# Fachstudie:

# Vergleich von Architekturen zur Gerätevernetzung

**Autoren:**

**Lilia Chamsieva, Jan Geiger, Matthias Wieland**

# 1 Organisatorisches

Dieses Dokument ist im Rahmen der Fachstudie „Vergleich von Architekturen zur Gerätevernetzung“ entstanden und wurde von Lilia Chamsieva, Jan Geiger und Matthias Wieland (Studenten des Studiengangs Softwaretechnik an der Universität Stuttgart) verfasst.

Die Anforderungen an das erstellte Dokument wurden von Herrn Alexander Leonhardi und Herrn Dr. Martin Simons (DaimlerChrysler AG) vorgegeben.

Betreut wurden wir bei der Durchführung der Fachstudie von Herrn Dr. Christian Becker (Abteilung Verteilte Systeme / IPVS / Universität Stuttgart).

Alle an der Fachstudie beteiligten Personen:


| | |
|---|---|
| Alexander Leonhardi | Alexander.leonhardi@daimlerchrysler.com |
| Dr. Martin Simons | Martin.simons@daimlerchrysler.com |
| Dr. Christian Becker | Christian.becker@informatik.uni-stuttgart.de |
| Lilia Chamsieva | liljek@gmx.de |
| Jan Geiger | geigerjn@gmx.net |
| Matthias Wieland | wielanms@web.de |

# 2 Einleitung

Bei der Entwicklung von Fahrzeugtelematiksystemen sollen in Zukunft verstärkt standardisierte Mechanismen für den Zugriff auf die daran beteiligten Geräte und Dienste zum Einsatz kommen. Damit soll erreicht werden, dass Geräte und Dienste auf einfache Weise ausgetauscht und erweitert werden können. Entsprechende Architekturen, die für diesen Zweck geeignet erscheinen, sind für die Vernetzung von Geräten der Unterhaltungselektronik entstanden.

In dem vorliegenden Dokument werden zunächst die grundlegenden Begriffe und Konzepte bei der Vernetzung von Geräten erläutert. Danach werden die folgenden Architekturen im Detail vorgestellt:

- Home Audio / Video Interoperability (HAVi)
- Universal Plug and Play (UPnP)
- Open Service Gateway Initiative (OSGI)
- Java-Technologien: Jini, Java Media Framework und Java Messaging System

Abgeschlossen wird dieses Dokument dann durch eine vergleichende Gegenüberstellung der verschiedenen Architekturen.

# 3 Grundlagen der Gerätevernetzung

## 3.1 Warum Vernetzung? Warum Architekturen zur Gerätevernetzung?

Aufgrund besserer Skalierbarkeit, Erweiterbarkeit und einem besseren Preis-/Leistungsverhältnis werden verschiedene Geräte in zunehmendem Maße vernetzt, um sämtliche Ressourcen (Speicherplatz, Rechenleistung, Funktionalität und Daten), welche diese Geräte bereitstellen, gemeinsam Nutzen zu können. Die Geräte, die wir betrachten unterscheiden sich in ihrer Leistung erheblich: es reicht von Mobiltelefonen und CD-Playern bis hin zu voll ausgestatteten PCs und Workstations. Der Schwerpunkt der Betrachtung liegt allerdings auf Geräten der Unterhaltungselektronik. Bei dieser Fachstudie betrachten wir Architekturen zur Vernetzung solcher Geräte. Was diese Architekturen vor allem bieten, ist die Möglichkeit einer spontanen Vernetzung all dieser Geräte, so dass keine komplizierte Konfiguration des Netzwerks notwendig ist und alle Geräte damit nach gewissen (von den jeweiligen Architekturen festgelegten) Standards integriert werden und somit aus dem gesamten Netzwerk heraus genutzt werden können .

In diesem Abschnitt möchten wir nun darauf eingehen, was die Architekturen, die sich als Grundlage zur Gerätevernetzung anbieten, alles gemeinsam haben und die wichtigsten Grundlagen der Gerätevernetzung erläutern.

## 3.2 Geräte und Dienste

Zuerst etwas zum Begriff **Gerät**: In diesem Abschnitt wird Gerät als allgemeiner Begriff für eine Entität gebraucht, die mit anderen Entitäten vernetzt ist. Dies können nicht nur Hardware-Geräte sein, sondern z.B. auch Software-Programme bzw. Prozesse. Um eine gewisse Flexibilität zu erreichen, kann ein Gerät noch in **Dienste** unterteilt werden. Ein Dienst ist eine logische Einheit in einem Gerät, die üblicherweise genau einer Funktionalität entspricht. Ein Gerät stellt dann einen oder mehrere Dienste bereit.
Als Beispiel für ein Gerät mag ein Fernseher mit integriertem digitalen Videorecorder dienen. Dieser enthält dann mehrere Dienste, die einzelnen Funktionalitäten entsprechen:

- einen Bildschirm (Anzeige von Videosignalen)
- die Videorecorderfunktionalität (Aufnahme des Programms xy zu einem bestimmten Zeitpunkt, Abspielen von bestehenden Aufnahmen)
- einen Tuner (Empfang von Fernsehsignalen)
- und vielleicht sogar ein Modem für den Internetzugang (elektronische Programmzeitschrift).

## 3.3 Client/Server und Peer-to-Peer

Geräte bzw. Dienste stehen bei den betrachteten Architekturen immer in einer **Client-Server**-Beziehung zueinander, d.h. ein Gerät (der Klient) macht gegenüber einem anderen Gerät (dem Server) eine Anfrage bezüglich einer Diensterbringung. Bei dem Client-Server-Konzept geht man üblicherweise von einer strikten Trennung der Client- und Server-Rollen aus. Es gibt allerdings auch noch den Sonderfall, dass sämtliche Geräte grundsätzlich gleichberechtigt sind und somit

ein Gerät sowohl die Client- als auch die Server-Rolle gleichzeitig einnehmen kann (aber natürlich nicht muss). In diesem Fall spricht man von **Peer-to-Peer-Vernetzung**.

## 3.4 Spontane Vernetzung

In diesem Abschnitt wurde der Begriff der **spontanen Vernetzung** schon kurz erwähnt. Dahinter verbirgt sich die Möglichkeit, Geräte zu einem beliebigen Zeitpunkt dem Netzwerk hinzuzufügen. Dadurch können sämtliche Klienten im Netzwerk diese neu hinzugekommenen Geräte und Dienste sofort und ohne irgendwelche manuelle Administration nutzen, sofern es für sie sinnvoll ist. Die Administration findet dabei hinter den Kulissen statt und wird von der Infrastruktur, die durch die entsprechende Architektur zur Gerätevernetzung zur Verfügung gestellt wird, durchgeführt (Plug&Play).

## 3.5 Discovery: Entdecken von Diensten

Die Vorgänge, die dabei stattfinden, werden unter dem Begriff **Discovery** zusammengefasst. Beim Discovery-Vorgang müssen dem Netzwerk neu hinzugefügte Geräte jeweils „auf sich aufmerksam machen", d.h. sie müssen sich in irgendwelcher Art und Weise über die Verteilungsinfrastruktur bei den schon vorhandenen Geräten registrieren. Dies kann entweder über eine zentrale Komponente der Verteilungsinfrastruktur geschehen (wie es z.B. bei Jini der Fall ist) oder durch Versenden von Ankündigungsnachrichten an alle relevanten Geräte (vgl. z.B. Universal Plug and Play).

Geräte, die Dienste anbieten (also „Server-Geräte"), müssen sich daher beim Discovery-Vorgang durch entsprechende Nachrichten bei einer zentralen Komponente registrieren oder an alle anderen Geräte im Netz diese Nachrichten schicken und somit auf sich aufmerksam machen. Die Nachricht wird in beiden Fällen an eine sogenannte **wohlbekannte Adresse** (wellknown address) geschickt. Eine solche wohlbekannte Adresse wird von einer entsprechenden Architektur zur Gerätevernetzung definiert. An dieser wohlbekannten Adresse müssen dann sämtliche Klienten oder die zentrale Komponente ständig hören, ob solche Nachrichten eintreffen und diese dann entsprechend verarbeiten.

Falls ein Klient einem Netzwerk beitritt, muss dieses Gerät ebenfalls ein Discovery durchführen. Discovery bedeutet für einen Klienten, dass er aktiv nach Diensten sucht, die er gerne benutzen möchten. Dazu muss er spezielle Suchnachrichten an eine wohlbekannte Adresse verschicken und in diesen Suchnachrichten spezifizieren, welche Dienste für ihn von Bedeutung sind.

Die Discovery-Nachrichten werden dann je nach verwendeter Discovery-Variante als **Unicast**-Nachrichten (Nachricht an einen Empfänger), als **Multicast**-Nachrichten (Nachricht an eine Gruppe von Empfängern, setzt üblicherweise eine Gruppenverwaltung voraus, wie sie z.B. auch von IP oder Ethernet zur Verfügung gestellt wird) oder als **Broadcast**-Nachrichten (Nachricht an alle Teilnehmer im Netzwerk) verschickt.

## 3.6 Bekanntmachung der Dienstschnittstelle

Beim Discovery muss der Klient, damit er sinnvoll einen Dienst nutzen kann, auf irgendeine definierte Art und Weise die Dienstschnittstelle in Erfahrung bringen können. Für die **Bekanntmachung der Dienstschnittstelle** gibt es zwei alternative Verfahren. Das erste Verfahren bedient sich der **Übermittlung einer Schnittstellenbeschreibung**, die z.B. mittels einer

Schnittstellenbeschreibungssprache (z.B. CORBA IDL) oder einem XML-Dokument beschrieben wird. Die zweite Möglichkeit bedient sich der **Codeübertragung**, wie sie z.B. in Java gegeben ist. Dabei wird dann die Schnittstelle durch die Bereitstellung und Übertragung eines **Dienststellvertreters** (Proxy) bekannt gemacht, der durch lokale Aufrufe benutzt wird und diese lokalen Aufrufe in Dienstaufrufe umwandelt.

## 3.7 Dienstnutzung

Hat ein Klient schließlich einen passenden Dienst gefunden, mit dem er interagieren möchte, so geschieht diese Interaktion üblicherweise durch **Kontrollnachrichten** (Requests) oder durch **entfernte Prozeduraufrufe (Remote Procedure Call, kurz RPC)** - bzw. aufs objektorientierte Programmierparadigma übertragen durch **entfernte Methodenaufrufe (Remote Method Invocation, kurz RMI)**. Diese Art der Interaktion entspricht einem **synchronen Kommunikationsmodell**, bei dem Klient und Dienst direkt miteinander kommunizieren und der Klient nach dem Absetzen seines Requests auf die Antwort (sofern es eine gibt) des Dienstes blockierend wartet. Ein alternatives Kommunikationsmodell, nämlich **asynchrone Kommunikation** über eine vermittelnde Instanz, wäre durch den Einsatz von **Messaging** zu erreichen. Dabei wird ein hoher Grad an Entkopplung von Klient und Dienst gewährleistet und außerdem eine **n:m-Kommunikation** (ein Klient kommuniziert mit mehreren Diensten simultan und umgekehrt) ermöglicht. Allerdings ist letztere Form der Interaktion in den betrachteten Architekturen zur Gerätevernetzung noch nicht sehr verbreitet.

## 3.8 Multimedia: Streaming

Eine besondere Form der Interaktion, die besonders im Multimediabereich (d.h. zur Übertragung von Audio- und Videodaten) von Interesse ist, besteht im sogenannten **Streaming**. Hierbei werden die Daten nicht in Form von einzelnen (meist unabhängigen) Nachrichten, sondern als ein kontinuierlicher Datenstrom von einer Datenquelle zu einer Datensenke übertragen. Da für die nicht beeinträchtigte Wiedergabe von Audio- und Videodaten immer eine gewisse Mindestübertragungskapazität garantiert sein muss, wird fürs Streaming neben dem großzügigen Puffern von Daten meist ein sogenannter **isochroner Datentransfer** benutzt, bei dem eine gewisse Bandbreite für den Datentransfer reserviert (und somit garantiert) wird.

## 3.9 Ereignisse

Neben der reinen Interaktion zwischen Klient und Dienst gibt es in den betrachteten Architekturen jeweils auch eine Infrastruktur zur Verteilung von bestimmten Ereignissen, die mitteilen, dass sich am Zustand eines Geräts im Netzwerk etwas geändert hat. Dieser Teil der Infrastruktur wird **Ereignisdienst** genannt. Dienste registrieren sich dann als Ereignisanbieter bei einem solchen Ereignisdienst und teilen dem Ereignisdienst in Form einer Ereignisnachricht immer mit, wenn sich bei ihnen etwas geändert hat. Auf der anderen Seite registrieren sich Geräte als Interessenten bzw. Ereignisempfänger ebenfalls bei einem oder mehreren Ereignisdiensten und bekommen dann von diesen Ereignisdiensten sämtliche Ereignisse mitgeteilt (eventuell besteht auch die Möglichkeit, dass Interessenten nur Interesse an bestimmten Ereignissen bekunden).

## 3.10 Abmelden von Diensten und Geräten

Was in diesem Abschnitt bis jetzt noch nicht betrachtet wurde, ist der Fall, dass ein Gerät aus dem Netzwerk wieder entfernt wird. Handelt es sich bei dem zu entfernenden Gerät um ein reines Client-Gerät, so sind außer dem Abmelden von den Ereignisdiensten keine weiteren Verwaltungsschritte vonnöten. Handelt es sich um ein Gerät, das auch Dienste anbietet, so müssen die anderen Teilnehmer des Netzwerks über das Entfernen informiert werden (Abmeldung), da sie sonst fälschlicherweise annehmen würden, dass diese Dienste weiterhin existieren. Zu diesem Zweck muss das zu entfernende Gerät entweder garantiert all diese Aufräumarbeiten übernehmen oder es muss für sämtliche Anmeldungen/Registrierungen (zusätzlich) ein sogenannter **Leasing**-Mechanismus verwendet werden. Leasing bedeutet in diesem Zusammenhang, dass ein Registrierungseintrag nur für eine bestimmte Zeitspanne gültig ist und innerhalb dieser Zeitspanne von dem betreffenden Gerät oder Dienst auch rechtzeitig wieder erneuert werden muss, da der Eintrag sonst gelöscht wird. Durch dieses Verfahren wird verhindert, dass sich über längere Zeit viele ungültige Verwaltungseinträge ansammeln, die einen unzutreffenden Zustand des Netzes beschreiben. Auch ist dieses Verfahren oft die einzige Möglichkeit, da in bestimmten Situationen – z.B. Abbruch von Funkverbindungen – ein ordnungsgemäßes Abmelden nicht möglich ist. Eine Alternative zu Leasing-Mechanismen wäre eine Programmierung der Geräte, welche die Geräte robust auf den Ausfall anderer Geräte reagieren ließe. Allerdings wäre dies mit einem hohem Aufwand bei der Programmierung und einer geringeren Effizienz der Geräte verbunden.

## 3.11 Szenarien

Um diese allgemeinen Erläuterungen zu verdeutlichen folgenden nun zwei Beispiele, in denen gezeigt wird, wie bei praktischen Anwendungen die Techniken der spontanen Gerätevernetzung eingesetzt werden könnten.

### 3.11.1 Office-Szenario

Nehmen wir an, ich hätte mir heute einen Drucker angeschafft und würde mit diesem nun nach Hause kommen. Um ihn in meinem gesamten Heimnetzwerk nutzen zu können, müsste ich ihn nur an das Netzwerk oder an einen PC anschließen. Sofort danach wäre er an allen Rechnern verfügbar, wenn man etwas ausdrucken will.

Wie kann das funktionieren? Und was passiert, wenn man diesen Drucker an das Netzwerk anschließt?

Als erstes ein paar Hinweise:

- Der Drucker ist allgemein gesehen ein Gerät, das einen Dienst zur Verfügung stellt.
- Das Programm ist ein Client, der den angebotenen Druck-Dienst nutzen möchte.
- Das Netzwerk ist eine beliebige Vernetzungsstruktur, die über eine Infrastruktur zur spontanen Gerätevernetzung verfügen muss.

Was geschieht nun nacheinander hinter den Kulissen?

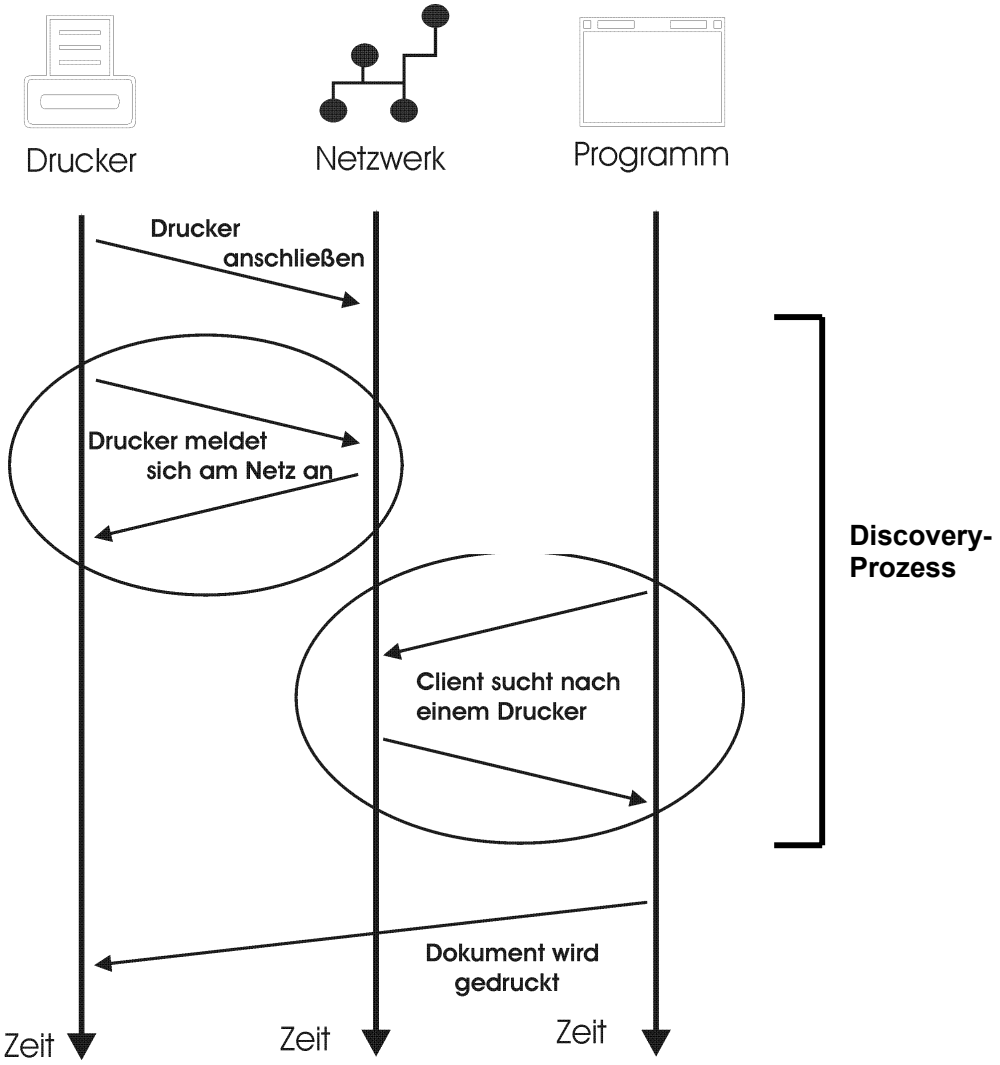



**Abbildung 1: Ablaufdiagramm des Office Szenarios**

Zuerst verbinde ich den Drucker mit dem Netzwerk auf Hardware-Ebene. Das kann natürlich z.B. auch per Funkverbindung geschehen.

Der Drucker erkennt selbstständig, dass er mit einem Netzwerk verbunden wurde und versucht sich an dem Netzwerk zu registrieren. Dies nennt man Discovery, damit gibt der Drucker bekannt, was er für Dienste anbietet und wie ich diese nutzen kann.

Zu einem späteren Zeitpunkt will nun jemand an einem Computer in dem Netzwerk drucken. Dazu bekommt er, wie üblich, ein Druckerauswahl-Dialog. In diesem Dialog werden alle Drucker angezeigt. Die Druckerliste bekommt der Client durch eine Anfrage bei der Geräte-Vernetzungs-Infrastruktur, der alle Dienste und somit auch sämtliche Druck-Dienste bekannt sind.

Nachdem er nun einen Drucker ausgewählt hat, kann das Programm seinen Druckauftrag an den Drucker abschicken.

Als Zusatz über das Bild hinaus noch ein Beispiel für ereignisbasierte Nachrichten. Nehmen wir an, der Drucker hat mitten im Druckvorgang nun kein Papier mehr. In diesem Fall wird er ein Ereignis auslösen, dieses Ereignis bekommen alle Benutzer, die sich zuvor registriert haben. Daraufhin könnte jemand Papier nachlegen, was wiederum ein Ereignis auslösen würde.

Eine andere Ereignisart wäre z.B. auch das Abschalten des Druckers. In diesem Fall würde er dann auch automatisch aus der Registrierung entfernt, damit er nicht mehr in der Druckerliste auftaucht – da er nach dem Abschalten ja auch nicht mehr benutzbar ist.

### 3.11.2 Multimedia-Szenario

Als nächstes betrachten wir ein Multimedia Szenario anhand eines DVD-Players. Diesen DVD-Player schließen wir z.B. per Firewire an unser Heimnetzwerk an. Daraufhin geschieht dasselbe wie bei dem Drucker zuvor. Nach dem Discovery ist es möglich, auf allen Geräten im Haus, welche in der Lage sind Filme anzuzeigen, DVDs abzuspielen. Man kann den DVD-Player auch von jedem Ausgabegerät aus steuern, obwohl dieses vielleicht in einem anderen Raum steht. Weitere Vorteile wären, dass eine gewisse Übertragungsrate garantiert wird, es kann darum nicht passieren, dass die Wiedergabe auf Grund von Übertragungsengpässen ruckelt. Als letztes wäre noch zu erwähnen, dass es auch möglich wäre eine Multicast-Wiedergabe zu starten und dadurch auf mehreren Anzeigegeräten gleichzeitig ein Film abgespielt werden könnte.

# 4 Umsetzung der Basiskonzepte

Nachdem im vorherigen Kapitel die grundlegenden Konzepte der Gerätevernetzung erläutert wurden, geben wir an dieser Stelle einen kurzen Überblick, wie diese Konzepte in den zu untersuchenden Architekturen realisiert sind.

## 4.1 Discovery

Bei **JINI** durchlaufen sowohl Dienstanbieter als auch Dienstnehmer die Entdeckungsphase - Discovery. Gesucht wird dabei nicht direkt nach einem Dienst oder Dienstnehmer, sondern nach einem sogenannten Vermittler – dem Lookup Service (LUS). Ein Lookup Service ist die zentrale Anlaufstelle in einem JINI-System. Ein Dienstanbieter und ein Dienstnehmer wenden sich an den LUS, wenn sie einen Dienst offerieren bzw. auffinden wollen. Der Lookup Service verwaltet alle Diensteinträge, sucht auf Anfrage nach einem gewünschten Dienst und sorgt für die Gültigkeit der Diensteinträge mittels des Leasing-Konzepts: die Dienstanmeldung im LUS ist nur für eine bestimmte Zeit gültig. Nach Ablauf dieser Zeit wird der Eintrag vom LUS automatisch entfernt, wenn die Verlängerung nicht explizit vom Dienstanbieter angefordert wurde. Ein Dienst macht sich bekannt, indem er seinen Stellvertreter, den sogenannten Dienst-Proxy als serialisiertes Objekt im LUS hinterlegt. Der Dienstnehmer gibt seinerseits eine Beschreibung des gesuchten Dienstes an den Proxy des LUS. Der LUS durchsucht seine Diensteinträge nach passenden Kandidaten und übermittelt deren Proxies an den Dienstnehmer. Der Dienstnehmer kann jetzt in Proxies Methoden aufrufen, als ob der Dienst lokal wäre.

Bei **OSGi** wird ein Dienst angemeldet, indem ein *Bundle* (Komponente, die Dienste anbietet) bei seinem *BundleContext*-Objekt (Vermittler zwischen einem Bundle und dem OSGi Framework) die Methode *registerService* aufruft. Dadurch wird es im Framework registriert und es wird ein *ServiceRegistration*-Objekt zurückgegeben. Es können nur Objekte, die dieses *ServiceRegistration-Objekt* besitzen, Änderungen am Service vornehmen, oder diesen wieder abmelden.

Wenn ein Klient nun einen angebotenen Dienst nutzen möchte fragt er bei seinem *BundleContext*-Objekt mit Hilfe der *getServiceReference*-Methode nach einer Referenz auf einen Service, der die übergebene Schnittstelle implementiert. Zurück erhält er ein so genanntes *ServiceReference*-Objekt. Mit diesem Objekt kann man nun mit der Methode *getService* ein Objekt erhalten, das den Service repräsentiert (Service-Objekt).

Bei **UPnP** sind sowohl Klienten als auch dienstanbietende Geräte aktiv. Wird ein Gerät dem Netzwerk hinzugefügt, so schickt es Multicast-Nachrichten über seine Verfügbarkeit (Advertisements) an alle UPnP-Geräte im Netzwerk (wohlbekannte Adresse). Wird ein Klient dem Netzwerk hinzugefügt, so fragt dieser – ebenfalls per Multicast-Nachrichten – nach verfügbaren Geräten nach. Dabei können Klienten auch bestimmte Einschränkungen der Suchanfrage vornehmen, indem sie z.B. nur nach Geräten bestimmten Typs suchen.

Ähnlich wie bei JINI wird auch bei UPnP das Leasing-Konzept angewandt: Advertisements verfügen immer über eine beschränkte Gültigkeitsdauer und müssen von den Geräten vor Ablauf der Gültigkeit verlängert werden. Wird ein Advertisement nicht verlängert, so gehen die Klienten davon aus, dass das entsprechende Gerät und seine Dienste nicht länger im Netz verfügbar sind.

In **HAVi** wird beim Anschluss eines neuen Gerätes an das Netzwerk ein sogenannter Bus-Reset auf dem Netzwerk-Medium ausgelöst und die bereits angeschlossenen Geräte über ihre Komponente zur Netzwerkkommunikation (CMM1394) davon benachrichtigt. Da ein Gerät bei HAVi in der Regel über mehrere Dienste verfügt, müssen sich diese jeweils bei ihrem lokalen Namensdienst (Registry) anmelden. Danach können sie von anderen Diensten oder Anwendungen gefunden werden. Bei der Suche wendet sich die suchende Komponente zunächst an ihre lokale Registry und kann dabei auch verschiedene Suchkriterien angeben (z.B. Dienst-ID oder Dienst-Typ). Die lokale Registry sucht dann zuerst in ihren Verwaltungseinträgen und befragt danach sämtliche anderen Registrys im Netzwerk (föderierter Namensdienst). Die Ergebnisse (eine Liste von Dienst-IDs) werden von der lokalen Registry aggregiert und abschließend der anfragenden Komponente zurückgeliefert.

## 4.2 Dienstbeschreibung

In **JINI** wird ein Dienst wird durch seine Schnittstelle, d.h. ein Java-Interface, definiert. Zusätzlich können Attribute bei seiner Registrierung im LUS zugeordnet werden, die den Dienst näher beschreiben oder sogar Programmcode enthalten. Attribute können auch nach der Registrierung geändert werden, um Zustandsänderungen des Dienstes wiederzugeben.

**OSGi** definiert einen Service ebenfalls durch ein Java Interface. Es können zusätzlich einem Service noch auf LDAP (Lightweight Directory Access Protocol) basierende Attribute hinzugefügt werden.

Bei **UPnP** erfolgt die Beschreibung von Geräten und Diensten mittels XML-Dokumenten. Die Beschreibungen enthalten neben allgemeinen Informationen (Hersteller, Gerätetyp, ...) auch sämtliche angebotenen Funktionen, Statusvariablen und Ereignisse (durch XML-Schemas vordefiniert). Diese XML-Dokumente können durch HTTP-GET-Requests von Klienten bei den jeweiligen Geräten angefordert werden - die URL dafür wird ihnen durch Advertisements oder Antworten auf Suchanfragen mitgeteilt.

Bei **HAVi** besitzt jedes Gerät eine eigene Gerätebeschreibung (SDD), auf die von außen lesend zugegriffen werden kann und deren Aufbau standardisiert ist. Eine solche SDD enthält dabei Informationen wie ID, Version und Typ. Geräte-Beschreibungen in HAVi besitzen allerdings keine Angaben zu den APIs einzelner Dienste. Allerdings definiert die HAVi-Spezifikation Standard-APIs für zahlreiche Dienste wie z.B. Ereignisdienst oder Registry, aber auch gerätespezifische Dienste wie VCR, Tuner und andere.

## 4.3 Dienstnutzung

In **JINI** nutzt der Dienstnehmer einen Dienst, indem er Methoden des Dienst-Proxys aufruft. Der Aufruf geschieht für den Dienstnehmer transparent. Die Kommunikation mit dem Dienst wird von seinem Proxy übernommen und kann sowohl mittels RMI als auch mit Hilfe eines anderen Protokolls realisiert werden.

Bei **OSGi** ruft man zur Nutzung des Dienstes einfach die Methoden des *Service*-Objektes, das man beim Discovery Prozess erhalten hat, auf.

Die Klienten können bei **UPnP** mit einem Dienst auf zweierlei Arten interagieren:

- Klienten können Funktionen aus der Dienstbeschreibung aufrufen
- oder Dienste nach den Werten ihrer Statusvariablen befragen
- Die Interaktion geschieht in UPnP immer durch SOAP-Nachrichten, die über HTTP transportiert werden.

Bei **HAVi** nutzt ein Klient einzelne Funktionen eines Dienstes, indem er einen Aufruf bei dem lokalen Standarddienst für die Nachrichtenübermittlung (Messaging System) macht. Damit ist es für den Klienten transparent, ob er eine lokale oder eine entfernte Funktion aufruft. Der Klient kann dabei entscheiden, ob er einen synchronen (Aufruf von *MsgSendRequestSync*) oder asynchronen (Aufruf von *MsgSendRequest*) Funktionsaufruf ausführen möchte.

Für den Aufruf muss der Klient unter anderem seine eigene ID, die ID des gewünschten Dienstes und die eindeutige Identifikation der aufzurufenden Funktion angeben. Das Messaging System verpackt dann die Aufrufparameter in entsprechende Nachrichtenpakete, die als Datagramme über den IEEE1394-Bus verschickt werden. Die Antwort wird analog dazu ebenfalls als Datagramm verschickt. Bei der Antwort auf einen asynchronen Funktionsaufruf kommt dabei die ID des Funktionsaufrufs mit zurück, damit der Klient die Antwort dem Aufruf zuordnen kann.

Zusätzlich zur reinen Interaktion gibt es in HAVi die Möglichkeit, dass Anwendungen und Dienste für sie interessante Dienste auf Verfügbarkeit überwachen lassen können. Meldet sich dann ein überwachter Dienst beim Netzwerk ab, so werden sie sofort davon benachrichtigt.

### 4.3.1 Streaming

**Jini** und **OSGi** bieten keine explizite Unterstützung von Multimedia-Streaming. Das **Java Media Framework (JMF)** unterstützt die Verarbeitung von Multimedia-Daten in Java Applikationen und Applets. Durch die Integration des **JMF** in **Jini** und **OSGi** werden diese Technologien um die Fähigkeit des Multimedia-Streaming erweitert.

**UPnP** bietet neben der SOAP-basierten Interaktion mit Diensten inzwischen Standards zum Streaming von Mulimedia-Inhalten (Audio und Video). Das Streaming wird dabei wie gehabt durch einen Klienten mittels Funktionsaufrufen bei den entsprechenden Devices initiiert (das API dazu ist von UPnP festgelegt), danach erfolgt die eigentliche Datenübertragung - allerdings ohne Zutun des Klienten. Für den Datentransfer können dabei dann beliebige, auch nicht IP-basierte, Protokolle verwendet werden, z.B. IEEE1394.

**HAVi** wurde speziell zur einfachen Vernetzung von Audio- und Videogeräten konzipiert. Dementsprechend bietet auch HAVi ein API für die Einrichtung und Nutzung von Streaming-Verbindungen. Dabei baut HAVi auf die Standards IEEE1394 und IEC 61883, die bereits detaillierte Vorschriften für die Paketformate und Transferprotokolle für das Multimedia-Streaming machen. IEEE1394 stellt dafür als Grundlage die Reservierung einzelner IEEE1394 für eine Datenübertragung bereit, während IEC 61883 das Streamingprotokoll und die Paketformate definiert.

## 4.4 Abmelden eines Dienstes

Bei **JINI** ist eine Dienstanmeldung im LUS nur für eine bestimmte Zeit gültig. Nach Ablauf dieser Zeit wird der Eintrag vom LUS automatisch entfernt, wenn die Verlängerung nicht explizit vom Dienstanbieter angefordert wurde. Die Ressourcen können auf diese Weise wieder freigegeben werden.

Bei **OSGi** wird ein Dienst abgemeldet indem man die *unregister*-Methode bei dem für das Bundle zuständigen *ServiceRegistration*-Objekt aufruft. Dadurch wird der Dienst aus dem Framework entfernt und *ServiceReference*-Objekte die auf diesen Dienste zeigen werden ungültig. Außerdem werden alle von dem Dienst abhängigen Dienste über das Abmelden durch ein Event informiert.

Geräte sollen sich bei **UPnP** direkt durch entsprechende Multicast-Abschiedsnachrichten (Cancel) beim Verlassen des Netzwerks abmelden. Tun sie dies nicht, greift der Leasing-Mechanismus und nach Ablauf einer gewissen Zeitspanne löschen alle Klienten die zugehörigen Geräte-Einträge aus ihren Caches.

Werden bei **HAVi** Geräte ausgeschaltet oder vom Netzwerk genommen, so wird ein IEEE1394-Bus-Reset ausgelöst, von dem dann die Messaging Systeme sämtlicher Geräte benachrichtigt werden. Somit kann dann auch das Verschwinden von auf Verfügbarkeit überwachter Dienste sofort den interessierten Anwendungen und Diensten mitgeteilt werden.

## 4.5 Entfernte Ereignisse

Das Ereignismodell von **JINI** realisiert eine asynchrone Benachrichtigung über mehrere Java Virtual Machines (JVM's). Dabei besteht keine Transportgarantie, da bei Jini keine zentrale Instanz existiert und somit partielle Fehler auftreten können. Zur gesicherten Weitergabe können Ereignisse (Crash-sicher, Racing-sicher) in JINI in einer Ereignis-Pipeline bei JINI delegiert werden. Eine spezielle Form der Ereignisweitergabe ist das Pipeline-Prinzip bei der Ereignis-Delegierung.

Die Verwendung einer asynchronen Kommunikation mittels Nachrichten erlaubt die Entkopplung von Sender und Empfänger. Diese Art der Kommunikation zwischen verteilten Objekten bietet **Java Message Service (JMS)** an. Eine Nachricht kann eine Meldung, ein Ereignis oder ein serialisiertes Objekt sein. Die Anlaufstelle für Nachrichten ist ein Message-Broker, als *Warteschlange* (*Queue*) oder *Stichwort* (*Topic*) realisiert. Die *Queue* und das *Topic* werden für die Point-to-Point- bzw. Publish-and-Subscribe-Kommunikation benutzt. Die Verwendung von Transaktionen ermöglicht das Zusammenfassen vom Empfangen und/oder Senden von Nachrichten in eine atomare Einheit.

Bei **OSGi** gibt es drei Klassen von Ereignissen. Für jede Klasse gibt es entsprechende *Listener*, über welche sich die *Bundles* registrieren können, um die Ereignisse zu erhalten. Die *FrameworkEvents* geben Bescheid, wenn das Framework gestartet wird, oder wenn Fehler auftreten. Die *BundleEvents* geben Veränderungen im Lebenszyklus von *Bundles* weiter. Diese Ereignisse dieser beiden Arten werden asynchron übergeben. Die dritte Art, die *ServiceEvents* findet als einzige synchron statt. Diese berichten über Anmeldung, Abmeldung und Änderungen an Services. Zusätzlich finden bei den *ServiceEvents* auch noch Sicherheitsprüfungen statt, sodass nur die *Bundles* benachrichtigt werden, die auch Zugriffsrechte auf dem benachrichtigenden Service besitzen.

Der Ereignisdienst ist bei **UPnP** verteilt realisiert. Jeder Dienst hat einen Ereignisdienst, über den er Veränderungen bestimmter Statusvariablen interessierten Klienten mitteilt. Damit ein Klient Ereignisse mitgeteilt bekommt, muss er sich bei jedem Ereignisdienst, für dessen Ereignisse er sich interessiert, registrieren. Als initiale Ereignisnachricht bekommt er dann die Werte aller durch den Ereignisdienst überwachten Statusvariablen mitgeteilt.

Daneben gibt es in UPnP auch die schon erwähnten Advertisement- und Cancel-Nachrichten, mit denen Klienten über neue oder abgemeldete Geräte automatisch informiert werden.

Auch bei **HAVi** ist der Ereignisdienst verteilt realisiert. Allerdings ist der Ereignisdienst in HAVi im Gegensatz zu UPnP föderiert. Damit ein Klient Ereignisse zugestellt bekommt, muss er sich beim lokalen Ereignisdienst (Event Manager) für jedes ihn interessierende Ereignis registrieren. Er wird dann durch eine Callback-Funktion über das Auftreten eines abonnierten Ereignisses benachrichtigt. Durch die Föderation der Event Manager kann ein Klient sowohl lokal begrenzte Ereignisse des eigenen Event Managers als auch „globale“ (auf das Netzwerk begrenzte) Ereignisse abonnieren. Gemäss dem HAVi-Standard gibt es sowohl von den einzelnen Diensten selbst definierte Ereignisse als auch bereits vordefinierte Ereignisse der Systemdienste.

# 5 Universal Plug and Play (UPnP)

Im Juni 1999 gegründet ist das UPnP Forum eine Non-Profit Organisation, die inzwischen aus über 500 Mitgliedern besteht. Treibende Kraft bei UPnP ist Microsoft, die den entsprechenden Java-Technologien zur Gerätevernetzung (insbesondere Jini) einen anderen, offenen Standard entgegensetzen möchte.

## 5.1 Begriffsdefinitionen

Diese Begriffe sind in UPnP von zentraler Bedeutung und werden daher in dieser Tabelle kurz erläutert.

| Begriff | Definition |
|---|---|
| **Action** | Bezeichnung für eine Funktion eines UPnP-Dienstes, kann entfernt aufgerufen werden |
| **Control Point** | Bezeichnung für ein Client-Gerät in einem UPnP-Netzwerk. |
| **Device** | Bezeichnung für ein Gerät mit Serverfunktionalität in einem UPnP-Netzwerk. Zu beachten ist, dass gemäss dem UPnP-Dienstmodell ein UPnP-Gerät gleichzeitig Control Point und Device sein kann (Peer to Peer Networking). |
| **Dienst** | Dienst, der von einem Device bereitgestellt wird und über ein API verfügt. |
| **HTTPMU** | HTTP over UDP Multicast. HTTP-Nachrichten über das UDP-Transportprotokoll per Multicast versenden (d.h. an eine Empfängergruppe). |
| **HTTPU** | HTTP over UDP Unicast. HTTP-Nachrichten über das UDP-Transportprotokoll per Unicast (d.h. ein einzelner Empfänger) versenden. |
| **UPnP-Gerät** | Überbegriff für Control Points und Devices, d.h. ein UPnP-Gerät bezeichnet sowohl Geräte mit Server- als auch mit Clientfunktionalität |
| **UUID** | Universally Unique ID. Bezeichner, der in einem UPnP-Netzwerk über den gesamten Betrieb des Netzwerks hinweg eindeutig ist. |

## 5.2 Überblick

### 5.2.1 Was bietet Universal Plug and Play?

Das Ziel von Universal Plug and Play (UPnP) besteht in der Peer to Peer-Vernetzung verschiedenster Geräte, d.h. die vernetzten Geräte können gleichzeitig sowohl Client- als auch Server-Funktionalität realisieren.

Ein UPnP-Netzwerk ist als ein selbstkonfigurierendes Netzwerk konzipiert, so dass sich die verschiedenen Geräte automatisch finden können und (fast) keine Administration erforderlich ist.

UPnP baut auf den Standard-Internet-Protokollen wie IP, TCP, UDP, HTTP oder SOAP auf, allerdings werden lediglich die Kommunikationsprotokolle und das Format der Dienstbeschreibungen festgelegt - ein API für UPnP wird nicht vorgeschrieben. Damit soll größtmögliche Unabhängigkeit von spezifischen Plattformen und Netzwerktechnologien erreicht werden.

Geräte, die kein UPnP unterstützen (Legacy-Geräte), können über sogenannte UPnP-Brückengeräte (Bridges) angebunden werden, wobei für die Implementierung dieser Bridges bislang keine Richtlinien bestehen bzw. nicht veröffentlicht sind.

### 5.2.2 UPnP-Networking Schritt für Schritt

Die Vernetzung von Geräten in einem UPnP-Netzwerk und deren anschließende Interaktion lässt sich in die folgenden sechs Schritte gliedern:

1. **Adressierung**: UPnP baut auf IP auf und daher ist es notwendig, dass UPnP-Geräte bei Anschluss an das Netzwerk IP-Adressen zugewiesen bekommen. Dies geschieht bevorzugt über einen zentralen DHCP-Server im Netzwerk oder – falls kein DHCP-Server verfügbar ist – durch einen verteilten Algorithmus namens Auto-IP, bei dem zufällige IP-Adressen aus einem reservierten Bereich genommen und – sofern keine Mehrfachbelegung dieser Adressen vorliegt - den UPnP-Geräten zugewiesen werden. Die Verwendung von DNS-Namen ist optional.

2. **Discovery**: Nachdem die UPnP-Geräte ihre IP-Adressen zugewiesen bekommen haben, ist es notwendig, dass die Control Points vollständige Informationen über im Netz verfügbare Devices und Dienste bekommen. Daher schicken Devices beim Beitritt zu einem Netz Nachrichten aus (sogenannte Advertisements), in denen sie ihre Verfügbarkeit ankündigen. Um zu verhindern, dass solche Informationen verloren gehen oder später beitretende Control Points sie verpassen, fragen Control Points bei ihrem Netzwerk-Beitritt mittels sogenannter Search-Requests nach verfügbaren Devices und Diensten. Damit sollen sämtliche Control Points in einem UPnP-Netzwerk jederzeit vollständige Informationen über den Zustand des Netzwerks besitzen. Advertisements und Antworten auf Search-Requests besitzen gemäss dem Leases-Prinzip nur eine eingeschränkte Gültigkeitsdauer und müssen vor dem Ablauf dieser Zeit erneuert werden. Falls Geräte das Netzwerk verlassen, sollen sie dies durch entsprechende Abschiedsnachrichten mitteilen. Durch das

Leases-Konzept und die Abschiedsnachrichten soll verhindert werden, dass im Netzwerk veraltete und ungültige Informationen über längere Zeiträume hinweg gespeichert werden.

3. **Beschreibung**: Control Points bekommen durch Discovery-Nachrichten neben einer UUID und einer Gültigkeitsdauer auch eine URL mitgeteilt, die auf die Beschreibung des jeweils entdeckten Devices verweist. Indem der Control Point einen HTTP-GET-Request an diese Adresse durchführt, bekommt er die Beschreibung des Devices im XML-Format. In der Device-Beschreibung befinden sich dann URLs, die auf die Beschreibungen der enthaltenen Dienste verweisen. Der Control Point kann diese ebenfalls mittels eines HTTP-GET-Requests anfordern und danach mit diesen Diensten interagieren (vgl. Abschnitt „Control"). Die Device- und Dienstbeschreibungen sind in einem standardisierten XML-Format gehalten, wobei Platzhalter für herstellerspezifische Informationen vorgesehen sind. Dienstbeschreibungen enthalten jeweils eine Liste von verfügbaren Statusvariablen sowie eine Liste von aufrufbaren Funktionen („Actions").

4. **Steuerung**: Nachdem ein Control Point die Beschreibung eines Dienstes erhalten hat, ist er in der Lage, Funktionen aufzurufen (entspricht RPCs, in der UPnP-Terminologie **Actions** genannt) oder Statusvariablen abzufragen (Polling). Sowohl die Funktionsaufrufe als auch das Polling werden über SOAP-Nachrichten realisiert, die Antworten bestehen ebenfalls aus SOAP-Nachrichten. Die Initiative zur Interaktion geht dabei immer vom Control Point aus. Devices können untereinander nicht kommunizieren.

5. **Ereignisverarbeitung**: In UPnP wird ein Ereignisdienst realisiert, der auf den verschiedenen UPnP-Geräten verteilt abläuft. Um den Ereignisdienst nutzen zu können, muss sich ein Control Point bei dem Ereignisdienst eines Dienstes anmelden. Dabei bekommt er im Erfolgsfall eine Bestätigung des Abonnements mit einem eindeutigen Bezeichner und einer Gültigkeitsdauer für das Abonnement (Leases-Konzept). Darüber hinaus bekommt ein solcher Control Point als initiale Ereignisnachricht die Werte aller mit dem Ereignisdienst verknüpften Statusvariablen mitgeteilt. Ändert sich an einer mit dem Ereignisdienst verknüpften Statusvariable etwas, so bekommen alle Abonnenten dies als Ereignis mitgeteilt (Publisher-Subscriber-Modell). Dies impliziert auch, dass ein Abonnent nicht selektiv Ereignisnachrichten von einzelnen Statusvariablen beziehen, sondern nur alle Ereignisse eines Dienstes abonnieren kann.

6. **Präsentation**: Falls der Benutzer ein Device über eine grafische Benutzungsoberfläche (GUI) bedienen oder konfigurieren möchte, so geschieht dies über HTML-Seiten, d.h. Devices, die über Präsentationsfähigkeiten verfügen, müssen die HTML-Seiten über einen HTTP-Server bereitstellen und Control Points müssen diese Seiten in einem HTML-Browser darstellen können. Weitere Festlegungen über den Aufbau der HTML-Seiten oder eventuelle technologische Einschränkungen bestehen nicht. Jeder Hersteller hat hier die Möglichkeit innerhalb der Grenzen von HTTP und HTML eigene Wege zu gehen.

### 5.2.3 Der UPnP-Protokollstack

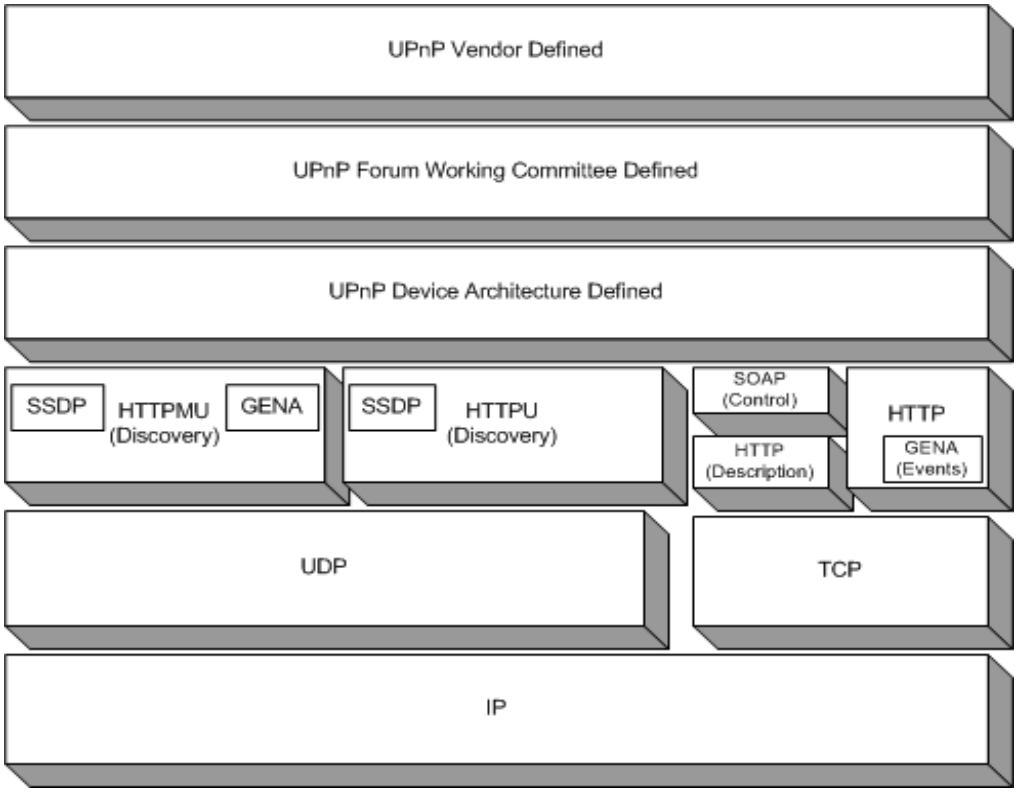



**Abbildung 2: Der UPnP-Protokollstack**

Der UPnP-Protokollstack besteht, wie in obiger Abbildung zu erkennen ist, aus den vertrauten Standardprotokollen des Internets. Von der zugrundeliegenden Netzwerk-Hardware und deren Spezifika wird abstrahiert. Die unterste Schicht, die spezifiziert ist (Netzwerkschicht), nutzt das Internet Protocol. Darüber werden als Protokolle für die Transportschicht entweder UDP (für Service-Discovery-Nachrichten) oder TCP (für Übermittlung der Dienstbeschreibungen, Ereignisverarbeitung und Steuerung) verwendet. Beim Service-Discovery-Vorgang kommen für die Search-Requests und die Advertisements Multicast-Nachrichten (HTTPMU) zum Zuge, ansonsten werden jeweils Unicast-Nachrichten verschickt. In den höheren Schichten ist die Verwendung von HTTP und teilweise auch XML-Formaten vorgegeben, wobei die festgelegten XML-Formate (UPnP Template Language) Platzhalter für verschiedene UPnP-Geräteklassen oder für herstellerspezifische Informationen beinhalten.

## 5.3 Internet Protocol als Basis – IP-Adressierung

Da UPnP das Internet Protocol als Basis voraussetzt, müssen die verschiedenen UPnP-Geräte in einem Netzwerk jeweils verschiedene IP-Adressen zugewiesen bekommen. Die IP-Adressvergabe erfolgt vorrangig über das Dynamic Host Configuration Protocol (DHCP) und setzt einen zentralen DHCP-Server im Netzwerk voraus, der konfiguriert und administriert werden muss. Da DHCP der vorrangige Mechanismus zur Vergabe von IP-Adressen ist, muss jedes UPnP-Gerät einen DHCP-Client implementieren.

Falls DHCP in einem UPnP-Netzwerk nicht verfügbar ist, wird die Verwendung eines alternativen Mechanismus vorgeschlagen: Auto-IP [AutoIP].

Auto-IP setzt im Gegensatz zu DHCP keinen zentralen Server voraus, das Verfahren läuft dezentral in der folgenden Art und Weise ab:

- Zunächst wählt ein UPnP-Gerät beim Beitritt zum Netzwerk zufällig eine Adresse aus dem Bereich 169.254/16 (d.h. aus dem Adressbereich 169.254.1.0 bis 169.254.254.255) aus. Die Adressen aus diesem Bereich sind speziell für Auto-IP reserviert und stellen keine „echten“ Internet-Adressen dar, sondern sind lediglich innerhalb eines lokalen Netzwerks gültig (Nachrichten an solche Adressen bzw. von solchen Adressen dürfen deshalb nicht an Router zur Weiterleitung geschickt werden). Sollen UPnP-Geräte auch Verbindungen über das Internet aufnehmen können, müssen sie entweder IP-Adressen von einem DHCP-Server beziehen oder die durch Auto-IP konfigurierten Adressen müssen von einem Internet Gateway mittels Network Address Translation (NAT) in „echte“ Internet-Adressen umgesetzt werden.
- Hat ein UPnP-Gerät eine Auto-IP Adresse ausgewählt, muss deren Einmaligkeit innerhalb des Netzwerks mittels eines ARP-Requests überprüft werden. Bei einem Fehlschlag der Gültigkeitsprüfung müssen Auswahl und Überprüfung solange wiederholt werden, bis das UPnP-Gerät eine gültige Auto-IP-Adresse besitzt.
- Auch wenn ein UPnP-Gerät eine gültige Auto-IP-Adresse besitzt, so soll es in regelmäßigen Intervallen (Vorschlag der Spezifikation [UPnP01] für die Intervall-Dauer: 5 Minuten) überprüfen, ob nicht doch ein DHCP-Server verfügbar ist. Wird ein solcher gefunden, so muss die Auto-IP-Adresse gegen eine vom DHCP-Server bezogene Adresse eingetauscht werden (sofern das UPnP-Gerät zu diesem Zeitpunkt keine Netzwerkverbindungen geöffnet hält).

Optional kann auch das Domain Name System (DNS) aus Gründen der Bedienfreundlichkeit als Namensdienst verwendet werden, sofern ein DNS-Server im Netzwerk verfügbar ist.

## 5.4 Dienstmodell

Das UPnP-Dienstemodell beschreibt den folgenden Aufbau von Devices: Jedes Device wird als Root-Device gesehen, das Dienste und eingebettete Devices (Embedded Devices) enthalten kann. Ein eingebettetes Device kann wiederum Dienste oder andere eingebettete Devices enthalten.

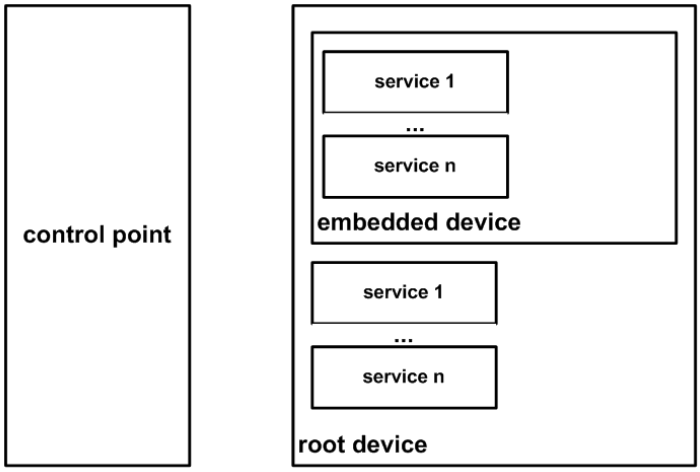



**Abbildung 3: Das UPnP-Dienstemodell**

Device- und Dienstbeschreibungen liegen bei UPnP-Geräten in einem XML-Format vor (UPnP Template Language). Die Spezifikation der UPnP-Architektur [UPnP01] beschreibt mit diesem Format allgemeine Vorlagen für Device- und Dienstbeschreibungen. Die Spezifikationen für verschiedene UPnP-Geräte- und Dienstklassen beschreiben wiederum spezialisiertere Vorlagen, die aber auch noch Platzhalter für herstellerspezifische Informationen vorsehen.

UPnP (aktuell Version 1.0) besitzt einen Versionierungsmechanismus der Form *major.minor*. Dabei wird bei Änderungen an der *minor*-Versionsnummer Abwärtskompatibilität gewährleistet, da Protokolle oder APIs der neueren Version immer eine Obermenge der Protokolle und APIs der früheren UPnP-Version darstellen müssen. Bei Änderungen der *major*-Versionsnummer muss hingegen keine Abwärtskompatibilität gewährleistet sein.

Damit ein Control Point Informationen über ein Device bzw. einen Dienst erhalten kann, muss er wie folgt vorgehen:

- Der Control Point muss in einer Discovery-Nachricht (entweder als Advertisement oder als Antwort auf einen Search-Request, vgl. den Abschnitt Kommunikation – Service Discovery weiter unten) die URL für die Device-Beschreibung erhalten.
- Der Control Point führt einen HTTP-GET-Request mit der URL für die Device-Beschreibung als Ziel durch. Das entsprechende Device antwortet mit einer Nachricht, welche die Device-Beschreibung enthält. Falls diese Antwort nicht innerhalb von 30 Sekunden vorliegt, kommt es auf der Seite des Control Points zu einem Timeout und der Control Point schickt einen erneuten HTTP-GET-Request an das Device.

- Nachdem der Control Point die Device-Beschreibung erhalten hat, kann er die darin enthaltenen Informationen auswerten. Zu diesen Informationen gehören neben den Eigenschaften des Devices (z.B. Hersteller, Gerätetyp) auch eine Liste von Beschreibungen eingebetteter Devices sowie eine Liste enthaltener Dienste. Ein Eintrag für einen Dienst enthält dann je eine URL für die Dienstbeschreibung, den Endpunkt für Kontrollnachrichten und den Ereignisdienst des Dienstes.
- Der Control Point holt sich mit einem HTTP-GET-Request die Dienstbeschreibung und kann daraus die angebotenen Funktionen und Statusvariablen des Dienstes in Erfahrung bringen. Diese Informationen können dann zur Interaktion zwischen dem Control Point und dem Dienst genutzt werden.

Eine Device-Beschreibung enthält folgende Angaben:

- Angabe der UPnP-Version, welche das Device unterstützt.
- Angaben zum Hersteller und zur Modellreihe des Devices.
- Angabe eines über den gesamten Lebenszyklus des Devices hinweg eindeutigen Bezeichners (UUID).
- Angabe der URL die zu der Präsentation des Devices (HTML-Seite) führt.
- Eine Liste bereitgestellter Dienste, wobei zu jedem Dienst folgende Angaben gemacht werden: Service-ID (eindeutig innerhalb des Devices), die URL der Dienstbeschreibung, die URL des Ports für Kontrollnachrichten sowie die URL für den zugehörigen Ereignisdienst.
- Eine Liste von Beschreibungen eingebetteter Devices, deren Aufbau analog zum Aufbau der Beschreibung des Root-Devices ist.

Eine Dienstbeschreibung enthält folgende Angaben:

- Angabe der UPnP-Version.
- Eine Liste aufrufbarer Funktionen inklusive der Beschreibung der Argumente und des Rückgabetyps.
- Eine Liste von Statusvariablen, die per Polling abgefragt oder (optional) als Ereignisquelle für den Ereignisdienst dienen können.

Solange ein Device verfügbar ist (Gültigkeit von Advertisement oder Search-Response noch nicht abgelaufen), kann ein Control Point davon ausgehen, dass auch die zugehörige(n) Beschreibung(en) verfügbar sind.

Möchte ein Device seine Beschreibung verändern, so ist dazu eine Abmeldung und – nach der Änderung – ein erneutes Advertisement nötig.

## 5.5 Kommunikationsmechanismen

### 5.5.1 Service Discovery

Beim Service Discovery geht es darum, dass Control Points für sie interessante Devices und Dienste finden können.

Die Kommunikation beim Service Discovery findet unter Benutzung von UDP-Multicasts (HTTPMU) mit begrenzter Reichweite (Vorgabe TTL = 4) statt. Dabei benutzt UPnP ein zweistufiges Discovery-Verfahren, das sogenannte Simple Service Discovery Protocol (SSDP):

Auf der einen Seite machen Devices auf sich und ihre Dienste gegenüber Control Points aufmerksam, wenn sie dem Netzwerk hinzugefügt werden (sog. Advertisements). Auf der anderen Seite fragen Control Points beim Beitritt zum Netzwerk nach verfügbaren Diensten (sogenannte Search-Requests).

Alle Devices sollten sich, bevor sie das Netzwerk verlassen, wieder abmelden (Cancel-Nachricht), damit sich die Control Points keine veralteten (und damit ungültige) Informationen über sie merken müssen. Ergänzt wird der Mechanismus der expliziten Abmeldung durch ein Leases-Konzept: Wenn sich Devices bekannt machen oder von Control Points entdeckt werden, so wird nur für eine begrenzte Zeitdauer ihre Existenz im Netzwerk vorausgesetzt, eine weiter andauernde Verfügbarkeit setzt eine explizite Verlängerung des Advertisements voraus. Damit werden zwei Ziele verfolgt: Zum einen sollen Informationsleichen vermieden werden, die vom abrupten Entfernen eines Devices aus dem Netzwerk herrühren können. Zum anderen sollen Control Points, falls Advertisement-Nachrichten verloren gehen, nach einer gewissen Zeit doch noch etwas über sämtliche vorhandenen Devices erfahren können.

Die folgende Grafik soll die beiden alternativen Abläufe (ein Device bzw. ein Control Point wird zu einem Netzwerk hinzugefügt) beim Service Discovery verdeutlichen:

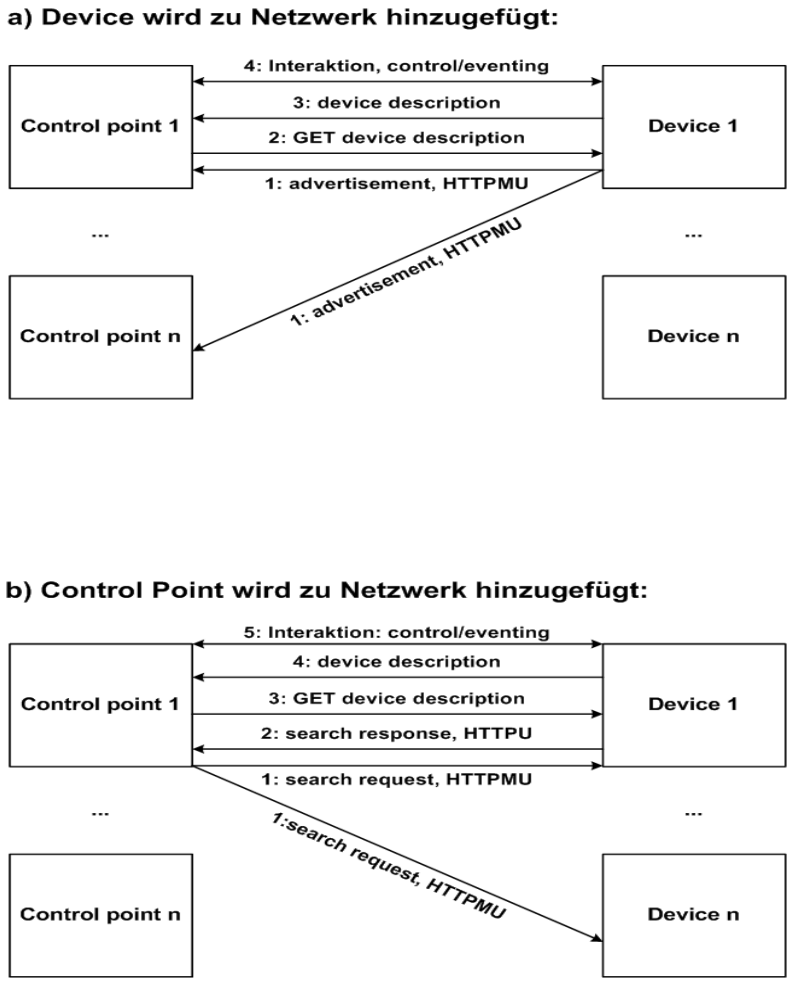



**Abbildung 4: Abläufe beim Service Discovery**

Wird ein Device einem Netzwerk hinzugefügt, so schickt es eine Reihe von Nachrichten (Advertisements) für sich sowie für jeden enthaltenen Dienst und jedes enthaltene Device per UDP-Multicast (HTTPMU) an eine wohlbekannte Adresse (239.255.255.250:1900). Alle am Netzwerk angeschlossenen Control Points müssen an dieser Adresse und an diesem Port hören, ob Advertisement-Nachrichten über neu hinzugekommene Devices ankommen. Damit keine wichtigen Informationen verloren gehen (was bei einem unzuverlässigen Kommunikationskanal, wie ihn UDP bereitstellt, nicht ausgeschlossen werden kann), werden die Nachrichten für jedes Root-Device sowie seine eingebetteten Devices mehrmals verschickt und zwar immer nach Ablauf der Gültigkeitsdauer des Advertisments.

Eine Advertisement-Nachricht wird als HTTP-Nachricht verschickt (sie besteht lediglich aus dem HTTP-Header - der Message-Body fehlt) und besitzt den folgenden Inhalt:

- Klassifizierung der Advertisement-Nachricht (*Notification Type* und *Notification Subtype*) als solche
- URL, unter der die Device-Beschreibung zu finden ist
- Angabe der Gültigkeitsdauer (>= 30 min) des Advertisements

- einen eindeutigen Bezeichner für dieses Advertisement(Advertisement-UUID)

Vor Ablauf der Gültigkeitsdauer soll ein erneutes Advertisement versandt werden, da Control Points bei Ablauf der Gültigkeit eines Advertisements oder einer Abschiedsnachricht die Informationen über das Device (und die eingebetteten Devices und Dienste) aus ihren Caches löschen sollten.

Ein alternatives Verfahren wird angewandt, wenn ein Control Point zu einem Netzwerk hinzugefügt wird. In diesem Fall schickt der entsprechende Control Point einen Search-Request als HTTPMU-Nachricht (wiederum an dieselbe wohlbekannte Adresse, an der sämtliche Devices hören müssen). Dabei hat er die Möglichkeit anzugeben, ob beliebige Devices und Dienste gefunden werden sollen oder nur solche, die einer bestimmten Qualifikation entsprechen (Devices oder Dienste bestimmten Typs, nur Root-Devices, nur Devices mit der angegebenen UUID). Alle Devices, die der angegebenen Qualifikation entsprechen, müssen in einer vom Control Point vorgegebenen Zeit (*maximumWait*, in Sekunden angegeben) auf den Search-Request antworten. Dabei müssen die antwortenden Devices die Antwort um eine Zeitdauer zwischen 0 und *maximumWait* Sekunden verzögern, damit der anfragende Control Point nicht zu viele Antwort-Nachrichten auf einmal verarbeiten muss.

Die Antwort auf einen Search-Request (Search-Response) erfolgt als Unicast-Nachricht (HTTPU) und wird an die im Search-Request angegebene Absenderadresse geschickt. Eine Search-Response hat folgende Informationen als Inhalt: Gültigkeitsdauer für Anwesenheit des Devices, URL für Device-Beschreibung, die Advertisement-UUID sowie den Zeitpunkt der Antwortgenerierung. Auch die Search-Request- bzw. Search-Response-Nachrichten bestehen lediglich aus einem HTTP-Header, ein Message-Body ist nicht vorhanden.

Soll ein Device aus dem Netzwerk entfernt werden, so muss es – wie bereits erwähnt – für jede Search Response-Nachricht bzw. für jede Advertisement-Nachricht eine Abschiedsnachricht schicken. Diese Nachricht wird ebenfalls als HTTPMU-Nachricht an die wohlbekannte Adresse 239.255.255.250:1900 verschickt.

### 5.5.2 Interaktion durch RPCs und Polling

Grundsätzlich gibt es neben der Nutzung des Ereignisdienstes zwei Interaktionsmöglichkeiten zwischen Control Points und Diensten: Zum einen können Control Points von einem Dienst dessen Funktionen aufrufen, zum anderen können Sie die Werte von Statusvariablen erfragen (Polling). Generell werden die Kontrollnachrichten im SOAP-Format über HTTP/TCP verschickt. Die Kontrollnachrichten entsprechen dabei der Konvention für SOAP-RPCs (Remote Procedure Calls, entfernte Prozeduraufrufe) über das HTTP-Protokoll (vgl. [SOAP]). Funktionsaufrufe und Query-Nachrichten (Polling) werden an die in der Beschreibung des entsprechenden Dienstes angegebene URL für Kontroll-Nachrichten (im Folgenden mit Control URL bezeichnet) geschickt, die Antworten an die in den Action-Nachrichten bzw. Query-Nachrichten enthaltenen Quell-URLs. Die Informationen, über welche Funktionen und Status-Variablen ein Device verfügt, bekommt ein Control Point aus den Device- bzw. Dienstbeschreibungen, die er für interessante Devices anfordern kann (vgl. Abschnitt über Service Discovery).

Control Points können davon ausgehen, dass ein Device Kontrollnachrichten verarbeiten und beantworten kann, solange ein Device über ein gültiges Advertisement bzw. eine gültige Search-Response verfügt. Control Points haben somit keine zusätzlichen Maßnahmen zu ergreifen, um die Verfügbarkeit eines Devices bzw. eines Dienstes vor dessen Nutzung sicherzustellen.

**Aufruf von Funktionen (Actions)**

- **Action Requests**: Der erste Schritt bei einem Aufruf einer Funktion besteht in einem SOAP-Request, den der Control Point an die Control URL des entsprechenden Dienstes sendet. Diese Aufrufnachricht enthält im HTTP-Header neben der Control URL als Zieladresse ein SOAPACTION-Headerfeld, das den Diensttyp und den Namen der Funktion angibt. Der Message-Body enthält einen SOAP-Envelope, in dem wiederum die Verwendung eines SOAP-Bodys vorgeschrieben ist. Die Serialisierung des Aufrufs inklusive seiner Argumente erfolgt gemäss der Konvention für SOAP-RPCs (vgl. [SOAP]).
- **Action Responses**: Auch die Antwort auf eine Aufrufnachricht hält sich an die in [SOAP] beschriebene SOAP-RPC-Konvention und enthält entweder den Rückgabewert des Funkionsaufrufs oder eine Fehlermeldung als SOAP-Fault-Element. Eine Action Response muss dem Control Point innerhalb von 30 Sekunden vorliegen, sonst kommt es bei ihm zu einem Timeout. Die Reaktion eines Control Points auf einen Timeout ist herstellerspezifisch und wird somit nicht von der UPnP-Spezifikation [UPnP01] vorgegeben. Damit es bei Funktionen, die einen Vorgang mit einer längeren Zeitdauer realisieren, nicht häufig zu Timeouts kommt, empfiehlt die Spezifikation, schon vor dem Ende der Ausführung eine Antwortnachricht zu generieren, die den Control Point darauf hinweist, dass die Ausführung noch länger dauert und er das Ergebnis zu einem späteren Zeitpunkt als Ereignisnachricht erhalten kann.

**Polling – Query-Nachrichten**

Das Polling des Wertes einer Statusvariablen läuft praktisch analog zum Aufruf von Funktionen ab. Der einzige Unterschied – neben der Semantik natürlich – besteht in einer speziellen Kennzeichnung der Query-Nachricht als solche. Zum einen besitzt das SOAPACTION-Headerfeld (vgl. SOAP-Spezifikation [SOAP]) einen anderen Wert - der Name der aufzurufenden Funktion ist bei Query-Nachrichten immer QueryStateVariable, zum anderen ist das Argument, das im SOAP-Message-Body angegeben wird, immer mit dem Namen varName bezeichnet und beinhaltet den Namen der Statusvariablen, dessen Wert der Control Point in Erfahrung bringen möchte.

### 5.5.3 Ereignisdienst (General Event Notification Architecture, GENA)

Als weitere Interaktionsmöglichkeit neben Funktionsaufrufen und Polling steht bei UPnP ein Ereignisdienst zur Verfügung. Der Ereignisdienst wird nicht von einer zentralen Instanz realisiert, sondern von jedem Dienst wird ein eigener Ereignisdienst angeboten. Als Ereignisse werden Informationen über jegliche Änderungen der an den Ereignisdienst gekoppelten Statusvariablen eines Dienstes an alle Abonnenten versendet. Sämtliche GENA-Nachrichten werden als HTTP-Nachrichten über TCP verschickt.

Die Interaktion zwischen einem Control Point und einem Ereignisdienst (vgl. Abbildung „Abläufe beim Ereignisdienst“ erfolgt gemäss dem Publisher-Subscriber-Modell:

- Zunächst sendet ein Control Point einen sogenannten Subscription-Request an die URL des Ereignisdienstes, die er aus der Dienstbeschreibung (vgl. Kapitel über Dienstbeschreibungen) erhalten hat.
- Daraufhin antwortet ihm der Ereignisdienst mit einer Bestätigung der Subscription, die entweder eine Fehlermeldung oder eine Bestätigung enthält. Die Antwort auf einen Subscription-Request muss dem Abonnenten innerhalb von 30 Sekunden vorliegen, sonst bekommt er einen Timeout. Eine Bestätigung beinhaltet vor allem einen eindeutigen Bezeichner der Subscription (Subscription ID, kurz SID) und eine Gültigkeitsdauer der Subscription.
- Sofort nach der Bestätigung erhält der abonnierende Control Point eine initiale Ereignisnachricht, welche die Werte aller an den Ereignisdienst gekoppelten Statusvariablen des entsprechenden Dienstes enthält.
- Im weiteren Verlauf bekommt ein Abonnent sämtliche Änderungen an den Statusvariablen des Anbieters als Ereignisnachrichten mitgeteilt. Es besteht keine Möglichkeit für den Abonnenten, sich nur auf Änderungen einzelner Variablen zu beschränken.
- Vor Ablauf der Gültigkeitsdauer muss der Abonnent eine Subscription-Renewal unter Angabe der SID durchführen, falls er weiterhin am Ereignisdienst teilnehmen möchte.
- Eine Subscription wird beendet, wenn ein Abonnent entweder aktiv eine Cancel-Nachricht mit Angabe der SID sendet oder wenn er eine Subscription nicht rechtzeitig vor dem Ablauf der Gültigkeit erneuert. Eine Subscription kann auch davon ausgehen, dass eine Subscription mit dem Abschied eines Devices aus dem Netzwerk beendet wird.

Die Verwaltungsnachrichten bei der UPnP-Ereignisverarbeitung, d.h. Subscription-Requests, Subscription-Bestätigungen, Subscription-Renewals und Cancel-Nachrichten, bestehen lediglich aus dem HTTP-Header, die Ereignisnachrichten dagegen enthalten auch einen Message-Body im XML-Format.

Dieser Message-Body enthält als Inhalt sämtliche geänderten Werte von Statusvariablen seit der letzten Ereignisnachricht. Im Header enthalten Ereignisnachrichten außer der Kennzeichnung als solche auch eine Sequenznummer, die der Abonnent bei Erhalt der Ereignisnachricht überprüfen sollte, um sicherzustellen, dass er keine Ereignisnachricht verpasst hat.

Hat ein Abonnent festgestellt, dass er eine Ereignisnachricht verpasst hat, so muss er das Abo abbrechen und einen erneuten Subscription-Request versenden um sicherzustellen, dass er wieder den aktuellen Zustand des Dienstes kennt.

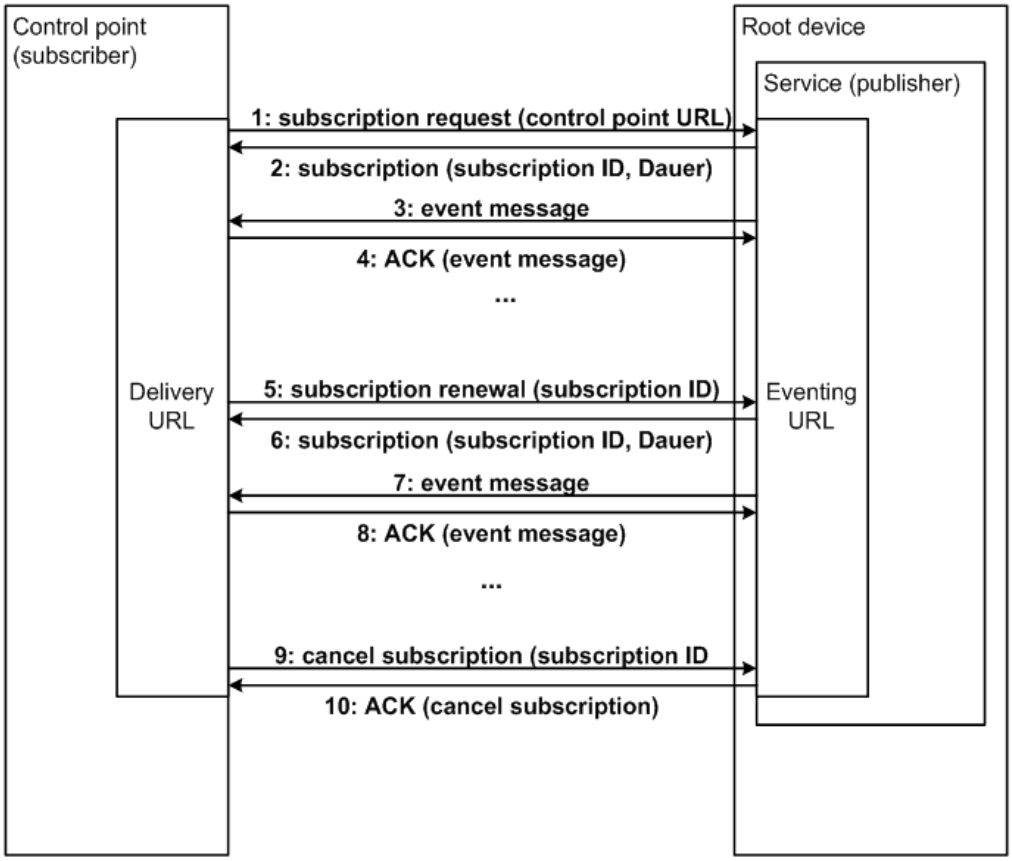



**Abbildung 5: Abläufe beim Ereignisdienst**

Da bei häufigen Änderungen von Statusvariablen sehr viele Ereignisnachrichten das Netzwerk belasten würden, schlägt UPnP mehrere Maßnahmen vor, um die Netzwerk-Belastung zu verringern:

- Variablen, die sich zu häufig ändern, können entweder vom Ereignisdienst ausgeschlossen werden und lassen sich dann nur über den Polling-Mechanismus abfragen oder sie können gefiltert werden. Der Filter-Mechanismus sieht vor, dass bei Wertänderungen einer Statusvariablen nur dann Ereignisnachrichten generiert werden, sofern die letzte Ereignisnachricht, die diese Variable betraf, eine gewisse Mindestzeit zurückliegt oder die Wertänderung eine bestimmte Mindestgröße überschreitet.
- Darüber hinaus sollen Ereignisdienste die Wertänderungen mehrerer Statusvariablen nach Möglichkeit in einer Nachricht zusammenfassen.

### 5.5.4 Streaming: AV Architecture (Version 1.0)

Die UPnP AV Architecture beschreibt Mechanismen zum Transfer von Multimedia-Daten (Streaming) sowie spezielle Device- und Diensttypen dafür. Das Streaming wird wie bei UPnP üblich durch Kontrollnachrichten der Control Points initiiert, läuft dann aber ohne weiteres Zutun durch den Control Point ab. Die dazu verwendeten Transportprotokolle können dieselben sein, die in der UPnP-Gerätearchitektur definiert sind (z.B. TCP, HTTP), es können allerdings auch andere Transferprotokolle wie z.B. IEEE1394 verwendet werden. Dieser Datentransfer muss somit auch nicht auch nicht zwingend über ein IP-Netzwerk oder IEEE1394 erfolgen – es wird neben dem Kanal für die UPnP-Kontrollnachrichten lediglich irgendein weiterer Kommunikationskanal für den Multimedia-Datenstrom benötigt.

Mit Media Servern und Media Renderern werden zwei spezialisierte Device-Typen eingeführt, deren enthaltene Dienste standardisiert sind.

Media Server sind Devices, die anderen UPnP-Geräten im Netzwerk Multimediainhalte (Bilder, Audio und Video) zur Verfügung stellen. Media Server stellen Funktionen zur Verfügung, mit denen sie Auskunft über die bereitgestellten Inhalte sowie über die von ihnen unterstützten Datenformate und Transportprotokolle geben. Ferner können sie ihre Inhalte zu anderen UPnP-Geräten übertragen oder Inhalte von anderen UPnP-Geräten importieren.

Media Renderer besitzen im Gegensatz dazu die Fähigkeit, Multimediainhalte bestimmter Datenformate wiederzugeben (Rendering). Um dies zu ermöglichen, stellen sie ebenso wie Media Server Informationen über die von ihnen unterstützten Datenformate und Transportprotokolle zur Verfügung. Media Renderer können auf Anforderung eines Control Points hin Multimediainhalte von einem Media Server beziehen und anzeigen.

Um diese Funktionalität auf eine einheitliche Art und Weise zu realisieren, sind sowohl Media Server als auch Media Renderer in ihrem Aufbau durch entsprechende Device-Beschreibungen im XML-Format standardisiert und müssen bestimmte Standard-Dienste für das Streaming von Multimediainhalten implementieren. Für diese Standard-Dienste ist die Programmierschnittstelle (API) durch vordefinierte Statusvariablen und Funktionen genau definiert. Die Standard-Dienste werden im folgenden kurz besprochen:

- **Content Directory**: Verzeichnisdienst, stellt Informationen über die von einem Media Server bereitgestellten Inhalte (z.B. Musikstücke, Videos, Fernsehprogramme ) zur Verfügung. Diese Inhalte können traversiert (Browsing) oder gezielt durch Angabe einer Suchmaske durchsucht werden. Ferner können Inhalte auch gelöscht, importiert oder neue Inhalte angelegt werden.
- **Rendering Control**: Stellt ein API bereit, um die Attribute eines Rendering-Vorgangs (z.B. Helligkeit, Kontrast, Lautstärke) zu identifizieren und zu konfigurieren.
- **Connection Manager**: Stellt Informationen über die von Media Servern bzw. Media Renderern unterstützten Transportprotokolle und Datenformate zur Verfügung. Diese Informationen können gemeinsam mit den Informationen über den Inhalt von einem Control Point bzw. einen Benutzer zur Herstellung einer geeigneten Streaming-Verbindung genutzt werden. Der Connection Manager ermöglicht es auch, neue Transferverbindungen aufzubauen (diese werden dann bei den beteiligten Connection Managern registriert) bzw. bestehende Verbindungen zu beenden. Daher können über

den Connection Manager auch Informationen über momentan aktive Datentransfers in Erfahrung gebracht werden.

- **AV Transport**: Ermöglicht es einem Control Point den Transport eines Streams zu kontrollieren (z.B. Play, Pause, Stop) und Informationen über den gerade transferierten Inhalt zu erhalten.

Die folgende Abbildung zeigt den Aufbau von Media Server und Media Renderer sowie deren Zusammenspiel mit einem Control Point.

Die Wiedergabe von Multimediadaten wird von einem Control Point durch Funktionsaufrufe veranlasst, der eigentliche Datentransfer (Streaming) findet dann allerdings ohne weiteres Zutun des Control Points statt (ein Control Point kann allerdings die Wiedergabe und damit das Streaming durch Aufrufe an einen AV Transport-Dienst steuern). Der Transfer zwischen Media Server und Media Renderer kann sowohl über isochrone Transportprotokolle wie IEEE1394 oder über asynchrone Transportprotokolle (z.B. HTTP) stattfinden. Bei der Verwendung isochroner Transportprotokolle ergibt sich dank QoS-Garantien (z.B. garantierte Bandbreite) eine einfachere Implementierung des Media Renderers (es muss kein Readahead-Puffer bereitgestellt werden).

Darüber hinaus kann der Transfer sowohl im Push- als auch im Pull-Modus durchgeführt werden. Im Push-Modus initiiert der Media Server den Transfer, der durch den AV Transport-Dienst des Media Servers kontrolliert werden kann. Im Pull-Modus initiiert der Media Renderer den Transfer und fordert vom Media Server die Datenpakete an.

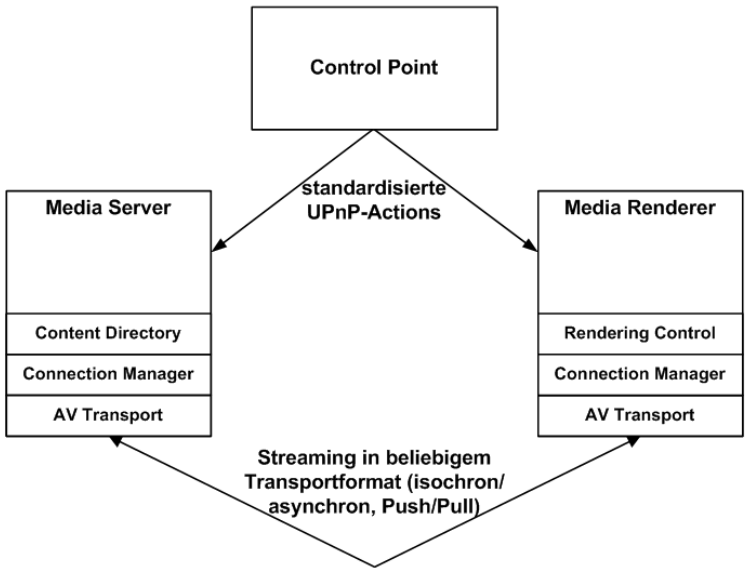



**Abbildung 6: UPnP-Architektur zur AV-Wiedergabe**

Der Aufbau und die Durchführung einer Streaming-Verbindung laufen allgemein nach folgendem Muster ab:

- Der Control Point entdeckt die beteiligten Devices (Media Server und Media Renderer) durch den UPnP-Discovery-Algorithmus.
- Der Control Point sucht nach den gewünschten Inhalten und holt sich Informationen über unterstützte Transportprotokolle und Datenformate von den betreffenden Media Servern.
- Der Control Point sucht passende Media Renderer anhand deren unterstützten Transportprotokollen und Datenformaten.
- Nach der Auswahl eines Media Servers (bzw. des entsprechenden Inhaltes) und eines Media Renderers veranlasst der Control Point den Media Server und den Media Renderer, die Verbindung aufzubauen.
- Media Server und Media Renderer führen eigenständig den Datentransfer durch, dabei hat der Control Point die Möglichkeit, die Wiedergabe zu steuern (über den entsprechenden AV Transport-Dienst) oder das Rendering zu beeinflussen (über entsprechenden Rendering Control-Dienst).
- Nach dem Ende der Wiedergabe veranlasst der Control Point Media Server und Media Renderer als letzten Schritt die Verbindung wieder abzubauen.

## 5.6 Anmerkungen und Schwachpunkte

UPnP baut ausschließlich auf verfügbare Internet-Technologien wie IP, HTTP, SOAP, etc. auf. Diese Strategie hat sowohl Vor- als auch Nachteile. So sorgt die Verwendung dieser Technologien auf der einen Seite mit Sicherheit für eine hohe Akzeptanz und Interoperabilität. Auf der anderen Seite muss dadurch auch in zu vernetzenden Kleingeräten ein vollständiger IP-Stack und - für Geräte, welche Dienste anbieten - ein HTTP-Server bereitgestellt werden, was unter Umständen diese Geräte stark verteuert. Dadurch, dass UPnP kein API spezifiziert, sind die UPnP-Entwickler völlig frei in der Wahl der Implementierungssprache, allerdings wird die Portabilität von UPnP-Komponenten dadurch erschwert.

## 5.7 Verfügbare Implementierungen

| Firma / Produktname | Weitere Informationen |
|---|---|
| **Allegro** ROMUPnP UPnP Toolkits (C): | http://www.allegrosoft.com/romupnp.html |
| **Atinav** aveLink UPnP (Java/C): | http://www.avelink.com/modules/upnp.htm |
| **Intel** Software for UPnP technology (Microsoft .NET, C): | http://www.intel.com/labs/connectivity/upnp/ |
| **Lantronix** UPnP Early Adopters Kit: | http://www.lantronix.com/support/utils/upnp/index.html |
| **Metro Link** EnableWorks UPnP Device SDK (C): | http://www.metrolink.com/solutions/upnp.html |
| **Microsoft** Device Kit for UPnP Devices (Microsoft Visual C++): | http://www.microsoft.com/hwdev/upnp |
| **ProtoSys** SDK for UPnP Devices and Control Points (Java/C): | http://www.protosys.com |
| **Siemens** SDK for UPnP Devices (Java/ C++): | http://www.plug-n-play-technologies.com |

**Tabelle 1: UPnP-Implementierungen**

Daneben enthalten Microsoft Windows ME und Microsoft Windows XP UPnP-Control Point Funktionalität. Windows XP stellt darüber hinaus die Funktionalität eines UPnP Internet Gateway Devices (IGD) bereit, über den andere UPnP-Geräte Internet-Zugriff erhalten können.

# 6 Java-Technologien

In folgenden Abschnitten werden die drei Java-Technologien – Jini, JMS und JMF – vorgestellt und beschrieben. Jini bietet eine Infrastruktur zur spontanen Vernetzung von Geräten/Diensten an, unterstützt aber keine Übertragung von Multimedia-Daten. Für die Übertragung und Verarbeitung von solchen Daten ist Java Media Framework (JMF) zuständig. JMS stellt Mechanismen zur asynchronen Kommunikation bereit, die in der Realisierung von der spontanen Geräte-Vernetzung einsetzbar sind. Allerdings gehen die von JMS angebotenen Mechanismen weit über diese Anforderungen hinaus und unterstützen beispielsweise die transaktionelle Verarbeitung der Nachrichten, wie sie im Umfeld der Unternehmenssoftware erforderlich und gebräuchlich ist.

Durch die Kombination der hier beschriebenen Java-Technologien entsteht eine Architektur zur Gerätevernetzung, die einen vergleichbaren Umfang wie beispielsweise UPnP oder HAVi bietet.

Es bietet sich darüber hinaus an, das Java Media Framework mit OSGi zu integrieren, da OSGi selbst ebenfalls die Unterstützung zur Verarbeitung von Multimedia-Daten fehlt.

## 6.1 Java Intelligent Network Infrastructure (Jini)

### 6.1.1 Begriffsdefinitionen

In dieser Tabelle wurden die wichtigsten Begriffe zusammengefasst und erläutert.

| Begriff | Definition |
|---|---|
| Dienst | Ein Dienst stellt Funktionen bereit, die von anderen Diensten (wie Geräte, Software, Datenbanken, Dateien) oder Nutzern verwendet werden können. Er wird als Java Objekt realisiert. Der Typ und die Methoden des Objektes stellen die Schnittstelle dar, über die der Dienst aufgerufen werden kann. |
| Discovery Protokoll | Das Protokoll zum Auffinden eines LUS in einem Jini-System. Es stellt drei verschiedene Alternativen bereit, um einen LUS zu finden: das Unicast Discovery Protokoll, das Multicast Request Protokoll und das Multicast Announcement Protokoll. |
| Join Protokoll | Mittels dieses Protokolls erfolgt die Anmeldung der Dienste bei einem LUS. |
| Leasing | Unter diesem Konzept wird die Vergabe der Ressourcen – in diesem Fall die Registrierung bei einem Lookup Service – für eine bestimmte Zeitdauer verstanden. |
| Lookup Service (LUS) | Eine zentrale Registrierungsinstanz in einem Jini-System. An den LUS wendet sich ein Dienstanbieter, um seinen Dienst im Netz zu offerieren, und ein Interessent, um den gewünschten Dienst zu finden. |
| Remote Events | Die Dienste können einander Mitteilungen über ihre Statusänderungen machen (jede Jini-Komponente kann von den anderen Ereignisse empfangen). |

### 6.1.2 Überblick

**Wozu ist Jini gut?**

Geistiger Urheber des Jini Projekts ist Bill Joy, Mitbegründer und Vice-President der *Sun Microsystems*, Inc. 1994 begann er in den *Sun* Aspen Smallworks Laboratorien in Colorado mit der funktionalen Definition Jinis. In der zum gleichen Zeitpunkt entwickelten Java Technologie sah Joy das Potential, Leistung und Funktionalität in allgegenwärtigen Netzen anzubieten. Das Java Computing-Modell bietet sich als Plattform für die verteilte Verarbeitung an, da sowohl Daten als auch ausführbare Programme beliebigen Netzknoten dynamisch zugeordnet und dort ausgeführt werden können. Jini Software ist kompakt und verfügt über eine klare Architektur. Nur 48 KB Java Binär-Code werden gebraucht, um die gesamte Funktionalität zu bieten [Jini01].

Jini ist die auf Java-Technologie basierte Architektur für spontane Vernetzung von Geräten/Diensten. Unter spontaner Vernetzung (”spontaneous networking”) wird die Möglichkeit verstanden, Geräte/Dienste ohne vorherige Konfiguration in ein Netzwerk einzubinden und alle dort vorhandenen Dienste nutzen zu können.

Zu beachten ist, dass es sich bei Jini um eine Architektur handelt, nicht aber um eine Implementierung. Diese Architektur ist durch die Jini-Spezifikation definiert. Weiterhin existiert eine Referenzimplementierung dieser Spezifikation, welche durch die Firma *Sun Microsystems* bereitgestellt wird.

Wenn Hardware ans Netz angeschlossen wird, kann sie ohne Mühe sofort benutzt werden. Das gleiche gilt auch für Software: man kann sich ein Textverarbeitungsprogramm vorstellen, das zur Rechtschreibprüfung ein Wörterbuch konsultiert (vorausgesetzt, die beiden sind „jinifähig“). Stellen Sie sich vor, Sie sind in einem Konferenzraum, haben Ihr Notebook an das unbekannte Netz angeschlossen und wollen ein paar Seiten drucken. Sie wählen den Druck-Modus aus, ein Dialogfenster erscheint und der Drucker steht Ihnen zur Verfügung ohne jegliche Konfigurationsaufwand. Was sich hinter diesem Ablauf verbirgt, wird im folgenden Abschnitt erläutert.

### 6.1.3 Infrastruktur und Programmiermodell

Um Jini zum Einsatz zu bringen, wurde das Java-Programmiermodell ergänzt und zu Java und RMI kamen einige Infrastrukturkomponenten hinzu. Dienste (in Jini ist eigentlich alles ein Dienst) werden in die Lage versetzt, einander und vor allem auch den *Lookup Service* (*LUS*) ohne Kenntnis des Netzes zu finden (”bootstrapping”). Den Rest besorgt die Jini-Infrastruktur: Dienste melden sich über ein ”*Discovery*&*Join*-Protokoll” bei einem (ihnen meist vorher nicht bekannten) *LUS* an, dieser sorgt nötigenfalls für die Konfiguration, und schon können die Dienste verwendet werden. Dienstnutzer finden Diensteanbieter ebenfalls über den *LUS*: sie durchsuchen diesen nach einem Interface, das z.B. eine Rechtschreibprüfung implementiert. Werden sie fündig, bekommen sie vom *LUS* ein ”Proxy-Objekt”, das als Stellvertreter des Dienstes die gesamte Kommunikation mit dem eigentlichen Dienst übernimmt. Wie diese wiederum realisiert ist, ist Sache des Dienstes. Für den Dienstenutzer scheint alles lokal vorhanden zu sein. Der *Lookup Service* ist dabei die zentrale Anlaufstelle, um Dienstanbieter und Dienstnutzer zusammenzuführen.

Das Java-Programmiermodell wird von Jini um einige einfache APIs erweitert; so gesellen sich zu bekannten Komponenten wie Beans oder Swing noch *Leasing*, *Transaktionen* und *Distributed*

*Events*. Da Jini eine allgemeine Architektur ist, sind die Strukturelemente keine festen Klassen, sondern Interfaces: Dienste implementieren sie nach ihren Bedürfnissen, werden dadurch aber gezwungen, "jinifähig" zu sein, d.h. die Art der Interaktion zwischen den Diensten wird festgelegt, nicht jedoch deren Implementierung an sich.

Das Kommunikationsprotokoll zwischen einem Druckerdienst und dem Drucker kann proprietär sein, für den Benutzer ist das gleichgültig.

### 6.1.4 Kommunikationsmechanismen

**Voraussetzungen**

Jedes Gerät, das ans Jini-System angeschlossen wird, muss folgenden Anforderungen genügen [Jini02]:

- es muss über eine JVM verfügen,
- eine IP-Adresse besitzen (unter der Annahme, dass IP benutzt wird),
- Unterstützung von Unicast TCP und Multicast UDP anbieten,
- Unterstützung beim Download von Java RMI-Stubs (z.B. mittels eines HTTP-Servers) ermöglichen.

**Spontane Vernetzung Schritt für Schritt**

Die zentrale Anlaufstelle in einem Jini-System ist der *Lookup Service*. An ihn wendet sich ein Dienstanbieter, um seinen Dienst im Netz zu offerieren, und ein Interessent, um an den gewünschten Dienst zu gelangen. Die "Bootstrap"-Phase des Dienstanbieters besteht aus folgenden Schritten:

- **Discovery**: Der Dienstanbieter sucht mittels *Discovery*-Protokoll nach erreichbaren *LUS*. Da *Lookup Services* vollständige Jini-Dienste sind, bieten sie ihre Dienste ebenfalls über Proxies an. Diese werden von den gefundenen *LUS* an den Dienstanbieter übermittelt.
- **Join**: Der Dienstanbieter registriert seinen Dienst im *LUS*. Dabei wird der Dienst-Proxy dort als serialisiertes Objekt hinterlegt.
- **Leasing**: Die Registrierung ist nur für eine bestimmte Zeit vereinbart. Nach Ablauf dieser Zeit wird er vom *LUS* automatisch entfernt. Soll der Eintrag länger bestehen bleiben, so muss der Dienstanbieter den *Lease* für den Eintrag rechtzeitig verlängern. Durch das *Leasing*-Konzept wird verhindert, dass der *LUS* nach einiger Zeit viele ungültige Einträge von Diensten, die nicht mehr verfügbar sind, enthält.

Damit ist der Dienst im Netz verfügbar. Zum Auffinden und Benutzen eines Dienstes sind folgende Schritte nötig:

- **Discovery**: Der Interessent sucht zuerst, genau wie der Dienstanbieter, via *Discovery*-Protokoll einen *LUS* und erhält dessen Proxy.
- **Lookup:** Der Interessent gibt eine Beschreibung des gesuchten Dienstes an den Proxy des *LUS*. Der *LUS* durchsucht seine Diensteinträge nach passenden Kandidaten und übermittelt die Proxies der gefundenen Dienste an den Dienstnehmer.

- **Use**: Der Dienstnehmer kann nun in den Proxies wie in lokalen Java-Objekten Methoden aufrufen und so den Dienst benutzen. Jegliche Kommunikation zwischen Proxy und Dienst bleibt dabei für ihn transparent.

Beginnend auf der Dienstseite sind für die Umsetzung folgende Schritte notwendig:

- Definition eines Java-Interfaces als Schnittstelle zwischen Dienst, Dienstnehmer und dessen Implementierung
- Erzeugen eines Proxy-Objekts
- Anmelden des Dienstes.

Der Pseudocode für den Dienstnutzer wird dann etwa so aussehen:

```
prepare for discovery
discover a lookup service
prepare a template for lookup search
lookup a service
call the service
```

Der Pseudocode für die Dienstseite wird folgender Maße aussehen:

```
prepare for discovery
discover a lookup service
create information about a service
export a service
renew leasing periodically
```

### Die Discovery Protokolle

Die *Discovery* Protokolle ermöglichen Diensten und Dienstnutzern das Auffinden der *Lookup Services* in einem Jini-System. Jini bietet eine Infrastruktur auf Dienstebene an, d.h. sie kann nicht dazu dienen, IP-Addressen und Subnetz-Masken von Rechnern zu konfigurieren, sondern setzt die Konfiguration auf niedriger Ebene voraus.

#### *Unicast Discovery Protocol*

Voraussetzung ist, dass die Adresse des *Lookup Services* bereits bekannt ist. Der LUS wird direkt über eine TCP-Verbindung auf Port 4160 kontaktiert. Als Antwort überträgt der *LUS* sein Proxy-Objekt und seine Gruppenzugehörigkeiten.

#### *Multicast Request Protocol*

Der Einsatz von Multicast-IP ermöglicht es, *Lookup Services* ohne Kenntnis ihrer konkreten IP-Adresse zu finden. Dazu werden UDP-Datagramme an die Multicast-Gruppe 224.0.1.85 und Port 4160 gesendet. Sie enthalten die Gruppennamen, an denen man interessiert ist, und *Service IDs* der bereits bekannten *LUS*. Ein *LUS* antwortet nur dann, wenn er noch nicht bekannt ist und zu einer der genannten Gruppen gehört. Er baut dazu eine TCP-Verbindung auf und übermittelt die Antwort wie im *Unicast Discovery Protocol*.

***Multicast Announcement Protocol***

Über das *Multicast Announcement Protocol* machen sich der *Lookup Service* im Netz bekannt. Dazu werden UDP-Datagramme an die Multicast-Gruppe 224.0.1.84:4160 geschickt. Sie enthalten die *Service ID*, die Netzadresse des *LUS* und dessen Gruppenzugehörigkeit. Falls der *LUS* noch nicht bekannt ist und eine interessante Gruppe verwaltet, fordern interessierte Teilnehmer den *LUS*-Proxy über das *Unicast Discovery Protocol* an.

Normalerweise werden die beiden Multicast-Protokolle verwendet, um einen *Lookup Service* im lokalen Netzsegment zu finden. Der Vorteil liegt vor allem im geringeren Konfigurationsaufwand: Der Benutzer muss in seinen Diensten und Anwendungen keine *LUS*-Adressen eintragen. Das verbirgt sich u.a. hinter dem Begriff "spontaneous networking". Die Multicast-Protokolle haben jedoch nur eine begrenzte Reichweite, da die meisten Router so konfiguriert sind, dass sie keine Multicast-Pakete weitergeben. Für Verbindungen über solche Netzsegmente hinaus, muss das *Unicast Discovery Protocol* verwendet werden.

### 6.1.5 Dienstbeschreibungen

Einem Dienst können beim Eintrag in einen *Lookup Service Attribute* zugeordnet werden, die den Dienst näher beschreiben oder Programmcode enthalten. Beispiele für beschreibende *Attribute* sind Herstellername, Versionsnummer des Dienstes oder ein Icon zur Darstellung in graphischen Browsers.

Neben diesen allgemeinen lassen sich auch spezielle, auf bestimmte Dienstklassen abgestimmte *Attribute* definieren. Für einen Druckdienst könnte man so zusätzlich angeben, ob es sich um einen Laser- oder Tintenstrahldrucker handelt. *Attribute* ermöglichen es, einerseits die Anzahl der Treffer einer Suchanfrage möglichst gering zu halten, andererseits einem menschlichen Benutzer Informationen für die Auswahl des "richtigen" Dienstes anzubieten.

Realisiert werden *Attribute* durch Java-Objekte. Jedes *public*-Feld einer Klasse, die von *Entry* bzw. *AbstractEntry* abgeleitet ist, wird dabei als *Attribut* interpretiert.

Ein *Entry*-Objekt stellt eine Menge von Attributen dar. Da einem Dienst mehrere solche Objekte zugeordnet werden können, speichert der *LUS* zu jedem Dienst demnach eine Menge von Attributmengen.

Attribute können auch nach der Registrierung noch geändert werden, um Zustandsänderungen des Dienstes wiederzugeben. Beispielsweise können so Statusinformationen in Attributen in den *LUS* hinterlegt werden (z.B. Fehlerzustände von Geräten). Es wird jedoch davon ausgegangen, dass *Attribute* eher statischen Charakter haben. Aus Effizienzgründen wird empfohlen, sie nicht öfter als 1 mal pro Minute zu ändern.

Da die *Attribute* in normalen Java-Klassen gespeichert werden, können diese auch mit Programmcode versehen sein. Damit ergibt sich eine weitere interessante Anwendungsmöglichkeit der Attribute, nämlich das Hinterlegen eines GUI zur Benutzung des Dienstes. Dieser Ansatz erlaubt es, dass andere Dienste und Programme den Dienst ganz normal über das Java-Interface ansprechen können. Zusätzlich bringt der Dienst für den Fall, dass ein menschlicher Benutzer ihn direkt verwenden will, sein eigenes GUI gleich mit.

### 6.1.6 Gruppen

Jini sieht vor, dass Dienste in Gruppen organisiert werden können. Jeder Dienst kann zu beliebig vielen Gruppen gehören. Beispielsweise könnte ein Drucker sich in den Gruppen ”Hardware”, ”Drucker” und ”Konferenzraum 2” anmelden oder es werden verschiedene Dienste zusammengefasst, weil sie gemeinsam eine Aufgabe lösen. Gruppen werden eingerichtet, indem man LUS die entsprechenden Gruppennamen zuweist. Bei *Sun* Referenzimplementierung müssen sie als Kommandozeilenparameter beim Start des *LUS* angegeben werden.

Der Name *public* ist als Standard-Gruppenname vergeben. Ein Dienstanbieter ist selbst für eine passende Registrierung in den *LUS* verantwortlich. Er muss seinen Dienst also in jedem ihm bekannten *LUS* genau dann registrieren, wenn der *LUS* mindestens eine Gruppe verwaltet, zu der dieser Dienst gehören soll. Daneben hat *Sun* auch ein *Admin*-Interface vorgesehen. Implementiert ein Dienst dieses Interface, so kann dessen Gruppenzugehörigkeit während der Laufzeit z.B. über den mitgelieferten Browser geändert werden. Bei den *LUS* kann man zusätzlich noch die Gruppen ändern, die sie verwalten sollen. Beim Aufruf registriert der *JoinManager* den Dienst in allen gefundenen *LUS* und damit auch in allen Gruppen. Indem man einen anderen Konstruktor des *JoinManagers* verwendet, kann man die Registrierung des Dienstes auf bestimmte Gruppen einschränken.

### 6.1.7 Das Leasing-Konzept

Im täglichen Leben wird beim Leasing durch einen der Vertragspartner dem anderen Vertragspartner die Nutzung einer Ressource für einen bestimmten, vorher definierten Zeitraum zugesichert. Dieses Modell lässt sich auch auf das *Leasing*-Interface bei Jini übertragen. Hier wird von einem Diensteanbieter dem *Lookup Service* zugesichert, dass der angebotene Dienst für eine bestimmte Zeit genutzt werden kann.

Das von Jini genutzte *Leasing*-Konzept entspricht dem von RMI aus dem JDK 1.2. Die dort definierte Klasse *Lease* dient dazu, den Gültigkeitszeitraum eines *Remote*-Objects zu spezifizieren. Läuft dieser Zeitraum ab, ohne dass die *Lease*-Time durch das Objekt verlängert wurde, wird das Object durch den automatischen Java Garbage-Collector entfernt.

Die Verwendung eines *Leasing*-Konzeptes ist notwendig, da die Verbindung zwischen zwei Objekten in einem verteilten System jederzeit ausfallen kann, sei es durch einen Netzwerkausfall oder durch den Ausfall eines der Objekte selbst. Gleichzeitig muss ein Objekt (im folgenden „Client“ genannt) aber zunächst von der Existenz eines *Remote*-Objects erfahren, bevor es dessen Funktionen nutzen kann. Dazu ist eine Registrierung des entfernten Objekts nötig, entweder bei einem *Lookup Service* (dieses Konzept nutzen *RMI* und Jini, daher wird im folgenden von der Nutzung dieses Ansatzes ausgegangen) oder direkt bei dem Client.

Bei dieser Registrierung werden gleichzeitig die Bedingungen für das Leasing ausgehandelt, d.h. das *Remote*-Object teilt innerhalb der vom *Lookup Server* festgelegten Grenzen mit, wie lange es voraussichtlich verfügbar sein wird. Läuft diese Zeit ab, muss das Objekt eine Verfügbarkeitsmeldung schicken, damit die *Leasing*-Zeit um den am Anfang definierten Zeitraum verlängert wird. Bleibt die Meldung aus, so wird die Registrierung vom Server gelöscht, um die verwendeten Ressourcen wieder freizugeben. Alle Clients, die das *Remote*-Object benutzen, werden von dessen Ausfall informiert.

Probleme können entstehen, wenn ein *Remote*-Object innerhalb seiner *Leasing*-Zeit ausfällt. Dieser Fall kann vom *LUS* nicht abgefangen werden und bedarf deshalb einer Fehlerbehandlung innerhalb des Clients. Ebenso entsteht durch die Verfügbarkeitsmeldungen eine je nach Anwendung und Netzkapazität nicht unerhebliche Netzlast, die beim Entwurf der Anwendung bedacht werden muss.

Das *Leasing*-Konzept lässt sich also durch folgende Punkte charakterisieren:

- Es existiert ein *Leasing*-Zeitintervall, innerhalb dessen ein Diensteanbieter dem Dienstenehmer seine Verfügbarkeit zusichert.
- Der Anbieter kann seinen Dienst innerhalb des Intervalls beim Dienstnehmer kündigen.
- Das *Leasing*-Intervall kann erneuert werden, wenn der Dienst weiterhin zur Verfügung steht.
- Das *Leasing*-Intervall kann verfallen, wenn ein Dienst nicht mehr zur Verfügung steht.

Bei Jini sind diese vier Punkte durch das Interface *net.jini.core.lease.Lease* realisiert. Jeder Diensteanbieter muss dieses Interface implementieren, um die Basisfunktionalitäten zur Verfügung stellen zu können.

Durch die Konstante *FOREVER* kann festgelegt werden, dass die *Lease*-Time des Dienstes niemals abläuft. Die Konstanten *DURATION* und *ABSOLUTE* definieren die *Leasing*-Zeit relativ bzw. absolut, jeweils in Millisekunden.

Über die Methode *getExpiration* kann die restliche Zeitspanne bis zum Ablaufen des *Leasing*-Intervalls abgefragt werden. Mit *cancel* kann der Diensteanbierter seinen Dienst beim *Lookup Service* kündigen, mit *renew* kann er das Intervall verlängern.

Wie bei Java üblich, kann ein Entwickler das Interface beliebig erweitern. Dadurch wird eine hohe Flexibilität bei der Implementierung erreicht.

### 6.1.8 Remote Events

Das Ereignismodell von Jini realisiert eine asynchrone Notifikation über mehrere JVM's. Dabei besteht keine Transportgarantie, da bei Jini keine zentrale Instanz existiert und somit partielle Fehler auftreten können. Es wird hierbei eine Methode *notify()* vom *RemoteEventListener* benutzt. Dafür wird der Interface *ReliableDeliveryDelegate* benutzt. Zur gesicherten Weitergabe können Ereignisse (Crash-sicher, Racing-sicher) in Jini in einer Ereignis-Pipeline bei Jini delegiert werden.

### 6.1.9 Jini heute

Die Jini Technologie wurde in einer Beta Version einer Reihe von Unternehmen, darunter *Computer Associates*, *Epson*, *Ericsson*, *FedEx*, *Novell*, *Quantum*, *Salomon Brothers* und *Toshiba* für Testzwecke zur Verfügung gestellt [Jini01]. Der Erfolg der Jini Technologie wird von der allumfassenden Verfügbarkeit und den innovativen Beiträgen der Entwickler vieler Unternehmen in vielen Branchen abhängig sein. *Sun* wird daher ein Distributionsprogramm aufsetzen, das diesen Gruppen Jini frei verfügbar macht. Zusätzlich wird der Source Code unter einer Jini Technology Public Licence (JTPL) angeboten.

## 6.2 Java Message Service (JMS)

Das folgende Kapitel gibt einen Überblick über Modelle der nachrichtenbasierten Kommunikation im Allgemeinen und beschreibt das Programmiermodell und grundlegende Interfaces des Java Message Services.

### 6.2.1 Begriffsdefinitionen

In dieser Tabelle wurden die wichtigsten Begriffe zusammengefasst und erläutert.

| Begriff | Definition |
|---|---|
| Consumer | Empfänger von Nachrichten |
| Destination | Die Nachrichtenvermittlung, über *Queue* oder *Topic* realisiert |
| Message Broker | Ein System, das eine Infrastruktur zur Vermittlung asynchroner Nachrichten realisiert. |
| Producer | Ersteller von Nachrichten |
| Queue | Präsentiert *Pont-to-Point*-Kommunikationsmodell |
| Topic | Präsentiert *Publish-and.Subscribe*-Kommunikationsmodell |

### 6.2.2 Überblick

Die erste Version der JMS-Programmierschnittstelle wurde 1998 von *Sun Microsystems* in Zusammenarbeit mit einer Gruppe von Firmen erarbeitet und veröffentlicht. Die Motivation war, die Technologie der nachrichtenorientierten Kommunikation – über Message-Broker – auch für die Java-Welt verwendbar zu machen. Die Spezifikation ist aktuell in der Version 1.1 verfügbar.

JMS definiert eine einheitliche, Java-basierte Schnittstelle (API) für den Zugriff auf Message-Oriented Middleware (MOM) Systeme. In der jetzigen Ausprägung stellt JMS eine clientseitige Programmierschnittstelle dar. JMS legt fest, wie Java-Clients auf Message-Systeme zugreifen können. Es wird nicht definiert, wie ein *Message-Broker* auszusehen hat. Ein Vorteil dabei ist, dass bei Bedarf das verwendete Messaging-System ausgetauscht werden kann, ohne dass der JMS-Client angepasst werden muss.

JMS unterstützt das *Point-to-Point-* und das *Publish-and-Subscribe-* Kommunikationsmodell. Diese beiden Modelle unterscheiden sich konzeptionell, technisch gesehen besitzen sie doch einige Gemeinsamkeiten. Aus diesem Grunde basieren *Point-to-Point* und *Publish-and-Subscribe* im JMS auf einem generischen Kommunikationsmodell.

**Ziele von JMS**

Das JMS Message Model definiert folgende Ziele [JMS01]:

- ein einheitliches Messaging API zur Verfügung stellen
- das API sollte die gängigen Nachrichtenformate unterstützen
- Unterstützung von heterogenen Applikationen
- Unterstützung von Nachrichten, die Java Objekte enthalten
- Unterstützung von Nachrichten, welche XML-Seiten enthalten
- Unterstützung von bestehenden MOM-Systemen, wie z.B. IBM MQSeries.

### 6.2.3 Messaging

**Kommunikationsparadigmen**

Stellen wir die beiden Kommunikationsparadigmen „entfernte Methodenaufrufe“ und „nachrichtenbasierte Kommunikation“ gegenüber, um die Unterschiede zwischen ihnen zu erläutern. Wir verwenden dabei abstrakt die Begriffe „Sender“ für ein Objekt oder einen Prozess, das/der einen RPC-Aufruf oder eine Nachricht absetzen kann und „Empfänger“ für ein Objekt oder einen Prozess, das/der einen Aufruf entgegennehmen und verarbeiten kann.

Wie läuft die synchrone Kommunikation zwischen verteilten Objekten ab? Insgesamt sind vier Schritte zu beobachten:

- Der Sender ruft eine Methode beim Empfänger auf. Der Methodenaufruf wird über das Netzwerk zum Empfänger übertragen.
- Der Empfänger erhält den Aufruf und führt die gewünschte Methode aus.
- Der Sender bleibt während der Ausführung der Methode blockiert. Wenn die Methode im Empfänger ausgeführt worden ist, wird eine Quittung – mit einem Rückgabewert – an den Sender zurückgeschickt.
- Der Sender erhält die Quittung seines Methodenaufrufes und kann mit den folgenden Bearbeitungsschritten fortfahren.

Die im Schritt 3 erwähnte Blockierung des Senders während der Ausführung der Methode im Empfänger ist eine wichtige Eigenschaft für das Kommunikationsparadigma „entfernte Methodenaufrufe“. Der Sender und der Empfänger sind miteinander eng gekoppelt. Die Verwendung einer asynchronen Kommunikation mittels Nachrichten erlaubt die Entkopplung von Sender und Empfänger. Beschreiben wir kurz dieses Szenario:

- Der Sender erzeugt eine Nachricht und schickt diese an einen *Message-Broker*.
- Im Gegensatz zur synchronen Kommunikation ist der Sender nicht blockiert und kann sofort mit der weiteren Verarbeitung fortfahren.
- Nachdem der *Message-Broker* die Nachricht empfangen hat, benachrichtigt er den registrierten Empfänger darüber, dass eine neue Nachricht angekommen ist, und gibt diese an den Empfänger weiter.

- Der Empfänger nimmt die Nachricht entgegen und verarbeitet sie. Er hat keine Möglichkeit, den Sender direkt über das Ergebnis der Verarbeitung zu informieren.

Ermöglicht wird die asynchrone Kommunikation von Applikationen mittels des Nachrichtenaustausches durch *Message-oriented Middleware* (*MOM*). Eine Nachricht kann eine Anfrage, eine Meldung oder ein Ereignis sein, die Nutzdaten (Text, Binärdaten) enthält. Nachrichten bieten eine Abstraktionsebene, die es erlaubt, zwischen Details des Zielsystems und dem Nachrichteninhalt zu trennen. Die Nachrichtenvermittlung wird über *Destinations* realisiert. D.h. ein Sender schickt eine Nachricht gezielt an eine *Destination* und ein Empfänger kann sich bei einer *Destination* als Nachrichtenempfänger registrieren. Man unterscheidet zwei Arten von *Destinations*:

- *Queue*
- *Topic*

*Queue* und *Topic* repräsentieren die beiden Kommunikationsmodelle *Point-to-Point* und *Publish-and-Subscribe*. Die *Queue* wird benutzt, wenn eine *Point-to-Point*-Kommunikation stattfinden soll. Das *Topic* wird zur *Publish-and-Subscribe*-Kommunikation eingesetzt.

### Point-to-Point (PTP)

Kennzeichnend für das Point-to-Point Modell ist, dass eine von einem Sender verschickte Nachricht nur genau einem Empfänger zugestellt wird. Selbst wenn sich mehrere Empfänger für einen bestimmten Nachrichtentyp beim *Message-Broker* registriert haben, erhält lediglich ein einziger Empfänger die Nachricht. Der Message-Broker entscheidet, welchem Empfänger er die Nachricht zustellt.

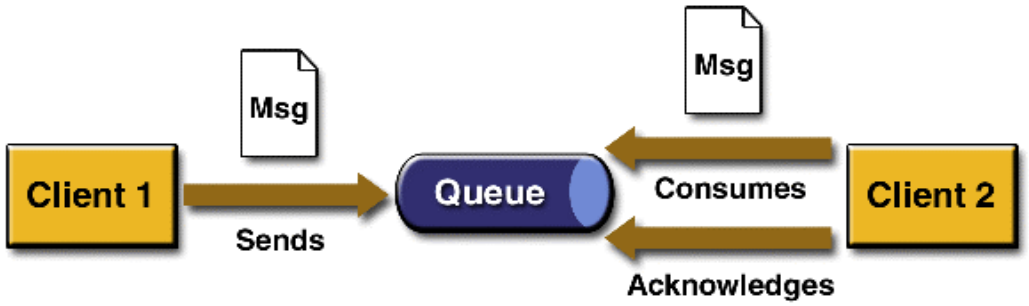



**Abbildung 7: Point-to-Point-Kommunikationsmodell**

### Publish-and-Subscribe

Im Gegensatz zum *Point-to-Point*-Kommunikation besteht die Haupteigenschaft von *Publish-and-Subscribe* darin, dass eine Nachricht nicht ausschließlich einem Empfänger zugestellt wird, sondern allen Empfängern, die sich für einen bestimmten Nachrichtentyp (ein sogenanntes *Topic*) registriert haben.

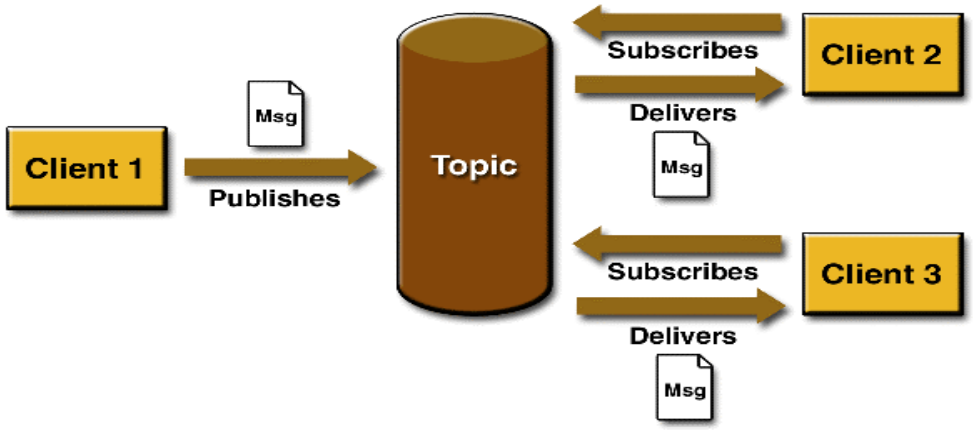



**Abbildung 8: Publish-and-Subscribe-Kommunikationsmodell**

### 6.2.4 Das Programmiermodell

Die Abbildung 4 illustriert das JMS-Programmiermodell. Die Anordnung der Interfaces entspricht dem Vorgehen eines Java-Klienten, um Nachrichten über einen *Message-Broker* zu senden oder von diesem zu empfangen. Auf der linken Seite sind die Interfaces dargestellt, die bei der Nutzung des *Point-to-Point*-Kommunikationsmodells verwendet werden. Die Interfaces auf der rechten Seite kommen beim Einsatz des *Publish-and-Subscribe*-Kommunikationsmodells zur Anwendung.

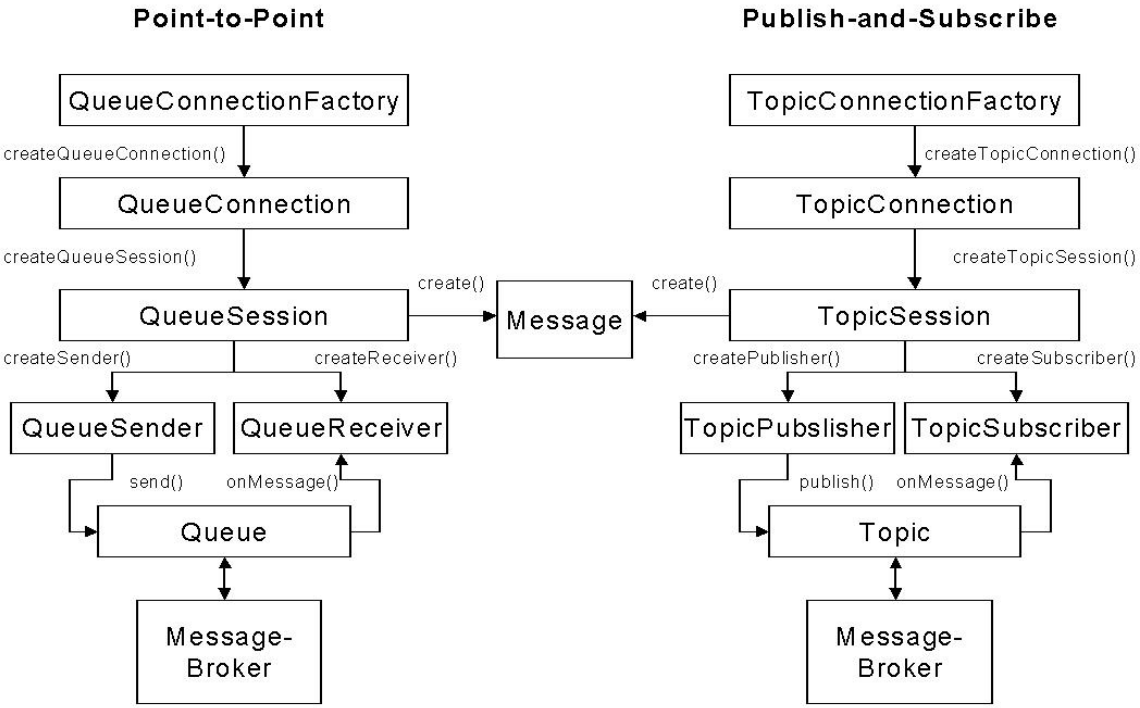



**Abbildung 9: Das JMS-Programmiermodell**

*QueueSender* und *TopicPublisher* werden generell als *Producer* bezeichnet – sie sind Ersteller von Nachrichten. *QueueReceiver* und *TopicSubscriber* werden hingegen als *Consumer* bezeichnet – sie sind Empfänger von Nachrichten. *Queue* und *Topic* sind sogenannte *Destinations*. Die verfügbaren Schnittstellen sind in der folgenden Tabelle beschrieben.

| Java-Interface | Beschreibung |
|---|---|
| QueueConnectionFactory, TopicConnectionFactory<br><br>Basic-Interface: ConnectionFactory | Werden über JNDI referenziert und dazu verwendet, Objekte vom Typ Connection zu erzeugen. Sie kapseln Parameter zur Konfiguration von Connections. |
| QueueConnection, TopicConnection<br><br>Basic-Interface: Connection | Repräsentieren eine offene Verbindung zu einem *Message-Broker* und werden verwendet, um Objekte vom Typ Session zu erzeugen. |
| QueueSession, TopicSession<br><br>Basic-Interface: Session | Repräsentieren den Kontext, in dem Nachrichten empfangen und versendet werden. Werden verwendet, um Objekte vom Typ MessageProducer, MessageConsumer und Message zu erzeugen. |
| Queue, Topic<br><br>Basic-Interface: Destination | Repräsentieren eine Nachrichtenschlange des Message-Brokers, verwalten eingehende Nachrichten und senden diese an registrierte MessageConsumer. |
| QueueSender, TopicPublisher<br><br>Basic-Interface: MessageProducer | Werden vom Client zum Senden von Nachrichten verwendet. |
| QueueReceiver, TopicSubscriber<br><br>Basic-Interface: MessageConsumer | Registrieren sich bei einer Destination und empfangen Nachrichten von dieser. |
| Message | Enthält die zu übermittelnden Informationen und wird von einer Session erzeugt. |

**Tabelle 2: JMS Interfaces**

### 6.2.5 Java Transaction Service (JTS)

JMS kann zusammen mit JTS verwendet werden, um verteilte Transaktionen, welche Nachrichten senden und empfangen, mit Datenbank-Updates und anderen JTS Diensten

kombiniert werden. Für die Vernetzung von Multimedia-Geräten in Fahrzeugen ist dieser Aspekt allerdings nicht relevant.

### 6.2.6 Was umfasst JMS nicht

JMS kümmert sich nicht um folgende Funktionalität [JMS01]:

- Load Balancing / Fault Tolerance: JMS spezifiziert nicht, wie Clients zusammenarbeiten müssen, um einen einheitlichen Dienst anbieten zu können.
- Error Notification: Nachrichten über Probleme oder Problemereignisse sind von JMS nicht standardisiert.
- Administration: kein Management API für die Administration von Messaging-Produkten.
- Security: JMS spezifiziert kein API für die Kontrolle der Geheimhaltung und Integrität von Nachrichten. Auch die Verteilungsmechanismen für digitale Signaturen oder Keys werden nicht spezifiziert.
- Wire Protocol: JMS spezifiziert kein Protokoll für die Nachrichtenübertragung.
- Message Type Repository: JMS enthält kein Repository für das Speichern von Nachrichtentyp-Definitionen und stellt keine Sprache zur Nachrichtentyp-Definition zur Verfügung.

### 6.2.7 Verfügbare Implementierungen

Folgende Firmen bieten JMS-API-Implementierungen an:

| Firma / Produktname | Weitere Informationen |
|---|---|
| **BEA Systems** Weblogic | http://www.beasys.com/ |
| **Hewlett-Packard** Message Service | http://www.hpmiddleware.com/products/hp-ms/default.htm |
| **IBM** MQSeries | http://www-3.ibm.com/software/ts/mqseries/api/mqjava.html |
| **Oracle** Oracle9iAS | http://www.oracle.com/ |
| **Sonic Software** SonicMQ | http://www.sonicsoftware.com/ |
| **Sun** ONE Message Queue | http://wwws.sun.com/software/products/message_queue/home_message_queue.html |

**Tabelle 3: JMS-Implementierungen**

## 6.3 Java Media Framework (JMF)

Das Java Media Framework - oder kurz: JMF - ist ein Plug-In-basiertes, plattform-unabhängiges Framework, welches dem Benutzer ermöglicht, in einfachster Weise, d.h. mit wenigen Befehlen Multimedia-Daten in seine Java-Programme einzubinden. Durch den Aufbau mit Plug-Ins ist es möglich, die unterschiedlichsten Arten von Audio- und Videodaten abzuspielen.

### 6.3.1 Begriffsdefinitionen

Um die weiteren Erläuterungen besser verstehen zu können, werden einige wichtigen Begriffe erläutert werden.

| Begriff | Definition |
|---|---|
| Codecs | Bezeichnet den Algorithmus für Daten-Kompression und -Dekompression. |
| Demultiplexer | Zertrennt einen Datenstrom in einzelne Teile, wie einen Video-Clip in Video- und Audiotrack. |
| Effektfilter | Dienen der Datenbearbeitung bzw. -veränderung. So kann beispielsweise unter einen bestehenden Audiotrack ein Echo gelegt werden. |
| Latenz | Die Verzögerung bis zum Beginn der Wiedergabe. |
| Media Stream | Hierunter sind die Daten zu verstehen, die aus einem Netzwerk oder einem Aufnahmegerät aber auch aus einer Datei gelesen werden können. Dabei können sie in zwei Gruppen unterschieden werden, die sich dahingehend unterscheiden auf welche Art und Weise die Daten empfangen werden. |
| Multimedia-Daten | Hierunter sind Daten zu verstehen, die sich mit zunehmender Zeit verändern. Audio- und Video-Clips zählen in erster Linie hierzu, aber auch Animationen. |
| Multiplexer | Ist die Umkehrung des Demultiplexers und fügt die einzelnen Tracks wieder zu einem Datenstrom zusammen. |
| Pull | Hierbei kontrolliert und initialisiert der Client die Übertragung. Als Beispiel könnte der vollständige Download einer Audiodatei mittels HTTP-Protokoll genannt werden. |
| Push | Der Server kontrolliert die Übertragung, z.B. über RTP (Realtime Transport Protokoll), das für Multimedia-Daten verwendet wird. |
| Renderer | Ist eine Abstraktion für ein Abspielgerät (z.B. Player, der Audiodaten auf der Soundkarte ausgibt) |

| Begriff | Definition |
|---|---|
| Streaming Media | Daten, die in Echtzeit auf einen Clienten abgespielt werden, ohne dass sie vorher komplett übertragen werden. Hierzu zählen z.B. das live-Radio oder live-Fernsehen. Die Daten sind ebenfalls wichtig, um Videokonferenzen zu entwickeln. Zur Übermittlung wird ein eigenes Protokoll benötigt. |

### 6.3.2 Überblick

JMF bietet Unterstützung für Multimedia-Anwendungen. Dabei ist es möglich, Multimedia-Daten in einem Applet oder einer Applikation zu verwenden. Die mittels JMF erstellten Anwendungen sollten dem Prinzip "write once, run anywhere" folgen.

Die Spezifikation ist aktuell in der Version 2.1.1 verfügbar. Die aktuelle Version des JMF vereinfacht die Verwendung von RTP, erhöht die Performance durch Benutzung von HotSpot in J2SE 1.3, stellt DirectAudioRenderer für die Latenz-Reduzierung zur Verfügung.

### 6.3.3 Architektur

Das Java Media Framework bietet zwei Arten von Architekturen an. Zum einen ist das die High-Level-Architektur, die mit einer privaten Multimediaanlage verglichen werden kann. Dabei ist z.B. die Videokamera das *CaptureDevice*, welches es ermöglicht Daten zu erzeugen, indem diese von einer realen Quelle "holt". Die Videokassette ist die *DataSource*, von der Daten gelesen und auf der Daten auch wieder geschrieben werden können. Der Videorekorder entspricht dabei dem Player, der die Daten abspielt, aufnimmt oder bearbeitet. Der Fernseher und die Videoanlage stellen die Ausgabegeräte dar. Auf der anderen Seite existiert auch eine Lower-Level API, die es dem Programmierer ermöglicht, eigene Komponenten hinzuzufügen.
Weitere Architekturdaten werden jetzt vorgestellt und erläutert:

- *TimeModel*:
  Die Exaktheit des JMF liegt bei 1ns. Klassen, welche das Zeitmodell unterstützen müssen das Clock-Interface implementieren, das das Timing und die Synchronisation der Daten übernimmt.

- *DataSink*:
  Wird zum Lesen von Mediendaten aus einer Datenquelle und Schreiben in einem Ziel benutzt. Ähnlich dem Player wird sie über einen Manager und einer DataSource erzeugt.

- *Event Modell*:
  Bei JMF werden die Events als MediaEvents bezechnet. Für jedes Objekt, welches ein MediaEvent auslösen kann existiert ein dazugehöriges Listener-Interface.

- *DataModel*:
  Ein Player benutzt normalerweise eine DataSource um Daten zu transferieren. Sie enthält die URL der Daten, sowie das Protokoll und die Software, die zur Übertragung verwendet werden. Je nach Datenart werden zwei Kategorien unterschieden:

- *Pull DataSource*

- *Push DataSource*

- *Controls*:
  Es existieren diverse Control-Interfaces zur Kontrolle über bestimmte Attribute von Objekten.

- *Controller*:
  Die Darstellung von Multimediadaten wird von einem Controller übernommen. Dieses Interface beinhaltet die Definitionen der grundlegenden Zustände und *control*-Mechanismen.

- *Player*:
  Ein Player kann solche Multimediadaten verarbeiten und wiedergeben, die dabei von einer DataSource geliefert werden.

- *Processor*:
  Dies ist eine spezialisierte Art von *Player*. Hier können die Daten noch durch Codecs und Effekt-Filtern nachgebessert werden.

- *Erweiterbarkeit*:
  Das JMF kann im wesentlichen durch zwei Mechanismen erweitert werden. Als erstes durch das Einspielen von Plug-Ins, die dann in die JMF-Registry eingetragen werden müssen. Als zweites werden sämtliche Interfaces direkt implementiert, was eine höhere Flexibilität zur Folge hat, aber auch einen erhöhten Aufwand.

- *Manager*:
  Es existieren vier verschiedene Manager:

  - *Manager*:
    stellt einen Zugriffspunkt für systemabhängige Ressourcen dar, wie *Player*, *DataSinks*, *Processors*, *DataSources*.

  - *PackageManager*:
    hiermit kann auf die Registrierungsdatenbank des JMF zugegriffen werden, um Pakete zu verwalten

  - *CaptureDeviceManager*:
    hier sind alle *CaptureDevices* in einer Datenbank erfasst

  - *PlugInManager*:
    verwaltet die verfügbaren Plug-Ins

### 6.3.4 Real-Time Protocol (RTP)

Das *Real-Time-Protokoll* stellt einen End-to-End-Netzwerkdienst zur Echtzeit-Übertragung von Daten zur Verfügung. *RTP* ist unabhängig von Netzwerk und Transportprotokoll. Es kann sowohl in Unicast- als auch in Multicast-Netzwerken eingesetzt werden. In Unicast-Netzwerken

werden die Datenpakete für jedes Ziel je einmal verschickt. In einem Multicast-Netzwerk liegt es in der Verantwortung des Netzwerkes, dass ein einmalig verschicktes Paket seine Ziele erreicht.

*RTP* ermöglicht es, den Typ der übertragenen Daten zu bestimmen, die Pakete in die richtige Reihenfolge zu bringen und Media-Streams aus verschiedenen Quellen zu synchronisieren. Für *RTP*-Datenpakete kann aber nicht garantiert werden, dass sie beim Empfänger überhaupt ankommen. Es liegt in der Verantwortung des Empfängers, die Reihenfolge zu rekonstruieren und fehlende Pakete zu bemerken.

**RTP-Sitzungen**

Eine *RTP*-Sitzung ist eine Menge von Anwendungen, die über *RTP* miteinander kommunizieren. Sie wird durch eine Netzwerkadresse und Ports identifiziert. Die verschiedenen Medientypen werden in getrennten Sitzungen übertragen. So kann bei einer Videokonferenz bei Mangel an Bandbreite nur der Audioteil empfangen werden.

Diese Sitzungen werden von einem *SessionManager* koordiniert. Der *SessionManager* ist die lokale Repräsentation der *RTP*-Sitzung. Der *RTP* Kontrollkanal wird ebenfalls vom *SessionManager* überwacht. Das Interface definiert Methoden, die RTP-Sitzungen initialisieren und durchführen.

**RTPEvents & Listener**

*RTP* nutzt die sogenannten *RTPEvents* zur Benachrichtigung über Zustände von Sitzungen und Datenströmen. Jedes *RTPEvent* ist auch ein *MediaEvent*. *RTPEvents* lassen sich in vier Teilbereiche einteilen. Jeder verfügt über entsprechende *Listener*.

- *SessionListener:* Empfängt Meldungen über Änderungen der Zustände von Sitzungen.
- *SendStreamListener:* Empfängt Meldungen über Änderungen der Zustände des übertragenen Datenstroms.
- *ReceiveStreamListener:* Empfängt Meldungen über Änderungen der Zustände des empfangenen Datenstroms.
- *RemoteListener:* Empfängt Meldungen über Ereignisse von anderen Sitzungsteilnehmern.

**RTP Daten**

Die Datenströme innerhalb einer Sitzung liegen in Form von *RTPStream*-Objekten vor. Dabei werden zwei Arten unterschieden*: ReceiveStream und SendStream.* Jeder Strom hat seine eigene gepufferte Datenquelle. Für *ReceiveStreams* ist dies immer *PushBufferDataSource*.

**Senden und Empfangen**

Um einen einzelnen *RTP*-Strom zu empfangen, reicht ein *Player* aus. Dieser benötigt nur einen *MediaLocator*, da die Koordination mehrerer Ströme entfällt.

Sobald mehr als ein Strom empfangen werden soll, wird ein *SessionManager* zur Verwaltung benötigt. Dieser schickt eine Benachrichtigung, sobald ein neuer Strom der Sitzung hinzugefügt wird und erzeugt pro Strom einen entsprechenden *Player*.

Sollen Daten über ein Netzwerk gesendet werden, wird ein Prozessor benötigt, der die Daten aufbereitet, in dem er sie zum Beispiel komprimiert. Die daraus entstehende Datenquelle wird an

den Aufruf von *createSendStream* übergeben. Die Datenübertragung wird durch die Methoden des *SendStream*-Objektes gesteuert.

### 6.3.5 JMF Pakete

Zur Zeit stehen 11 Java-Pakete zur Verfügung:

- *javax.media*
  Enthält vordefinierte Schnittstellen (z. B. *Player* als MediaHandler, um zeitbasierte Multimediadaten darzustellen und zu kontrollieren), Klassen (z. B. *Manager* die Zugriffsstelle, um systemabhängige Resourcen wie *Player*, Datenquellen u.a. zu erhalten), Ausnahmen (z. B. *NoPlayerException*, wenn der *Manager* einen *Player* für eine bestimmte URL nicht finden kann) und Fehler.
- *javax.media.bean.playerbean*
  Die JavaBean Komponente, besteht aus Klassen (z.B. *MediaPlayer* und *MediaPlyerBeanInfo*).
- *javax.media.control*
  Schnittstellen zur Steuerung (z. B. *MpegAudioControl* um die Parameter für MPEG Audios anzugeben).
- *javax.media.datasink*
  *DataSink* ist eine Schnittstelle aus *javax.media* für Objekte, die Mediendaten von einer Quelle lesen und für ein Ziel wiedergeben (z. B. Abspeichern in einer Datei). Dieses Package stellt eine Schnittstelle und drei Klassen für *DataSink* zur Verfügung.
- *javax.media.format*
  Ermöglicht Formatangabe und -umwandlung der Medien, enthält Klassen (z.B. *AudioFormat* enthält Formatinformationen für Audiodateien) und eine Ausnahme (*UnsupportedFormatException*, wenn ein *FormatChange* fehlschlägt, weil das Zielformat nicht unterstützt wird).
- *javax.media.protocol*
  Schnittstellen und Klassen, die Angaben für das Lesen und Schreiben von Mediendaten erlauben.
- *javax.media.renderer*
  Schnittstellen für *Renderer* und *Player*.
- *javax.media.rtp*
  Bietet Unterstützung für *RTP* (Real-Time Transport Protocol). RTP ermöglicht das Versenden und Empfangen von Echtzeit-Mediendaten über das Netzwerk.
- *javax.media.rtp.event*
  RTP-Unterstützung
- *javax.media.rtp.rtcp*
  RTP-Unterstützung
- *javax.media.util*
  Utilities für das Konvertieren von Video Buffern in AWT Bildobjekte und umgekehrt.

In der aktuellen Version unterstützt JMF folgende Videoformate: Cinepak, MPEG-1, H.261, H.263, JPEG, Indeo. Unterstützte Audioformate sind: PCM, Mu-Law, ADPCM (DVI, IMA4), MPEG-1, MPEG Layer 3, GSM, G.723.

### 6.3.6 Verfügbare Implementierungen

Folgende Firmen bieten JMF-Implementierungen an:

| Firma / Produktname | Weitere Informationen |
|---|---|
| **Sun** Java Media Framework | http://java.sun.com/products/java-media/jmf/ |

# 7 Home Audio Video interoperability (HAVi)

HAVi wurde 1999 als Non-Profit Organisation von führenden Herstellern der Unterhaltungselektronik gegründet. Das Ziel von HAVi besteht in der Etablierung eines Standards zur Heimgerätevernetzung, insbesondere von Geräten der Unterhaltungselektronik.

## 7.1 Begriffsdefinitionen

Die folgende Tabelle enthält die Erläuterungen der wichtigsten Begriffe von HAVi.

| Begriff | Definition |
|---|---|
| BAV | **Base AV device**. Gerät mit minimaler HAVi-Funktionalität. Enthält lediglich ein DCM (als Java-Bytecode), welches in ein Full AV Device (FAV) übertragen und dort ausgeführt werden kann. Base AV Devices können schon direkt an ein HAVi-Netzwerk angeschlossen werden und werden dort als solche automatisch erkannt. |
| DCM | **Device Control Module**. Software-Modul, das in einem IAV oder FAV ausgeführt werden kann und dessen API der HAVi-Spezifikation entspricht. Ein einzelnes physisches Gerät kann mehrere DCMs als logische Geräte bereitstellen. |
| FAV | **Full AV device**. Gerät mit vollständiger HAVi-Funktionalität. Stellt eine Ausführungsumgebung (JRE) für Java-Codemodule anderer Geräte zur Verfügung. |
| FCM | **Functional Component Module**. Software-Modul, das innerhalb eines DCMs existiert. Dient zur feineren Unterteilung eines Gerätes in verschiedene Software-Module. |
| IAV | **Intermediate AV device**. HAVi-Gerät, das ebenfalls Codemodule anderer Geräte beherbergen kann, allerdings besitzt ein IAV im Gegensatz zu FAVs unter anderem keine Java-Ausführungsumgebung (JRE). |
| LAV | **Legacy AV device**. Geräte, die keinerlei Unterstützung für HAVi mitbringen und daher über Gateways, die sowohl die HAVi-Protokolle als auch die proprietären Protokolle eines LAVs verstehen, mit einem HAVi-Netzwerk verbunden werden. |
| SDD | **Self Describing Device**. Informationen, die ein HAVi-Gerät beschreiben und in einem standardisierten Format (IEEE 1212 Command and Status Register Standard) abgelegt sein müssen |
| SEID | **Software Element Identifier.** Eindeutige ID für ein einzelnes Software-Modul (z.B. ein DCM). Software-Module werden immer durch ihre SEIDs identifiziert und adressiert. Eine SEID muss nur solange konstant bleiben, wie ein Software-Modul in einem Netzwerk aktiviert ist. |

## 7.2 Überblick

Home Audio Video interoperability (HAVi) ist ein Standard zur Vernetzung von Audio- und Video- Geräten. HAVi zielt vorrangig auf Geräte aus der Unterhaltungselektronik, da sie besondere Anforderungen bezüglich der benötigten Übertragungsbandbreite und der Echtzeitfähigkeit besitzen (Streaming von Audio und Video in guter Qualität), bietet jedoch auch die Möglichkeit zur Vernetzung anderer Geräte.

HAVi bietet analog den UPnP- oder Jini-Architekturen ebenfalls automatische Konfiguration eines Netzwerks. Wenn neue Geräte zu einem HAVi-Netzwerk hinzukommen oder verschwinden, werden die anderen Geräte darüber benachrichtigt und alle Geräte automatisch so konfiguriert, dass die vorhandenen Geräte miteinander interagieren können, ohne dass es manueller Administration bedarf.

Einen großen Anteil an diesen Fähigkeiten von HAVi besitzt die IEEE1394-Netzwerkschnittstelle, die sogenanntes Hot Plugging (Geräte können bei laufendem Betrieb zu einem Netzwerk hinzugefügt oder von einem Netzwerk entfernt werden, das Netz konfiguriert sich dann entsprechend neu) und isochrone Datenübertragung bereitstellt. HAVi-Geräte müssen die IEEE1394-Netzwerkschnittstelle anbieten, es ist ihnen allerdings freigestellt, auch noch weitere Netzwerkschnittstellen zu implementieren.

HAVi-Netzwerke müssen daher nicht zwangläufig isoliert sein, sondern können auch mit Nicht-HAVi-Geräten oder dem Internet verbunden sein.

Die HAVi-Spezifikation ist plattformunabhängig, es werden keine Aussagen über die zugrundeliegende Hardware-Architektur oder Betriebssysteme gemacht (lediglich echtzeitfähig sollte ein Betriebssystem in einem HAVi-Gerät sein). Als Programmiersprache wird Java wegen seinen Fähigkeiten (Übertragung und entfernte Ausführung von Java-Bytecode) und seiner weiten Verbreitung vorgeschlagen. Es besteht jedoch auch die Möglichkeit, HAVi-Geräte zu entwickeln, deren HAVi-Stack in anderen Programmiersprachen (Native Code) implementiert ist. Ein HAVi-Gerät kann gleichzeitig sowohl Dienstnutzer (Client, in der HAVi-Terminologie *Controller* genannt) als auch Dienstanbieter (in der HAVi-Terminologie C*ontrolled Device* genannt) sein und es gibt keine zentrale Kontrollinstanz (Master) im Netzwerk. Somit ist ein HAVi-Netzwerk ein Peer-to-Peer-Netzwerk (einzige Einschränkung: Hierarchie zwischen FAVs/IAVs und BAVs/LAVs und sämtliche Dienste wie Messaging oder Eventing laufen verteilt ab.

Die HAVi-Geräte können unabhängig von Hersteller und Netzwerkkonfiguration interagieren, da eine umfangreiche Schnittstelle auf höherer Abstraktionsebene (Interoperability API) angeboten wird. Realisiert wird dieses API durch eine Reihe von standardisierten Diensten, die den Zugriff auf die Netzwerk-Schnittstelle kapseln (1394 Communication Media Manager), die verfügbaren Ressourcen verwalten (Resource Manager) , Informationen über in einem Netzwerk vorhandene Geräte und Dienste bereitstellen (Registry) oder Kommunikationsmöglichkeiten anbieten (Event Manager, Messaging System). Neben den allen Geräten gemeinsamen Diensten, kann jedes physische Gerät auch noch zusätzlich spezifische Geräteabstraktionen mit einem definierten API (DCMs) anbieten, mit dem spezifische Fähigkeiten der Geräte benutzt werden können.

Um bestehende Geräte (Legacy Devices, LAVs), die keinerlei Kenntnis der Möglichkeiten von HAVi besitzen, in ein HAVi-Netzwerk integrieren zu können beschreibt die HAVi-Spezifikation die Möglichkeit, solche Geräte über einen Controller mit dem restlichen Netzwerk verbinden zu

können. Die Kommunikation zwischen einem LAV und seinem Controller läuft über proprietäre Protokolle ab.

## 7.3 Geräte-Architektur und Dienstbeschreibungen

Der Aufbau eines HAVi-Geräts besteht grundsätzlich immer aus 3 Schichten (vgl. Abbildung „Architektur eines HAVi-Geräts“). Die unterste Schicht bilden dabei herstellerspezifische Funktionen (plattformspezifische API). Die mittlere Schicht besteht aus HAVi-Softwaremodulen, den sogenannten Software-Elementen, die entweder von HAVi in ihrer Funktionalität genau spezifiziert werden (System-Softwareelemente) oder von denen lediglich ihr Aufbau festgelegt ist (DCMs und FCMs).

Die Softwaremodule der mittleren Schicht sind – von Einschränkungen aus Sicherheitsgründen einmal abgesehen - auch von anderen HAVi-Geräten aus zugreifbar und bilden eine Art Geräteschnittstelle nach außen. HAVi hat aus diesem Grund die Funktion einer Middleware für verteilte Systeme, vergleichbar CORBA oder Jini und stellt die Infrastruktur für eine sinnvolle Vernetzung verschiedener Geräte dar (Interoperabilitätsschicht).

Die oberste Schicht bilden dann sogenannte Applikationen und GUI-Elemente, mit denen der Benutzer direkt interagieren kann und die auf die Funktionalität der Interoperabilitätsschicht aufsetzen.

Um die Kompatibilität verschiedener HAVi-Geräte festzustellen und zu gewährleisten, benutzt HAVi einen Versionierungsmechanismus der Form major.minor. In der aktuellen Version der HAVi-Spezifikation (zu dem Zeitpunkt, als dieses Dokument verfasst wurde, war das Version 1.1) wird die Aussage gemacht, dass das bestehende HAVi-API in einer neueren HAVi-Version nicht geändert oder gekürzt, sondern lediglich um neue Teile erweitert werden wird. Abwärtskompatibilität wird für System-SE, DCMs und FCMs gefordert, für Applikationen hingegen nicht.

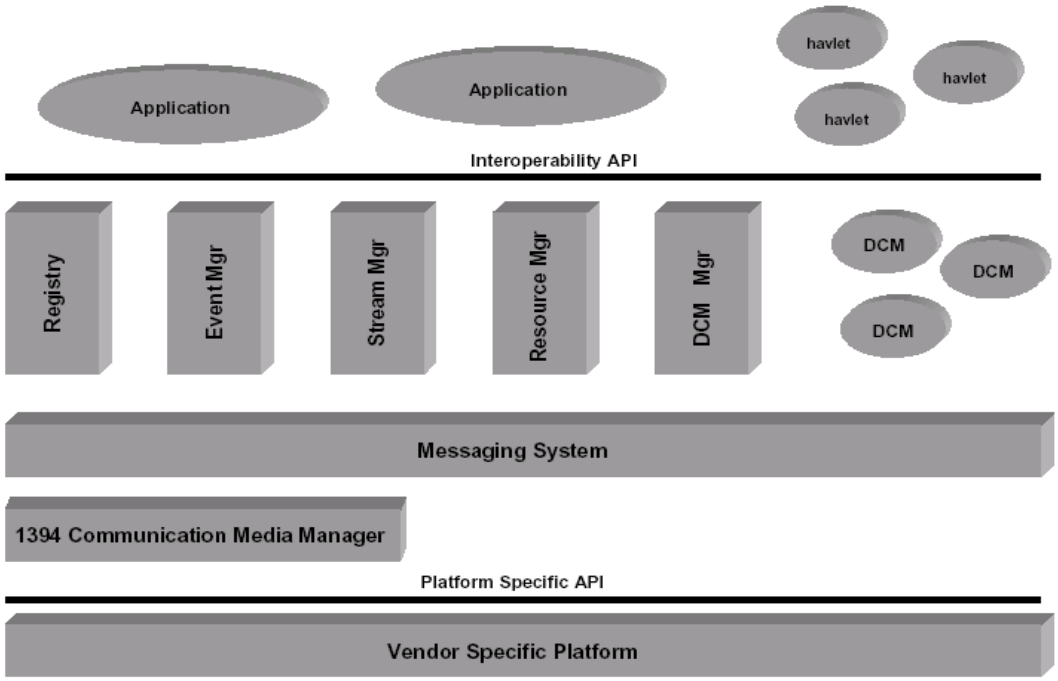



**Abbildung 10: Architektur eines HAVi-Gerätes**

Der obigen Abbildung ist nicht zu entnehmen, dass jedes Gerät auch noch eine Gerätebeschreibung besitzt (SDD), auf die von außen lesend zugegriffen werden kann und die gemäß dem IEEE1212–Standard (IEEE1212 Configuration ROM) aufgebaut ist. Eine SDD enthält dabei mindestens die folgenden Einträge:

| Datum | HAVi-spezifisch | Bedeutung |
|---|---|---|
| GUID | nein | Eindeutiger Bezeichner des Geräts, setzt sich zusammen aus Hersteller-Angabe (vendor_ID) und Chip-ID |
| Version | nein | Versionsnummer |
| HAVi_Device_Profile | ja | Enthält Angaben zur Geräteklasse (BAV, IAV oder FAV), über die Existenz eines DCM Managers, eines Stream Managers und eines Resource Managers sowie über Display-Fähigkeiten und den Gerätestatus (aktiv oder inaktiv) |

| Datum | HAVi-spezifisch | Bedeutung |
|---|---|---|
| Descriptor for HAVi_Device_Profile | ja | Name des Gerätes, der dem Benutzer angezeigt wird |
| HAVi_DCM | ja | Zeiger auf die DCM Code Unit eines BAV-Gerätes (Java Bytecode) |
| HAVi_DCM_Reference | ja | URL für entfernten Zugriff auf DCM Code Unit eines BAV- Gerätes |
| HAVi_DCM_Profile | ja | Angaben über Länge der DCM Code Unit in Bytes, deren Speicherbedarf bei Installation und Ausführung sowie der unterstützen Version des HAVi Messaging Systems |
| HAVi_Device_Icon_Bitmap | ja | Zeiger auf Bitmap, die Gerät als Icon repräsentieren soll |
| HAVi_Message_Version | ja | Version des unterstützen HAVi Messaging Systems |

**Tabelle 4: Einträge in der SDD eines HAVi-Gerätes**

Wie bereits erwähnt spezifiziert HAVi vier verschiedene Geräteklassen, die sich anhand ihrer Funktionalität und damit auch ihrer Komplexität unterscheiden:

**FAV (Full AV Device)**

Geräte dieser Klasse enthalten einen vollständigen HAVi-Stack inklusive einer Java Virtual Machine zur Ausführung von Java-Bytecode.

**IAV (Intermediate AV Device)**

Geräte dieser Klasse enthalten ebenfalls einen weitgehend kompletten HAVi-Stack, ihnen fehlt aber auf jeden Fall eine Java Virtual Machine als Ausführungsumgebung für Java Bytecode.

**BAV (Base AV Device)**

BAVs sind Geräte mit IEEE1394-Anschluss und einer in Java geschriebenen DCM Code Unit, die gemäß dem IEEE1212-Standard im Configuration ROM abgelegt ist. BAVs enthalten keinen HAVi-Stack und benötigen noch einen sogenannten Controller, der eine Ausführungsumgebung für die DCM Code Unit des BAV bietet, damit sie angesteuert werden können.

**LAV (Legacy AV Device)**

Diese Geräte entsprechen den gegenwärtigen Geräten der Unterhaltungselektronik und sind streng genommen keine HAVi-Geräte. Damit sie in einem HAVi-Netzwerk eingesetzt werden können, benötigen sie einen Controller (IAV oder FAV), mit dem sie über proprietäre Protokolle kommunizieren.

Die nachfolgende Tabelle soll zeigen, welche Bestandteile des HAVi-Stacks (Software-Elemente) die verschiedenen Geräteklassen implementieren:

| Software-Element | FAV | IAV | BAV | LAV |
|---|---|---|---|---|
| Java Virtual Machine | ✓ | | | |
| Applikation | (✓) | (✓) | | |
| DDI Controller - Benutzungsoberfläche | (✓) | (✓) | | |
| Resource Manager **(RM)** | ✓ | (✓) | | |
| Stream Manager **(SM)** | ✓ | (✓) | | |
| DCM Manager **(DCMM)** | ✓ | (✓) | | |
| Registry **(REG)** | ✓ | ✓ | | |
| Event Manager **(EM)** | ✓ | ✓ | | |
| Messaging System **(MS)** | ✓ | ✓ | | |
| 1394 Communication Media Manager **(CMM1394)** | ✓ | ✓ | | |
| SDD | ✓ | ✓ | ✓ | |
| DCM | ✓ | (✓) | ✓ | ✓ (Native Code) |

**Tabelle 5: Konfiguration der HAVi-Geräteklassen**

Wie man der vorhergehenden Tabelle entnehmen kann, werden verschiedene Arten von Software-Elementen (SE) unterschieden:

**System-SE** mit definierter API (CMM1394, MS, REG, EM, SM, RM, DCMM)

**DCMs und FCMs**

DCMs abstrahieren Geräte und stellen die Schnittstelle zu den Geräten in der HAVi-Architektur dar. FCMs entsprechen einer einzelnen Funktionalität eines Gerätes (d.h. sie entsprechen dem Dienstbegriff in anderen Architekturen zur Gerätevernetzung) und sind immer mit einer DCM verbunden, in anderen Worten: ein FCM ist einem DCM zugeordnet.

Da der Schwerpunkt von HAVi auf der Vernetzung von Geräten aus der Unterhaltungselektronik liegt, gibt es schon standardisierte FCMs für folgende Gerätearten: *Tuner, VCR, Clock, Camera, AV disc, Amplifier, Display, AV display, Modem,* und *Web proxy*.

Implementiert sind DCMs und die zugehörigen FCMs in einer DCM Code Unit. DCMs und FCMS sind HAVi-Objekte (Software-Elemente) und sind dementsprechend jeweils durch eine SEID gekennzeichnet. Eine SEID besteht aus einer GUID (Globally Unique ID), die das Gerät bezeichnet sowie einem swHandle, das ein SE innerhalb eines Gerätes eindeutig bezeichnet. Da die swHandles nicht persistent sind, sondern bei jeder Anmeldung eines Gerätes für seine SE neu vergeben werden, wird noch ein anderer Mechanismus benötigt, der es ermöglicht, dass ein bestimmtes SE immer Zugriff auf dasselbe SE haben kann, solange es im Netzwerk verfügbar ist. Diesen Mechanismus stellen sogenannte HUIDs (HAVi Unique Identifiers) bereit, die Software-Elemente ebenfalls identifizieren und über Netzwerk-Resets hinweg persistent sind. Für LAVs gibt es keine HUIDs.

**Applikationen und UI-Elemente:**

Applikationen benutzen das Interoperabilitäts-API von HAVi und bauen somit auf der von System-SE und DCMs angebotenen Funktionalität auf. Applikationen stellen in vielen Fällen auch eine grafische Benutzungsoberfläche (GUI) bereit. Eine GUI kann in HAVi auf zwei Arten realisiert werden:

- Durch Schicken von DDI (Data Driven Interaction)-Nachrichten an einen auf einem FAV oder IAV befindlichen DDI-Controller. Der DDI-Controller verarbeitet dann die Ein- und Ausgaben in Bezug auf die GUI.
- Durch sogenannte Havlets. Havlets sind Java-Bytecode-Module, die eine GUI vollständig, d.h. inklusive der Ein- und Ausgabeverarbeitung, enthalten. Havlets basieren auf einer Untermenge von Java AWT (Abstract Window Toolkit).

Nachdem die Grundkonzepte des Geräteaufbaus beschreiben wurden, sollen nun die verschiedenen System-SE noch kurz vorgestellt werden:

**Messaging System (MS)**:

Das Messaging System ist verantwortlich für den Nachrichtenaustausch zwischen zwei Software-Elementen. Um das Messaging System benutzen zu können, müssen sich SE bei ihrer Installation jeweils bei ihrem lokalen Messaging System registrieren und dabei Callback-Funktionen zur Benachrichtigung durch das MS angeben. Wenn sich ein SE bei seinem lokalen MS registriert, bekommt es dabei auch eine SEID vom MS zugewiesen und ist unter dieser SEID dann im Netzwerk adressierbar. Das MS stellt auch eine Art Namensdienst für lokale System-SE bereit:

Durch Aufrufe von `Msg::MsgGetSystemSEID` können unter Angabe des Typs (z.B. Event Manager, Registry) die SEIDs von lokalen System-SE erhalten werden.

Eine weitere Funktionalität, die das Messaging System anbietet, besteht in der Überwachung einzelner SE im Netz auf Verfügbarkeit durch ein sogenanntes *MsgWatchOn*. Meldet sich dann ein überwachtes SE vom Netzwerk ab, so bekommt das Messaging System eines jeden Geräts ein Ereignis zugestellt. Dieses Ereignis wird dann an das bzw. die SE weitergegeben, die das entsprechende SE überwachen lassen wollten.

Als Hauptfunktionalität stellt das Messaging System über sein API Funktionen zum Versenden von Nachrichten mit und ohne Empfangsbestätigung bereit und bietet basierend auf dieser Funktionalität Abstraktionen für synchrone, blockierende Funktionsaufrufe und asynchrone, nicht–blockierende Funktionsaufrufe.

Wird eine Funktion eines SE von einem anderen SE aufgerufen, so werden vor der Übertragung der eigentlichen Nachricht die Funktionsparameter ins CDR-Format (vgl. CORBA) serialisiert. Für die Behandlung von Fehlern (z.B. Fehlererkennung durch CRC, zuverlässige Verbindungen) ist nicht das MS verantwortlich, sondern darunterliegende Schichten, im Falle von HAVi der IEEE1394-Bus.

**Communication Media Manager (CMM1394)**:

Der Communication Media Manager stellt die Schnittstelle zum Übertragungsmedium IEEE1394-Bus dar. Der CMM1394 bietet eine Abstraktion des IEEE1394-Busses bezüglich der Netzwerkverwaltung, indem er Ereignisnachrichten über den Event Manager, z.B. im Falle eines Bus-Resets, an die lokalen SE verschickt. Dadurch können sämtliche HAVi-Geräte auf Topologieänderungen im Netzwerk (Anschließen bzw. Ausstecken von Geräten) reagieren. Daneben stellt der CMM1394 die Grundfunktionalität zur Kommunikation mit anderen Geräten bereit. Die Grundfunktionalität wird durch folgende Funktionen realisiert:

- **Read**: Initiiert eine IEEE1394-Lesetransaktion mit einem entfernten Gerät.
- **Write**: Initiiert eine IEEE1394-Schreibtransaktion mit einem entfernten Gerät.
- **Lock**: Initiiert eine IEEE1394-Sperrtransaktion auf einem entfernten Gerät.
- **EnrollIndication**: Registriert eine Art Event-Handler auf einem lokalen Speicherbereich, der benachrichtigt wird, sobald auf diesen Speicherbereich zugegriffen wird.
- **DropIndication**: Entfernt den mit EnrollIndication registrierten Event-Handler wieder.

**Registry (REG):**

Die Registry realisiert ein Verzeichnis lokaler SE (DCMs, FCMS, Applikationen). Diese SE registrieren sich bei der Registry jeweils mit ihrer SEID und Attributen zur näheren Beschreibung ihrer Eigenschaften.

In einem HAVi-Netzwerk werden die Registries der einzelnen Geräte grundsätzlich zu einem netzweiten Verzeichnisdienst föderiert. Bei einer Suchanfrage durch ein SE wird immer zuerst eine lokale Suche durchgeführt, anschließend wird die Suchanfrage an alle anderen im Netzwerk vorhandenen Registries weitergeleitet. Das Ergebnis einer Suchanfrage ist eine SEID-Liste der gefundenen passenden Software-Elemente.

**Event Manager (EM)**:

Der Event Manager implementiert einen Ereignisdienst, der bei Zustandsänderung eines SE Ereignisnachrichten an sämtliche registrierte Interessenten verschickt. Der Ereignisdienst ist in HAVi ebenfalls als eine Föderation realisiert. Daher wird in HAVi auch zwischen lokalen (nur die lokalen Interessenten werden benachrichtigt) und globalen Ereignissen (zusätzlich zur Benachrichtigung lokaler Interessenten wird ein globales Ereignis auch an alle anderen Event Manager verschickt) unterschieden.

**Stream Manager (SM)**:

Der Stream Manager bietet Funktionen für Verbindungen zwischen zwei FCMs mit isochronem Datentransfer, d.h. die Datenübertragung erfolgt mit garantiertem Timing und mit garantierter Bandbreite (Echtzeit-Streams).

Dabei wird von Seiten des SM zwischen drei Verbindungstypen unterschieden:

- Geräteinterne Verbindung zwischen zwei lokalen FCMs.
- Direkte Kabelverbindungen zwischen 2 Geräten (nicht über den IEEE1394-Bus).
- Verbindungen über das IEEE1394-Netzwerk (bei diesen Verbindungen wird weiter unterschieden zwischen Punkt-zu-Punkt- oder Broadcast-Verbindungen)
-

Der SM hat dabei die folgenden Aufgaben:

- Verwaltung von Verbindungsinformationen
- Test der Kompatibilität zwischen den zu verbindenden FCMs. Dabei ist der SM nur für den Test anhand der von den FCMs bereitgestellten Informationen zuständig, für evtl. notwendige Konvertierungen sind die FCMs selbst verantwortlich.
- Reservierung und Freigabe von Transportressourcen (Bandbreite, IEEE1394-Kanäle)
- Wiederherstellung von Verbindungen nach Netzwerk-Resets für lokale Anwendungen, d.h. ein SM kann von ihm erstellte Verbindungen wiederherstellen.
- Geräteinterne Konfiguration der Verbindungen

**Resource Manager (RM)**:

Der Resource Manager ist im Gegensatz zum SM nicht für die Verwaltung von Transport- sondern von Geräteressourcen verantwortlich, sofern schreibend auf sie zugegriffen wird. Er bietet dafür Funktionen zum Reservieren und Freigeben von Geräteressourcen (FCMs) und Strategien zur Lösung von Zugriffskonflikten. Durch Reservierungen soll sichergestellt werden, dass SE, die ein FCM reserviert haben, nicht durch andere SE gestört werden. Möchten nun zwei verschiedene SE ein FCM reservieren, definiert HAVi ein Verhandlungsprotokoll, mit dem Klienten durch ihre lokalen Resource Manager über eine solche Reservierung verhandeln können. Damit Klienten diese Funktionalität benutzen können, müssen sie sich bei ihrem lokalen Resource Manager mit Angabe einer Callback-Funktion für Benachrichtigungen anmelden. Bei den Klienten wird ferner zwischen Systemklienten und Benutzern unterschieden, wobei Benutzer die Möglichkeit haben, die Reservierung eines FCM von einem Systemklienten zu erzwingen. Haben Klienten ein FCM reserviert und benötigen es nicht mehr, so sind sie selbst dafür verantwortlich, die Reservierung wieder freizugeben.

Geräteressourcen können zum gegenwärtigen Zeitpunkt oder für geplante Aktionen in der Zukunft (z.B. ein Videorecorder reserviert einen Tuner für mehrere Stunden am nächsten Tag um die Aufnahme einer Sendung sicherzustellen) reserviert werden.

**DCM Manager (DCMM)**:

Der DCMM ist verantwortlich für die (De-)Installation von DCM Code Units in FAVs. Durch den Einsatz von DCMMs soll sichergestellt werden, dass nur maximal eine Instanz eines DCM gleichzeitig installiert ist.

Um diese Funktionalität zu realisieren, reagieren DCMMs auf Netzwerk-Events vom CMM1394 und machen dann in einem definierten Protokoll aus, welcher DCMM die jeweilige neue DCM Code Unit installiert. Das von HAVi definierte Protokoll soll dabei die gleichmäßige Auslastung

aller Controller im Netzwerk bezüglich der Anzahl installierter DCM Code Units sicherstellen. DCM Code Units können den Installationsprozess allerdings durch die optionale Angabe von Präferenzen beeinflussen (z.B. Installation in einem Controller des Herstellers xy erwünscht).

## 7.4 Kommunikationsmechanismen

### 7.4.1 Die Rolle von IEEE1394

Da ein vollständiger IP-Stack unter Umständen für einige Geräte aus der Unterhaltungselektronik zu komplex und nicht problemangemessen ist, haben sich die Autoren der HAVi-Spezifikation dafür entschieden, von IP abzusehen und sich eng an die IEEE1394-Technologie angelehnt. Dadurch ist die Benutzung von IP allerdings nicht ganz ausgeschlossen, da auch IP-Pakete über den IEEE1394-Bus transportiert werden könnten (z.B. zum Zugriff auf AV-Streams im Internet), außerdem wurde in der HAVi-Spezifikation bereits das API für sogenannte Web Proxies als FCM festgelegt.

Die Festlegung auf IEEE1394 wurde vor allem durch die folgenden Eigenschaften von IEEE1394 begünstigt:

- Unterstützung isochronen Datentransfers für AV-Streams.
- hohe Geschwindigkeit und geringe Kosten.
- Unterstützung von Hot Plugging, d.h. Geräte können während des laufenden Betriebs an das Netzwerk angeschlossen und wieder entfernt werden.
- Es ist lediglich ein Netzwerk-Interface für Nachrichtenübertragung und Streaming notwendig.
- IEEE1394 ist das einzige NW-Interface, das HAVi-Geräte implementieren müssen. Damit können dann alle HAVi-Geräte miteinander vernetzt werden. Auch ist in HAVi das API des Communication Media Manager lediglich für IEEE1394 spezifiziert.

### 7.4.2 Discovery

Dienste (SE) müssen sich in HAVi jeweils bei ihrer lokalen Registry anmelden. Da die Registry in HAVi föderiert angelegt ist, können auch nicht-lokale SE diese Dienste auffinden, ohne dass sich neue Dienste mittels Multicast- oder Broadcast-Nachrichten bekannt machen müssen.

Bevor sich SE allerdings bei ihrer lokalen Registry anmelden können, müssen sie sich zuerst durch einen Aufruf von `Msg::MsgOpen` beim MS eine SEID holen und die SEID der Registry mittels `Msg::MsgGetSystemSEID` herausfinden. Danach kann sich ein SE bei seiner lokalen Registry über `Registry::RegisterElement` anmelden.

Dabei gibt das SE seine SEID und eine Reihe zusätzlicher qualifizierender Attribute (die ein- oder mehrwertig sein können) an. Die nachfolgende Tabelle gibt eine Übersicht über die wichtigsten Standardattribute, die bei der Registrierung angegeben werden (müssen).

| Attributname | IDL-Typ | mehrwertig? | verpflichtend für |
|---|---|---|---|
| ATT_AV_LANG | AvLanguage | nein | - |
| ATT_HUID | HUID | nein | DCM/FCM/App |
| ATT_SE_TYPE | SoftwareElementType | nein | alle SE |
| ATT_SE_VERS | SoftwareElementVersion | nein | DCM/FCM/System-SE |
| ATT_TARGET_ID | TargetID | nein | DCM/FCM/App |
| ATT_USER_PREF_NAME | UserPreferredName | nein | DCM/FCM |
| ATT_VENDOR_ID | VendorID | nein | DCM/FCM/App |

**Tabelle 6: wichtige vordefinierte Registry-Attribute**

Beim Anschließen eines BAV- oder LAV-Gerätes laufen folgende Schritte in der angegebenen Reihenfolge ab:

1. Anschluss (physisch)

2. Erkennung durch den IEEE1394-Bus (Bus-Reset wird ausgelöst), CMM1394 teilt das Ereignis (*NetworkReset*) jedem FAV/IAV mit.

3. Jeder DCMM wird durch dieses Ereignis benachrichtigt (alle DCMM haben diese Ereignisart bei ihrem lokalen CMM1394 abonniert) und holt sich eine Liste von GUIDs neuer und abgemeldeter Geräte.

4. Jeder DCMM holt sich Informationen über diese Geräte (standardisiertes SDD auslesen) und stellt den Typ jedes dieser Geräte fest (BAV oder LAV).

5. Bestimmung eines DCMM (wenn möglich derjenige mit der geringsten Anzahl installierter DCM Code-Units, der die Anforderungen der DCM Code Unit erfüllt), welcher die DCM Code Unit installiert. Der ausgewählte DCMM generiert bei der Installation ein *DcmInstallIndication*-Ereignis.

6. Vergabe der SEIDs durch das lokale Messaging System (`Msg::MsgOpen`), DCMs und FCMs melden sich damit bei der lokalen Registry (lokal = im Controller) an, die Registry generiert dabei *NewSoftwareElement*-Ereignisse.

7. Evtl. unterbrochene Streams werden bei Anschluss der entsprechenden Geräte durch den Stream Manager wieder aufgenommen.

Wird ein FAV- oder ein IAV-Gerät angeschlossen, ist die Reihenfolge der Schritte folgendermaßen:

1. Anschluss (physisch)

2. Erkennung durch den IEEE1394-Bus (Bus-Reset wird ausgelöst), CMM1394 teilt das Ereignis (*NetworkReset*) jedem FAV/IAV mit.

3. Jeder DCMM wird durch dieses Ereignis benachrichtigt (alle DCMM haben diese Ereignisart bei ihrem lokalen CMM1394 abonniert) und holt sich eine Liste von GUIDs neuer und abgemeldeter Geräte.

4. Alle DCMM holen sich Informationen über neue Geräte aus deren SDD und stellen den Typ des neuen Gerätes fest (FAV oder IAV).

5. Der Installationsprozess für DCM Code Units muss eventuell neu durchgeführt werden.

6. Evtl. unterbrochene Streams werden bei Anschluss der entsprechenden Geräte durch den Stream Manager wieder aufgenommen.

Suchen Klienten dann ein SE bestimmten Typs oder mit bestimmten Eigenschaften, so können sie sich mit einer Suchanfrage an die Registry wenden:

Mittels `Registry::GetElement(in SimpleQuery query, out sequence<SEID> seidList)` kann die Registry befragt werden. In *query* sind dabei die Attribute und Vorgaben für ihre Werte enthalten. *seidList* enthält als Rückgabeparameter dann eine Liste mit SEIDs von zur Suchanfrage passenden Software-Elementen. Im *query*-Objekt können dabei auch boolesche Vergleichsoperationen (equal, not equal, >, <, >=, <=) auf Attributwerte spezifiziert werden.

### 7.4.3 Dienstnutzung (entfernte Methodenaufrufe)

Wenn das gewünschte SE schließlich vom Klienten gefunden wurde, kann der Klient Funktionen des SE aufrufen. Der Aufruf einer Funktion eines anderen SE wird immer durch einen lokalen Aufruf beim MS realisiert. Damit ist es für den Klienten transparent, ob er eine lokale oder eine entfernte Funktion aufruft. Für den Aufruf muss der Klient unter anderem seine eigene SEID, die SEID des Ziel-SE und den eindeutigen OpCode der Funktion angeben.

Da Steuernachrichten unter Umständen länger sein können, bietet das Messaging System durch das sogenannte Transport Adaptation Module (TAM) die Segmentierung von Nachrichten (Vergabe von Sequenznummern) an. Das TAM ist derjenige Teil des MS, der vom Übertragungsmedium abhängig ist und bildet das HAVi-Nachrichtenformat auf IEEE1394-Nachrichtenpakete ab. Die Kommunikation über IEEE1394 selbst ist verbindungslos (Nachrichten als Datagramme).

Eine Ausnahme von dieser Art der Kommunikation besteht bei der Kommunikation zwischen einem LAV-Gerät und seinem Controller. Hierbei kommen proprietäre Kommunikations-protokolle zum Einsatz.

Ein Funktionsaufruf schließlich kann entweder synchron oder asynchron erfolgen:

- **Synchron**: der Klient blockiert solange bis er das Ergebnis des Funktionsaufrufs erhalten hat.

- **Asynchron**: der Aufruf kehrt sofort mit der transaction ID als vorläufigem Ergebnis zurück und die Antwortnachricht wird später mit derselben transaction ID zum Klienten geschickt.

**Synchroner Funktionsaufruf:**

1. Der Klient ruft
   `MsgSendRequestSync(in SEID sourceSeid, in SEID destSEID, in OperationCode opCode, in long timeout, …)`
   bei seinem lokalem Messaging System auf.
2. `MsgSendRequestSync` aktiviert MS-intern die Funktion `MsgSendReliable`, die die Zustellung an das Empfänger-MS versucht und entweder eine Empfangsbestätigung für eine erfolgreiche Zustellung oder eine Fehlermeldung zurückliefert.
3. Die Aufruf-Nachricht kommt beim Empfänger-MS an. Dieses ruft die registrierte Callback-Funktion asynchron beim Empfänger-SE auf. Wenn der Callback aufgerufen werden konnte, wird eine Empfangsbestätigung an das Sender-MS zurückgeschickt. Erhält das Sender-MS nicht innerhalb der festgelegten Timeout-Zeitspanne die Empfangsbestätigung, so wird dem Klienten ein Fehler zurückgemeldet und der Aufruf abgebrochen um zu lange Blockierungen zu verhindern.
4. Das Empfänger-SE liefert das Ergebnis durch einen Aufruf von `MsgSendResponse` bei seinem MS zurück.
5. Das Empfänger-MS schickt die Nachricht mit dem Ergebnis durch Aufruf der Funktion `MsgSendSimple` (unzuverlässiger Versand ohne Empfangsbestätigung) zurück.
6. Der Aufruf von `MsgSendRequestSync` kehrt mit dem Ergebnis zurück.

Die folgende Grafik soll diesen Ablauf verdeutlichen:

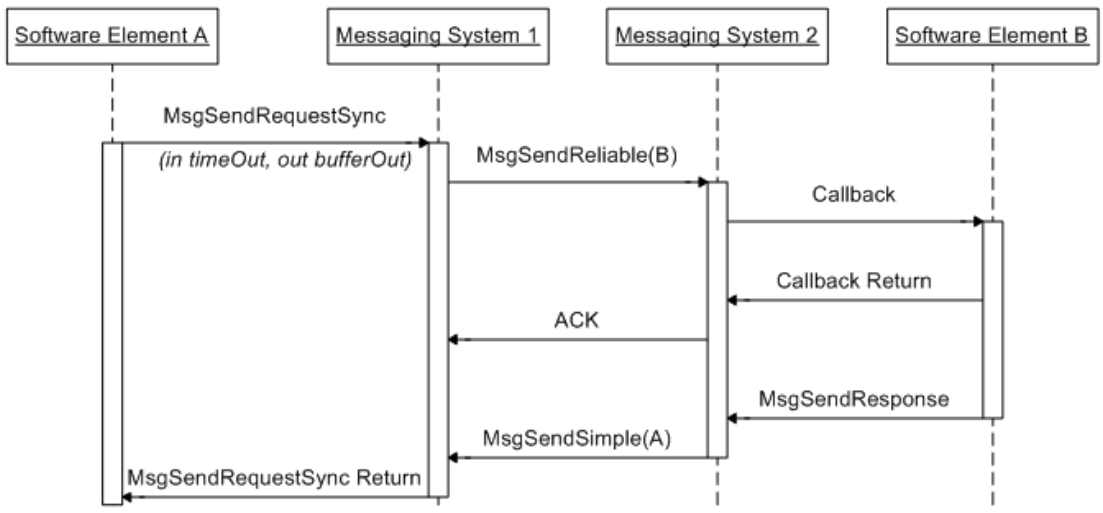



**Abbildung 11: Ablauf eines synchronen Funktionsaufrufs**

**Asynchroner Funktionsaufruf**:

Alternativ sind auch asynchrone Aufrufe durch Aufruf von `MsgSendRequest` möglich, bei denen der Klient nur bis zum Erhalt einer Empfangsbestätigung (Inhalt: transaction ID) blockiert. Durch den Erhalt der transaction ID ist es dem Klienten möglich, die Antwort zu einem späteren Zeitpunkt dem Aufruf zuzuordnen. Die folgende Grafik skizziert den Ablauf eines solchen asynchronen Funktionsaufrufs.

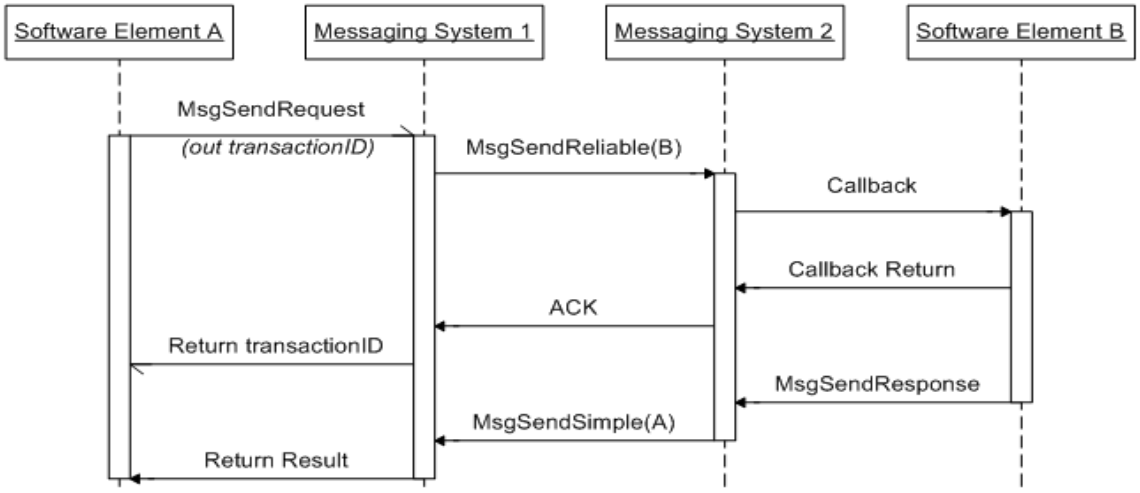



**Abbildung 12: Ablauf eines asynchronen Funktionsaufrufs**

### 7.4.4 Ereignisdienst

Der Ereignisdienst wird in HAVi durch die Föderation aller Event Manager im Netzwerk realisiert. Jede Zustandsänderung eines SE wird dem lokalen Event Manager als ein Ereignis mitgeteilt. Zu jeder Art von Ereignissen führt der Event Manager eine Liste mit an dem Ereignis interessierten lokalen SE – die sogenannte Abonnentenliste.

Um benachrichtigt zu werden muss ein Interessent an einem bestimmten Ereignis (Abonnent) sich bei seinem lokalen Event Manager registrieren und dabei eine Callback-Funktion angeben. Über diese Callback-Funktion wird der Abonnent beim Auftreten des entsprechenden Ereignisses benachrichtigt.

Neben beliebigen Ereignissen in den verschiedenen DCMs, FCMs und Applikationen beschreibt HAVi eine Reihe von vordefinierten Ereignissen, die von den System-SE ausgelöst werden können. Diese Ereignisse werden in Kapitel 11.9 der HAVi-Spezifikation beschrieben.

### 7.4.5 Streaming

Da sich HAVi insbesondere für Geräte der Unterhaltungselektronik eignen soll, legt die Spezifikation auch Mechanismen zum Streaming von Audio- und Videodaten über das Netzwerk

fest. Die Mechanismen werden dabei nicht in der HAVi-Spezifikation selbst definiert, sondern es werden die Streamingmechanismen der Basis-Technologien IEEE1394 und IEC 61883 verwendet.

IEEE1394 bietet dabei die Grundlagen für isochronen Datentransfer (d.h. Bandbreiten- und Timing-Garantien). Die Übertragung von Audio- und Videodaten findet dabei durch einen Broadcast-Mechanismus auf einem oder mehreren der 64 IEEE1394-Kanäle statt. Entsprechend der geforderten Quality of Service findet im Fehlerfall keine wiederholte Datenübertragung statt und die benötigte Bandbreite muss vor der eigentlichen Datenübertragung reserviert werden. IEC 61883 selbst bietet dann ein Streaming-Protokoll auf der Grundlage von IEEE1394 zur Übertragung von Multimediadaten. Dieses Protokoll regelt die Abbildung von Datenströmen auf IEEE1394-Pakete und führt ein sogenanntes Plug-Konzept (Datenquellen und Datensenken, die bei erfolgreichem Kompatibilitätstest miteinander verbunden werden können) ein. Daneben definiert IEC 61883 das sogenannte Function Control Protocol (FCP) zur Übertragung von Steuernachrichten an AV-Geräte als asynchrone IEEE1394-Nachrichtenpakete und spezifiziert auch die Konkretisierung der allgemeinen Konzepte für bestimmte Datenformate (u.a. für MPEG2).

### 7.4.6 Entfernen von Geräten aus dem Netzwerk

In HAVi gibt es automatisierte Administrationsmechanismen, die beim Entfernen eines Gerätes das Netzwerk neu konfigurieren und die anderen Geräte davon benachrichtigen. HAVi unterscheidet beim Entfernen eines Gerätes, ob das zu entfernende Gerät ein Controller für andere Geräte sein kann (also ein FAV bzw. IAV) oder ob es sich um ein „einfaches“ Gerät (BAV bzw. LAV) handelt.

Der Unterschied in der Behandlung dieser beiden Fälle liegt darin, dass ein Controller eventuell DCM Code Units von anderen Geräten installiert hat, die dann in einem Controller installiert werden müssen. Dementsprechend sind die Schritte beim Entfernen von Geräten aus dem Netzwerk wie folgt:

Ein BAV oder LAV wird entfernt:

1. Über den CMM1394 wird die Topologieänderung durch NetworkReset- und GoneDevice-Ereignisse bekanntgegeben.

2. Alle DCMMs empfangen die Ereignisnachrichten und der verantwortliche DCMM des Host-Gerätes deinstalliert die DCM Code Unit des zu entfernenden BAV- oder LAV-Gerätes.
   Beim Deinstallationsvorgang deregistrieren sich die DCM Komponenten bei der Registry (wodurch ein GoneSoftwareElement-Ereignis ausgelöst wird) und anschließend dem Messaging System (MsgLeave-Ereignis wird ausgelöst) des Controllers.

3. Der DCMM generiert ein DcmUninstallIndication-Ereignis, die MS jeden Gerätes empfangen dieses Ereignis. MS, die eine SE im Auftrag eines anderen SE überwachen, benachrichtigen dann das überwachende SE über das Verschwinden des überwachten SE.

FAV oder IAV wird entfernt

1. Über den CMM1394 wird die Topologieänderung durch NetworkReset- und GoneDevice-Ereignisse bekanntgegeben.

2. Der DCMM des zu entfernenden Gerätes deinstalliert die bei ihm installierten DCM Code Units und generiert dabei für jede entfernte DCM Code Unit ein DcmUninstallIndication-Ereignis, von dem sämtliche MS und die überwachenden SE benachrichtigt werden.

3. Die DCMM jeden Gerätes führen anschließend den DCM-Installationsprozess (wie im Kapitel „Discovery“ beschrieben) durch.

## 7.5 Anmerkungen und Schwachpunkte

Da sich HAVi bislang die Steuerung von Audio- und Videogeräten sowie die Datenübertragung zwischen diesen Geräten konzentriert, können andere zu vernetzende Geräte – insbesondere solche ohne IEEE1394-Netzwerkschnittstelle - bislang nur sehr eingeschränkt von den Vorteilen dieses Standards profitieren (proprietäre Anbindung solcher Geräte an ein HAVi-Netzwerk) .

Laut Aussagen auf der HAVi-Website (www.havi.org) ist die HAVi-Organisation aber auch bemüht, Standards für Brücken zu anderen Architekturen der Gerätevernetzung wie z.B. Jini oder UPnP zu entwickeln. Inwieweit diese Anstrengungen schon Früchte tragen lässt sich allerdings schwierig beurteilen, da noch keine Dokumente öffentlich vorliegen, die solche Standards spezifizieren.

Ein weiterer Nachteil besteht in der Beschränkung auf lokale Netzwerke.

## 7.6 Verfügbare Implementierungen

| Firma | Weitere Informationen |
|---|---|
| **VividLogic** FireBus HAVi Stack und FireBus SDK: | http://www.vividlogic.com |
| **TwonkyVision** HAVi Development Environment: | http://www.twonkyvision.de |
| **dmn** HAVi stack for embedded systems<br><br>und **dmn** HAVi SDK: | <br><br>http://www.dmn.at |

**Tabelle 7: HAVi-Implementierungen**

# 8 OSGi Service Platform

Die Open Services Gateway Initiative (OSGi) wurde im März 1999 als Non-Profit Organisation ins Leben gerufen. Es sind bis heute schon mehr als 80 Firmen der Initiative beigetreten. OSGi ist auf dem Weg zum Standard im Bereich der **Services-Gateways** zu werden, worauf man aus der langen Liste der beteiligten namhaften Firmen schießen kann [OSGI-Mitglieder]. Zudem haben viele dieser Firmen auch schon eigene Implementierungen auf den Markt gebracht, womit sie zeigen, dass sie der OSGi gute Zukunftsaussichten einräumen. OSGi unterscheidet sich stark von den anderen betrachteten Architekturen, da es nicht dazu bestimmt ist, den Zugriff auf verteilte Dienste zu ermöglichen, sondern als ein Gateway zwischen einem Heimnetzwerk und entfernten Diensten fungiert.

## 8.1 Begriffsdefinitionen

Die folgenden Begriffe werden häufig verwendet und sind von zentraler Bedeutung. Darum werden sie hier übersichtlich in einer Tabelle zusammengefasst.

| Begriff | Definition |
|---|---|
| Bundle | Funktions- und Verteilungs-Einheit für das Ausliefern von Services.<br><br>(In Form eines Java JAR Files.) |
| Gateway Operator | Administriert den Service Gateway. |
| OSGi Environment | Kombination aus dem OSGi Framework und dem Java Runtime Environment. |
| OSGi Framework | Umgebung in die man Services einbilden kann. |
| Services Gateway | Schnittstelle zwischen WAN und LAN, an den Services Gateway werden die Endgeräte angeschlossen. Er besteht aus dem OSGi Framework und zusätzlichen Diensten. |
| Services Platform | Fast das selbe wie der Services Gateway, allerdings um einige Services erweitert. Wurde in Release 2 der OSGi Spezifikation eingeführt. |
| Service Provider | Stellt einen Service zur Verfügung. |
| Services | Java Klassen, die eine bestimmte, meist durch ein Interface definierte, Funktionalität bieten. (Dienste) |

## 8.2 Überblick

Das Ziel der OSGi ist es offene Standards zu definieren, welche die Verwaltung von Services und deren Verteilung über ein WAN auf lokale Netze und schließlich Endgeräte regeln.

Dazu definiert die OSGi die API eines Service Gateway, welche als Schnittstelle zwischen LAN und WAN dient. Dieser Service Gateway ist ein administrationsfreier eingebetteter Server der auf der JVM ausgeführt wird.

Zusätzlich werden von der OSGi in der Form von APIs Schnittstellen für das Life Cycle Management von Diensten, Data Management, Device Management, Client Access, Ressource Management und Security definiert.

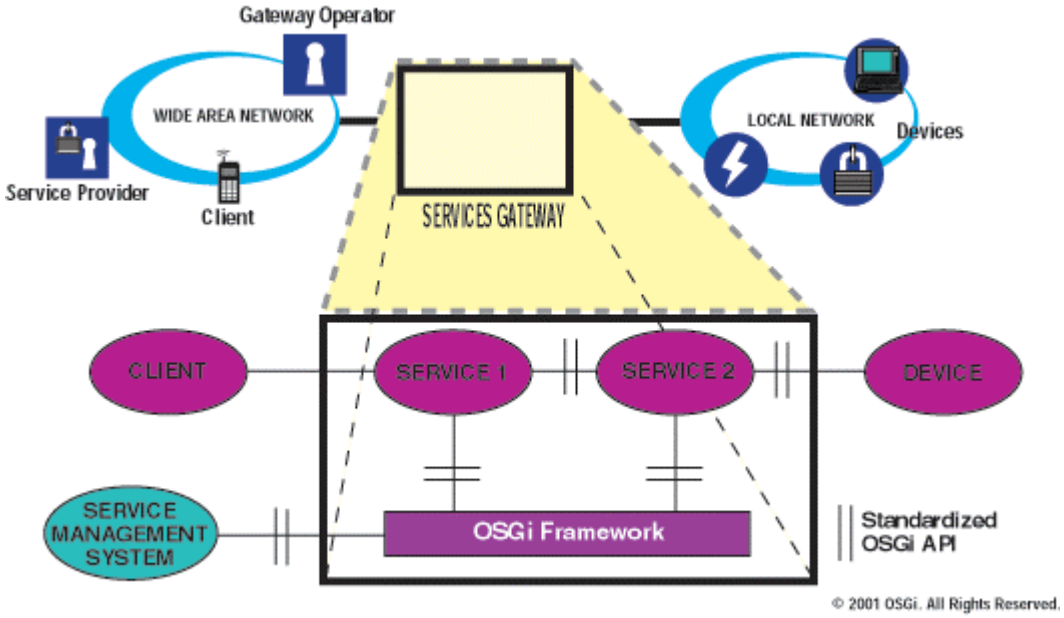



**Abbildung 13: Architektur der OSGi Services Platform (Quelle: OSGi Spezifikation)**

### 8.2.1 Was bietet die OSGi?

**Plattform-Unabhängigkeit**

Die zur Zeit existierenden Implementierungen in Java sind auf allen Hardware Plattformen, für die eine JVM (Java 2 Standard Edition) existiert, ausführbar. Da aber von der OSGi nur APIs spezifiziert werden, könnte man diese natürlich auch für zusätzliche Hardware Plattformen und Betriebssysteme implementieren.

**Sicherheit**

Die OSGi Spezifikation beinhaltet viele unterschiedliche Stufen von Sicherheits-Features. Diese reichen vom digitalen Signieren der übertragenen Module bis hin zu fein abgestufter Zugriffskontrolle auf einzelne Objekte.

**Zahlreiche Services**

OSGi ermöglicht das Hosten einer großen Anzahl von Services von verschiedenen Service Providern auf einer einzigen Service Platform. Diese Flexibilität ermöglicht es einem Gateway Operator seinen Kunden eine sehr große Auswahl an verfügbaren Services anzubieten.

**Integration zahlreicher Technologien und Standards**

Obgleich die Kerntechnologie, auf die OSGi zurückgreift, Java (insbesondere die JVM) ist, lassen sich ebenso Wireless Protocols wie Bluetooth, HomeRF und ShareWave integrieren, als auch mit Jini konkurrierende Vernetzungsstandards wie MS Universal Plug and Play(UPnP) oder Echelon LonWORKS, sowie außerdem z.B. HomePNA , X-10, HAVi und CEBus.

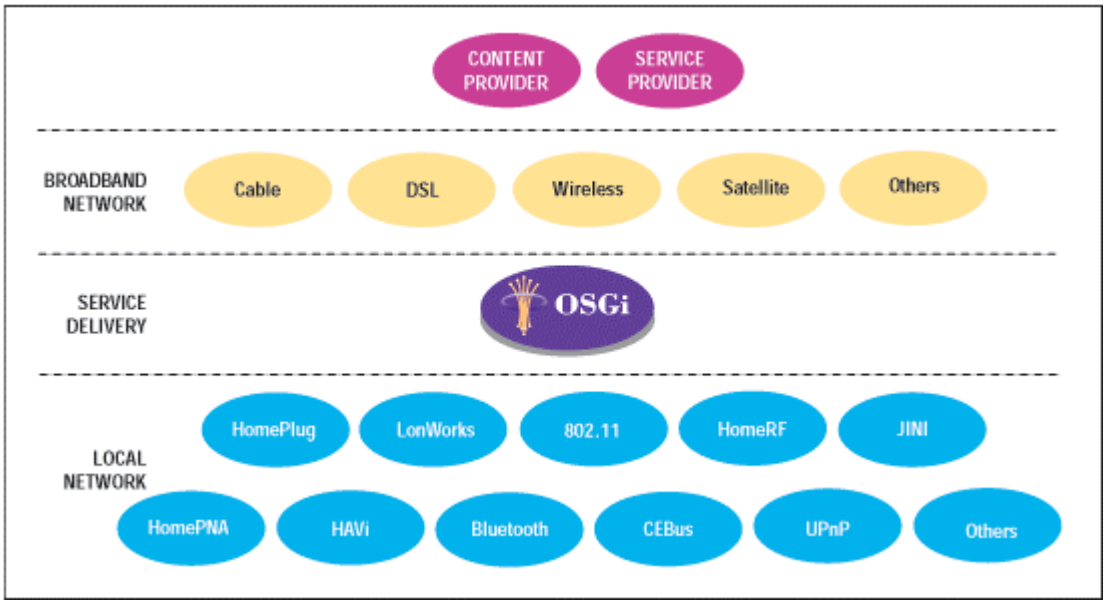



**Abbildung 14: Mit der OSGi in Beziehung stehende Techniken (Quelle: OSGi Spezifikation)**

Diese Integration ist möglich, da die Technologien zur Gerätevernetzung wie z.B. UPnP und Jini sehr ähnlich aufgebaut sind - was wohl daher rührt, dass sie alle geschaffen wurden um ungefähr die selben Probleme zu lösen.

### 8.2.2 Einsatzmöglichkeiten der OSGi

Denkbare Anwendungsgebiete für OSGi-basierte Lösungen sind:

- Geteilte Internet-, Sprach-, Daten-, Multimedia-Dienste,
- Alarm und Sicherheit (Home Security),
- Energiekontrolle, -messung und -management,
- Gesundheitswesen (Patientenüberwachung),
- Haushaltsausrüstung: Gateway für Beobachtung (Monitoring) und Kontrolle,
- Content und E-Commerce Dienste (z.B. automatisierte Lagerhaltung im Retail-Bereich),
- Automobil / Telematik und
- Industrieautomation.

### 8.2.3 Architekturüberblick

Das OSGi Architektur Modell ist als eine Ende-zu-Ende Lösung konzipiert..

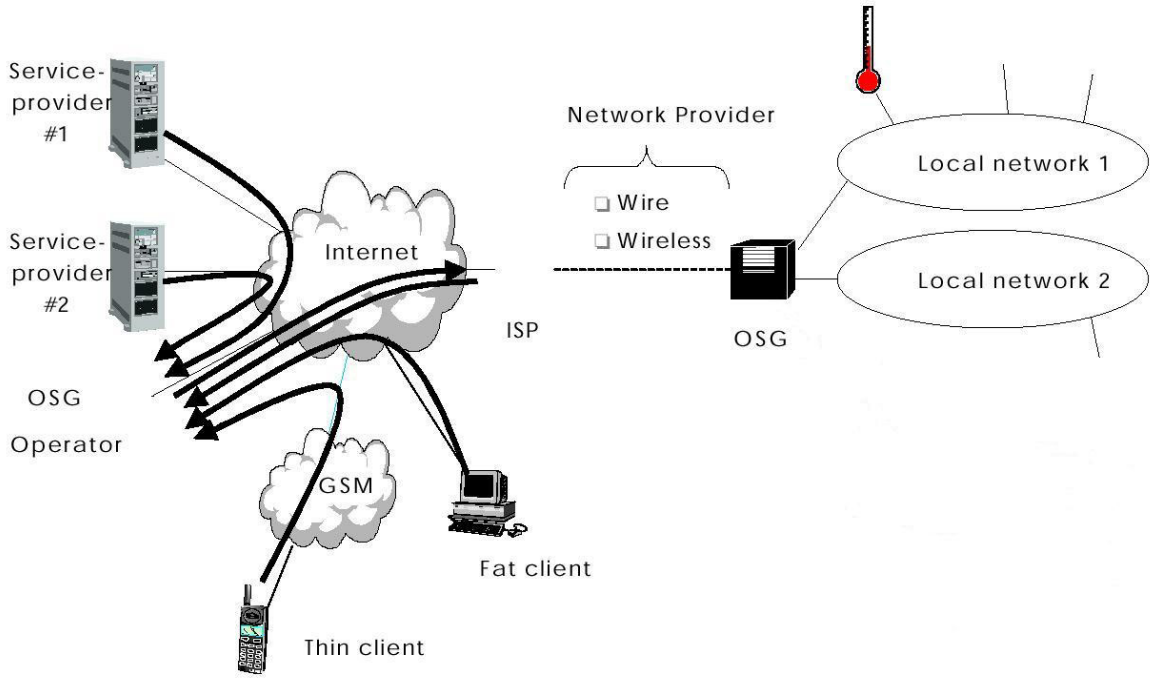



**Abbildung 15: Die Ende-zu-Ende Architektur der OSGi (Quelle: OSGi Spezifikation)**

In den folgenden Abschnitten werden nun die wichtigsten Komponenten dieser Architektur vorgestellt. Das Bild soll die Beziehungen zwischen diesen Komponenten verdeutlichen

**Service Provider**

Der Service Provider stellt als Anbieter einen Service zur Verfügung. Dieser ist eine Software Anwendung, die von dem Gateway Operator auf den jeweiligen Service Gateway transferiert wird.

Wenn einem Service Provider von dem Gateway Operator vertraut wird, sorgen sichere Download-Techniken dafür, dass nur vertrauenswürdiger Quellcode auf dem Service Gateway installiert wird. Dies wird z.B. mit Hilfe von digital signiertem Code und Verschlüsselungstechniken erreicht.

Ein *Service Aggregator* ist eine spezielle Art eines Service Providers, er stellt eine Auswahl von Services zur Verfügung und sorgt dafür, dass die angebotenen Services alle zueinander kompatibel sind und keine Konflikte bei der Nutzung der Endgeräte auftreten können.

**Gateway Operator**

Der Gateway Operator steuert und pflegt den Service Gateway über das WAN. Dadurch muss der Benutzer des Service Gateways diesen nicht selbst administrieren.

Einige Aufgaben, die der Gateway Operator erfüllt, sind:

- Downloaden, Entfernen, Starten und Stoppen eines Service.
- Den Status des Gateways überwachen, Versionen der installierten Services aktuell halten.
- Zugriffsrechte zwischen dem Gateway und dem Service Provider vergeben und überwachen.
- Sichere Kommunikation zwischen dem Gateway und dem Service Provider ermöglichen.
- Abhängigkeiten und Zugriffsrechte zwischen den Services überwachen.

**WAN und Provider (ISP)**

Ermöglicht die Kommunikation zwischen dem Service Gateway, dem Gateway Operator und dem Service Provider (Aggregator). Das WAN ist meistens das Internet, welches von einem Internet Service Provider (ISP) zur Verfügung gestellt wird.

Für den praktischen Einsatz wäre ein ISP, der gleichzeitig auch die Funktion des Gateway Operator übernehmen würde, eine geschickte Kombination.

**LAN und Geräte**

Dieser Bestandteil der Architektur macht den Service Gateway zu mehr als nur einem normalen Computer, da man nun an den Gateway direkt Endgeräte mit Hilfe beliebiger Netzwerktechnologien anschließen kann. Dies können herkömmliche Techniken wie Ethernet, Firewire oder USB sein, aber auch drahtlose Techniken wie z.B. BlueTooth oder WLAN. Durch den Gateway wird es auch möglich die Geräte nicht nur innerhalb des lokalen Netz zu nutzen, sondern sie können auch von Klienten aus der Entfernung, z.B. über das Internet genutzt werden.

**Service Gateway (OSG)**

Der Open Service Gateway ist die zentrale Komponente der OSGi Service Plattform. Er ist ein embedded Server der auf einem Gerät wie z.B. einem DSL-Modem läuft und er stellt die Schnittstelle zwischen dem WAN einerseits und dem LAN sowie dessen Geräten andererseits dar. Der Service Gateway muss lokal nicht administriert werden, da dies, wie vorher beschrieben der OSG Operator macht.

### 8.2.4 Service Framework

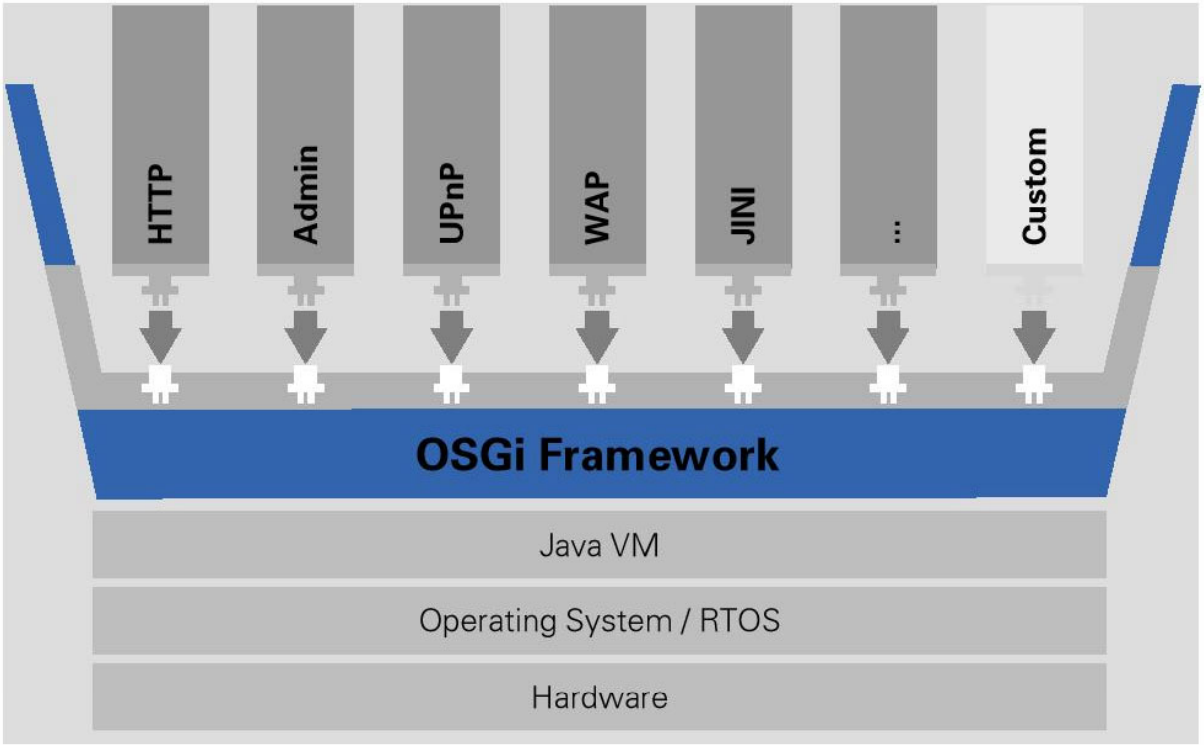
Das Service Framework ist die Umgebung, auf der die Services installiert und ausgeführt werden. Das Service Framework befindet sich auf dem Service Gateway, dadurch hat es eine direkte Verbindung zu allen Geräten im lokalen Netz und ebenfalls auch Anschluss an das WAN.

Die wichtigsten Bestandteile und Aufgaben des Service Framework sind:

- Bundles (Software Einheiten, die man im Framework installieren kann)
- Services (Dienste die von einem Bundle zur Verfügung gestellt werden)
- Das An- und Abmelden von Services mit Hilfe der „Framework Service Registry“
- Kommunikation mit Hilfe vom Events
- Sicherheitsfunktionen des Frameworks

Diese Bestandteile werden im Folgenden näher erläutert. Jedoch ist damit das Framework noch nicht komplett. Es werden zusätzlich noch weitere Funktionen benötigt (grundlegende Services), damit das Framework nutzbar ist. Diese Services werden im nächsten Kapitel näher beschrieben.

Auf der folgenden Abbildung sieht man eine schematische Darstellung des Frameworks. Es basiert auf der JVM, welche natürlich für die genutzte Hardware und das Betriebssystem verfügbar sein muss. In das Framework können zur Laufzeit die **Bundles**, welche einen speziellen Service implementieren, eingebunden werden.




**Quelle** ProSyst, WestLB Panmure

**Abbildung 16: Das OSGi Framework**

Ein Bundle ist die Einheit um eine Java basierte Applikation einzusetzen. Ein Bundle ist eine spezielle Form einer gewöhnlichen Java Archiv Datei (JAR). Es können von einem Bundle Funktionen für Endbenutzer zur Verfügung gestellt werden, aber auch Komponenten, die von anderen Bundles genutzt werden können. Diese Komponenten werden dann **Services** genannt.

Ein Service wird von seinem *service interface* definiert. Dieses Interface ist ein gewöhnliches Java Interface. Implementiert wird das Service Interface von einem *service object*. Diese Service-Objekte werden von den Bundles zur Verfügung gestellt und verwaltet. Um die Funktionalität eines Bundles anderen Bundles und dem Benutzer verfügbar zu machen, müssen sich die Bundles bei einer so genannten „**Framework Service Registry“** anmelden. Bei der Registrierung können zu einem *service object* noch Attribute angegeben werden. Mit Hilfe dieser Attribute steht dann eine sehr differenzierte Methode zum Auffinden eines passenden Objekts zur Verfügung. Bei der Registrierung wird ein **ServiceEvent** ausgelöst. Damit werden die anderen Bundles auf die Änderung aufmerksam gemacht. Es gibt drei Arten von Events, die im Kapitel über die Kommunikationsmechanismen näher beschrieben werden.

Die **Sicherheit** basiert auf der Java Sicherheits-Architektur für JDK 1.2. Dadurch finden umfangreiche Prüfungen der Zugriffsrechte statt. Es ist möglich folgende drei Typen von Rechten zu vergeben:

- *AdminPermission*: Gibt das Recht das Framework zu administrieren.
- *ServicePermission*: Kontrolliert, ob ein Bundle das Recht hat einen bestimmten Service anzubieten. Auf der anderen Seite wird damit auch geregelt, welche Bundles berechtigt

sind einen angebotenen Service zu nutzen. Ein Bundle, das an einem bestimmten Service kein Nutzungsrecht besitzt, ist nicht einmal dazu berechtigt dessen Präsenz festzustellen, wenn es nach einem Service dieser Art sucht.

- *PackagePermission*: Kontrolliert, welche Pakete von einem Bundle importiert und exportiert werden dürfen.

### 8.2.5 Grundlegende System-Dienste

Das Framework selbst wird als System-Bundle bezeichnet. Ihm wird immer der Bundle-Bezeichner *Null* zugewiesen. Mit Hilfe des System-Bundles können sich andere Bundles registrieren und auffinden. Zusätzlich gibt es noch viele weitere wichtige Standard-Dienste, ohne die ein Funktionieren nicht möglich ist.

Einige von diesen werden in der folgenden Tabelle kurz erläutert.

| Dienstname | Kurzbeschreibung der Funktion des Dienstes |
|---|---|
| Package Admin Service | Funktionen zur Verwaltung gemeinsam genutzter Packages. Wie z.B. Statusabfrage oder Aktualisierung eines Packages. |
| Permission Admin Service | Verwaltet die Rechte einzelner Bundles und stellt Standardrechte für alle Bundles zur Verfügung. Eine FilePermission ist ein Beispiel für eine solche Berechtigung (diese Berechtigung würde einem Bundle erlauben auf eine spezielle Datei zuzugreifen). |
| Log Service | Erlaubt das Aufzeichnen von Nachrichten. Dies kann man z.B. zum Aufzeichnen von Fehlermeldungen benutzen. Der Log Service stellt auch Funktionen zum Finden aufgezeichneter Informationen bereit. Dies können die Bundles nutzen, um sich gegenseitig Nachrichten zu schicken. |
| http Service | Erlaubt den Zugriff auf Services über einen Web Browser, d.h. der http Service stellt ein User-Interface mit Hilfe der Standard-Technologien http, html, xml und Java-Servlets zur Verfügung. |
| Service Tracker | Da sich an der Framework Service Registry ständig Services an- und abmelden, ist es nötig, dass Änderungen immer verfolgt werden. Ansonsten kann es z.B. passieren, dass man ein Service nutzen möchte, der sich jedoch kurz zuvor abgemeldet hat.<br><br>Dazu gibt es den Service Tracker. Er erstellt beim Start eine initiale Liste aller Services. Zur Laufzeit wird jede Änderung verfolgt und die Liste aktualisiert. Außerdem werden alle Events aufgezeichnet und an Bundles, die am Service Tracker registriert sind, weitergegeben. |

| Dienstname | Kurzbeschreibung der Funktion des Dienstes |
|---|---|
| Device Access Specification | Stellt Techniken zur automatischen Erkennung und Verbindung von Geräten mit der OSGi Umgebung zur Verfügung. Ermöglicht Hot-Plugging und –Unplugging von neuen Geräten, sucht automatisch passende Gerätetreiber und installiert diese.<br><br>Im Kapitel Kommunikationsmechanismen wird genauer auf die Device Access Specification eingegangen. |
| Configuration Admin Service | Erlaubt das Konfigurieren von installierten Bundles. |
| Preferences Service | Mit Hilfe dieses Services können Bundles Informationen permanent speichern. Dies können z.B. Einstellungen sein, die ein Benutzer für sich gemacht hat, aber auch globale Einstellungen für das gesamte Bundle. Die Daten werden in einer hierarchischen Baumstruktur abgelegt. |
| User Admin Service | Mit diesem Service ist es möglich, Benutzer zu verwalten, zu authentifizieren und zu autorisieren. Es werden auch Gruppen und Rollen zur effizienteren Verwaltung von Benutzerrechten angeboten. |

**Tabelle 8: Die Standard-Dienste der OSGi Services Platform.**

## 8.3 Kommunikationsmechanismen

### 8.3.1 An- und Abmeldung von Services

Ein Bundle enthält eine beliebige Anzahl an Services. Diese Services muss ein Bundle bei dem Framework registrieren, damit sie von anderen Bundles oder Benutzern gefunden und benutzt werden können. Zu diesem Zweck stellt das OSGi Framework die *Framework serice registry* zur Verfügung. Diese wird genutzt, indem das Bundle für jeden Service eine dieser beiden Methoden aufruft:

- registerService(String, Object, Dictionary)
- registerService(String[], Object, Dictionary)

Dabei wird in dem **String** der Name des Service-Interface angegeben, das der Service implementiert. Wenn er mehrere Service-Interfaces implementiert, steht die zweite Methode mit einem String-Array zur Verfügung. Das **Objekt** ist des Service-Objekt selbst. Durch das **Dictionary** kann der registrierte Service mit Hilfe der darin enthaltenen Attribute näher beschrieben werden. (siehe Kapitel Dienstbeschreibungen).

Zurück gibt die Methode ein *ServiceRegistration*-Objekt. Dieses Objekt ist der einzige zulässige Weg um Änderungen an dem Service vorzunehmen.

Zudem kann nur mit diesem Objekt ein Service auch wieder abgemeldet werden. Die Abmeldung bewerkstelligt die Methode `unregister()` des zuständigen ServiceRegistration-Objektes. Dieses Objekt kann auch weitergegeben werden, und so kann nicht nur das Bundle, das einen Service registriert hat, diesen wieder abmelden, sondern jeder, dem das Bundle dieses Objekt anvertraut hat.

Zur Verdeutlichung nun ein Beispiel für das Anmelden des Greeting Services, dessen Interface in Kapitel 8.4 beschrieben wird.

```
package greeting2.impl;

import java.util.Properties;
import org.osgi.framework.*;
import greeting2.service.GreetingService;

public class Activator implements BundleActivator {
   private ServiceRegistration reg = null;

   public void start(BundleContext ctxt) throws BundleException {
      GreetingService greetingSvc = new CasualGreetingImpl();
      Properties props = new Properties();
      props.put("description", "casual");
      reg = ctxt.registerService("greeting2.service.GreetingService",
                                 greetingSvc, props);
   }

   public void stop(BundleContext ctxt) throws BundleException {
      if (reg != null)
      reg.unregister();
   }
}
```

### 8.3.2 Nutzung eines Service

Auf einen Service wird im Allgemeinen durch ein *ServiceReference*-Objekt zugegriffen. Über dieses *ServiceReference*-Objekt kann abgefragt werden, welche Service-Interface-Klassen der Service implementiert, und es können die Eigenschaften des Service, die bei dessen Registrierung gesetzt wurden, in Erfahrung gebracht werden. Um diese *ServiceReference* zu erhalten, ruft man die Methode `getServiceReference(String)` des zuständigen BundleContexts auf. Dabei ist der String der Name des Interfaces, das der gesuchte Service implementieren soll. Zurück gegeben wird dann die gewünschte ServiceReference. Wird kein passender Service gefunden bekommt man NULL als Rückgabewert.

Der nächste Schritt besteht darin, das eigentliche ServiceObjekt zu besorgen. Dies geschieht wieder durch den BundleContext. Durch das Anfordern des Services entsteht eine Abhängigkeit von dessen Funktionstüchtigkeit. Das bedeutet, wenn der angeforderte Service ausfällt muss auch der Service, der diesen benutzt, benachrichtigt werden um entsprechend reagieren zu können. Dadurch, dass man das Service Objekt am BundleContext anfordert, werden diese Abhängigkeiten automatisch verwaltet. Zudem wird der Nutzungs-Zähler des Services um eins erhöht. Dieser wird geführt, damit dem Service bekannt ist, wie oft er insgesamt genutzt wird, oder ob er ungenutzt ist.

Die Methode `BundleContext.getService(ServiceReference)` gibt das gewünschte Service Objekt, entsprechend der angegebenen ServiceReference zurück. Das erhaltene Objekt muss nun noch auf dessen Typ gecastet werden (d.h. eine Typkonvertierung muss durchgeführt werden), und danach können seine Methoden - wie sie definiert sind - aufgerufen werden.

Wird das Objekt nicht mehr benötigt, so kann es mittels der Methode `BundleContext.ungetService(ServiceReference)` wieder gelöscht werden. Dadurch wird der Nutzungs-Zähler wieder um eins reduziert, und die überwachten Abhängigkeiten entfernt.

### 8.3.3 Nachrichten-basierte Kommunikation

Bei OSGi werden die folgenden drei Klassen von Nachrichten unterschieden:

- *FrameworkEvents* (asyncron) benachrichtigen über das Starten bzw. Stoppen des Frameworks oder über aufgetretene Fehler.
- *BundleEvents* (asynchron) geben Veränderungen im Lebenszyklus von Bundles weiter. Ein Bundle kann installiert, gestartet, gestoppt, erneuert und deinstalliert werden.
- *ServiceEvents* (synchron) benachrichtigen über Anmeldung, Abmeldung und Änderungen von Services. Zusätzlich finden bei den ServiceEvents auch Sicherheitsprüfungen statt, so dass nur die Bundles benachrichtigt werden, die auch Zugriffsrechte auf den benachrichtigenden Service besitzen.

Für jede dieser Klasse gibt es entsprechende Event-Listener. Will ein Bundle Nachrichten von einem der drei Event-Typen erhalten so registriert es den zuständigen Event-Listener an der Stelle, die diese Nachrichten verschicken könnte.

### 8.3.4 Log Service

Der Log Service kann zur Kommunikation eines Bundles mit dem Gateway Operator eingesetzt werden, indem das Bundle z.B. Fehlermeldungen oder Statusberichte über den Log Service bekannt gibt.

Dadurch dass es den Bundles auch möglich ist die vormals gemachten Eintragungen zu filtern und zu lesen, kann der Log Service natürlich auch gut eingesetzt werden um Nachrichten zwischen den Bundles auszutauschen. Dabei besteht gegenüber Ereignis-basierter Kommunikation der Vorteil, dass die beiden Kommunikationspartner nicht zur selben Zeit aktiv sein müssen.

### 8.3.5 Device Access Specification

Die Device Access Specification beschreibt, wie die Gerätevernetzung unter OSGi vonstatten gehen soll. Dazu schreibt diese Spezifikation keine bestimmten Geräte oder Netzwerktechnologien vor, die hier vorgestellten Techniken dienen nur als Beispiele zur Verdeutlichung. Genauso wenig wird eine bestimmte Discovery-Methode vorgegeben. Durch die flexible Architektur ist es möglich, fast jede Art von Gerätevernetzungs-Technologien, wie z.B.

Firewire, USB oder serielle Schnittstellen zu integrieren und die vernetzten Geräte dann einheitlich anzusprechen.

Eine Implementierung dieser Spezifikation muss folgende Punkte erfüllen, damit sie als OSGi-kompatibel gelten kann:

- Hot-Plugging zu jeder Zeit, wenn die Hardware des Gerätes dies unterstützt.
- Legacy Geräte müssen unterstützt werden. Damit sind Geräte gemeint, die von sich aus kein Hot-Plugging unterstützen.
- Dynamisches Laden eines passenden *DeviceDriver.*
- Mehrfache Geräte-Repräsentationen, d.h. ein Gerät muss von verschiedenen Abstraktionsebenen aus angesprochen werden können. Dieses wichtige Prinzip wird in dem Beispiel gegen Ende dieses Kapitels noch näher erläutert.
- Tiefe Bäume: Geräte werden in Form einer Baumstruktur, aus verschiedenartigen Netzwerktechnologien, unterschiedlicher Tiefe, angeschlossen.
- Unabhängigkeit von der Netzwerkarchitektur: Die Schnittstelle eines Gerätes muss unabhängig davon, wo und wie es angeschlossen wurde, verfügbar sein.
- Komplexe Geräte: Es müssen Geräte unterstützt werden, die mehre Funktionen besitzen und verschiede Einstellungsmöglichkeiten haben.

Diese Spezifikation selbst ist kein Service wie man denken könnte, sondern sie definiert ein Verhalten. Dieses Verhalten wird durch mehre Services erreicht. Die Hauptbestandteile sind die in der folgenden Tabelle beschriebenen.

| Name der Komponente | Beschreibung der Funktion |
|---|---|
| Device Manager | Kontrolliert die Initiierung des Verbindungsprozesses. |
| Device Category | Definiert wie ein Device Service und ein Driver Service zusammenarbeiten können. |
| Driver Service | Kümmert sich darum, die Device Services entsprechend der erkannten Device Category mit dem Framework zu verbinden. |
| Device Service | Repräsentiert ein physisch vorhandenes Gerät. |
| DriverLocator | Hilft beim Finden eines Bundles das einen Driver Service zur Verfügung stellt. |
| DriverSelector | Hilft bei der Erkennung, welcher Driver Service am besten zu einem Device Service passt. |

**Tabelle 9: Die Hauptkomponenten der Device Access Specifikation**

Nun zur Verdeutlichung ein Beispiel. Es wird beschrieben, wie an ein OSGi Framework mit USB Unterstützung eine USB Maus angeschlossen wird. Durch das Einstecken der Maus wird folgender Ablauf ausgelöst:

1. Die USB Hardware erkennt, das ein neues Gerät angeschlossen wurde. Darum registriert sie ihren Device Service. Dieser Service repräsentiert ein generisches USB Gerät und hat noch keinerlei Eigenschaften einer Maus. Dies ist die erste Form der Repräsentation der USB Maus als Gerät im Framework: **Ein Standard-USB Gerät**.

2. Der **Device Manager** bemerkt dadurch, dass ein neues USB Gerät angeschossen wurde und startet nun eine Sequenz, die den Zweck hat immer genauere und spezifischere Treiber für das neue Gerät zu finden.

3. Zuerst sucht der **DriverLocator** nach passenden Driver Services die den bisherigen Device Service (USB Gerät) besser beschreiben.

4. Der oder die gefundenen Driver Services werden installiert und gestartet.

5. Dies erkennt wiederum der Device Manager und stellt mit Hilfe des **DriverSelector** fest, welcher Driver am besten zu dem vormals registrierten Gerät passt.

6. Dieser Driver wird nun aufgefordert, sich mit dem Device zu verbinden. Gelingt dies, so wird eine Abhängigkeit zwischen den betroffenen Bundles erzeugt.

7. Wenn die Verknüpfung vollständig ist, registriert der neue Driver einen neuen Device Service. In unserem Beispiel wird ein Mouse-Service registriert, welcher die allgemeinen Funktionen einer Maus repräsentiert. Dies ist die zweite Form der Repräsentation der USB Maus als Gerät im Framework: **Eine Standard-Maus**.

8. Nun folgt wieder der selbe Schritt wie bei 3. und es wird versucht das Gerät noch genauer zu beschreiben. Z.B. könnte man sich vorstellen, dass nun ein Treiber für einen bestimmten Typ von Maus geladen wird, die ein Scroll-Rad besitzt. Wenn der Driver Locator in Schritt 4 nichts mehr findet bricht der Vorgang ab und der Anschlußvorgang ist beendet.

## 8.4 Dienstbeschreibungen

Dienste werden bei OSGi durch das sogenannte Service-Interface definiert. Dieses Service Interface ist ein Java-Interface. Die OSGi-Spezifikation definiert mehrere generische Standard Dienste. In der Zukunft sollen noch weitere definiert werden.

Zur Verdeutlichung folgt nun ein Code-Beispiel für ein Service-Interface:

```
package greeting2.service;

public interface GreetingService {

/**

* Prints a form of greeting to the stdout for the named

* individual.

* @param firstName First name of the person

* @param lastName Last name of the person

* @param title Title of the person, e.g., Mr./Ms./Sir

*/

  public void greet(String firstName, String lastName, String title);

}
```

Dieser `GreetingService` besitzt nur eine Funktion, die durch die Methode `greet` zur Verfügung gestellt wird. Der Service dient dazu einen Gruß zurückzugeben.

Als zusätzliche Möglichkeit der Beschreibung eines Dienstes stehen Eigenschaften (Properties) zur Verfügung. Diese können als Schlüssel-Wert Paare angegeben werden, wobei der Schlüssel als eine Zeichenkette und der Wert als ein beliebiges Objekt repräsentiert werden. Bei der Angabe der Zeichenkette für den Schlüssel wird nicht auf Groß- und Kleinschreibung geachtet. Falls einmal Schlüssel eingegeben werden, die sich nur in der Groß- und Kleinschreibung unterscheiden, wird eine Ausnahme ausgelöst.

Mit Hilfe des Filter-Interfaces, einem Service des OSGi Framework, wird ein komplexes Filtern der Attribute aller Services möglich, um ein passendes Service Objekt aufzufinden.

In der folgenden Tabelle werden die Standard-Attribute für OSGi Services aufgelistet und erläutert.

| Schlüssel der Eigenschaft | Typ des Wertes | Beschreibung der Eigenschaft |
|---|---|---|
| service.description | String | Kurzbeschreibung der Funktionen des Services. |
| service.id | Long | Ein eindeutiger Wert, der alle registrierten Services voneinander unterscheidbar macht. Die id wird ständig inkrementiert, d.h. ein Service der sich nach einem Anderen registriert hat, wird dadurch sicher auch eine höhere id Nummer, als dieser zugewiesen bekommen. |
| service.pid | String | Ein eindeutiger Name für den Service. Dieser Name sollte immer gleich bleiben. Dadurch ist es anderen Services möglich, dauerhaft Informationen über diesen Service abzuspeichern. |
| service.ranking | Integer | Wenn es mehrere gleiche Services gibt, wird von dem Framework unter Zuhilfenahme dieses Ranking Wertes und wenn nötig der Service id ermittelt, welcher Service zurückgegeben wird. |
| service.vendor | String | Name des Herstellers, welcher den Service zur Verfügung stellt. |

**Tabelle 10: Vorgegebene Standard Attribute von Diensten.**

## 8.5 Verfügbare Implementierungen

Von den folgenden Firmen sind Implementierungen der OSGi Services Platform verfügbar.

| Firmenname | Weitere Informationen |
|---|---|
| GateSpace | http://www.gatespace.com/telematics/osgi.shtml |
| Sun Microsystems | http://www.sun.com/software/embeddedserver |
| IBM | http://www.embedded.oti.com/download/technologies/wesd.html |
| ProSyst | http://www.prosyst.com/ |

**Tabelle 11: OSGi Implementierungen**

Außerdem sind auch folgende Open Source Implementierungen erhältlich:

| Projektname | Weitere Informationen |
|---|---|
| Oscar | http://oscar-osgi.sourceforge.net |
| Service Tango<br>( wird nicht mehr gewartet!) | http://servicetango.sourceforge.net |

**Tabelle 12: Open Source OSGi Projekte**

## 8.6 Anmerkungen und Schwachpunkte

Da die bisherigen Implementierungen des OSGi Framework in JAVA erfolgten ist die Technik von JAVA abhängig. Dies bedeutet, dass das Framework nur auf Hardware ausgeführt werden kann, für die eine JVM verfügbar ist. Außerdem ergeben sich so auch die rechtlichen Einschränkungen denen die Nutzung von JAVA unterliegen.

Dieses Problem wäre lösbar, indem man eine frei verfügbare Technik nutzt und mit dieser die von der OSGi vorgegebenen Spezifikationen implementiert. Allerdings wäre dies mit einem sehr großen Aufwand verbunden, da es bis heute keine besser für die Implementierung der Vorgaben geeignete Technik als JAVA gibt.

# 9 Fazit

An dieser Stelle möchten wir noch einmal einen kurzen Überblick über die verschiedenen Architekturen zur Gerätevernetzung geben. Die nachfolgende Tabelle beschreibt die wichtigsten Voraussetzungen, Fähigkeiten und Einschränkungen der einzelnen Technologien:

| | | Jini | Java Message Service (JMS) | Java Media Framework (JMF) |
|---|---|---|---|---|
| **Discovery-Mechanismen** | | Multicast , Broadcast<br>Lookup Service | keine | Nicht relevant |
| **Dienst-Beschreibung** | | Typ & Attribute | keine | Nicht relevant |
| **Automatische Konfiguration (Netzwerk/Geräte/Dienste)** | | nein/ja/ja | nicht spezifiziert | Nicht relevant |
| **Dienst-Nutzung** | | Lokale Methodenaufrufe beim Dienst-Proxy; Kommunikation zwischen Proxy & Dienst über beliebige Protokolle | nicht spezifiziert | Nicht relevant |
| **Sicherheits-mechanismen** | Verschlüsselung | Nicht spezifiziert | nicht spezifiziert | Nicht spezifiziert |
| | Authentifizierung | ja (*principal control list)* | nicht spezifiziert | Nicht spezifiziert |
| | Autorisierung | ja (*access control list)* | nicht spezifiziert | Nicht spezifiziert |
| | Verifizierung signierten Programm-Codes | Nicht spezifiziert | nicht spezifiziert | Nicht relevant |
| **Unterstützung von AV-Streaming** | | nein | nein | ja |

| | | Jini | Java Message Service (JMS) | Java Media Framework (JMF) |
|---|---|---|---|---|
| **Code-Übertragung** | | Ja (Dienst-Proxy) | nein | nein |
| **Voraus-setzungen** | **HW-Architektur** | keine Festlegung | keine Festlegung | keine Festlegung |
| | **Betriebssystem** | keine Festlegung, es muss eine Java VM für die Hardware verfügbar sein. | keine Festlegung, es muss eine Java VM für die Hardware verfügbar sein. | keine Festlegung, es muss eine Java VM für die Hardware verfügbar sein. |
| | **Netzwerk** | protokollunabhängig | keine Festlegung | keine Festlegung |
| | **sonstige** | JVM | Java ab JDK 1.1 | Java ab JDK 1.1 |
| **IPv6-Unterstützung** | | nicht spezifiziert | nicht spezifiziert | nicht spezifiziert |
| **Skalierbarkeit** | | ja | nicht spezifiziert | nicht spezifiziert |
| **Legacy-Unterstützung** | | ja | nicht spezifiziert | nicht spezifiziert |
| **Zugrundeliegende Standards** | | Java, RMI, HTTP | keine Festlegung | HTTP, FTP, RTP,<br>AIFF, AU, AVI, GSM, MIDI, MP2, MP3, QT, RMF, WAV<br>AVI, MPEG-1, QT, H.261, H.263 |
| **Status/Version** | | Version 1.2 | Version 1.1 | Version 2.1.1 |

Fortsetzung der Tabelle:

| | | HAVi | OSGi | UPnP |
|---|---|---|---|---|
| **Discovery-Mechanismen** | | IEEE 1394-NW-Resets als Ereignisse an Systemdienste, föderierter Namensdienst (Registry) | Auf Hardwareebene von der jeweiligen benutzten Vernetzungstechnik abhängig.<br>Im OSGi Framework steht ein Dienst (service registry) zum Anmelden und Suchen von Dienstanbietern zur Verfügung. | Direkt über Multicast-Nachrichten: neue Dienstanbieter geben ihre Verfügbarkeit bekannt, neue Klienten fragen nach verfügbaren Dienstanbietern |
| **Dienst-Beschreibung** | | Nach IEEE 1212-Standard:<br>Typ, ID, Version, Anforderungen an Controller; **keine** freien Attribute | Definiert durch ein Service Interface (Java Interface Klasse). Dienste können bei ihrer Registrierung mit Attributen versehen werden. | Im XML-Format:<br>Typ, ID, Attribute, Funktionen, Statusvariablen |
| **Automatische Konfiguration (Netzwerk/Geräte/Dienste)** | | ja (durch IEEE 1394)/ja/ja | ja/ja/ja<br>(Device Access Specifikation) | ja/ja/ja |
| **Dienst-Nutzung** | | Nachrichten über IEEE 1394, RPC-Abstraktion | Methodenaufrufe | Durch SOAP-Nachrichten (XML-Format) über HTTP. |
| **Sicherheits-mechanismen** | Verschlüsselung | nein | Ja | nein |
| | Authentifizierung | Nur für Software-Elemente, durch Signatur | Ja | nein |
| | Autorisierung | Zugriffsrechte für Softwareelemente, System-SE und vertrauenswürdige SE haben Privilegien | Ja | nein |
| | Verifizierung signierten Programm-Codes | Ja, Prüfung bei Installation von Bytecode | Ja | nein |
| **Unterstützung von AV-Streaming** | | Ja, über IEEE 1394-Bus gemäß IEC 61883 | Ja, kann durch Integration dafür geeigneter Techniken ermöglicht werden. | Ja, Datenübertragung beim Streaming über beliebige Protokolle (auch nicht-IP) |

| | | HAVi | OSGi | UPnP |
|---|---|---|---|---|
| **Code-Übertragung** | | Ja (Java-Bytecode) | Ja | nein |
| **Voraus-setzungen** | **HW-Architektur** | keine Festlegung | keine Festlegung, es muss eine Java VM für die Hardware verfügbar sein. | keine Festlegung |
| | **Betriebssystem** | keine Festlegung | keine Festlegung, es muss eine Java VM für das Betriebssystem verfügbar sein. | keine Festlegung |
| | **Netzwerk** | IEEE 1394-Netzwerk | Nahezu jede Art von Netzwerk integrierbar. | IP-basiertes Netzwerk; DHCP-Server empfohlen |
| | **sonstige** | Java als bevorzugte Implementierungssprache | Java als bevorzugte Implementierungssprache | Devices müssen HTTP-Server implementieren,<br>alle Geräte müssen DHCP-Clients implementieren |
| **IPv6-Unterstützung** | | Nicht relevant | Möglich, aber noch nicht spezifiziert. | Ja, spezifiziert |
| **Skalierbarkeit** | | Limitierende Faktoren:<br>max. 64000 Geräte in IEEE1394-Netzwerk, jede FAV kommuniziert mit jeder anderen FAV im Netzwerk | Abhängig von der Implementierung und den eingesetzten Techniken. | Limitierende Faktoren:<br>Discovery-Mechanismus (sollten $<< 10^5$ Geräte sein);<br>max. $2^{16}$ Geräte wg. AutoIP-Adressierung |
| **Legacy-Unterstützung** | | Nur über Geräte mit Brückenfunktion (FAVs bzw. IAVs) | Muss von einer OSGi konformen Implementierung der Device Access Specifikation unterstützt werden. | nur über Brücken |
| **Zugrundeliegende Standards** | | IEEE 1394, IEC 61883, IEEE 1212 | Abhängig von den eingesetzten Techniken. | SOAP, XML, HTTP, AutoIP, DHCP, TCP, UDP |
| **Status/Version** | | 1.1 | Release 2.0 | 1.0 |

# 10 Referenzen


[UPnP01] *Universal Plug and Play Device Architecture*,

Microsoft Corporation,

http://www.upnp.org/resources/documents.asp

[UPnP02] *Understanding Universal Plug and Play - Whitepaper*,

Microsoft Corporation,
http://www.upnp.org/resources/whitepapers.asp

[UPnP03] *Universal Plug and Play AV Architecture*,

Microsoft Corporation,

http://www.upnp.org/resources/standards.asp

[UPnP04] *Multicast and Unicast UDP HTTP Messages*,

UPnP Forum Technical Committee,

http://www.upnp.org/resources/specifications.asp

[UPnP05] *UPnP AV Architecture, Media Server Device Template, Media Renderer Device Template*,

Microsoft Corporation
http://www.upnp.org/resources/standards.asp

[AutoIP] *Dynamic Configuration of IPv4 link-local addresses*,

IETF Internet draft,
http://www.upnp.org/resources/specifications.asp

[SSDP] *Simple Service Discovery Protocol/1.0*,

IETF Internet draft,

http://www.upnp.org/resources/specifications.asp

[GENA] *General Event Notification Architecture Base*,

IETF Internet draft,

http://www.upnp.org/resources/specifications.asp

[SOAP] *SOAP – Simple Object Access Protocol Version 1.2*

W3C Working Draft

http://www.w3.org/2000/xp/Group/

[HAVi01] *The HAVi Specification Version 1.1*,

HAVi Inc.,

http://www.havi.org/techinfo/index.html

[HAVi02] *HAVi, the A/V digital network revolution*, Whitepaper,

HAVi Inc.,
http://www.havi.org/techinfo/whitepaper.html

[HAVi03] *HAVi-Einführung*,

Technische Universität Wien,
http://www.ict.tuwien.ac.at/ieee1394/havi/havi_intro.htm

[HAVi04] *Vididlogic HAVi documentation*,

Vividlogic,

http://www.vividlogic.com/hbx/index.html

[IEEE1394] *IEEE 1394 High Performance Serial Bus*, Vortrag

Technische Universität Wien,

http://www.ict.tuwien.ac.at/ieee1394/lehre/lehre.html

[OSGi01] *Open Services Gateway Initiative*

http://www.osgi.org

[OSGi02] *Lightweight Directory Access Protocol* (LDAP)

http://directory.google.com/Top/Computers/Software/Internet/Servers/Directory/LDAP/

[OSGi03] *The Java 2 Platform API Specification*

Standard Edition, Version 1.3, Sun Microsystems

http://java.sun.com/j2se/1.3

[OSGi04] *The Java Language Specification*

Second Edition, Sun Microsystems, 2000

http://java.sun.com/docs/books/jls/index.html

[OSGi05] *The Java Security Architecture for JDK 1.2*

Version 1.0, Sun Microsystems, October 1998

http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.fm.html

[OSGi06] *HTTP 1.0 Specification RFC-1945*

http://www.ietf.org/rfc/rfc1945.txt, May 1996

[OSGi07] *HTTP 1.1 Specification RFC-2616*

http://www.ietf.org/rfc/rfc2616.txt, June 1999

[OSGi08] *Java Servlet Technology*

http://java.sun.com/products/servlet/index.html

[Jini01] *Jini-Überblick*,

Sun Microsystems Deutschland,

http://www.sun.de/Produkte/Software/Systeme_und_Plattformen/Jini/

[Jini02] *Jini(TM) Technology Core Platform Specification*,

Sun Microsystems,

http://wwws.sun.com/software/jini/specs

[[JMS01] *JavaTM Message Service Specification*,

Sun Microsystems,

http://java.sun.com/products/jms/docs.html

[[JMF01] *JMF API Documentation*,

Sun Microsystems,

http://java.sun.com/products/java-media/jmf/2.1.1/apidocs/