Event und Notification Service in CORBA

Marcus Denker

Event und Notification Service in CORBA

Marcus Denker

Zusammenfassung

Mittels CORBA können verteilte Applikationen einfach über ein Netzwerk miteinander kommunizieren. Doch das von CORBA zur Verfügung gestellte Kommunikationsmodell reicht in manchen Fällen nicht aus, es wird ein Modell zur asynchronen Kommunikation zwischen lose gekoppelten Objekten benötigt. Der *CORBA Event Service* ist ein Versuch, ein solches Kommunikationsmodell bereitzustellen. Der Event Service hat sich aber als unzureichend herausgestellt. Daher wurde er um einige Aspekte zum *Notification Service* erweitert.

1 Einleitung

1.1 Wozu Events?

Ein zentraler Aspekt der objektorientierten Programmierung ist die Kommunikation zwischen den Objekten mittels Nachrichten. Alan Kay, einer der Erfinder des objektorientierten Paradigmas, bezeichnet die Nachrichtenübermittlung sogar als die zentrale Idee von OOP [**?**].

In den meisten existierenden objektorientierten Systemen sind Nachrichten synchron, das heißt, daß der Sender der Nachricht so lange blockiert, bis der Empfänger die Aufgabe erledigt hat und evtl. ein Ergebnis zurückliefert. Eine Nachricht in einem solchen System ist auch immer an einen einzigen bestimmten Empfänger gerichtet.

Beispiele für solch synchrone Nachrichten sind einmal die Methodenaufrufe der objektorientierten Programmiersprachen, aber auch das von CORBA breitgestellte Request/Response Model (Abbildung **??**).

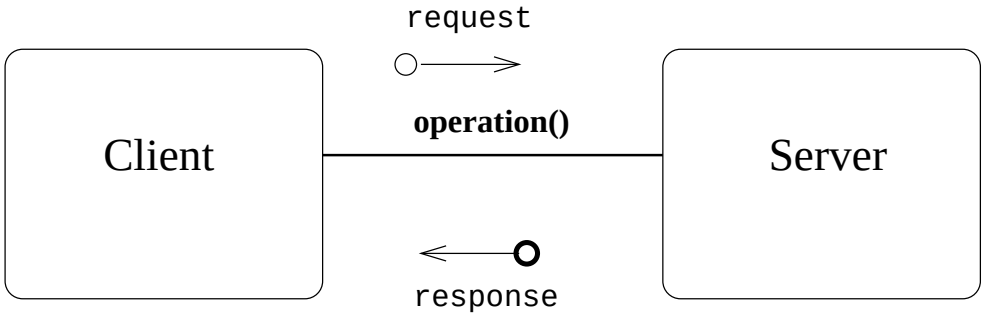

Abbildung 1: Das CORBA Request/Response Modell

Der Klient (Dienstnehmer) ruft eine Operation des Zielobjektes auf, das sich auch auf einem anderen Rechner befinden kann. Der Dienstnehmer blockiert so lange, bis der Dienstgeber seine Aufgabe erledigt hat und evtl. ein Ergebnis zurückliefert.

In der Praxis hat sich diese einfache synchrone eins-zu-eins Kommunikation als nicht ausreichend herausgestellt. Daher wurde versucht, in CORBA auch andere Kommunikationsmodelle zur Verfügung zu stellen. Der *Event Service* und seine spätere Erweiterung, der *Notification Service*, sind Beispiele für ein solches Kommunikationsmodell.

Ein Event-Service bietet folgende Erweiterungen:

- Er entkoppelt die Kommunikation, d.h. die Objekte müssen nichts mehr voneinander wissen, können aber trotzdem Nachrichten austauschen.
- Es können Nachrichten $1 : n$ und auch $m : n$ ausgetauscht werden. Eine $1 : n$ Kommunikation findet statt, wenn eine Nachricht von einem Sender an viele Empfänger gesendet wird. Die $m : n$ Kommunikation erweitert noch die Anzahl der Sender, d.h. es senden m verschiedene Sender an die gleichen n Empfänger.
- Die Übertragung ist asynchron. Der Sender blockiert nicht.

1.2 Beispiele

Es folgen einige Beispiele um zu verdeutlichen, in welchen Situationen ein solches Event-System eingesetzt werden kann.

- Eine Netzwerkverwaltungsprogramm ist an einer Benachrichtigung interessiert, wenn der Speicherplatz auf einer Festplatte eines Rechners im lokalen Netz erschöpft ist. Dabei können im Betrieb neue Platten und auch weitere Verwaltungsprogramme hinzugefügt werden.
- Eine Anwendung möchte gerne informiert werden, wenn sich die von ihr benötigten Daten ändern. Zum Beispiel möchte ein CASE-Tool benachrichtigt werden, wenn der mit dem CASE-Tool bearbeitete Quellcode von anderen Tools verändert wurde. Das CASE-Tool kann dann den veränderten Code analysieren und seine internen Strukturen auf einen aktuellen Stand bringen.
- Ein Börsenkurs-Programm möchte die Kurse an die angeschlossenen Rechner weitergeben, sobald neue Daten zur Verfügung stehen. Dabei können zur Laufzeit Klienten hinzugeschaltet oder abgemeldet werden.

1.3 Weiteres Vorgehen

Die Seminararbeit gliedert sich in 2 Teile.

Im ersten Teil wird der in CORBA implementierte Event Service vorgestellt. Nach einem Überblick und genauerer Erläuterung der Architektur des Event Service folgt ein Beispiel anhand dessen die genaue Verwendung verdeutlicht wird. Anschließend werden die Probleme analysiert, die zu einigen Erweiterungen des Event Service geführt haben.

Der zweite Teil erläutert nun diese Erweiterungen, wie sie für einen CORBA Notification Service vorgeschlagen werden.

2 CORBA Event Service

2.1 Suppliers/Comsumers

Der CORBA Event Service definiert für Objekte zwei Rollen: „Suplier“ und „Consumer“ (Erzeuger und Verbraucher).

Der Event Suplier erzeugt Ereignisse, der Consumer „verbraucht“ d.h. verarbeitet Ereignisse.

Die Ereignisse werden mittels normaler CORBA-Requests zwischen Erzeuger und Verbraucher ausgetauscht. Die Ereignisse selber sind dabei nicht als CORBA-Objekte implementiert, da das verteilte Objekt Modell von CORBA es nicht erlaubt, Objekte als Parameter zu übergeben (call by value).

2.2 Kommunikationsmodelle

Erzeuger und Verbraucher von Ereignissen können beide sowohl eine aktive als auch eine passive Rolle einnehmen.

2.2.1 Push-Modell

Wenn das *Push-Modell* verwendet wird, initiert der Erzeuger die Kommunikation. Der Erzeuger sendet das Ereignis zum Verbraucher durch Aufruf seiner `push()`-Operation (siehe Abbildung **??**).

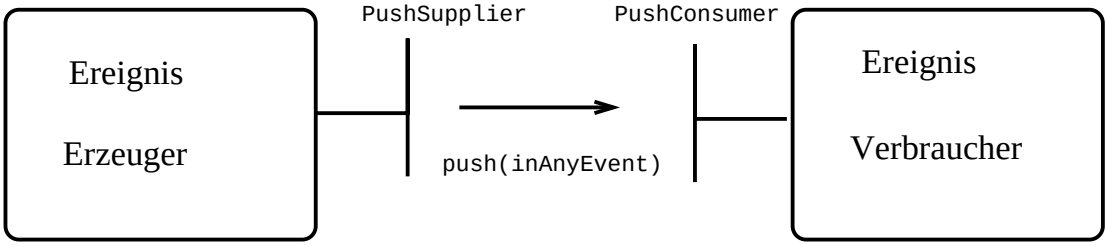

Abbildung 2: Das *Push-Modell*

2.2.2 Pull-Modell

Im Gegensatz dazu ist beim *Pull-Modell* der Verbraucher aktiv: Er ruft die `pull()`-Operation des Erzeugers auf, die ein Ereignis zurückliefert, falls es vorhanden ist (siehe Abbildung **??**). Der Erzeuger bietet sowohl blockierende als auch nichtblockierende Varianten der `pull()`-Operation.

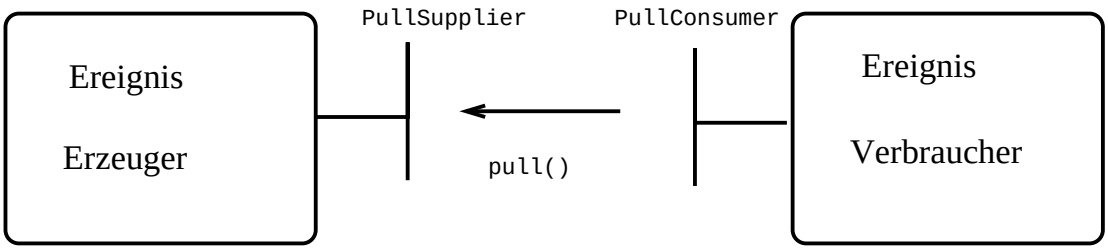

Abbildung 3: Das *Pull-Modell*

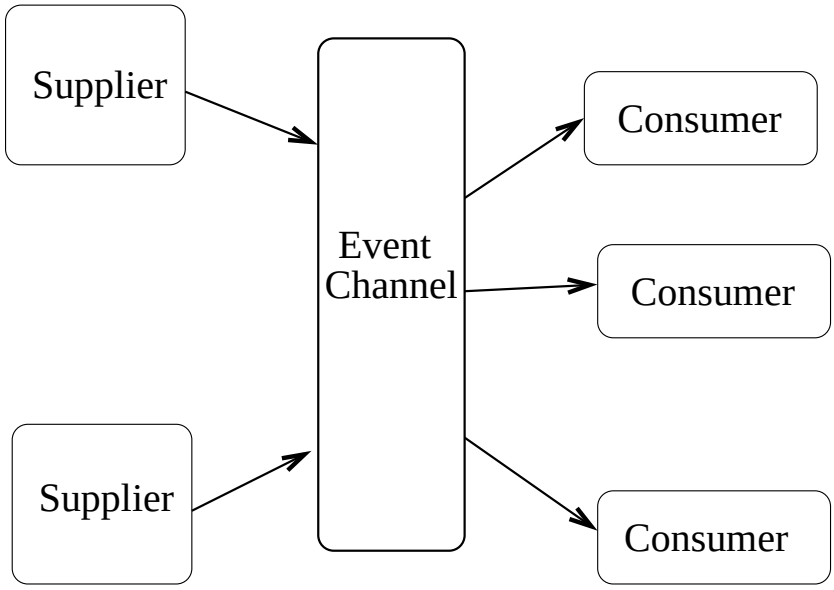

Abbildung 4: Der Event Channel

2.3 Event Channel

Der Event Channel spielt eine zentraler Rolle innerhalb des Event Service. Er entkoppelt Erzeuger und Verbraucher und ermöglicht Gruppenkommunikation.

Ein Event Channel ist im Prinzip ein Objekt, das Ereignisse von mehrereren Erzeugern empfängt und diese an mehrere Emfänger weiterleitet.

Der Event Channel erlaubt einem Erzeuger, Ereignisse an alle interessierten Verbraucher weiterleiten, ohne daß er genau wissen muß, an welche Verbraucher und an wieviele insgesamt das Ereignis weitergeleitet wird. Die Anzahl der Verbraucher kann sich sogar dynamisch zur Laufzeit ändern.

Eine weitere Eigenschaft des Event Channels ist die Umsetzung vom Push-Modell ins Pull-Modell. Das heißt, daß z.B. ein Verbraucher die Ereignisse per Pull-Modell beim Event Channel abholen kann, obwohl der Erzeuger sie mittels Push-Modell angeliefert hat.

2.4 Die IDL-Schnittstellen des Event Services

Wie alle Schnittstellen der OMG *Common Object Services* (COS) ist auch der Event Service mittels OMG IDL definiert.

2.4.1 Das `CosEventComm`-Modul

Das `CosEventComm` Modul definiert die Schnittstellen der Erzeuger und Verbraucher. Es existieren sowohl für Erzeuger als auch Verbraucher jeweils zwei Schnittstellen, eine für das Push- und Pull-Modell.

```
module CosEventComm
{
  exception Disconnected {};

  interface PushConsumer {...};

  interface PushSupplier {...};

  interface PullSuplier  {...};

  interface PullConsumer {...};
};
```

Wenn das Push-Modell verwendet werden soll, muß der Ereigniserzeuger die `PushSupplier`-Schnittstelle implementieren, der Verbraucher implementiert die `PushConsumer`-Schnittstelle.

```
interface PushConsumer {
   void push (in any data) raises(Disconnected);
   void disconnect_push_consumer();
};
```

Die `PushConsumer`-Schnittstelle definiert zwei Methoden, die vom Erzeuger aufgerufen werden können. Zum einen die `push()`-Operation, mit der Ereignisse übergeben werden, zum anderen kann der Verbraucher mit der `disconnect_push_consumer()`-Methode dazu gebracht werden, die Verbindung aufzugeben.

Sollte die Verbindung beim Aufruf von `push()` schon unterbrochen sein, wird eine `Disconnected`-Exception ausgelöst.

```
interface PushSupplier {
     void disconnect_push_supplier();
};
```

Der Erzeuger muß bei Verwendung des Push-Modells einzig die Methode `disconnect_push_supplier()` implementieren.

Im Pull-Modell wird die aktive Rolle vom Verbraucher eingenommen, also ist entsprechen zum `PushSupplier` einzig eine `disconnect_pull_consumer()`-Operation zu definieren.

```
interface PullConsumer {
   void disconnect_pull_consumer();
};
```

Der Verbraucher kann nun die vom Erzeuger zur Verfügung gestellten Methoden aufrufen, im einzelnen `pull()`, `try_pull()` und `disconnect_pull_supplier()`.

```
interface PullSuplier {
   any pull() raises(Disconnected);
   any try_pull(out boolean has_event)
     raises(Disconnected);
   void disconnect_pull_supplier();
};
```

Der Verbraucher kann Ereignisse sowohl durch Aufruf von `pull()` oder `try_pull()` abrufen. Dabei blockiert die `pull()`-Operation, bis ein Ereignis vorhanden ist. Die `try_pull()`-Operation blockiert nicht. Wenn ein Ereignis verfügbar ist, wird es zurückgeben und der Parameter `has_event` wird auf `true` gesetzt. Sollte kein Event vorhanden sein, so hat `has_event` den Wert `false` (Der Rückgabewert ist in diesem Fall undefiniert).

Sollte die Kommunikation schon unterbrochen worden sein, wird in beiden Fällen die `Disconnected`-Exception ausgelöst.

2.4.2 Das `CosEventChannelAdmin`-Modul

Dieses Modul definert die Schnittstellen zum Event Channel.

Zum einen enthält dieses Modul alle Schnittstellen, die Erzeuger und Verbraucher benötigen, um eine Verbindung zum Event Channel aufzubauen. Zum anderen sind hier die Schnittstellen definiert, über die Ereignisse mit dem Event Channel ausgetauscht werden.

```
interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_supplier();
    void destroy();
};
```

Mittels der `destroy()`-Methode kann der Event Channel zerstört werden. Daneben wird jeweils für Erzeuger und Verbraucher eine Methode angeboten: `for_consumers()` und `for_suppliers()`. Diese geben ein `ComsumerAdmin-` bzw. `SupplierAdmin`-Objekt zurück:

```
interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
};
```

```
interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
};
```

Der `ConsumerAdmin` bzw. `SupplierAdmin` bietet nun Methoden an, mit denen das entsprechende Objekt die Referenzen ihres Gegenparts innerhalb des Event Channel bekommen kann. Wenn ein Erzeuger bei dem Event Channel Ereignisse abliefern möchte, so muß der Event Channel ein Interface bereitstellen, das dem eines Verbrauchers entspricht. Entsprechend muß für einen Verbraucher eine Erzeuger-Schnittstelle vorhanden sein.

Für diesen Zweck werden vier sogenannte *Proxy*-Schnittstellen definiert. Diese erben von den entsprechenden Schnittstellen im `CosEventComm`-Modul, fügen diesen aber jeweils noch eine Methode hinzu. Mit dieser Methode, z.B. `connect_push_supplier()` bei der Schnittstelle `ProxyPushConsumer`, stellt das Objekt die Verbindung mit dem Event Channel her.

```
interface ProxyPushConsumer: CosEventComm::PushConsumer {
   void connect_push_supplier(
           in CosEventComm::PushSupplier push_supplier)
      raises(AlreadyConnected);
  };

interface ProxyPullSupplier: CosEventComm::PullSupplier {
  void connect_pull_consumer(
         in CosEventComm::PullConsumer pull_consumer)
      raises(AllreadyConnected);
};

interface ProxyPullConsumer: CosEventComm::PullConsumer {
  void connect_pull_supplier(
         in CosEventComm::PullSupplier pull_supplier)
      raises(AlreadyConnected, TypeError);
};

interface ProxyPullSupplier: CosEventComm::PushSupplier {
  void connect_push_consumer(
         in CosEventComm::PushConsumer push_consumer)
      raises(AlreadyConnected, TypeError);
};
```

2.5 Beispiel

Um das Zusammenspiel von Erzeuger, Verbraucher und Event Channel zu verdeutlichen soll ein einfaches Beispiel vorgestellt werden, bei dem ein Erzeuger über einen Event Channel mit einem Verbraucher kommuniziert. Sowohl Erzeuger als auch Verbraucher verwenden das Push-Modell.

2.5.1 Verbindungsaufbau

Bevor Ereignisse ausgetauscht werden können, müssen sich Erzeuger und Verbraucher mit dem Event Channel verbinden. Der Verbraucher muß dazu folgende drei Schritte durchführen:

1. Der Verbraucher ruft die `for_consumers()`-Methode des Eventchannel auf, um eine Referenz auf ein `CunsumerAdmin`-Objekt zu bekommen.

2. Nachdem die `for_consumers()`-Methode aufgrufen wurde, kann nun eine Methode des `ConsumerAdmin`-Objektes aufgerufen werden. Entsprechend dem gewählten Kommunikationsmodell (*push* oder *pull*) ruft der Verbaucher entweder `obtain_pull_supplier()` oder `obtain_push_supplier()` auf, um eine entsprechende Referenz auf ein Proxy-Objekt zu bekommen.

3. Wenn der entprechende Erzeuger-Proxy bekannt ist, kann sich der Verbraucher mit dessen Hilfe mit dem Event Channel verbinden. Dazu existiert die Methode `connect_push_consumer()`. Als Parameter übergibt er eine Referenz auf sich selbst (`this`).

Abbildung **??** stellt noch einmal alle Schritte dar, die für den Verbindungsaufbau benötigt werden.

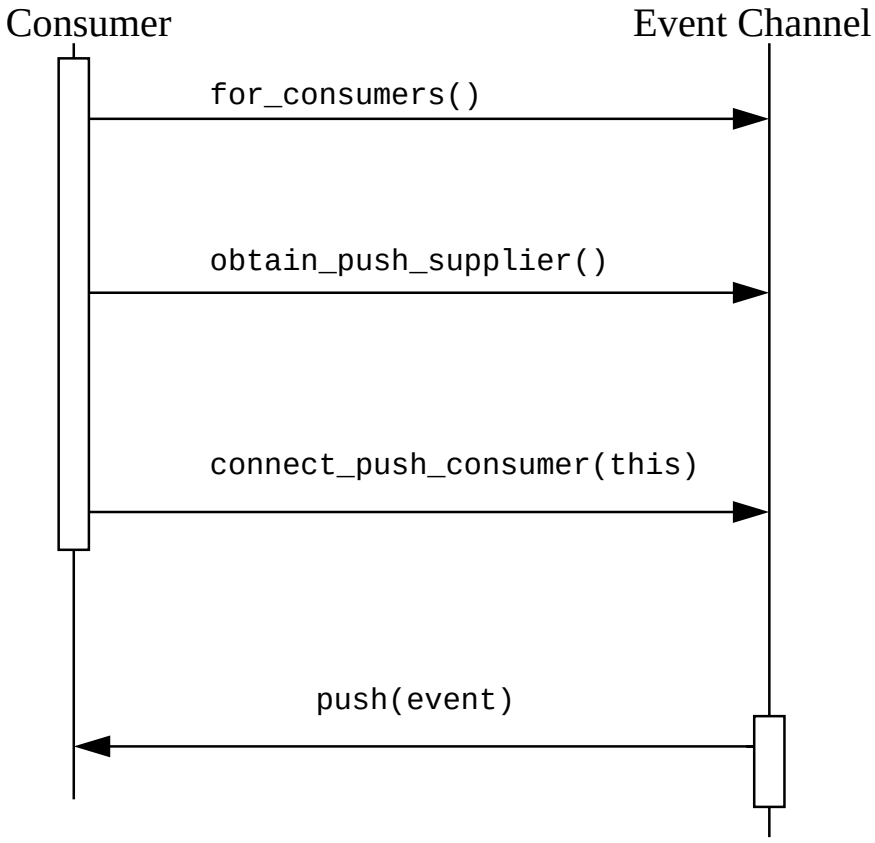

Abbildung 5: Der Verbindungsaufbau

Der Verbindungsaufbau zwischen Erzeuger und Event Channel ist symmetrisch zu dem oben Beschriebenen: Er ruft zuerst die `for_suppliers()`-Methode des Event Channel auf und erhält eine Referenz auf einen `SupplierAdmin`. Dessen `obtain_push_cosumer()`-Methode liefert einen Verbraucher-Proxy, mit dessen Hilfe (Methode `connect_push_supplier()`) sich der Erzeuger mit dem Event Channel verbindet.

2.5.2 Event-Kommunikation

Nachdem sowohl Erzeuger als auch Verbraucher die Verbindung zum Event Channel aufgebaut haben, können sie damit beginnen, miteinander zu kommunizieren.

Die zwischen Erzeuger und Verbraucher ausgetauschten Ereignisse werden mittels IDL spezifiziert, damit man sie in den CORBA-Typ `Any` konvertieren kann.

Für das Beispiel wurde angenommen, daß einzig das Push-Modell Verwendung findet. Daher wird der Erzeuger das Ereignis mittels der `push()`-Methode an den Event Channel übergeben. Der Event Channel selber wiederum ruft die `push()`-Methode des Verbrauchers auf.

2.6 Typed Events

Neben dem beschriebenen einfachen Event Service bietet CORBA auch eine typisierte Variante („Typed Events“).

Der bis jetzt betrachtete „einfache“ Event Service verlangt, daß die Ereignisse vom CORBA-Typ `Any` sind. Dadurch gehen alle Typinformationen verloren, eine Fehlerquelle entsteht.

Die im CORBA Event Service spezifizierten Typed Events sollten es erlauben, ganze in IDL spezifizierte Schnittstellen über den Ereignis-Mechanismus zu verwenden. Das heißt, der Erzeuger kann z.B. die in dieser Schnittstelle definierten Methoden des Verbrauchers mit Hilfe des Event Channels aufrufen.

In der Praxis hat es sich aber herausgestellt, daß die Spezifikation nicht einfach umsetzbar war: Sie ist (laut [?]) nicht in allen ORB implementiert. Eine genaue Beschreibung der Typed Events findet sich in [?].

2.7 Probleme

Beim Einsatz des CORBA-Event-Systems hat sich herausgestellt, daß einige schwerwiegende Probleme in der realisierten Architekur existieren.

Kein Filtern von Events Ein Verbraucher, der sich bei einem Event Channel anmeldet, erhält alle Events, auch wenn er eigentlich nur ein Teil dieser Events wirklich benötigt.

Netzwerküberlastung Die fehlenden Möglichkeiten zur Eventfilterung führen dazu, daß alle Events an alle Verbraucher geliefert werden müssen. Eine unnötig hohe Netzwerkbelastung ist die Folge.

Quality of Service Es gibt keine Möglichkeit, die Dienstgüte festzulegen. Die Auslieferung von Ereignissen an die Verbraucher sollte sich je nach gewünschter Qualität konfigurieren lassen. Bei verlangter hohe Diestgüte müßte z.B. die Auslieferung an jeden einzelnen Verbraucher garantiert sein, d.h. der Event Channel müßte den Event puffern und die Auslieferung immer wieder versuchen. Falls aber eine geringe Dienstgüte ausreicht, könnte der Event Channel so konfiguriert werden, daß er nur einen Versuch unternimmt, das Ereignis auszuliefern.

3 CORBA Notifikation Service

Die in Kapitel **??** erläuterten Probleme des Event Service gaben Anlaß, einen neuen, erweiterten Ereignisdienst zu entwickeln.

Der *CORBA Notification Service* ist eine Erweiterung des CORBA Event Service. Er ist abwärtskompatibel zum Event Service, und auch die Erweiterungen fügen sich in die Konzepte des Event Service ein.

Auf eine ausführliche Beschreibung der IDL-Schnittstelle wird verzichtet. Eine genaue Beschreibung findet sich in [**?**].

3.1 Überblick

Die wichtigsten Erweiterungen betreffen zwei Bereiche: *Event Filtering* und *Quality of Service*.

Die Verbraucher können den Proxy-Interfaces Filter zuordnen. Filter sind Objekte, die z.B. den einzelnen Verbraucher-Proxies zugeordnet werden und die alle Arten von Ereignisse beschreiben, die zum Verbraucher weitergeleitet werden sollen. Alle anderen werden herausgefiltert.

Unter *Quality of Service* versteht man die Definition von einzuhaltenden Dienstgüten. Der Notifikation Service definiert einige solche Parameter (siehe Kapitel **??**).

Der Event Channel des Notifikation Service erlaubt es, mehrere Admin-Objekte zu definieren. Sowohl Event Filter als auch Dienstgüteparameter eines Admin-Objekts gelten für alle verwalteten Erzeuger und Verbraucher. Daher lassen sich also Objekte mit gleichen Parametern zu Gruppen mit gleichen Filtern und gleicher Dienstgüte zusammenfassen.

Möglich wurden diese Erweiterungen durch die Einführung von *Structured Events*. Diese erlauben es, neben den eigentlichen Ereignisdaten auch weitere Daten zu enthalten, etwa die Dienstgüteparameter.

3.2 Erweiterungen

Im folgenden werden einige der im Notifikation Service vorgenommenen Erweiterungen genauer beschrieben: *Event Filtering* und *Quality of Service*.

3.2.1 Event Filtering

Die Filter-Objekte erlauben es, die Auslieferung von Ereignissen auf genau die Ereignisse einzuschränken, an denen der Empfänger interessiert ist.

Die Filter können durch sog. *Constraints* (Einschränkungen) bestimmen, welche Ereignisse ausgeliefert werden. Constraints werden dadurch definiert, daß jedem Ereignistyp ein Ausdruck zugeordnet wird. Dieser Ausdruck wird in einer definierten Sprache, der *Filtering Constraint Language* formuliert.

Eine genaue Beschreibung dieser Sprache findet sich in [**?**]. Hier sollen einige kleine Beispiele genügen.

Der Notification Service erlaubt es, Ereignisse durch gewisse Eigenschaften zu beschreiben. So können den Ereignissen z.B. ein Typ zugeordnet und Namen für Ereignisse definiert werden.

- Nehme alle Ereignisse vom Typ `CommunicationsAlarm` an, aber keine `lost_package`-Ereignisse:

  ```
  $Type_name == 'CommunicationsAlarm' and not
  ($event_name == 'lost_packet')
  ```

- Nehme `CommunicationsAlarm` Ereignisse mit einer Priorität größer 5 an:

  ```
  $type_name == 'CommunicationsAlarm' and
  $priority > 5
  ```

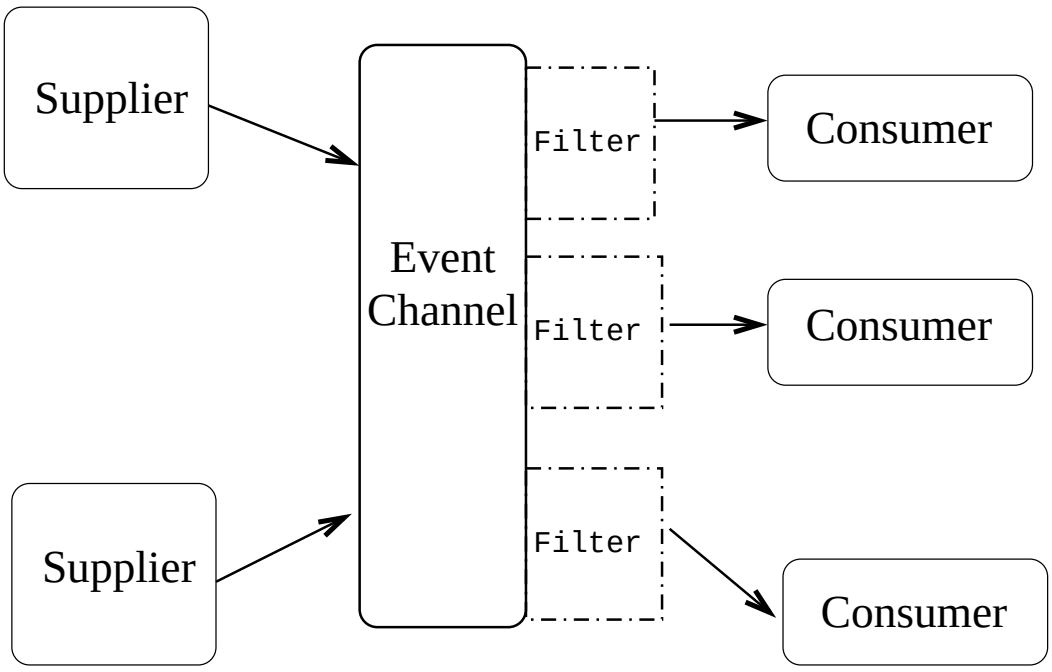

Abbildung 6: Event Channel mit Filtern

3.2.2 Dienstgüte

Der Notification Service erlaubt es, die Dienstgüte der Ereignisauslieferung mittels verschiedener Parameter zu steuern. Dabei kann man diese Parameter nicht nur global für den Notification Channel setzen, sondern auch beschränkt auf einzelne Gruppen von Ereignissen und sogar für einzelne Ereignisse definieren.

Die Dienstgüteparameter können im einzelnen für folgende Objekte gesetzt werden:

- Den komplette Notification Channel.
- Admin-Objekt der Erzeuger und Verbraucher.
- Die Proxy-Objekte.
- Den einzelnen Event.

Setzt man Dienstgüteparameter für ein Admin-Objekt, so gelten sie für alle über dieses AdminObjekt erzeugte Proxy-Objekte. Eine für ein Proxy-Objekt definierte Dienstgüte gilt dann für alle über diesen Proxy laufenden Ereignisse.

Es folgt eine kurze Beschreibung der wichtigsten Dienstgüteparameter.

Reliabiliy Die Verläßlichkeit kann auf zwei Ebenen geregelt werden: Einmal für den Kanal und einmal für das einzelne Ereignis. Siehe [**?**], Seite 53.

Priority Normalerweise definiert der Notification Service keine Reihenfolge für die Auslieferung. Bei Angabe einer Priorität (zwischen -32767 und $+32767$) versucht der Event Channel die Ereignisse in Reihenfolge dieser Prioritäten auszuliefern.

Expiry Time Es können verschiedene Werte für die zeitliche Gültigkeit konfiguriert werden: `StopTime` definiert einen exakten Zeitpunkt, ab dem das Ereignis ungültig ist. `TimeOut` dagegen definiert diesen Zeitpunkt relativ zur aktuellen Zeit.

Earliest Delivery Time Durch den Wert `StartTime` kann festglegt werden, ab wann ein Ereignis ausgeliefert werden darf.

Order Policy Mit diesem Parameter kann festgelegt werden, in welcher Reihenfolge die Ereignisse ausgeliefert werden. Mögliche Werte dieses Parameters sind:

`AnyOrder` Jede Ordnung ist erlaubt.

`FifoOrder` Auslieferung in Reihenfolge der Einlieferung.

`PriorityOrder` Auslieferung bestimmt durch die Prioritäten.

`DeadlineOrder` Die *Expiry Time* legt die Auslieferung fest: Als erstes werden die Ereignisse ausgeliefert, die als erstes ungültig würden.

Discard Policy Daduch kann festgelegt werden, in welcher Reihenfolge bei einer Überlastung Ereignisse gelöscht werden können.

4 Zusammenfassung

Der CORBA Notification Service erweitert die Möglichkeiten der Kommunikation zwischen Objekten. Das normale Request/Response Modell wird um eine Möglichkeit zur asynchronen, lose gekoppelten $m : n$ Kommunikation ergänzt, die in der Praxis häufig benötigt wird. Die Probleme, die beim Event Service anfänglich vorhanden waren, wurden durch die Erweiterungen des Notification Service beseitigt.

Literatur

[Gree98] M.J. Greenberg (Hrsg.). *Notification Service – Joint Revised Submission.* OMG. November 1998.

[Kay97] Alan Kay. *The Computer Revolution Hasn't Happend Yet! Keynote OOPSLA 1997.* UVC Video. Oktober 1997.

[OMGr97] Object Management Group (Hrsg.). *Event Service Specification*, Kapitel 4. OMG. 1997.

[ScVi97a] D.C. Schmidt und S. Vinoski. Object Interconnections – Overcoming Drawbacks in the OMG Events Service. *SIGS C++ Report Magazine*, Februar 1997.

[ScVi97b] D.C. Schmidt und S. Vinoski. Object Interconnections – the OMG Events Service. *SIGS C++ Report Magazine*, Februar 1997.

[Tech98] IONA Technologies. *Notification White Paper.* Mai 1998.