher	23
Tabelle 9 - Callback Service für den Outgoing Adapter	24
Tabelle 10 - Persistenz Services für den Processor	26

Abbildungen und Tabellen

Tabelle 11 - FRAG Rule Engine Services für den Processor (Laufzeit)	26
Tabelle 12 - Interface externer Rule Engine Services	26
Tabelle 13 - Processing Services für Statistics Client	27
Tabelle 14 - Statistics Client - Tabboxspalten	46
Tabelle 15 - Rule Engine Interface	52
Tabelle 16 - Rule Engine Parameter	54
Tabelle 17 - Rule Engine Ergebnisdefinition	54
Tabelle 18 - FRAG Erweiterung	55
Tabelle 19 - RBDTime Methoden	56
Tabelle 20 - RBDTime format	56
Tabelle 21 - RBDTime parse	56
Tabelle 22 - RBDTime now	57
Tabelle 23 - RBDService Methoden	59
Tabelle 24 - RBDService paramList	60
Tabelle 25 - RBDService contains	61
Tabelle 26 - RBDService search	62
Tabelle 27 - RBDService getService	62
Tabelle 28 - RBDService restrict	63
Tabelle 29 - RBDService numServices	63
Tabelle 30 - RBDService featureList	64
Tabelle 31 - RBDService featureKeys	65
Tabelle 32 - RBDService featureValues	65
Tabelle 33 - RBDService numFeatures	66
Tabelle 34 - RBDService hasFeature	66
Tabelle 35 - RBDService getFeature	66
Tabelle 36 - RBDService sort	67
Tabelle 37 - RBDService buildResult	68
Tabelle 38 - RBDIncoming Methoden	69
Tabelle 39 - RBDIncoming groupName	70
Tabelle 40 - RBDIncoming groupId	70
Tabelle 41 - RBDIncoming requestName	70
Tabelle 42 - RBDIncoming requestId	71
Tabelle 43 - RBDIncoming soapHeader	71
Tabelle 44 - RBDIncoming soapBody	71
Tabelle 45 - RBDIncoming paramList	72
Tabelle 46 - RBDIncoming paramKeys	72
Tabelle 47 - RBDIncoming paramValues	72
Tabelle 48 - RBDIncoming numParams	73
Tabelle 49 - RBDIncoming hasParam	73
Tabelle 50 - RBDIncoming clearParam	73
Tabelle 51 - RBDIncoming getParam	74
Tabelle 52 - RBDIncoming setParam	74
Tabelle 53 - RBDIncoming featureList	75
Tabelle 54 - RBDIncoming featureKeys	76
Tabelle 55 - RBDIncoming featureValues	76
Tabelle 56 - RBDIncoming numFeatures	77
Tabelle 57 - RBDIncoming hasFeature	77
Tabelle 58 - RBDIncoming getFeature	77
Tabelle 59 - RBDResult Methoden	78
Tabelle 60 - RBDResult Exception	78
Tabelle 61 - RBDResult addService	79
Tabelle 62 - RBDResult addEngine	80
Tabelle 63 - RBDResult trace	80
Tabelle 64 - RBDResult clearTrace	81
Tabelle 65 - Entität INCOMING_REQUEST	84
Tabelle 66 - Entität CLIENT_SERVICE	85
Tabelle 67 - Entität CLIENT_SERVICEGROUP	86
Tabelle 68 - Entität REQUEST_SERVICE	87

Abbildungen und Tabellen

Tabelle 69 - Entität SERVICE_CALLBACK.....	88
Tabelle 70 - Entität RULE_ENGINE.....	89
Tabelle 71 - Entität RULE_SCRIPT	90
Tabelle 72 - Entität SCRIPT_TEMPLATE	91
Tabelle 73 - Entität RUNTIME_STATS	93
Tabelle 74 - Projektstruktur	97
Tabelle 75 - Mule Inbound Routers	106
Tabelle 76 - Mule Outbound Routers	107
Tabelle 77 - Mule Catch-all Strategien	108
Tabelle 78 - FUSE Routing Strategien	109

12 Literaturverzeichnis

- [Apache 2008] Apache Camel - Enterprise Integration Patterns, <http://activemq.apache.org/camel/enterprise-integration-patterns.html>
- [Assisi 2005] R. Assisi. J2EE mit Eclipse 3 und JBoss, Carl Hanser Verlag München, 2005
- [Buschmann 2007] F. Buschmann, K. Henney, D.C.Schmidt. A Pattern Language for Distributed Computing. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2007
- [Dispatcher-Project] J. Bobik, Regelbasierter Dispatcher, Eclipse Projekt, verfügbar auf CD als Beilage zu dieser Arbeit.
- [EJB 3.0] EJB 3.0 Tutorial, <http://www.jboss.org/jbossejb3/docs/tutorial/>
- [ESQL-Ref] ESQL Reference, http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp?topic=/com.ibm.etools.mft.doc/ak04860_.htm
- [Fuse 2008] FUSE Mediation Router - Content Based Routing, <http://fusesource.com/products/enterprise-camel/>
- [Hentrich et al. 2007] C. Hentrich, U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures. In Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, pages 1-45, July, 2006.
- [Hentrich-1 et al. 2007] C. Hentrich, U. Zdun. Service Integration Patterns for Invoking Services from Business Processes. In Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007), Irsee, Germany, pages 1-45, July, 2007.
- [Hitz et al. 2005] M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger. UML@Work. dpunkt.verlag, 2005
- [Hohpe et al. 2004] G. Hohpe, B. Woolf. Enterprise Integration Patterns. Addison-Wesley 2004
- [Jain et al. 1997] P. Jain, D.C. Schmidt. Dynamically Configuring Communication Services with the Service Configurator Pattern. In C++ Report, SIGS, Vol. 9, No. 6, June, 1997.
- [JBoss 2008] JBoss Enterprise Middleware Documentation, <http://www.jboss.com/docs/index>
- [JBoss Ext 2008] JBoss EJB 3.0 Reference Documentation, Chapter 5.1, <http://docs.jboss.org/ejb3/app-server/reference/build/reference/en/html>
- [Mule 2008] Mule - Open Source ESB - SOA and Integration Platform, <http://mulesource.org/display/MULE/Home>
- [MySQL 2008] MySQL Documentation, <http://dev.mysql.com/doc/>
- [Rozanski 2007] Uwe Rozanski. Enterprise JavaBeans 3.0 mit Eclipse und JBoss, Redline GmbH, Heidelberg, 2007

Literaturverzeichnis

- [Schmidt et al. 2000] D.C. Schmidt, M. Stal, H. Rohnert and F. Buschman. Patterns for Concurrent and Networked Objects. Pattern Oriented Software Architecture. J. Wiley and Sons Ltd., 2000
- [Völter et al. 2005] M. Völter, M. Kircher, U. Zdun. Remoting Patterns. Foundations of Enterprise, Internet and Realtime Distributed Object Middleware. J. Wiley and Sons Ltd., 2005
- [WebSphere MB] WebSphere Message Broker,
<http://www-01.ibm.com/software/integration/wbimessagebroker/>
- [WebSphere P1] Integrating applications using WebSphere Business Integration Message Broker V5 -- Part 1: Integration scenario for creating a Web order processing application
http://www-128.ibm.com/developerworks/websphere/library/techarticles/0402_kalia1/0402_kalia1.html
- [WebSphere P3a] Integrating applications using WebSphere Business Integration Message Broker V5: Part 3A: Using message flows to integrate applications
http://www.ibm.com/developerworks/websphere/library/techarticles/0409_kalia3a/0409_kalia3a.html
- [WebSphere P4] Integrating applications using WebSphere Business Integration Message Broker V5: Part 4: Message transformation using Message Broker mappings
http://www.ibm.com/developerworks/websphere/library/techarticles/0506_chari4/0506_chari4.html
- [Wiki-DSL] Wikipedia, "domänenspezifische Sprache",
http://de.wikipedia.org/wiki/Dom%C3%A4nenspezifische_Sprache
- [Zdun 2006] U. Zdun, The FRAG Language, <http://frag.sourceforge.net/>, 2006