Seminar Datenbanksysteme:
Entwicklungen in den Bereichen Datenintegration und Workflow Management:

Sommersemester 2005

Event-basierte – Publish/Subscribe-Architektur

(Thema 22)

Seminararbeit im Fach Wirtschaftsinformatik
Vorgelegt von
Tobias Schlaginhaufen
26. Mai 2005, Zürich, Schweiz
t.schlaginhaufen@access.unizh.ch

Matrikel-Nummer: 99-905-721

Angefertig am Institut für Informatik
Universität Zürich
Prof. Dr. Klaus R. Dittrich
Jurate Vysniauskaite

Inhaltsverzeichnis

1. Zusammenfassung

Diese Arbeit soll in die Thematik event-basierter Systeme und in die Publish/Subscribe-Softwarearchitektur einführen.

Kapitel 3 erläutert die grundlegende Funktionsweise von message-orientierten Systemen und deren Unterschied zum Request/Reply-Paradigma. Darauf aufbauend wird das Konzept von event-basierten Systemen und der Publish/Subscribe-Architektur vorgestellt.

In Kapitel 4 werde die Anforderungen Skalierbarkeit, Interoperabilität, Verlässlichkeit, Administrierbarkeit und Erweiterbarkeit, die an ein global verteiltes System gestellt werden behandelt.

In Kapitel 5 werden Channels, subjektbasierte, muster- und wertevergleichenden Subskriptionsmechanismen in Publish/Subscribe Systemen vorgestellt.

Den Themen Synchronität (Kapitel 6) und Kopplung (Kapitel 7) werden zwei Kapitel gewidmet.

In Kapitel 8 werden Lösungen vorgestellt, wie nicht event-basierte Systeme in solche integriert werden.

2. Einführung

Event-basierte[1] beziehungsweise Publish-Subscribe-Systeme werden als die Lösung angepriesen, die den Anforderungen heutiger global verteilter Workflowmanagement-Systemen gerecht werden. Mit solchen Systemen können in relativ kurzer Zeit robuste, komplexe global verteilte Workflowmanagement-Systeme erstellt werden, die nicht zentral administriert werden müssen und dynamisch erweiterbar sind. Sie gelten als skalierbar, unterstützen asynchrone Kommunikation und lose Kopplung. Alles Eigenschaften, die von RPC-Middleware nicht unterstützt werden.

Die seit den 1980er Jahren entwickelte auf dem *Request/Reply*-Paradigma basierende Middleware für verteilte Systeme funktioniert gut, solange sie in kleineren bis mittleren Netzwerken eingesetzt werden, wo die Anzahl Komponenten überschaubar ist, die Übertragungsbandbreite garantiert ist, die Latenz kurz ist und die Komponenten immer verfügbar sind.

3. Überblick

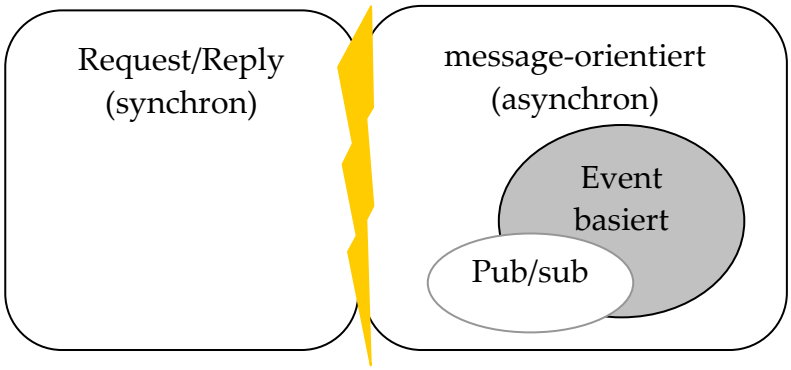

Abbildung 1: Überblick Request/Reply vs. message-orientiert

Um einen Überblick zu bekommen und um event-basierte und Publish/Subscribe-Systeme einordnen zu können, wird an dieser Stelle zuerst das Request/Reply-Paradigma und dann die message-orientierten Systeme vorgestellt, wobei event-basierte und Publish/Subscribe-Systeme in die zweiteren einzuordnen sind.

[1] *Event basiert* und *Publish/Subscribe* werden von vielen Autoren als Synonyme angesehen und es wird je nach Fokus der eine oder andere Begriff verwendet. Genaugenommen gibt es Systeme, die nur das eine oder nur das andere sind. Diese Arbeit bezieht sich auf die Schnittmenge von *Event basiert* und *Publish/Subscribe*

3.1. Request/Reply Paradigma

RPC[2] Middleware funktionieren so, dass ein lokales Programm eine Methode in einem entfernt laufenden Programm aufruft (*request*) und darauf einen Antwort-Wert (*reply*) zurück erhält. Es kommunizieren immer jeweils ein einzelner Client mit einem einzelnen Server. Ein global verteiltes Systemen wäre mit dieser 1-zu-1-Kommunikation nicht skalierbar, da bei wachsender Anzahl an Clients die mit einem Server kommunizieren dieser zwangsläufig überlastet würde. Ein weiteres Problem ist Synchronität im *Request*/*Reply*-Paradigma. Methodenaufrufe müssen synchron erfolgen da sonst die in den Komponenten ablaufenden Programme blockiert würden.

3.2. Message-orientierte Kommunikation

Message-orientiert bezeichnet man die auf Nachrichtenaustausch basierende Kommunikationsmetapher, die sich vom Methodenaufruf (Remote Procedure Call) wie folgt unterscheidet: Im Gegensatz zu RPC werden nicht andere Programme aufgerufen, sondern es wird über Nachrichten-Queues kommuniziert. In den meisten Systemen existiert eine zentrale Komponente, ein neutraler Hub [LEYMAN], der sogenannte Message Broker oder Message-Queues-Broker, der für die Weiterleitung der Nachrichten verantwortlich ist. Die Kommunikation zwischen den lose gekoppelten Komponenten erfolgt asynchron, d.h. die Kommunikationspartner müssen für die Nachrichtenübermittlung nicht unbedingt verfügbar und mit dem Netzwerk verbunden sein. Der Message-Broker kann Nachrichten speichern und bei der nächsten Verbindung zur Komponente die Nachricht übertragen.

Durch die Kommunikation über Queues lassen sich Multi- und Broadcasting sehr einfach durchführen.

Das Queue-Konzept benötigt für die Kommunikation zwischen Client und einer Message Queue im wesentlichen folgende Befehle:

- CONNECT/DISCONNECT für das Verbinden und Abkoppeln mit dem Message (Queue) Broker
- OPEN/CLOSE für das Öffnen und Schliessen eine Verbindung mit einer bestimmten Message Queue.
- PUT/GET für das Senden und Empfangen von Nachrichten.

[2] Remote-Procedure-Call

3.3. Event-basierte Architektur

Die hier behandelten event[3]-basierten Systeme verwenden message-orientierte Kommunikation. Die Kommunikation zwischen den Komponenten eines event-basierten Systems geschieht durch Event-Notifikation: Beim Eintreten eines kritischen Ereignisses bei einer Komponente, generiert diese eine Nachricht. Komponenten, die an diesem Ereignis interessiert sind, erhalten ihre Informationen, indem sie Nachrichten vom Event-Service erhalten. Die detaillierte Ausgestaltung des Routings zwischen dem Erzeuger des Ereignisses und den Empfängern kann unterschiedlich realisiert werden. Diese Unterschiede werden in dieser Arbeit später beleuchtet.

3.4. Publish/Subscribe

Unter dem Begriff *Publish/Subscribe* versteht man ein typisches Routing-Model in event-basierten Systemen. Ein *Publish/Subscribe* System besteht aus einem Netzwerk von *Publishern* und *Subscribern*. Jeder Datenaustausch wird als ein Event angesehen, das von einem Erzeuger *(Publisher* oder *object of interest)* an die dafür interessierten Empfänger (*Subscriber*), d.h. die Komponenten, die eine Subskription für dieses Event gemacht haben, geleitet wird. Normalerweise generiert eine Komponente eine Event-Notifikation, wenn sie die andern Komponente des Systems darüber informieren will, dass ein relevantes Ereignis eingetreten ist. Die Nachrichten sind nicht direkt adressiert sondern werden aufgrund ihres Subjekts oder ihres Inhalt vom Message Broker gefiltert und an die Subscriber weitergeleitet. Event Notifikationen sind häufig Multicasts oder Broadcasts. Der Versender der Event Notifikation muss sich dabei nicht darum kümmern und braucht nicht zu wissen, welche Komponenten die Nachrichten erhalten.

---

[3] Im Zusammenhang mit Event-Basierter Architektur macht es wenig Sinn, den Begriff *Event* als *Ereignis* zu übersetzen, weil präzis betrachtet *event* in der englischen Literatur sowohl für ein *Ereignis*, aber in diesem Kontext vor allem für *eine Nachricht betreffend eines Ereignisses* verwendet wird.

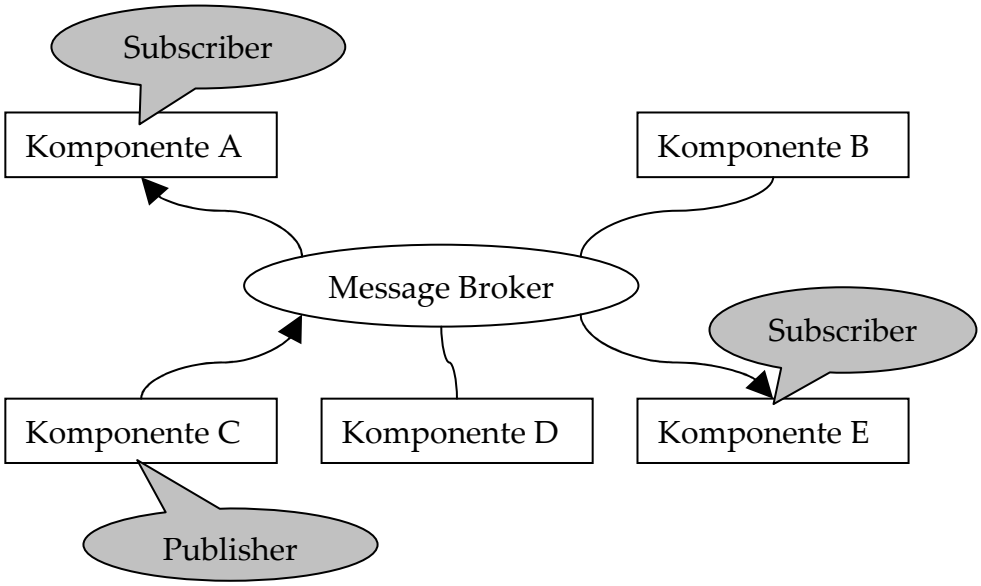

Abbildung 2: Event-basierte Architektur mit Publish/Subscribe

Im hier graphisch dargestellten Beispiel setzt Komponente C eine Nachricht über ein Ereignis ab. Die Nachricht wird nicht direkt an die Empfänger geleitet wie bei einem Peer-To-Peer System, sondern an den Message Broker. Dieser leitet diese dann zum Beispiel aufgrund des Subjektes an die für dieses Subjekt eingetragenen Komponenten A und E weiter. C muss sich nicht um die Empfänger kümmern und kennt diese in der Regel auch nicht.

4. Anforderungen

Angsichts der in Abschnitt 3.1 geschilderten Probleme von auf Request/Reply basierenden verteilten Systemen ist eine event-basierte Middleware für global verteilte Systeme von grossem Vorteil. Es folgt ein Überblick über die wichtigsten Anforderungen die gemäss [PIETZUCH02] ein global verteiltes (Workflow-management-)System zu erfüllen hat:

4.1. Skalierbarkeit

Skalierbarkeit ist eine entscheidende Anforderung an eine internet-weite Applikation. Ein System ist nur skalierbar, wenn keine Flaschenhälse, wie zum Beispiel eine zentrale Komponenten vorhanden sind. Zudem müssen die Algorithmen der Middleware darauf verzichten einen globalen Status anzustreben und natürlich sind Ressourcen wie Netzwerklast und Speicher effizient auszulasten.

Um ein event-basiertes System skalierbar zu bauen, muss davon abgesehen werden, den Event-Service als zentralistisches Element zu implementieren, sonst wird dieser schnell zu einem Flaschenhals im System.

JEDI, eine Publish/Subscribe Middleware, versucht dieses Problem zu lösen, in dem geographisch nahe zusammenliegende Komponenten, jeweils zu Gruppen zusammengefasst werden und duch einen Event-Service Knoten verknüpft werden. Alle diese Knoten werden durch eine beliebig tiefe Hierarchie zu einem Netzwerk zusammengehängt. Das Problem bleibt, dass wenn häufig zwischen in der Hierarchie weit auseinanderliegenden Knoten kommuniziert wird, der Event-Service Knoten der an der Spitze dieser Hierarchie steht als Flaschenhals auftritt.

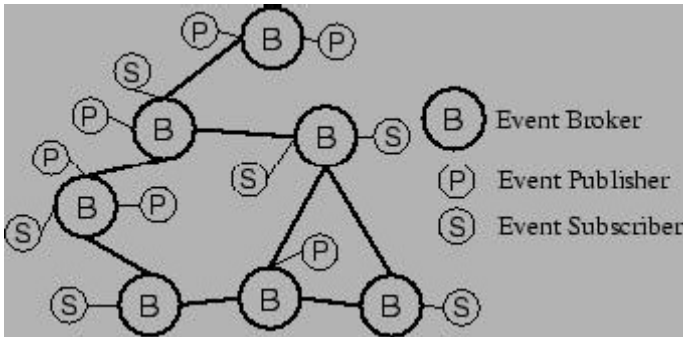

Abbildung 3: Hermes (aus [PIETZUCH02])

[PIETZUCH02] Hermes, eine weitere event-basierte Publish/Subscribe Middleware, basiert auf einem Peer-To-Peer-Netzwerk. Es existieren *Event Clients* und *Event Brokers*. *Event Clients* sind entweder *Event Publisher* oder *Event Subscriber*. Durch den Peer-To-Peer-Ansatz fällt der zentrale Event-Broker weg und macht Hermes skalierbar.

Eine weitere Anforderung an ein skalierbares System ist, dass Multicasts und Broadcasts effizient durchgeführt werden können.

4.2. Interoperabilität

Da ein global verteiltes System meistens aus einer Vielzahl heterogener Systeme und Koponenten zusammengesetzt ist, muss eine Middleware Interoperabilität unter-stützen. Dies kann durch die lose Kopplung zwischen den Systemen erreicht werden und indem sprach- und plattformunabhängig über Nachrichten kommuniziert wird. In [PIETZUCH02] wird ein offener Standard für Nachrichtenformate vorgeschlagen, wie zum Beispiel XML.

Durch Interoperabilität können Endgeräte wie Mobiltelefone, Überwachungskameras oder Sensoren ebenso verwendet werden, wie herkömliche Applikationsserver und Workstations. Ein Tatsache macht diese Architektur nicht nur für global verteilte Systeme interessant, sondern auch für das Gebiet des Ubiques Computing.

4.3. Verlässlichkeit

Eine event-basierte Middleware sollte verschiedene Stufen von *Quality of Service*-Garantien anbieten. Fehlertoleranz durch persistene Speicherung der Event-Notifikationen bei den Event-Brokern muss zur Verfügung stehen, damit auch unverlässliche Komponenten wie mobile Stationen unterstützt werden. Wie in jedem System muss auch hier zwischen Verlässlichkeit und (teurer) Redundanz abgewogen werden.

4.4. Administrierbarkeit und Erweiterbarkeit

Ein event-basiertes System wächst schnell zu einem grossen, dynamischen und komplexen System heran, dass aus vielen heterogenen Komponenten besteht. Es ist daher wichtig ein System so zu entwerfen, dass es sich "selbstadministriert", das heisst, dass die Komponenten autonom sind und sich kein zentraler Administrator um diese kümmern muss. Zudem sollte das Hinzufügen von Komponenten ein einfacher "Plug in"-Vorgang sein.

5. Subskriptionsmechanismen

Die Implementation vom Subskriptionsmechanismus in Publish/Subscribe Systemen kann man grundsätzlich in zwei Kategorien einteilen: In den statischen Channel-Ansatz und in dynamischeres inhaltsabhängiges Routing.

5.1. Channels

Das Channel-Konzept ist eine sehr einfache Version eines Event-Services. Kompo-nenten die an einen Channel angebundenen sind erhalten alle Nachrichten die an den Channel gesendet werden. Die Subskription kann demzufolge für einen oder mehrere Channels erfolgen.

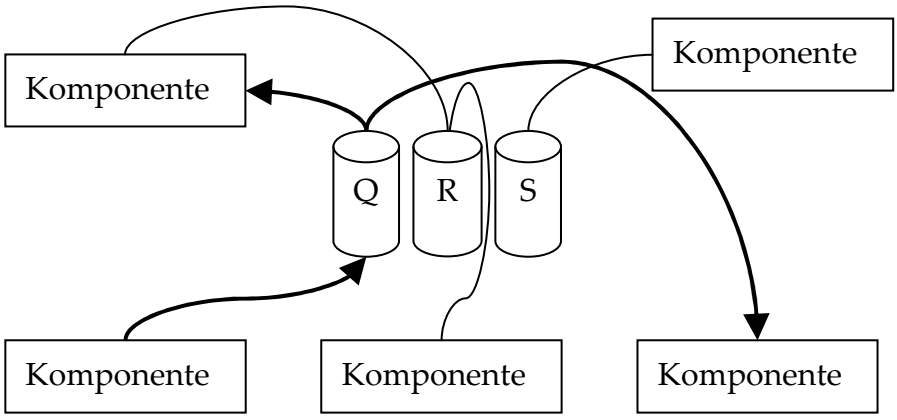

Abbildung 4: Channels

Im abgebildeten Beispiel sind Komponenten A und E mit Channel Q und erhalten somit alle Nachrichten, die an den Channel Q gesandt werden.

Der Channel-Ansatz ist ein sehr statischer Subskriptions-Mechanismus und für Anwendungen gedacht, wo die Subskriptionen nicht oft ändern. Die Anzahl möglicher Subskriptionen wird durch die Anzahl Channels bestimmt.

Ein bekannter Vertreter für den Channel Ansatz sind die Channels in CORBA.

5.2. Subjektbasiertes Routing *(topic based)*

Beim subjektbasierten Routing besteht eine Nachricht aus einem Subjekt- und einem Datenteil. Eine Komponente erhält Nachrichten über das Subjekt, für das sie angemeldet ist. Ein Subjekt ist somit ein virtueller Konnektor zwischen Publisher und Subscriber. Die meisten heute kommerziell angebotenen Publish/Subscribe-Systeme wie *TIBCO's Smartsockets* und *Rendezvous*, als auch Java JMS bieten subjektbasiertes Routing an.

5.3. Musterbasiertes Routing *(pattern matching)*

Eine etwas flexiblere Lösung wie das rein subjektbasierte Routing ist musterbasiertes Routing. Dabei können mittels so genannter *Wildcards* Nachrichten aufgrund ihres Musters gefiltert und weitergeleitet werden.

JEDI verwendet diese Methode: Eine JEDI-Nachricht besteht aus einem Namen und einer bestimmten Anzahl von Parametern: zum Beispiel: `Alarm(PC1, HALTED)`. Eine Subskription wird mittels Mustervergleich gemacht. So erhält z.B. durch eine Subskription `Alarm*(_,_)` sämtliche Nachrichten, deren Namen mit Alarm beginnt und zwei Parameter haben.

5.4. Wertevergleichendes Routing

Eine Nachricht in einem System, das wertevergleichenes Routing unterstützt, besteht aus Attributen und Werten. Eine Subskriptionen geschieht in der Form, dass eine Komponente Nachrichten erhält, wenn ein oder mehrere Werte in einem definierten Wertebereich liegen.

Elvin stellt ein komplett inhaltbasiertes und wertevergleichendes Routing zur Verfügung. Die Nachrichten bestehen aus einer Liste von Namen und Wertzuweisungen.

```
sensor-id: "manifold temp"
reading:   287.45
time:      12323564566
```

Empfänger melden sich für die interessierenden Wertebereiche in der folgenden Form an:

```
sensor-id == "manifold temp" && reading > 250
```

5.5. Vergleich

Grundsätzlich kann man festhalten, dass umso statischer der Routing-Algorithmus implementiert ist, desto mehr Overhead wird erzeugt, das heisst mehr Nachrichten erreichen Komponenten, die diese nicht unbedingt interressieren. Durch inhalt-abhängiges Routing oder noch besser durch wertvergleichendes Routing können Nachrichten viel genauer gefiltert werden und somit die Anzahl unnötiger Nachrichten minimiert werden. Andereseits steigt beim inhaltsabhängigen Routing die Anzahl virtueller Konnektoren zwischen den Knoten ins Unermessliche und eine Übersicht oder Kontrolle wird unmöglich. Bei Channels oder auch noch beim Subjekt basierten Routing ist die Anzahl virtueller Konnektoren beschränkt.

Beim Channel-Ansatz besteht zusätzlich das Problem, dass Channels einen zentralen Event-Service erfordern, wobei inhaltsabhängiges besser dezentral betrieben werden kann.

6. Synchronität

Eine wichtige Eigenschaft von Kommunikationsmodellen in einem verteilen System ist die Synchronität. Ob die Komponenten eines System synchron oder asynchron kommunizieren hat einerseits direkte Auswirkungen auf die darauf basierende Anwendung. Andererseits bestimmt die Heterogenität und die Verfügbarkeit der Komponenten, ob überhaupt synchrone Kommunikation möglich ist. Da message-orientierte Architekturen auf asynchrone Kommunikation ausgelegt sind, sind sie für synchrone Anwendungen nur bedingt geeignet, obschon es beispielsweise in TIBCO's Rendezvous und die Möglichkeit zur pseudo-synchronen Kommunikation und Transaktionsunterstützung gibt.

Was die Vor- und Nachteile von asynchroner Kommunikation sind und welche Modelle es gibt um in asynchronen Systemen trotzdem Synchronität zu garantieren werden nachfolgend erläutert:

Besondere Bedeutung bekommt die Synchronität, wenn es sich um ein System mit mobilen Komponenten handelt, wo die Erreichbarkeit und genügend Bandbreite häufig nicht gegeben ist. Auch bei global über das Internet verbundenen verteilten Systemen ist die Verfügbarkeit ein grosses Problem.

6.1. Asynchrone Kommunikation

Bei asynchroner Kommunikation müssen die Kommunikationspartner nicht während der gesamten Kommunikation direkt miteinander verbunden sein. Bei message-orientierten Systemen ist ein der Message-Broker dafür verantwortlich, dass Nachrichten den Empfänger auch dann erreichen, wenn dieser gerade nicht mit dem Netzwerk verbunden ist.

Auf eine Anfrage einer Komponente A wird eine Antwort von Komponente B mit zeitlicher Verzögerung eintreffen. Intern muss A also dagegen robust sein, dass Antwortnachrichten verzögert oder in zufälliger Reihenfolge eintreffenden. Der Vorteil von asynchroner Kommunikation ist folglich, dass sie auch schlecht verfügbare Kommunikationspartner geeignet ist.

6.2. Synchrone Modelle

In Anwendung in denen Transaktionssicherheit gefordert ist, führt asynchrone Kommunikation unweigerlich zu Blockaden und vielen Transaktionsabbrüchen und Synchronität ist unbedingt erforderlich. [CHEVERST] sieht eine Möglichkeit zur Lösung diese Problems indem Synchronität nur für einen Teil des Systems, sinnvollerweise für die "sicheren" Komponenten, gefordert wird. Dazu teilt [CHEVERST] synchrone Kommunikation in folgende drei Modelle ein:

6.2.1. Enge Synchronität *(Close Synchronity Model)*

Enge Synchronität garantiert eine totale Ordnung bei der Nachrichtenübertragung. Diese Bedingung zieht hohe Kosten mit sich: Es müssen alle Nachrichten mit einem logischen Zeitstempel versehen und dann beim Message-Broker sortiert werden. Dieser Vorgang blockiert unter Umständen das ganze System für kurze Zeit. Für ein System mit eingeschränkt verfügbaren Komponenten ist enge Synchronität nicht erreichbar.

6.2.2. Virtuelle Synchronität *(Virtual Synchronity Model)*

Virtuelle Synchronität ist eine etwas schwächere Einschränkung als die enge Synchronität. Garantiert wird hier, dass nur kausal zusammenhängende Nachrichten in ihrer Reihenfolge erhalten bleiben.

6.2.3. Erweiterte virtuelle Synchronität *(Extended Virtual Synchronity Model)*

Noch eine Stufe schwächer ist die Bedingung für erweiterte virtuelle Synchronität. Hier wird virtuelle Synchronität nur für einen Teil des Netzwerks verlangt. Das macht

besonders dann Sinn, wenn unzuverlässige Komponenten wie Mobilegeräte involviert sind.

7. Kopplung

Unter der Kopplung versteht man in einem verteilten System das Mass an Interaktion zwischen den Komponenten [DEWAN]. Die Kopplung hat direkten Einfluss auf die für die Kommunikation benötigte Bandbreite und daher auf die Verwendung von mobilen Geräten in einem System.

Man unterscheidet zwischen eng und lose gekoppelten Systemen [CHEVERST]:

7.1. Enge Kopplung

Für Anwendungen bei denen ein hohes Mass an Interaktion nötig ist, z.B. bei einem verteilten Entwurfswerkzeug für Baupläne, ist es nötig, dass das gemeinsame Artefakt stets auf allen angeschlossenen Geräten identisch ist. Es muss also jede Änderung in Echtzeit an alle anderen kommuniziert werden. Die Granularität von Updates ist klein und der daraus resultierende Kommunikationsaufwand wird dementsprechend sehr hoch. Generell gilt also, dass je enger die Kopplung zwischen Komponenten, desto höher die benötigte Bandbreite für die Kommunikation.

Da bei enger Kopplung die Anzahl ausgetauschter Nachrichten viel grösser ist, sind die Schnittstellen in der Regel auch komplexer. Das macht das Hinzufügen von neuen Komponenten schwieriger als bei loser Kopplung.

Enge Kopplung zieht meistens auch Synchronität in der Übertragung mit sich. Wie oben erläutert ist Synchronität in event-basierten ein Problem und damit enge Kopplung nicht für event-basierte Systeme geeignet.

7.2. Lose Kopplung

Auf der anderen Seite übermitteln Komponenten die lose gekoppelt sind nur Benachrichtigungen bei kritischen Ereignissen. Durch die geringe Anzahl an Interaktion ist es möglich, die Schnittstellen einfach zu halten.

Lose gekoppelte Komponente sind dadurch charakterisiert, dass sie unabhängig für sich arbeiten und sich nicht um die Existenz von andern Komponenten kümmern müssen. Die Kopplung ist auch ein Indikator für die Komplexität der Schnittstelle. Lose gekoppelte Komponenten sind meistens durch eine einfachere Schnittstelle charakterisiert, und sie können einfacher hinzugefügt oder ausgetauscht werden ohne an den bestehenden

Komponenten Änderungen vornehmen zu müssen.

8. Integration

Ein wesentlicher Problemkreis beim Einsatz von einer event-basierten System ist die Anbindung von bereits bestehenden Anwendungen, anderen message-basierten Systemen oder auch Datenbanken. Zwei wesentliche Aufgaben bei dieser Integration ist das Programmieren von Adaptern und der Einsatz eines Services, der Übersetzungen zwischen den Nachrichtenformaten vornehmen kann.

8.1. Adapter

Wie in Kapitel 3 erläutert bestehen bezüglich Kommunikation grundsätzliche Unterschiede zwischen message-orientierten *Remote Procedure Call*-Systeme.

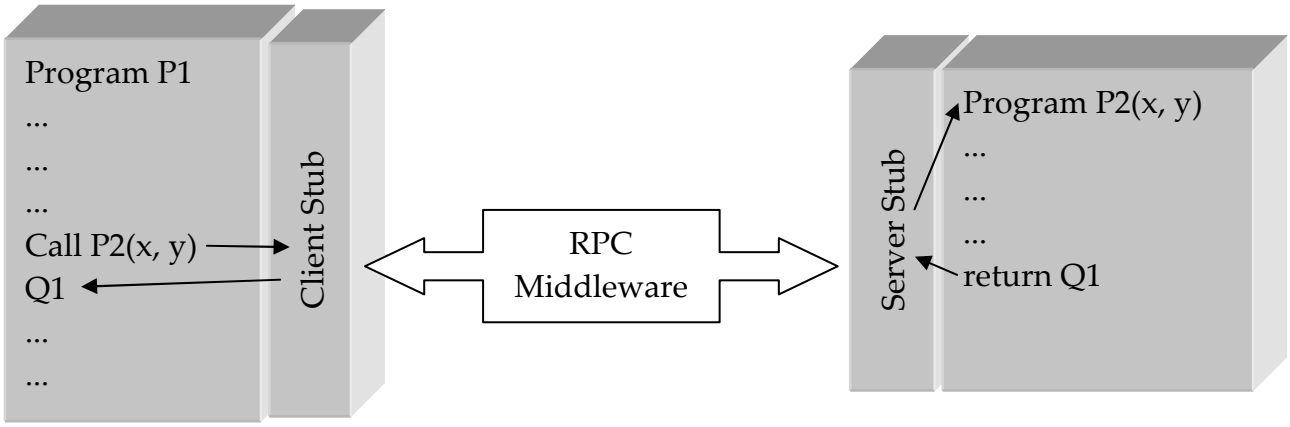

Abbildung 5: Remote Procedure Call

Abbildung 5 zeigt einen *Remote Procedure Call* in dem das Programm P1 P2 mit Inputparametern aufruft. Dabei wird die Methode im *Client Stub* aufgerufen, der einen Request generiert und an den Server Stub schickt. Der Server Stub ruft dann das Programm P2 auf (*request*) und schickt den Rückgabewert (*reply*) an den Client Stub zurück, der ihn P1 übergibt.

Um ein RPC-System in ein message-basiertes System zu integrieren sind Adapter erforderlich. Bei einem Knoten wird ein Adapter[4] benötigt, der einen RPC in eine Nachricht umwandelt und an den Message-Broker weiterleitet. Die Applikation ruft also nicht die Methode im entfernten Programm auf, sondern eine Methode des *Front-end-Adapters*. Beim anderen Knoten empfängt ein *Back-end-Adapter* die Nachricht, dekodiert diese und ruft die Methode beim lokalen Programm auf.

[4] Den Adapter bei der aufrufenden Applikation nennt man Front-end-Adapter. Den Adapter bei der aufgerufenen Applikation wird Back-end-Adapter genannt.

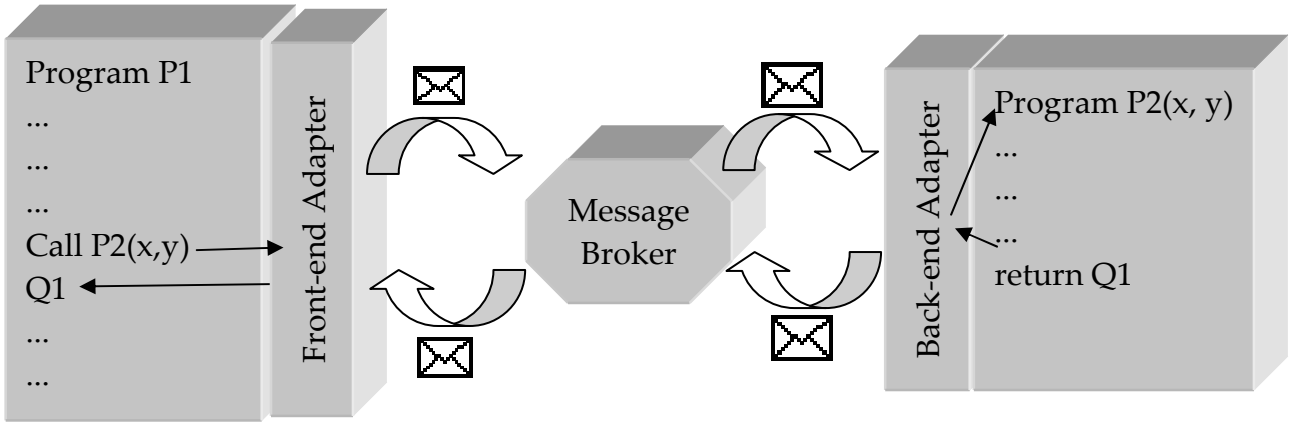

Abbildung 6: Remote Procedure Call in ein Message-orientiertes (Event-basiertes) System integriert

Zu beachten ist, dass bei der für asynchrone Kommunikation ausgelegten message-orientierten Architektur die Zeit zwischen *Request* und *Reply* eher länger dauern wird als beim herkömmlichen System, was eventuell Probleme verursachen kann.

Da Adapter häufig von Hand implementiert werden müssen und die Anzahl benötigter Adapter in einem Peer-To-Peer-System bei N Knoten N·(N-1) beträgt, ist es sinnvoll, einen neutralen Hub zu verwenden, der die Komplexität verringert [LEYMANN]. Dadurch wird die Anzahl benötigter Adapter auf maximal N reduziert.

Bei der Anbindung von Datenbanken geschieht das analoge wie bei RPC-Anwendungen. Es wird ein Adapter benötigt, der eine hereinkommende Nachricht in ein SQL-Statement umwandelt, die Anfrage an die Datenbank stellt und das Resultat wieder in eine Nachricht übersetzt und weiterleitet.

8.2. Nachrichtenformate

Bei der Integration verschiedener message-orientierter Systeme zu einem message-orientierten beziehungsweise event-basierten System kommt es oft vor, dass die Nachrichtenformate (*message sets*) der bestehenden Systeme nicht untereinander kompatibel sind. Durch den Message-Broker ist es möglich, dass die Nachrichten nötigenfalls von einem in das jeweils andere Format umgewandelt werden. Die dafür benötigten Informationen ruft der Message Broker im Message Repository ab. [LEYMAN] Im Message Repository sind Information über die Struktur der Messagesets, Abbildungsregeln, sowie Transformationsregeln für spezielle semantische Übersetzungen. Beispielsweise könnte eine Nachricht von EDIFACT[5] in XML umgewandelt werden müssen und zusätzlich die Preisangaben von netto (inkl. MwSt) in brutto (ohne MwSt). Dafür müsste die Struktur der beiden Formate und die Abbildungsregeln bekannt sein,

[5] EDIFACT steht für *Electronic Data Interchange For Administration, Commerce and Transport*. EDIFACT ist ein internationaler Standard nach ISO 9735

aber auch die Mehrwertsteuersätze verschiedener Länder müssen im Message-Repository verfügbar sein.

9. Fazit

Dieser kurze Überblick über die event-basierte beziehungsweise Publish/Subscribe-Architektur mit dem Fokus auf global verteilte Systeme zeigt, dass in solchen Systemen das Hauptproblem die Skalierbarkeit ist. Vor allem die Topologie des Event-Services als auch der Subskriptionsmechanismus sind entscheidend, ob ein System, das global über das Internet verteilt ist, performant betrieben, administriert und ausgebaut werden kann. Es muss jedoch immer abgewogen werden, welche Bedürfnisse abgedeckt werden sollen, da bis heute noch kein Projekt oder Produkt existiert, welches sämtliche Anforderungen gleichzeitig erfüllt.

10. Literaturverzeichnis

[LEYMANN] Frank Leymann, Dieter Roller: *Production Workflow: concepts and techniques*; Prentice Hall 2002

[CARZANIGA] Antonio Carzaniga, Elisabeta Di Nitto, David S. Rosenblum, Alexander L. Wolf: *Issues in Supporting Event-based Architectural Styles*

[PIETZUCH02] Peter R. Pietzuch: Event-Based Middleware: *A New Paradigm for Wide-Area Distributed Systems?*, 2002

[PIETZUCH04] Peter R. Pietzuch, *Hermes: A scalable event-based middleware*, 2004

[CHEVERST] Keith William John Cheverst: *Development of a Group Service to Support Collaborative Mobile Groupware*

[DEWAN] Prasun Dewan, Rajiv Choudhary: *Coupling the User Interfaces of a Multiuser Program*, 1995

[BELOKOS] Andras Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, Ken Moody: *Role-Based Access Control for Publish/Subscribe Middleware Architectures*; University of Cambridge Computer Laboratory

[ELVIN] *What is Elvin;* http://elvin.dstc.edu.au/index.html

[JMS] *Java Message Service Tutorial* http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/jms_tutorialTOC.html

[YEAST] Balachander Krishnamurthy, David S. Rosenblum: *YEAST: A General Purpose Event-Action System*; IEEE Transactions on Software Engineering, vol. 21, no. 10, Oct. 1995

[SIENA] Gianpaolo Cugola, Gian Pietro Picco, and Amy L. Murphy: *Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Middleware*

[TRIANTAFILLOU] Peter Triantafillou, Andreas Economides: *Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems*