Diplomarbeit

Konzeption und Umsetzung eines effizienten XML-Ereignisfilters für inhaltsbasierte Publish/Subscribe-Systeme

von Volker Suschke

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

14. Oktober 2010

Diplomarbeit zur Erlangung des akademischen Grades

DIPLOM-MEDIENINFORMATIKER

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl Rechnernetze

Bearbeitung:

Volker Suschke (Matrikel-Nr. 3120960)

Betreuer:

Dipl.-Inf. Josef Spillner (TU Dresden)
Dr.-Ing. Eberhard Grummt (SAP AG)

Verantwortlicher Hochschullehrer:

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Eigenständigkeitserklärung:

Hiermit bestätige ich, dass ich die vorliegende Arbeit mit dem Titel „Konzeption und Umsetzung eines effizienten XML-Ereignisfilters für inhaltsbasierte Publish/Subscribe-Systeme“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Dresden, 14. Oktober 2010

Volker Suschke

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name, Vorname:	Suschke, Volker
Studiengang:	Medieninformatik
Matr.-Nr.:	3120960
Thema:	**Konzeption und Umsetzung eines effizienten XML-Ereignisfilters für inhaltsbasierte Publish/Subscribe-Systeme**

EINFÜHRUNG

Ereignisgesteuerte verteilte Systeme gewinnen im Unternehmensumfeld eine zunehmend große Bedeutung. Durch den asynchronen Austausch von *Ereignissen* können Anwendungen gekoppelt werden, ohne dass sie die spezifischen Schnittstellen des Kommunikationspartners kennen müssen. Eine Schlüsseltechnologie sind dabei *Publish/Subscribe-Systeme*. Sie stellen eine Kommunikationsinfrastruktur bereit, über die Anwendungen Ereignisdaten versenden und empfangen können, z. B. in einem XML-Format. Die Empfänger formulieren dabei über so genannte *Subscriptions*, an welchen Arten von Ereignissen sie interessiert sind. Die Ausdrucksmächtigkeit der verwendeten Subscription-Syntax bestimmt maßgeblich, wie genau solche Interessen formuliert werden können. *Inhaltsbasierte Subscriptions* stellen dabei den allgemeinsten Ansatz dar. Dabei können Ereigniskonsumenten (*Subscriber*) ihre Interessen mittels logischer Ausdrücke über den Inhalt von Ereignissen ausdrücken. Ihre Flexibilität impliziert aber auch besondere Anforderungen an das Publish/Subscribe-System, da dieses sehr große Mengen von Ereignissen anhand einer potenziell sehr hohen Anzahl von Subscriptions filtern muss.

SCHWERPUNKTE

In dieser Diplomarbeit soll untersucht werden, wie ein XML-Ereignisfilter für ein Publish/Subscribe-System gestaltet und umgesetzt werden kann. Dabei soll insbesondere Wert auf eine hohe Laufzeiteffizienz gelegt werden. Als Ausgangspunkt dienen dabei erste Ergebnisse aus dem Projekt ADiWa bzgl. einer XML-basierten Subscription-Sprache. Diese soll als Basis für die zu entwickelnden Algorithmen dienen und kann ggf. angepasst oder erweitert werden. Weiterhin soll untersucht werden, wie etablierte Schnittstellen und Sprachen zur Formulierung von Subscriptions (insbesondere JMS, WS-Notification und XPath) auf diese Sprache abgebildet werden können. Ausgewählte Teile des zu entwickelnden Konzepts sind prototypisch zu implementieren. Die Effizienz der Lösung ist anhand von Leistungsmessungen quantitativ nachzuweisen. Hierbei könnte die eigene Lösung auch mit einfachen Möglichkeiten der Filterung (z. B. mit der Java XPath-API) verglichen werden.
Folgende Teilaufgaben sind zu lösen:

- Systematisierung des Standes von Forschung und Technik in den relevanten Forschungsbereichen wie z. B. Publish/Subscribe-Systeme und XML-Filterung
- Formulierung von Anforderungen sowie Anwendungs- und Testfällen

- Untersuchung der Abbildbarkeit von Anfrageschnittstellen und Sprachen (insbesondere JMS und XPath) auf die vorliegende Subscription-Sprache
- Konzeption eines Systems zur Entgegennahme und Filterung von Ereignissen anhand von Subscriptions
- Implementierung ausgewählter Teile des Konzepts
- Evaluierung des Konzepts anhand von Leistungsmessungen, der Anwendungs- und Testfälle sowie der aufgestellten Anforderungen

Betreuer:	Dipl.-Inf. Josef Spillner
externer Betreuer:	Dr.-Ing. Eberhard Grummt, SAP AG
Verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Institut:	Institut für Systemarchitektur
Lehrstuhl:	Rechnernetze
Beginn am:	15.04.2010
Einzureichen am:	14.10.2010

Unterschrift des verantwortlichen Hochschullehrers

Verteiler: 1 x Prüfungsamt, 1 x HSL, 1 x Betreuer, 1 x Student

INHALT

1 EINLEITUNG

In vielen Bereichen des täglichen Lebens führen fehlende oder unzureichende Informationen zu ausbleibenden oder gar falschen Reaktionen. Ein einfaches Beispiel zeigt die in Abbildung 1 dargestellte Situation der Kommunikation in einem Unternehmen. Das Management dieses Unternehmens besteht aus einem Geschäftsführer und seinem Stellvertreter, welcher für den erfolgreichen Betrieb der Produktionsanlagen verantwortlich ist. Wäre es in dieser Situation denkbar, dass der Geschäftsführer über ungewöhnliche Ereignisse (z.B. das Abbrennen einer Fabrik) nur dann informiert wird, wenn er danach fragt? Sicher nicht, denn die meisten Organisationen verwenden mindestens implizite Verträge, in denen festgelegt ist, was und wann etwas dem Geschäftsführer oder Vorgesetzten mitgeteilt werden soll. So ist beispielsweise der Stellvertreter dafür verantwortlich, den Geschäftsführer zu informieren, wenn eine Produktionsanlage in Brand geraten ist. Es ist nicht die Aufgabe des Geschäftsführers, die wichtigen Informationen aus dem täglichen Betrieb zu extrahieren. Die gemeinsame Erwartung über das, was kommuniziert werden soll, ist somit ein implizites Abonnement (engl. subscription). [Int05]

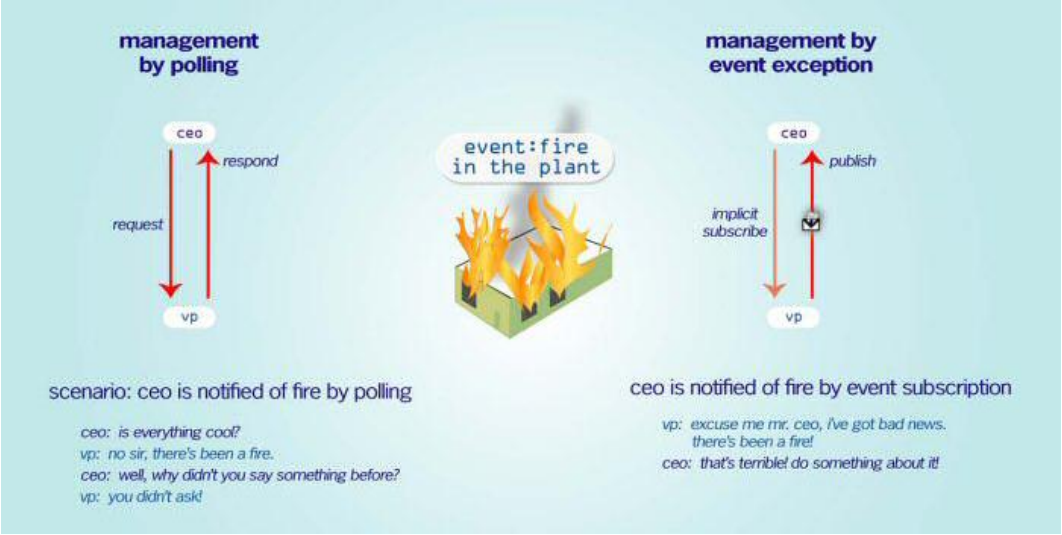


Abbildung 1: Beispiel synchroner (links) und asynchroner (rechts) Kommunikation zwischen Geschäftsführer (CEO) und seinem Stellvertreter (VP) (Quelle: [Int05])

Bereits mit diesem einfachen und weit verbreiteten Beispiel lässt sich die Funktionsweise von ereignisgesteuerten Architekturen (engl. Event Driven Architectures, EDA) beschreiben, welche die asynchrone Kommunikation und somit den Austausch von Nachrichten bzw. Ereignissen zwischen voneinander unabhängigen Komponenten ermöglichen.

Wie in diesem einfachen Beispiel zu erkennen ist, ergeben sich in bestimmten Situationen entscheidende Vorteile aus dem Einsatz von ereignisgesteuerten Architekturen und somit einer asynchronen Kommunikation. Das Zusammenwirken von einer Subscription und dem Veröffentlichen von Ereignissen (engl. publish) wird auch als

Publish/Subscribe-Paradigma bezeichnet. Eine direkte Kontrolle der beteiligten Systemkomponenten wird durch die Umsetzung des Publish/Subscribe-Paradigmas vermieden, was zu einer weitgehenden Entkopplung von Ereignisproduzenten und Ereigniskonsumenten führt. Die Vermittlung der Ereignisse wird durch das System geregelt und erfordert eine Filterung der Ereignisse, z. B. auf der Basis des Inhaltes, damit nur die relevanten Ereignisse den interessierten Empfängern zugestellt werden, ohne dass diese eine weitere Filterung vornehmen müssen.

Die vorliegende Arbeit diskutiert die Problematik einer effizienten Filterung von Ereignissen in inhaltsbasierten Publish/Subscribe-Systemen unter Berücksichtigung von technologischen, strukturellen sowie ökonomischen Herausforderungen. Dabei wird Wert darauf gelegt, nicht nur ein Konzept zur Umsetzung eines effizienten XML-Ereignisfilters zu präsentieren. Ein wichtiger Aspekt in dieser Arbeit ist ebenso die Implementierung eines Prototyps und die Durchführung von Performancemessungen, um das Systemverhalten evaluieren zu können. Dadurch wird es gleichzeitig möglich, zukünftige Ausprägungen und alternative Entwicklungen zu skizzieren und zu bewerten. Abgegrenzt wird dieses Konzept durch eine Analyse des aktuellen Stands der Technik bezüglich Filterung in Publish/Subscribe-Systemen. Auf mögliche sicherheitsrelevante Aspekte, welche sich aus der Verarbeitung von Ereignisdaten in einem verteilten Publish/Subscribe-System ergeben, wird an den relevanten Stellen hingewiesen. Eine detaillierte Analyse der Sicherheitsaspekte erfolgt in weiterführenden Arbeiten [Gei10], weshalb der Fokus vollständig auf das Konzept der Filterung und deren Umsetzung gelenkt werden kann.

1.1 Motivation

Die zunehmende elektronische Vernetzung von physischen Objekten über das Internet, was auch mit dem Begriff Internet der Dinge (engl. internet of things) bezeichnet wird, führt zu einer immer stärkeren Verbreitung von ereignisgesteuerten, verteilten Architekturen. Dies bietet neben dem asynchronen Austausch von Ereignissen die Möglichkeit, Anwendungen lose miteinander zu koppeln, ohne dass sie die spezifischen Schnittstellen des Kommunikationspartners kennen müssen. Ereignisgesteuerte, verteilte Systeme gewinnen im Unternehmensumfeld eine zunehmend große Bedeutung, da es so möglich wird, komplexe und dynamische Geschäftsprozesse auf der Basis von Informationen aus der realen Welt zu steuern oder gar zu entwickeln. Eine Grundvoraussetzung für ein funktionierendes Internet der Dinge bildet eine schnelle Verarbeitung und zeitnahe Übermittlung von Ereignisdaten zwischen den physischen Objekten bzw. Softwaresystemen. [ADi10]

Als Schlüsseltechnologie gelten dabei Publish/Subscribe-Systeme, welche eine Kommunikationsinfrastruktur bereitstellen. Die Infrastruktur soll es Anwendungen ermöglichen, Ereignisdaten (z. B. in einem XML-Format) versenden und empfangen zu können. Die Empfänger formulieren dabei über so genannte Subscriptions, an welchen Arten von Ereignissen sie interessiert sind. Die Ausdrucksmächtigkeit der verwendeten

Subscription-Syntax bestimmt maßgeblich, wie genau solche Interessen formuliert werden können. Inhaltsbasierte Subscriptions stellen dabei den allgemeinsten Ansatz dar, indem sie den Ereigniskonsumenten (Subscribern) ermöglichen, ihre Interessen mittels logischer Ausdrücke über den Inhalt von Ereignissen auszudrücken. Die Flexibilität des inhaltsbasierten Ansatzes impliziert aber auch besondere Anforderungen an das Publish/Subscribe-System, da dieses sehr große Mengen von Ereignissen anhand einer potenziell sehr hohen Anzahl von Subscriptions filtern muss.

1.2 Zielstellung

Im Rahmen dieser Diplomarbeit soll untersucht werden, wie ein XML-Ereignisfilter für ein Publish/Subscribe-System konzeptionell gestaltet und realisiert werden kann. Der Schwerpunkt liegt dabei insbesondere auf der Erreichung einer hohen Laufzeiteffizienz bei gleichzeitiger Skalierbarkeit und universellen Einsetzbarkeit des Systems. Als Ausgangspunkt dienen dabei erste Ergebnisse aus dem Forschungsprojekt ADiWa[1] bzgl. einer XML-basierten Subscription-Sprache. Diese Sprache bildet die Basis für die zu entwickelnden Algorithmen, wobei analysiert werden soll, ob ggf. Anpassungen oder Erweiterungen erforderlich sind. Weiterhin soll untersucht werden, wie etablierte Schnittstellen und Sprachen zur Formulierung von Subscriptions (insbesondere JMS, WS-Notification und XPath) auf diese Sprache abgebildet werden können. Diese so entstehenden Adapter ermöglichen die gleichzeitige Nutzung des XML-Ereignisfilters durch Kommunikationspartner mit verschiedenen, auch proprietären Schnittstellen. Primäres Einsatzgebiet dieses Filters sind die eigenständigen und voneinander unabhängigen Softwarekomponenten einer verteilten Publish/Subscribe-Architektur. Diese so genannten Broker sind neben der Vermittlung von Ereignissen zwischen Produzenten und Konsumenten auch für die Kommunikation mit anderen Brokern verantwortlich. Hier kann untersucht werden, inwiefern es möglich ist, die Adapter bei Bedarf zwischen mehreren Brokern in einem Netzwerk auszutauschen und wie dieses Netzwerk dazu aufgebaut werden soll.

Ausgewählte Teile des zu entwickelnden Konzepts sollen prototypisch implementiert werden, wobei die Effizienz der Lösung anhand von Leistungsmessungen quantitativ nachzuweisen ist. Hierbei kann die eigene Lösung auch mit alternativen Ansätzen bzw. einfachen Möglichkeiten der Filterung (z. B. mit der Java XPath-API) verglichen werden.

Zusammengefasst ergeben sich somit folgende Ziele der Arbeit:

- Systematisierung des Standes von Forschung und Technik in den relevanten Forschungsbereichen, wie z. B. Publish/Subscribe-Systeme und Ereignisfilterung
- Formulierung von Anforderungen sowie Anwendungs- und Testfällen
- Untersuchung der Abbildbarkeit von Anfrageschnittstellen und Sprachen (insbesondere JMS, WS-Notification und XPath) auf die vorliegende Subscription-Sprache

[1] Allianz digitaler Warenfluss, siehe auch http://www.adiwa.net/

- Konzeption eines Systems zur Entgegennahme, Filterung und Verteilung von Ereignissen anhand von inhaltsbasierten Subscriptions
- Abgrenzung des Konzepts von bereits bekannten Lösungen
- Implementierung ausgewählter Teile des Konzepts
- Evaluierung des Konzepts durch Performancemessungen, Anwendungs- und Testfällen sowie dem Vergleich mit alternativen Ansätzen zur Filterung von Ereignisdaten

1.3 Aufbau der Arbeit

Nach einer kurzen Einführung und der Vorstellung der einzelnen Ziele werden im nächsten Kapitel zunächst die Grundlagen anhand von relevanten Technologien und Architekturkonzepten gelegt. In Kapitel 3 erfolgt eine umfassende Analyse der Problemstellung durch Anwendungsfälle sowie der Ableitung von Anforderungen an das zu entwickelnde Konzept. Mit einer Vorstellung verwandter Arbeiten werden in Kapitel 4 auswählte alternative Ansätze und vorhandene Konzepte beleuchtet, um ähnliche Ideen für das eigene Konzept zu nutzen, aber auch, um deren Stärken und Schwächen zu skizzieren. Nach dieser Abgrenzung folgt mit Kapitel 5 eine ausführliche Beschreibung des Konzepts zur Realisierung eines effizienten XML-Ereignisfilters für die beschriebenen Anforderungen. Dabei wird der Ansatz bereits recht implementierungsnah beschrieben, bevor mit der Validierung in Kapitel 6 die Funktionsfähigkeit anhand einer Testumgebung einschließlich Performanceanalyse demonstriert wird. Abschließend werden in Kapitel 7 wichtige Erkenntnisse aus dieser Arbeit diskutiert und ein Ausblick auf weiterführende Arbeiten gegeben.

2 GRUNDLAGEN

In diesem Kapitel werden die erforderlichen Grundlagen zur Konzeption eines Ereignisfilters für XML-basierte Publish/Subscribe-Systeme bereitgestellt. Dabei werden sowohl zentrale Begriffe erläutert als auch die später verwendeten Technologien und Konzepte vorgestellt. Dazu zählen vor allem ereignisgesteuerte und verteilte Systemarchitekturen sowie Komponentenmodelle zur Realisierung des Gesamtsystems. Der Fokus liegt dabei auf einer konsistenten Semantik der in den Folgekapiteln genutzten Begriffe. Neben den Grundlagen und einer Klassifikation von Publish/Subscribe-Systemen werden in diesem Kapitel konkrete Systeme mit ihren jeweiligen Funktionalitäten diskutiert. Eine strikte Abgrenzung zwischen den einzelnen Systemkategorien der vorhandenen Implementierungen ist allerdings nicht immer möglich, da die Übergänge durch verschiedene Ausprägungen zum Teil verschwimmen.

2.1 Ereignisgesteuerte und verteilte Systemarchitekturen

Ereignisgesteuerte, verteilte Systeme stellen eine konkrete Realisierung des EDA-Modells dar und weisen somit besondere Eigenschaften auf, die im Folgenden näher erläutert werden. In einem verteilten System, basierend auf dem Publish/Subscribe-Paradigma, kann ein Interessent bestimmte Arten von Nachrichten abonnieren. Das Zusammenwirken der einzelnen Komponenten in diesem Modell wird in Abbildung 2 veranschaulicht.

Eine spezielle Ausprägung der Nachrichten, die zwischen Sender und Empfänger übermittelt werden, stellen Ereignisse dar. Ereignisse (engl. Events) sind im theoretischen Sinne Beobachtungen einer Zustandsveränderung, die in der Vergangenheit aufgetreten ist [Esp09]. Zu beachten ist, dass sich in diesem Zusammenhang hinter einem Ereignis immer die softwareseitige Abbildung des tatsächlichen Ereignisses verbirgt und nicht die Erfassung selbst [Dav10].

Durch die ereigniserzeugende Anwendung (engl. Publisher) wird eine Nachricht für zahlreiche Abonnenten (engl. Subscriber) im System veröffentlicht. Alle empfangenden Anwendungen, welche dazu berechtigt sind und sich für die entsprechende Nachricht subskribiert haben, erhalten eine unabhängige Kopie der Nachricht. [IBM09]

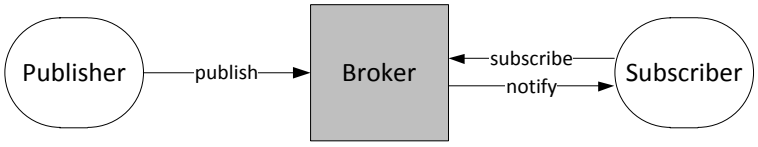


Abbildung 2: Publish/Subscribe-System (Quelle: eigene Darstellung nach [IBM09] und [Int05])

Die Publish/Subscribe-Architektur beschreibt somit ein n:m (viele zu viele) Kommunikationsmodell, indem der Informationsfluss zwischen Sender und Empfänger auf Grundlage der Interessen des Empfängers gesteuert wird. Somit ermöglicht dieses Modell dem Publisher Ereignisse zu generieren, ohne die Empfänger beziehungsweise die Subscriber kennen zu müssen. Die Filterung der Ereignisse und die Übermittlung an

den Empfänger übernimmt ein Vermittler (engl. Broker). Die Weiterleitung von Ereignisdaten an die entsprechenden Empfänger wird auch als Routing bezeichnet. Die Empfänger werden benachrichtigt (engl. notify), sobald ein Ereignis vorhanden ist, das die Interessen des Subscribers trifft. [Yan10] Es wird nicht vorausgesetzt, dass Subscriber und Empfänger denselben Teilnehmer im Netzwerk darstellen. Somit ist es möglich, eine Subscription mit einem anderen Netzwerkteilnehmer am Broker anzumelden.
Verteilte Systeme realisieren gewöhnlich eine 3-Schichten-Architektur. In diesem Fall ist zwischen der Präsentation- und Datenschicht eine Middleware-Schicht zu finden, welche die verschiedenen Informationssysteme integriert. Traditionelle Middleware-Infrastrukturen sind eng aneinander gekoppelt, wie zum Beispiel der entfernte Prozeduraufruf (engl. Remote Procedure Call, RPC). Publish/Subscribe-Systeme haben hier die Aufgabe, diese enge Kopplung zu überwinden.
Zur Kommunikation verwenden eng gekoppelte Systeme statische Punkt-zu-Punkt-Verbindungen zwischen Sendern und Empfängern. Insbesondere muss ein Sender alle seine Empfänger vor dem Senden von Daten kennen. Diese Form der Kommunikation ist nicht ausreichend skalierbar für große, dynamische Systeme, bei denen sich Sender und Empfänger häufig an- und abmelden. Publish/Susbcribe-Systeme bieten eine lose Kopplung von Sendern und Empfängern, indem sie die Daten ohne Kenntnis des Betriebsstatus oder sogar ohne Kenntnis der Existenz des jeweils anderen austauschen. [Yan10]
In ereignisgesteuerten Systemen können Nachrichtenbroker zu einem Brokernetzwerk hinzugefügt werden, um Publisher und Subscriber noch stärker voneinander zu entkoppeln und die Last gleichmäßig auf das Netzwerk zu verteilen. [YiH10]

2.1.1 Publish/Subscribe-Architekturen

Im folgenden Abschnitt wird zur vereinfachten Beschreibung der Kommunikation in den verschiedenen Architektur-Paradigmen davon ausgegangen, dass unter Client eine Softwarekomponente zu verstehen ist, die Informationen benötigt und der Server eine Softwarekomponente darstellt, die Informationen bereitstellen kann.

Klassische serviceorientierte Architekturen (SOA) sind durch einen synchronen Informationsaustausch zwischen Client und Server gekennzeichnet. Die einzelnen Komponenten werden in diesem System als Dienste bezeichnet. Der Client ist dafür verantwortlich, bei der Instanz nachzufragen, welche die Information bereitstellen kann. Dies erfolgt in der Regel durch den Aufruf eines Dienstes auf einem Server. Der Server hat somit keinerlei Verantwortung zur Verbreitung der Informationen, da er diese ausschließlich auf Anforderung bereitstellt.

In ereignisgesteuerten Architekturen (EDA) erfolgt die Informationsverbreitung durch einen asynchronen Ansatz. Der Client, welcher eine Information benötigt, muss am bereitstellenden Server eine Subscription mit den gewünschten Informationen anmelden und diese gegebenenfalls aktualisieren, wenn sich sein Informationsbedarf ändert. Der Server hingegen ist verpflichtet, die verfügbaren Informationen entsprechend der angemeldeten Subscriptions an die Clients kontinuierlich zu verteilen.

Im Wesentlichen können drei Aspekte herausgestellt werden, die ein ereignisgesteuertes System leistungsfähiger machen als ein serviceorientiertes System:

- Die Vertragsdauer zwischen Client und Server ist auf eine dauerhafte Verbindung ausgelegt, statt auf einen einmaligen Aufruf.
- Der Client hat die Verantwortung, seine Subscriptions zu aktualisieren, sodass er zu jeder Zeit nur die Informationen vom Server (Publisher) erhält, die er benötigt.
- Der Server (Publisher) muss bestimmte Dienstgüteanforderungen (Service Level Agreements, SLA) einhalten, um relevante Informationen rechtzeitig dem Client (Subscriber) bereitzustellen.

Bei dem Vergleich von klassischen Publish/Subscribe-Systemen ohne inhaltsbasierte Filterung und ereignisgesteuerten Systemen mit inhaltsbasierter Filterung stellt sich heraus, dass diese nur einen Bruchteil der Möglichkeiten bieten können, die eine inhaltsbasierte Filterung in ereignisgesteuerten Architekturen ermöglicht. Die folgenden Punkte zeigen die wesentlichen Unterschiede dieser beiden Architekturparadigmen auf:

- Umfangreiche Subscriptionsprache
  Themenbasierte Technologien nutzen sehr einfache Subscriptions, um das Interesse an bestimmten Informationen auszudrücken. Eine solche Subscription könnte beispielsweise den Filter „Nachrichten zum Thema XML“ aufweisen, wohingegen ereignisgesteuerte Systeme mit inhaltsbasierten Filtern eine große Vielfalt an Abonnement-Typen bereitstellen, um anspruchsvolle Subscriptions formulieren zu können.
  In ereignisgesteuerten Systemen kann eine Subscription nicht nur ein Filter auf einem einzelnen Ereignis sein, sondern durchaus auch Korrelationen über historische und mehrere Ereignisströme ausdrücken. Weiterhin ermöglichen diese Architekturen eine kontinuierliche Aktualisierung von bereits angemeldeten Subscriptions.
- Erweiterbares Netzwerk
  Traditionelle Push-Architekturen weisen in der Regel eine zentralisierte Netzwerkstruktur auf. Dabei werden Ströme an einen Hub gesendet, der eine einfache Filterung auf den Daten durchführt und diese an die Subscriber weiterleitet. Die Skalierbarkeit ist bei dieser Variante kaum gegeben, da das komplette System von der Leistungsfähigkeit einer zentralen Komponente abhängig ist.
  Bei ereignisgesteuerten Systemen auf der Basis eines Brokernetzwerkes ist ein organisches Wachstum des Netzwerkes möglich. Unter einem Broker wird in diesem Paradigma ein Server verstanden, welcher die Steuerung des Netzwerkes und die Vermittlung von Ereignissen zwischen Publisher, Subscriber und weiteren Brokern regelt. Es können stets neue Broker hinzugefügt werden, welche die Kapazität des Gesamtnetzes erhöhen und die Last stets gleichmäßig auf alle Broker verteilen. Durch die architekturbedingte Vermeidung von Engpässen ist die Skalierbarkeit in dieser Form von ereignisgesteuerten Systemen gegeben.
- Strukturierte Ereignisse
  Ereignisgesteuerte Architekturen ermöglichen nicht nur den Umgang mit unstrukturierten und semistrukturierten Ereignissen. Durch ihre Fähigkeit,

Metadaten während der Filterung zu verarbeiten, können auch strukturierte Ereignisse mit wertvollen Metadaten genutzt werden.

- Allgegenwärtige Standards
  Gerade der Einsatz von Web-Services ermöglicht die Nutzung von vorhandenen Kommunikationsstandards. Der Einsatz von proprietären Protokollen zur Übermittlung von Ereignissen an einen Subscriber kann so vermieden werden. Zwei Vertreter dieser Standards werden im folgenden Kapitel vorgestellt. [Int05]

2.1.2 Publish/Subscribe auf der Basis von Web-Services

Die Entwicklung von Publish/Subscribe-Systemen, basierend auf standardisierten Web-Services-Technologien, gewährleistet eine zuverlässige Kommunikation zwischen den Komponenten. Zwei Varianten für einen standardisierten Ansatz stellen WS-Notification und WS-Eventing dar. Beide Spezifikationen ermöglichen Web-Services-Anwendungen die Nutzung des Publish/Subscribe-Messaging-Musters, egal ob es sich um die Empfangsbereitschaft für Benachrichtigungen über ein bestimmtes Ereignisvorkommen oder das Senden von Ereignisbenachrichtigungen an das System für andere Anwendungen oder Systemverwaltungstools handelt [IBM09]. Der auf offenen Standards basierende Charakter von WS-Notification und WS-Eventing bedeutet, dass Anwendungen unabhängig von den zugrunde liegenden Hardwareplattformen, Softwaresprachen und Systemumgebungen mit anderen Anwendungen kommunizieren können.

Im Folgenden werden die Standardfamilie WS-Notification Version 1.3, die unter der Aufsicht der Organization for the Advancement of Structured Information Standards (OASIS)[2] entwickelt wurde, sowie die beim World Wide Web Consortium (W3C)[3] eingereichte Spezifikation WS-Eventing kurz vorgestellt und die wesentlichen Unterschiede aufgezeigt. Diese Standards definieren im Wesentlichen den Nachrichtenaustausch für Web-Services in Publish/Subscribe-Architekturen. [IBM09]

WS-Notification (aktuelle Version 1.3 seit 2006 als OASIS-Standard[4])

Die Standardfamilie WS-Notification (WSN) in der Version 1.3 besteht aus den drei folgenden Spezifikationen [OAS06]:

- WS-BaseNotification
  definiert die grundlegende Interaktion zwischen Erzeuger und Konsument von Benachrichtigungen (Punkt-zu-Punkt-Messaging)

[2] Die OASIS ist eine internationale Organisation, die sich mit der Weiterentwicklung von E-Business- und Web-Service-Standards beschäftigt; vgl. http://www.oasis-open.org
[3] Das W3C ist ein Gremium zur Standardisierung von Techniken, die das World Wide Web betreffen; vgl. http://www.w3.org
[4] WS-Notification Spezifikation: http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf

- WS-BrokeredNotification
  definiert die Schnittstellen für Notification Broker (indirekte Publish/Subscribe-Kommunikation)
- WS-Topics
  definiert hierarchische Themenbäume (engl. topic trees), damit jeder Nachricht ein Thema (engl. topic) zur Kennzeichnung des Themenbereiches zugeordnet werden kann (Topic-based Publish/Subscribe-Kommunikation)

WS-Eventing (aktuelle Version seit 2006 als W3C-Submission[5])

WS-Eventing (WSE) spezifiziert die Schnittstellen für die direkte asynchrone Kommunikation zwischen Webservices (Publisher und Subscriber). Dabei weist WS-Eventing große Ähnlichkeiten in Bezug auf Architektur und Funktionen zu WS-BaseNotification auf.

Dieser Standard beschreibt das einfachste Level an Web-Services-Schnittstellen für Ereignisproduzenten und Ereigniskonsumenten inklusive der Standard-Nachrichtenaustauschmuster, welche von den Dienstanbietern bereitgestellt werden müssen, um in diesen Rollen agieren zu können. [IBM04]

Architektur von WS-BaseNotification und WS-Eventing [YiH10]

Wie aus Abbildung 3 und Abbildung 4 hervorgeht, definieren beide Spezifikationen die Komponenten Subscriber und Subscription Manager. Die Komponente Event Sink bei WSE ist vergleichbar mit dem Notification Consumer bei WSN. Da die Subscriber von den Notification Consumern getrennt sind, sind diese nur für das Empfangen von Nachrichten verantwortlich. Die Komponenten Publisher und Notification Producer werden in WSE unter Event Source zusammengefasst.

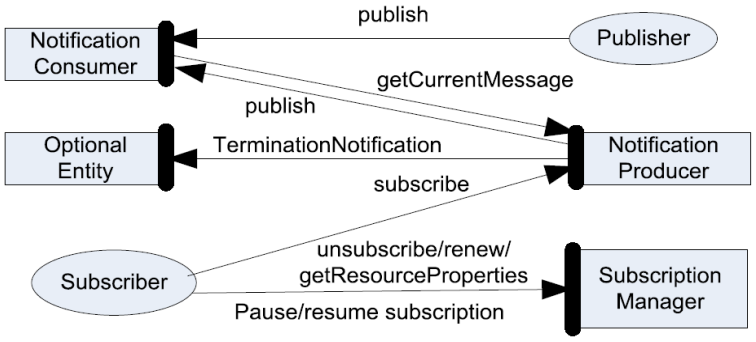


Abbildung 3: WS-BaseNotification (Quelle: [YiH10])

[5] http://www.w3.org/Submission/WS-Eventing/

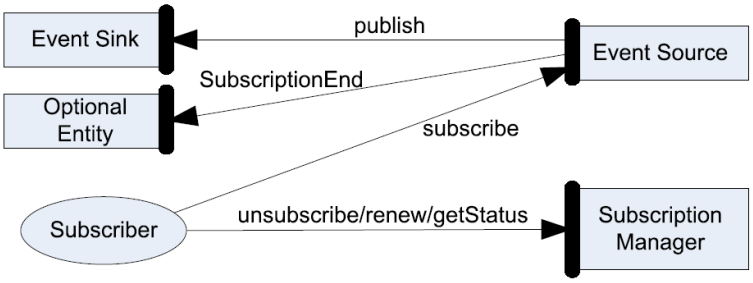


Abbildung 4: WS-Eventing (Quelle: [YiH10])

Funktionen und Schnittstellen

Tabelle 1 zeigt vergleichend die Operationen von WS-BaseNotification und WS-Eventing, welche durch die beteiligten Komponenten bereitgestellt werden müssen:

WS-BaseNotification	WS-Eventing	Beschreibung
Subscribe	Subscribe	Erzeuge Subscription für eine Event Sink
Renew	Renew	Nachrichten zur Verwaltung von Subscriptions zwischen Subscriber und Subscription Manager
Unsubscribe	Unsubscribe	
indirekt über WSRF (getResourceProperties)	GetStatus	
indirekt über WSRF (TerminationNotification)	SubscriptionEnd	Wird bei unerwarteter Beendigung einer Subscription an die im Subscription Request festgelegte Adresse gesendet
Pause/Resume Subscription	nicht möglich	Anhalten und Fortsetzen einer Subscription
GetCurrentMessage	nicht möglich	Liefert die aktuelle Nachricht an den Notification Consumer

Tabelle 1: Operationen von WS-BaseNotification und WS-Eventing (Quelle: eigene Darstellung nach [YiH10])

Bedingungen in Subscriptions

- WS-Notification definiert drei verschiedene Typen von Filtern in Subscription Requests: TopicExpression, ProducerProperties und MessageContent. Das folgende Listing zeigt als Beispiel einen Auszug aus einer Subscribe-Message:

```
…
<wsnt:Filter>
  <wsnt:TopicExpression Dialect="xsd:anyURI">
         abc:ExampleTopic
  </wsnt:TopicExpression>
  <wsnt:ProducerProperties Dialect="xsd:anyURI">
         boolean(/*/MgtPubInterval > 100)
  </wsnt:ProducerProperties>
  <wsnt:MessageContent Dialect="xsd:anyURI">
         /bankTransfer[value > 1000000]
  </wsnt:MessageContent>
</wsnt:Filter>
…
```

Listing 1: Beispiel eines Filters in einer Subscription in WS-Notification

- WS-Eventing erlaubt maximal einen Filter pro Subscription Request. Standardmäßig ist dies ein inhaltsbasierter Filter unter Verwendung von XPath-Ausdrücken. Das folgende Listing zeigt als Beispiel einen Auszug aus einer Subscribe-Message:

```
…
<wse:Filter Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116">
        weather.storms
</wse:Filter>
…
```

Listing 2: Beispiel eines Filters in einer Subscription in WS-Eventing

2.2 Klassifikation von Publish/Subscribe-Architekturen

Nach [Yan10], [Mic10] und [Pie07] gibt es zahlreiche Möglichkeiten, Publish/Subscribe-Systeme zu klassifizieren. Rein funktional lassen sich diese Systeme nach Übertragungsformat, Subscription- / Ereignismodell oder Programmierschnittstelle (API) unterscheiden, was von Pietzuch et al. in [Pie07] vorgenommen wird. Ein anderer, aber naheliegender Ansatz ist die Einteilung der Publish/Subscribe-Systeme nach der Art, wie eine Routingentscheidung getroffen wird. Somit ergeben sich die in Abbildung 5 dargestellten drei Hauptkategorien themenbasierter (engl. topic-based), typbasierter (engl. type-based) und inhaltsbasierter (engl. content-based) Systeme.

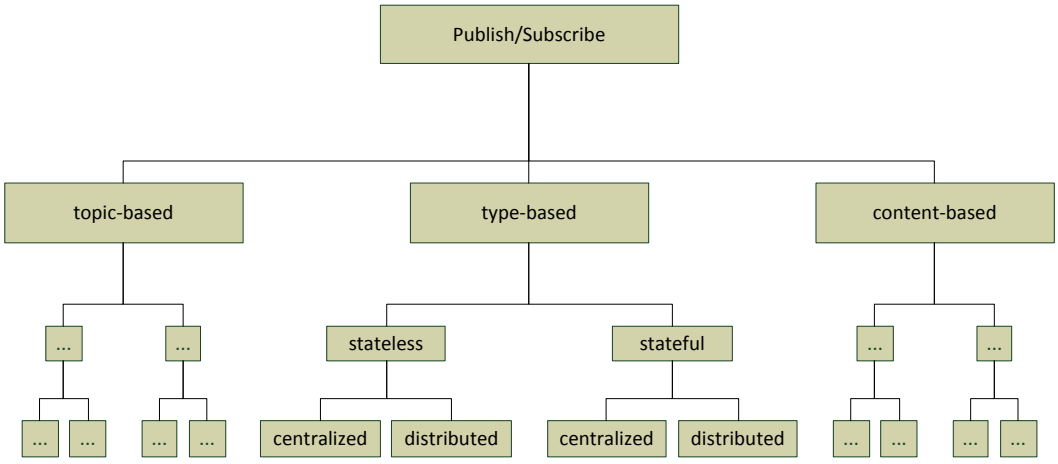


Abbildung 5: Klassifikation von Publish/Subscribe-Systemen (Quelle: eigene Darstellung nach [Yan10])

Eine weitere Unterteilung von Publish/Subscribe-Systemen in einer tieferen Ebene ist die Unterscheidung nach der Art der Verarbeitung von Subscriptions. Werden die Subscriptions auf jede Nachricht einzeln angewendet, zum Beispiel bei der Filterung von Nachrichten/Ereignissen, ohne jegliche Interaktion über die Nachrichtengrenzen hinweg, so spricht man von statuslosen Systemen oder einer statuslosen Verarbeitung (engl. stateless processing). Den Gegensatz dazu bildet die Gruppe der statusbehafteten Systeme (engl. stateful processing), welche die Fähigkeit der Verarbeitung von Datenströmen voraussetzen. Dabei wird über einen langen Strom von Informationen ein Status aufrechterhalten.

Abschließend lassen sich Publish/Subscribe-Systeme zusätzlich nach dem Grad der Verteilung der einzelnen Architekturkomponenten einteilen, was ebenfalls in Abbildung 5 skizziert wird. Somit wird zwischen zentralem und dezentralem Systemdesign unterschieden. Bei einer zentralen Architektur existiert ein Broker, welcher die alleinige Aufgabe der Vermittlung übernimmt. Bei einem dezentralen Design wird die Last auf viele Broker verteilt, wodurch auch gut skalierende Systeme möglich sind. Bei dieser Variante steigt allerdings die Anforderung an eine effiziente Rountingfunktionalität zur Kommunikation zwischen den Brokern. [Yan10]

Eine wichtige Eigenschaft von Publish/Subscribe-Systemen ist die Fähigkeit, Nachrichtenströme oder Ereignisströme zu verarbeiten (engl. event stream processing). Dabei werden die Subscriptions in einem Broker als kontinuierliche Anfragen gespeichert, welche bei jedem Eintreffen von neuen Nachrichten oder Ereignissen ausgewertet werden. Durch die Möglichkeit der Verarbeitung von Datenströmen ergeben sich zusätzliche Herausforderungen für ein Publish/Susbscribe-System, insbesondere wenn die Struktur der Ereignisse nicht mehr offensichtlich ist, sondern durch die Subscriptions bedingt wird [Yan10]. Die zentralen Anforderungen an ein leistungsfähiges Publish/Subscribe-System können somit wie folgt lauten:

- **Skalierbarkeit:**

Eine der Schlüsselanforderungen an ein Publish/Subscribe-System ist die Skalierbarkeit. Insbesondere auf der Seite der Subscriber müssen sowohl Hunderte als auch Millionen von Subscriptions bedient werden können. Angesichts dieser Zahlen ist die effiziente Filterung ein herausragendes Thema um die Nachrichten zu finden, welche den Subscriptions zugeordnet werden können.

- **Stabilität:**

Eine zweite Voraussetzung des Brokers ist die Fähigkeit, in hochdynamischen Umgebungen zu arbeiten, in denen ständig Teilnehmer dem System beitreten oder es verlassen und die ihre Interessen an den Informationen im Laufe der Zeit ändern. Somit müssen die Broker in der Lage sein, schnell auf die ständig wechselnden Abfragen zu reagieren, ohne die Verarbeitung eingehender Nachrichten zu beeinträchtigen.

- **Verteilung:**

Aufgrund der hohen Menge von Nachrichten/Ereignissen und Subscriptions, ist es für große Publish/Subscribe-Systeme erforderlich, die Last der Filterung und Auslieferung von Ereignissen auf viele Broker zu verteilen. In diesem Fall ergibt sich eine weitere Herausforderung des effizienten Routings von Nachrichten von der Seite des Publishers zu der Menge an Brokern, welche die relevanten Subscriptions besitzen, um eine vollständige Verarbeitung der Nachrichten zu gewährleisten.

2.2.1 Themenbasierte Publish/Subscribe-Systeme

Wie in Abbildung 6 dargestellt ist, veröffentlicht bei themenbasierten Publish/Subscribe-Systemen (engl. subject-based oder topic-based) ein Publisher jede Nachricht unter

einem bestimmten Thema, was zum Beispiel durch ein Label gekennzeichnet ist. Diese Themen stammen aus einer vordefinierten Menge (z.B.: „Börsenkurse“) oder einer Hierarchie (z.B.: „Sport/Golf“) mit Themen. Die Empfänger können somit eine Subscription für ein gewünschtes Thema anmelden und erhalten anschließend die entsprechenden Nachrichten übermittelt. Bei hybriden Verfahren kann die Subscription durch inhaltsbasierte Filter weiter verfeinert werden. [Yan10]

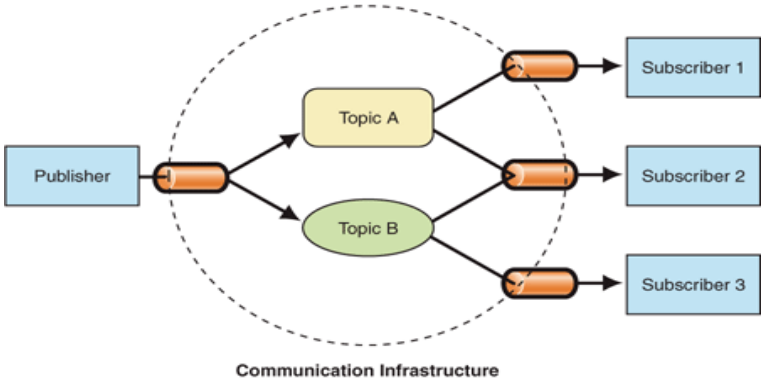


Abbildung 6: Themenbasiertes Publish/Subscribe-System (Quelle: [Mic10])

Als Spezialfall der themenbasierten Systeme können die broadcastbasierten Publish/Subscribe-Systeme angesehen werden. Der konkrete Zusammenhang wird deutlich, wenn man davon ausgeht, dass in einem themenbasierten System nur ein Thema existiert, für das sich alle Subscriber anmelden und die Publisher auch nur unter diesem einen Thema veröffentlichen können. So ist beispielsweise das Local Area Network (LAN) im weitesten Sinne ein Vertreter dieser Kategorie, da ein Teilnehmer eine Nachricht an eine Broadcast-Adresse senden kann, welche anschließend allen anderen Teilnehmern zur Verfügung steht. Jeder Empfänger eines Broadcasts muss die Nachricht entgegennehmen und entscheiden, ob er die Nachricht verarbeiten soll. Falls der Empfänger sich als nicht zuständig erkennt, verwirft er die Nachricht stillschweigend. Will man nun die Verbindung zu den themenbasierten Publish/Subscribe-Systemen herstellen, so kann davon ausgegangen werden, dass lediglich ein globales Thema existiert, für das sich alle Teilnehmer angemeldet haben. Wird nun eine Nachricht mit dem Thema „Broadcast“ gesendet, so erhalten alle angemeldeten Teilnehmer diese zugestellt. Die Aufgabe der Nachrichtenauswahl wird somit vom Broker auf den Empfänger verlagert.
Diese sehr einfache Variante stellt eine effektive Methode zur Entkoppelung von Produzenten und Konsumenten dar. Durch die Tatsache, dass alle Nachrichten an alle Knoten im Netzwerk gesendet werden und jeder selbst dafür verantwortlich ist, ungewünschte Nachrichten herauszufiltern, spricht man hier auch von reaktiver Filterung. [Mic10]

Zu den bekanntesten Vertretern der themenbasierten Systeme zählen die Forschungsprojekte Scribe [Cas02] als Peer-to-Peer Variante und Gryphon [Ast04], aber auch XMPP [Mil10] mit einer speziellen Erweiterung für Publish/Subscribe.

Aufgrund des offenen Charakters stellt das „Extensible Messaging and Presence Protocol“ (kurz XMPP) mit der Publish/Subscribe-Erweiterung XEP-0060 [Mil10] einen passenden Vertreter zur Beschreibung des Broadcast-Verfahrens dar. Obwohl dieses System als themenbasiert konzipiert ist, kann mit folgendem Beispiel eine broadcastbasierte Anwendung demonstriert werden. Die Grundidee dabei ist relativ simpel. Einzelne Teilnehmer können am System bestimmte Themen erstellen, in diesem Fall das Thema „Broadcast“.

(1) Alle Subscriber melden sich am System für das einzige Thema „Broadcast“ an.
(2) Ein Publisher veröffentlicht nun eine Nachricht unter dem Thema „Broadcast“ und sendet diese an das Publish/Subscribe-System.
(3) Das Publish/Subscribe-System übermittelt nun die Nachricht an alle autorisierten Teilnehmer.

2.2.2 Typbasierte Publish/Subscribe-Systeme

Ein sehr ähnlicher Ansatz wird mit den typbasierten Publish/Subscribe-Systemen (engl. type-based) verfolgt. Hier erfolgt die Filterung allerdings nicht auf der Basis von Themen, sondern wird durch den Typ eines Ereignisses festgelegt [Eug03]. Durch diese Entscheidung auf struktureller Ebene wird eine bessere Integration von gängigen Programmiersprachen und Middleware erreicht. Voraussetzung ist allerdings die Spezifikation möglicher Ereignistypen, um die getypten Ereignisdaten erkennen und filtern zu können. Die Ähnlichkeit zum vorangegangenen Ansatz wird ebenfalls deutlich, wenn man bedenkt, dass sich beide Kategorien ineinander überführen lassen. Dazu müssen lediglich die Themen eines Ereignisses in Ereignistypen transformiert werden.

Als eine mögliche Umsetzung typbasierter Publish/Subscribe-Systeme implementiert das Projekt Hermes [Pie02] ein Netzwerk von Ereignis-Brokern unter Nutzung eines Overlay-Netzwerks. Der Fokus liegt bei diesem Projekt somit auf der Verteilung. Das Ereignismodell ist XML-basiert, wobei die Ereignistypen stets über ein XML Schemadokument definiert werden müssen. Das Subscriptionmodell ist typ-/attributbasiert, wobei die Angabe von XPath-Ausdrücken möglich ist. Neben diesen klassischen Funktionalitäten bietet Hermes einige weitere Middleware-Dienste wie Sicherheit, erweiterte Ereigniserkennung oder Stauerkennung an. Wichtiger Bestandteil dieses Systems ist auch die XML-Schnittstelle zur automatischen Übersetzung zwischen XML und Java-Objekten.

2.2.3 Inhaltsbasierte Publish/Subscribe-Systeme

Bei den inhaltsbasierten Publish/Subscribe-Systemen (engl. content-based) wird häufig zwischen Prädikat-Filtern und XML-Filtern unterschieden, mit denen die Ergebnismenge eingeschränkt wird. Eine Subscription besteht somit aus einer Menge an Filtern.
Bei prädikatbasierten Filtern wird der Nachrichteninhalt als eine Menge an Attribut-Wert-Paaren modelliert. Dies ermöglicht dem Interessenten die gewünschte

Ergebnismenge feingranular zu beschreiben, indem Subscriptions über mehrere solcher Paare unter Verwendung von Operatoren wie „and“ und „or“ formuliert werden können. Zum Beispiel könnte eine prädikatbasierte Anfrage nach bestimmten Aktienkursen folgendermaßen aussehen: „Type=‘stock quote‘ and Symbol=’ABC’ and (Change > 1 or Volume > 50.000)”.
Mit XML-basierten Filtern lässt sich die große Ausdrucksmächtigkeit von XML-kodierten Nachrichten, vor allem die hierarchische und flexible Struktur von XML, ausnutzen. Die Nutzer können ihre Subscription beispielsweise in Form einer existierenden XML-Anfragesprache, wie XQuery, formulieren. Diese Strukturen ermöglichen somit ein sehr akkurates Filtern und weitere Verarbeitungsschritte für ein benutzerdefiniertes Ausliefern der Ergebnisse. [Yan10]

Eine der ersten Implementierungen von verteilten und inhaltsbasierten Publish/Subscribe-Systemen ist das SIENA [Car011] Projekt. Siena basiert auf einer Broker Netzwerkarchitektur, welche über einer TCP/UDP-Schicht aufgebaut ist. Das Routing erfolgt über spezielle filterbasierte Algorithmen auf einer, mit Forwardingtabellen aufgebauten, Datenstruktur. Detailliert wird diese in den Verwandten Arbeiten in Kapitel 4.1.3 beschrieben. Zur Optimierung wird hier auch das Konzept der Advertisements eingeführt, um ein Verteilen der Subscriptions auf alle Broker im Netzwerk zu vermeiden. Für Clientanwendungen stellt das Siena-System zwei APIs zur Verfügung, Java und C++. Ereignisse sind in Siena nicht XML-basiert, sondern als strukturierte Verbünde von Attribut/Wert-Paaren aufgebaut. Subscriptions hingegen sind Konjunktionen von Attributfiltern.

Mit einem komplett verschiedenen Subscriptionmodell ermöglicht das inhaltsbasierte Publish/Subscribe-System XMessages [Slo02] die Filterung von XML-basierten Ereignissen. Die Subscriptions können in Form von SQL-Anweisungen am System registriert werden. Das Designziel von XMessages ist ein Grid-basiertes Publish/Subscribe-System, welches mit Web-Services kompatibel ist. Dadurch wird der Einsatz des Systems für andere Anwendungen möglich, die sich nicht in der lokalen Umgebung befinden, sondern den Nachrichtendienst über das Internet nutzen möchten.

Ein weiteres inhaltsbasiertes System realisiert das Projekt REBECA [Müh02], welches sich auf Mechanismen konzentriert, um die Skalierbarkeit von Routing-Algorithmen zu erhöhen und gleichzeitig kein globales Wissen über das Netzwerk voraussetzt. Die Algorithmen ermöglichen es, bestimmte Ähnlichkeiten zwischen Subscriptions bzw. Überdeckungen (engl. covering) zu erkennen und diese miteinander zu verschmelzen (engl. merging).

2.3 Relevante Technologien

An dieser Stelle werden nun einige weitere Technologien vorgestellt, die für die Konzeption des XML-basierten Ereignisfilters benötigt werden. Da diese Technologien nicht zum Kern der Arbeit gehören, wird nach einer recht kurzen Beschreibung auf die entsprechende Literatur für zusätzliche Informationen verwiesen. Aufgrund der

geforderten Implementierung in der Sprache Java handelt es sich bei den vorgestellten Technologien ausschließlich um Java-spezifische Konzepte, wobei diese durchaus in ähnlicher Form auch bei anderen Programmiersprachen, wie C# zu finden sind.

2.3.1 OSGi-Komponentenmodell

Die OSGi Service Platform spezifiziert ein hardwareunabhängiges und dynamisches Modulsystem für Java. Module können in der OSGi Service Platform in Form von Softwarekomponenten (engl. Bundles) zur Laufzeit installiert, gestartet, gestoppt, aktualisiert und deinstalliert werden. Abhängigkeiten zwischen Bundles werden dabei auf Package-Ebene explizit und feingranular beschrieben. Aufbauend auf dem Modulsystem können innerhalb einer Java Virtual Machine Dienste (engl. Services) publiziert und anderen Bundles zur Verfügung gestellt werden. Die dynamische Integration und das Fernmanagement können zur Laufzeit erfolgen, ohne dass die Plattform als Ganzes angehalten oder neugestartet werden muss. [Wüt08]

Basiskomponente der Serviceplattform ist das OSGi Framework, das einen Container für Bundles und Services implementiert, was in Abbildung 7 dargestellt ist.

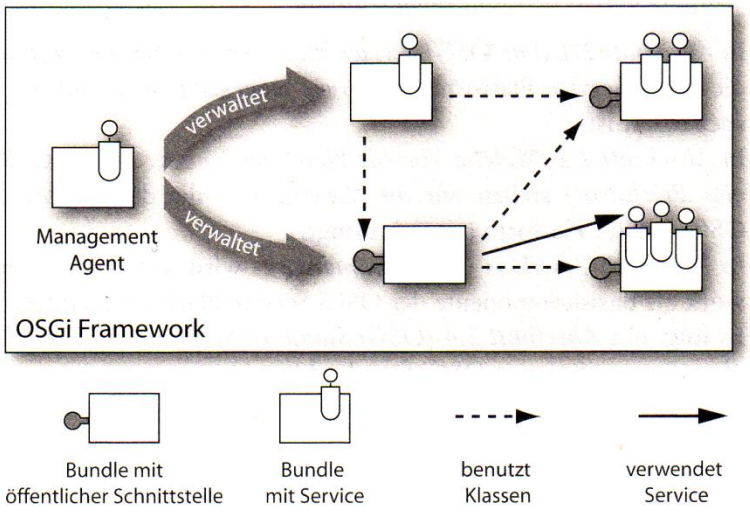


Abbildung 7: Das OSGi Framework (Quelle: [Wüt08])

Ein Bundle ist eine fachlich oder technisch zusammenhängende Einheit von Klassen und Ressourcen, das eigenständig im Framework installiert und deinstalliert werden kann. Um diese Klassen für andere Bundles sichtbar zu machen, muss ein Fragment die entsprechenden Klassen explizit exportieren.

Ein weiteres Mittel zur Entkopplung zwischen unterschiedlichen Modulen stellen die Services dar, die systemweit zur Verfügung gestellt werden. Die Registrierung erfolgt über die Service Registry, die zentral und bundleübergreifend bereitsteht und an der die angemeldeten Dienste von beliebigen Bundles abgefragt werden können.

Als Fragment Bundles (kurz Fragments) werden spezielle Bundles bezeichnet, die beispielsweise eine Implementierung von verschiedenen Anwendungsvarianten

ermöglichen. Ein Fragment unterscheidet sich von einem normalen Bundle dadurch, dass es keinen eigenen Lebenszyklus besitzt, sondern von einem anderen Bundle, dem Host Bundle, hinzugefügt wird. Das Host Bundle hat dabei keinerlei Kenntnis über die zugeordneten Fragments, wobei die Fragments über ihr Manifest einem konkreten Host Bundle zugeordnet sind.

Für eine detaillierte Beschreibung der Eigenschaften der OSGi Service Platform sei an dieser Stelle auf die Literatur von Wütherich et al. „Die OSGi Service Platform“ verwiesen [Wüt08].

2.3.2 XML-Binding mit JAXB

Wenn XML-Dokumente auf einer höheren Abstraktionsebene verarbeitet werden sollen, bietet das Konzept des XML-Binding eine einfache Möglichkeit aus XML-Dokumenten Java-Objekte zu erzeugen.

Mit diesem Konzept wird von der dokumentnahen Verarbeitung von XML-Daten abstrahiert, wobei das Modell stets die semantische Bedeutung der XML-Elemente widerspiegelt. Im Vergleich zum Parsen mit dem Document Object Model (DOM) enthält das XML-Binding bereits einen weiteren Verarbeitungsschritt, die Interpretation der Modellklassen gemäß der eigenen Anwendungslogik.

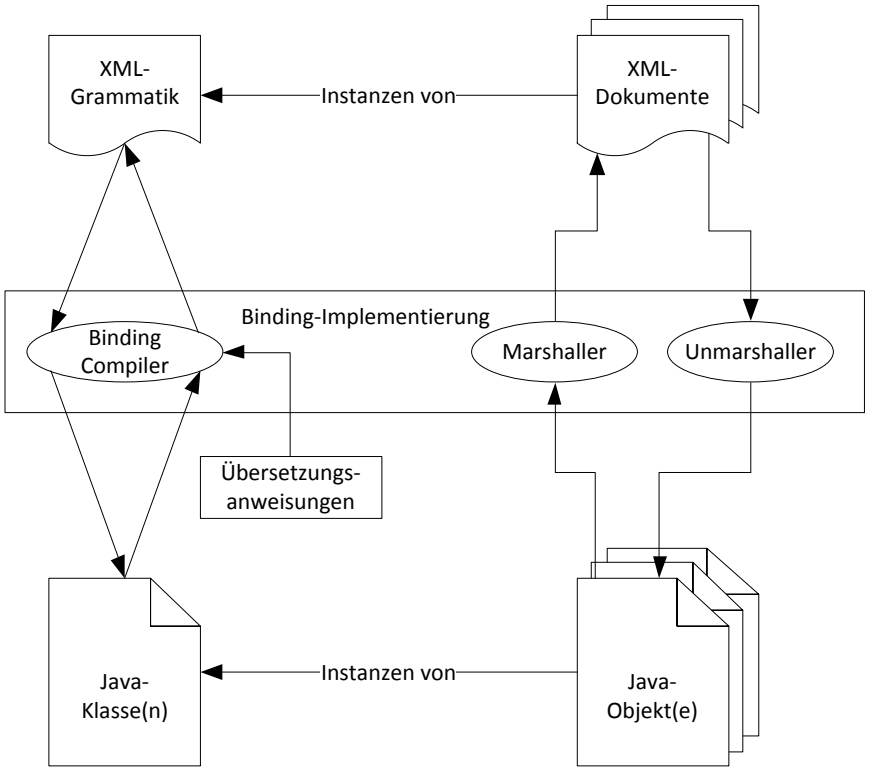


Abbildung 8: Prinzip des XML-Bindings (Quelle: [Sch09] S.98)

Weitere Features ergeben sich direkt aus dem in Abbildung 8 dargestellten Prinzip. Zu beachten sind dabei auch die speziellen Bezeichnungen der drei Hauptkomponenten

Binding Compiler, Marshaller und Unmarshaller. Wie bereits vorgestellt, wird die Generierung von Modellklassen als Binding-Vorgang bezeichnet. Dabei gibt es durch Übersetzungsanweisungen ein Standardverhalten, das durch so genannte Binding Declarations erweitert oder modifiziert werden kann. Das Serialisieren von Java-Objekten bezeichnet man als Marshalling, das Parsen von XML-Dokumenten demzufolge als Unmarshalling. Erkennbar wird durch diese drei Komponenten die Trennung in eine Java- und eine XML-Ebene, wobei jeweils eine Vorschrift (XML-Grammatik, Java-Klassen) und eine beliebige Anzahl konkreter Instanzen dieser Vorschriften (XML-Dokumente, Java-Objekte) existieren.

Die Java API for XML Binding (kurz JAXB) stellt als Referenzimplementierung eine der möglichen Umsetzungen in der Java-Welt für das XML-Binding dar. Da JAXB in der Version 2.0 bereits fester Bestandteil des Java SDK 6.0 ist, bietet sich die Verwendung dieses gängigen Standards für das Erfassen von XML-basierten Subscriptions an. Für eine detaillierte Beschreibung der Funktionsweise von JAXB sei an dieser Stelle auf die Literatur von Scholz, Niedermeier „Java und XML“, S. 391 ff verwiesen [Sch09].

2.3.3 Strombasierte Verarbeitung von XML mit StAX

Für das Parsen und Serialisieren von XML-Dokumenten in Form von Datenströmen steht mit der Streaming API for XML (StAX) eine Java-Technologie zur Verfügung, welche die Ansätze von DOM und SAX vereinen soll.

Beim StAX-Parsing wird gemäß dem Pull-Prinzip die Steuerung des Parsers durch die Anwendung übernommen. Ähnlich wie beim Lesen aus einem Datenstrom werden die Informationen hier aus dem XML-Dokument abgerufen. Dies erfolgt allerdings nicht byte- oder char-weise, sondern auf der Basis von typischen XML-Bausteinen, wie Elementen, Attributen und Text.

Der nächste klassische Ansatz, das SAX-Parsing, erweist sich aufgrund der Nachteile aus dem Push-Prinzip auf den zweiten Blick ebenfalls als weniger geeignet. Hierbei wird der Programmfluss durch den Parser über das Auslösen von Ereignissen gesteuert. Jeder Aufruf eines so genannten Callback-Handler enthält lediglich die aktuellen Knoteninformationen ohne zusätzlichen Kontext über die Struktur des XML-Dokumentes. [Sch09]

Ein weiterer wichtiger Vorteil ist die Möglichkeit der Serialisierung von Objekten in ein XML-Dokument, was in Verbindung mit dem Versand von gefilterten Ereignisdaten benötigt wird.

Zusammenfassend werden die wesentlichen Unterschiede zwischen den beiden Varianten SAX und StAX in der nachfolgenden Abbildung 9 verdeutlicht.

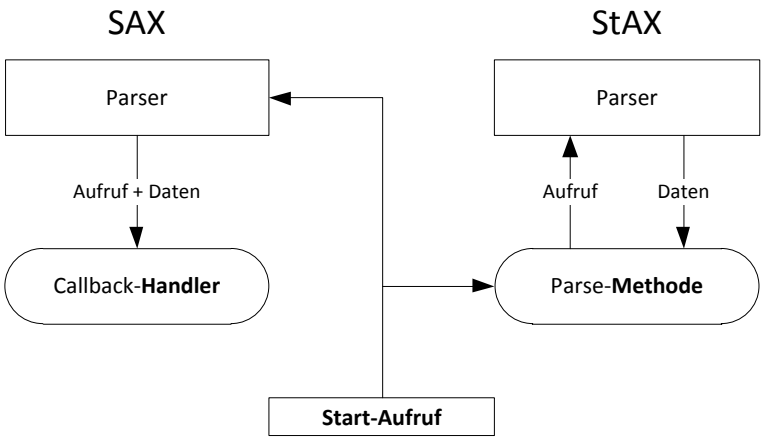


Abbildung 9: Unterschiede zwischen SAX und StAX (Quelle: [Sch09], S.82)

Bei der Verwendung von StAX stehen zwei konzeptionell verschiedene Programmierschnittstellen zur Verfügung, die Cursor-API und die Event-Iterator-API. Mit Hilfe der Cursor-API als Low Level Schnittstelle wird das XML-Dokument in die elementaren Bestandteile aufgeteilt und ein Cursor von einem Baustein zum nächsten bewegt. Da diese Variante etwas performanter als die auf einem höheren Level angesiedelte Event-Iterator-API arbeitet, kommt die Cursor-API für das Einlesen der Ereignisdaten zum Einsatz. Für eine detaillierte Beschreibung der Funktionsweisen und Unterschiede beider Programmierschnittstellen sei an dieser Stelle auf die Literatur von Scholz, Niedermeier „Java und XML“, S. 307 ff verwiesen [Sch09].

3 ANALYSE

In diesem Kapitel sollen zunächst anhand von zwei Anwendungsszenarien aus verschiedenen Einsatzbereichen die Herausforderungen aufgezeigt werden, welche sich aus dem Einsatz von verteilten, ereignisgesteuerten Architekturen ergeben. Dadurch wird es möglich, Anforderungen an ein Publish/Subscribe-System abzuleiten, um schließlich Lösungen zur Konzeption eines skalierbaren und effizienten Ereignisfilters im folgenden Kapitel zu finden.

3.1 Problemanalyse

Viele Anwendungsgebiete von ereignisgesteuerten Systemen sind durch ein hohes Ereignisaufkommen und zahlreiche beteiligte Instanzen gekennzeichnet. Diese Instanzen können stark verteilt vorliegen und weisen oft spezifische Schnittstellen zur Kommunikation auf. Publish/Subscribe-Systeme haben hier die Aufgabe, eine Kommunikationsinfrastruktur bereitzustellen, um den asynchronen Austausch von Ereignisdaten zwischen den verschiedenen Anwendungen gewährleisten zu können. Genau an diesem Punkt werden zwei zentrale Probleme für den Broker deutlich. Zum einen muss dieser eine große Anzahl von Ereignissen vieler Produzenten effizient verarbeiten und dabei den richtigen Empfängern zuordnen und zum anderen muss der Broker mit den verschiedenen Kommunikationstechniken und -schnittstellen der beteiligten Systeme zurechtkommen.

Um diese Herausforderungen zu verdeutlichen, wird im folgenden Abschnitt ein Anwendungsszenario aus dem Bereich der Produktionslogistik vorgestellt. Dieses Szenario steht exemplarisch für ein Anwendungsgebiet mit sehr hohem Ereignisaufkommen und einer großen Zahl von Interessenten an diesen Ereignisdaten. Der verteilte Ansatz gewährleistet neben einer hohen Laufzeiteffizienz gleichzeitig eine hohe Skalierbarkeit des Systems, was mit einem zweiten beispielhaften Szenario in Abschnitt 3.1.2 gezeigt wird. Dieses Szenario stammt aus dem Bereich der Gebäudeautomatisierung und demonstriert somit die Möglichkeit der Übertragung der zugrundeliegenden Idee auf ein weiteres Anwendungsfeld, welches eine Netzwerkstruktur mit wenigen Brokern und moderatem Ereignisaufkommen darstellt. Um die Skalierbarkeit zu verdeutlichen, wird gezeigt, wie das System mit verhältnismäßig geringem Aufwand an wachsende Anforderungen angepasst werden kann.

3.1.1 Anwendungsszenario Automobilindustrie

Ein international tätiger Automobilkonzern mit 60 eigenen Fertigungsstätten produziert in 21 Ländern auf 5 Kontinenten 6,3 Mio. Fahrzeuge im Jahr (ca. 415 Fahrzeuge pro Arbeitstag und Fertigungsstätte). Weltweit arbeitet der Konzern mit nahezu 36.000 Zulieferern aus den verschiedenen Ebenen des Zulieferernetzwerks zusammen. Über 60

Prozent der durchschnittlich 8.500 Einzelteile pro Fahrzeug stammen mittlerweile von Zulieferern. Die restlichen Teile werden durch den Hersteller selbst produziert. Verkauft werden die Fahrzeuge der 9 verschiedenen Konzernmarken über Groß- und Einzelhändler in 153 Ländern. [6]

Brokernetzwerk

Der Warenfluss zwischen den Beteiligten innerhalb des Netzes aus Zulieferer, Hersteller, Großhändler und Einzelhändler wird durch Logistikunternehmen unterstützt. Zu dem Waren- und Materialfluss kommt der Informationsfluss durch den Einsatz von RFID-Technologie entlang der gesamten Wertschöpfungskette. Dabei werden sowohl die Einzelteile als auch die produzierten Einheiten an bestimmten Orten bzw. zu bestimmten Zeiten erfasst. Diese Erfassung der Informationen von ereignisproduzierenden Einheiten wie Sensoren, aber auch betriebswirtschaftlichen Systemen erfolgt dezentral in den einzelnen Fertigungsstätten, wobei der Austausch von Informationen in diesem Netzwerk asynchron durch ein ereignisgesteuertes System geregelt ist.

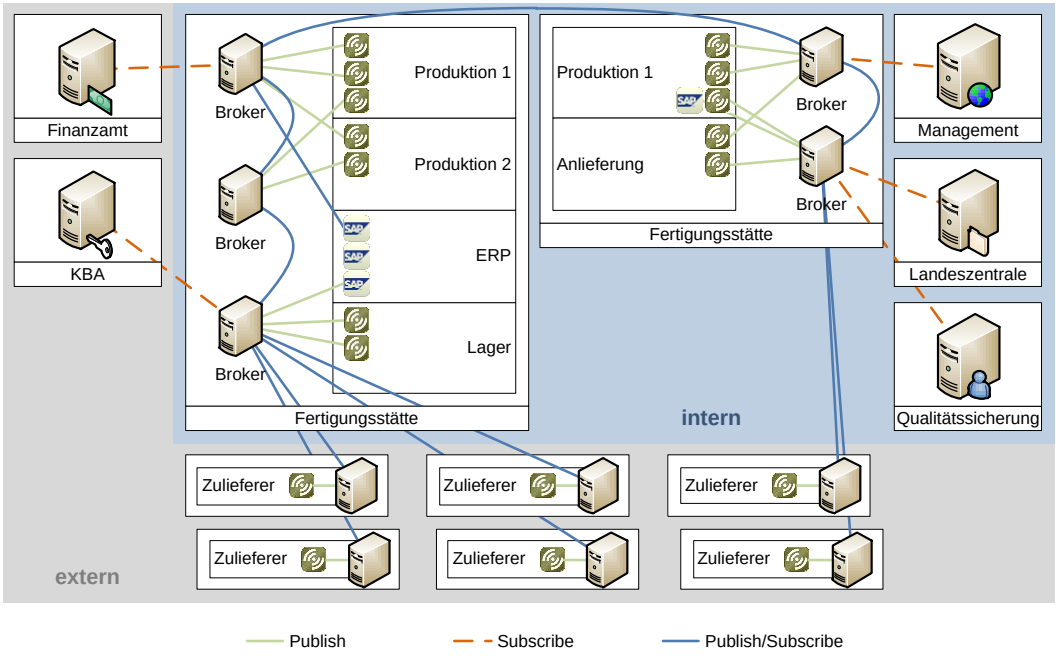


Abbildung 10: Informationsfluss im Broker-Netzwerk (Quelle: eigene Darstellung)

Wie in Abbildung 10 auszugsweise dargestellt ist, sieht die Netzwerktopologie des Herstellers folgendermaßen aus: Der Hersteller produziert auf mehreren Kontinenten und in mehreren Ländern, wo er mehrere Fabriken und jeweils eine Zentrale hat. Die Fabriken haben wiederum mehrere Produktionshallen, Lager oder weitere Bauwerke. In jedem dieser Gebäude befindet sich schließlich eine Menge von RFID-Lesegeräten sowie anderen ereignisproduzierenden Einheiten (zum Beispiel ERP-Systeme). Einige der Ereignisproduzenten teilen sich jeweils einen Industrie-PC, der als Edge Broker fungiert

[6] Die Zahlen aus diesem Szenario basieren überwiegend auf Informationen der Volkswagen AG (Konzernwebsite [Vol102], B2B-Platform [Vol101] sowie Geschäftsbericht 2009 [Vol09]), ergänzt um weitere Quellen [sue10].

und die Daten entgegennimmt. Dieser ist dann zyklenfrei mit den anderen Brokern im gleichen Bauwerk verbunden, wobei einer oder mehrere wiederum mit einem oder mehreren Brokern in den benachbarten Bauwerken verbunden sind. Diese Kette lässt sich schließlich beliebig zu einem vermaschten Netz fortführen.

Innerhalb und außerhalb des Herstellers gibt es potentiell viele Interessenten für die Ereignisdaten, wobei hier nur ein paar ausgewählte exemplarisch vorgestellt werden: Die jeweilige Landeszentrale möchte immer aktuell über Produktionszahlen informiert werden und abonniert Ereignisse von bestimmten Lesepunkten und Produkten. Ein Qualitätssicherungssystem möchte Produkte mit bestimmten Einzelteilen beim Eintreten bestimmter Zustände während der Produktion verfolgen. Die Lieferanten erwarten für eine effiziente Verwaltung der Bestände beim Hersteller (Vendor Managed Inventory) stets aktuelle Informationen über die Lagerbestands- und Nachfragedaten, um die Bestände ihrer Produkte beim Hersteller optimieren zu können. Das Kraftfahrt-Bundesamt (KBA) muss neue Fahrzeugtypen genehmigen und möchte über Zulassungen informiert werden, wobei Daten über einzelne Bestandteile, wie Partikelfilter oder Motor benötigt werden. Auch zur Abwicklung von Rückrufaktionen fordert diese Stelle Informationen über betroffene Fahrzeuge an.

Die Empfänger der Ereignisdaten, wie Landeszentrale des Herstellers, Qualitätssicherung, Management, Zulieferer, Kraftfahrtbundesamt oder Finanzamt, melden ihr Interesse an einem bestimmten Teil der Informationen in Form von Subscriptions einem der Broker. Dieser Broker sorgt mit einem effizienten Ereignisfilter für die schnelle und bedarfsgerechte Weiterleitung der relevanten Ereignisdaten von Ereignisproduzent zu Ereigniskonsument. Dabei dient der Broker gleichzeitig als Vermittler zwischen den unterschiedlichen Schnittstellen der beteiligten Systeme.

Ereignis- und Subscriptionaufkommen ➔ Effizienz

Um die Größenordnung der anfallenden Ereignisse an RFID-Lesegeräten zu verdeutlichen, lässt sich anhand des Anwendungsszenarios ein vereinfachtes Modell aufstellen. An jeder der s Produktionsstätten werden pro Tag n Einheiten gefertigt, wobei jede Einheit aus m Einzelteilen besteht. Jedes dieser Einzelteile wird o-mal erfasst und jede produzierte Einheit p -mal. Daraus ergibt sich ein Gesamtereignisaufkommen von $s \cdot n \cdot (p + o \cdot m)$ Ereignissen pro Tag und Hersteller. Für den bereits erwähnten Hersteller ergibt sich somit unter der Annahme $o = 10$ und $p = 25$ eine Summe von $60 \cdot 415 \cdot (25 + 10 \cdot 8.500) = 2.117.122.500$ Ereignissen pro Tag. Durchschnittlich fallen bei diesem Automobilhersteller somit 24.504 Ereignisse pro Sekunde an, welche durch das Brokernetzwerk verarbeitet werden müssen.

Eine ähnliche, exemplarische Rechnung lässt sich auch für die Anzahl der Interessenten aufstellen, was sich in Form von Subscriptions ausdrückt. In den 21 Staaten mit Fertigungsstätten gibt es jeweils ein Finanzamt und für jede der 60 Fertigungsstätten ist eine Gemeinde zur Berechnung von Steuern zuständig, wobei jeweils mindestens zwei Subscriptions angemeldet werden. Weiterhin gibt es in jedem der 153 Länder mit Verkaufsniederlassungen zentrale Zulassungsbehörden, die sich mit jeweils einer

Subscription angemeldet haben. Die 21 Landeszentralen und 5 Zentralen für jeden Kontinent haben jeweils 10 Subscriptions im System registriert. Dazu kommen noch die jeweils 5 Subscriptions für die 36.000 Zulieferer und jeweils 45 Subscriptions durch die verschiedenen Abteilungen in jeder der 61 Fertigungsstätten. Daraus ergeben sich für das verteilte Brokernetzwerk des Herstellers $(21+60)\cdot 2+153\cdot 1+(21+5)\cdot 10+36.000\cdot 5+60\cdot 45=183.275$ Subscriptions mit Interesse an einem ausgewählten Teil der Ereignisdaten.

3.1.2 Anwendungsszenario Gebäudeautomatisierung

In einem 5-geschossigen Wohn- und Geschäftshaus gibt es insgesamt 20 Wohnungen unterschiedlicher Größe sowie 3 Gewerberäume im Erdgeschoss. Die Tiefgarage bietet Platz für 120 Fahrzeuge. Elektrische Geräte, wie Lampen, Steckdosen, Rollläden, Displays, usw. lassen sich automatisch oder von beliebigen Stellen durch den jeweiligen Besitzer steuern. Zahlreiche Sensoren, darunter fallen intelligente Strom- und Wasserzähler, so genannte Smart Meter, sowie Feuermelder, Umweltsensoren oder Lichtschalter werden zum Teil gemeinsam von mehreren Personen oder anderen Interessenten genutzt. Sämtliche Geräte in diesem Szenario besitzen die Möglichkeit, Ereignisdaten beziehungsweise ein Interesse daran als Subscription zu generieren und über ein Netzwerk zu versenden. Dabei können verschiedene Arten der Anbindung, wie kabellos oder kabelgebunden, unterschieden werden. Feste Zuordnungen von beispielsweise Lampe und Lichtschalter, wie es in klassischen Haussteuerungen vorkommt, gibt es hier nicht. Es erfolgt somit eine Entkopplung der Geräte.

Möglich wird dies durch die Netzwerkfähigkeit der Sensoren und Aktoren, was einen asynchronen Nachrichtenaustausch ermöglicht. Die Netzwerkfähigkeit kann fest integriert sein oder durch die Anbindung an netzwerkfähige Steuergeräte, welche für die Generierung und Verarbeitung von Ereignisdaten verantwortlich sind, erreicht werden. Die Steuergeräte selbst besitzen die Möglichkeit, über einen Webserver die Netzwerkkommunikation mit dem Brokernetzwerk durchzuführen. Ein Vorteil bei der Realisierung von Hausautomationssystemen basierend auf den beschriebenen Steuergeräten ist die gleichzeitige Nutzung von konventionellen und somit kostengünstigen elektrischen Geräten ohne eigene Fähigkeiten zum Umgang mit Ereignisdaten in Kombination mit „intelligenten Geräten“, welche bereits die Fähigkeit zum Versenden und Empfangen von Subscriptions / Ereignisdaten über ein Netzwerk integriert haben. Die jeweilige Aufgabe muss dem Steuergerät initial per Konfiguration zugewiesen werden. Um ein neues Steuergerät zu konfigurieren, meldet sich dieses im Netzwerk per Broadcast an und erhält so die Information, mit welchem Broker kommuniziert werden kann. Durch die Konfiguration als Aktor (Subscriber) wird vom Steuergerät eine Subscription formuliert und an den zugeordneten Broker gesendet. Das Gerät ist somit in der Lage, Ereignisse zu empfangen, welche vom Inhalt her der Aufgabe des Steuergerätes entsprechen. Dient das Steuergerät lediglich zur Erfassung von Sensorwerten (Publisher), wird keine Subscription, sondern ein Advertisement zur Information über die Arten der zu produzierenden Ereignisse an den Broker gesendet.

Im anschließenden Betriebsmodus ist das Gerät nun in der Lage, Ereignisse aus den Informationen angeschlossener Sensoren zu generieren und an den Broker zu senden bzw. Ereignisse vom Broker entgegenzunehmen und damit angeschlossene Aktoren zu steuern.

Kabellose Anbindung von Publishern/Subscribern

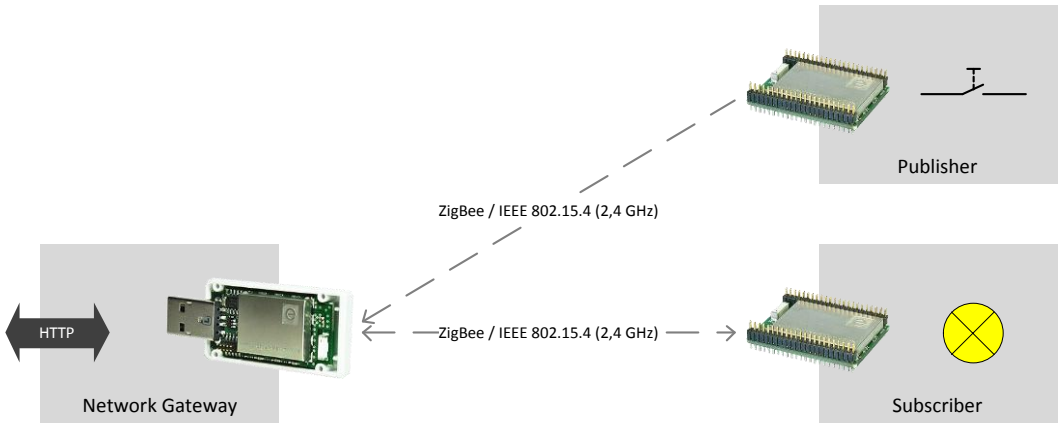


Abbildung 11: Kabellose Anbindung von elektronischen Geräten an den Broker (Quelle: eigene Darstellung[7])

Sehr flexible Einsatzmöglichkeiten bietet die kabellose Anbindung der Geräte in die Netzwerkarchitektur und somit an den Broker wie es in Abbildung 11 dargestellt ist. Die abgebildeten Module mit Mikrocontroller und einer integrierten Chip-Antenne basieren auf dem Standard ZigBee / IEEE 802.15.4 und sind dadurch besonders energieeffizient. Mit diesen Modulen werden im Freifeld Funkweiten von über 200 m bei einer Datenrate von bis zu 2 Mb/s [dre10] [Atm10] erreicht. In dem beschriebenen Szenario würden diese Funkmodule die Aufgabe der Steuergeräte übernehmen.

Zur Verdeutlichung der Anbindung eines elektrischen Geräts per Funkschnittstelle soll nun als Beispiel eine ereignisgesteuerte Lampe beschrieben werden. Diese Lampe übernimmt die Rolle des Subscribers, indem sie ihr Interesse an Schaltinformationen an das Publish/Subscribe-System übermittelt. Gleichzeitig fungiert die Lampe als Empfänger von Ereignisdaten, um beim Eintreffen des richtigen Ereignisses aktiviert zu werden. Die entsprechende Zuordnung übernimmt der Mikrocontroller des Funkmoduls inklusive dem Generieren der XML-Nachricht mit den entsprechenden Inhalten. Jedes dieser Publish/Subscribe-Module sendet die XML-Nachricht anschließend an das Funkmodul eines Netzwerk-Gateways, bei dem es angemeldet ist. Das Netzwerk-Gateway hat die Aufgabe, die Nachrichten der angemeldeten Publisher/Subscriber per HTTP-Schnittstelle in das Brokernetzwerk zu übermitteln, beziehungsweise umgekehrt bei dem Empfang von Ereignisdaten. Die Rolle des Publishers übernimmt ein Lichtschalter, welcher bei Betätigung über sein Funkmodul ein Ereignis generiert und via Netzwerk-Gateway an den Broker überträgt. Auf dem Broker kann nun der Ereignisfilter anhand der enthaltenen Ortsinformationen des Ereignisses überprüfen, an welche Lichtquelle das Ereignis gesendet werden soll. An dieser Stelle sind zahlreiche weitere

[7] Die Produktfotos stammen von http://www.dresden-elektronik.de/shop/cat4.html

Einsatzmöglichkeiten denkbar. Ändert die Lampe beispielsweise ihren Standort, kann dies mit Unterstützung der RFID-Technologie erfasst werden. Somit sind das Abmelden der Subscription bei Verlassen des Raumes und das Anmelden einer neuen Subscription bei Betreten eines Raumes erforderlich. Diese Funktion führt zu einer kontextsensitiven Nutzung der einzelnen Geräte, da die Lampe nur auf den Lichtschalter reagiert, der sich im selben Raum befindet. [Uhs09]

Brokernetzwerk

Sämtliche elektrischen Geräte, wie Sensoren und Aktoren in diesem Haus sind über ein IP-basiertes Netzwerk miteinander verbunden, welches ähnlich zu dem in Abschnitt 3.1.1 beschriebenen Szenario, dezentral strukturiert ist. In der folgenden Abbildung 12 ist die beschriebene Struktur auszugsweise für den Einsatz in einem Szenario zur Gebäudeautomatisierung grafisch dargestellt. Bei diesem Szenario kann davon ausgegangen werden, dass sämtliche dargestellten Komponenten die Funktionalität eines Steuergerätes bereits integriert haben und somit Subscriptions oder Ereignisdaten produzieren beziehungsweise verarbeiten können.

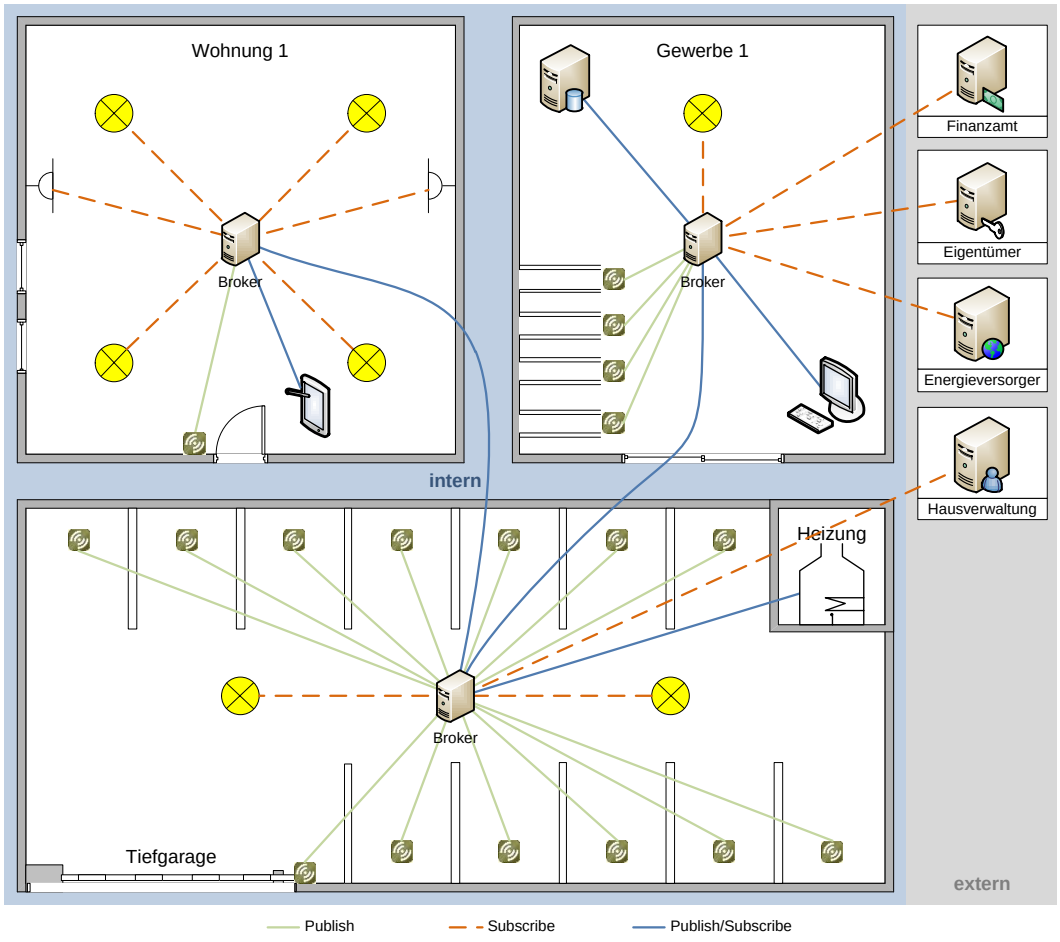


Abbildung 12: Informationsfluss im Brokernetzwerk am Beispiel einer Gebäudeautomatisierung (Quelle: eigene Darstellung)

Eine wichtige Eigenschaft dieses Systems ist die Möglichkeit, dass die Eigentümer der drei Bereiche Gewerbe, Wohnen und Parken sich jeweils für andere Systeme zur Hausautomation entschieden haben, was sich durch unterschiedliche Schnittstellen der einzelnen Steuergeräte bemerkbar macht. Jede Wohnung bzw. jeder Gewerberaum besitzt einen Broker, welcher zwischen Ereignisproduzent und -konsument vermittelt und die erforderlichen Schnittstellen der Beteiligten kennt. Alle Geräte können Ereignisdaten an den zugeordneten Broker senden oder ihr Interesse an ausgewählten Informationen anderer Geräte an einem Broker anmelden. Somit entsteht eine Publish/Subscribe-Architektur zur ereignisbasierten Kommunikation der Geräte untereinander. Die Filterung der Ereignisdaten und Weiterleitung an die richtigen Schnittstellen der Interessenten auf der Basis von XML-Nachrichten stellen die wesentlichen Unterschiede zu bereits existierenden Hausautomationssystemen dar.

Als Subscriber dienen in diesem Szenario, vereinfacht gesehen, sowohl beteiligte Personen (z.B.: Bewohner, Vermieter, Energieversorger, Hausmeister) über einen Rechner oder ein Terminal als auch Aktoren (z.B.: Lampe, Heizung, Steckdose, Kaffeemaschine). So bekundet beispielsweise der Energieversorger Interesse an den Stromverbrauchsdaten für jede einzelne Wohnung oder der Hausmeister möchte bei technischen Störungen umgehend informiert werden. In der Gruppe der Aktoren könnte die Lampe beispielsweise Interesse an Ereignissen haben, welche von Sensoren des Typs Lichtschalter kommen und die Eigenschaft Wohnzimmer aufweisen. Potentielle Empfänger melden somit ihr Interesse an bestimmten Informationen einem Broker und erhalten diese, sobald entsprechende Ereignisse vorhanden sind in dem bei der Anmeldung der Subscription vereinbarten Format.

Typische Ereignisproduzenten in diesem System sind alle Sensoren (z.B. Smart Meter, Temperatursensoren, Lichtschalter), die jeweils einem Broker zugeordnet sind. Die Sensoren senden nun Ereignisse an den Broker, ohne den Empfänger oder dessen Schnittstellen kennen zu müssen.

Bedarfsgerechte Erweiterung des Systems ➔ Skalierbarkeit

Im Falle einer Erweiterung des Gebäudes lässt sich das Publish/Subscribe-System durch das Hinzufügen von neuen Brokern zu dem bereits vorhandenen Brokernetzwerk beliebig erweitern. Neue Geräte werden dann Brokern mit freien Kapazitäten zugeordnet, welche für eine erfolgreiche Vermittlung sorgen. Somit ist auch die Verkleinerung des Systems durch Entfernen von einzelnen Brokern möglich, ohne dass Änderungen am Gesamtsystem notwendig werden.

3.2 Anforderungsanalyse

Aus der Zielstellung in Kapitel 1.2 und den beschriebenen Szenarien aus dem vorangegangenen Unterkapitel sollen nun die Anforderungen an einen Ereignisfilter analysiert werden. Zunächst werden sämtliche funktionalen Anforderungen gefolgt von den nichtfunktionalen Anforderungen beschrieben. Mit diesen Anforderungen lässt sich

schließlich ein Konzept erarbeiten, welches die Umsetzung eines Ereignisfilters beschreibt. In Kapitel 6 kann schließlich die Gültigkeit der Anforderungen und somit auch des Konzepts im Rahmen eines Integrationsszenarios und einer Performanceanalyse erörtert werden.

3.2.1 Funktionale Anforderungen

Verarbeiten von Subscriptions:

A010. Subscriber können Interesse an Ereignissen in Form von XML-Subscriptions am System anmelden.

A020. Subscriptions müssen valide zum ADiWa-Subscriptionformat vorliegen, um verarbeitet werden zu können. Die Struktur einer Subscription wird mittels XML-Schema-Definition spezifiziert. Mögliche Datentypen, Operatoren und Ereignistyphierarchien können durch dieses Schema vorgegeben werden.

A030. Jeder Empfänger wird durch einen Endpunkt eindeutig identifiziert, welcher als Ziel für die Weiterleitung der Ereignisse dient.

A040. Der Subscriber muss nicht zwangsläufig der Empfänger der Ereignisse sein, sondern kann auch das Interesse an Informationen für einen anderen Empfänger anmelden.

A050. Das An- und Abmelden von Subscriptions zur Laufzeit wird unterstützt.

A060. Die XML-basierte Subscriptionsprache aus dem ADiWa-Projekt wird nativ unterstützt. Weitere Formate müssen somit in diese Sprache überführt werden.

A070. Entsprechende Adapter ermöglichen die Abbildung etablierter Schnittstellen, wie JMS, WS-Notification und XPath sowie proprietäre Schnittstellen auf die XML-Sprache.

Verarbeiten von Ereignisdaten:

A100. Publisher senden Ereignisse in Form von XML-Nachrichten an die entsprechende Schnittstelle des Ereignisfilters.

A110. Heterogene Ereignistypen und Schnittstellen sollen durch den Filter unterstützt werden, d.h. es erfolgt keine Typisierung der Ereignisdaten in der XML-Nachricht. Die Identifizierung von Ereignissen und die Abbildung auf Empfänger erfolgt allein auf der Basis von Subscriptions.

A120. Die Verarbeitung von eingehenden XML-basierten Ereignissen erfolgt alternativ mittels DOM-API (wahlfreier Zugriff) oder Pull-API (sequentieller Zugriff).

A130. Eine Entscheidung über die Weiterleitung der Ereignisse erfolgt stets inhaltsbasiert.

A200. Der Ereignisfilter unterstützt verschachtelte Subscriptions, welche die Manipulation von Ereignissen zur Laufzeit ermöglichen.

Allgemeine Anforderungen:

A210. Der Broker muss alle Schnittstellen der beteiligten Instanzen kennen und entsprechende Adapter besitzen.

A220. Die Ereignisdaten / Subscriptions können zum Beispiel per HTTP über einen Adapter entgegengenommen werden.

A230. Der Broker muss direkt mit Publishern und Subscribern als auch mit weiteren Brokern in einem Brokernetzwerk kommunizieren können.

3.2.2 Nichtfunktionale Anforderungen:

A300. Die Implementierung erfolgt in der Programmiersprache Java unter Verwendung des OSGi-Konzepts.

A310. Der XML-Ereignisfilter kann als OSGi-Komponente in einen Broker integriert werden, welcher weitere Funktionalitäten durch OSGi-Bundles bereitstellt.

A320. Gewährleistung einer hohen Performanz bei gleichzeitiger Skalierbarkeit des Systems. Je nach Anwendungsfall und Einsatzgebiet muss der Ereignisfilter wenige hundert bis hin zu tausenden von Ereignissen pro Sekunde in Echtzeit verarbeiten können.

A330. Hohe Laufzeiteffizienz bedeutet gleichzeitig das Herausfiltern einer Vielzahl von Ereignissen bereits in frühen Verarbeitungsstufen, wenn diese keine benötigten Informationen tragen und somit für eine Filterung nicht in Betracht kommen. Somit kann durch eine frühestmögliche Filterung der Aufwand in nachfolgenden Verarbeitungsstufen reduziert werden.

A340. Das System muss sich nicht nur flexibel in verschiedenen Anwendungsszenarien verwenden lassen, sondern muss auch mit den speziellen Eigenschaften verteilter Systeme, wie z.B. dem Auftreten von Fehlern während der Verarbeitung, umgehen können.

A350. Ein quantitativer Nachweis der Effizienz erfolgt durch Leistungsmessungen.

4 VERWANDTE ARBEITEN

In diesem Kapitel wird der aktuelle Forschungsstand auf dem Gebiet von ereignisgesteuerten Architekturen, insbesondere von Publish/Subscribe-Systemen beleuchtet. Dazu werden zunächst die Eckpunkte einiger ausgewählter Datenstrukturen skizziert, welche in bereits verfügbaren Publish/Subscribe-Systemen zum Einsatz kommen. Anschließend werden die Möglichkeiten und Schwachstellen der verschiedenen Systeme analysiert, um Potentiale für die Konzeption eines XML-basierten Ereignisfilters ableiten zu können. Der Fokus liegt in diesem Kapitel auf den Vertretern der inhaltsbasierten Publish/Subscribe-Systeme. Im zweiten Teil dieses Kapitels werden neben der Datenstruktur weitere zentrale Konzepte aus verwandten Arbeiten beschrieben, welche der Konzeption eines XML-basierten Ereignisfilters zugrunde liegen.

4.1 Datenstrukturen von Publish/Subscribe-Systemen

Unter der Datenstruktur eines Publish/Subscribe-Systems ist die interne Darstellung der angemeldeten Subscriptions zu verstehen, welche dem Filtersystem die Abbildung der eintreffenden Ereignisse auf die entsprechenden Empfänger ermöglicht (engl. Matching). Für diese Struktur existieren zahlreiche verschiedene Ansätze, welche bereits durch die Umsetzung in Forschungsprojekten validiert werden konnten. Betrachtet werden an dieser Stelle drei ausgewählte und grundlegend verschiedene Konzepte von inhaltsbasierten Publish/Subscribe-Systemen. Aufgrund der geforderten Eigenschaft einer inhaltsbasierten Filterung für das zu entwickelnde System bieten diese Ansätze ein großes Potential, was jeweils analysiert wird

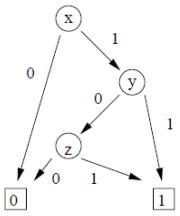
4.1.1 Binary Decision Diagram

Ein vielversprechender Ansatz ist die Darstellung als binäres Entscheidungsdiagramm (engl. Binary Decision Diagram, BDD), wie es in [Cam01] beschrieben wird. Motiviert wird dieser Ansatz durch den oft auftretenden Flaschenhals bei der Verarbeitung von eintreffenden Ereignisdaten. Um die Effizienz zu steigern leiten die Autoren folgende Anforderungen an die Filterkomponente ab, um veröffentlichte Ereignisse gegen angemeldete Subscriptions zu prüfen:

- **Ausdruckskraft**: Die Sprache zum Formulieren von Subscriptions sollte reichhaltig sein. Als Beispiel wird dazu der Java Message Service (JMS) genannt.
- **Effizienz**: Das Matching sollte möglichst in Echtzeit erfolgen.
- **Skalierbarkeit**: Der Filtervorgang soll durch eine sehr große Anzahl an Subscriptions nicht beeinträchtigt werden.

Gleichzeitig soll die Datenstruktur neben dem Anmelden auch das Abmelden von existierenden Subscriptions unterstützen.

Der Grundgedanke beim Design der Entscheidungsdiagramme ist die Vermeidung von mehrfachen Einträgen gleicher Subausdrücke und somit von unnötigen Auswertungsoperationen. Wenn davon ausgegangen wird, dass sich Subscriptions stark ähneln und nur in bestimmten Teilen unterscheiden, ergibt sich dadurch ein enormes Potential für Effizienzsteigerungen. Durch diesen Ansatz wird es erforderlich, sowohl die Subscriptions als auch die Ereignisse in atomare Subausdrücke zu zerlegen. Für jedes veröffentlichte Ereignis wird somit eine partielle Abbildung auf den booleschen Wert einer atomaren Formel vorgenommen. Wenn ein Ereignis das System erreicht, werden die passenden Subscriptions durch einen umgekehrten Traversierungsalgorithmus innerhalb des Diagramms gefunden. Zur Verdeutlichung von Struktur und Algorithmus dient das in Abbildung 13 grafisch dargestellte Beispiel der Booleschen Funktion: $x\ AND\ (y\ OR\ z)$. Jeder Knoten entspricht einer atomaren Booleschen Funktion, die Blattknoten entsprechen den Konstanten 0 und 1. Jeder Knoten, welcher nicht Blattknoten ist, hat den Ausgangsgrad 2, jeweils mit den Werten 0 oder 1 versehen.



Algorithm EvalBDD$(\mathcal{O}, \alpha)$
1 for $v := n$ downto 1
2 if v is terminal
3 then $value[v] := label[i]$
4 else $value[v] :=$
ITE$(\alpha(label[v]), value[low[v]], value[high[v]])$
5 output $value$

Abbildung 13: BDD für die Funktion $x\ AND\ (y\ OR\ z)$ sowie Filteralgorithmus (Quelle: [Cam01])

Entscheidend für den Aufbau eines BDD und dessen Größe ist die Ordnung der Variablen. Diese Ordnung ist bei dem genannten Beispiel mit $x < y < z$ schnell gefunden. Somit sind bereits für die Erstellung der Datenstruktur spezielle Algorithmen nötig, welche die optimale Variablenordnung berechnen, um die Anzahl der Knoten möglichst klein zu halten. Des Weiteren ist der ebenfalls in Abbildung 13 dargestellte Algorithmus zur Evaluation eines Entscheidungsdiagramms erforderlich.

Bei der Wahl einer geeigneten Datenstruktur zur Konzeption eines XML-Ereignisfilters wurde das Verfahren der BDD ausführlich betrachtet. Dabei zeigten sich einige Schwachpunkte, welche den Einsatz verhinderten. Gerade durch die Verwendung von XML-Nachrichten als Subscription-Format ergeben sich Probleme bei der Ordnung der atomaren Elemente, in die ein Dokument zerlegt werden muss. Bei jeder neu angemeldeten Subscription kann es so erforderlich werden, den kompletten BDD neu zu generieren, da sich eine andere Ordnung ergibt. Außerdem lassen sich die flexiblen Ausdrucksmöglichkeiten von XML, welche gerade durch die ADiWa-Subscriptionsprache ermöglicht werden, nur schwer als binäre Entscheidungsgraphen ausdrücken ohne doppelte Knoten zu erzeugen. Dies tritt vor allem dann auf, wenn die Ordnung über die XPath-Ausdrücke der Elemente ermittelt wird. Ein weiteres Problem stellt das Entfernen von Subscriptions dar, insbesondere wenn der beschriebene Ansatz von gemeinsamen

BDDs mit einer Zusammenfassung von atomaren Funktionen Anwendung findet. Durch die Eigenschaft, ein Ereignis stets auf 0 oder 1 abzubilden, stellt sich die Frage, wie die Routingentscheidung getroffen werden kann. Wenn es jeweils einen BDD für jede Empfängeradresse geben würde, muss geklärt sein, wie man damit bei der Filterung umgehen kann und wie man zahlreiche Redundanzen verhindern will.

4.1.2 Database Management System

Eine weitere Gruppe von Datenstrukturen für Publish/Subscribe-Systeme bilden die tabellenbasierten Filter (engl. Table Based Filter). In einem Projekt von Ashayer et al. [Ash02] wurde diesbezüglich untersucht, wie sich ein Publish/Susbcribe-System unter Verwendung von Standard Datenbanktechnologien implementieren lässt. Der grundlegende Ansatz basiert auf dem typischen Ablauf der Filterung: Zuerst werden die einzelnen Bedingungen (bei diesem Ansatz als Prädikate (engl. Predicates) bezeichnet), welche in der Datenstruktur erfasst sind, geprüft und anschließend daraus die passenden Subscriptions abgeleitet. Dabei stellten die Autoren fest, dass in der Praxis die Anzahl der möglichen Werte für die einzelnen Bedingungen oft auf eine feste, abzählbare Menge beschränkt ist, was durch Domänen bestimmt wird. Somit gelten als Hauptziele bei dieser Untersuchung die Entwicklung eines Algorithmus und einer Datenstruktur zur effizienten Verarbeitung von Prädikaten über einer festen Anzahl von Domänen und oft auch über einem festen Wertebereich. Des Weiteren soll durch das Ziehen von Schlussfolgerungen die Effizienz gesteigert werden. So lässt sich bei der Erfüllung eines Prädikats für eine bestimmte Domäne auf die Gültigkeit oder Ungültigkeit anderer Prädikate, bezogen auf diesen Bereich, schließen. Einen großen Vorteil sehen die Autoren in der Nutzung von vorhandenen Datenbank Management Systemen (DBMS), was die Neuentwicklung von anwendungszweckspezifischen Lösungen für die Verwaltung von Subscriptions in speziellen Datenstrukturen erfordert. Gleichzeitig wird darauf hingewiesen, dass die Realisierung mittel DBMS von der Performance her nicht mit den Spezialanwendungen mithalten kann, was sich vor allem bei hoher Last mit einer großen Zahl an Subscriptions zeigt.

Bei der Konzeption eines tabellenbasierten Ansatzes werden die drei Operationen Einfügen, Löschen und Aktualisieren von Prädikaten als Grundvoraussetzung betrachtet. Zur Beschreibung der Struktur dient das in Abbildung 14 veranschaulichte Beispiel. Demnach ist jedes Attribut eines Ereignisses über einer Domäne D definiert, wobei $type(D) \in \{integer, string, enumeration\}$ gilt. Die Prädikatentabelle der Abbildung entspricht einem zweidimensionalen Array für ein Attribut in einer bestimmten Domäne mit den Indexen Wert (attribute value) und Operator (predicate operator). Somit lässt sich durch einen Eintrag das Prädikat aus dem Tripel Attribut, Operator, Wert für eine bestimmte Domäne beschreiben ➔ $p(a, op, v)$. Ein Ereignis muss zur Filterung in Attribut-Wert-Paare zerlegt werden, wobei anhand des Hash-Wertes des Attributs die passende Prädikatentabelle ausgewählt wird. Anschließend wird der Index mit dem passenden Wert gesucht. Anhand des Operators kann nun für die jeweilige Subscription

eine entsprechende Schlussfolgerung getroffen werden. Für den >-Operator sind beispielsweise alle Prädikate an der Position $[>][l \ldots v-1]$ erfüllt.

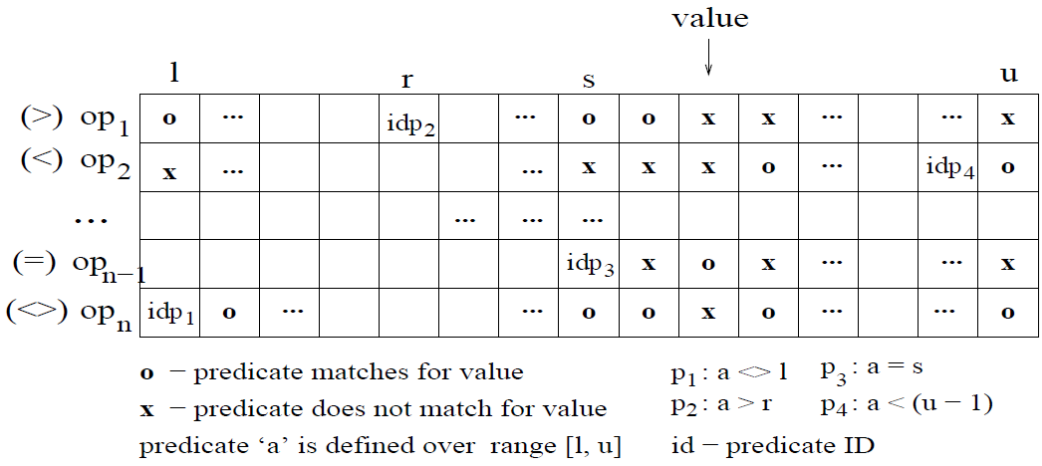


Abbildung 14: Datenstruktur für Prädikate einer bestimmten Domäne über einen endlichen Wertebereich (Quelle: [Ash02])

Dieser Ansatz ermöglicht es, in nur einem Tabellenaufruf alle passenden Prädikate zu ermitteln. Um aus diesen Information die entsprechenden Subscriptions ableiten zu können, wird ein Counting-Algorithmus genutzt. Unter der Annahme, dass Subscriptions als Konjunktionen von Prädikaten zu verstehen sind, kann mit diesem einfachen Algorithmus die Anzahl erfolgreicher Prädikate mit der Gesamtanzahl an Prädikaten einer Subscription bei der Filterung für jedes Ereignis überprüft werden. Bei gleicher Anzahl lässt sich schließlich über eine Assoziationstabelle die entsprechende Subscription ermitteln. Alternativ wird dafür auch die Struktur einer verknüpften Liste beschrieben, um die Assoziation herzustellen, was zu einer verbesserten Performance führt. Wie aus der Beschreibung bereits hervorgeht, ist die tabellenbasierte Datenstruktur sehr gut zur Umsetzung durch ein (relationales) DBMS geeignet, wobei die Subscriptions als Relationen gespeichert und manipuliert werden. Mit entsprechenden SQL-Anfragen lassen sich die Operationen durchführen.

Trotz des vielversprechenden Ansatzes lässt sich die tabellenbasierte Variante nicht problemlos für den gesuchten Ereignisfilter nutzen. Durch den geforderten universellen Charakter kann nicht davon ausgegangen werden, dass die Anzahl möglicher Domänen und Attribute in überschaubarer Größe bleibt, was zu einer enormen Anzahl an Datenbanktabellen führen würde. Gerade für Anwendungen mit einer großen Anzahl an verwalteten Subscriptions und einer hohen Ereignisverarbeitungsrate ist eine spezialisierte Implementierung dem tabellenbasierten Ansatz vorzuziehen.

4.1.3 Forwarding Table

Einen weiteren Ansatz zur schnellen Verarbeitung von Ereignisdaten in inhaltsbasierten Publish/Subscribe-Systemen beschreiben Carzaniga et al. in [Car03]. Im Vordergrund steht dabei die Entwicklung eines effizienten Algorithmus auf der Basis von sogenannten Weiterleitungstabellen (engl. Forwarding Table), welche eine schnelle und

inhaltsbasierte Zuordnung zwischen Nachrichten und Empfängern ermöglichen. Dieser Ansatz lässt sich von den bisher vorgestellten Systemen am besten mit den Anforderungen für einen XML-basierten Ereignisfilter verknüpfen, weshalb einige Punkte auch als Grundlage für die Konzeption dienen. Da dieser Ansatz auf einem inhaltsbasierten Netzwerk aufbaut, werden zunächst einige Grundprinzipien erläutert, welche die Basis für die Datenstruktur bilden. Hier zeigt sich, dass die Struktur nicht nur durch den Inhalt der Nachrichten, sondern auch durch die Gegebenheiten des Netzwerks bedingt ist.

In dem von Carzaniga et al. [Car03] beschriebenen, inhaltsbasierten Netzwerk werden die Ereignisse als Nachrichten (engl. Message) und die Subscriptions als Auswahlprädikate (engl. Selection Predicate) bezeichnet. Dabei wird vorausgesetzt, dass sich jede Nachricht als Menge von Attribut/Wert-Paaren darstellen lässt. Ein Attribut kann demnach durch seinen Namen, Typ und Wert beschrieben werden. Unter Prädikat wird eine logische Disjunktion von Konjunktionen von elementaren Bedingungen über einzelne Attribut/Werte-Paare (engl. Constraint) verstanden. Jedes Constraint hat somit einen Namen, einen Typ, einen Operator und einen Wert.

Als Unterschied zu IP-basierten Netzwerken wird in diesem Ansatz auf die Angabe einer Empfängeradresse verzichtet. Demzufolge wird in den Ausführungen von Carzaniga et al. die Subscription P eines Interessenten n auch als seine inhaltsbasierte Adresse p_n bezeichnet. Somit ist ein Ereignis m über seinen Inhalt implizit adressiert, wenn die Aussage $p_n(m) = true$ gilt.

Der Matching-Vorgang, d.h. die Zuordnung von Nachricht und Constraint, ist wie folgt definiert: Eine Nachricht passt zu einem Constraint, wenn es Attribute mit dem gleichen Namen und Typ enthält und wenn der Wert des Attributs dem Wert des Constraints genügt, was durch den Operator definiert ist.

Als Beispiel dient die Nachricht `[class="alert", severity=6, device-type="web-server", alert-type="hardware failure"]`, welche vom Prädikat `[alert-type="intrusion" ∧ severity>2 ∨ class="alert" ∧ device-type="web-server"]` adressiert wird. Als klassische Anwendung für ein solches abstrakt beschriebenes, inhaltsbasiertes Modell sehen die Autoren ein Publish/Subscribe-System. Ziel der weiteren Betrachtungen für diesen Ansatz ist es nun, geeignete Funktionen zu finden, um eine möglichst schnelle und skalierbare Weiterleitung auf der Basis von Forwardingtabellen zu gewährleisten. Dabei müssen sowohl zahlreiche komplexe Subscriptions als auch eine große Anzahl an Ereignissen berücksichtigt werden, welche jeweils von separaten Anwendungen generiert werden.

Als Referenzimplementierung für diesen Ansatz wird das Forschungsprojekt Siena (Scalable Internet Event Notification Architecture) beschrieben, welches durch die Autoren maßgeblich vorangetrieben wurde und als Prototyp frei verfügbar ist[8].

[8] Forschungsprojekt Siena: http://www.inf.usi.ch/carzaniga/siena/software/index.html

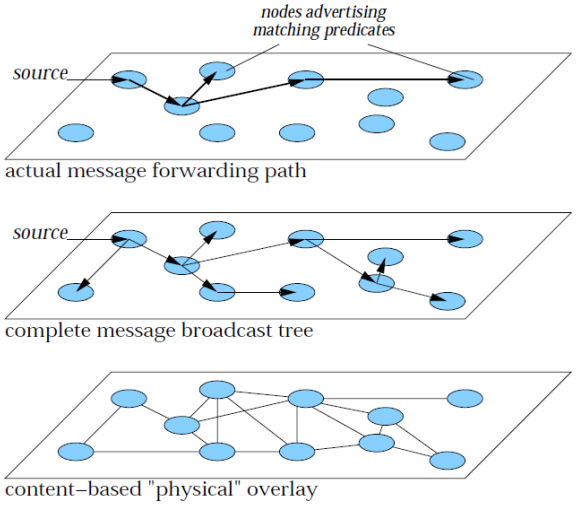


Abbildung 15: Inhaltsbasiertes Overlay-Netzwerk für High-Level-Routing (Quelle: [Car03])

Wie in Abbildung 15 dargestellt, läuft die inhaltsbasierte Weiterleitung (engl. Routing) auf einer höheren Anwendungsebene, dem Overlay-Netzwerk ab. Die mittlere Ebene der Abbildung zeigt das zyklenfreie Overlay-Netzwerk mit den logisch verbundenen Knoten, was für das Versenden von Broadcastnachrichten nötig ist. Dies entspricht somit dem Routing bei Realisierung als Broadcast-Protokoll. Auf der obersten Ebene wird das Routing von Nachrichten für ein inhaltsbasiertes Routingprotokoll dargestellt, wobei die Weiterleitung von Ereignissen nur auf Knoten beschränkt ist, welche anhand der Subscriptions als Empfänger identifiziert werden können.

Um die Verteilung der Subscriptions auf die einzelnen Netzwerkknoten zu optimieren, wurde für das Projekt Siena ein Advertisement-Konzept eingeführt [Pie07]. Dabei wird ein sogenanntes Fluten mit Subscriptions (engl. Subscription Flooding), d.h. die Verteilung der Subscriptions auf alle Netzwerkknoten per Broadcast, vermieden. Die Advertisements dienen dem Publisher, das Netzwerk über die Art der zu produzierenden Ereignisse zu informieren. Subscriptions müssen somit nur an die Netzwerkknoten verteilt werden, an denen auch entsprechende Ereignisse zu erwarten sind.

Zur Spezifikation eines inhaltsbasierten Routingprotokolls dient als zentrales Element eine Forwardingtabelle, wie sie auch in der linken Hälfte von Abbildung 16 abgebildet ist. Demnach repräsentiert die Tabelle eine Abbildung zwischen Empfängern (in diesem Projekt als Interface i bezeichnet) und Subscriptions (Prädikat p), wobei p_i als Assoziation zwischen Prädikat und Interface bezeichnet wird. Als Filter f wird in der Darstellung eine Konjunktion von Constraints bezeichnet ➔ zweite Spalte. Die Filter sind jeweils disjunktiv einem Interface zugeordnet ➔ erste Spalte. Die Constraints auf den einzelnen Attributen sind in der dritten Spalte beschrieben.

Um ein eintreffendes Ereignis weiterleiten zu können, wird eine Menge an Interfaces berechnet, die bestimmten Prädikaten zugeordnet sind, welche durch das Ereignis erfüllt werden. Die Funktion zur Berechnung dieser Menge basiert auf einem erweiterten Counting-Algorithmus, welcher an den Kontext eines Publish/Subscribe-

Systems angepasst wurde. Ziel ist es, sowohl die beschriebenen Konjunktionen als auch die Disjunktionen von Filtern zu unterstützen.

I_1	$f_{1.1}$	*string* $dest =$ *int* $price < 500$
	$f_{1.2}$	*string* $stock = DYS$ *int* $quantity > 1000$ *int* $price < 500$
I_2	$f_{2.1}$	*string* $airline = UA$ *string* $orig = Chicago$ *string* $dest = Atlanta$
	$f_{2.2}$	*string* $dest = New\ York$ *int* $price < 200$
	$f_{2.3}$	*string* $orig = Chicago$
	$f_{2.4}$	*string* $airline = UA$ *bool* $upgradeable = true$
I_3	$f_{3.1}$	*string* $stock = MSFT$ *int* $price < 200$

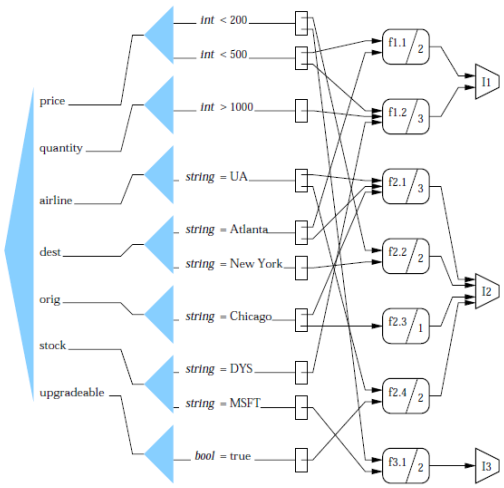


Abbildung 16: Beispielinhalt einer Forwardingtabelle a) inklusive dessen Repräsentation b) (Quelle: [Car03])

In Teil b) der Abbildung 16 ist die Zweiteilung der internen Datenstruktur erkennbar. Auf der linken Seite der Struktur befindet sich der Index mit allen Constraints der erfassten Prädikate, wobei deren Ausgänge mit den booleschen Eingängen der rechten Seite verbunden sind. Wenn der Algorithmus nun einen passenden Constraint findet, welcher der Nachricht entspricht, wird der jeweilige Ausgang auf true gesetzt. Die rechte Seite implementiert nun ein Netzwerk aus logischen Verbindungen, welche die Konjunktionen von Constraints in Filter und die Disjunktionen von Filtern in Interfaces übersetzen. Dabei sollte beachtet werden, dass diese Datenstruktur wie ein Wörterbuch aufgebaut und somit auf Suchoperationen (engl. lookup operation) optimiert ist. Modifikationen hingegen sind aufwändig und erfordern unter Umständen eine Neugenerierung der Datenstruktur als Ganzes. Dies liegt der Annahme zugrunde, dass der Nachrichtentraffic (also Suchoperationen) viel größer ist als der Steuerungstraffic (also Modifizierungen), welcher auch gepuffert abgearbeitet werden kann.

Der Counting-Algorithmus funktioniert nun wie folgt. Für eine eintreffende Nachricht m iteriert der Algorithmus durch die Attribute $a_1, a_2, \dots, a_k$ von m. Für jedes Attribut a_i wird ein Constraint $c_{i,1}, c_{i,2}, \dots, c_{i,n_i}$ gesucht, welches von a_i getroffen wird. Nun kann durch die erfolgreichen Constraints $c_{1,1}, c_{1,2}, \dots, c_{1,n_1}, \dots, c_{k,1}, c_{k,2}, \dots, c_{k,n_k}$ iteriert werden, um über die rechte Seite der Datenstruktur die passenden Filter zu finden. Dieser Algorithmus ist als Pseudocode in Listing 3 dargestellt. Für die Zuordnung zu den Interfaces benutzt der Algorithmus zwei Datenstrukturen. Eine Tabelle mit Zählern (engl. counter) für die teilweise getroffenen Filter sowie eine Ergebnismenge mit Interfaces, zu denen die Nachricht weitergeleitet werden soll. Für jedes gefundene Constraint wird der Zähler von allen verknüpften Filtern inkrementiert. Wenn nun ein Zähler die Gesamtzahl von Constraints erreicht, ist der Filter erfüllt und das zugeordnete Interface wird der Menge von Interfaces hinzugefügt, an die die Nachricht weitergeleitet werden soll.

An dieser Stelle werden von Carzaniga et al. nun ein paar Optimierungen beschrieben, um den Filtervorgang zu beschleunigen. Beispielsweise kann das Suchen in der

Zählertabelle für alle Filter eines Interfaces verhindert werden, wenn das Interface bereits in der Ergebnismenge enthalten ist. Auf der anderen Seite kann der Filtervorgang für eine Nachricht abgebrochen werden, sobald alle Interfaces in der Ergebnismenge enthalten sind, da durch eine weitere Filterung keine zusätzlichen Informationen gewonnen werden können. Ein weiterer Ansatz zur Optimierung ist die Implementierung der Datentypen des booleschen Netzwerks (rechte Seite der Datenstruktur) als Bitvektoren, da diese sehr effizient arbeiten.

```
proc counting CBF(message m) {
  map<filter,int> counters = ∅
  set<interface> matched = ∅
  foreach a in m {
    set<constraint> C = matching constraints(a)
    foreach c in C {
      foreach f in c.filters {
        if f.interface ∉ matched {
          if f ∉ counters {
            counters := counters ∪ ⟨f,0⟩
          }
          counters[f] := counters[f] + 1
          if counters[f] = f.size {
            output(m,f.interface)
            matched := matched ∪ {f.interface}
            if |matched| = total interface count {
              return
            }
          }
        }
      }
    }
  }
}
```

Listing 3: Pseudocode des Counting-Algorithmus aus dem Siena Projekt (Quelle: [Car03])

Dieser Carzaniga-Ansatz bietet zahlreiche relevante Strukturen und Prozesse zur Konzeption, was im folgenden Unterkapitel 4.2 näher erläutert wird. Dennoch soll an dieser Stelle auf ein paar Nachteile hingewiesen werden, weshalb der Ansatz nicht alle Anforderungen erfüllt. Das Ereignismodell lässt sich als strukturierter Verbund von Attribut/Wert-Paaren beschreiben, was die Nutzungsmöglichkeiten stark einschränkt. Um Ereignisdaten filtern zu können, müssen diese entsprechend typisiert sein, um erkannt zu werden. Die geforderte Unterstützung einer XML-basierten Sprache sowohl für Subscriptions als auch Ereignisse ist bei diesem Ansatz in der beschrieben Grundversion nicht gegeben, obwohl die Integration mit entsprechenden Erweiterungen durchaus denkbar wäre. Das schließt somit auch die Angabe von XPath-Ausdrücken als Attributnamen mit ein.

4.2 Relevante Ansätze für einen XML-Ereignisfilter

Wie bereits angedeutet, verspricht das Siena-Projekt ein großes Potential bei der Konzeption eines Ereignisfilters, da so bereits ein Großteil der Anforderungen abgedeckt werden kann. Gerade die beschriebene Datenstruktur bietet die Möglichkeit, mit ein paar Anpassungen XML-fähig zu werden. Dabei kann die Zweiteilung und die Nutzung

von Forwardingtabellen beibehalten werden. Auch die Optimierungen von Siena bezüglich des Advertisementkonzepts sollen in die Konzeption einfließen.

Um den Vorgang bei der Zuordnung von Constraints zu beschleunigen, führen Carzaniga et al. einen sogenannten Multioperatorindex ein, womit ein Großteil der Operatoren bereits durch die Identifizierung des Attributtyps und der Indexierung von Attributwerten ausgeschlossen werden kann. Dieser Sachverhalt ist in Abbildung 17 dargestellt.

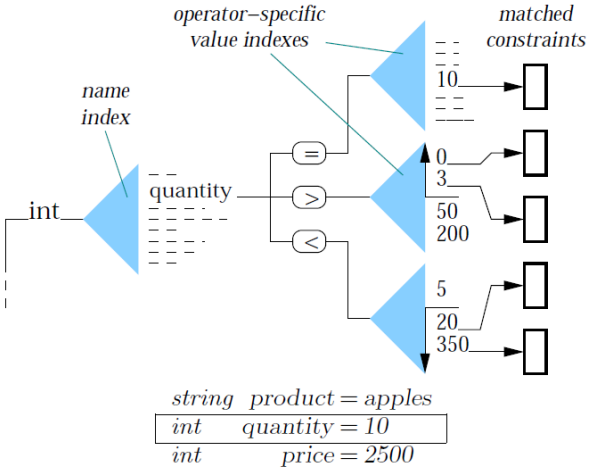


Abbildung 17: Beispiel für das Indexieren von Integer Constraints (Quelle: [Car03])

Die Indexierung kann beispielsweise durch eine aufsteigende Sortierung der Werte erreicht werden. Durch diese Ordnung von Integerwerten kann der Suchvorgang optimal ausgestaltet werden, beispielsweise über Suchbäume (TST).

Einen weiteren Optimierungsansatz bieten Selectivity-Objekte. Die Idee dahinter ist der Ausschluss möglichst vieler, potentieller Interfaces in einem frühen Verarbeitungsstadium. Der Ausschluss eines Interfaces kann dadurch die Auswertung einer großen Anzahl an Filtern und somit Constraints vermeiden. Dies wird teilweise bereits durch den im vorigen Unterkapitel beschriebenen Counting-Algorithmus realisiert. Tests dieses Ansatzes haben gezeigt, dass die Nutzung zusätzlicher Selectivity-Objekte die Performance in kritischen Situationen deutlich verbessert ohne zusätzliche Kosten zu verursachen.

Als Abgrenzung zum Carzaniga-Ansatz dienen somit die folgende Punkte:

- Unterstützung einer XML-basierten Subscriptionsprache, die durch ein Schema definiert ist
- Erkennen und Filtern von Ereignissen aus semistrukturierten XML-Daten ohne Kenntnis über deren Aufbau, das bedeutet ohne Schemadokument
- Austauschbarkeit und Erweiterbarkeit einzelner Bestandteile durch den Einsatz eines Komponentenmodells
- Unterstützung unterschiedlicher Clienttechnologien und deren Anbindung durch verschiedene Kommunikationsadapter
- Routing der Ereignisdaten im XML-Format

Um diese Aspekte gewährleisten zu können, ist die effiziente Verarbeitung von XML-Dokumenten ein zentraler Bestandteil des Konzepts. Auf der einen Seite muss die mächtige Subscriptionssprache unterstützt werden, welche aus dem ADiWa-Projekt stammt. Da diese durch ein Schema definiert ist, bietet sich der Einsatz der JAXB-Technologie an, was in einem separaten Abschnitt 5.5.2 beschrieben wird. Um einzelne Attribute in den XML-basierten Ereignisdaten adressieren zu können, steht der Subscriptionsprache der Einsatz von eingeschränkten XPath-Ausdrücken zur Verfügung. Zur Verarbeitung dieser Ausdrücke sind verschiedene Ansätze, wie die Nutzung eines Teils des XFilter-Konzepts [Alt00] denkbar. XFilter bietet die Möglichkeit, Subscriptions, welche aus reinen XPath-Ausdrücken bestehen, gegen ein XML-basiertes Ereignis zu überprüfen.

Die andere Seite der Verarbeitung von XML-Dokumenten beschreibt das Einlesen von XML-Datenströmen zur Identifizierung und Filterung von Ereignisdaten. Dazu beschreiben Diao et al. in einem Artikel [Yan101] die Verarbeitung von XML-Datenströmen, was den Anforderungen aus Kapitel 3.2 entspricht. Unter strombasierter Verarbeitung (engl. stream-based processing) verstehen die Autoren folgendes Prinzip. In XML-basierten Publish/Subscribe-Systemen kommen XML-Daten kontinuierlich von externen Quellen an und Subscriptions, welche im Broker als kontinuierliche Anfrage verfügbar sind, werden jedes Mal ausgewertet, wenn neue Daten empfangen werden. Gerade bei dem Eintreffen von großen Nachrichten zeigen sich die Vorteile des strombasierten Ansatzes, da die Verarbeitung bereits beginnen kann, bevor die Nachrichten komplett empfangen wurden. Diese Eigenschaft reduziert die durch den Filter verursachte Verzögerung enorm. Wie es auch in Abbildung 18 dargestellt ist, wird in dieser Ausarbeitung die Simple API for XML (SAX) zum Einlesen von XML-Dokumenten (engl. parsing) vorgeschlagen.

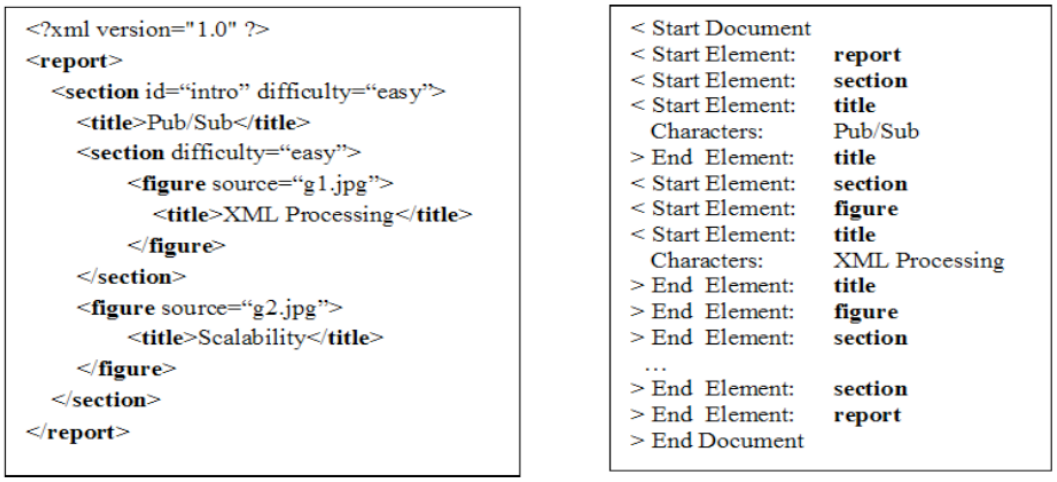


Abbildung 18: Als Beispiel ein XML-Dokument und die Ergebnisse des SAX-Parsings (Quelle: [Yan101])

Dabei wird die Struktur eines XML-Dokuments in eine lineare Sequenz von SAX-Ereignissen heruntergebrochen, was in der rechten Hälfte der Abbildung erkennbar ist.

5 KONZEPTION

Mit dem hier präsentierten Konzept wird ein Ansatz zur Realisierung eines vollständig XML-basierten Ereignisfilters vorgestellt. Bedingt durch die Umsetzung als Prototyp unter dem Namen „XMLEventFilter“ sind bereits in diesem Kapitel zahlreiche technische Details bezüglich Datenstrukturen und Java-spezifischen Implementierungsmöglichkeiten enthalten.

Zunächst wird der Gesamtkontext vorgestellt, um Integrationsmöglichkeiten zu skizzieren. Im Anschluss daran bildet die Subscriptionsprache als zentrales Element in einem Publish/Subscribe-System den nächsten Schwerpunkt. Darauf aufbauend, werden die Systemarchitektur und benötigte Komponenten des Ereignisfilters erläutert, bevor näher auf die Möglichkeiten einer Parallelisierung der Ereignisverarbeitung eingegangen wird. Den Abschluss dieses Kapitels bildet die detaillierte Beschreibung der Funktionsweise des Ereignisfilters anhand ausgewählter Beispiele.

5.1 XML-Ereignisfilter im Gesamtkontext

Wie bereits im einführenden Kapitel beschrieben, soll der zu entwickelnde XML-Ereignisfilter als eigenständige Komponente in einen Broker eingebettet werden. Dafür steht mit der OSGi-Softwareplattform eine flexible und dynamische Technologie bereit, welche das Grundgerüst für jeden Broker in dem beschriebenen Publish/Subscribe-System bildet. Die Grundlagen der OSGi-Softwareplattform wurden bereits im Kapitel 2.3.1 beschrieben, sodass in den folgenden Abschnitten die Einordnung des XML-Ereignisfilters in ein Gesamtsystem vorgenommen werden kann. Dabei gilt es, die benötigten Schnittstellen zu entwickeln, benötigte Adapter zu ergänzenden Technologien zu beschreiben und weitere Aspekte wie Sicherheit und Datenschutz zu beleuchten. Die zentrale Eigenschaft des Ereignisfilters ist die vollständige XML-fähigkeit. Dies bedeutet, dass keinerlei Typisierung der Ereignisse vorgenommen wird und so einen universellen Einsatz ermöglicht. Ein Ereignis liegt somit in einem völlig neutralen XML-Dokument vor und kann einzeln betrachtet auch nicht als solches erkannt werden. Erst wenn eine passende Subscription im Ereignisfilter vorliegt, kann das Ereignis im XML-Dokument gefunden und gefiltert werden. Es erfolgt also mit Hilfe der Subscription eine Abbildung von Ereignissen auf die Empfänger.

5.1.1 Brokerarchitektur

Um die beschriebene Funktionalität zu gewährleisten, sind somit neben der eigentlichen Filterkomponente zahlreiche weitere Bausteine erforderlich. Sämtliche, zur Vermittlung von Ereignisdaten zwischen Publisher und Subscriber, benötigte Komponenten werden auf einem Netzwerkknoten, welcher Broker genannt wird, zusammengefasst.

Die Architektur eines solchen Brokers gliedert sich in drei verschiedene Bereiche. Dies sind die Kommunikationspartner Publisher, Subscriber und Receiver, wobei Subscriber und Receiver die gleiche Rolle einnehmen können, wenn der Subscriber seine eigene Adresse als Empfänger angibt. Die zweite Gruppe bilden die Kommunikationsschnittstellen, welche Nachrichten in verschiedenen Formaten empfangen oder versenden können. Diese Schnittstellen bilden somit die Verbindung zwischen den Kommunikationspartnern und der dritten Gruppe, dem eigentlichen Broker Service. Abbildung 19 stellt die Architektur eines Brokers als Komponentenschaubild dar.

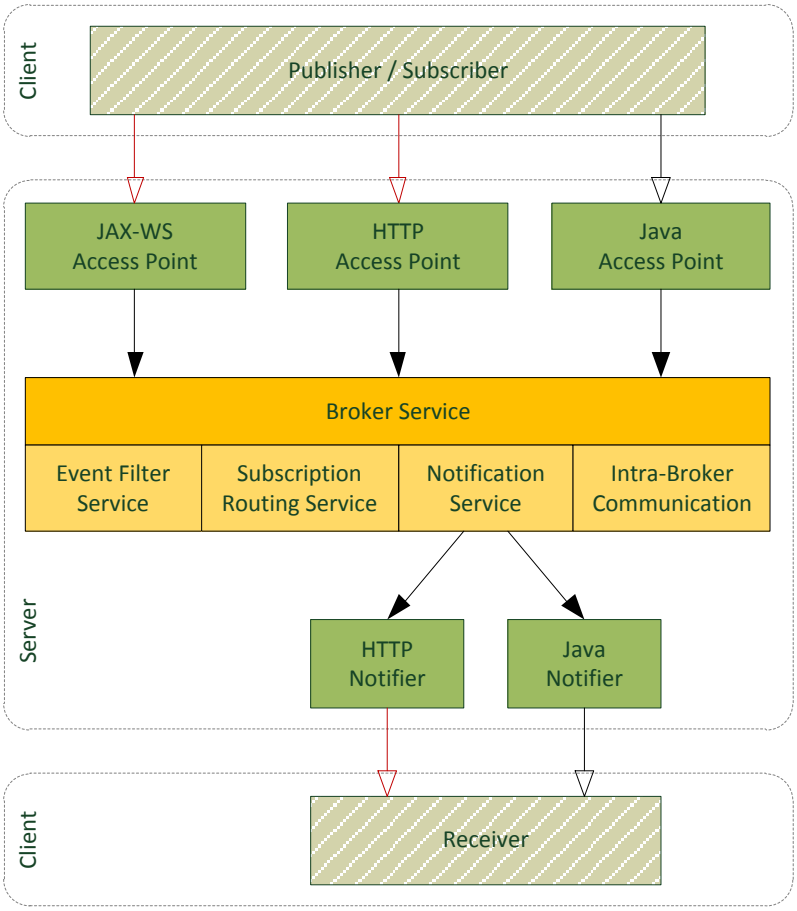


Abbildung 19: Übersicht zur Architektur eines Brokers (Quelle: eigene Darstellung nach [Gei10])

Die schraffiert dargestellten Bestandteile aus der ersten Gruppe sind nicht direkt auf dem Broker angesiedelt, treten aber bei der Nutzung seiner Funktionalitäten in Erscheinung. Somit werden diese auch brokerexterne Komponenten genannt und sind der Architektur hinzuzuzählen. Der zentrale Broker Service lässt sich durch vier Hauptkomponenten beschreiben. Der Event Filter Service bildet den zentralen Dienst zur Filterung der Ereignisdaten, was in den folgenden Abschnitten ausführlich erläutert wird. Daneben existiert der Subscription Routing Service, welcher für die optimale Verteilung der Subscriptions innerhalb des Brokernetzwerks sorgt. Im derzeitigen Konzept wird die Strategie des Flutens (engl. Flooding) verfolgt, worauf in Abschnitt 5.1.2 separat eingegangen wird. Der Notification Service ist für die Weiterleitung der Ereignisdaten über die richtige Schnittstelle an den oder die richtigen Empfänger

verantwortlich. Die letzte der vier Komponenten bildet die Intra-Broker Communication mit der Funktionalität zur Kommunikation der Broker untereinander. [Gei10]

Durch den Einsatz des Bundle-Konzepts lassen sich die verschiedenen Aspekte eines Brokers gut kapseln. Die verschieden Ausprägungen einzelner Komponenten können als Bundles realisiert werden, was einen einfachen Austausch ermöglicht. Auch bei den verschiedenen Eingabeadaptern für die Subscriptions (Kommunikationsschnittstellen) ist dieses Plug-In-Konzept sinnvoll, um eine Erweiterbarkeit und somit eine Anpassung an individuelle Bedürfnisse zu gewährleisten. Auch die jeweiligen Bundles der einzelnen Komponenten lassen sich in weitere kleine Bundles, so genannte Fragments, unterteilen. Somit können beispielsweise die Vergleichsoperatoren für die Ereignisfilterung kleine Bundles darstellen. Ein großer Vorteil ist, dass man die Bundles zur Laufzeit nachladen kann und eventuell sogar Operatoren zwischen Brokern austauschen kann.

Das Plug-In-Konzept wird auch einer weiteren Anforderung gerecht, da Bundles mehrfach in vielen Bereichen des Brokers genutzt werden können, wo mehrere Alternativen für prinzipiell die gleiche Funktionalität zur Verfügung stehen müssen.

5.1.2 Brokernetzwerk

Der zu entwickelnde XML-Ereignisfilter bildet einen zentralen Bestandteil in einem Publish/Subscribe-System und ist in der Regel auf einem Broker angesiedelt. Im einfachsten Fall steht der Broker als Vermittler in direktem Kontakt zwischen Ereignisproduzent und Ereigniskonsument. Mit dem Begriff Publisher werden im verteilten Brokernetzwerk Einheiten bezeichnet, welche Ereignisse an einen Broker senden. Einheiten, die ihr Interesse an diesen veröffentlichten Ereignissen bekunden, werden als Subscriber bezeichnet. Da ein Broker nicht nur Ereignisse entgegennimmt, sondern diese nach dem Filtern gezielt weiterleitet (zum Beispiel an einen weiteren Broker), kann er somit gleichzeitig Publisher und Subscriber sein. Die Verknüpfung einer beliebigen Anzahl an Brokern zu einem Netzwerk dient vorrangig der Lastverteilung und somit der Performancesteigerung. Alle Broker sind über eine physische Netzwerkverbindung (kabelgebunden oder kabellos) möglichst zyklenfrei miteinander verbunden und können sich somit Nachrichten senden. Das Hinzufügen von weiteren Brokerknoten zu einem vorhanden Netzwerk ist problemlos möglich, was eine gute Skalierbarkeit des Gesamtsystems gewährleistet. Die Anbindung von Publishern und Subscriber ist individuell von den unterstützten Protokollen des jeweiligen Brokers abhängig. Das Zusammenwirken der einzelnen Komponenten in einem Publish/Subscribe-System bei Verwendung der drei verschiedenen Nachrichtentypen Publish, Subscribe und Advertisement soll nun anhand eines Beispiels verdeutlicht werden. Das beschriebene Brokernetzwerk mit seinem Informationsfluss ist in Abbildung 20 dargestellt. Die Knoten P1/P2 stellen Publisher, B1-B7 stellen Broker und S1 stellt den Subscriber dar.

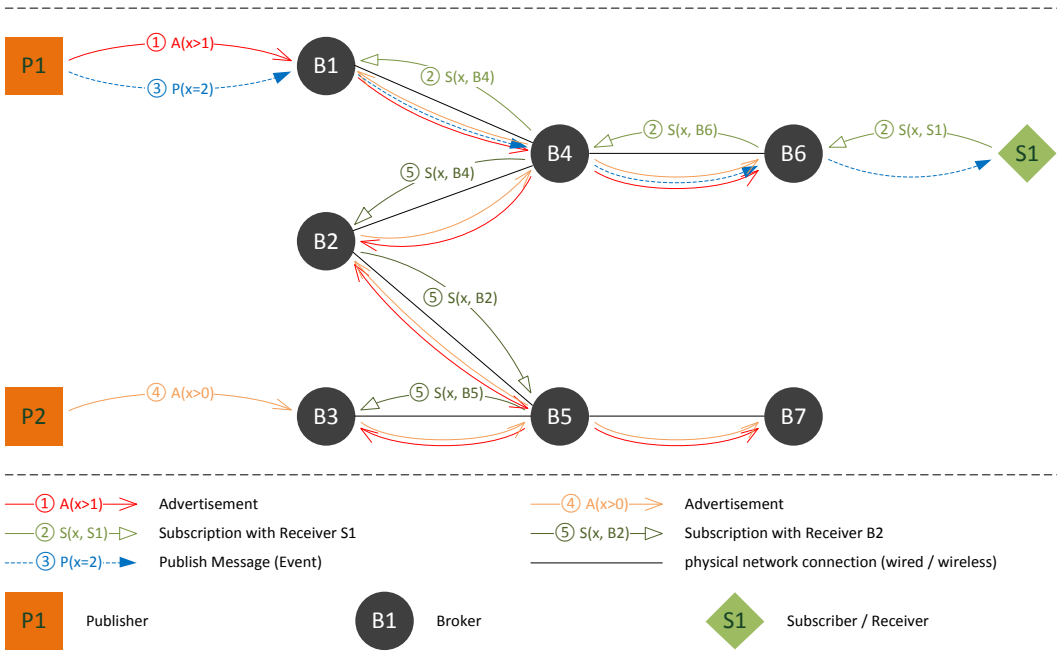


Abbildung 20: Informationsfluss innerhalb eines Brokernetzwerks (Quelle: eigene Darstellung)

Die einzelnen Schritte sind in der Abbildung fortlaufend nummeriert und werden im Folgenden erläutert.

Schritt 1 Advertise durch P1 und Flooding

Im ersten Schritt sendet Publisher 1 ein Advertisement, mit der Ankündigung später Ereignisse mit dem Wert x > 1 zu versenden an Broker 1. Diese Nachricht wird per Flutungsalgorithmus an alle Broker im Netzwerk verteilt (engl. Flooding), wobei sich jeder Broker merkt, von wem er die Nachricht erhalten hat.

Schritt 2 Subscribe durch S1

Subscriber 1 meldet sein Interesse an Ereignissen mit beliebigem x-Wert an Broker 6 an und trägt sich als Empfänger in die Nachricht ein. Broker 6 prüft nun anhand der bekannten Advertisements, ob er die Subscription weiterleiten muss. Da er ein Advertisement mit dem Wert x > 1 besitzt und dieses nicht von einem Publisher, sondern von Broker 4 erhalten hat, leitet er die Subscription an diesen Broker weiter. Dabei überschreibt Broker 6 die Empfängeradresse mit seinem Namen. Dieser Vorgang wiederholt sich bis ein so genannter Edge-Broker erreicht wurde und keine erneute Weiterleitung erfolgen kann.

Schritt 3 Publish durch P1

Publisher 1 sendet ein Ereignis an Broker 1, wohin dieser auch sein Advertisement gesendet hat. Da sich auf Broker 1 eine passende Subscription befindet, leitet dieser das Ereignis an den Empfänger (Broker 4) weiter. Die Weiterleitung wird solange wiederholt, bis der Subscriber erreicht wurde.

Schritt 4 Advertise durch P2

Nun sendet Publisher 2 ein Advertisement an Broker 3. Auch hier setzt nun das Fluten des Advertisements auf alle weiteren Brokerknoten ein. Da es nun aber Subscriptions im Netzwerk gibt, wird gleichzeitig auf jedem Broker geprüft, ob das Advertisement mit den lokalen Subscriptions kompatibel ist.

Schritt 5 Subscription forwarding

Wurde eine, zum Advertisement passende, Subscription gefunden, wird an diesem Broker Schritt 2 fortgesetzt, um die Subscriptions entlang des Advertisement-Pfades zu verteilen.

Auf weitere Schritte wurde aufgrund einer besseren Übersicht in diesem ersten Beispiel verzichtet. Soll nun auf die Advertisements verzichtet werden, wie es auch im derzeitigen Prototyp vorgesehen ist, so ergibt sich ein etwas anderer Ablauf. Der wesentliche Unterschied liegt in der Verteilungsstrategie der Subscriptions, die bei dieser Variante Fluten (engl. flooding) heißt und somit alle Subscriptions auf sämtliche Brokerknoten verteilt. Die Abbildung 21 gibt diesen Sachverhalt als zweites Beispiel grafisch wieder. Durch den Verzicht auf Advertisements beschränkt sich der Ablauf auf lediglich zwei Schritte, um den gleichen Zustand wie im ersten Beispiel zu erreichen.

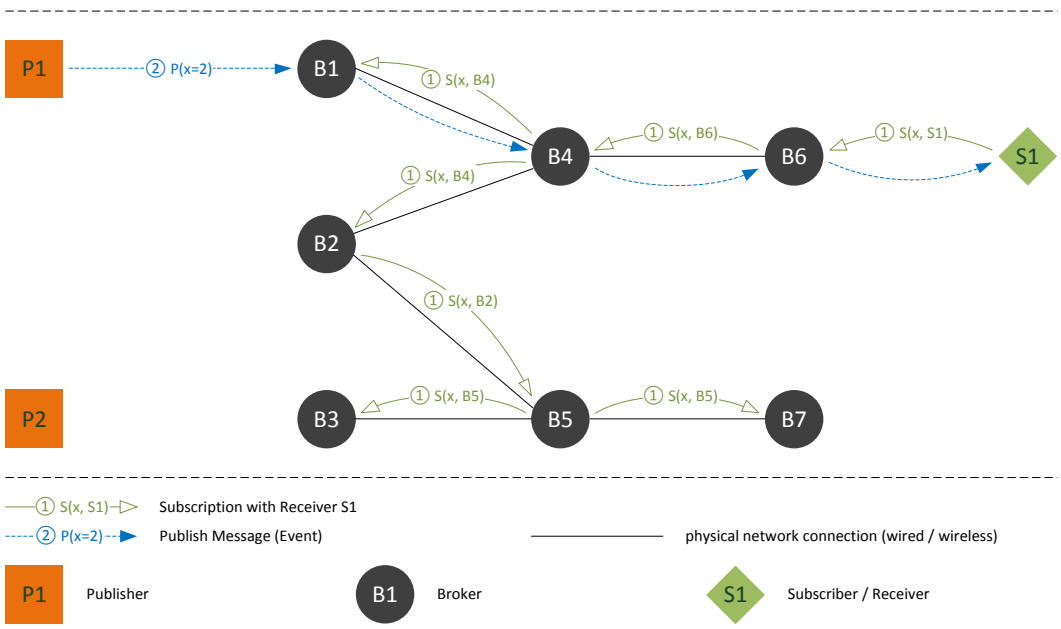


Abbildung 21: Informationsfluss innerhalb eines Brokernetzwerks ohne den Einsatz von Advertisements (Quelle: eigene Darstellung)

Wie in dieser Darstellung zu erkennen ist, führt das Fluten von Subscriptions nicht zwangsläufig zu einer höheren Last durch den Austausch von Nachrichten. Der Overhead, welcher durch den Einsatz von Advertisements entsteht, kann somit komplett entfallen. In Beispiel eins konnte lediglich ein Knoten bei der Verteilung von Subscriptions ausgespart werden, was einem Aufwand von 14 Advertisement-Nachrichten gegenübersteht. Vergleicht man den Aufwand bereits nach dem dritten

Schritt des ersten Beispiels, so stehen vier eingesparten Weiterleitungen von Subscription-Nachrichten nur noch sieben Advertisement-Nachrichten gegenüber. Aus diesen Beispielen lässt sich zwar keine allgemeingültige Aussage ableiten, dennoch zeigt es, dass die Größe des Brokernetzwerks und somit der Verwendungszweck über den Einsatz von Advertisements entscheiden sollte und beide Varianten betrachtet werden müssen.

5.1.3 Schnittstellen zur Kommunikation

Wie in Abbildung 19 zu erkennen ist, werden die Schnittstellen zur Kommunikation in zwei Kategorien aufgeteilt. Dies sind die Schnittstellen zur Eingabe und die Schnittstellen zur Ausgabe. Eingabe, was in der Grafik als Access Point bezeichnet wird, bedeutet für einen Broker die Entgegennahme von Subscriptions (subscribe) sowie das Empfangen von Ereignisdaten (publish). In beiden Fällen kann eines der bereitgestellten Verfahren genutzt werden. Konkret realisiert wurden bisher die Schnittstellen JAX-WS Access Point, HTTP Access Point und Java Access Point. Aus dem jeweiligen Namen lässt sich bereits die Aufgabe des Moduls ableiten. So stellt der JAX-WS Access Point per Application Server einen Webservice bereit, um Ereignisdaten oder Subscriptions mittels SOAP-Nachrichten entgegen nehmen zu können. Der HTTP Access Point nimmt Subscriptions oder Ereignisse als XML-Nachricht per HTTP entgegen, was durch einen eigenständigen Server realisiert werden kann. Über den Java Access Point können die Nachrichten als serialisierte Java-Objekte empfangen werden.

Die zweite Gruppe bilden die Ausgabeschnittstellen, was in der Grafik als Notifier bezeichnet wird. Diese dienen zur Auslieferung der gefilterten Ereignisdaten an die entsprechenden Empfänger mit passenden Subscriptions. Hier wird im Grundkonzept zwischen HTTP Notifier und Java Notifier unterschieden. Im ersten Fall wird aus den gefilterten Ereignissen ein XML-Dokument generiert und anschließend als HTTP Nachricht verschickt. Im anderen Fall werden die Ereignisobjekte direkt, das bedeutet ohne Berücksichtigung einer XML-Struktur, serialisiert und an den Empfänger gesendet.

Alle Access Points und alle Notifier sind jeweils als separate Bundles (auch Plug-Ins genannt) konzipiert und dadurch austausch- bzw. erweiterbar. Auch hier lassen sich wieder die Vorteile des OSGi-Konzeptes nutzen, um späteren und noch nicht im Konzept berücksichtigten Anforderungen gerecht werden zu können. Die speziellen Datentypen und Schnittstellenformate bilden die gemeinsame Grundlage für die Realisierung der Module als Plug-Ins. Diese werden im Wesentlichen als Java-Interface im Projekt `ADiWaCommonInterfaces`[9] spezifiziert und an entsprechender Stelle implementiert.

[9] Eine Übersicht zu `ADiWaCommonInterfaces` und allen am XML-Ereignisfilter beteiligten OSGi-Bundles befindet sich in Anhang 1 auf Seite 101.

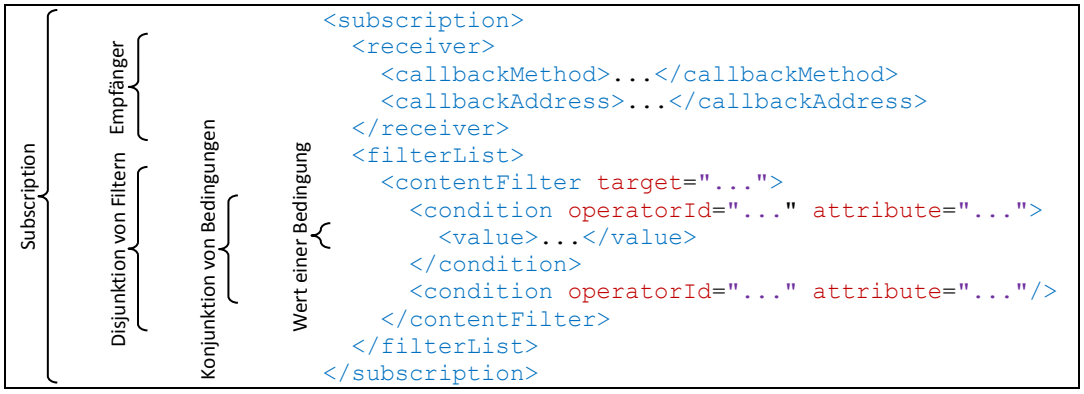
5.2 XML-basierte Subscriptionsprache

Die Ausdrucksmächtigkeit der Sprache für Subscriptions bestimmt maßgeblich die Funktionalität und Akzeptanz eines Publish/Subscribe-Systems. Somit steht für den XML Ereignisfilter eine XML-basierte Sprache zur Verfügung, welche aus dem ADiWa-Projekt stammt und durch einige wesentliche Eigenschaften gekennzeichnet ist. Diese Subscriptionsprache Projekt wird durch eine XML-Grammatik beschrieben, was in Form von Schemadefinitionen erfolgt.[10]

Da die Subscriptionsprache aus dem ADiWa-Projekt sehr eng mit dem Konzept des XML-basierten Ereignisfilters verknüpft ist und gleichzeitig dessen Basis darstellt, wird diese auch im Rahmen der Konzeption und nicht bei den Grundlagen beschrieben. Dennoch soll nun darauf hingewiesen werden, dass die Ausführungen in diesem Unterkapitel im Wesentlichen auf der Umsetzung der Vorgaben eines internen Arbeitsdokuments vom SAP Research Center Dresden [Gru10] basieren, welches die Subscriptionsprache beschreibt. Ein weiterer Grund für die Platzierung im Rahmen des Konzepts ist der verbliebene Interpretationsspielraum, den es zu füllen gilt.

5.2.1 Aufbau von Subscriptions

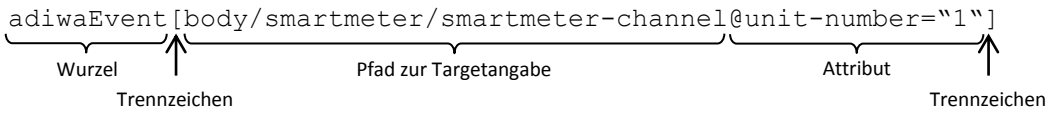
Der grundlegende Aufbau von Subscriptions wird in Listing 4 dargestellt. Demnach besteht jede Subscription zunächst aus einer Empfängerangabe (engl. Receiver), welche ein Ziel für die angeforderten Informationen festlegt. Dieses Ziel besteht zunächst aus einer optionalen Versandmethode für die Ereignisse (engl. Callback Method) und einer Zieladresse (engl. Callback Address). Fehlt die Callback Method, so wird in der eingelesenen Subscription dieser Parameter auf die Methode gesetzt, mit der die Subscription gesendet wurde. Die Empfängeradresse ist zwar obligatorisch, dennoch kann es vorkommen, dass auch diese nicht gesetzt ist, was sich durch ein leeres XML-Element ausdrücken lässt. Auch in diesem Fall bekommt die eingelesene Subscription als Zieladresse die Adresse zugewiesen, von der die Subscription gesendet wurde.



Listing 4: Aufbau einer Subscription

[10] Die kompletten Schemadefinitionen sind in Anhang 2 auf Seite 102 zu finden.

Zweiter Bestandteil einer Subscription ist die FilterList, welche eine Disjunktion von Filtern darstellt. Somit ist eine Subscription erfüllt, sobald einer der Filter durch ein Ereignis erfüllt wird. Ein Filter, zum Beispiel ein ContentFilter, besitzt stets eine Zielangabe (engl. Target), wo das gewünschte Ereignis in einem XML-Dokument gefunden werden kann. Eine Targetangabe wird in einer eingeschränkten Variante eines XPath-Ausdrucks angegeben, was zum Beispiel folgendermaßen aussehen kann:



Eingeschränkt bedeutet hier, dass keine Funktionen innerhalb von Prädikaten sowie Achsen oder Platzhalter wie „*“ zulässig sind [W3C07].

Zwingend notwendig ist stets die Angabe einer Wurzel, mit der ein Ereignis innerhalb eines XML-Dokuments identifiziert werden kann. Befindet sich die Targetangabe eines Ereignisses nicht im Wurzelknoten des Ereignisses, so kann mit einem Prädikat (in eckigen Klammern) ein Subausdruck als Pfad zu diesem Target angegeben werden. Bei Bedarf ist auch die Arbeit mit XML-Attributen möglich, was durch das Zeichen „@“ gekennzeichnet wird. Zum besseren Verständnis wird nun die Struktur der entstehenden Ereignisse aus den folgenden zwei Varianten einer Targetangabe verglichen (Listing 5). Als Beispiel findet sich direkt unter der Targetangabe ein mögliches resultierendes Ereignis.

Target 1:

```
adiwaEvent[body/smartmeter/smartmeter-channel]
```

Ereignis 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<adiwaEvent xmlns="http://www.adiwa.net/adiwaevent/1.0">
  <head>
    <eventType>
      <name>SmartMeterEvent</name>
      <version>1.0</version>
    </eventType>
    <eventId>2199780083199012781</eventId>
    <timestamp>2010-09-01T16:31:49.505-07:00</timestamp>
  </head>
  <body>
    <smartmeter-update>
      <smartmeter-channel>
        <voltage value="230.39" unit="V"/>
        <current value="1.94" unit="A"/>
        <power value="361.0" unit="W"/>
...
```

Target 2:

```
//smartmeter-channel
```

Ereignis 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<smartmeter-channel xmlns="http://www.adiwa.net/adiwaevent/1.0">
  <smartmeter-channel>
    <voltage value="230.39" unit="V"/>
    <current value="1.94" unit="A"/>
    <power value="361.0" unit="W"/>
...
```

Listing 5: Beispiele für Targetangaben in einer Subscription

Beide Versionen liefern jeweils die gleichen Attribute, allerdings fehlen bei Ereignis 2 die XML-Elemente, die sich außerhalb des „smartmeter-channel“ Knotens befinden, da dieser nun Wurzelknoten des resultierenden Ereignisses ist. Somit ergibt sich jeweils eine unterschiedliche Ereignisstruktur.

Jeder Filter besteht aus einer Konjunktion von atomaren Bedingungen (engl. Condition). Diese müssen stets vollständig durch ein Ereignis erfüllt werden, damit ein Filter erfüllt wird. Die Conditions beziehen sich auf je genau ein Ereignisattribut, besitzen einen Vergleichsoperator sowie eine beliebige Anzahl an Operanden. Diese Anforderungen werden in einer Condition durch Attribute und Subknoten ausgedrückt. So enthält eine Condition mindestens die zwei Attribute „attribute“ und „operatorId“ sowie optional das Attribut „quantifier“. Dazu kommen 0 bis n Operanden, was auch als Wert (engl. Value) bezeichnet wird und als Subknoten verwendet wird. Mit dem Attribute „attribute“ kann ein XML-Element oder XML-Attribut in einem Ereignis adressiert werden. Dies erfolgt, wie auch die Targetangabe, über eine eingeschränkte XPath-Schreibweise. Es kann dabei entweder der Pfad von der Wurzel des Dokumentes aus oder unterhalb der Wurzel des Ereignisses angegeben werden. Der Vergleichsoperator (engl. operatorId) beschreibt die Art und Weise, wie mit den Werten eines Ereignisattributes umgegangen werden soll. So prüft „attribute-exists“ beispielsweise lediglich das Vorhandensein eines Ereignisattributes und benötigt somit keinerlei Operanden. Der Operator „string-equals“ hingegen vergleicht den oder die Werte der Condition gegen den Wert des Ereignisattributes. Ein Beispiel für eine Liste mit mehr als einem Wert liefert der Operator „string-in-set“, welcher prüft, ob der Wert des Ereignisattributs in der Liste der Conditionwerte enthalten ist. An dieser Stelle können beliebige weitere Operatoren realisiert werden. Die optionale Angabe eines Quantors ermöglicht die Mehrwertigkeit von Ereignisattributen. Der Quantor „all“ bedeutet, dass alle Werte eines Ereignisattributs die Condition erfüllen müssen. Der Quantor „any“ bedeutet, dass es ausreicht, wenn lediglich ein Wert des Ereignisattributes die Condition erfüllt.

5.2.2 Beispiel einer Subscription

In Listing 6 wird eine Subscription im ADiWa XML-Format vorgestellt, welche Ereignisdaten mit Informationen von bestimmten Energiemessgeräten (engl. Smart Meter) abruft.

```
...
<subscription>
  <receiver>
    <callbackMethod>HTTP</callbackMethod>
    <callbackAddress>http://localhost:4002/submit</callbackAddress>
  </receiver>
  <filterList>
    <contentFilter target="adiwaEvent[body/smartmeter-update/smart-meter-
          channel]">
      <condition operatorId="string-equals" attribute="/body/smartmeter-
          update/smart-meter-channel/@smart-meter-unit-number">
        <value>3</value>
      </condition>
```

```
      <condition operatorId="string-equals" attribute="/body/smartmeter-
          update/smart-meter-channel/@smart-meter-channel-number">
        <value>1</value>
      </condition>
    </contentFilter>
  </filterList>
</subscription>
...
```

Listing 6: Beispiel einer Subscription im ADiWa XML-Format

5.2.3 Feingranulare Filterung

Ist eine Modifizierung von Ereignissen während der Filterung erforderlich, so spricht man von feingranularer Filterung. Dabei kommen so genannte geschachtelte Filter zum Einsatz, die dazu führen, dass Ereignisse den Ereignisfilter nicht mehr nur komplett passieren können oder verworfen werden. Vielmehr wird es dadurch möglich, dass einzelne Teile aus einem Ereignis herausgelöscht werden können. Um dies zu erreichen, wird das komplette Ereignis einer zusätzlichen Filterung unterworfen.

Generell sind für die Subscriptionsprache zwei verschiedene Profile vorgesehen, das erste Profil arbeitet ohne und das zweite Profil mit Unterstützung geschachtelter Filter. In der gegenwärtigen Implementierung des Konzepts wird das ausschließlich erste Profil unterstützt.

Um das Konzept des zweiten Profils näher beleuchten zu können, dient das mit Listing 7 eingeführte Beispiel. Hier wird ein Content Filter innerhalb eines anderen Filters (Grundfilter) angegeben und ist somit Teil der Konjunktion von Conditions. Dadurch ist es zwingend erforderlich, dass dieser Filter ebenso wie alle Conditions des Grundfilters erfüllt ist.

```
...
<contentFilter target="//ObjectEvent">
  <condition attribute="bizLocation/id" operatorId="string-equal">
    <value>urn:readpoint:bizloc3293</value>
  </condition>
  <contentFilter target="epcList/epc">
    <condition attribute="#content" operator="match-epc">
      <value>urn:epc:sgtin:12345.*</value>
    </condition>
  </contentFilter>
</contentFilter>
...
```

Listing 7: Geschachtelte Filter (Quelle: [Gru10])

Im Beispiel wird definiert, dass ein Ereignis durch den Knoten `ObjectEvent` im XML-Dokument identifiziert werden kann und anhand der Bedingung `bizLocation/id=="urn:readpoint:bizloc3293"` gefiltert werden soll. Der eingebettete zweite Filter wählt mit seiner Targetangabe alle Elemente `epcList/epc` des jeweiligen Ereignisses aus und startet eine erneute Filterung (rekursiv) in einem höheren Level. Mit der Condition wird angegeben, dass der Inhalt (`#content`) mit dem Operator `match-epc` gefiltert werden sollen. Die nicht zutreffenden EPCs werden somit aus der Liste herausgefiltert.

Anschließend verlässt der Filter diese Ebene wieder und muss die EPC-Liste mit dem restlichen Ereignis verschmelzen (konsolidieren). Im gefilterten ObjectEvent sind also nur noch solche `<epc>`-Elemente enthalten, deren Inhalt der `match-epc`-Bedingung genügt.

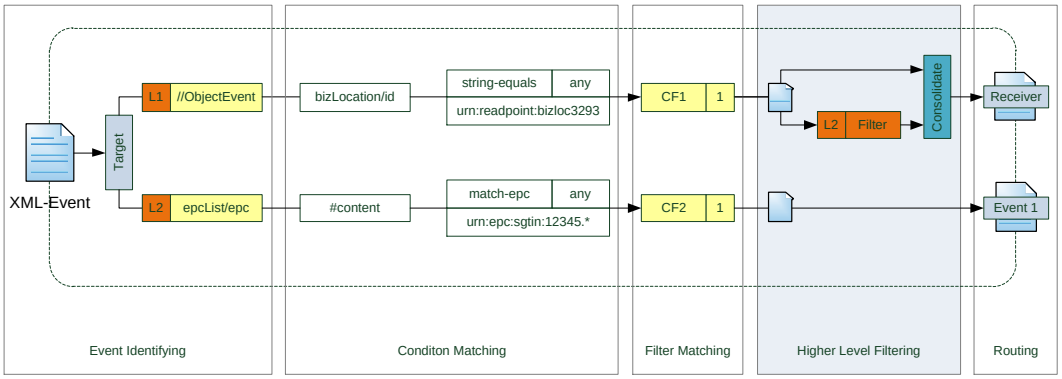


Abbildung 22: Filtervorgang mit geschachtelten Filtern (Quelle: eigene Darstellung)

Abbildung 22 stellt den Vorgang einer feingranularen Filterung nach Anmeldung der, in Listing 7 angegebenen, Subscription vereinfacht dar. Der rekursive Aufruf ist somit zwischen der eigentlichen Filterung und dem Routing angesiedelt. In jeder Rekursionsstufe wird somit ein höherer Filter-Level, in diesem Fall `L2`, aufgerufen. Die einzelnen Schritte der Filterung sind identisch mit dem ersten Profil der Subscriptionsprache, welches ohne Unterstützung geschachtelter Filter arbeitet. Dieser Vorgang wird ausführlich in Abschnitt 5.5.5 behandelt.

5.2.4 Konzeption einer Ereignistyphierarchie

Neben den beschriebenen Eigenschaften sind in der ADiWa Subscriptionsprache einige weitere Funktionalitäten, wie zum Beispiel eine Ereignistyphierarchie vorgesehen. Dadurch wird es möglich, Subscriptions anzumelden auch wenn zum Zeitpunkt des Anmeldens noch nicht bekannt ist, wo die Ereignisse in einem XML-Dokument zu finden sind. In diesem Fall wird der Subscription als Target ein Platzhalter mitgegeben, zum Beispiel `target="#ObjectEvent"`, welcher den Ereignisfilter veranlasst, in der Ereignistyphierarchie nachzuschauen, ob ein passender Eintrag vorhanden ist. Das Präfix # dient dabei als Unterscheidungsmerkmal zwischen abstrakter Targetangabe und XPath-Ausdruck. Erst wenn ein passender Eintrag zu einer solchen abstrakten Angabe gefunden wurde, kann auch das Ereignis im XML-Dokument identifiziert werden.

Als weiterer Vorteil ergibt sich aus dieser Hierarchie die Fähigkeit, einen allgemeinen Basistyp anzugeben, welcher verschiedene Arten von Ereignissen selektieren kann. Lautet die Targetangabe beispielsweise `target="#EPCISEvent"`, soll ausgedrückt werden, dass alle Subtypen von EPCISEvent[11] ausgewählt werden. EPCISEvent ist eine abstrakte Basisklasse von ObjectEvent, AggregationEvent, usw. und kann selbst keine Ereignisse adressieren, da es nie als XML-Elementname auftreten wird. Der Ereignisfilter prüft nun anhand der Ereignistyphierachie welche Elementnamen diesem Eintrag zugeordnet sind. Im Beispiel wären das die Einträge #ObjectEvent und #AggregationEvent. Da hier wieder das Präfix # auftritt, wird weiter rekursiv nach den jeweiligen XPath-Ausdrücken in der Ereignistyphierarchie gesucht. Somit kann eine Hierarchie aufgebaut werden und das Anmelden von mehreren Subscriptions mit dem gleichen Inhalt und lediglich verschiedener Targetangabe wird vermieden. Die Unterschiede zwischen der klassischen Erfassung und der Verwendung einer Ereignistyphierarchie wird in Abbildung 23 beispielhaft dargestellt.

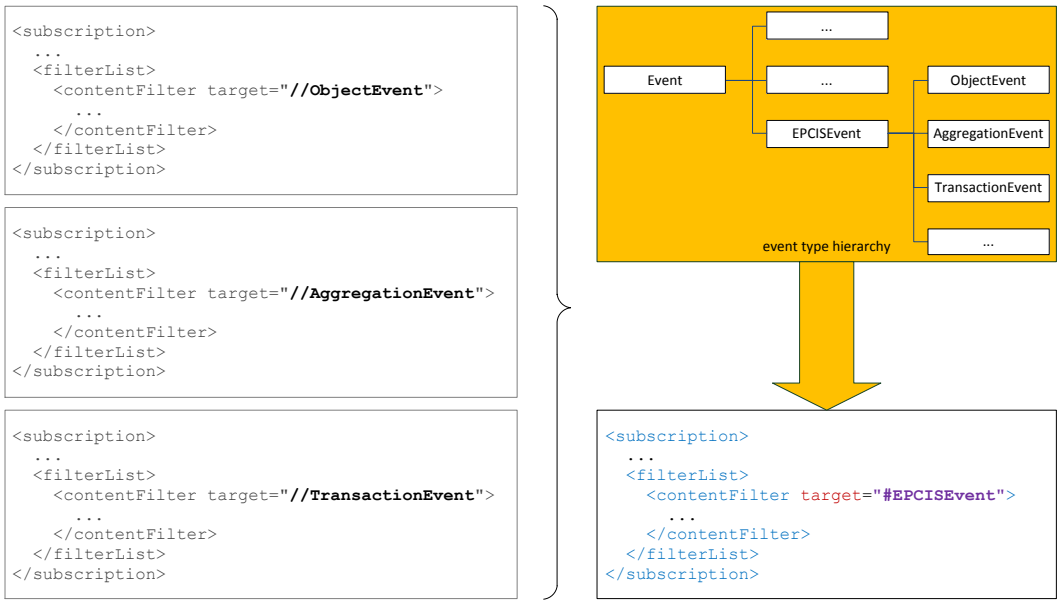


Abbildung 23: Subscriptions ohne und mit Einsatz einer Ereignistyphierarchie (Quelle: eigene Darstellung)

Um diese Anforderungen berücksichtigen zu können, muss die Möglichkeit bestehen, zur Laufzeit eine dynamische Hierarchie aufzubauen. Eine Erfassung von Ereignistypen als XPath-Ausdruck sollte somit im Vorfeld möglich sein oder innerhalb der Subscription übermittelt werden können. Auch ein selbstlernender Ansatz zur Optimierung der Datenstruktur ist hier möglich. Wenn der Subscription Manager des Ereignisfilters erkennt, dass sich die anzumeldende Subscription lediglich in der Targetangabe (als XPath-Ausdruck angegeben) von einer bereits vorhandenen Subscription unterscheidet, so wird die Targetangabe durch einen eindeutigen Bezeichner ersetzt und beide XPath-

[11] Der Indutriestandard EPCIS definiert zur Abbildung unterschiedlicher Anwendungsfälle vier Kernereignistypen (engl. Core Event Types), welche allesamt von einem generischen Ereignistyp mit dem Namen EPCISEvent abgeleitet sind: ObjectEvent, AggregationEvent, QuantityEvent und TransactionEvent. Gemäß der Spezifikation ist jeder dieser EPCIS-Ereignistypen mit einer Auswahl an Attributen (engl. Event Fields) ausgestattet, um die Fragen nach den vier Dimensionen Objekt, Zeitpunkt, Ort und Geschäftskontext beantworten zu können. [EPC07]

Ausdrücke gemeinsam mit dem Bezeichner der Ereignistyphierarchie hinzugefügt. Die Subscription muss somit nicht erneut in den Subscription Manager aufgenommen werden.

5.3 Systemarchitektur

Unabhängig von der Beschreibung des Integrationsszenarios aus den vorangegangenen Abschnitten, steht in diesem Unterkapitel die Architektur des XML-Ereignisfilters im Vordergrund. Neben den einzelnen Bestandteilen soll im Folgenden auch die Eigenständigkeit des entwickelten Konzepts zum Ausdruck kommen. Dazu werden einige zusätzliche Komponenten vorgestellt, die zum Beispiel der Kommunikation dienen. Auch an dieser Stelle können wieder die Vorzüge der OSGI-Technologie genutzt werden. Bevor technische Details erläutert werden, geht es zunächst eher abstrakt um die eigentliche Architektur der Filterkomponente.

5.3.1 XMLEventFilter als zentrale Komponente

Als Herzstück eines Publish/Subscribe-Systems stellt der Ereignisfilter einige wesentliche Funktionalitäten bereit. Dies sind neben der eigentlichen Filterung von Ereignisdaten die Verwaltung von Subscriptions und die Ermöglichung einer parallelen Verarbeitung. Einen detaillierten Einblick in das Thema Parallelisierung gibt das Kapitel 5.4. Aus den genannten Aufgaben und den dazu benötigten Schnittstellen ergibt sich unmittelbar eine Architektur der Filterkomponente, wie in Abbildung 24 dargestellt ist.

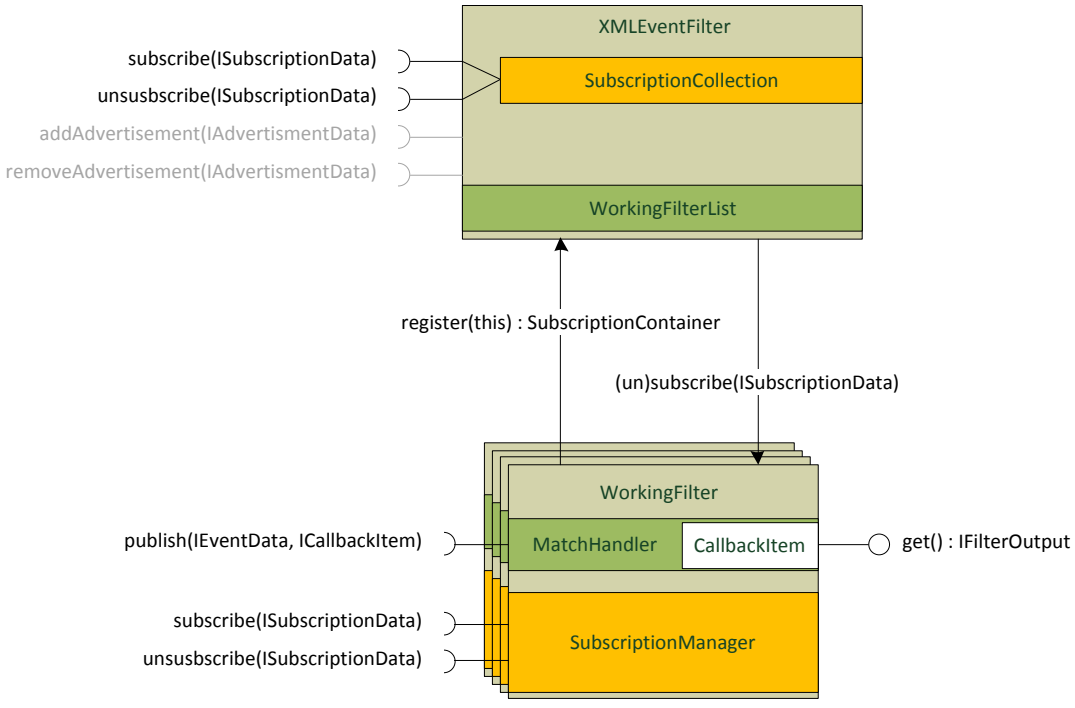


Abbildung 24: Systemarchitektur eines Ereignisfilters (Quelle: eigene Darstellung)

Diese ist im Wesentlichen zweigeteilt und besteht aus den Komponenten XMLEventFilter und WorkingFilter. Der XMLEventFilter dient als Steuereinheit und darf auf jedem Broker nur einmal instanziiert werden. Realisiert wird diese Komponente somit als Variante des Singleton-Entwurfsmuster, um zu verhindern, dass mehrere Instanzen des XMLEventFilters existieren [Eil10]. Daneben existiert eine beliebige Anzahl an WorkingFilter-Instanzen, welche parallel die Verarbeitung von Ereignisdaten durchführen.

Der XMLEventFilter besitzt zwei essentielle Datenstrukturen, welche die parallele Verarbeitung durch die einzelnen WorkingFilter ermöglichen. Dazu gehört die SubscriptionCollection als eine Art Masterliste. Diese Sammlung enthält alle Subscriptions in unveränderter Form, wie sie vom Subscriber übermittelt wurden. Die SubscriptionCollection ermöglicht die unkomplizierte Generierung einer Arbeitskopie für den Subscription Manager jedes WorkingFilters. Da jede Subscription über ihren Hashwert in die Collection aufgenommen wird, ist es nicht möglich, zwei identische Subscription am Ereignisfilter anzumelden. Existiert bereits eine Subscription mit dem gleichen Hashwert, so wird die neue Subscription abgewiesen. Um einen konsistenten Zustand für jeden WorkingFilter zu erhalten, erfolgt der Schreibzugriff auf diese Datenstruktur stets exklusiv. Die Verwendung einer eigenen Datenstruktur für jeden WorkingFilter ermöglicht somit die parallele Verarbeitung von Ereignisdaten in mehreren Threads.

Die zweite wichtige Datenstruktur ist die WorkingFilterList, welche Referenzen auf alle laufenden WorkingFilter enthält. Bei der Instanziierung meldet sich der WorkingFilter am XMLEventFilter an und wird in diese Liste aufgenommen. Als Rückgabe erhält dieser eine Kopie der SubscriptionCollection zur Erzeugung einer eigenen Datenstruktur. Die WorkingFilterList ermöglicht es nun, jeden einzelnen WorkingFilter über eintreffende Subscriptions zu informieren.

Da der XMLEventFilter die öffentlichen Schnittstellen für das An- und Abmelden von Subscriptions bereitstellt, übernimmt dieser gleichzeitig die Steuerung der resultierenden Prozesse bis zur Delegierung an die Working Filter. Die Hauptaufgabe dabei ist die Entgegennahme der Subscriptions in den verschiedenen Ausprägungen, also zum Beispiel als fertiges Subscription-Objekt oder in Form eines XML-Dokumentes, welches über einen Datenstrom in das System gelangt. Bei der ersten Variante wird das Objekt direkt in die Masterliste übernommen und eine Kopie der Subscription an alle WorkingFilter gesendet, welche in der WorkingFilterLitst registriert sind. Somit werden stets alle WorkingFilter über Veränderungen an der Datenstruktur informiert. Bei der zweiten Variante erfolgt vor diesen Schritten zusätzlich das Parsen des XML-Dokumentes und die Abbildung auf ein Objektmodell, was als XML-Binding bezeichnet wird. Da durch die Anforderungsanalyse die Implementierung mit der Java-Technologie vorausgesetzt wird, bietet sich die Java API for XML Binding (kurz JAXB) zur Realisierung des XML-Bindings an, was in den Abschnitten 2.3.2 beziehungsweise 5.5.2 genauer beschrieben wird.

Auch das An- und Abmelden von Advertisements soll durch den XMLEventFilter realisiert werden. Da nach dem derzeitigen Konzept allerdings die Strategie des Subscription flooding[12] innerhalb des Brokernetzwerks verfolgt wird, kann auf den Einsatz von Advertisements verzichtet werden. Die zentralen Eigenschaften, wie eine optimale Verteilung von Subscriptions auf die einzelnen Broker durch das Wissen über erwartete Ereignisse, kommen erst bei einem Wechsel der Verteilstrategie zum Tragen. Somit kann auch die Handhabung von Advertisements erst zu diesem Zeitpunkt konzipiert werden. Aufgrund der Ähnlichkeit zu Subscriptions ist allerdings davon auszugehen, dass diese ebenfalls per XML-Grammatik spezifiziert und mit der JAXB-Technologie verarbeitet werden können.

Der zweite Teil des in Abbildung 24 dargestellten Ereignisfilters ermöglicht die Ausführung des eigentlichen Filtervorgangs. Diese, als WorkingFilter bezeichnete, Komponente ist durch die beiden zentralen Datenstrukturen MatchHandler und SubscriptionManager gekennzeichnet. Der MatchHandler wird jedem Ereignis zugewiesen, welches vom Filter verarbeitet werden soll. Somit ist es möglich Zwischenergebnisse, welche bei der Filterung anfallen, zu speichern oder Zeitmessungen durchzuführen. Des Weiteren wird im MatchHandler ein als CallbackItem bezeichnetes Objekt gekapselt, was der Rückgabe der Filterergebnisse an die aufrufende Komponente dient (in den meisten Fällen sollte dies der Broker sein). Der Broker wartet mit diesem Objekt, welches das Future-Konzept[13] von Java implementiert, bis die aufbereiteten Ereignisdaten nach der Filterung mit den zugehörigen Empfängern abgeliefert werden. Dies geschieht über die blockierende get-Schnittstelle des CallbackItems. Der SubscriptionManager verwaltet die eigene Datenstruktur des WorkingFilters und ermöglicht mit seinen subscribe- und unsubscribe-Schnittstellen das An-/Abmelden von Subscriptions (Abschnitt 5.5.1). Der Filtervorgang wird jeweils durch Nutzung der publish-Schnittstelle gestartet, wobei jeweils Ereignisdaten an den WorkingFilter übergeben werden. Auch an dieser Stelle sind zwei wesentliche Formen der Ereignisse vorgesehen. Zum einen können diese als XML-Dokument in Form eines Datenstroms übermittelt werden und zum anderen als Ereignisobjekte. Bei der strombasierten Variante erfolgt zunächst das Parsen des Dokumentes mit der Streaming API for XML (kurz StAX), wohingegen die Objekte direkt verarbeitet werden können (Abschnitt 5.5.5).

5.3.2 Datenstruktur des XML-Ereignisfilters

Um die Vielzahl an möglichen Subscriptions verwalten zu können, ist eine spezielle Datenstruktur erforderlich, die bestimmte Voraussetzungen erfüllen muss.

[12] Mehr zum Thema Brokernetzwerk und der daraus resultierenden Lastverteilung ist in Abschnitt 5.1.2 zu finden.

[13] Ein Future-Objekt repräsentiert das Ergebnis einer asynchronen Berechnung. Dazu werden Methoden bereitgestellt, um das Ende der Berechnung zu prüfen bzw. darauf zu warten und das Ergebnis abzurufen. Das Ergebnis kann nur über die `get`-Methode abgerufen werden, wenn die Berechnung abgeschlossen ist, sonst blockiert die Methode.
vgl. http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html

- Gewährleistung der effizienten Abbildung von Ereignissen auf Subscriptions
  ➔ unterstützt durch die Ideen des Carzaniga-Ansatzes bezüglich der Datenstruktur und des Prinzips der Filterung.
- Berücksichtigung der Erweiterbarkeit bei Änderungen an der Subscriptionsprache
  ➔ unterstützt durch den Einsatz der JAXB-Technologie zur Generierung von Modellklassen mit angepassten Binding Declarations.
- Bildung einer gemeinsamen Basis für die Konzeption von Ereignisfilter und Broker
  ➔ unterstützt durch Bereitstellung von gemeinsamen Java-Interfaces und abgeleiteten Klassen zur individuellen Erweiterung, welche auch bei Modifizierung der Subscriptionsprache Bestand haben.

Die Aufnahme von Subscriptions zur effizienten Filterung erfordert somit eine spezielle Datenstruktur, welche zunächst anhand eines UML-Klassendiagramms dargestellt ist (Abbildung 25). Damit werden die Struktur der Subscription-Objekte und die Datentypen sowie deren Beziehungen untereinander verdeutlicht. Die Abbildung von Subscriptions in Form von XML-Dokumenten auf diese Datenstruktur erfolgt mit der JAXB-Technologie, was in Abschnitt 5.5.2 detaillierter beschrieben wird. Dazu ist eine XML-Schemadefinition erforderlich, welche den Aufbau und den möglichen Inhalt einer Subscription definiert.

Der jeweilige Name des Java-Pakets (engl. Package), dem eine Klasse zugeordnet ist, befindet sich in Abbildung 25 jeweils vor dem fettgedruckten Klassennamen. Alle in dem UML-Diagramm blau dargestellten Klassen wurden direkt aus dem XML-Schema-Dokument generiert und tragen das Suffix „Type“ im Namen. Diese Klassen modellieren somit die Struktur eines Subscription-Objekts. Durch eventuelle Änderungen an der Struktur einer Subscription und somit am XML-Schema-Dokument wird ein erneutes Generieren dieser „-Type“-Klassen erforderlich. Um dies zu berücksichtigen existiert zu jeder „-Type“-Klasse eine abgeleitete Klasse, welche stets dem Referenzieren in weiteren Datenstrukturen dient (in der Abbildung orange dargestellt). Die Funktionen und der Aufbau dieser Klassen (`Subscription`, `FilterList`, `ContentFilter`, `Condition`, `Receiver`, `ConceptFilter`, `SecurityInformation`) werden im Folgenden kurz beschrieben. Auch die generierten Klassen nutzen jeweils die abgeleiteten Klassen.

Die grau hinterlegten Klassen (`Target`, `Operator`, `Quantifier`, `Attribute`, `AbstractAttribute`, `Selectivity`, `SubscriptionAction`, `SubscriptionContainer`, `FilterContainer`, `ConditionContainer`, `ValueContainer`) stellen zusätzliche Datenstrukturen bereit, welche ebenfalls für die Funktionalitäten des Ereignisfilters erforderlich sind. Auch auf diese wird im Folgenden näher eingegangen. Weiß dargestellte Klassen entsprechen vorhandenen Klassen der Java-Standardbibliotheken und dienen in dieser Darstellung lediglich der Orientierung. Weitere Informationen dazu sind in den entsprechenden Dokumentationen zu finden.

Zur verbesserten Kapselung und Nutzbarkeit der Datenstruktur in weiteren Projekten, wie beispielsweise im Rahmen der Broker-Entwicklung, sind die meisten, der nun

vorgestellten, Klassen im separaten Plug-In-Projekt `XMLEventFilterData`[14] zu finden. Gekennzeichnet ist im Klassendiagramm das jeweilige Paket der einzelnen Klassen durch ein Präfix vor dem Klassennamen.

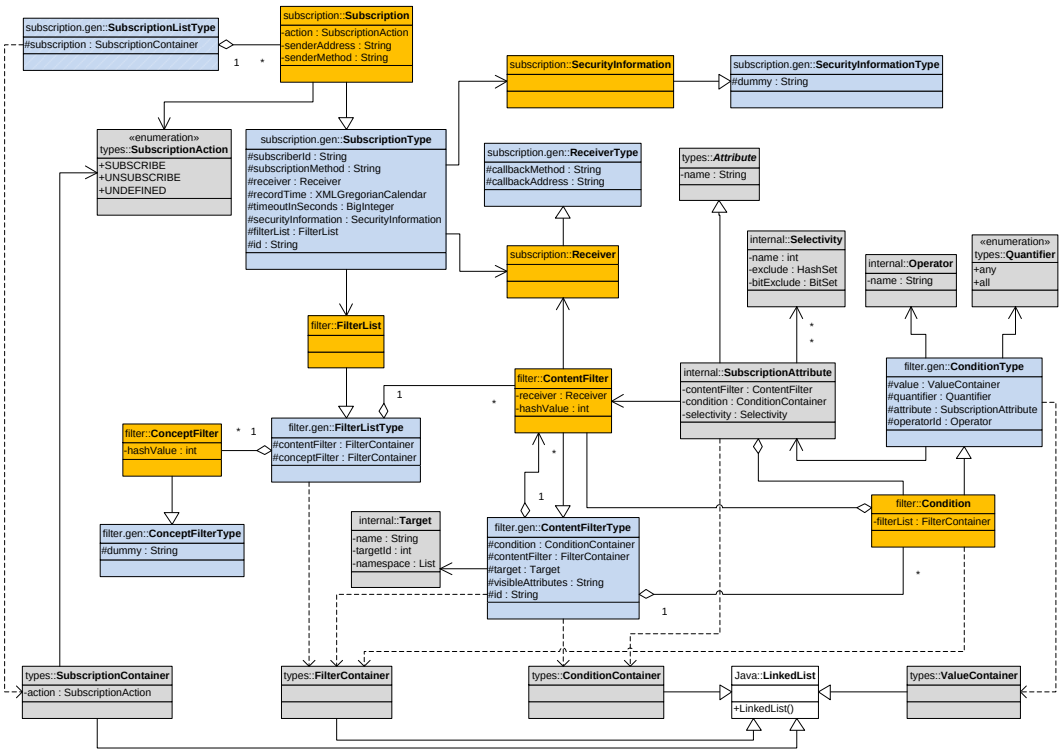


Abbildung 25: UML-Klassendiagramm zur Modellierung der Struktur und der Beziehungen von Subscription-Objekten (Quelle: eigene Darstellung)[15]

5.3.3 Plug-In-Projekte „XMLEventFilterData“ und „XMLEventFilter“

Im Folgenden werden die Aufgaben ausgewählter Klassen der Plug-In-Projekte `XMLEventFilterData` und `XMLEventFilter` vorgestellt, welche hauptsächlich an der Darstellung der Datenstruktur beteiligt sind. Es wird dabei auf das UML-Klassendiagramm in Abbildung 25 Bezug genommen.

- Package `subscription` und `filter` ➔ abgeleitete Klassen zur Beschreibung der Datenstruktur (orange dargestellt):

  `subscription:`**`Subscription`**

  Als `Subscription` wird eine Menge von Filter-Objekten (`ContentFilter` oder `ConceptFilter`) bezeichnet, welche disjunktiv verknüpft sind. Zu jeder

[14] Eine Übersicht zu `XMLEventFilterData` und allen am XML-Ereignisfilter beteiligten Java-Projekten befindet sich in Anhang 1 auf Seite 101.
[15] Zur besseren Lesbarkeit des Klassendiagramms ist eine vergrößerte Darstellung dieser Abbildung in Anhang 4 auf Seite 104 zu finden.

`Subscription` gehören genau ein `Receiver` sowie weitere Statusinformationen. Dies können ein eindeutiger Bezeichner (Id aus dem Hashwert), eine Angabe bezüglich des Typs (subscribe oder unsubscribe), Sicherheitsinformationen (z.B. Zugriffskontrolle) oder eine Subscriptionmethode (z.B. HTTP) sein.

`subscription:`**`Receiver`**

Ein `Receiver` als Bestandteil einer `Subscription` beschreibt mit einer `callbackAddress` einen Endpunkt, an den Ereignisse geliefert werden sollen. Über das zusätzliche Attribut `callbackMethod` wird eine Auslieferungsmethode, wie HTTP, definiert.

`subscription:`**`SecurityInformation`**

Mit `SecurityInformation` können bestimmte Sicherheitsaspekte, wie Zugriffskontrolle, realisiert werden. Dies ist im derzeitigen Konzept noch nicht vorgesehen.

`filter:`**`FilterList`**

Eine `FilterList` kapselt eine Sammlung von `ContentFilter`-Objekten und `ConceptFilter`-Objekten, was jeweils als `FilterContainer` realisiert ist.

`filter:`**`Condition`**

Eine `Condition` beschreibt eine elementare Bedingung, indem ein `SubscriptionAttribute` um einen Operator, einen `Quantifier` und einen `ValueContainer` zu einem 4-Tupel (`attribute`, `operatorid`, `quantifier`, `value`) erweitert wird.

`filter:`**`ContentFilter`**

Ein `ContenFilter` ist eine als Menge dargestellte Konjunktion von `Conditions` über den Inhalt von Ereignisdaten. Erweitert werden diese Bedingungen durch eine `target`-Angabe, um ein Ereignis in einem XML-Datenstrom identifizieren zu können.

`filter:`**`ConceptFilter`**

Ein ConceptFilter ist bisher nicht konzipiert, aber bereits in der Subscriptionsprache und somit auch in der Datenstruktur vorgesehen. Denkbar ist allerdings der Einsatz zur Auswahl bestimmter Ereignisse unabhängig von deren Inhalt. Der inhaltsbasierte Ereignisfilter wäre somit um zusätzliche Konzepte bezüglich der Filterung erweiterbar.

- Package `internal` und `types` ➔ weitere Klassen zur Beschreibung der Datenstruktur (grau dargestellt):

  `internal`**:Target**

  Der Typ Target dient der Kapselung einer Pfadangabe (XPath) zur eindeutigen Identifizierung von Ereignissen in einem XML-Dokument als semistrukturierte Datenquelle. Des Weiteren sind hier Angaben zu XML-Namespaces möglich, welche beim Parsen von Ereignissen eingelesen werden.

  `internal`**:Operator**

  Der `Operator` dient zur Selektion eines Algorithmus für die Überprüfung von Attributwerten aus Ereignissen gegen die Werte einer `Condition`. Hat der `Operator` beispielsweise den Namen `string-equals`, so liefert die `getOperator()`-Methode den Operator `OpStringEquals` zurück, um zwei Zeichenketten auf Gleichheit zu überprüfen. Alle `Op`*-Klassen implementieren das `IOperator`-Interface und stellen somit eine match-Methode zur Durchführung der Vergleichsoperationen bereit.

  `internal`**:SubscriptionAttribute**

  Ein `SubscriptionAttribute` erbt von `Attribute` und dient zur Kapselung eines Attributes aus einer `Subscription`, wobei sich neben dem Namen Verknüpfungen zu dem entsprechenden `ContentFilter` und den betreffenden Conditions abbilden lassen. Weiterhin ist jedem `SubscriptionAttribute` ein `Selectivity`-Objekt zugeordnet, mit dem sich bestimmte Empfänger ausschließen lassen.

  `internal`**:Selectivity**

  In einem `Selectivity` wird einem Attributnamen eine Liste von `Receiver`-Objekten zugeordnet. Diese `Receiver` können erst dann genutzt werden, wenn das entsprechende Attribut in dem Ereignis auftritt. Der Name des Attributes sowie der Empfänger werden im `Selectivity` als Hashwert erfasst.

  `types`**:Attribute**

  Der generische Datentyp `Attribute` dient zur Beschreibung von Attributen in einem Ereignis oder einer `Subscription` und wird durch eine `name`-Angabe beschrieben. Diese Angabe kann ein XPath-Ausdruck zur Adressierung in einem XML-Dokument darstellen.

  `types`**:Quantifier**

  Der `Quantifier` dient dem `ContentFilter` als Entscheidungshilfe bei mehrwertigen Ereignisattributen. So fordert der Quantor `all`, dass alle Ereignisattribute die Bedingung der Subscription erfüllen, wohingegen `any` lediglich

die Erfüllung durch mindestens ein Attribut verlangt. Als Standardverhalten gilt der `all`-Quantor.

`types:`**`SubscriptionContainer`**

Ein `SubscriptionContainer` realisiert eine Sammlung von `Subscription`-Objekten, die sich flexibel an wachsende Anforderungen bezüglich der Datenstruktur anpassen lässt. Im derzeitigen Konzept ist eine `LinkedList` von Java vorgesehen.

`types:`**`FilterContainer`**

Ein `FilterContainer` realisiert eine Sammlung von generischen Objekten zur Aufnahme von Filtern, die sich flexibel an wachsende Anforderungen bezüglich der Datenstruktur anpassen lässt. Im derzeitigen Konzept ist eine `LinkedList` von Java vorgesehen.

`types:`**`ConditionContainer`**

Ein `ConditionContainer` realisiert eine Sammlung von generischen Objekten zur Aufnahme von Conditions, die sich flexibel an wachsende Anforderungen bezüglich der Datenstruktur anpassen lässt. Im derzeitigen Konzept ist eine `LinkedList` von Java vorgesehen.

`types:`**`ValueContainer`**

Ein `ValueContainer` realisiert eine Sammlung von generischen Objekten zur Aufnahme von Werten, die sich flexibel an wachsende Anforderungen bezüglich der Datenstruktur anpassen lässt. Im derzeitigen Konzept ist eine `LinkedList` von Java vorgesehen.

`types:`**`SubscriptionAction`**

Der Typ `SubscriptionAction` legt fest, welche Schnittstelle durch eine `Subscription` bedient werden soll, `subscribe` oder `unsubscribe`.

- Package `types` und `xmleventfilter:filter` ➔ für den Ereignisfilter relevante Klassen (nicht abgebildet):

  `types:`**`AttributeIndex`**

  Der `AttributeIndex` verwendet eine `HashMap` von Java zum Aufbau einer Liste aller `Attribute`-Objekte die einem bestimmten `Target` zugeordnet sind. Im Vergleich zu dem beschrieben Carzaniga-Ansatz aus Kapitel 4.1.3 entspricht dieser Index dem Index mit allen Constraints, wobei hier eine Erweiterung bezüglich der `Target`-Angabe vorgenommen wurde.

types:**FilterIndex**

Der FilterIndex verwendet ebenfalls eine HashMap, allerdings werden hier FilterContainer dem jeweiligen Receiver zugeordnet. Dies entspricht, verglichen mit dem Carzaniga-Ansatz, dem Netzwerk aus logischen Verbindungen, welche die Konjunktionen von Constraints in Filter und die Disjunktionen von Filtern in Interfaces übersetzen. Dadurch wird eine Anwendung des Counting-Algorithmus möglich.

types:**Event**

Mit einem Event-Objekt wird ein Ereignis aus einem XML-Dokument repräsentiert, welches intern die Attribut/Wert-Paare als eine Liste von Attribute-Objekten erfasst. Das in diesem Objekt abgebildete Ereignis kann in einem XML-Dokument durch einen XPath-Ausdruck lokalisiert werden, der als Target-Angabe in dem Event-Objekt gesichert wird. Somit unterstützt diese Target-Angabe gleichzeitig die Serialisierung in ein XML-Dokument, falls das Ereignis nach der Filterung beispielsweise erneut per HTTP-Nachricht verschickt werden soll.

xmleventfilter:filter:**SelectivityTable**

Die SelectivityTable ist eine geordnete Liste mit Selectivity-Objekten, welche den Ausschluss von Empfängern ohne das Überprüfen von Typen, Operatoren, Werten und Filtern ermöglicht. Der erste Eintrag in dieser Liste ist das Element mit dem höchsten Selectivity-Level. Je mehr Receiver einem Attribut zugeordnet sind, desto höher steigt das entsprechende Selectivity in der Liste (Selectivity-Level). Dies bedeutet: Fehlt das Attribut mit dem höchsten Selectivity-Level in dem zu verarbeitenden Ereignis, kann die größte Anzahl an Receivern ausgeschlossen werden. Das letzte Element in der SelectivityTable hat den kleinsten Selectivity-Level. Mit der Angabe von preprocess_rounds kann die Anzahl der zu überprüfenden Selectivities in dieser Liste beschränkt werden. Durch diese Vorverarbeitung kann die Filterung durch Verkleinerung der möglichen Empfängermenge zur Laufzeit optimiert werden.

5.4 Parallelisierung

Die parallele Verarbeitung von Ereignisdaten bedeutet in diesem Zusammenhang das Verteilen der Last, was in zwei Arten erfolgen kann. Die erste Variante ist die brokerinterne Parallelisierung, welche durch die Brokerarchitektur unterstützt wird. Somit wird es möglich, Systeme mit mehreren Kernen oder Prozessoren bei hohem Ereignisaufkommen optimal auszulasten. Bei dem Eintreffen von Ereignissen am Broker können diese auf die instanziierten WorkingFilter aufgeteilt werden, welche auf verschiedenen Prozessorkernen arbeiten können. Jeder WorkingFilter läuft somit in einem einzelnen Thread, der bei entsprechender Anzahl an Prozessorkernen parallel ausgeführt werden können. Die Verwaltung der einzelnen Threads wird durch einen Thread-Pool gesteuert, der vom Broker bereitgestellt werden muss. Die Verteilung der

einzelnen Threads auf die Prozessorkerne wird in der Regel vom Betriebssystem organisiert.

Die zweite Art der Parallelisierung ist die brokerexterne Lastverteilung. Diese wird durch das Brokernetzwerk gewährleistet und ermöglicht eine betriebssystemunabhängige, parallele Ereignisverarbeitung auf physisch voneinander getrennten Brokern. Die Verteilung von Subscriptions und Ereignisdaten erfolgt gemäß den Routingalgorithmen über die vorgesehenen Kommunikationskanäle der per Netzwerk miteinander verbundenen Broker.

5.5 Funktionsweise des Ereignisfilters

Die Filterung von heterogenen, XML-basierten Ereignisdaten anhand von registrierten Subscriptions erfolgt stets nach einem sechsstufigen Schema, welches in Abschnitt 5.5.5 im Detail erläutert wird. An dieser Stelle werden auch gewisse Ähnlichkeiten zu dem bereits in Kapitel 4.1.3 beschriebenen Carzaniga-Ansatz deutlich, zum Beispiel durch den Einsatz des Counting-Algorithmus oder die Zweiteilung der Datenstruktur in einen Index zur Abbildung von Attributen und einen Index zur Abbildung der Filter.

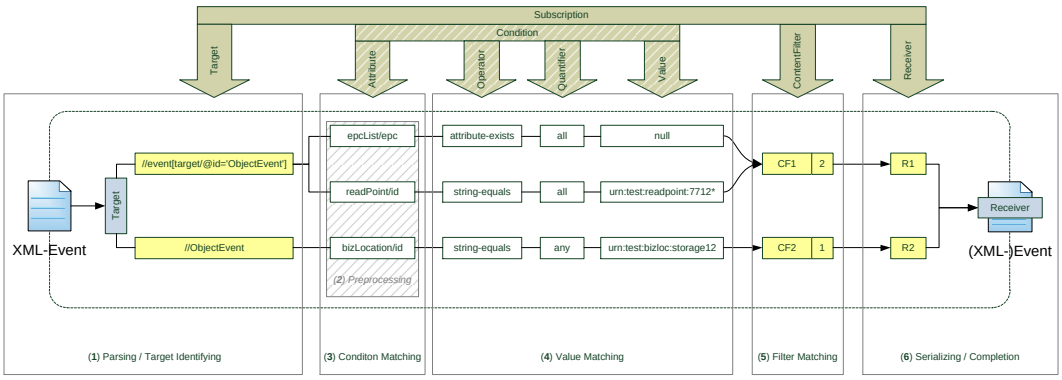


Abbildung 26: Funktionsweise eines XML-basierten Ereignisfilter (Quelle: eigene Darstellung)

Bevor auf die einzelnen Schritte des in Abbildung 26 beispielhaft dargestellten Filtervorgangs eingegangen wird, werden zunächst die beiden zentralen Schnittstellen zur Kommunikation, Publish und Subscribe, vorgestellt.

5.5.1 Abonnieren von Ereignisdaten per Subscription

Mit Hilfe der Subscribe-Schnittstelle kann das Interesse an bestimmten Ereignisdaten in Form von Abonnements (engl. subscriptions) ausgedrückt werden. Dafür stehen dem Ereignisfilter nach dem derzeitigen Konzept zwei Varianten der Schnittstelle zur Verfügung. Die Erfassung kann zum einen in Form von XML-Dokumenten geschehen und zum anderen in Form von fertigen Subscriptions als Java-Objekt. Erfolgt die Registrierung als Java-Objekt, so muss diese lediglich in die interne Struktur zur Speicherung von Subscriptions aufgenommen werden.

Der zweite Fall ist etwas aufwändiger, allerdings auch wesentlich wahrscheinlicher, da ein Client sein Interesse an Ereignissen meist in Form von XML-Dokumenten ausdrücken wird. Diese Subscription muss zwingend valide zum Schema der ADiWa-Subscriptionsprache sein, um vom Ereignisfilter erfasst werden zu können. Dies wird stets beim Einlesen der Subscriptions abgeprüft, um Fehler beim eigentlichen Parsen der Subscriptions zu vermeiden. Realisiert wird das Abbilden der Subscriptions auf die Modellklassen mit der internen Datenstruktur des Ereignisfilters durch die JAXB-Technologie, was im folgenden Abschnitt näher erläutert wird.

5.5.2 Abbildung von XML-Subscriptions auf interne Datenstruktur mit JAXB

Wie bereits in Abschnitt 5.2.2 beschrieben, liegt dem XML-Ereignisfilter eine sehr mächtige Sprache zur Definition von Subscriptions zugrunde. Diese Subscriptions können, wie auch die Ereignisdaten, in Form von XML-Nachrichten im System ankommen. Um die Subscriptions effizient verarbeiten zu können, werden diese auf eine Java-basierte Datenstruktur abgebildet. Dieser Vorgang wird XML-Binding genannt und führt zu Modellklassen, welche der gegebenen XML-Grammatik entsprechen[16]. Zwei Nachteile ergeben sich in diesem Zusammenhang, welche bei dem verfolgten Zweck zu vernachlässigen sind. Zum einen kann das Modell nur für diese eine Grammatik verwendet werden und zum anderen muss das Dokument vor einer Verarbeitung stets komplett eingelesen werden. Da es allerdings nur eine Grammatik im Konzept des XML-Ereignisfilters für das Formulieren von Subscriptions gibt, ist der erste Punkt von vorn herein irrelevant. Der zweite Aspekt lässt sich dadurch entkräften, dass das Anmelden von Subscriptions keinen zeitkritischen Vorgang darstellt. Jede Subscription kann erst dann aktiv werden, wenn diese vollständig eingelesen wurde. Somit würde auch eine strombasierte Verarbeitung, wie es zum Beispiel bei SAX der Fall wäre, keinen Vorteil ergeben, da erst das fertige Subscription-Objekt genutzt werden kann.

Zur Beschreibung der ADiWa-Subscriptionsprache existieren als XML-Grammatik die Schemadefinitionen `filterFormat.xsd` und `subscriptionFormat.xsd`[17], welche die Grundlage für die Generierung der Java-Modellklassen bilden. Das Binding ermöglicht es, aus bestehenden XML-Grammatiken (in diesem Fall die Subscriptionsprache), passende Dokumente in der Anwendung zu verarbeiten. Die Nutzung der voreingestellten Übersetzungsanweisungen führt zu einer Datenstruktur unter Verwendung von Standarddatentypen, ohne auf die speziellen Anforderungen für ein späteres Filtern einzugehen. Aus diesem Grund ist bereits an dieser Stelle eine Optimierung der Übersetzung möglich und auch erforderlich, um die aus den XML-Subscriptions entstandenen Java-Objekte direkt in den SubscriptionManager des Filters aufzunehmen. Diese Optimierung ist durch die Angabe von modifizierten Übersetzungsanweisungen (engl. Binding Declarations) möglich.

[16] Der Vorgang des XML-Binding wird anhand der JAXB-Technologie in Kapitel 2.3.2 beschrieben.
[17] Die kompletten Schemadefinitionen sind in Anhang 2 auf Seite 102 finden.

5.5.3 Anmelden von Subscriptions – Subscribe

Die modifizierten Übersetzungsanweisungen bekommt der Binding Compiler zur Übersetzung auf die in Abschnitt 5.3.2 beschriebene, intern verwendete Datenstruktur[18]. Abbildung 27 stellt nun die Datenstruktur einer, durch Binding Declarations optimierten, Subscription (oben) und die spätere Repräsentation als Sammlung von Subscriptions (unten) vergleichend dar. Neu hinzugekommene Objekte und Verknüpfungen sind dabei in Rot hervorgehoben. Pfeildarstellungen mit einem Rechteck am Ende repräsentieren jeweils eine Liste. Wenn Objekte als logische Einheit betrachtet werden können (z. B. eine Condition), so ist dies auch farblich entsprechend dargestellt.

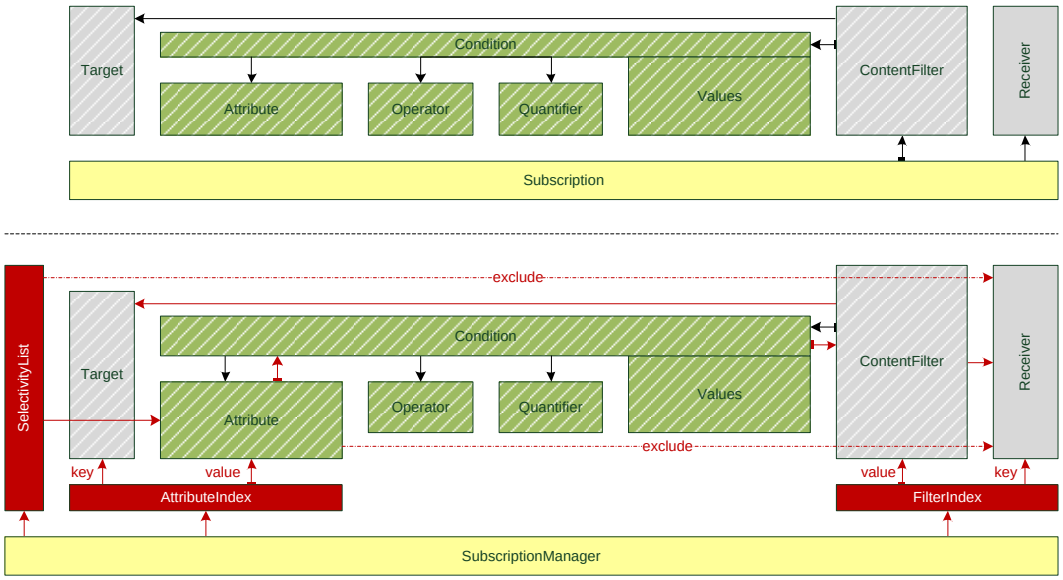


Abbildung 27: Strukturvergleich zwischen generiertem Subscriptionobjekt und Repräsentation im SubscriptionManager des XMLEventFilters (Quelle: eigene Darstellung)

Wie bei der Abbildung gut zu erkennen ist, entsprechen die Java-Objekte, welche durch das Unmarshalling entstehen, der gewünschten Struktur und können somit direkt in die Subscription-Datenstruktur übernommen werden. Die Aufgabe der Integration von Subscriptions in die interne Datenstruktur und deren Sicherung während der Laufzeit übernimmt der SubscriptionManager. Im Folgenden werden die Begriffe interne Datenstruktur und SubscriptionManager aus diesem Grund auch synonym verwendet. Bevor die Integration durch den SubscriptionManager stattfinden kann, werden noch kleinere Optimierungen durchgeführt, welche eine vereinfachte Handhabung des Filters ermöglichen. So kann es in Einzelfällen vorkommen, dass eine Subscription weder eine Empfängeradresse, noch eine Empfangsmethode aufweist. Dieser Fall tritt zum Beispiel dann auf, wenn ein Client sein eigenes Interesse an Informationen per TCP- oder HTTP-Verbindung am System anmeldet, aber durch technische Hindernisse seine IP-Adresse

[18] Eine Übersicht zu den Möglichkeiten, welche sich durch die Binding Declarations ergeben, liefert das Kapitel 7 der JAXB 2.1 Spezifikation [Sun06]. Die konkrete Umsetzung dieser Binding Declarations für das Parsen der Subscriptions findet sich als kommentierte Version in Anhang 3 auf Seite 104.

nicht ermitteln kann. Wird die beschriebene Situation erkannt, so bekommt die Subscription automatisch die IP-Adresse und die Verbindungsmethode zugewiesen, welche zum Anmelden der Subscription verwendet wurde. Die gefundenen Ereignisdaten werden somit direkt an den Absender der Subscription übermittelt. Ein ähnliches Vorgehen wird im Falle des Fehlens eines Quantors innerhalb einer Condition angewendet. Hierbei wird aus Sicherheitsgründen davon ausgegangen, dass stets alle Werte eines Ereignisattributs die Bedingung des Filters erfüllen müssen. Als Standardverhalten wird der Quantifier somit auf `all` gesetzt.

Die eigentliche Integration erfolgt nun durch den SubscriptionManager in wenigen Schritten.

(1) Da beim Hinzufügen einer neuen Subscription stets ein exklusiver Zugriff auf die Datenstruktur erforderlich ist, wird zunächst der Status des SubscriptionManager überprüft. Eine Veränderung der Datenstruktur, während gerade ein Ereignis gefiltert wird, würde zu einem undefinierten Verhalten führen und muss somit unterbunden werden. Sollte der Manager nicht bereit sein, so kommt die Subscription in eine Warteschlange und wird erst dann abgearbeitet, wenn beispielsweise das Ereignis fertig gefiltert wurde. Im anderen Fall wird die Datenstruktur für die weitere Verarbeitung von Ereignisdaten gesperrt und die Integration kann gestartet werden.

(2) Aus dem bereits vorhandenen FilterIndex wird die Menge an ContentFiltern (FilterContainer) herausgesucht, welche bereits dem Receiver der Subscription zugeordnet sind. Existiert bisher noch kein Wert für diesen Receiver im FilterIndex, so wird ein neuer FilterContainer für den Receiver der Subscription angelegt und dem Index hinzugefügt.
Für jeden ContentFilter der Subscription wird ein eindeutiger Hashwert ermittelt, damit dieser während der Filterung schnell identifiziert werden kann. Gleichzeitig wird ein Verweis auf den Receiver im ContentFilter erfasst. Anschließend wird der Filter dem in (2) ermittelten FilterContainer hinzugefügt.

(3) Mit dem Target der Subscription wird die Menge der zugeordneten Attribute aus dem SubscriptionManager geholt. Auch hier wird ein neues AttributeSet mit dem Target aus der Subscription erzeugt, wenn kein Eintrag gefunden wurde.
Für jede Condition aus diesem Filter wird die bereits beschriebene Optimierung bezüglich des Quantors vorgenommen. Anschließend wird das Attribut der aktuellen Condition dem vorher ermittelten AttributeSet hinzugefügt und diesem eine Referenz auf die Condition hinzugefügt. Die Condition hingegen bekommt eine zusätzliche Referenz auf den ContentFilter.
Für den schnellen Ausschluss von Receivern beziehungsweise ContentFiltern während der Filterung wird der Receiver als Hashwert dem Selectivity des Attributes hinzugefügt. Ebenso wird ein Paar aus Attribut und Receiver (jeweils in Form von Hashwerten) der globalen SelectivityTable des Subscription Manager hinzugefügt.

(4) Zum Abschluss wird der Status des SubscriptionManagers wieder auf `available` gesetzt, damit eine Filterung von Ereignisdaten erfolgen kann.

Zum besseren Verständnis werden die beschrieben Schritte in Abbildung 28 im Rahmen eines Programmablaufplans grafisch dargestellt. Als Schritt (0) wird in dieser Abbildung auch das Hinzufügen der Subscription zur Masterliste des XMLEventFilters dargestellt, was insbesondere für das Verständnis des Abmeldens von Subscriptions wichtig ist.

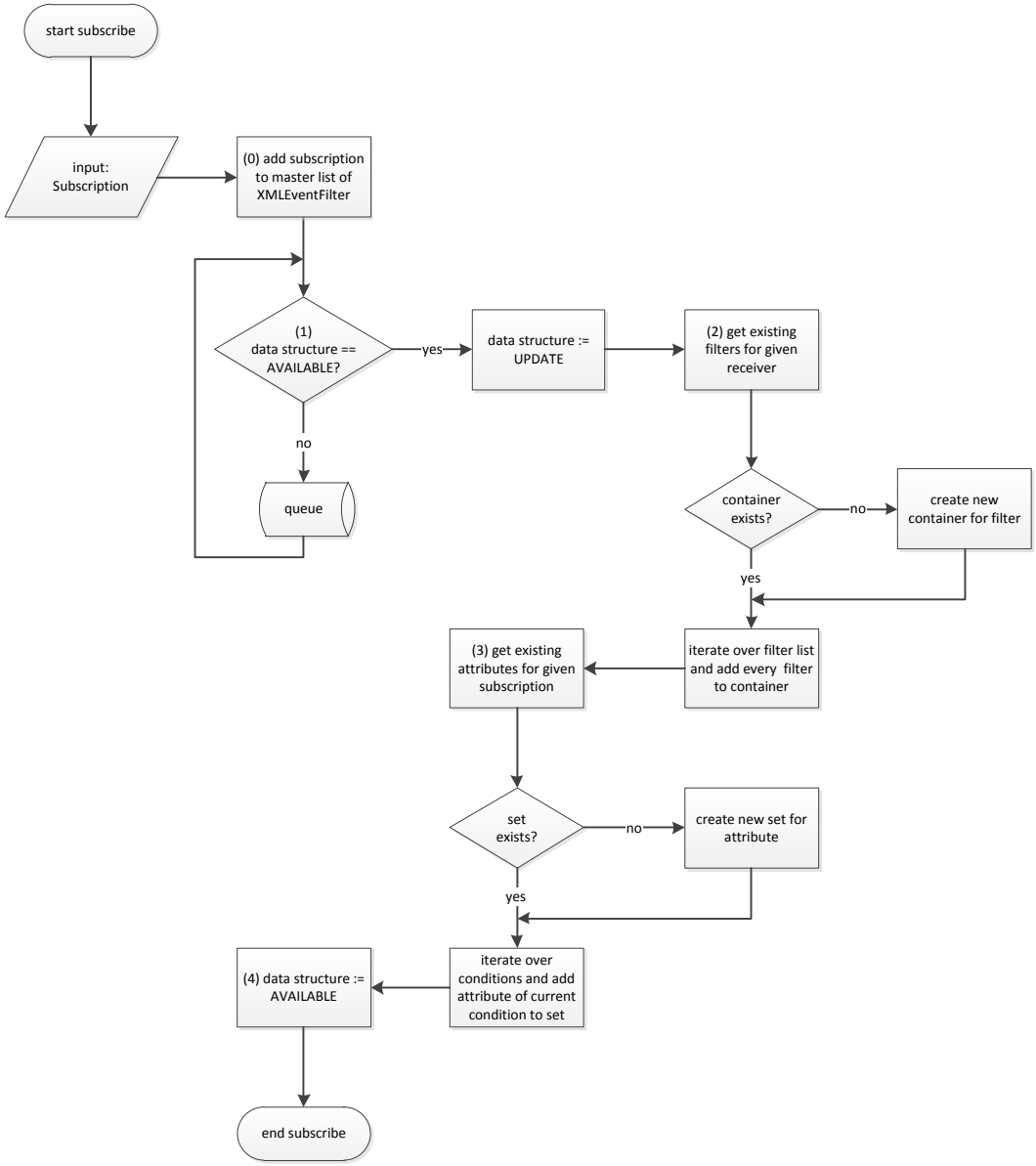


Abbildung 28: Programmablaufplan für das Anmelden einer Subscription (Quelle: eigene Darstellung)

5.5.4 Abmelden von Subscriptions – Unsubscribe

Beim Abmelden von Subscriptions sind mehrere Konzepte denkbar. Für die prototypische Umsetzung wurde aus Komplexitätsgründen eine vereinfachte, dadurch aber auch eher ineffiziente Variante gewählt. Dabei wird zunächst die entsprechende Subscription aus der Masterliste des XMLEventFilters mit den reinen Subscriptions entfernt. Anschließend wird der Subscription Manager neu instanziiert und somit eine

komplett neue Datenstruktur aus der Masterliste aufgebaut. Dieser Vorgang ist als Programmablaufplan in Abbildung 29 dargestellt.

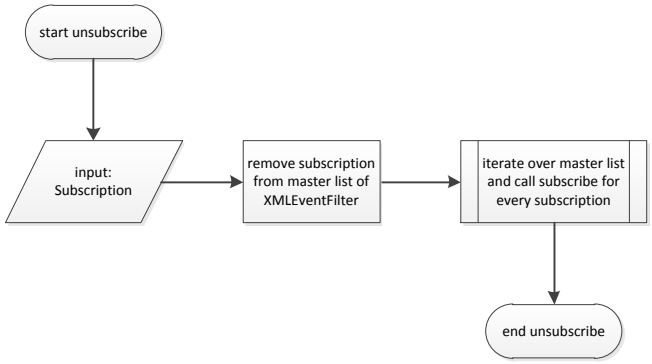


Abbildung 29: Programmablaufplan für das Abmelden einer Subscription (Quelle: eigene Darstellung)

Weniger zeit- und ressourcenaufwändig wäre das einzelne Herauslösen von Subscriptions direkt aus dem Subscription Manager. Das Generieren der kompletten Datenstruktur würde somit entfallen und ein Modifizieren von einzelnen Susbcriptions ist ebenfalls denkbar. Bei dieser Variante wird zunächst ebenfalls die Subscription aus der Masterliste entfernt. Anschließend wird der Subscription Manager aber nicht zerstört, sondern lediglich über die Änderung informiert, sodass dieser seine Datenstruktur aktualisieren kann.

Da sich jede Instanz des Ereignisfilters zu einem anderen Zeitpunkt in dem Status `available` befinden kann[19], erfolgt die Aktualisierung nicht synchronisiert. Dies führt dazu, dass auch die Subscription Manager der einzelnen Ereignisfilter-Instanzen nicht identisch hinsichtlich ihrer Datenstruktur sein müssen. Wenn diese mögliche Inkonsistenz der Subscription Manager in einem Brokernetzwerk beachtet wird, ergeben sich allerdings einige Vorteile. Eine aufwändige Synchronisierung, welche unter Umständen lange Wartezeiten nach sich zieht, entfällt. Bei einem synchronen Update würde das komplette Brokernetzwerk während dieser Zeit für eine Filterung nicht zur Verfügung stehen. Im Fall der unabhängigen Aktualisierung jedes einzelnen Subscription Managers sollte der Filterprozess im Idealfall ohne größere Verzögerungen fortgesetzt werden können. Mögliche Laufzeitunterschiede durch den Einsatz verschiedener Übertragungsprotokolle und Schnittstellen unterstützen diese Entscheidung zusätzlich.

5.5.5 Filterung von Ereignisdaten – Publish

Für den Empfang von Ereignisdaten sind derzeit zwei verschiedene Eingabevarianten der Publish-Schnittstelle vorgesehen. Die Übermittlung kann somit entweder im XML-Format (z.B. Ereignisse als Zeichenkette oder als Datenstrom basierend auf XML-Dokumenten) oder in Form von Java-Objekten (z.B. Ereignisse als Liste von Ereignisobjekten) erfolgen. Die Filterung ist in beiden Fällen identisch, der wesentliche Unterschied liegt in der zusätzlichen Komponente zum Parsen des XML-Ereignisstromes.

[19] Dies ist durch das Brokernetzwerk bedingt, was in Abschnitt 5.1.2 beschrieben wurde.

Der Parser ist dafür verantwortlich, aus dem Strom, welcher im Prinzip eine beliebig lange Zeichenkette darstellt, die gewünschten Ereignisse zu erkennen. Dabei wird bereits eine erste Stufe der Filterung durchlaufen. Handelt es sich in dem Strom um Ereignisse, welche einen unbekannten Typ aufweisen, so bleiben diese vom Parser unberücksichtigt, da keine Subscriptions für diese Art von Informationen im System existieren.

Da für die Veröffentlichung (engl. publishing) von Ereignisdaten nur eine Schnittstelle im Ereignisfilter existiert, müssen alle Eingabevarianten das Java-Interface `IEventData` implementieren. Somit können sowohl Eingabestrom als auch Java-Objekt als einheitlicher Datentyp auftreten.

5.5.6 Parsen von XML-basierten Ereignisdaten mit StAX

Im Gegensatz zu dem in Absatz 5.5.2 diskutierten Verfahren zur Abbildung von XML-basierten Subscriptions auf die interne Datenstruktur mit der JAXB-Technologie kommt für das Parsen der XML-basierten Ereignisdaten eine andere Variante zum Einsatz. Die Verwendung der Streaming API for XML (kurz StAX) bietet an dieser Stelle einige entscheidende Vorteile.[20] Das Parsen von XML-Dokumenten in eine DOM-Baumstruktur, wie es bei JAXB der Fall ist, kommt aufgrund der kontinuierlichen streaming-Eigenschaften der Ereignisdaten nicht in Frage. In diesem Fall müsste stets auf das Ende des Datenstroms gewartet werden, bevor der DOM-Baum aufgebaut und somit die Filterung gestartet werden kann. Der nächste klassische Ansatz, das SAX-Parsing, erweist sich aufgrund der Nachteile aus dem Push-Prinzip auf den zweiten Blick ebenfalls als weniger geeignet. Hierbei wird der Programmfluss durch den Parser über das Auslösen von Ereignissen gesteuert. Jeder Aufruf eines so genannten Callback-Handlers enthält lediglich die aktuellen Knoteninformationen ohne zusätzlichen Kontext über die Struktur des XML-Dokumentes. [Sch09]

Ein weiterer wichtiger Vorteil von StAX ist die Möglichkeit der Serialisierung von Objekten in ein XML-Dokument, was in Verbindung mit dem Versand von gefilterten Ereignisdaten benötigt wird. Die als XMLEventWriter bezeichnete Komponente übernimmt die Aufgabe der Serialisierung von Ereignissen und wird in Abschnitt 6.1.2 vorgestellt. [Sch09]

5.5.7 Filterung von XML-basierten Ereignisdaten

Die einzelnen Schritte des in Abbildung 26 dargestellten Filtervorgangs werden nun detaillierter beschrieben. Zunächst dient ein vereinfachter Programmablaufplan (Abbildung 30) der Visualisierung der einzelnen Schritte.

[20] Die Funktionsweise des Parsens mit der StAX-Technologie wird in Kapitel 2.3.3 beschrieben

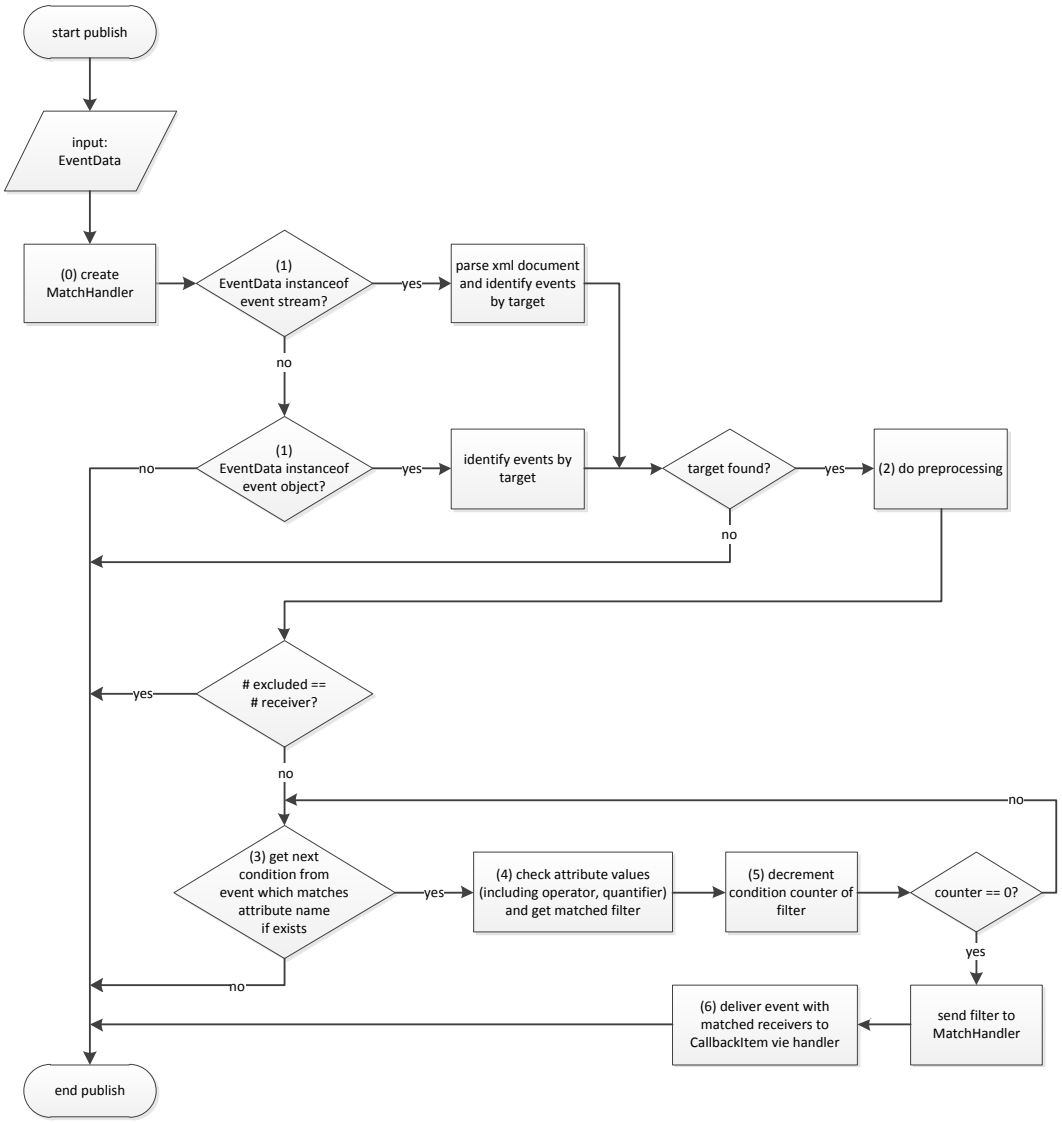


Abbildung 30: Programmablaufplan für das Filtern eines Ereignisses (Quelle: eigene Darstellung)

Schritt 0 Vorbereitung der Filterung

Dieser Schritt dient der Vorbereitung und Steuerung des Filtervorgangs. Zunächst wird ein MatchHandler erzeugt, welcher die komplette Filterung der Ereignisdaten begleitet und für die Speicherung der Resultate oder Zwischenergebnisse verantwortlich ist. Dazu enthält dieser Handler ein CallbackItem, was die Rückgabe von Ereignissen und deren Empfängern an den Broker ermöglicht.

Nach dieser Instanziierung erfolgt abhängig vom entsprechenden Datentyp der Ereignisdaten eine Übergabe des Kontrollflusses an Schritt 1 des Filtervorgangs. Nach der vollständigen Filterung kehrt der Kontrollfluss wieder zu diesem Schritt zurück und dem CallbackItem wird schließlich das Ende der Filterung signalisiert.

Folgender, als Auszug dargestellter, Algorithmus liegt Schritt 0 zugrunde:

➔ **`WorkingFilter.publish(IEventData, ICallbackItem)`**
- Konstruiere ein Matchhandler-Objekt zur Unterstützung der Filterung
- Falls Ereignisse als Instanz von EventStringReader oder EventStreamReader vorliegen
  - Übergebe Ereignisse an XMLEventReader zur Weiterverarbeitung (Schritt 1)
- FallsEreignisse als EventList vorliegen
  - Übergebe Ereignisse an TargetFilter zur Weiterverarbeitung (Schritt 1)

Schritt 1 Parsen der XML-Ereignisdaten und Ereignistypidentifizierung

Für den Fall, dass die Ereignisdaten in Form eines XML-Dokumentes in das System gelangen, übernimmt der `XmlEventReader`, wie in Abschnitt 2.3.3 beschrieben wurde, die Funktion des Parsers.

Bereits in der ersten Phase der Filterung erfolgt die Identifizierung des Ereignistyps[21]. Dazu wird während des Parsens bei jedem Start-Element und dessen Attributen sowie bei eingelesenem Knoteninhalt mit Hilfe der `checkTarget`-Methode geprüft, ob es sich um einen Ereignistyp handelt, für den eine Subscription existiert. Diese Prüfung erfolgt allerdings nur, wenn aktuell kein Ereignis eingelesen wird. Die Vorgehensweise ermöglicht die Verarbeitung von heterogenen Ereignisdaten, welche allein über die Subscriptions spezifiziert werden können.

Als Rückgabe liefert die `checkTarget`-Methode eine Zahl (Integer), welche folgende Werte haben kann[22]:

< 0 Der Parser hat einen bekannten Ereignistyp erfasst, das Ereignis startet allerdings in einer höheren Ebene des XML-Baumes.

0 Es wurde ein passender Ereignistyp gefunden und das Ereignis beginnt mit dem aktuell eingelesen Knoten.

> 0 Es gibt keinen passenden Ereignistyp. Allerdings existieren mögliche Typen, welche in tieferen Ebenen des XML-Baumes bei weiterem Parsen getroffen werden können.

null Für das aktuell eingelesene XML-Element existiert kein bekannter Ereignistyp und auch entlang dieses XML-Pfades kann kein Ereignistyp mehr getroffen werden.

[21] Der Ereignistyp wird in der zugrundeliegenden Subscriptionsprache und somit auch in der verwendeten Datenstruktur als Target bezeichnet. Weitere Informationen dazu und zur Konzeption einer Ereignistyphierarchie sind im Abschnitt 5.2 zum Thema Subscriptionsprache zu finden.

[22] Zum besseren Verständnis der möglichen Rückgabewerte ist in Anhang 5 auf Seite 106 eine Übersicht mit zahlreichen Beispielen zu finden.

Sobald diese Methode einen Wert kleiner oder gleich 0 zurückliefert, wird somit ein neues Java-Objekt erzeugt, welches alle folgenden durch den Parser erfassten Attribut-Wert-Paare aufnehmen kann. Sobald wieder die Ebene des Wurzelknotens erreicht wurde, ist das Ereignis vollständig und kann an die zweite Stufe des Ereignisfilters übergeben werden.

Sollten die Ereignisdaten nicht in Form eines XML-Dokumentes, sondern bereits als fertige Javaobjekte vorliegen, so gelangen diese nicht in den Parser sondern direkt in einen `TargetFilter`, welcher sich lediglich um die Identifizierung des Ereignistyps kümmert und das Objekt bei Bedarf zur weiteren Filterung an den zweiten Verarbeitungsschritt weiterreicht.

Zusammengefasst wird in diesem Schritt folgender Algorithmus ausgeführt:

➔ **`XmlEventReader.process(IEventData, MatchHandler)`**
- Parse das XML-Dokument (`IEventData`) mit StAX (Cursor API)
- Für jedes Element, welches im XML-Eingabestrom vorkommt:
  a) Falls Start-Element
     - Falls gerade kein `Event`-Objekt erstellt wird, dann prüfe, ob es sich um eine `Target`-Angabe handelt und erzeuge ggf. ein `Event`-Objekt
     - Für jedes Attribut, welches im Startelement vorkommt:
       → Falls gerade ein `Event`-Objekt erzeugt wird, füge aktuelles Attribut/Wert-Paar der Attributliste des `Event`-Objektes hinzu
       → Sonst prüfe, ob es sich um eine Target-Angabe handelt und erzeuge ggf. ein `Event`-Objekt
  b) Falls Character-Element
     - Falls gerade ein `Event`-Objekt erzeugt wird, füge aktuelles Attribut/Wert-Paar der Attributliste der `Event`-Objektes hinzu
     - Sonst prüfe, ob es sich um eine Target-Angabe handelt und erzeuge ggf. ein `Event`-Objekt
  c) Falls End-Element
     - Falls gerade ein `Event`-Objekt erzeugt wird, übergebe dieses dem Attributfilter zur Weiterverarbeitung (Schritt 2)

➔ **`TargetFilter.process(IEventData, MatchHandler)`**
- Überprüfe für jedes `Event`-Objekt aus der EventList:
  a) Falls die Target-Angabe mit einem Ereignistyp aus den registrierten Subscriptions übereinstimmt, delegiere dieses `Event`-Objekt an den Attributfilter zur Weiterverarbeitung (Schritt 2)

Schritt 2 Vorverarbeitung für schnellen Ausschluss von Ereignissen (optional)

Der zweite Schritt dient dazu, möglichst früh entscheiden zu können, ob ein Ereignis den Filter passieren kann oder nicht. Dazu wird eine bestimmte Anzahl an Selectivitys aus der geordneten Selectivity-Liste der Datenstruktur genutzt. Kommt der Attributname des Selectivity nicht in dem Ereignis vor, so werden die Positionen der entsprechenden Receiver in einem Bitvektor auf 1 gesetzt. Ist nach diesem Schritt die Anzahl der Einsen in dem Bitvektor gleich der Anzahl aller möglichen Receiver, so können bereits alle potentiellen Empfänger ausgeschlossen werden und das Ereignis muss nicht

weiterverarbeitet werden. Können in Schritt 2 noch nicht alle Receiver ausgeschlossen werden, so dient der Bitvektor in den folgenden Schritten als Prüfvektor, was zu einer schnelleren Entscheidung führt, falls das Ereignis keinen Empfänger in den Subscriptions findet.

Zusammenfassung von Schritt 2:

➔ **`AttributeFilter.preprocess(Event, BitVector)`**

- Nimm die ersten `<preprocessRounds>`-Elemente (Parameter ist in den Einstellungen frei wählbar) aus der `SelectivityList` und prüfe, ob Attribute in dem Ereignis vorkommen
  - Wenn nicht, dann schließe Receiver bei weiterer Verarbeitung aus → Merke die ausgeschlossenen Receiver in Bitvektor
- Wenn Anzahl ausgeschlossener Receiver gleich der Anzahl registrierter Receiver, dann verwerfe Ereignis
- Sonst übergebe Ereignis zur Weiterverarbeitung an Schritt 3

Schritt 3 Identifizierung von Conditions anhand der Attributnamen

In dieser Phase der Filterung wird nach Attributen gesucht, welche dem bereits bekannten Ereignisttyp (Target) zugeordnet sind. Falls mehr als ein Attribut in der Liste der registrierten Subscriptions gefunden wurde, so kann über einen XPath-Vergleich geprüft werden, ob es in dem Ereignis ein identisches Attribut gibt. Wenn dies der Fall ist, wird über das Attribut eine Liste mit möglichen Conditions ermittelt, welche zusammen mit dem Wert des Ereignisattributs an den nächsten Schritt übergeben werden.

Von besonderer Bedeutung ist dabei der XPath-Vergleich, welcher zuverlässig und möglichst effizient gestaltet werden muss. Im einfachsten Fall handelt es sich hier um einen Zeichenkettenvergleich. Mit diesem kann überprüft werden, ob die Position des Attributes im Ereignis, das während des Parsens in einen XPath-Ausdruck überführt wurde, durch die XPath-Angabe in der Subscription erfasst wird. Denkbar ist an dieser Stelle auch der Einsatz von speziellen Frameworks zur Verarbeitung von XPath-Ausdrücke, wobei dies häufig zu unerwünschtem Overhead führt [Alt00].

Auch in diesem Schritt erfolgt für jedes getestete Attribut aus der Subscription-Menge wieder ein Bitvektorvergleich. Somit wird anhand der Anzahl der Einsen in dem Bitvektor überprüft, ob bereits alle möglichen Empfänger ausgeschlossen werden können und somit keine weitere Verarbeitung des Ereignisses erfolgen muss.

Zusammenfassung von Schritt 3:

➔ **`AttributeFilter.conditionMatch(Event, MatchHandler)`**

- Ermittle mit der Target-Angabe aus dem Event den Ereignistyp
- Hole die zugehörige Attributliste des Ereignistyps aus dem `AttributeIndex` der registrierten Subscriptions
- Iteriere über die Attributliste und führe für jedes Attribut folgende Aktionen durch:

- Suche nach einem Attribut mit gleichem Namen (XPath) im Ereignis
- Falls Attribut gefunden:
  - Ermittle Conditions anhand des Attributs, Wert des Attributs aus dem Ereignis
  - Delegiere Filterung an Schritt 4
- Sonst:
  - Führe eine Oder-Verknüpfung des Bitvektors aus dem Matchhandler (kennzeichnet die bereits ausgeschlossenen Receiver) mit dem Bitvektor des aktuellen Attributs durch
  - Prüfe, ob bereits alle möglichen Receiver ausgeschlossen werden können
    → wenn Anzahl ausgeschlossener Receiver gleich der Anzahl verfügbarer Receiver, dann verwerfe Ereignis

Schritt 4 Überprüfung der Attributwerte inklusive Operatoren und Quantoren

In Schritt 4 erfolgt das eigentliche Matching der Attributwerte. Den Einstieg dazu bildet der Operator, welcher der Condition zugeordnet ist. Jeder Operator besitzt eine `match`-Methode, welche jeweils ein anderes Verhalten aufweisen kann. In jedem Fall benötigt diese Methode 3 Parameter, Wert des Conditionattributs, Wert des Ereignisattributs und einen Quantifier. Der Rückgabewert ist stets `true` oder `false`, in Abhängigkeit von der Erfüllung der Condition durch den Wert des Ereignisattributs.

Beispiele für das Verhalten von Operatoren:

- `string-equal`:
  → Falls Quantifier = `all`: liefert `true` zurück, wenn der Wert des Ereignisattributs identisch zu den Werten aller Conditionattribute ist, sonst `false`.
  → Falls Quantifier = `any`: liefert `true` zurück, wenn der Wert des Ereignisattributes indentisch mit mindestens einem der Werte der Conditionattribute ist, sonst `false`.
- `attribute-exists`:
  liefert `true` für jeden Quantifier, da Operator bereits erfüllt ist, wenn das Attribut vorhanden ist, d.h. sonst würde die `match`-Methode gar nicht aufgerufen werden.

Weitere Operatoren ergeben sich neben den Möglichkeiten der Subscriptionsprache auch aus den Anforderungen durch die heterogenen Ereignistypen. Deshalb ist es erforderlich, die bereits im System bekannten Operatoren um weitere Operatoren zur Laufzeit zu ergänzen. Dies kann durch so genannte OSGi-Fragments[23] realisiert werden, welche einzelne Operatoren repräsentieren und zwischen den Brokern bei Bedarf ausgetauscht werden können.

[23] Eine Erläuterung zu OSGi-Fragments ist im Grundlagenkapitel unter 2.3.1 zu finden.

Zusammenfassung von Schritt 4:

➔ **`AttributeFilter.valueMatch(ConditionContainer, String, MatchHandler)`**
- Prüfe für jede Condition, ob der Attributwert den Operator, Quantifier und Wert der Condition erfüllt
- Wenn dies der Fall ist, ermittle die Filter aus der Condition und übergebe Filterung an Schritt 5

Schritt 5 Zählen der positiven Conditions zur Erfassung getroffener Filter

Im vorletzten Schritt der Filterung erfolgt die Erfassung getroffener Filter. Hintergrund ist hier die Tatsache, dass ein Filter erst dann als erfüllt gilt, wenn alle zugeordneten Conditions erfüllt sind (Konjunktion der Conditions eines Filters). Demzufolge besitzt jeder Matchhandler die Anzahl der bereits erfüllten Conditions für jeden Contentfilter. Dieser Zähler wird somit bei jedem Treffer um 1 dekrementiert. Sobald ein Filter die Anzahl 0 erreicht hat, wird dieser als getroffen gewertet und der zugehörige Receiver kann dem Ereignis hinzugefügt werden. Dies entspricht der Vorgehensweise des bereits beschriebenen Counting-Algorithmus aus Abschnitt 4.1.3 des Carzaniga-Ansatzes.

Zusammenfassung von Schritt 5:

➔ **`AttributeFilter.filterMatch(FilterContainer, MatchHandler)`**
- Iteriere über erhaltene ContentFilter
- Dekrementiere jeweils den Condition-Zähler der Filtervektors aus dem Matchhandler um 1 oder füge einen neuen Eintrag mit entsprechendem Zählerwert dem Filtervektor hinzu
- Wenn Zähler = 0, füge den Receiver des Filters der Receiverliste des Matchhandlers hinzu

Schritt 6 Serialisierung und Beendigung der Ereignisfilterung

Im sechsten und letzten Schritt wird der eigentliche Filtervorgang abgeschlossen. Dazu wird ein Objekt erzeugt, welches neben dem Ereignis auch eine Liste mit gefunden Receivern enthält. Dieses Ausgabeobjekt kann anschließend an die aufrufende Instanz, welche durch das CallbackItem festgelegt ist (in der Regel ist dies der Broker selbst), übergeben werden.

Bei Bedarf kann aus dem Ereignisobjekt an dieser Stelle mit Hilfe des StAX Serializers ein XML Dokument erzeugt werden. Hierbei wird aus dem gefilterten Ereignisobjekt wieder ein XML-Dokument generiert, welches alle durch die Subscription festgelegten Elemente und Attribute enthält.

Zusammenfassung von Schritt 6:

➔ **`AttributeFilter.filterAttributes(IEventData, MatchHandler)`**

- Erzeuge Ausgabeobjekt für Ereignis und gefundene Receiver
- Erzeuge bei Bedarf aus dem Ereignisobjekt ein XML-Dokument
- Übergebe Ergebnis-Objekt an den Broker

5.6 Filtervorgang am Beispiel

Um die Funktionsweise zu verdeutlichen, dient folgendes kurzes Beispiel. Aus einem XML-Dokument mit drei verschiedenen Subscriptions, welches per HTTP an die Subscribe-Schnittstelle des XML-Ereignisfilters gesendet wird, ergibt sich die in Abbildung 31 dargestellte Datenstruktur. In dieser Datenstruktur sind ebenfalls die einzelnen Schritte des Filtervorgangs gekennzeichnet.

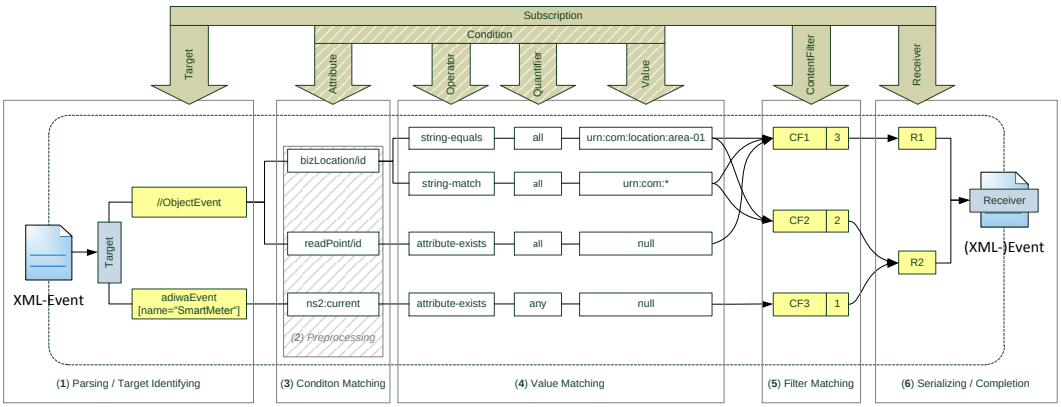


Abbildung 31: Beispiel zur Funktionsweise des Ereignisfilters (Quelle: eigene Darstellung)

```
...
<subscription>
  <receiver>
    <callbackMethod>HTTP</callbackMethod>
    <callbackAddress>http://localhost:4001/submit</callbackAddress>
  </receiver>
  <filterList>
    <contentFilter target="//ObjectEvent">
      <condition operatorId="string-equals" attribute="bizLocation/id">
        <value>urn:com:location:area-01</value>
      </condition>
      <condition operatorId="attribute-exists" attribute="readPoint/id"/>
      <condition operatorId="string-match" attribute="bizLocation/id">
        <value>urn:com:*</value>
      </condition>
    </contentFilter>
  </filterList>
</subscription>
...
```

Listing 8: Beispiel einer Subscription als Auszug aus einem XML-Dokument[24]

[24] Das vollständige XML-Dokument mit drei Subscriptions findet sich in Anhang 6 auf Seite 107.

Die Abbildung der Subscriptions erfolgt gemäß der in Abschnitt 5.5.2 beschriebenen Vorgehensweise. Als Auszug ist eine dieser Subscriptions mit einem Content-Filter und drei Conditions in Listing 8 dargestellt. Der Empfänger ist hier ebenfalls per HTTP zu erreichen und wünscht die Ereignisdaten an die Adresse „`http://localhost:4001/submit`“.

Treffen nun Ereignisdaten in Form eines XML-Dokumentes am Filter ein (dies kann wieder über einen vorgeschalteten HTTP-Server geschehen), so erfolgt der Filtervorgang anhand der oben genannten Schritte.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<epcis:EPCISDocument xmlns:epcis="urn:epcglobal:epcis:xsd:1">
  <EPCISBody>
    <EventList>
      <ObjectEvent>
        <eventTime>2010-04-12T14:05:05Z</eventTime>
        <eventTimeZoneOffset>+02:00</eventTimeZoneOffset>
        <epcList>
          <epc>urn:epc:id:sgtin:1000001</epc>
        </epcList>
        <action>OBSERVE</action>
        <bizStep>urn:com:bizstep:receiving</bizStep>
        <disposition>urn:com:disposition:active</disposition>
        <readPoint>
          <id>urn:com:readpoint:entrance-01</id>
        </readPoint>
        <bizLocation>
          <id>urn:com:location:area-01</id>
        </bizLocation>
      </ObjectEvent>
...
```

Listing 9: Auszug aus einer Publish-Nachricht mit Beispiel eines Ereignisses[25]

Listing 9 zeigt als Fragment einer Publish-Nachricht ein Ereignis, welches den Filter durchläuft und somit möglicherweise auf einen oder mehrere Empfänger abgebildet wird. Dazu folgt nun eine kurze Beschreibung der wichtigsten Vorgänge während der Filterung mit dem Fokus auf den Schritten 3 bis 5. Die Schritte 0 bis 2 werden übersprungen und als bereits abgearbeitet vorausgesetzt:

- ein MatchHandler zur Aufnahme von Empfängern existiert
- das Ereignis wurde beim Parsen des XML-Dokuments aus einem Datenstrom bereits identifiziert und liegt als Event-Objekt vor
- die Vorverarbeitung wurde bereits abgeschlossen, wobei nicht alle Empfänger ausgeschlossen werden konnten.

Schritt 3 conditionMatch

(1) nimm Target T aus Event-Objekt

T	//ObjectEvent

(2) ermittle Liste mit Attributen A aus der Subscription-Datenstruktur, welche zu T passen

A	bizLocation/id	readPoint/id

[25] Das vollständige XML-Dokument mit drei Ereignissen findet sich in Anhang 7 auf Seite 108.

(3) Iteriere über die Liste A

(4) Für jedes Attribut iteriere über die Attributliste B des Ereignisses und prüfe, ob die Attributnamen übereinstimmen

B	...	readPoint/id	bizLocation/id

(5) Wenn in B ein gleiches Attribut gefunden wurde (A-Attribut == B-Attribut), dann prüfe Wert des Attributes aus B gegen Conditions des Attributes aus A
➔ delegiere an Schritt 4

Schritt 4 valueMatch

(1) nimm die Ergebnisse von Schritt 3, d. h. die Werte V1.1, V2.1 von den gefundenen Attributen aus dem Event-Objekt und die Conditions C1.1, C1.2, C2.1, die den Attributen in der Datenstruktur zugeordnet sind

C1.1	bizLocation/id(string-equals, all, urn:com:location:area-01)
C1.2	bizLocation/id(string-match, all, urn:com:*)
C2.1	readPoint/id(attribute-exists, all, null)
V2.1	readPoint/id(urn:com:readpoint:entrance-01)
V1.1	bizLocation/id(urn:com:location:area-01)

(2) Prüfe jeweils, ob der Attributwert V1.1, V2.1 des Events den Wert der Conditions C1.1, C1.2, C2.1 in Abhängigkeit von Operator und Quantor erfüllt

(3) Wenn ja, ermittle die Filterliste der jeweiligen Condition und wende den Counting-Algorithmus an
➔ delegiere an Schritt 5

Schritt 5 filterMatch

(1) nimm die Ergebnisse von Schritt 4, d. h. die Filter F, welche den erfüllten Conditions zuzuordnen sind

F	ContentFilter1	ContentFilter2	ContentFilter1

(2) Iteriere über die ContentFilter-Liste F

(3) Falls der ContentFilter bereits im MatchHandler enthalten ist
→ verringere Conditionzähler um 1
→ sonst füge ContentFilter mit (Conditionanzahl - 1) dem MatchHandler hinzu

(4) Falls Anzahl Conditions == 0, füge den Receiver des Filters der MatchedReceivers-Liste im MatchHandler hinzu

➔ In diesem Beispiel werden die Conditionzähler allerdings nicht null (Zähler CF1 = 1 und Zähler CF2 = 1), somit erfolgt auch keine Weiterverarbeitung des Ereignisses in Schritt 6. Das Serialisieren und Weiterleiten des Ereignisses entfällt.

6 VALIDIERUNG

In diesem Kapitel steht die Validierung der eigenen Konzepte aus dem vorangegangenen Kapitel im Fokus. Dazu wird im ersten Unterkapitel ein Testszenario vorgestellt, welches die Implementierung eines lauffähigen Prototyps beschreibt und somit einen ersten Nachweis der Praxistauglichkeit der entworfenen Konzepte liefert. Gleichzeitig bildet dieses Szenario die technische Grundlage für die Performanceanalyse, welche in Unterkapitel 6.3 durchgeführt wird. Im Anschluss an die Vorstellung der Testumgebung wird ein Integrationsszenario skizziert, welches im Rahmen des Forschungsprojekts ADiWa die praktische Nutzbarkeit von XML-basierten Ereignisfiltern demonstriert.

6.1 Testumgebung für die Komponente XMLEventFilter

Mit der Entwicklung der Testumgebung wurde eine Minimalkonfiguration geschaffen, um den autarken Betrieb der Ereignisfilterkomponente demonstrieren zu können. Diese Konfiguration dient gleichzeitig zur Evaluation der Anforderungen aus Kapitel 3, was im Rahmen einer Performanceanalyse durchgeführt wird. Die Komponente `XMLEventFilter` kann durch die Anbindung verschiedener Kommunikationsschnittstellen nun ohne die Integration in einen Broker (wie er im Rahmen der Konzeption beschrieben wurde) beziehungsweise ein Brokernetzwerk und somit vollkommen selbstständig genutzt werden[26].

Zur besseren Unterscheidung zwischen dem bereits bekannten Broker und der Filterkomponente mit seinen essentiellen Kommunikationsschnittstellen wird nun der Begriff Event-Broker benutzt. Damit im Rahmen des Event-Brokers die autarke Funktion des Ereignisfilters demonstriert werden kann, sind ein paar zusätzliche Komponenten erforderlich, was grafisch in Abbildung 32 dargestellt ist. Ähnlich wie die in Kapitel 5.1.1 beschriebene Architektur des Brokers lässt sich auch diese Publish/Subscribe-Realisierung in mehrere Clients und einen Server aufteilen. Die Clients übernehmen Aufgaben als Subscriber, Publisher und Receiver und sind über ausgewählte Adapter am System angebunden. Diese Adapter sind flexibel austausch- und erweiterbar, was eine problemfreie Anpassung an neue Bedürfnisse ermöglichen soll. Der Server als Event-Broker übernimmt hier die Aufgabe des Brokers, allerdings mit geringerem Funktionsumfang als in Abschnitt 5.1.1 beschrieben. So ist es beispielsweise nicht möglich, mit dieser Konfiguration ein Netzwerk mit mehreren Brokern aufzubauen. Diese Umsetzung dient somit als Referenzimplementierung für fortführende Arbeiten auf diesem Gebiet.

[26] Hinweise zur Installation und Konfiguration sind in Anhang 8 auf Seite 109 zu finden.

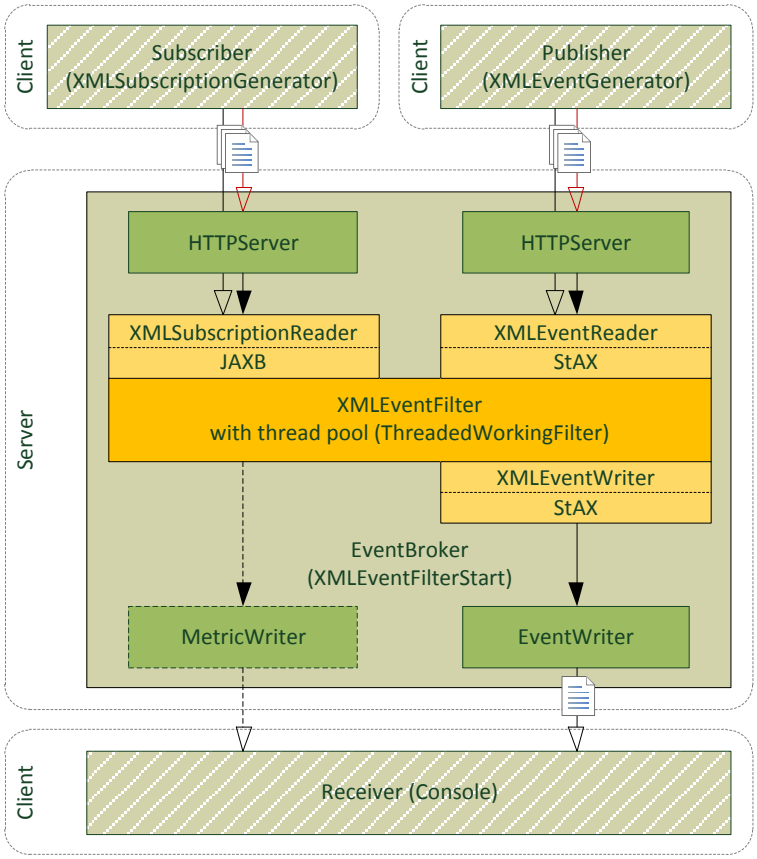


Abbildung 32: Architekturschaubild der Testumgebung für den XMLEventFilter (Quelle: eigene Darstellung)

Eine weitere, bereits sehr praxisnahe Implementierung wird mit dem Integrationsszenario aus dem Bereich Energy Management in Kapitel 6.2 beschrieben. Im Integrationsszenario kommt der Ereignisfilter allerdings nicht mehr autark, sondern integriert in einen Broker und somit als Bestandteil eines Brokernetzwerks zum Einsatz. Einige der für diese Testumgebung entwickelten Komponenten finden allerdings auch im Rahmen des Integrationsszenarios Verwendung.

6.1.1 Funktionsumfang der Testumgebung

Mit der Konfiguration der Testumgebung lassen sich zentralisierte Anwendungsszenarien mit nur einem Broker realisieren. Dabei lassen sich Subscriptions und Ereignisdaten im XML-Format generieren und einlesen. Diese können sowohl per HTTP-Nachricht als auch als Java-interner Aufruf an den Broker gesendet werden. Nach der Verarbeitung werden die Ereignisdaten wieder serialisiert und auf der Konsole ausgegeben. Zur Visualisierung der Interaktion zwischen den Komponenten dient die folgende Abbildung 33.

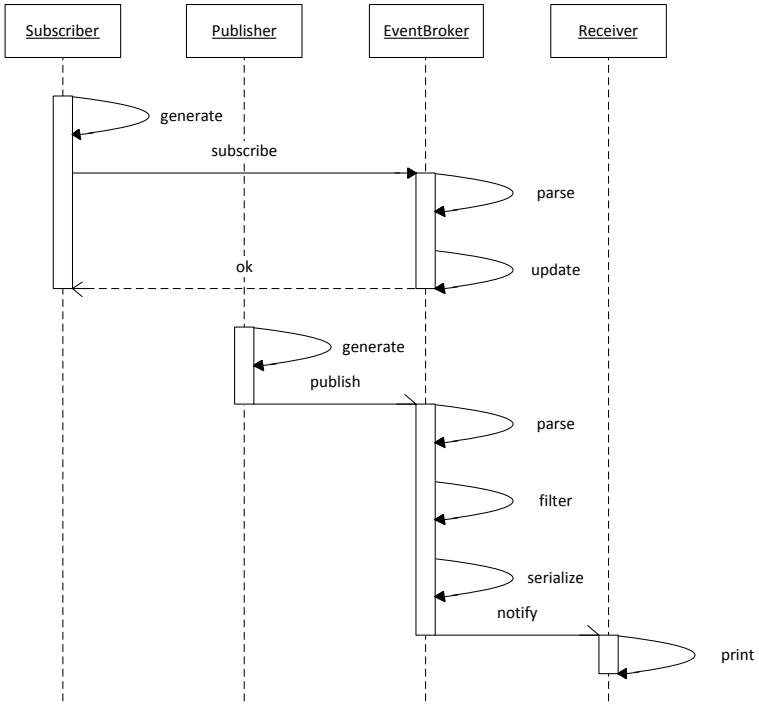


Abbildung 33: Sequenzdiagramm zur Visualisierung der Kommunikation zwischen den Komponenten im Rahmen des Testszenarios (Quelle: eigene Darstellung)

6.1.2 Komponenten zum Aufbau eines Publish/Subscribe-Systems

Die im Folgenden vorgestellten, eigenständigen Komponenten entsprechen den in Abbildung 32 als Client und Server dargestellten Komponenten zum Aufbau eines Publish/Subscribe-Systems. Die Verwendung des in Kapitel 5 konzipierten Ereignisfilters bildet dabei einen zentralen Bestandteil. Weitere abgebildete Komponenten werden zum Teil gemeinsam genutzt und in den zugehörigen Hauptkomponenten erläutert.

`XMLSubscriptionGenerator`

Der `XMLSubscriptionGenerator` dient als Client zum Senden von Subscription im XML-Format an den EventBroker. Diese XML-Dokumente können entweder in beliebiger Anzahl generiert oder aus einer XML-Datei eingelesen werden. Bei der Generierung dienen bestimmte Attribute und Werte aus dem EPCIS-Umfeld als Grundlage. Diese werden mit Zufallsdaten angereichert, um keine Duplikate zu senden. Per Aufrufparameter können Eigenschaften, wie Quelle, Ziel oder Anzahl festgelegt werden. Die Komponente `XMLSubscriptionGenerator` aus dem gleichnamigen Projekt ist als OSGi-Bundle realisiert worden und lässt sich dadurch sowohl separat als eigenständiger Client oder als Plug-In des EventBrokers verwenden.

- externer Client
  Mit Hilfe des externen Clients kann das Anmelden von Subscriptions per HTTP-Nachricht (z.B. von einem externen Rechner im lokalen Netzwerk) simuliert

werden. Dazu wird die Komponente als separate Anwendung genutzt. Beim Starten des ausführbaren Java-Archivs `SubscriptionGenerator.jar` sind bis zu 4 optionale Aufrufparameter zulässig, wobei die angegebene Reihenfolge eingehalten werden muss. Werden keine Parameter beim Aufruf angegeben, dienen Standardwerte als Initialisierung der Komponente.

(1) Quelle (`source`) ➔ zulässige Werte sind hier `GENERATE` und `FILE`:

Dieser Parameter spezifiziert den Einlesemodus von Subscriptions. Der Wert `GENERATE` signalisiert die automatische Generierung von Subscriptions, welche valide bezüglich der XML-Grammatik für die ADiWa-Subscriptionsprache sind.

Über den Wert `FILE` ist es möglich, Subscriptions aus einer XML-Datei einzulesen. In diesem Modus muss der Anwender selbst die Validität bezüglich des ADiWa-Subscriptionschemas gewährleisten.

(2) Quellenparameter (`sourceParam`):

Handelt es sich bei der Quelle um `GENERATE`, so dient dieser Parameter zur Angabe der Anzahl der zu generierenden Subscriptions. Dabei werden mindestens 3 Subscriptions generiert, die bei jedem Aufruf identisch sind. Die restlichen `(sourceParam – 3)` Subscriptions werden mit Zufallszahlen angereichert und sind in der Regel verschieden, da die Zufallszahlen über die Systemzeit ermittelt werden. Bei jeder Ausführung des Clients im `GENERATE`-Modus sollte somit eine Fehlermeldung den Anmeldeversuch von 3 bereits vorhandenen Subscriptions signalisieren.

Im Modus `FILE` können die Subscriptions direkt als XML-Datei eingelesen werden. Der Parameter `sourceParam` dient dann als Angabe eines Pfades (als URL) zu dieser Datei, wobei stets vom Verzeichnis ausgegangen wird, in dem sich das Archiv des Clients befindet.

(3) Ziel (`target`) ➔ zulässige Werte sind hier `HTTP`und `FILE`:

Dieser Parameter spezifiziert den Ausgabemodus der Komponente `XMLSubscriptionGenerator`. Wird als Ziel der Wert `HTTP` angegeben, so erfolgt die Auslieferung der Subscriptions in Form einer HTTP-Nachricht mit Hilfe der eingebetteten HTTP-Server Komponente.

Im Ausgabemodus `FILE` ist es möglich, die automatisch generierten Subscriptions in Form einer XML-Datei auf einem Datenträger zu speichern.

(4) Zielparameter (`targetParam`):

Im Ausgabemodus `HTTP` dient dieser Parameter der Angabe einer Zieladresse inklusive Port (als URL), wobei sowohl lokale als auch entfernte Adressen zulässig sind. Ein Beispiel für diesen Parameter wäre der Wert `http://localhost:8002`, wenn der HTTP-Server des Ereignisfilters auf dem gleichen Rechner für lokale Nachrichten unter dem Port `8002` erreichbar ist.

Im Modus `FILE` dient `targetParam` ähnlich wie der Quellparameter zur Angabe eines Pfades (als URL), wo die Datei mit den Subscriptions gespeichert werden soll. Auch hier wird von dem Verzeichnis ausgegangen, in dem sich das Archiv des Clients `XMLSubscriptionGenerator` befindet.

- interner Client
  Die Nutzung als interner Client ist ausschließlich durch die Einbindung als OSGi-Bundle zulässig. Dafür holt sich der Client eine Instanz der Hilfsklasse `XMLEventFilterStart`, die eine Anmeldung der Subscriptions über die entsprechenden Java-Schnittstellen des Ereignisfilters veranlasst. Das Verschicken der XML-Nachrichten über den HTTP-Server und der damit verbundene Overhead entfallen durch diese Anbindung. Diese Variante ist somit zwar wenig praxisrelevant, aber für Performancemessungen durchaus sinnvoll.
  Die Parameter für die Quellenangabe sind mit dem Modus `FILE` und der Pfadangabe `xml/test/adiwa_subscription_0012.xml` bereits voreingestellt, die Angabe eines Ziels entfällt.

`XMLEventGenerator`

Mit der Clientkomponente `XMLEventGenerator` können Ereignisdaten in Form von XML-Nachrichten an den `EventBroker` gesendet werden. Der `XMLEventGenerator` ist ähnlich strukturiert wie der bereits beschriebene `XMLSubscriptiongenerator` und ermöglicht somit wahlweise die Generierung von Ereignissen in beliebiger Anzahl oder das Einlesen von Ereignisdaten aus einer XML-Datei. Bei der Generierung dienen bestimmte Attribute und Werte aus dem EPCIS-Umfeld als Grundlage. Diese werden mit Zufallsdaten angereichert, um keine Duplikate zu senden. Per Aufrufparameter können Eigenschaften, wie Quelle, Ziel, Anzahl oder Art der HTTP-Übertragung festgelegt werden. Die Komponente `XMLEventGenerator` aus dem gleichnamigen Projekt ist als OSGi-Bundle realisiert worden und lässt sich dadurch sowohl separat als eigenständiger Client oder als Plug-In des EventBrokers verwenden.

- externer Client
  Mit Hilfe des externen Clients kann das Senden von Ereignisdaten per HTTP-Nachricht (z.B. von einem externen Rechner im lokalen Netzwerk) simuliert werden. Dazu wird die Komponente als separate Anwendung genutzt. Beim Starten des ausführbaren Java-Archivs `EventGenerator.jar` sind bis zu 5 optionale Aufrufparameter zulässig, wobei die angegebene Reihenfolge eingehalten werden muss. Werden keine Parameter beim Aufruf angegeben, dienen Standardwerte als Initialisierung der Komponente.
  (1) Quelle (`source`) ➔ zulässige Werte sind hier `GENERATE` und `FILE`:
  Dieser Parameter spezifiziert den Einlesemodus von Ereignissen. Der Wert `GENERATE` signalisiert eine automatische Generierung von Ereignissen, die von den gegebenenfalls automatisch generierten Subscriptions erfasst werden. Über den Wert `FILE` können Ereignisdaten aus einer XML-Datei eingelesen werden.
  (2) Quellenparameter (`sourceParam`):
  Handelt es sich bei der Quelle um `GENERATE`, so dient dieser Parameter zur Angabe der Anzahl der zu generierenden Ereignisse. Dabei werden mindestens 14 Ereignisse generiert, die bei jedem Aufruf identisch sind. Die restlichen

(sourceParam - 14) Ereignisse werden an ausgewählten Attributwerten mit fortlaufenden Nummern angereichert und sind deshalb innerhalb einer Nachricht stets verschieden. Ist sourceParam größer als 10.000, so wird die Nachricht automatisch in mehrere Teile gesplittet, um einen Speicherüberlauf durch zu große Zeichenketten zu vermeiden.

Im Modus FILE können die Ereignisse direkt als XML-Datei eingelesen werden. Der Parameter sourceParam dient dann als Angabe eines Pfades (als URL) zu dieser Datei, wobei stets vom Verzeichnis ausgegangen wird, in dem sich das Archiv des Clients befindet.

(3) Ziel (target) ➔ zulässige Werte sind hier HTTPund FILE:

Dieser Parameter spezifiziert den Ausgabemodus der Komponente XMLEventGenerator. Wird als Ziel der Wert HTTP angegeben, so erfolgt die Veröffentlichung der Ereignisse in Form einer HTTP-Nachricht mit Hilfe der eingebetteten HTTP-Server Komponente.

Im Ausgabemodus FILE ist es möglich, die automatisch generierten Ereignisse in Form einer XML-Datei auf einem Datenträger zu speichern.

(4) Zielparameter (targetParam):

Im Ausgabemodus HTTP dient dieser Parameter der Angabe einer Zieladresse inklusive Port (als URL), wobei sowohl lokale als auch entfernte Adressen zulässig sind. Ein Beispiel für diesen Parameter wäre der Wert http://localhost:8001, wenn der HTTP-Server des Ereignisfilters auf dem gleichen Rechner für lokale Nachrichten unter dem Port 8001 erreichbar ist.

Im Modus FILE dient targetParam ähnlich wie der Quellparameter zur Angabe eines Pfades (als URL), wo die Datei mit den Ereignisse gespeichert werden soll. Auch hier wird von dem Verzeichnis ausgegangen, in dem sich das Archiv des Clients XMLEventGenerator befindet.

(5) Senderanzahl (generatorNumber):

Mit Hilfe des Parameters generatorNumber kann angegeben werden, wie viele identische Clients möglichst gleichzeitig gestartet werden sollen. Die parallele Ausführung wird über einen Java-internen ExecutorService realisiert, welcher selbstständig die Verwaltung der einzelnen Threads organisiert.

- interner Client

  Die Nutzung als interner Client ist ausschließlich durch die Einbindung als OSGi-Bundle zulässig. Dafür holt sich der Client eine Instanz der Hilfsklasse XMLEventFilterStart, die eine Veröffentlichung der Ereignisse über die entsprechenden Java-Schnittstellen des Ereignisfilters veranlasst. Das Verschicken der XML-Nachrichten über den HTTP-Server und der damit verbundene Overhead entfallen durch diese Anbindung. Diese Variante ist somit zwar wenig praxisrelevant, aber für Performancemessungen durchaus sinnvoll.

  Die Parameter für die Quellenangabe sind mit dem Modus GENERATE und der Pfadangabe xml/test/adiwaEvent_0001.xml bereits voreingestellt, die Angabe eines Ziels entfällt.

XMLEventFilterStart

Zentrale Komponente in diesem Publish/Subscribe-System ist der `EventBroker` in Form der Komponente `XMLEventFilterStart`. Dieser `EventBroker` wird als OSGi-Anwendung mit Hilfe der Eclipse Entwicklungsumgebung gestartet.

Herzstück des EventBrokers ist die, um einen Thread Pool für WorkingFilter erweiterte, Komponente XMLEventFilter. Der Thread Pool mit dem Namen ThreadedWorkingFilter ermöglicht die parallele Verarbeitung von Ereignisdaten mit einer, per Konfigurationsdatei einstellbaren Anzahl an WorkingFilter-Instanzen [27]. Da jeder WorkingFilter seine eigene Datenstruktur besitzt, kann die Verarbeitung von Ereignisdaten vollkommen unabhängig voneinander und somit ohne gegenseitiges Warten ablaufen. Wird der WorkingFilter nicht im Rahmen dieser Testumgebung eingesetzt, ist zu beachten, dass dieser ohne geeigneten Thread Pool nicht threadsicher arbeitet. Somit ist jeder Broker selbst dafür verantwortlich, einen entsprechenden Thread Pool bereitzustellen um die parallele Ausführung zu ermöglichen. Durch die Erzeugung einer eigenen Datenstruktur für den SubscriptionManager eines jeden WorkingFilters, muss dem EventBroker in Abhängigkeit von der Threadanzahl genügend Arbeitsspeicher zur Verfügung stehen. Durch den hohen Speicherbedarf, gerade bei der Anmeldung einer großen Zahl an Subscriptions, sollte bei alternativen Implementierungen des Thread Pools darauf geachtet werden, dass eine Wiederverwendung der WorkingFilter-Instanzen nach der Bearbeitung einer Publish-Nachricht möglich ist oder dass der Speicher dieser Instanzen nach der Benutzung wieder freigegeben wird.

Weitere am `EventBroker` beteiligte Komponenten sind der HTTPServer zur Entgegennahme von HTTP-Verbindungen, der MetricWriter zur Ausgabe von Statusinformationen, der EventWriter zur Ausgabe von Ereignisdaten und die zum Parsen bzw. Serialisieren genutzen Komponenten XMLSubscriptionReader, XMLEventReader, und XMLEventWriter.

- `XMLEventFilterHttpServer`

  Mit Hilfe der Komponente `XMLEventFilterHttpServer` können Nachrichten über das Protokoll HTTP verschickt werden. Diese Komponente stellt neben den Serverfunktionalitäten zum Empfang von HTTP-Nachrichten auch die Funktionalitäten zum versenden von HTTP-Nachrichten über einen Client zur Verfügung. Wie in der Abbildung 32 zu erkennen ist, laufen auf dem `EventBroker` zwei Instanzen des HTTP-Servers unter jeweils einem Port. Der erste Server dient als Subscribe-Server dem Anmelden von Subscriptions und der zweite Server kann als Publish-Server für das Veröffentlichen von Ereignisdaten genutzt werden. Die jeweiligen Adressen und Ports für beide Server lassen sich über die Konfigurationsdatei `XMLEventFilterHttpServer.properties` einstellen, wobei als Standardwerte bereits `http://localhost:8001` als

[27] Die Anzahl der `WorkingFilter`-Instanzen ist über den Parameter `workers` in der Konfigurationsdatei `XMLEventFilter.properties` einstellbar.

Endpunkt für den Publisher und `http://localhost:8002` als Endpunkt für den Subscriber vorkonfiguriert sind.

Zur Gewährleistung der Serverfunktion realisiert die Klasse HttpServerStart.java einen Thread Pool zur parallelen Annahme einer frei konfigurierbaren Anzahl an Verbindungen (`PooledHttpConnectionHandler.java`) [28] und die Weiterleitung der Nachrichten an die Parser-Komponenten `XMLSubscriptionReader` und `XMLEventReader` des Ereignisfilters. Im `PooledHttpConnectionHandler` werden die HTTP-Nachrichten auf XML-Dokumente untersucht und diese in einen Datenstrom überführt, welcher zur Weiterverarbeitung an den Ereignisfilter übergeben wird. Die bereitgestellten Clientfunktionalitäten werden von den Komponenten `XMLEventGenerator` und `XMLSubscriptionGenerator` zum Versenden von XML-Dokumenten genutzt. Dazu kann der Client die XML-Dokumente in Form eines `Vector`-Objekts oder als `File`-Datenstrom entgegennehmen.

- `EventWriter`

  Der `EventWriter` ermöglicht die Ausgabe der Ereignisdaten, welche den Filter passiert haben und auf einen bzw. mehrere Empfänger abgebildet werden konnten. Statt des Routings an diese Empfänger werden in der Testumgebung an dieser Stelle die Ereignisse als XML-Dokument angefordert und an die Komponente `Console` weitergeleitet. Soll die Ausgabe auf alternative Weise erfolgen, so kann das in der Komponente `EventWriter` realisiert werden.

- `XMLSubscriptionReader`

  Die Komponente `XMLSubscriptionReader` dient zum Einlesen (Parsen) von XML-basierten Subscriptions und wurde in der Testumgebung mit JAXB-Technologie realisiert. Im Abschnitt 5.5.2 wurde der Ablauf des Parsens mit JAXB bereits diskutiert. Durch den modularen Aufbau der Komponente lassen sich an dieser Stelle auch alternative Möglichkeiten zum Parsen von XML-Dokumenten realisieren.

- `XMLEventReader`

  Die Komponente `XMLEventReader` dient zum Einlesen (Parsen) von XML-basierten Ereignisdaten und wurde in der Testumgebung mit StAX-Technologie realisiert. Im Abschnitt 5.5.6 wurde der Ablauf des Parsens mit StAX bereits diskutiert. Durch den modularen Aufbau der Komponente lassen sich an dieser

[28] Die Anzahl der `PooledHttpConnectionHandler`-Instanzen ist über den Parameter `workers` in der Konfigurationsdatei `XMLEventFilter.properties` einstellbar. Mit dem zusätzlichen Parameter `queue` kann angegebenen werden, wie viele Verbindungen gleichzeitig durch den Server angenommen werden können um anschließend in einer Warteschlange auf die Abarbeitung zu warten. Der Parameter `waitForFinish` gibt eine Pufferzeit in Millisekunden an, um bei Unterbrechungen des Datenstromes zu warten, bis die HTTP-Verbindung geschlossen wird.

Stelle auch alternative Möglichkeiten zum Parsen von XML-Dokumenten, zum Beispiel mit der Iterator-Variante von StAX, realisieren.

- `XmlEventWriter`

  Die Komponente XmlEventWriter ermöglicht die Serialisierung von gefilterten Ereignisdaten in ein XML-Dokument, das dem ursprünglichen Aufbau des eingelesenen Ereignisses entspricht. Im Rahmen der Testumgebung wurde zu diesem Zweck ebenfalls die StAX-Technologie in der Cursor-Variante genutzt, wie es auch beim Einlesen der Fall war. Auch hier lassen sich durch den modularen Aufbau der Komponente alternative Möglichkeiten zum Serialisieren von `Event`-Objekten realisieren.

`Console`

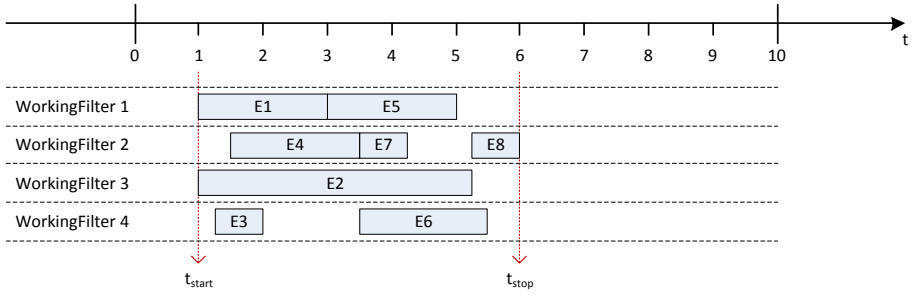
Zur Ausgabe von gefilterten Ereignisdaten dient in dieser Testkonfiguration die Java-Standardausgabe über die Konsole. Die Ausgabe erfolgt hier mit Unterstützung der `XMLEventWriter`-Komponente in Form eines XML-Dokuments. In einem Produktivsystem würde an dieser Stelle die Weiterleitung der Ereignisdaten an die ermittelten Empfänger erfolgen. Zu Testzwecken und für eine Performanceanalyse bietet diese Ausgabevariante aufgrund der direkten Rückmeldung flexible Auswertemöglichkeiten. Für die Filterung einer großen Menge an Ereignissen sollte die Ausgabe auf der Konsole in der Konfigurationsdatei des Ereignisfilters auskommentiert werden, da die in großer Zahl verursachten I/O-Operationen den Filtervorgang nachteilig beeinflussen können.

`MetricWriter`

Der `MetricWriter` läuft als gesonderter Thread im Rahmen der `EventBroker`-Komponente zur Visualisierung verschiedener Messwerte. Diese Komponente dient somit vorrangig der Performanceanalyse indem Informationen über verarbeitete Ereignisse wie Anzahl und Dauer der Filterung angezeigt werden. Die Funktionen des `MetricWriters` werden bereits an dieser Stelle beschrieben und dienen somit als Vorbereitung der Performanceanalyse im Kapitel 6.3.

Bei der Berechnung von Durchsatz und Verweilzeit wird zwischen zwei Arten der Zeitmessung unterschieden, die Verarbeitungszeit (engl. processing time) und die Laufzeit (engl. running time). Die Verarbeitungszeit eines Ereignisses gibt an, wie lange der Filtervorgang inklusive dem Vorgang des Parsens für dieses einzelne Ereignis gedauert hat. Ermittelt wird diese Zeit durch die Differenz der Systemzeiten vor dem Parsen und nach der anschließenden Filterung, gemessen in Nanosekunden. Bei der durchschnittlichen Angabe der Verarbeitungszeit werden die Einzelzeiten der Ereignisse aufsummiert und durch die Anzahl der von der Komponente `XMLEventGenerator` gesendeten Ereignisse geteilt. Mit der durchschnittlichen Verarbeitungszeit wird somit

die Zeit berechnet, mit der ein Ereignis im Durchschnitt einen Thread beansprucht. Angenommen, es laufen nun mehrere Threads zur Verarbeitung von Ereignisdaten parallel, zum Beispiel auf mehreren Kernen eines Prozessors, so ändert sich diese Zahl nicht signifikant, da ein Ereignis weiterhin nur von einem Thread verarbeitet werden kann. Um diesen Sachverhalt berücksichtigen zu können, wurde die Ausgabe der Laufzeit eingeführt. Bei der Berechnung der Laufzeit werden nicht mehr die Einzelzeiten aufsummiert, sondern die Start- und Endzeit der Verarbeitung ermittelt und daraus eine Differenz gebildet, welche durch die Anzahl der von der Komponente `XMLEventGenerator` gesendeten Ereignisse geteilt wird. Anhand der folgenden Abbildung 34 sollen die Unterschiede zwischen Verarbeitungszeit (engl. processing time) und Laufzeit (engl. running time) verdeutlicht werden.



$$\emptyset\ processingTime = \frac{t_{E1} + t_{E2} + \cdots + t_{En}}{n} = \frac{3 + 5{,}25 + 0{,}75 + 3 + 3 + 0{,}75 + 3 + 0{,}75}{8} = \mathbf{2{,}4375}$$

$$\emptyset\ runningTime = \frac{t_{stop} - t_{start}}{n} = \frac{6 - 1}{8} = \mathbf{0{,}625}$$

Abbildung 34: Unterschiede zwischen Verarbeitungszeit und Laufzeit des Filtervorgangs (Quelle: eigene Darstellung)

Mit dieser Darstellung wird von einer gleichzeitigen Verarbeitung der Ereignisdaten auf vier Prozessorkernen, also mit vier `WorkingFilter`-Instanzen, ausgegangen. Die Berechnung der Verarbeitungszeit und der Laufzeit ergibt sich aus den dargestellten Formeln.

In der Klasse `MetricWriter.java` kann über die Konstante `removedEventsPerGenerator` die Anzahl an Ereignissen angegeben werden, die der Filter planmäßig herausgefiltert hat (d. h. diese Ereignisse können nicht auf einen Empfänger abgebildet werden). In der Konfigurationsdatei `XMLEventFilter.properties` kann die Anzahl der Ereignisgeneratoren und die Anzahl der gesendeten Ereignisse pro Generator eingestellt werden. Diese Angabe ist für zuverlässige Performancemessungen erforderlich, da Ereignisse, die nicht auf einen Empfänger abgebildet werden können, vom Filter unter Umständen auch nicht als Ereignis erkannt werden. Somit können diese Ereignisse nicht automatisch gezählt werden, obwohl sie verarbeitet wurden. Würden nur die Ereignisse zur Berechnung von Verarbeitungszeit und Laufzeit berücksichtigt, die den Filter passieren, kommt es zu verfälschten Ergebnissen. Dieser Effekt soll mit der manuellen Konfiguration vermieden werden.

6.2 Integrationsszenario

Im Rahmen der Forschungsprojekte ADiWa und Sustainable Energy and Production wurde bei SAP ein Prototyp für den Bereich Energy Management entwickelt. Ziel dieses Projekts ist die Entwicklung eines Energy-Monitoring-Tools für Fabriken inklusive der Integration intelligenter Messgeräte, so genannte Smart Meter. Die erfassten Energieverbrauchsdaten sind dabei sehr feingranular und können bestimmten Organisationsstrukturen oder Produktionsprozessen zugeordnet werden. Dadurch wird es möglich, den Energieverbrauch eines jeden Produktes zu ermitteln. [SAP10]

Mit diesem Szenario wird das Zusammenspiel zahlreicher Softwarekomponenten unterschiedlicher Technologien in Kombination mit der Integration von echten Energiemessgeräten praxisnah demonstriert.

Die Kernkomponenten dieses Integrationsszenarios werden als Architekturschaubild in Abbildung 35 visualisiert. Daraus ergibt sich die folgende Rollenverteilung. Die Aufgabe des Subscribers übernimmt ein virtuelles Armaturenbrett (engl. Dashboard), indem die Verbrauchsdaten bestimmter Geräte ausgewählt werden können. Die Anbindung erfolgt über eine reine TCP-Verbindung. Da das Dashboard gleichzeitig der Visualisierung dient, bekommt dieses auch die Rolle des Empfängers. Die Ereignisdaten werden in Form von XML-Nachrichten ebenfalls über die TCP-Verbindung gesendet.

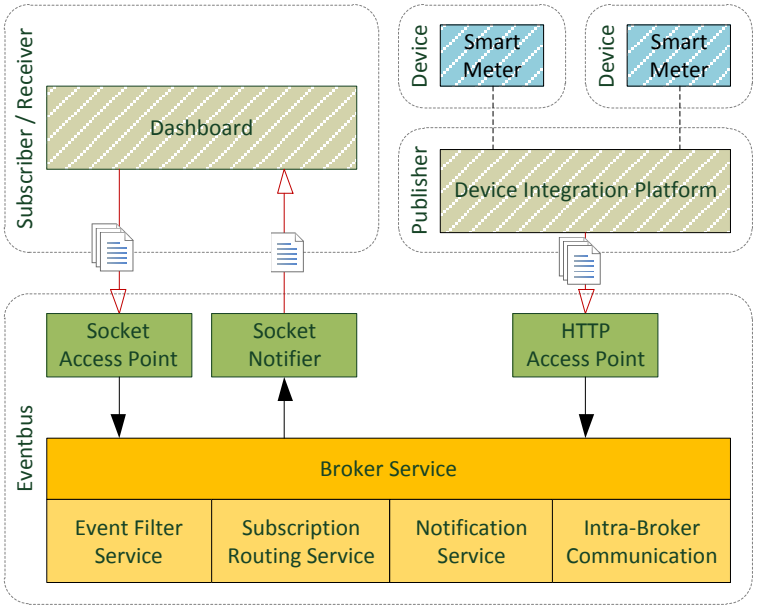


Abbildung 35: Architekturschaubild des Integrationsszenarios (Quelle: eigene Darstellung nach [SAP10])

Als Publisher dient in diesem Szenario die Geräteintegrationsplattform (engl. Device Integration Platform), welche die Ereignisdaten in Form von XML-Nachrichten über eine HTTP-Verbindung versendet. Diese Geräteintegrationsplattform dient somit als Schnittstelle zwischen den einzelnen Geräten (engl. Devices), zum Beispiel Smart Meter, und dem Brokernetzwerk und ist gleichzeitig für die Generierung der Ereignisdaten zuständig. Die Energiemessgeräte können beispielsweise per USB an die Plattform angebunden sein. Der Broker, wie er in Kapitel 5.1.1 beschrieben wurde, wird in diesem

Szenario als Eventbus bezeichnet. Die Funktionen dieser Brokerimplementierung sind dennoch identisch.

6.2.1 Dashboard

Das Dashboard ist eine auf Silverlight basierende Softwarekomponente zur Visualisierung ausgewählter Energieverbrauchsdaten. Mit der Dashboard-Komponente wird es möglich, die eigene Infrastruktur zu definieren und zu modellieren. Auf der Basis der Smart Meter auf der niedrigsten Stufe lassen sich logische Hierarchien von weiteren Objekten wie Gebäuden definieren, wobei Regeln zur Aggregation dieser Objekte erstellt werden können. In Abbildung 36 wird dieses Dashboard als Beispiel zur Visualisierung von ausgewählten Verbrauchsdaten dargestellt.

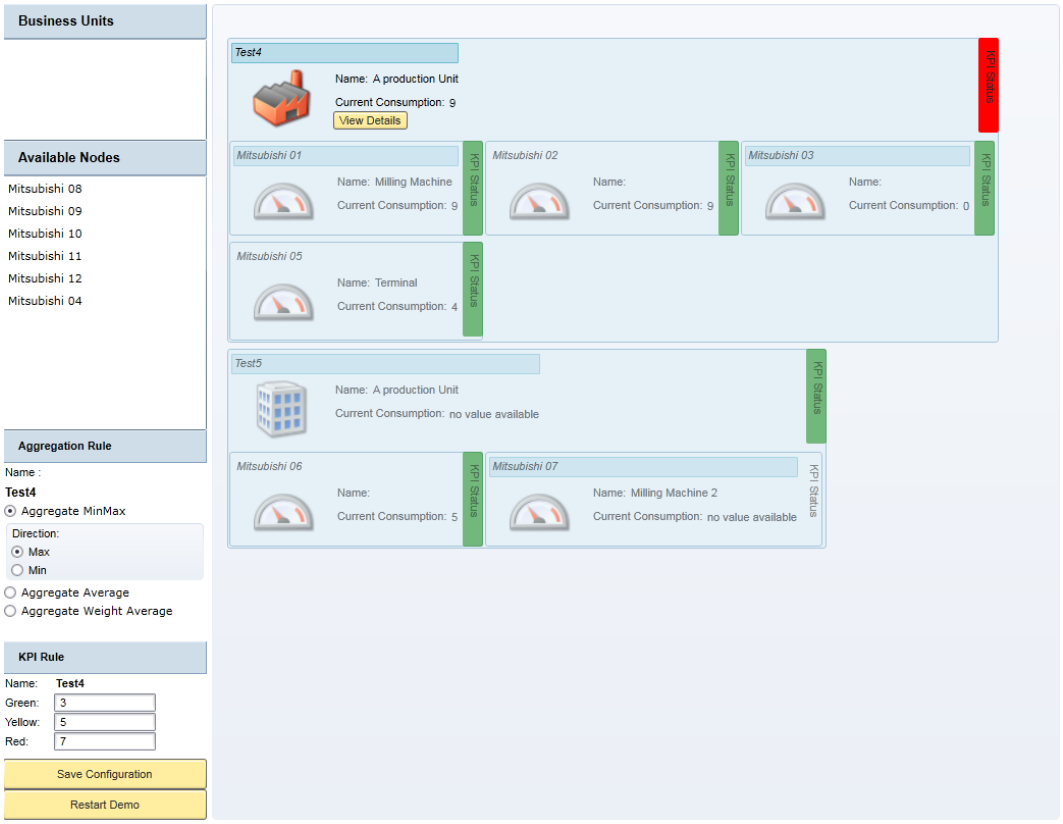


Abbildung 36:Dashboard zur Visualisierung von Verbrauchsdaten (Quelle: [SAP10])

Die Geräteintegrationsplattform greift in bestimmten Intervallen auf die Messwerte der Smart Meter zu und veröffentlicht diese als XML-Nachricht im Eventbus. Das Dashboard subskribiert sich nun für diese veröffentlichten Ereignisdaten am Eventbus. Anschließend bekommt das Dashboard die gewünschten Ereignisse übermittelt und kann diese auf der Benutzungsoberfläche visualisieren. [SAP10]

6.2.2 Device Integration Platform

In diesem Szenario werden intelligente Messgeräte, so genannte Smart Meter, zur Erfassung verschiedener Messwerte genutzt, um daraus Ereignisdaten für den Eventbus generieren zu können. Zu dieser Anbindung von elektronischen Geräten wurde bei SAP Research eine Geräteintegrationsplattform entwickelt, welche auf der OSGi-Plattform basiert und somit modular aufgebaut ist. Mit dieser Plattform wird es möglich, verschiedene elektrische Geräte in Form von Sensoren (RFID-Lesegeräte, Smart Meter, usw.) und Aktoren (Lichtsignalanlagen, Lagersteuerungen, usw.) über einen zentralen Punkt konfigurieren und verwalten zu können. Diese Integration führt zu einheitlichen Schnittstellen für einen unkomplizierten Zugriff auf diese Geräte in unterschiedlichen Anwendungsprogrammen.

Man unterscheidet dabei zwischen Adapter-Modulen, die Objekte der realen Welt, wie ein RFID-Lesegerät, repräsentieren und Logik-Modulen, die Logik-Funktionalitäten zur Verfügung stellen. Jedes Modul besteht dabei aus zwei Teilen. Der eine Teil wird zur Laufzeit ausgeführt, der andere liefert die Konfigurationsumgebung. Die Konfiguration erfolgt in Form von XML-Dateien, die in einer zentralen Konfigurationsdatenbank abgelegt werden.

Die Kommunikation zwischen den Modulen erfolgt ebenso auf Grundlage von XML-Nachrichten. Diese können entweder synchron ausgetauscht werden, d.h. ein Modul sendet eine Nachricht an ein konkretes anderes Modul, oder asynchron. Bei dieser Variante sendet ein Modul eine Nachricht in Form eines Broadcast an die Geräteintegrationsplattform. Auf diese Nachricht können sich dann Module subskribieren und diese somit empfangen und verarbeiten. In der Geräteintegrationsplattform stehen einige Module zur Verfügung, die Schnittstellen für Geräte sowie Schnittstellen zum ERP-System anbieten. Einige dieser Module, darunter ein Modul zur Anbindung von Smart Meter Geräten an die Geräteintegrationsplattform, kommen im Rahmen dieses Integrationsszenarios zum Einsatz. [SAP101]

6.2.3 Eventbus

Wie bereits beschrieben übernimmt der Eventbus in diesem Integrationsszenario die Funktion des Brokers. Der Unterschied zu Kapitel 5.1.1 liegt in der Anbindung der externen Komponenten und der Rollenverteilung. In dieser Implementierung ist der Subscriber gleichzeitig Empfänger der Ereignisdaten. Somit werden in der Subscription keine Empfängerinformationen übertragen. Wenn dies durch den Ereignisfilter erkannt wird, werden die Absenderdaten des Subscribers als Empfängerdaten genutzt. Zur Anbindung des Dashboards als Subscriber bzw. Empfänger kommt hier ein Socketadapter zum Einsatz, der über eine TCP-Verbindung kommuniziert.

6.3 Performanceanalyse

Nachdem bereits die Funktionsfähigkeit der Implementierung anhand von zwei Szenarien unter Beweis gestellt wurde, folgt nun eine Performanceanalyse zur Untersuchung des Ereignisfilters in verschiedenen Lastsituationen. Die Grundlage der Messungen bildet die in Kapitel 6.1 beschriebene Testumgebung.

Zur Analyse der Performance wurden zwei Testfälle ausgewählt, welche sowohl den EventBroker als auch den Hostrechner an seine Belastungsgrenze bringen sollen. Im ersten Fall wurden bei konstanter Ereignisanzahl (1000 Generatoren senden jeweils 100 Ereignisse) die Anzahl der angemeldeten Subscriptions in jedem Durchlauf um den Faktor 10 erhöht. Somit soll das Verhalten des Ereignisfilters mit steigender Subscriptionbelastung gezeigt werden. Der zweite Fall demonstriert das Verhalten des Ereignisfilters bei konstanter Subscriptionanzahl (10 angemeldete Subscriptions) mit steigender Anzahl zu verarbeitender Ereignisse. Dabei werden in jedem Durchlauf zwei Parameter modifiziert, die Anzahl der Ereignisgeneratoren und die Anzahl der von jedem Generator produzierten Ereignisse. Auch hier erfolgt jeweils eine Erhöhung um den Faktor 10 in jedem Durchlauf. Um starke Schwankungen durch das Senden über eine HTTP-Verbindung zu vermeiden, wurde für die Performanceanalyse der Einsatz der internen Clients bevorzugt.

6.3.1 Konfiguration

Für die Durchführung der Performancemessungen standen drei verschiedene Rechner zur Verfügung, ein Single-Core-, ein Double-Core- und ein Quad-Core-PC, jeweils mit dem Betriebssystem Windows 7. Der PC mit Quad-Core-Prozessor diente zum Experimentieren in drei verschiedenen Konfigurationen, wobei einzelne Prozessorkerne abgeschaltet wurden. Einen Überblick zu den verschiedenen Rechnerkonfigurationen gibt die folgende Tabelle 2.

<table>
<tr><th></th><th>PC1 (SC)</th><th>PC2 (DC)</th><th>PC3 (SC)</th><th>PC3 (DC)</th><th>PC3 (QC)</th></tr>
<tr><td>CPU</td><td>Intel Pentium M760 2,00 GHz</td><td>Intel Core 2 Duo E 6750 2,66 GHz</td><td colspan="3">AMD Phenom 2 X4 940 3,00 GHz</td></tr>
<tr><td>Kerne</td><td>1</td><td>2</td><td>1</td><td>2</td><td>4</td></tr>
<tr><td>RAM</td><td>2 GB</td><td>4 GB</td><td colspan="3">8GB</td></tr>
<tr><td>OS</td><td>Windows 7 Professional 32-Bit</td><td>Windows 7 Enterprise 64-Bit</td><td colspan="3">Windows 7 Professional 64-Bit</td></tr>
<tr><td>JDK</td><td>Java 6 U21 32-Bit</td><td>Java 6 U21 64-Bit</td><td colspan="3">Java 6 U21 64-Bit</td></tr>
</table>

Tabelle 2: Rechnerkonfigurationen zur Durchführung von Performancemessungen

Auf allen Testrechnern kam zum Starten der OSGi-Konfiguration die Entwicklungsumgebung Eclipse Helios als 32-Bit Version zum Einsatz. Sämtliche Experimente wurden mehrfach durchgeführt, wobei keine signifikanten Unterschiede der Laufzeiten festgestellt wurden. Um die, durch die Java Virtual Machine (kurz JVM),

entstehenden Schwankungen bei den ersten Durchgängen nach einem Neustart zu vermeiden, wurden vor den eigentlichen Messungen mindestens fünf Probeläufe durchgeführt, bis sich die Laufzeiten bei wiederholtem Ausführen der gleichen Experimente auf einem konstanten Niveau eingependelt hatten.

6.3.2 Subscriptiongenerierung

Die Erzeugung der Subscriptions erfolgt mit der in 6.1.2 beschriebenen Komponente `XMLSubscriptionGenerator` in Form des internen Clients. Für die Performanceanalyse wurde diese Variante gewählt, da sich in vorbereitenden Tests herausgestellt hat dass das Senden von Subscriptions über eine HTTP-Verbindung neben der zusätzlichen Verzögerung auch zu Schwankungen in der Verarbeitung führt. Aus diesem Grund erfolgt nun die Einbettung des Generators als OSGi-Bundle in den Eventbroker, wodurch es möglich wird die generierten Ereignisdaten in Form eines XML-Dokuments direkt per Java-Aufruf an den Ereignisfilter zu übermitteln. Die entsprechenden Konfigurationsparameter wurden dazu an folgenden Stellen des `XMLSubscriptionGenerator`-Bundles gesetzt:

`Constants.java`

- Anzahl Subscriptions: `SOURCE_GENERATE`
- XML-Datei: `SOURCE_FILE`

`Activator.java`

- Auswahl der Subscriptionquelle (`GENERATE` oder `FILE`) und der entsprechenden Parameter (`SOURCE_GENERATE` oder `SOURCE_FILE`) im Aufruf der `main`-Methode.

Für das alternative Testen der HTTP-Verbindung stehen ausführbare Batch-Dateien zur Verfügung, die keine Konfiguration in den Java-Klassen erfordern. Über entsprechende Aufrufparameter kann hier ebenfalls eine Konfiguration vorgenommen werden, wie das folgende Beispiel zeigt.

```
java -jar -Xms1024m -Xmx1024m SubscriptionGenerator.jar
GENERATE 100 HTTP http://localhost:8002
```

Bei dem Start mit dieser Konfiguration werden 100 Subscriptions generiert und an die Adresse `http://localhost:8002` geschickt.

6.3.3 Ereignisgenerierung

Die Erzeugung von Ereignisdaten übernimmt die in 6.1.2 beschriebene Komponente `XMLEventGenerator` in Form des internen Clients. Auch an dieser Stelle wurde für die Performanceanalyse diese Variante gewählt, um die bei Verwendung von HTTP-Verbindungen auftretenden Schwankungen zu vermeiden. Der Ereignisgenerator ist

somit auch in Form eines OSGi-Bundles in den EventBroker eingebettet. Nachdem die generierten Ereignisdaten in Form eines XML-Dokuments an den Ereignisfilter übermittelt wurden, können diese direkt an den Parser übergeben werden. Die entsprechenden Konfigurationsparameter für die Performanceanalyse wurden dazu an folgenden Stellen gesetzt:

`XMLEventGenerator: Constants.java`

- Anzahl Ereignisse: `SOURCE_GENERATE`
- XML-Datei: `SOURCE_FILE`
- Anzahl Generatoren: `GENERATOR_NUMBER`

`XMLEventGenerator: Activator.java`

- Auswahl der Ereignisquelle (`GENERATE` oder `FILE`) und der entsprechenden Parameter (`SOURCE_GENERATE` oder `SOURCE_FILE`) im Aufruf der `main`-Methode. Der Parameter `GENERATOR_NUMBER` bleibt hier stets unverändert.

`XMLEventFilter: MetricWritter.java`

Anzahl der durch die generierten Subscriptions herausgefilterten Ereignisse pro Ereignisgenerator werden mit der Variablen `removedEventsPerGenerator` eingestellt. In diesem Szenario ergeben sich folgende Werte:

- Einlesen einer XML-Datei mit Beispieldaten:
  `event_0001.xml`: 0
  `event_0010.xml`: 2
- Automatisches Generieren von Ereignisdaten:
  unabhängig von der Anzahl (muss stets größer 14 sein): 11

`XMLEventFilter: XMLEventFilter.properties`

- Anzahl Ereignisse: `generate`
- Anzahl Generatoren: `generators`

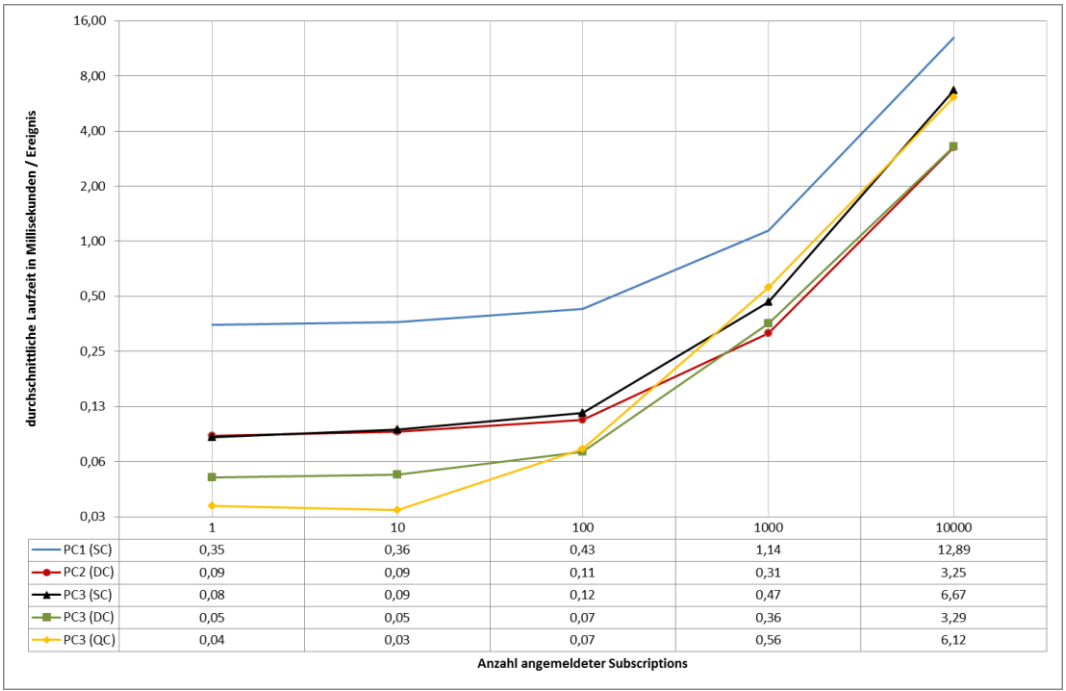
Für das Testen der HTTP-Verbindung stehen ausführbare Batch-Dateien zur Verfügung, die keine Konfiguration in den Java-Klassen erfordern. Über entsprechende Aufrufparameter kann hier ebenfalls eine Konfiguration vorgenommen werden, wie das folgende Beispiel zeigt.

```
java -jar -Xms1024m -Xmx1024m EventGenerator.jar GENERATE
100 HTTP http://localhost:8001 1000
```

Bei dem Start mit dieser Konfiguration werden mit 1000 Generatoren jeweils 100 Ereignisse generiert und an die Adresse `http://localhost:8001` geschickt. Für eine korrekte Berechnung der Messergebnisse ist es trotzdem erforderlich, die oben beschriebenen Einstellungen im Bundle `XMLEventFilter` vorzunehmen (siehe dazu auch Abschnitt 6.1.2).

6.3.4 Auswertung der Messergebnisse

Im ersten Testfall wurde in jedem Durchlauf von 1000 Generatoren eine konstante Anzahl von 100 Ereignissen generiert. Mit dem kontinuierlichen Erhöhen der Subscriptionanzahl (1, 10, 100, 1.000, 10.000) soll somit das Verhalten der Filterleistung bei einer exponentiell wachsenden Größe der Datenstruktur gezeigt werden. Gemessen wurde dazu die Laufzeit der Ereignisse auf dem Broker als Grundlage zur Berechnung der durchschnittlichen Laufzeit und des Durchsatzes des EventBrokers. Abbildung 37 und Abbildung 39 stellen die berechneten Werte für Laufzeit und Durchsatz bei den Testläufen auf den 5 verschiedenen Rechnerkonfigurationen grafisch dar.



	1	10	100	1000	10000
PC1 (SC)	0,35	0,36	0,43	1,14	12,89
PC2 (DC)	0,09	0,09	0,11	0,31	3,25
PC3 (SC)	0,08	0,09	0,12	0,47	6,67
PC3 (DC)	0,05	0,05	0,07	0,36	3,29
PC3 (QC)	0,04	0,03	0,07	0,56	6,12

Abbildung 37: Durchschnittliche Laufzeit eines Ereignisses in Abhängigkeit der Subscriptionanzahl (Quelle: eigene Darstellung)

Jede Kurve in Abbildung 37 zeigt dabei die durchschnittliche Laufzeit eines Ereignisses in Abhängigkeit der Subscriptionanzahl bei konstanter Ereignismenge. Alle Kurven zeigen dabei einen relativ geringen Anstieg der Laufzeit in den niedrigen Bereichen der Subscriptionanzahl. Erst ab einer Menge von ca. 1000 angemeldeten Subscriptions setzt ein exponentieller Anstieg der Laufzeit ein. Ein ähnliches Verhalten konnte auch bei den Performancemessungen zum Carzaniga-Ansatz in [Car03] nachgewiesen werden, wobei der exponentielle Anstieg dort bei ca. 4 Millionen Constraints beginnt.

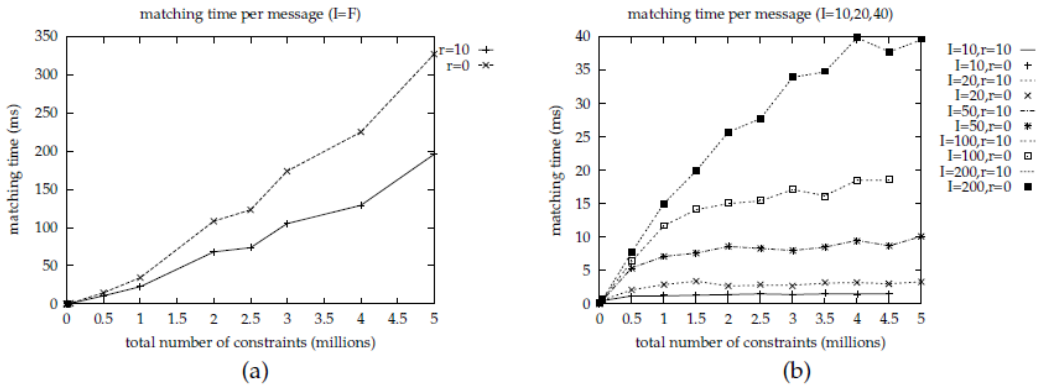


Abbildung 38: Performance des Forwarding Algorithmus von Carzaniga et al. bei Aufbau einer zentralisierten Architektur (a) und einer verteilten Architektur (b) (Quelle: [Car01])

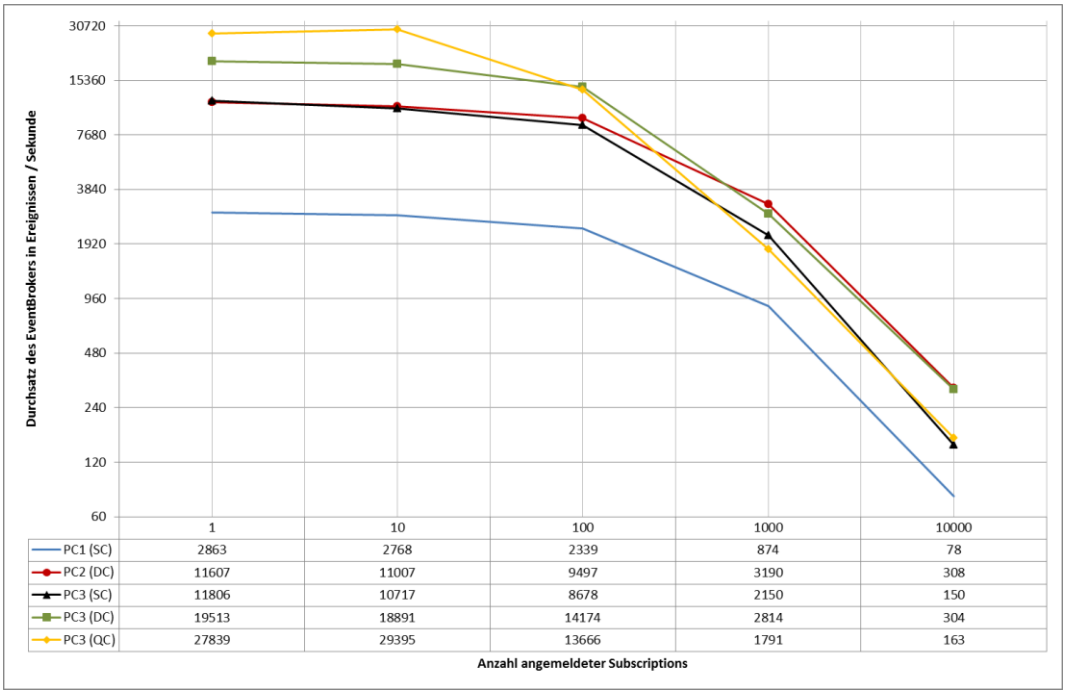
Da die vorgestellte Testumgebung mit nur einem EventBroker zentralisiert aufgebaut ist, lassen sich die Messergebnisse am besten mit den Ergebnissen von Carzaniga et al. in Teil (a) der Abbildung 38 vergleichen. Die stark abweichenden Schwellenwerte ergeben sich vermutlich aus dem abweichenden Subscriptionmodell, was durch die ADiWa-Subscriptionsprache im Vergleich recht komplex aufgebaut ist. Die ebenfalls um Größenordnungen abweichenden Messwerte lassen sich durch die schwächere Rechnerkonfiguration erklären, die mit einem 950 MHz-Prozessor und 512 MB RAM angegeben wurde. Der in [Car01] und [Car03] getestete Forwarding Algorithmus wurde in C++ implementiert.

Neben dem Verhalten bei zunehmender Subscriptionbelastung sollte mit den Experimenten gezeigt werden, wie der Ereignisfilter beim Einsatz von Mehrkernprozessoren skaliert. Dazu wurden die Messungen auf den beschriebenen fünf Rechnerkonfigurationen mit drei unterschiedlichen PCs durchgeführt. Bei PC1 mit Single-Core-Prozessor und PC2 mit Double-Core-Prozessor zeigt sich das erwartete Verhalten in zwei relativ parallel verlaufenden Kurven mit deutlichem Abstand. PC3 mit Quad-Core-Prozessor hingegen wurde nacheinander mit allen Kernen sowie mit zwei oder drei abgeschalteten Kernen betrieben. Durch dieses Experiment zeigte sich ein merkwürdiges Verhalten, welches in dieser Form nicht erwartet wurde. Der Betrieb mit einem sowie zwei Kernen führte zu ähnlichen Ergebnissen wie bei PC1 und PC2, die Durchläufe mit allen vier Kernen zeigen ab einer Zahl von ca. 100 angemeldeten Subscriptions keine Performancesteigerungen mehr. Im Gegenteil schneidet diese Kurve des Vierkerndurchlaufes fast alle anderen Kurve, was auf einen Einbruch der Leistungsfähigkeit bei zunehmender Kernanzahl schließen lässt. Die Suche nach einer Begründung für dieses Verhalten gestaltet sich allerdings schwierig, da hier zahlreiche Ansatzpunkte herangezogen werden können. Zunächst beschränken sich diese Untersuchungen auf das Betriebssystem mit seinem Verhalten beim Umgang mit Multithreading-fähigen Softwarearchitekturen.

So besitzt das eingesetzte Betriebssystem Windows 7 die Möglichkeit, einzelne Threads auf einen anderen Prozessorkern zu migrieren, wenn dies erforderlich erscheint [Lan10]. Die einzelnen Testläufe haben gezeigt, dass die Auslastung der Prozessorkerne stets gleichmäßig verteilt ist, mit zunehmender Zahl an Subscriptions bis zu 100%. Hier liegt

nun die Vermutung nahe, dass das Bestriebssystem durch ständiges Migrieren der Threads alle Kerne gleichmäßgig auszulasten versucht. In diesem Fall würde sich somit die Erhöhung der Kernanzahl durch steigenden Migrationsaufwand negativ auf die Laufzeiten auswirken, wie es auch die Kurve des Rechners mit Quad-Core-Prozessor zeigt. Dieser Aspekt bietet einen interessanten Ansatzpunkt für weiterführende Arbeiten, um zu untersuchen, wie die Datenstruktur und der damit verbundene Filtervorgang optimiert werden können. So könnte eine bessere Skalierung auf Rechnersystemen mit mehr als zwei Kernen erreicht werden.

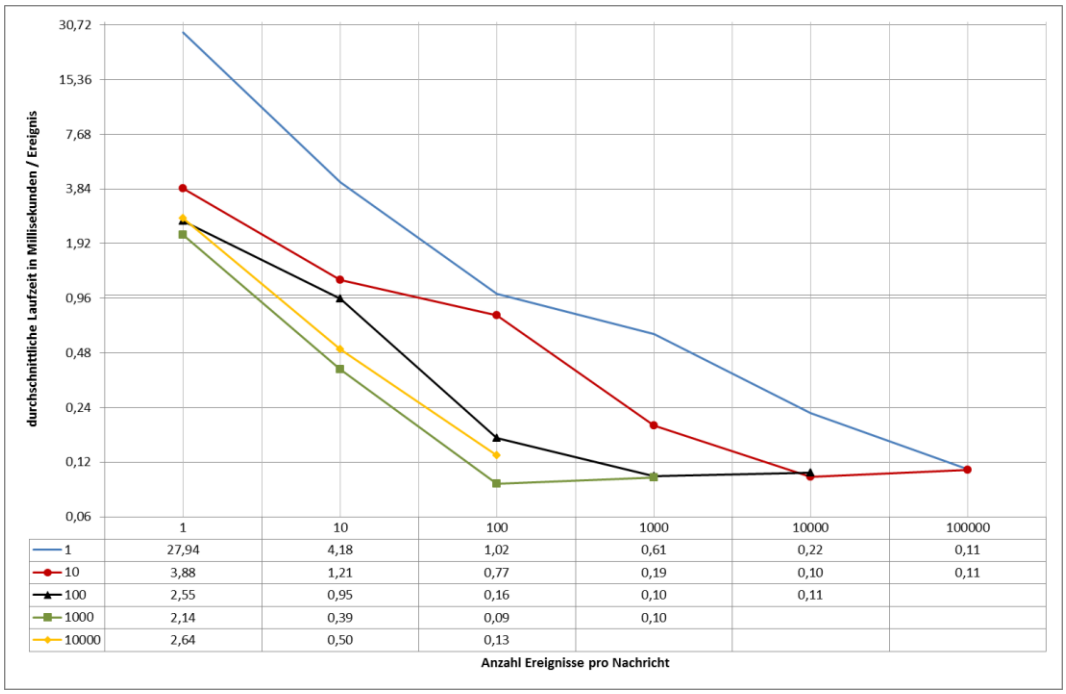
Analog zu den Berechnungen der Laufzeit ergibt sich für den Durchsatz auf einem EventBroker die in Abbildung 39 dargestellte Grafik. Auch hier entsprechen die einzelnen Kurven den Messreihen auf den verschiedenen Rechnerkonfigurationen.



	1	10	100	1000	10000
PC1 (SC)	2863	2768	2339	874	78
PC2 (DC)	11607	11007	9497	3190	308
PC3 (SC)	11806	10717	8678	2150	150
PC3 (DC)	19513	18891	14174	2814	304
PC3 (QC)	27839	29395	13666	1791	163

Abbildung 39: Durchsatz des EventBrokers in Abhängigkeit der Subscriptionanzahl (Quelle: eigene Darstellung)

Im zweiten Testfall wurde eine konstante Zahl von 10 am EventBroker angemeldeten verwendet. In jedem Durchlauf wurde sowohl die Ereignisanzahl als auch die Zahl der Ereignisgeneratoren variiert. Gemessen wurde auch hier wieder die Laufzeit, anhand derer die durchschnittliche Laufzeit pro Ereignis und der Durchsatz des Brokers berechnet werden kann, was in Abbildung 40 und Abbildung 41 dargestellt ist. Mit diesem Testfall soll die Leistungsfähigkeit des Ereignisfilters bei steigender Nachrichtengröße und steigender Anzahl an Publishern demonstriert werden.



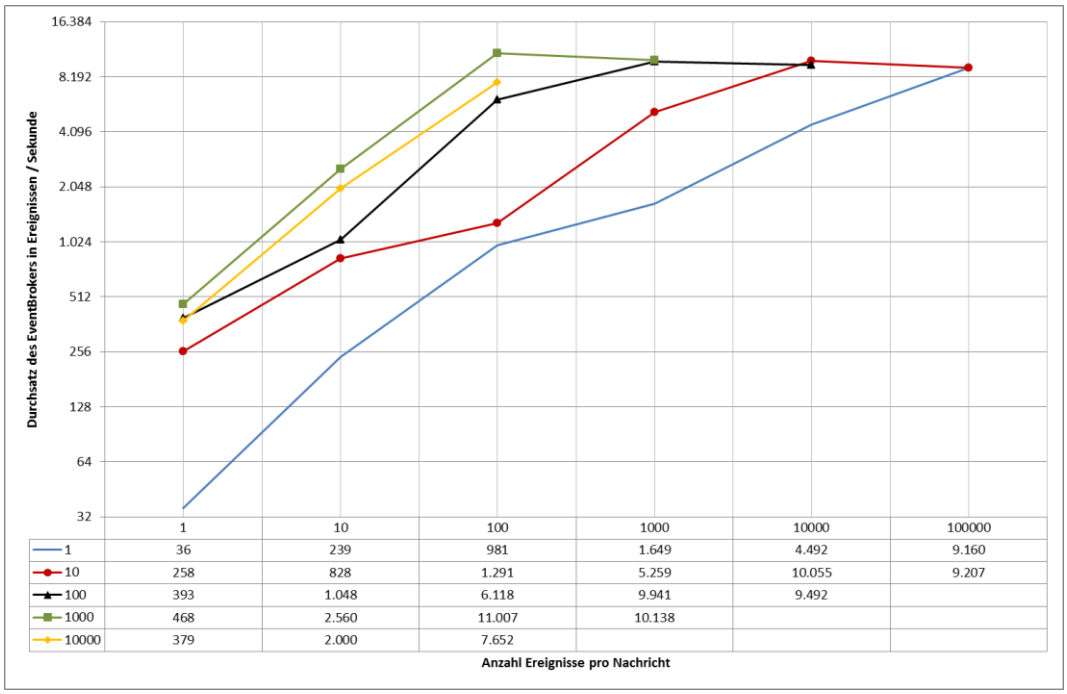
	1	10	100	1000	10000	100000
1	27,94	4,18	1,02	0,61	0,22	0,11
10	3,88	1,21	0,77	0,19	0,10	0,11
100	2,55	0,95	0,16	0,10	0,11	
1000	2,14	0,39	0,09	0,10		
10000	2,64	0,50	0,13			

Abbildung 40: Durchschnittliche Laufzeit eines Ereignisses in Abhängigkeit der Anzahl von Ereignissen pro Nachricht (Quelle: eigene Darstellung)

Die Werte beider Abbildungen wurden anhand der gemessenen Laufzeiten des Filtervorgangs berechnet, wobei die Rechnerkonfiguration PC2 genutzt wurde. Aufgrund des hohen Speicherbedarfs für das Generieren der Ereignisdaten konnten mit dieser Konfiguration nicht alle gewünschten Experimente durchgeführt werden. Das ist somit auch der Grund für die fehlenden Werte in den Datentabellen und Kurven.

Jede der Kurven in Abbildung 40 zeigt dabei die Entwicklung der Filterleistung mit exponentiell steigender Ereignisanzahl (1, 10, 100, 1.000, 10.000, 100.000) für eine bestimmte Anzahl an Publishern (Ereignisgeneratoren), die ebenfalls exponentiell wächst (1, 10, 100, 1.000). Die erwartete kontinuierliche Zunahme der Laufzeit wurde mit diesen Experimenten nicht bestätigt, vielmehr ist eine Abnahme der durchschnittlichen Laufzeiten bis zu einem gewissen Optimum an Nachrichtengröße zu beobachten. Erst wenn die Anzahl an Ereignissen pro Nachricht weiter ansteigt, nimmt auch die Laufzeit zu. Wie das Laufzeitdiagramm anschaulich zeigt, hängt dieses Optimum nicht zwangsläufig von der Nachrichtengröße ab, sondern von der Gesamtzahl der relativ gleichzeitig vom Filter verarbeiteten Ereignisanzahl. Diese lag in den Testläufen mit PC2 bei ca. 100.000 Ereignissen.

Wie auch im ersten Testfall zeigt das folgende Diagramm (Abbildung 41) nun den aus der Laufzeit berechneten Durchsatz, welcher sich für den Filtervorgang auf einem EventBroker ergibt. Die einzelnen Kurven entsprechen wieder der Anzahl der gleichzeitig sendenden Publisher.



	1	10	100	1000	10000	100000
1	36	239	981	1.649	4.492	9.160
10	258	828	1.291	5.259	10.055	9.207
100	393	1.048	6.118	9.941	9.492	
1000	468	2.560	11.007	10.138		
10000	379	2.000	7.652			

Abbildung 41: Durchsatz des EventBrokers in Abhängigkeit der Anzahl von Ereignissen pro Nachricht (Quelle: eigene Darstellung)

Die in diesem Abschnitt diskutierten zwei Testfälle stellen nur einen kleinen Ausschnitt der möglichen Testfälle für eine umfassende Performanceanalyse dar. Neben den untersuchten Parametern Subscriptionanzahl und Ereignisanzahl gibt es zahlreiche weitere Stellschrauben, die modifiziert werden können, um die Filterung von Ereignisdaten in einem Publish/Subscribe-System weiter zu optimieren. So könnte beispielsweise untersucht werden, welchen Einfluss die Anzahl an WorkingFilter-Instanzen auf die Laufzeit und den Durchsatz haben oder in welcher Größenordnung der, durch die Nutzung von HTTP-Verbindungen, anfallende Übertragungsoverhead ausfällt, wobei wiederum zwischen lokalem Senden und dem Senden über verschiedene Netzwerkkonfigurationen unterschieden werden kann.

Auch filterintern gibt es zahlreiche Ansatzpunkte für weiterführende Performancemessungen, wie das Verhalten mit alternativen Datenstrukturen für die einzelnen Bestandteile des SubscriptionsManagers oder weitere Varianten des Parsens von XML-Dokumenten. Hinsichtlich der Architektur könnten mit alternativen Brokerimplementierungen die Leistungsfähigkeiten von zentralisierten und verteilten Brokernetzwerken verglichen werden. Aber auch die Modifikation einzelner Parameter, wie preprocessingRounds, könnte weitere interessante Erkenntnisse liefern.

7 FAZIT

Welche Ziele wurden mit dieser Arbeit verfolgt und was ist das Innovative dabei?

Das zentrale Thema dieser Arbeit bilden Publish/Subscribe-Systeme aus der Gruppe der ereignisgesteuerten Architekturen. Hierbei galt es zu untersuchen, wie ein Ereignisfilter konzipiert werden muss, um möglichst effizient die Entscheidung treffen zu können, an welchen Empfänger ein veröffentlichtes Ereignis gesendet werden soll. Eine Herausforderung dabei war es, dieses System nicht nur skalierbar, sondern auch universell einsetzbar zu gestalten.

Bei der Frage nach dem innovativen Charakter dieser Ausarbeitung stehen sowohl das Ereignis- als auch das Subscriptionmodell im Fokus. In beiden Fällen wurde ein vollständig XML-basierter Ansatz gewählt, der in dieser Form bei den betrachteten verwandten Arbeiten bisher nicht realisiert wurde. Für das Subscriptionmodell wurde im Rahmen des ADiWa Projekts bei SAP Research eine sehr mächtige und flexible Subscriptionsprache entwickelt, die eine ideale Grundlage dieser Arbeit bildet. Mit dieser Sprache als Voraussetzung konnte somit nach einem Konzept gesucht werden, um allein auf der Basis einer Subscription in einem nicht typisierten Datenstrom nach Ereignissen zu suchen und eine Abbildung auf einen Empfänger vornehmen zu können, falls die inhaltlichen Bedingungen erfüllt sind.

Ein weiteres großes Ziel war die praktische Validierung des Konzepts für einen Ereignisfilter durch die Entwicklung eines Java-basierten Prototyps. Mit Unterstützung des OSGi-Komponentenmodells ist dieser Prototyp nicht nur flexibel erweiterbar (z.B. um zusätzliche Kommunikationsschnittstellen), sondern kann gleichzeitig als Kernbestandteil eines Brokers integriert werden.

7.1 Zusammenfassung

Wofür sind die Erkenntnisse aus dieser Arbeit nützlich?

Um die Relevanz dieser Arbeit zu fokussieren, wurde mit der Beschreibung zweier Anwendungsszenarien mit grundlegend verschiedenen Intentionen die praktische Einsetzbarkeit von Publish/Subscribe-Systemen skizziert. So wurde in Kapitel 3.1 der Einsatz dieser Architekturen in der Produktionslogistik der Automobilindustrie erläutert. Dabei kristallisierten sich, bedingt durch ein hohes Ereignis- und Subscriptionaufkommen, die Anforderungen an ein solches System bezüglich Effizienz und Skalierbarkeit heraus. Bereits bei diesem Szenario wurde die Notwendigkeit eines Brokernetzwerks gezeigt, um den Anforderungen in der Praxis gerecht werden zu können. Mit dem zweiten Szenario wurde ein Publish/Subscribe-System aus dem Bereich der Gebäudeautomatisierung beschrieben, welches zum Ziel hat, Sensoren und Aktoren ereignisbasiert zu steuern. Auch die Idee einer kabellosen Anbindung von

Publishern und Subscribern wurde an dieser Stelle vorgestellt. Die Anforderungen bezüglich der Verarbeitung von Subscriptions und Ereignissen sind in diesem Beispiel wesentlich geringer, dafür aber auch flexibel, was wiederum eine gute Skalierbarkeit erfordert.

Wie wurde bei der Umsetzung der Anforderungen vorgegangen und welche Probleme traten dabei auf?

Bevor mit der eigentlichen Konzeption eines Ereignisfilters begonnen werden konnte, wurden einige verwandte Arbeiten auf ihre Eigenschaften hin untersucht. Dabei stellte sich die Datenstruktur als Herzstück eines solchen Filters heraus, was zu den drei Ansätzen Binary Decision Diagram (Abschnitt 4.1.1), Database Management System (Abschnitt 4.1.2) und Forwarding Table (Abschnitt 4.1.3) führte. Trotz der vielversprechenden Ideen hinter diesen Ansätzen stellte sich die Abbildung der vorausgesetzten XML-basierten Subscriptionsprache auf einen dieser Ansätze als problematisch dar. Auf der Grundlage des Forwarding Table Ansatzes wurde schließlich in Kapitel 5 ein Konzept zur vollständigen XML-basierten Realisierung eines Ereignisfilters entwickelt. Zu diesem Konzept gehört auch die Möglichkeit der Integration in einen Broker und dieser wiederum in ein Brokernetzwerk. Die konkrete Architektur und deren Umsetzung wurde parallel zu dieser Arbeit von Geißler in [Gei10] entwickelt.

Ein wichtiger Schritt auf dem Weg zu einem funktionierenden Ereignisfilter bildete die Analyse und Interpretation der Subscriptionsprache. Hier bestand die Herausforderung neben der Konzeption darin, die essentiellen von den optionalen Features zu trennen, um so eine Rangfolge der Implementierung festlegen zu können. Außer den Grundkonzepten zum Aufbau der Subscriptions wie Receiver, ContentFilter oder Condition, konnten bis zur Fertigstellung dieser Arbeit somit die Funktionalitäten für Targetangaben und Attributnamen in Form von XPath-Ausdrücken, die Angabe von Quantoren und die Unterstützung verschiedener Operatoren umgesetzt werden.

Was sind die Kernideen dieser Arbeit?

Bei der Wahl einer geeigneten Systemarchitektur lag der Fokus darauf, eine parallele Ausführung in mehreren Threads zu ermöglichen, um den Filtervorgang beschleunigen zu können. Hier zeigten sich große Schwierigkeiten beim Zugriff auf gemeinsam genutzte Datenstrukturen, was zu einer Trennung in eine zentrale XMLEventFilter-Instanz und beliebig viele WorkingFilter-Instanzen mit je einer eigenen Datenstruktur führte. Das Zusammenfügen von Subscriptionsprache und Systemarchitektur resultierte in einer, für die Filterung, optimierten Datenstruktur, die sich bei Änderungen des Subscriptionformats anpassen lässt.

Der modulare Aufbau durch den Einsatz des OSGi-Komponentenmodells ermöglicht somit auch zur Laufzeit das Erweitern des Ereignisfilters, beispielsweise um zusätzliche Kommunikationsschnittstellen für den Empfang von Subscriptions oder Ereignisdaten. Durch die Spezifikation der Subscriptionsprache in Form von XML-Grammatiken hat sich zur Abbildung auf die interne Datenstruktur die Form des XML-Bindings mit der JAXB-

Technologie als sinnvoll erwiesen. Das Parsen von Ereignisdaten führt durch die fehlende Typisierung der Ereignisse und das Einlesen als Datenstrom zu größeren Herausforderungen. Zur Lösung dieses Problems wurde mit StAX eine Technologie gewählt, welche neben diesen Eigenschaften auch das Serialisieren der Ereignisdaten nach dem Filtervorgang unterstützt.

Eine spannende Herausforderung stellte die Umsetzung des in Kapitel 6.2 beschriebenen Integrationsszenarios dar, wobei neben kleineren Anpassungen am System auch zusätzliche Komponenten neu entwickelt werden mussten. Dazu zählte beispielsweise die Implementierung des XMLWriters, der die Aufgabe der Serialisierung von Ereignissen übernimmt. Die Bedeutung und der universell einsetzbare Charakter von Publish/Subscribe-Systemen stehen im Verlauf der Arbeit stets im Vordergrund. Dazu diente nicht nur die Testumgebung mit dem entwickelten Prototyp als Grundlage für Performancemessungen, sondern vor allem auch die Vorstellung des Integrationsszenarios. Dieses Szenario demonstriert praxisnah das Zusammenspiel von Ereignisfilter, Broker und weiteren Komponenten der unterschiedlichsten Technologien, wie zum Beispiel Silverlight-Benutzungsoberflächen oder Hardwarekomponenten mit einer Geräteintegrationsplattform.

7.2 Ausblick

Was ist nach dieser Arbeit offen geblieben?

Mit der Demonstration der Einsetzbarkeit stellte sich heraus, dass es notwendig ist, weitere Anforderungen umzusetzen. Es wurden einzelne Konzepte aus den Möglichkeiten der Subscriptionsprache analysiert, bisher jedoch nicht im Rahmen des Prototyps realisiert. Dazu zählt das Konzept der fein granularen Filterung, um Ereignisdaten während des Filtervorgangs manipulieren zu können (Abschnitt 5.2.3). Auch die Umsetzung einer Ereignistyphierarchie zur Minimierung der Subscriptionanzahl (Abschnitt 5.2.4) wurde im bisher implementierten Prototyp noch nicht berücksichtigt. Im Zusammenhang mit der Brokerimplementierung und dem Einsatz in einem Brokernetzwerk verspricht das analysierte Advertisementkonzept (Abschnitt 5.1.2) einen signifikanten Performancezuwachs durch die optimale Verteilung der Subscriptions auf die einzelnen Broker.

Wie sollte in eventuell folgenden Arbeiten an dem Thema weitergearbeitet werden?

Durch die Konzeption und die anschließende Validierung im Rahmen von Implementierung und Performanceanalyse ergaben sich weitere Fragen, die in fortführenden Arbeiten betrachtet werden können. Derzeit erfolgt die Ausgabe des Ereignisfilters stets in Einzelereignissen, auch wenn die Ereignisdaten in Form von Nachrichten mit vielen Ereignissen am Ereignisfilter eintreffen. Hier wäre interessant zu untersuchen, ob eine Pufferung der Ereignisse für eine gemeinsame Weiterleitung von Ereignissen an den gleichen Empfänger sinnvoll ist oder das einzelne Versenden unmittelbar nach der Filterung bereits die optimale Variante darstellt.

Wie aus der Performanceanalyse in Abschnitt 6.3.4 hervorgeht, kann in weiteren Arbeiten untersucht werden, inwiefern die Datenstruktur optimiert werden kann, um den exponentiellen Anstieg der Laufzeit pro Ereignis bei exponentiell ansteigender Subscriptionanzahl in Grenzen zu halten. Dafür sind detaillierte Analysen notwendig, um die entsprechenden Ansatzpunkte zu finden.

Eine interessante Forschungsfrage wäre der Einfluss einer Ereignisverarbeitungseinheit auf die Filterleistung. Hierfür ist es vorstellbar, den Ereignisfilter mit einer CEP-Engine (z.B. Esper) zu kombinieren, um auch komplexe Ereignisse oder Korrelationen zwischen Ereignissen berücksichtigen zu können. Dies würde außerdem zu einer Verlagerung der Ereignisverarbeitung vom Empfänger auf den Broker führen, da dieser in vielen Fällen eine größere Rechenleistung bereitstellen kann. Nach ersten Ideen ergeben sich daraus zwei wesentliche Vermutungen:

- Dieser Ansatz spart Rechenzeit, da potentiell viele Interessenten die Daten der gleichen komplexen Ereignisse benötigen.
- Der Empfänger würde somit direkt die komplexen Ereignisse erhalten und muss sich nicht mehr darum kümmern, diese aus dem Strom von Einzelereignissen herauszufiltern, was zu schlankeren Systemen führt.

ANHANG

Anhang 1 OSGi-Bundles für EventBroker-Implementierung

Projekt / OSGi-Bundle	Codezeilen	Klassen	Methoden
AdiwaBindings			
Enthält die XML-Schemadefinitionen und die Binding-Declarations der ADiWa-XML-Sprache zur Abbildung von Subscriptions.			
AdiwaCommonInterfaces	1482	40	140
Dieses Bundle bildet die gemeinsamen Schnittstellen zwischen Ereignisfilter und Brokerimplementierung ab.			
AdiwaCommonLibs			
Enthält gemeinsam genutzte Java-Bibliotheken.			
XMLEventFilter	2637	39	162
Zentrale Komponente der Ereignisfilterimplementierung.			
XMLEventFilterData	2555	49	305
Dieses Bundle enthält sämtliche Datenstrukturen des Ereignisfilters sowie des Brokers.			
XMLEventFilterHttpServer	724	9	50
Implementiert einen HTTP-Server zur Kommunikation zwischen Ereignisfilter und Subscriber / Publisher.			
XMLEventGenerator	591	6	14
Komponente zur Generierung und den Versand von Ereignisdaten.			
XMLSubscriptionGenerator	407	6	13
Komponente zur Generierung und den Versand von Subscriptions.			

Anhang 2 XML-Schema der ADiWa-Subscriptionsprache

- subscriptionFormat.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.adiwa.net/subscription/1.0"
       xmlns:tns="http://www.adiwa.net/subscription/1.0"
       xmlns:fi="http://www.adiwa.net/filter/1.0"
       elementFormDefault="qualified">

  <import namespace="http://www.adiwa.net/filter/1.0"
     schemaLocation="./filterFormat.xsd"/>

  <element name="subscriptionList" type="tns:SubscriptionListType"/>
  <complexType name="SubscriptionListType">
    <sequence>
      <element name="subscription" type="tns:SubscriptionType" minOccurs="0"
    maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  <complexType name="SubscriptionType">
    <sequence>
      <!-- something that can be used to distinguish subscribers -->
      <element name="subscriberId" type="string" minOccurs="0" maxOccurs="1"/>
      <!-- in what format was the subscription originally submitted?
       e.g. JMS, WS-Notification, ASL (ADiWa Subscription Language) -->
      <element name="subscriptionMethod" type="string" minOccurs="0"
       maxOccurs="1"/>
      <!-- -->
      <element name="receiver" type="tns:ReceiverType" minOccurs="1"
       maxOccurs="1"/>
      <!-- date and time when the subscription was recorded (or renewed) in the
       system -->
      <element name="recordTime" type="dateTime" minOccurs="0" maxOccurs="1"/>
      <!-- specifies after how many seconds the subscription shall be be
       discarded -->
      <element name="timeoutInSeconds" type="integer" minOccurs="0"
       maxOccurs="1"/>
      <!-- holds security-related information such as subscriber authentication
       tokens -->
      <element name="securityInformation" type="tns:SecurityInformationType"
       minOccurs="0" maxOccurs="1"/>
       <!-- the filters that determine which events are subscribed -->
      <element name="filterList" type="fi:FilterListType" minOccurs="1"
       maxOccurs="1" />
    </sequence>
    <!-- unique identifier of this subscription -->
    <attribute name="id" type="string"/>
  </complexType>
  <!-- TODO to be defined -->
  <complexType name="SecurityInformationType">
    <sequence>
      <element name="dummy" type="string"/>
    </sequence>
  </complexType>
  <complexType name="ReceiverType">
    <sequence>
      <!-- the form in which the events shall be delivered (HTTP POST, WS-
       Notification Web Service response etc.) -->
      <element name="callbackMethod" type="string" minOccurs="1" maxOccurs="1"/>
      <!-- the callback address where events matching this subscription shall be
       delivered -->
      <element name="callbackAddress" type="anyURI" minOccurs="1"
       maxOccurs="1"/>
    </sequence>
```

```
  </complexType>
</schema>
```

Listing 10: Schemadefinition der ADiWa-Subscriptionsprache, Teil 1 (Quelle: [Gru10])

- filterFormat.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.adiwa.net/filter/1.0"
  xmlns:tns="http://www.adiwa.net/filter/1.0"
  elementFormDefault="qualified">

  <!-- as root element, we have "filterList" -->
  <element name="filterList" type="tns:FilterListType"/>
  <!-- the root element can contain any number of contentFilter and conceptFilter
       elements -->
  <complexType name="FilterListType">
    <sequence>
      <element name="contentFilter" type="tns:ContentFilterType" minOccurs="0"
       maxOccurs="unbounded" />
      <element name="conceptFilter" type="tns:ConceptFilterType" minOccurs="0"
       maxOccurs="unbounded" />
    </sequence>
  </complexType>
  <!-- a contentFilter element can contain any number of condition elements and
       other nested contentFilter elements -->
  <complexType name="ContentFilterType">
    <sequence>
      <element name="condition" type="tns:ConditionType"  minOccurs="0"
       maxOccurs="unbounded" />
      <element name="contentFilter" type="tns:ContentFilterType"  minOccurs="0"
       maxOccurs="unbounded" />
    </sequence>
    <!--  -->
    <attribute name="target" type="string" use="required"/>
    <!-- we need to discuss whether visibleAttributes should be supported. -->
    <attribute name="visibleAttributes" type="string"/>
    <attribute name="id" type="string"/>
  </complexType>
  <complexType name="ConditionType">
    <sequence>
      <element name="value" type="string" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="quantifier" use="optional">
      <simpleType>
        <restriction base="NMTOKEN">
          <enumeration value="any"/>
          <enumeration value="all"/>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="attribute" type="string" use="required"/>
    <attribute name="operatorId" type="string" use="required"/>
  </complexType>
  <!-- TODO -->
  <complexType name="ConceptFilterType">
    <!-- to be specified -->
    <sequence>
      <element name="dummy" type="string"/>
    </sequence>
  </complexType>
</schema>
```

Listing 11: Schemadefinition der ADiWa-Subscriptionsprache, Teil 2 (Quelle: [Gru10])

Anhang 3 Binding-Declarations zur Abbildung der ADiWa-Subscriptionsprache

- subscriptionBindings.jxb

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxb:bindings version="2.0"
       schemaLocation="subscriptionFormat.xsd"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:jaxb="http://java.sun.com/xml/ns/jaxb">

  <!-- Customizing JAXB Bindings -->
  <!-- Defines the customization value which is the collection type for the
       <property> Declaration"collectionType" is a class name that implements
       java.util.List -->
  <jaxb:bindings node="//xs:element[@name='subscription']">
    <!-- This modifying of collectionType prohibits lazy loading! -->
    <jaxb:property
       collectionType="xmleventfilterdata.types.SubscriptionContainer" />
    <jaxb:class name="SubscriptionType"
       ref="xmleventfilterdata.subscription.Subscription" />
  </jaxb:bindings>
  <!-- end -->

  <!-- Specify the name of the value class that is provided outside the schema
       compiler. This customization causes a schema compiler to refer to this
       external class, as opposed to generate a definition. -->
  <jaxb:bindings node="//xs:complexType[@name='ReceiverType']">
    <jaxb:class name="ReceiverType"
       ref="xmleventfilterdata.subscription.Receiver" />
  </jaxb:bindings>
  <jaxb:bindings node="//xs:complexType[@name='SecurityInformationType']">
    <jaxb:class name="SecurityInformationType"
       ref="xmleventfilterdata.subscription.SecurityInformation" />
  </jaxb:bindings>
  <jaxb:bindings node="//xs:complexType[@name='SubscriptionType']">
    <jaxb:class name="SubscriptionType"
       ref="xmleventfilterdata.subscription.Subscription" />
  </jaxb:bindings>
  <!-- end -->

  <!-- Customization of target package name. -->
  <jaxb:schemaBindings>
    <jaxb:package name = "xmleventfilterdata.subscription.gen" />
  </jaxb:schemaBindings>
  <!-- end -->
</jaxb:bindings>
```

Listing 12: Binding Declarations zur Abbildung von Subscriptions, Teil 1

- filterBindings.jxb

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxb:bindings version="2.0"
       schemaLocation="filterFormat.xsd"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:jaxb="http://java.sun.com/xml/ns/jaxb">

  <!-- Customizing JAXB Bindings -->
  <!-- Defines the customization value which is the collection type for the
       <property> Declaration"collectionType" is a class name that implements
       java.util.List -->
  <jaxb:bindings node="//xs:complexType[@name='FilterListType']
       //xs:element[@type='tns:ContentFilterType']">
```

```
  <!-- This modifying of collectionType prohibits lazy loading! -->
  <jaxb:property collectionType="xmleventfilterdata.types.FilterContainer" />
  <jaxb:class name="ContentFilterType"
     ref="xmleventfilterdata.filter.ContentFilter" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='FilterListType']
     //xs:element[@type='tns:ConceptFilterType']">
  <!-- This modifying of collectionType prohibits lazy loading! -->
  <jaxb:property collectionType="xmleventfilterdata.types.FilterContainer" />
  <jaxb:class name="ConceptFilterType"
     ref="xmleventfilterdata.filter.ConceptFilter" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='ContentFilterType']
     //xs:element[@type='tns:ContentFilterType']">
  <!-- This modifying of collectionType prohibits lazy loading! -->
  <jaxb:property collectionType="xmleventfilterdata.types.FilterContainer" />
  <jaxb:class name="ContentFilterType"
     ref="xmleventfilterdata.filter.ContentFilter" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='ContentFilterType']
     //xs:element[@type='tns:ConditionType']">
  <!-- This modifying of collectionType prohibits lazy loading! -->
  <jaxb:property collectionType="xmleventfilterdata.types.ConditionContainer" />
  <jaxb:class name="ConditionType"
     ref="xmleventfilterdata.filter.Condition" />
</jaxb:bindings>
<jaxb:bindings node="//xs:element[@name='value']">
  <!-- This modifying of collectionType prohibits lazy loading! -->
  <jaxb:property collectionType="xmleventfilterdata.types.ValueContainer" />
</jaxb:bindings>
<!-- end -->

<!-- Specify the implementation class. This customization only impacts the
     return value for classNames factory method. -->
<jaxb:bindings node="//xs:complexType[@name='ContentFilterType']">
  <jaxb:class name="ContentFilterType"
     implClass="xmleventfilterdata.filter.ContentFilter" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='ConceptFilterType']">
  <jaxb:class name="ConceptFilterType"
     implClass="xmleventfilterdata.filter.ConceptFilter" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='ConditionType']">
  <jaxb:class name="ConditionType"
     implClass="xmleventfilterdata.filter.Condition" />
</jaxb:bindings>
<jaxb:bindings node="//xs:complexType[@name='FilterListType']">
  <jaxb:class name="FilterListType"
     implClass="xmleventfilterdata.filter.FilterList" />
</jaxb:bindings>
<!-- end -->

<!-- Customization of a binding of an XML schema element to its Java
     representation as an enum type. -->
<jaxb:bindings node="//xs:attribute[@name='quantifier']/xs:simpleType">
  <jaxb:typesafeEnumClass ref="xmleventfilterdata.types.Quantifier">
    <jaxb:typesafeEnumMember value="any" name="any" />
    <jaxb:typesafeEnumMember value="all" name="all" />
  </jaxb:typesafeEnumClass>
</jaxb:bindings>
<!-- end -->

<!-- Customization of a base type for a JAXB property. -->
<jaxb:bindings node="//xs:attribute[@name='attribute']">
  <jaxb:property>
    <jaxb:baseType>
      <jaxb:javaType name="xmleventfilterdata.internal.SubscriptionAttribute"
```

```
              parseMethod="new"
              printMethod="toString" />
      </jaxb:baseType>
    </jaxb:property>
  </jaxb:bindings>
  <jaxb:bindings node="//xs:attribute[@name='operatorId']">
    <jaxb:property>
      <jaxb:baseType>
        <jaxb:javaType name="xmleventfilterdata.internal.Operator"
              parseMethod="new"
              printMethod="toString" />
      </jaxb:baseType>
    </jaxb:property>
  </jaxb:bindings>
  <jaxb:bindings node="//xs:attribute[@name='target']">
    <jaxb:property>
      <jaxb:baseType>
        <jaxb:javaType name="xmleventfilterdata.internal.Target"
              parseMethod="new"
              printMethod="toString" />
      </jaxb:baseType>
    </jaxb:property>
  </jaxb:bindings>
  <!-- end -->

  <!-- Customization of target package name. -->
  <jaxb:schemaBindings>
    <jaxb:package name = "xmleventfilterdata.filter.gen" />
  </jaxb:schemaBindings>
  <!-- end -->
</jaxb:bindings>
```

Listing 13: Binding Declarations zur Abbildung von Subscriptions, Teil 2

Anhang 4 UML-Klassendiagramm der Subscriptionstruktur

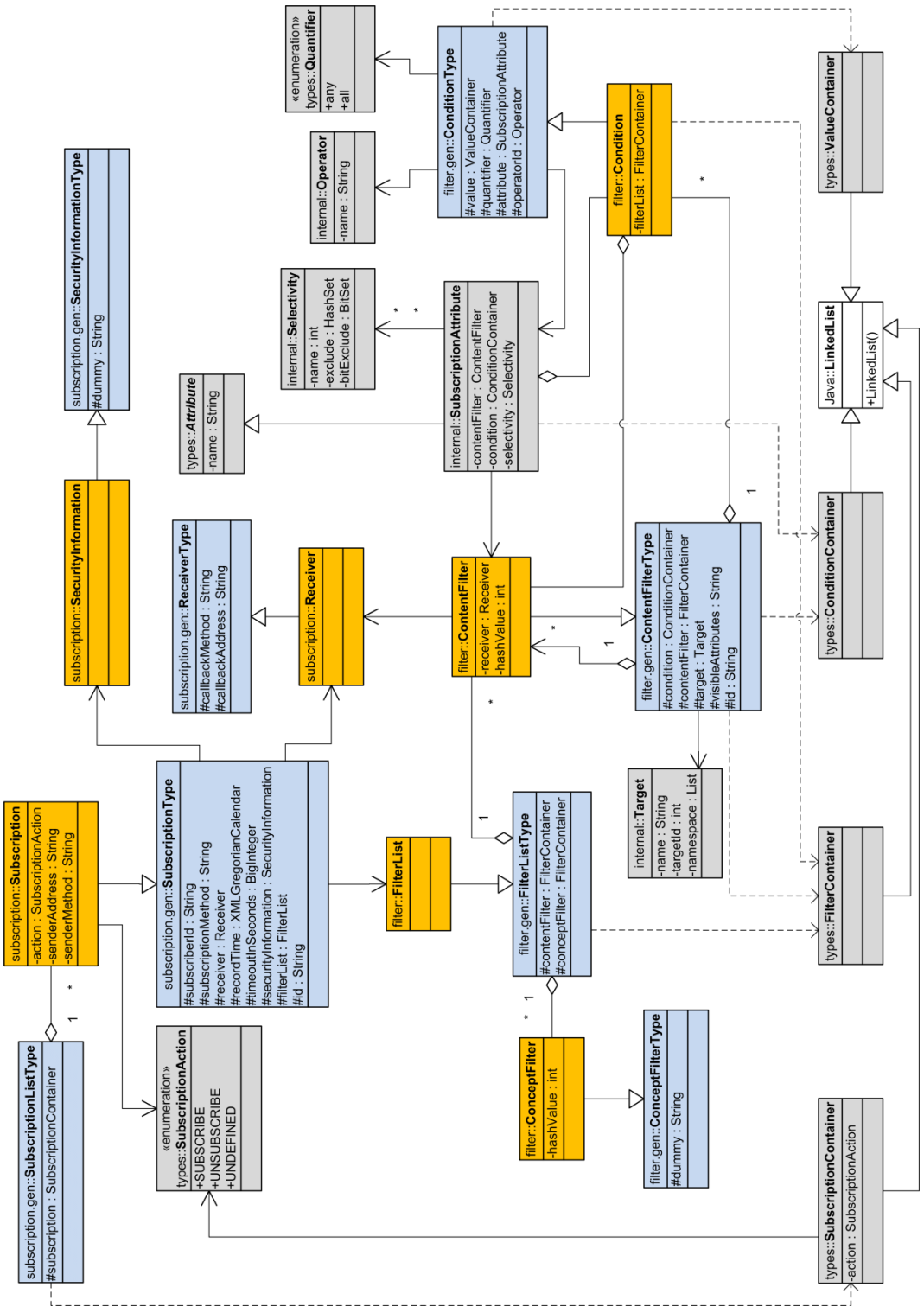


Abbildung 42: UML-Klassendiagramm zur Modellierung von Subscriptions (Quelle: eigene Darstellung)

Anhang 5 Rückgabewerte der checkTarget-Methode

Possible *Integer* results of *checkTarget* -method in *AttributeIndex.java*

XML Event (ParseStack) \ Subscription (TargetStack)	//Event	//ObjectEvent	//Event/ObjectEvent	//Event[@Type="ObjectEvent"]	//Event[Target@Type="ObjectEvent"]	//Event[Type="ObjectEvent"]	//Event[Target/Type="ObjectEvent"]	//Event[target/type@id='ObjectEvent']	//Event[target/type/class@id='ObjectEvent']
/EPCIS/Body/Event	0	null	1	1	2	2	3	3	4
/EPCIS/Body/Event/ObjectEvent	null	0	0	null	2	2	3	3	4
/EPCIS/Body/Event/Type	null	null	null	null	2	1	3	3	4
/EPCIS/Body/Event/@Type="ObjectEvent"	null	null	null	0	2	2	3	3	4
/EPCIS/Body/Event/@Class="Test"	null	null	null	1	2	2	3	3	4
/EPCIS/Body/Event/Target/@Type="ObjectEvent"	null	null	null	null	-1	2	1	2	3
/EPCIS/Body/Event/Type="ObjectEvent"	null	null	null	null	2	-1	3	3	4
/EPCIS/Body/Event/Target/Type="ObjectEvent"	null	null	null	null	1	2	-2	2	3
/EPCIS/Body/Event/Target/Type/@id="ObjectEvent"	null	null	null	null	null	2	1	-2	2
/EPCIS/Body/Event/Target/Type/Class/@id="ObjectEvent"	null	null	null	null	null	2	1	null	-3

possible values:	
< 0	target is matched by parser; event starts in a higher level of xml tree (defined by value)
0	target is matched by parser; current node is root of event
> 0	target is not matched, but root of event is defined in a lower level of xml tree along the current path
null	target is not matched by current node and any node in lower level of xml tree along the current path

Tabelle 3: Mögliche Rückgabewerte der checkTarget-Methode anhand einiger Beispiele (Quelle: eigene Darstellung)

Anhang 6 Beispiel einer Subscription

```
<?xml version="1.0" encoding="UTF-8"?>
<subscriptionList xmlns="http://www.adiwa.net/subscription/1.0"
       xmlns:aff="http://www.adiwa.net/filter/1.0">
  <subscription>
    <receiver>
      <callbackMethod>HTTP</callbackMethod>
      <callbackAddress>http://localhost:4001/submit</callbackAddress>
    </receiver>
    <filterList>
      <aff:contentFilter target="//ObjectEvent">
        <aff:condition operatorId="string-equal" attribute="bizLocation/id">
          <aff:value>urn:com:location:area-01</aff:value>
        </aff:condition>
        <aff:condition operatorId="attribute-exist" attribute="readPoint/id" />
        <aff:condition operatorId="string-match" attribute="bizLocation/id">
          <aff:value>urn:com:*</aff:value>
        </aff:condition>
      </aff:contentFilter>
    </filterList>
  </subscription>
  <subscription>
    <receiver>
      <callbackMethod>HTTPS</callbackMethod>
      <callbackAddress>http://localhost:4002/submit</callbackAddress>
    </receiver>
    <filterList>
      <aff:contentFilter target="//ObjectEvent">
        <aff:condition operatorId="string-equal" attribute="bizLocation/id">
          <aff:value>urn:com:location:area-01</aff:value>
        </aff:condition>
        <aff:condition operatorId="attribute-exist" attribute="readPoint/id" />
      </aff:contentFilter>
    </filterList>
  </subscription>
  <subscription>
    <receiver>
      <callbackMethod>HTTPS</callbackMethod>
      <callbackAddress>http://localhost:4002/submit</callbackAddress>
    </receiver>
    <filterList>
      <aff:contentFilter target="adiwaEvent[//eventType/name='SmartMeterEvent']">
        <aff:condition operatorId="attribute-exist" attribute="ns2:current" />
      </aff:contentFilter>
    </filterList>
  </subscription>
</subscriptionList>
```

Listing 14: Beispiel einer Subscription

Anhang 7 Beispiel einer Nachricht mit drei Ereignissen

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<epcis:EPCISDocument xmlns:epcis="urn:epcglobal:epcis:xsd:1"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <EPCISBody>
    <EventList>
      <ObjectEvent>
        <eventTime>2010-04-12T14:04:05Z</eventTime>
        <eventTimeZoneOffset>+02:00</eventTimeZoneOffset>
        <epcList>
          <epc>urn:epc:id:sgtin:1000002.1670</epc>
        </epcList>
        <action>OBSERVE</action>
        <bizStep>urn:com:bizstep:receiving</bizStep>
        <disposition>urn:com:disposition:active</disposition>
        <readPoint>
          <id>urn:com:readpoint:warehouse-01:entrance-01</id>
        </readPoint>
        <bizLocation>
          <id>urn:com:location:warehouse-01</id>
        </bizLocation>
      </ObjectEvent>
      <ObjectEvent>
        <eventTime>2010-04-12T14:05:05Z</eventTime>
        <eventTimeZoneOffset>+02:00</eventTimeZoneOffset>
        <epcList>
          <epc>urn:epc:id:sgtin:1000001.3246</epc>
        </epcList>
        <action>OBSERVE</action>
        <bizStep>urn:com:bizstep:receiving</bizStep>
        <disposition>urn:com:disposition:active</disposition>
        <readPoint>
          <id>urn:com:readpoint:area-01:entrance-01</id>
        </readPoint>
        <bizLocation>
          <id>urn:com:location:area-01</id>
        </bizLocation>
      </ObjectEvent>
      <ObjectEvent>
        <eventTime>2010-04-12T14:06:06Z</eventTime>
        <eventTimeZoneOffset>+02:00</eventTimeZoneOffset>
        <epcList>
          <epc>urn:epc:id:sgtin:1000002.1670</epc>
        </epcList>
        <action>OBSERVE</action>
        <bizStep>urn:com:bizstep:producing</bizStep>
        <disposition>urn:com:disposition:readyforproduction</disposition>
        <readPoint>
          <id>urn:com:readpoint:plant-01:entrance-01</id>
        </readPoint>
        <bizLocation>
          <id>urn:com:location:plant-01</id>
        </bizLocation>
      </ObjectEvent>
    </EventList>
  </EPCISBody>
</epcis:EPCISDocument>
```

Listing 15: Beispiel einer Publish-Nachricht mit 3 Ereignissen

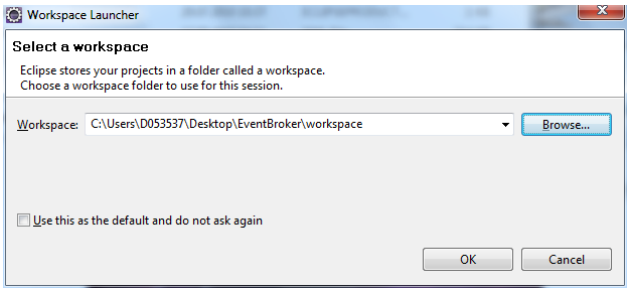
Anhang 8 Installation und Konfiguration der Testumgebung

Die folgende Konfigurationsanleitung beschreibt die Einrichtung einer Testumgebung für den in Kapitel 6.1 vorgestellten EventBroker. Alle genannten Softwarekomponenten und Einstellungen beziehen sich dabei auf die Verwendung mit dem Betriebssystem Windows 7 in der 64-Bit-Version mit mindestens 2 GB Arbeitsspeicher.

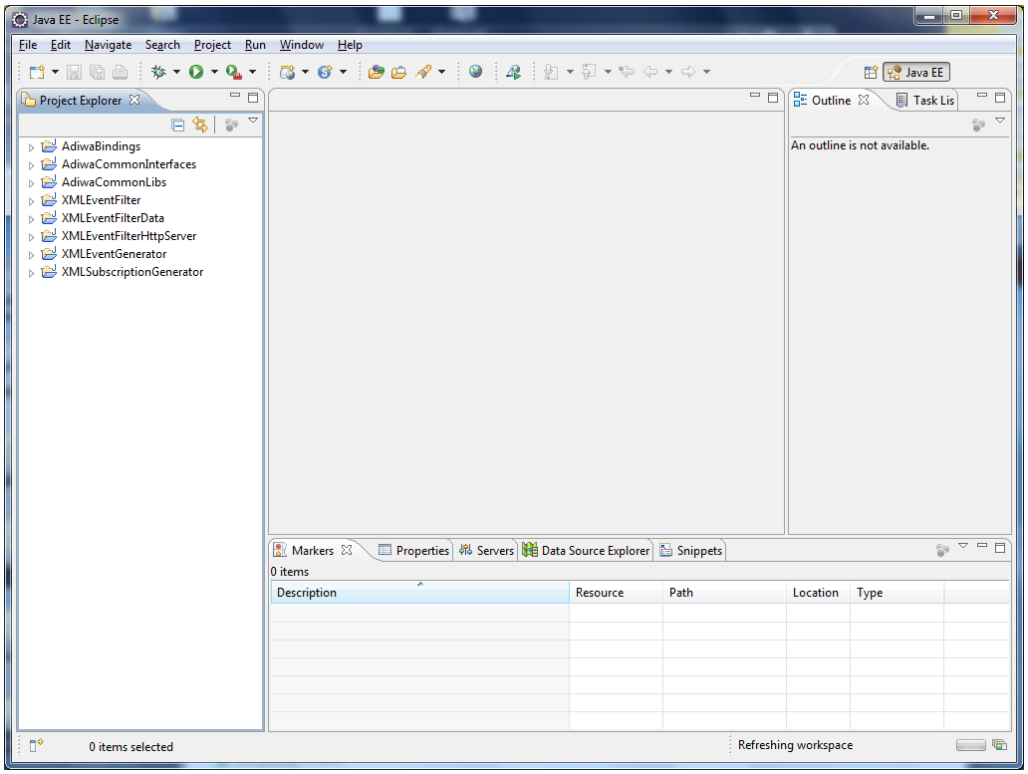
- Java SE Development Kit 6u21 (JDK 6 Update 21) installieren
  DVD: software\java\jdk-6u21-windows-x64.exe
  WWW: http://www.oracle.com/technetwork/java/javase/downloads/
- Eclipse IDE for Java EE Developers (Helios SR1) entpacken
  DVD: software\eclipse\eclipse-jee-helios-SR1-win32.zip
  WWW: http://www.eclipse.org/downloads/

- EventBroker entpacken (enthält bereits Eclipse)
  DVD: EventBroker\EventBroker_eclipse+workspace.zip

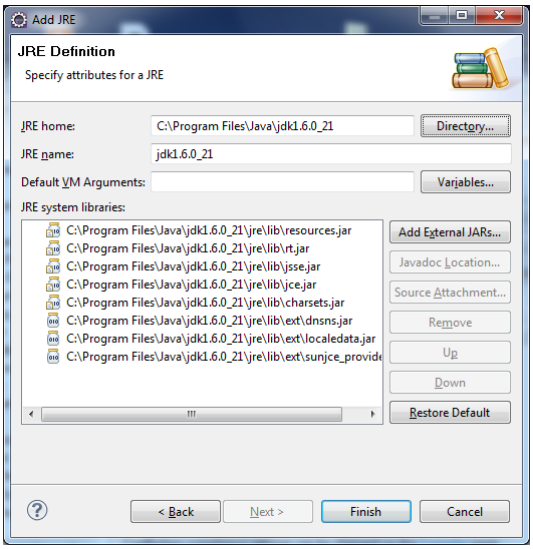
- Eclipse starten und mit entpacktem Workspace verknüpfen



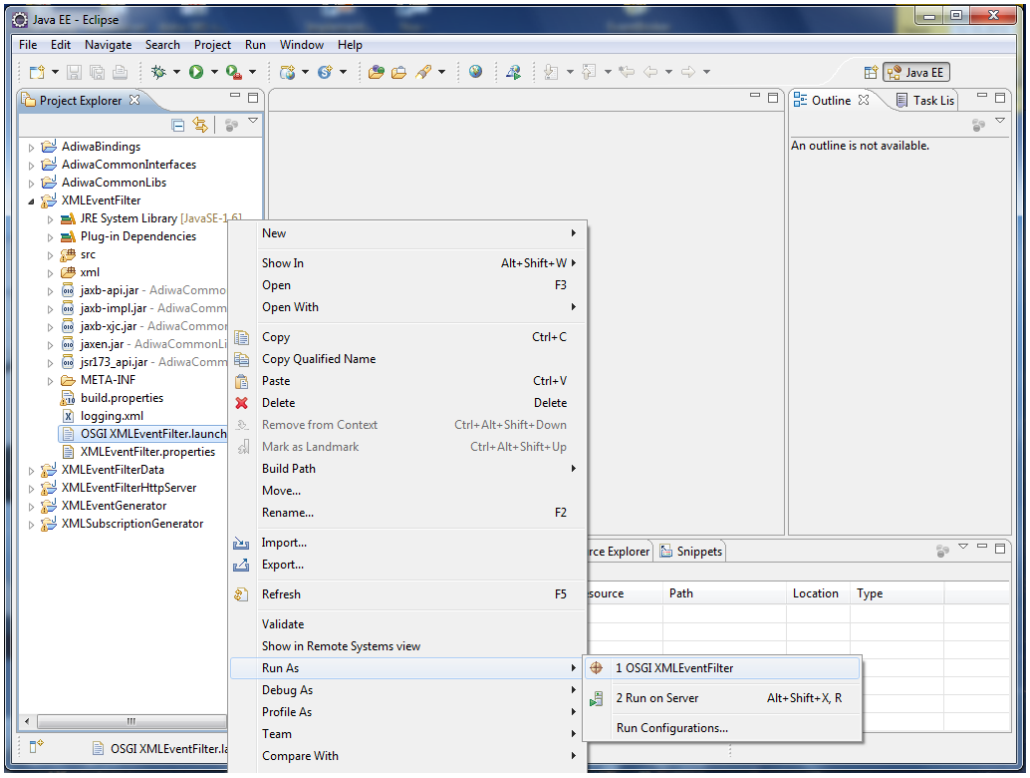
- Alle Projekte aus dem Workspace importieren
  Eclipse: > File > Import > General > Existing Projects into Workspace



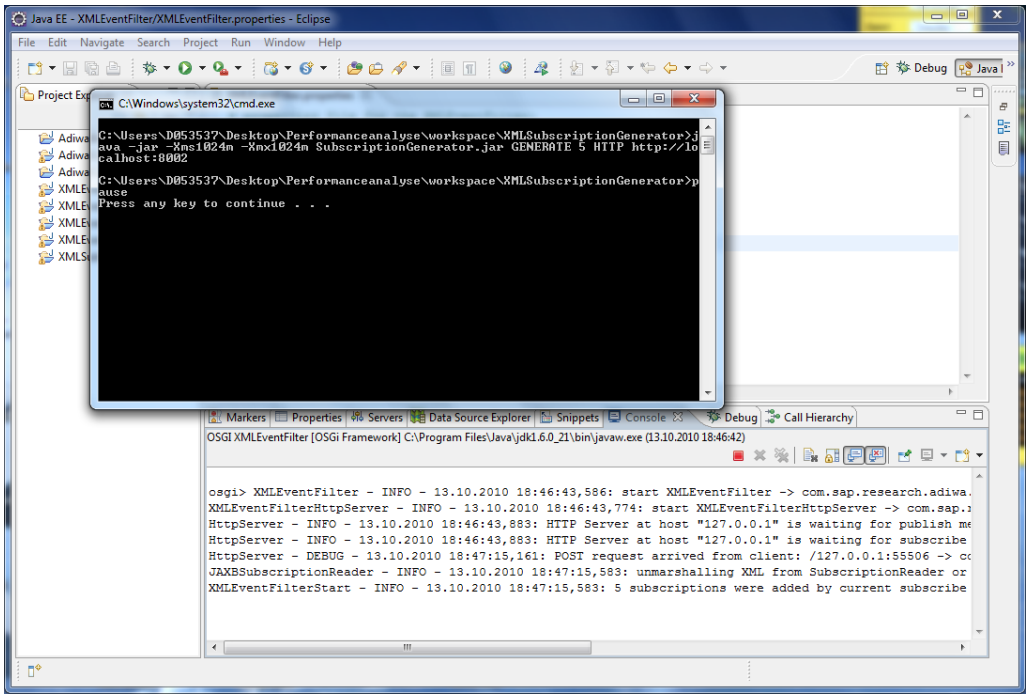
- Java Runtime Environment (JRE) konfigurieren
  Eclipse: > Window > Preferences > Java > Installed JREs
  ➔ Auswählen der installierten 64-Bit Version



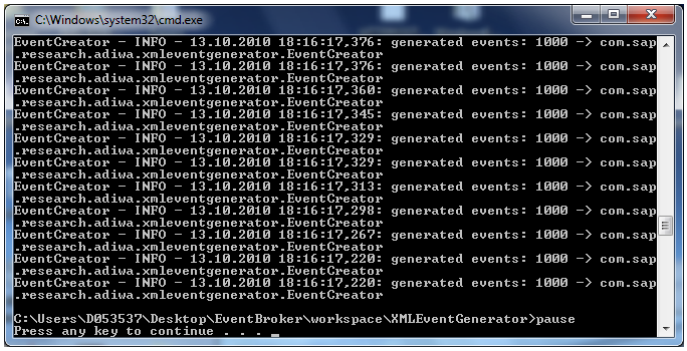
- OSGi-Konfiguration des EventBrokers starten (Die OSGi-Bundles für Publisher und Subscriber als interne Clients sind standardmäßig integriert, aber nicht gestartet)



- Senden einer gewünschten Anzahl an Subscriptions (vorkonfiguriert: 5) mit Hilfe der beiliegenden Batch-Dateien über eine HTTP-Verbindung
  …\workspace\XMLSubscriptionGenerator\SubscriptionGenerator_local.bat



- Senden einer gewünschten Anzahl an Ereignissen (vorkonfiguriert: 100 Verbindungen mit je 1000 Ereignissen) mit Hilfe der beiliegenden Batch-Dateien über eine HTTP-Verbindung
  …\workspace\XMLEventGenerator\EventGenerator_multiple_connections_local.bat



LITERATURVERZEICHNIS

[ADi10] ADiWa. 2010.
ADiWa - Allianz Digitaler Warenfluss. Zugriff am 15.04.2010.
http://www.adiwa.net/index.php?id=125

[Alt00] Altinel, Mehmet und Franklin, Micheal J. 2000.
Efficient Filtering of XML Documents for Selective Dissemination of Information.
http://www.cs.berkeley.edu/~franklin/Papers/XFilterVLDB00.pdf

[Ash02] Ashayer, Ghazaleh, Yau, Hubert Ka und Jacobsen, H.-Arno. 2002.
Predicate Matching and Subscription Matching in Publish/Subscribe Systems. IEEE Computer Society, 2002. S. 539-548.
http://cchen1.csie.ntust.edu.tw/teaching/computer-network/Publish%20Subscribe%20Systems%20-%20University%20of%20Toronto.pdf

[Ast04] Astley, Mark, et al. 2004.
Achieving scalability and throughput in a publish/subscribe system.
http://www.research.ibm.com/people/s/sbhola/myhome_files/scalability_rc23103.pdf

[Atm10] Atmel Corporation. 2010.
ATmega128RFA1. Zugriff am 20.09.2010.
http://www.atmel.com/dyn/products/product_card.asp?part_id=4692

[Cam01] Campailla, Alexis, et al. 2001.
Efficient filtering in publish-subscribe systems using binary decision diagrams. IEEE Computer Society. S. 443-452.
http://portal.acm.org/citation.cfm?id=381473.381519&type=series

[Car01] Carzaniga, Antonio und Wolf, Alexander L. 2001.
Fast Forwarding for Content-Based Networking.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.8490&rep=rep1&type=pdf

[Car03] Carzaniga, Antonio und Wolf, Alexander L. 2003.
Forwarding in a Content-Based Network.
http://portal.acm.org/citation.cfm?id=863975

[Car011] Carzaniga, Antonio, Rosenblum, David S. und Wolf, Alexander L. 2001.
Design and evaluation of a wide-area event notification service.
http://www.computer.org/portal/web/csdl/doi/10.1109/FITS.2003.1264940

[Cas02] Castro, Miguel, et al. 2002.
SCRIBE: A large-scale and decentralized application-level multicast infrastructure.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.3271&rep=rep1&type=pdf

[Dav10] David Luckham, Roy Schulte.
Event Processing Glossary - Version 1.1. Zugriff am 10.06.2010.
http://www.ep-ts.com/component/option,com_docman/task,doc_download/gid,66/Itemid,84/

[dre10] dresden elektronik ingenieurtechnik gmbh.
dresden elektronik - Funkmodul mit Single-Chip-Lösung deRFmega128-22A00 28182. Zugriff am 20.09.2010.
http://www.dresden-elektronik.de/shop/prod127.html

[Eil10] Eilebrecht, Karl und Starke, Gernot. 2010.
Patterns kompakt: Entwurfsmuster für effektive Software-entwicklung. Springer Verlag. ISBN 978-3827415912.

[EPC07] EPCglobal Inc. 2007.
EPC Information Services (EPCIS) Version 1.0.1 Specification.
http://www.epcglobalinc.org/standards/epcis/epcis_1_0_1-standard-20070921.pdf

[Esp09] EsperTech Inc. 2009.
Esper Reference Documentation 3.0.0.
http://esper.codehaus.org/esper-3.0.0/doc/reference/en/pdf/esper_reference.pdf

[Eug03] Eugster, Patrick Th., et al. 2003.
The many faces of publish/subscribe.
http://doi.acm.org/10.1145/857076.857078

[Gei10] Geißler, Marco. 2010.
Basisdienste für ein sicheres, XML-basiertes publish/subscribe-System. *Diplomarbeit.*

[Gru10] Grummt, Eberhard. 2010.
Filter-Formate (Version 0.1). SAP internes Arbeitsdokument.
SAP intern

[IBM04] IBM. 2004.
Web Services Eventing.
http://www.ibm.com/developerworks/webservices/library/specification/ws-eventing/

[IBM09] IBM. 2009
WS-Notification: Übersicht. *WebSphere Application Server Version 7.0.* Zugriff am 10.04.2010.
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.pmc.express.iseries.doc/concepts/cjwsn_overview.html

[Int05] Internet.com - The Network for Technology Professionals. 2005.
Event-Driven Architecture vs. Publish-Subscribe Systems. Zugriff am 02.06.2010.
http://www.developer.com/print.php/3499031

[Lan10] Lane, Jason. 2010.
Multi-Core Support in Windows 7. *IT Expert Voice.* Zugriff am 10.10.2010.
http://itexpertvoice.com/home/multi-core-support-in-windows-7/

[Mic10] Microsoft Corporation. 2010.
Publish/Subscribe. *MSDN Library.*
http://msdn.microsoft.com/en-us/library/ff649664.aspx

[Mil10] Millard, Peter, Saint-Andre, Pete und Meijer, Ralph. 2010.
XEP-0060: Publish-Subscribe. Zugriff am 22.09.2010.
http://xmpp.org/extensions/xep-0060.html

[Müh02] Mühl, Gero. 2002.
Large-scale content-based publish/subscribe systems.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.9995&rep=rep1&type=pdf

[OAS06] OASIS. 2006.
Web Services Base Notification 1.3.
http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf

[Pie02] Pietzuch, Peter R. und Bacon, Jean. 2002.
Hermes: A Distributed Event-Based Middleware Architecture. IEEE Computer Society, 2002. S. 611-618.
http://www.doc.ic.ac.uk/~prp/manager/doc/prp-debs2002-hermes.pdf

[Pie07] Pietzuch, Peter, et al. 2007.
Towards a common API for publish/subscribe. ACM, 2007. S. 152-157.
http://www.cl.cam.ac.uk/research/srg/opera/publications/papers/peter-debs07.pdf

[SAP10] SAP Research Center Dresden.
Energy Management for Manufacturing. *CEC Dresden Wiki.* Zugriff am 01.10.2010.
SAP intern

[SAP101] SAP Research Center Dresden.
intern. *CEC Dresden Wiki.* Zugriff am 29.09.2010.
SAP intern.

[Sch09] Scholz, Michael und Niedermeier, Stephan. 2009.
Java und XML. Galileo Press.
ISBN 978-3-8362-1308-0.

[Slo02] Slominski, Aleksander, et al. 2002.
XEVENTS/XMESSAGES: Application Events and Messaging Framework for Grid.
http://www.extreme.indiana.edu/xgws/papers/xevents_xmessages_tr.pdf

[sue10] sueddeutsche.de.
10.000 Einzelteile in einem Auto - viel Potenzial für Fehler. Zugriff am 20.04.2010.
http://www.sueddeutsche.de/automobil/608/504816/text/4/

[Sun06] Sun Microsystems, Inc. 2006.
The Java Architecture for XML Binding (JAXB) 2.1.

[Uhs09] Uhse, Fabian. 2009.
Einsatz von kontext-sensitiven Geräten in intelligenter Umgebung am Beispiel eines Hausautomationssystems. *Masterarbeit.*

[Vol101] Volkswagen AG. 2010.
B2B-Plattform. Zugriff am 20.04.2010.
http://www.vwgroupsupply.com/b2b/vwb2b_folder/supply2public/de/zusammenarbeit/beschaffung.html

[Vol09] Volkswagen AG. 2009.
Geschäftsbericht 2009. Zugriff am 20.04.2010.
http://geschaeftsbericht2009.volkswagenag.com/

[Vol102] Volkswagen AG. 2010.
Konzern. Zugriff am 20.04.2010.
http://www.volkswagenag.com/vwag/vwcorp/content/de/the_group.html

[W3C06] W3C. 2006.
Web Services Eventing (WS-Eventing). Zugriff am 10.01.2010.
http://www.w3.org/Submission/WS-Eventing/

[W3C07] W3C. 2007.
XML Path Language (XPath) Version 2.0. Zugriff am 05.10.2010.
http://www.w3.org/TR/xpath20/

[Wüt08] Wütherich, Gerd, et al. 2008.
Die OSGI Service Platform. dpunkt.verlag.
ISBN 978-3-89864-457-0.

[Yan10] Yanlei Diao, Michael J. Franklin. 2010.
Publish/Subscribe over Streams.
http://www.cs.umass.edu/~yanlei/publications/StreamPubSub.pdf

[Yan101] Yanlei Diao, Michael J. Franklin. 2010.
XML Publish/Subscribe.
http://www.cs.umass.edu/~yanlei/publications/XMLPubSub.pdf

[YiH10] Yi Huang, Dennis Gannon.
A Comparative Study of Web Services-based Event Notification Specifications.
http://www.cs.indiana.edu/~yihuan/research/yhuang-comparativeStudy.pdf

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGSVERZEICHNIS