Fehlertolerante Integration limitierter Geräte in SOA unter Verwendung von Geräte-Proxies

von der
Fakultät für Elektrotechnik und Informationstechnik
der Technischen Universität Dortmund
genehmigte Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften

von

Peter Schramm

Dortmund, Januar 2008

Hauptreferent:	Prof. Dr.-Ing. Uwe Schwiegelshohn
Korreferent:	Prof. Dr. Jakob Rehof
Tag der Einreichung:	29. Mai 2007
Tag der Prüfung:	16. Januar 2008

Ich danke allen, die mich unterstützt haben...

Inhaltsverzeichnis

1. Einleitung

Informationstechnologie hält zunehmend Einzug in den menschlichen Alltag. Dieser Trend wird in der Fachwelt *Ubiquitous-Computing*[1] genannt. Geprägt wurde der Begriff von Mark Weiser in seinem Aufsatz „The Computer of the 21st Century“ [105]. Auslöser des Trends sind die großen Fortschritte der Informationstechnik in den vergangenen Jahren: Nach dem *Mooreschen Gesetz* verdoppelt sich die Leistungsfähigkeit integrierter Schaltungen alle zwei Jahre. Damit werden informationstechnische Geräte immer kleiner und preiswerter. Zusätzlich nehmen die Kommunikationsfähigkeiten von Geräten permanent zu, wodurch vormals voneinander isolierte Geräte in verteilten Systemen aufgehen. Eine wesentliche Eigenschaft von Ubiquitous-Computing-Systemen ist, dass die Informationstechnologie für den Menschen unsichtbar wird.

Typische Ubiquitous-Computing-Szenarien spielen im intelligenten Heim, im intelligenten Büro oder auch in zunehmendem Maße in industriellen Umgebungen. Dabei werden untereinander vernetzte, intelligente Geräte verwendet, um den Menschen in seiner häuslichen Umgebung oder bei seiner Arbeit zu unterstützen oder Arbeitsabläufe dynamischer und flexibler zu gestalten. Anwendungsfälle erstrecken sich von der Umsetzung reiner Komfortfunktionen wie automatische Beleuchtungssteuerungen oder der Steuerung von Multimedia-Geräten über ökonomisch relevante Funktionen wie die intelligente Regelung von Klimasystemen.

Die Kopplung von physikalischer und informationsstrukturierter Umgebung wird durch *Sensoren* und *Aktuatoren* erreicht. Sensoren werden verwendet, um den Zustand der physikalischen Umgebung zu ermitteln und an die informationsverarbeitenden Komponenten weiterzuleiten. Einfache Beispiele für Sensoren aus dem Bereich des intelligenten Heims sind die Messung der Raumtemperatur, Helligkeit oder Luftfeuchtigkeit. Darüber hinaus sind komplexere Sensoren wie Präsenzmelder oder auch Sensoren zur Lokalisierung von Menschen und Gegenständen zu nennen. Aktuatoren (Stellglieder) haben die umgekehrte Funktion von Sensoren: Sie werden von den informationsverarbeitenden Komponenten angesprochen, um Einfluss auf die physikalische Umgebung zu nehmen. Einfache Beispiele für Aktuatoren sind Fenster- und Türöffner, Heizungsregulatoren oder Beleuchtungssteuerungen.

Aufgrund der engen Verzahnung von physikalischer und informationsstrukturierter Umgebung sind Ubiquitous-Computing-Systeme in der Regel komplex. Daneben zählen Heterogenität und Dynamik zu den typischen Eigenschaften. Die Heterogenität rührt von der Verwendung unterschiedlicher Arten von Geräten, deren Schnittstellen mitunter verschiedenen Standards entsprechen oder gar proprietär sind. Dynamisch sind die Umgebungen,

[1] Ubiquitous (engl.): allgegenwärtig. Oft wird auch das industriell geprägte Synonym *Pervasive Computing* verwendet, von pervasive (engl): durchdringend.

weil sie mitunter mobile Geräte enthalten, die dem System zu beliebiger Zeit beitreten und auch wieder aus ihm entfernt werden können.

Die genannten Eigenschaften der Systeme können mit klassischen „starren“ Netzwerkstrukturen quasi nicht beherrscht werden. Ein aus dem Umfeld verteilter Systeme bekanntes Hilfsmittel, um mit diesen Eigenschaften umzugehen, ist die *Middleware*. Middleware ist eine Adaptionsschicht, die Komplexität, Heterogenität und Dynamik der Systeme beherrscht und für die Anwendungen transparent macht [30]. Damit ist die Middleware das Schlüsselelement für den Aufbau *Service-orientierter Architekturen (SOA)* [14, 31]. Eine zunehmend in den Vordergrund rückende Spielart von SOA sind *Device-level SOA* [48, 9]. Hierbei werden SOA im Sinne des Ubiquitous-Computing durch alltägliche Geräte gebildet. In einer Device-level SOA bieten Geräte Dienste an und können die Dienste anderer Geräte nutzen. Dabei bleibt die eigentliche Abwicklung der Kommunikation für die Anwendungen transparent. Ferner können sich Geräte für *Ereignisbenachrichtigungen (Events)* bei anderen Geräten registrieren. Über *Discovery-Verfahren* können Geräte und ihre Dienste automatisch im Netzwerk gefunden werden. Die sich aus dem Einsatz von SOA ergebenden Vorteile sind ein geringerer Implementierungsaufwand, erhöhte Robustheit der Systeme sowie erleichterte Wartbarkeit der Systeme im Vergleich zu klassischen Systemstrukturen [9].

Middleware-Unterstützung limitierter Geräte

Aufgrund der Vorteile von SOAs bietet es sich an, auch sehr kleine Geräte wie zum Beispiel Sensoren mit einer Middleware-Unterstützung auszustatten und in SOAs aufzunehmen. Dies stellt nach dem heutigen Stand der Technik kein Problem dar, solange der Preis der Geräte keine Rolle spielt. Wird vorausgesetzt, dass kleine Geräte möglichst preiswert sein müssen, so ist anzunehmen, dass sie nur über entsprechend geringe Ressourcen wie Rechenleistung, Arbeitsspeicher, Kommunikationsbandbreite und Energie verfügen. Leider stellen typische Middleware-Systeme relativ hohe Anforderungen an kleine Geräte. So setzen viele Systeme wie zum Beispiel *Universal Plug and Play (UPnP)*[2] die Fähigkeit voraus, XML-Nachrichten parsen beziehungsweise dynamisch erzeugen zu können oder, wie im Falle der *Jini-Technologie*[3], eine *Java Virtual Machine (JVM)* auszuführen. Aufgrund dieser Diskrepanz sind viele Anwendungsfälle denkbar (wie bei Verwendung kleiner, batteriebetriebener Sensoren...), in denen kein geeigneter Kompromiss zwischen Leistung und Preis von kleinen Geräten gefunden werden kann, beziehungsweise in denen bei einem festgelegten Preis die Leistungsfähigkeit der Geräte nicht ausreicht, die Anforderungen einer Middleware zu erfüllen.

Wie eingangs erwähnt steigt nach dem Mooreschen Gesetz die Leistungsfähigkeit von Geräten bei gleich bleibendem Preis stetig an. Daher ist durchaus zu erwarten, dass die Leistungsprobleme, die heute für eine bestimmte Klasse von Geräten gelten, in Zukunft obsolet werden. Es gibt allerdings zwei weitere Faktoren, die bei dieser Fragestellung berücksichtigt werden müssen: Zum einen kann erwartet werden, dass nicht nur die Leistungsfähigkeit von kleinen Geräten steigt, sondern auch die Ansprüche steigen, die an die Geräte gestellt werden. Zum anderen ist die Leistungsfähigkeit, die von einem Gerät er-

[2] http://www.upnp.org
[3] http://www.sun.com/jini

wartet werden kann, nicht nur eine Frage des Preises, sondern wird auch von der Größe (das heißt von den physikalischen Abmessungen) des Gerätes beeinflusst. Aufgrund dieser Überlegungen muss angenommen werden, dass es stets Situationen gibt, in denen Anforderungen an kleine Geräte gestellt werden, die sie nicht auf eigenständige Weise erfüllen können.

Nutzung akkumulierter Ressourcen

Geräte in Ubiquitous-Computing-Umgebungen lassen sich in zwei Klassen einteilen, wie in Abbildung 1.1 dargestellt. Die erste Klasse (Innenbereich der Darstellung) enthält Geräte, die bezüglich ihrer Ressourcen als unbeschränkt eingestuft werden können, da sie über entsprechend große Mengen an Rechenleistung, Arbeits- und Festspeicher, Bandbreite sowie Energie verfügen. Die zweite Klasse (äußerer Ring) enthält kleine, leistungsarme Geräte, die zwar über Kommunikationsschnittstellen verfügen, jedoch aus Gründen ihrer geringen Leistungsfähigkeit keinen von Geräten der ersten Klasse angewendeten Middleware-Standard unterstützen.

Abb. 1.1.: Geräteklassen in Ubiquitous-Computing-Umgebungen (Klasse 1: innerer Kreis, Klasse 2: äußerer Ring).

Der Ansatz, der im Rahmen der vorliegenden Arbeit verfolgt wurde, um das Problem des möglichen Missverhältnisses zwischen erforderlicher und verfügbarer Geräteleistung zwischen den zwei genannten Geräteklassen zu lösen beziehungsweise zu umgehen ist, die kleinen Geräte zu befähigen, akkumulierte Ressourcen im verteilten System für ihre Zwecke zu nutzen. So kann zum Beispiel die Rechenleistung oder der Arbeitsspeicher anderer Geräte, die in Ubiquitous-Computing-Umgebungen ohnehin vorhanden sind, genutzt werden, um ein limitiertes Gerät bei der Erfüllung seiner Aufgaben zu entlasten. Hierzu wurde das Konzept der *Geräte-Proxies* angewendet.

Die Vorteile, die sich durch Anwendung des Proxy-Konzeptes ergeben, werden durch eine aufwendige Lebenszyklusverwaltung sowie einen hohen Entwicklungsaufwand der Proxies erkauft. Außerdem wird durch die Anwendung des Proxy-Konzeptes die Robustheit der Dienste im Vergleich zu Standalone-Geräten reduziert. Das liegt daran, dass Gerät und Proxy ein Seriensystem bilden. Der Dienst des Gerätes fällt nicht nur aus, wenn das Gerät selbst ausfällt, sondern auch, wenn der Proxy des Gerätes ausfällt.

Sindrion

Eine Java-basierte Implementierung des im Rahmen dieser Arbeit entwickelten Systems liegt unter dem Namen *Sindrion Application Manager* vor und ist im Auftrag und mit Unterstützung von Infineon Technologies entstanden [13, 35]. Das Sindrion-System bindet batteriebetriebene, kabellose Sensoren (sog. *Sindrion-Transceiver*, siehe Anhang A.2) über Proxies in UPnP-Netzwerke ein. Der Sindrion Application Manager sorgt dabei für die Lebenszyklusverwaltung der Proxies bzw. den Bezug des Proxy-Codes. Ferner spezifiziert Sindrion als Erweiterung zum UPnP-Standard Geräte, die ihre eigenen Steuerapplikationen in Form von UPnP-basierten *Specific Control Points (SCPs)* oder Java-basierten *Sindrion-Applets* bereit stellen. Sindrion wird heute im Rahmen verschiedener Projekte wie z. B. den EU-Projekten *Promise*[4] oder *CoBIs*[5] eingesetzt.

1.1. Einordnung in das thematische Umfeld

Die vorliegende Arbeit reiht sich in das Umfeld des *Ubiquitous Computing* [105, 16] ein. Die kleinsten Komponenten dieser Systeme sind o. g. Sensoren bzw. Aktuatoren, aus denen *Sensornetzwerke* aufgebaut werden [25, 94]. Besondere Anforderungen an Ubiquitous Computing Umgebungen [26, 76] wie der Umgang mit Dynamik und Heterogenität werden mit dem Einsatz von *Middlewaresystemen* [30, 60] adressiert. Mit Hilfe von Middleware werden *Service-Oriented Architectures (SOA)* aufgebaut [9]. Wird, wie im Rahmen der vorliegenden Arbeit eine SOA aus relativ kleinen Geräten aufgebaut, so spricht man von *Device-Level SOA* [48].

Relevante Arbeiten und Projekte lassen sich grob in die folgenden Bereiche eingliedern: Der erste Bereich umfasst Kommunikationsinfrastrukturen bzw. Middlewareproto-

[4] http://www.pabadis-promise.org
[5] http://www.cobis-online.de

kolle. Verbreitete Middlewareprotokolle sind *Jini* [38], *JXTA* [66], *CORBA* [102] und *UPnP* [64, 51, 82]. Ansätze wie *Salutation* (bzw. *HP JetSend*) [12] haben heute praktisch keine Bedeutung mehr. Allgemein ist ein Trend hin zu einer Anlehnung von Middlewareprotokollen an Internet-Technologien zu beobachten. Dieser Trend wird bereits durch UPnP verfolgt und durch Technologien wie *DPWS*[6] [48] fortgesetzt.

Der zweite Bereich umfasst *Geräte-* bzw. *Protokolladapter*. Häufig werden *Gateways* eingesetzt, um unterschiedliche (Middleware-)Protokolle aufeinander abzubilden [1, 72, 106, 39]. Viele Ansätze sind speziell darauf ausgelegt, ressourcenarme Geräte in Middlewarebasierte Umgebungen zu integrieren [55, 17, 45, 81, 89, 78, 13, 35, 8]. Oft wird auch *Codegenerierung* eingesetzt, um Code an bestimmte Plattformeigenschaften oder Anwendungsfälle anzupassen [57, 59, 23, 20, 65, 97, 43]. Viele verwandte Arbeiten verwenden Proxies, um limitierte Geräte in verteilten Systemen zu vertreten [61, 75, 43]. Die limitierten Geräte sind allerdings oft ausschließlich Clients, die über einen Proxy in die Lage versetzt werden, einen bestimmten Dienst in Anspruch zu nehmen. Das in dieser Arbeit angewendete Proxy-Konzept kommt auch in ähnlicher Form in der *Jini Surrogate Architecture* [89] zum Einsatz. Im Bereich von Sensornetzwerken sind Arbeiten entstanden, die in die vorliegende Arbeit eingeflossen sind [78].

Der dritte Bereich umfasst Verwaltungsstrukturen (Frameworks), die eine komponentenbasierte Verwaltung verteilter Systeme erlauben. Ein verbreiteter Standard ist hier *OSGi* [2, 34, 62, 44] – teils mit kommerziellen Implementierungen [101]. Viele aktuelle Projekte sind im Bereich *kontextsensitiver Systeme* [36, 22] oder im Bereich *Location-Awareness* [40] angesiedelt. Andere Ansätze wie das Web-basierte *CoolTown* (HP) [52] oder *Gaia* (nutzt CORBA) [21, 70] basieren nicht auf OSGi, sondern auf proprietären Architekturen. Viele Systeme sind speziell auf das Umfeld des intelligenten Heims wie im Falle von *Easy Living* [19] oder des intelligenten Konferenzraumes (ICrafter [69]) ausgelegt.

Der Status einzelner Systemkomponenten wird durch *Monitoring-Verfahren* ermittelt. Hier kommt erschwerend hinzu, dass die genutzten Systeme oft auf *Managed Code* wie *Java* oder *.Net* basieren und daher zum Teil besondere Verfahren zur Ermittlung des Zustandes erfordern [95]. Häufig spielen auch Echtzeitanforderungen eine wichtige Rolle. Die Erfüllung dieser Anforderungen wird allerdings wie schon im Umfeld des Monitoring durch die auf Managed Code basierenden Softwareumgebungen erschwert [15]. Systeme im hier betrachteten Umfeld sind daher i. d. R. nicht in vollem Umfang echtzeitfähig, sondern binden allenfalls echtzeitfähige Sub-Komponenten ein [92]. Einige Projekte adressieren allerdings die Integration von Echtzeitfähigkeit in Middlewarearchitekturen [56].

Ferner wird in vielen Systemen das Modell von *mobilen Agenten* angewendet, um Anforderungen wie Dynamik und Heterogenität zu entsprechen [33, 10, 7]. Neben der Migration einzelner Komponenten oder Applikationen ist auch eine Migration virtueller Maschinen denkbar [24]. Beispiele sind *Java Party* [68] und *Mole* [84]. Ein spezieller Fall mobiler Applikationen sind *Follow-Me Applications* [83, 93, 61, 73, 74], die einem Benutzer durch ein System folgen.

[6] Device Profile for Web Services

Das im Rahmen der vorliegenden Arbeit vorgestellte System verbindet die Bereiche der Sensornetzwerke mit dem Bereich der Middleware-basierten Device-Level SOA. Um diese Bereiche zu verbinden, wird das o. g. Konzept der Geräte- bzw. Protokolladapter angewendet. Verwaltungsstrukturen sowie Monitoring-Verfahren werden eingesetzt, um die Vorgänge im System regeln zu können. Das Konzept der Mobilen Agenten ist schließlich für die im Rahmen der vorliegenden Arbeit eingesetzte *Migration* von Proxies relevant, die als Mittel der Vorbeugung von Störungen dient. Fehlertoleranzverfahren [47] kommen schließlich zum Einsatz, um mit Ausfällen angemessen umgehen zu können.

1.2. Struktur der vorliegenden Arbeit

Die Struktur der vorliegenden Arbeit orientiert sich an der Vorgehensweise der Entwicklung, wie in Abbildung 1.2 dargestellt. Im ersten Teil der Arbeit wird eine Infrastruktur für Proxy-basierte Systeme konzipiert und ihre Realisierung beschrieben. Im zweiten Teil wird die Infrastruktur um dedizierte Fehlertoleranzmerkmale erweitert, die anschließend evaluiert werden. Dazu kommen zum einen Messungen am implementierten System und zum anderen Simulationen zum Einsatz. Das der Simulation zugrunde liegende Modell wurde dabei mit Hilfe der Messungen am implementierten System geeignet parametriert.

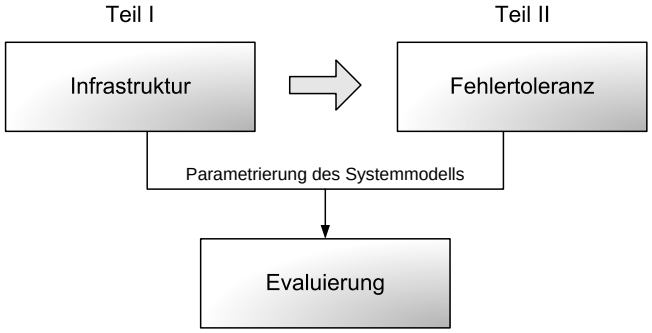


Abb. 1.2.: Vorgehensweise und Struktur.

Teil I.

Infrastruktur

2. Die Middleware Universal Plug and Play

Die im Rahmen der vorliegenden Arbeit vorgenommenen Untersuchungen lassen sich im Grunde auf alle gängigen Middleware-Systeme beziehen. Um jedoch einen möglichst konkreten Bezug zu einer existierenden Technologie herzustellen, wurde die Middlware *Universal Plug and Play (UPnP)* als Referenz ausgewählt – zumal sie im Rahmen der Implementierung des Sindrion-Systems (siehe Kapitel 1) zum Einsatz kam. Um eine Verständnisgrundlage zu schaffen, wird die grundlegende Architektur von UPnP im Folgenden in knapper Form vorgestellt. Für eine genauere Betrachtung sei auf weiterführende Literatur verwiesen [49, 98].

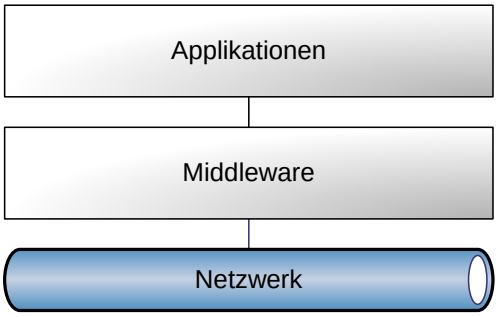


Abb. 2.1.: Middleware als Verbindungsschicht.

Allgemein wird eine Middleware, wie in Abbildung 2.1 dargestellt, als Verbindungsschicht zwischen Applikationen und dem Netzwerk definiert [30]. Ihre zentrale Aufgabe ist der Umgang mit systeminhärenten Eigenschaften wie *Heterogenität* und *Dynamik*. Somit kommt Middleware in beinahe sämtlichen IT-Bereichen zum Einsatz. Auch im Umfeld von Ubiquitous-Computing-Umgebungen spielt Middleware eine zentrale Rolle. In diesem Umfeld hat eine Middleware folgende konkrete Aufgaben:

- **Device- und Service-Discovery**
  Die Middleware muss in der Lage sein, Geräte und Dienste zur Laufzeit bei Bedarf zu finden. Dies kann auf zwei Arten erfolgen: Entweder wird ein Lookup-Dienst verwendet (Jini-Technologie), bei dem verfügbare Geräte sich und ihre Dienste anmelden. Alternativ können Advertisement-Nachrichten verschickt werden, über die Geräte sich anderen Geräten bekannt machen.
- **Übermittlung von Steuerbefehlen**
  Die von Geräten bereitgestellten Dienste enthalten Aktionen und Zustandsvariablen.

Es ist Aufgabe der Middleware, Steuerbefehle an ein Gerät zu übermitteln. Das Gerät führt daraufhin Aktionen aus und der Status der Zustandsvariablen verändert sich.

- **Verwaltung von Ereignissen**
  Wenn der Wert einer Zustandsvariablen sich verändert, sendet das Gerät ein Ereignis-Signal an beliebig viele Interessenten (Multicast). Diese Interessenten müssen sich zuvor für bestimmte Ereignisse oder Ereignistypen registrieren.

Die genannten Aufgaben werden von der Middlware Universal Plug and Play erfüllt. Sie dient der herstellerübergreifenden Ansteuerung von Geräten über ein IP-basierendes Netzwerk. Die Technologie basiert auf einer Reihe von standardisierten Netzwerkprotokollen und Datenformaten. Ursprünglich von der Firma Microsoft eingeführt, spezifiziert heute das UPnP-Forum[1] den UPnP-Standard und zertifiziert Geräte, die dem Standard entsprechen.

Allgemein werden *UPnP Devices* und *UPnP Control Points* unterschieden. Das Device ist ein Dienstanbieter, der Control Point ist Dienstnehmer. Ein physikalisches Gerät kann sowohl Device als auch Control Point sein. Ein Device bietet, wie in Abbildung 2.2 veranschaulicht, einen oder mehrere *Services* an, die wiederum eine oder auch mehrere *Actions* enthalten. Eine Action entspricht einer Kontrollnachricht, die dem Device gesendet werden kann, um eine bestimmte Aktion durchzuführen. Neben den Actions enthält ein Service auch Zustandsvariablen (*State Variables*), die durch die Actions beeinflusst werden können oder Ereignisbenachrichtigungen (*Events*) an Control Points senden können, sobald ihr Status sich ändert. Darüber hinaus kann ein Device beliebig viele *Embedded Devices* enthalten, die ihrerseits wieder Services anbieten. Zur besseren Unterscheidung wird das „äußere“ Device auch *Root Device* genannt.

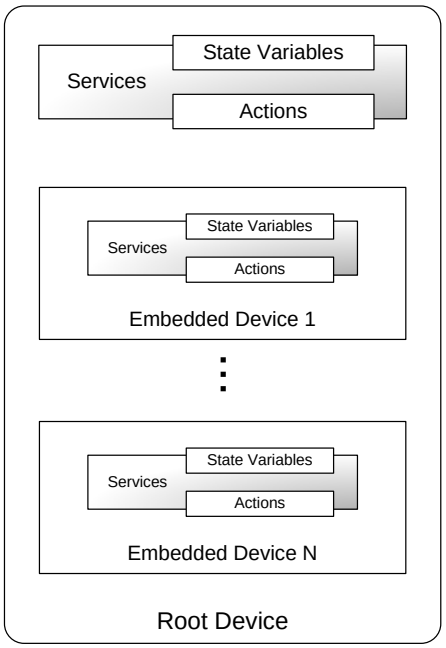


Abb. 2.2.: Struktur eines UPnP Device.

Abbildung 2.3 zeigt den UPnP-Protokoll-Stack, wie er in der *UPnP Device Architecture* [98] definiert ist. Wie der Abbildung zu entnehmen ist, basiert UPnP auf TCP/IP und UDP/IP. Als eigentliches Transportprotokoll wird HTTP verwendet, wobei zwei spezielle Varianten von HTTP für die UDP-basierte Übertragung zum Einsatz kommen: HTTPU[2] und HTTPMU[3]. Die eigentlichen UPnP-Protokolle sind SSDP[4], über das Discovery-Funktionen bereitgestellt werden, GENA[5] für das Ver-

[1] http://www.upnp.org
[2] Hypertext Transfer Protocol UDP
[3] Hypertext Transfer Protocol Multicast UDP
[4] Simple Service Discovery Protocol
[5] Generic Event Notification Architecture

senden und den Empfang von Ereignisbenachrichtigungen (Events), sowie SOAP[6] für den Aufruf von Actions. Die drei oberen Schichten des Protokoll-Stacks definieren die allgemeine Architektur eines UPnP-konformen Gerätes (UPnP Architecture Defined), welche Schnittstellen von dem Gerät bereitgestellt werden (UPnP Forum defined) und schließlich, welche Hersteller-spezifischen Erweiterungen ggf. im Gerät enthalten sind (UPnP Vendor defined).

UPnP-Stacks sind für eine große Anzahl an Plattformen verfügbar. Als Beispiele seien der UPnP-Stack *Cyber Garage*[7] (C++ und Java), die *Windows Mobile Server Components* für Pocket PC (C++), sowie die Embedded UPnP Stacks von *Atinav*[8] oder *EBSnet*[9] zu nennen. Im Rahmen der vorliegenden Arbeit wurde ein Java-basierter Stack von Infineon Technologies (Infineon UPnP/1.0 Protocol Stack 1.21) eingesetzt.

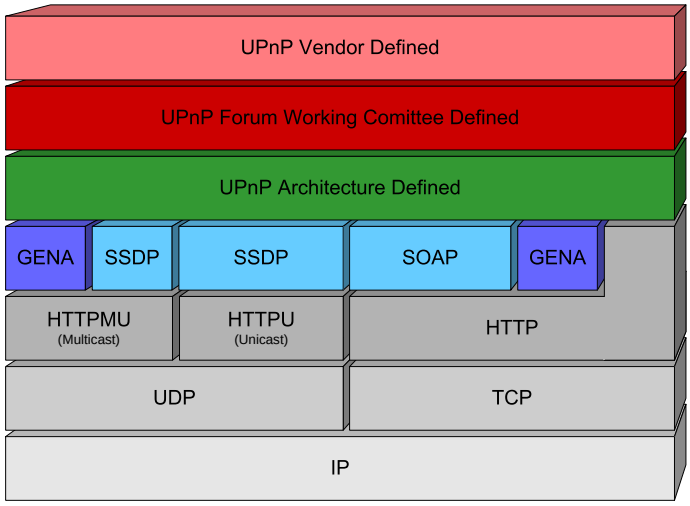


Abb. 2.3.: Der UPnP Protokoll-Stack.

Der Ablauf der UPnP-Kommunikation wird allgemein in 5 Phasen eingeteilt (siehe Abbildung 2.4). Phasen 0-2 finden stets in angegebener Reihenfolge statt, während für die Phasen 3-5 keine feste Reihenfolge vorgesehen ist [98]:

0. **Addressing**
   Da UPnP auf IP-Netzwerken basiert, muss dem Gerät im ersten Schritt eine IP-Adresse zugeordnet werden. Dies kann nach dem UPnP-Standard einerseits via DHCP erfolgen oder, sofern kein DHCP-Server verfügbar ist, via AUTO-IP. Da die Zuordnung der IP-Adresse eine elementare Grundvoraussetzung für die folgenden Schritte ist, wird dieser Schritt oft als „Schritt 0“ bezeichnet.

[6] Simple Object Access Protocol
[7] http://www.cybergarage.org
[8] http://www.atinav.com
[9] http://www.ebsnetinc.com

1. **Discovery**
   Sobald ein Gerät über eine IP-Adresse verfügt, muss es seine Existenz im Netzwerk bekannt machen. Dies erfolgt durch das Versenden von Multicast-Nachrichten auf Basis des SSDP-Protokolls. Dabei enthält die *Discovery Message* nur die wichtigsten Angaben über das Gerät und seine Dienste, wie z. B. den Gerätenamen, Gerätetyp und eine URL zur genauen Beschreibung des Gerätes. Umgekehrt können Control Points auch aktiv nach UPnP-Geräten im Netzwerk suchen.

2. **Description**
   Sobald ein Control Point ein Gerät im Netzwerk entdeckt hat, lädt er Beschreibungsdateien von dem Gerät herunter und wertet sie aus. Diese XML-formatierten *Device- und Service Descriptions* beinhalten neben allgemeinen Geräteinformationen Angaben über die von dem Gerät bereitgestellten Dienste sowie eine genaue Beschreibung ihrer Schnittstellen. Beispiele für UPnP Descriptions befinden sich in Anhang B.

3. **Control**
   Anhand der Informationen, die der Control Point der Beschreibungsdateien des Gerätes entnommen hat, kann er nun SOAP-Mitteilungen an das Gerät schicken, um es zu steuern.

4. **Eventing**
   Um den Zustand von Zustandsvariablen nicht über Polling-Verfahren abfragen zu müssen, können Control Points sich für Ereignisbenachrichtigungen registrieren. Somit werden sie bei jeder Änderung einer Zustandsvariablen automatisch über das GENA-Protokoll benachrichtigt.

5. **Presentation**
   Neben der Möglichkeit zur Steuerung über das SOAP-Protokoll und der GENA-Ereignisbenachrichtigung bietet jedes UPnP-Gerät eine eigene Web-Seite an, auf die über einen herkömmlichen Web-Browser zugegriffen werden kann. Neben allgemeinen Informationen über das Gerät kann die sog. *Presentation Page* auch eine Benutzeroberfläche zur direkten Steuerung bzw. Konfiguration des Gerätes enthalten.

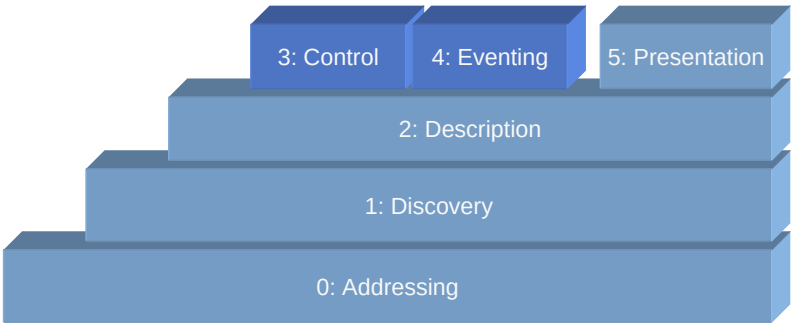


Abb. 2.4.: Phasen der UPnP-Kommunikation.

3. Konzeptionelle Überlegungen zur Infrastruktur

Im Folgenden werden einige grundlegende Überlegungen, die zum einen die Analyse des Problems, limitierte Geräte in Middleware-basierte Netzwerkgemeinschaften zu integrieren und zum anderen mögliche Lösungsansätze betreffen, diskutiert. Die in der vorliegenden Arbeit präferierte Lösung ist schließlich die Unterstützung limitierter Geräte über Geräte-Proxies. Während in diesem Kapitel lediglich grundlegende Konzepte diskutiert werden, erfolgt die Beschreibung einer Lebenszyklusverwaltung für Geräte-Proxies in Kapitel 4.

3.1. Merkmale limitierter Geräte

Allgemein zeichnet sich ein limitiertes Gerät durch sein Unvermögen aus, existierende Anforderungen aufgrund beschränkter Ressourcen nicht bzw. nicht zufriedenstellend zu erfüllen. Im Rahmen der vorliegenden Arbeit sprechen wir dann von limitierten Geräten, wenn sie aufgrund ihrer Beschränkungen Middleware-Protokolle, die zur Teilnahme an der Netzwerkgemeinschaft erforderlich sind, nicht oder nur unzureichend unterstützen. Typische Beispiele für limitierte Geräte sind *Sensoren* und *Aktuatoren*. Typische Kenngrößen limitierter Geräte sind:

- **Geringe Rechenleistung**
  Die Hardware-Plattform basiert auf einem 8-Bit oder 16-Bit Mikrocontroller.
- **Geringe Speicherkapazität**
  Es stehen ca. 64 kB ROM / 16 kB RAM oder weniger zur Verfügung.
- **Geringe Energiereserven**
  Es steht i. d. R. keine permanente Energieversorgung zur Verfügung.
- **Eingeschränkte Konnektivität**
  Das Gerät verfügt über zumeist kabellose Konnektivität, selten werden kabelgebundene Verbindungen verwendet. Brutto-Datenraten liegen im Bereich von 50 kbit/s.

Die im Rahmen der vorliegenden Arbeit betrachtete Referenzimplementierung eines limitierten Gerätes ist der *Sindrion-Transceiver*, der im Rahmen des Sindrion-Projektes parallel zur vorliegenden Arbeit entwickelt wurde – für Evaluierungen allerdings nur in Form eines Emulators zur Verfügung stand. Nähere Informationen zum Sindrion-Transceiver befinden sich in Anhang A.2.

3.2. Problemanalyse

Es gilt zu klären, warum die Unterstützung einer Middleware für ein limitiertes Gerät wie den Sindrion Transceiver nicht oder nur eingeschränkt möglich ist: Die Fähigkeit einer Middleware, mit System-inhärenten Merkmalen wie *Heterogenität* und *Dynamik* umzugehen, wird i. d. R. durch den Einsatz plattformunabhängiger und damit zumeist komplexer Protokolle erkauft. So basiert z. B. die Jini-Technologie wesentlich auf *Java Remote Method Invocation (RMI)*, während UPnP XML-basierte Protokolle wie *SOAP* verwendet (siehe Kapitel 2). Um ein vollwertiges UPnP Device bereit stellen zu können, müsste ein limitiertes Gerät in der Lage sein, XML-basierte SOAP-Actions auszuwerten. Listing 3.1 zeigt beispielhaft die SOAP-Action *SwitchPower* eines *UPnP Binary Light* [100]. Wie dem Beispiel zu entnehmen ist, erfordert selbst ein vergleichsweise simpler Vorgang, wie das Setzen eines neuen Lichtwertes (Zeilen 7 bis 9), die Verarbeitung von XML-Nachrichten.

Lst. 3.1: Aufbau einer SOAP-Action.

```
1 POST /SwitchPower_1_control HTTP/1.1 HOST: 192.168.1.4:1040
  CONTENT-TYPE: text/xml; charset="utf-8" CONTENT-LENGTH: 319
3 SOAPACTION: "urn:schemas-upnp-org:service:SwitchPower:1#SetTarget"

5 <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
      s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <s:Body>
7     <u:SetTarget xmlns:u="urn:schemas-upnp-org:service:SwitchPower:1">
        <newTargetValue xmlns="urn:schemas-upnp-org:service:SwitchPower:1">
9         1
        </newTargetValue>
11    </u:SetTarget>
    </s:Body>
13 </s:Envelope>
```

Als UPnP Control Point müsste ein limitiertes Gerät in der Lage sein, XML-basierte Device bzw. Service Descriptions zu verarbeiten wie in Anhang B dargestellt. Alles in allem muss ein vollwertiges UPnP-Gerät (Device oder Control Point) folgende Protokolle unterstützen:

- Ein Netzzugangsprotokoll (Ethernet, WLAN etc.),
- das Internetprotokoll (IP),
- DHCP und Auto IP (zum Beziehen einer IP-Adresse),
- TCP und UDP als Transportprotokolle,
- HTTP (HTTPU/HTTPMU), sowie
- SOAP, GENA und SSDP als Anwendungsprotokolle.

Neben den entsprechenden Protokoll-Stacks werden folgende zusätzliche Komponenten benötigt:

- ein (einfacher) Webserver,

- ein XML Parser.

Embedded UPnP-Stacks wie von *Atinav* oder *EBSnet* (siehe Kapitel 2) belegen allein ca. 80 - 120 kB ROM [53], während embedded TCP/IP Stacks 10 - 40 kB ROM (uIP[1] bzw. IwIP[2]) belegen. Für einen einzelnen SOAP-Service fallen allein 65 kB RAM an [11]. Insgesamt übersteigt dieser Speicherbedarf die Kapazitäten eines limitierten Gerätes wie dem Sindrion Transceiver bei weitem. Natürlich trifft diese Problematik auch – und mitunter in höherem Maße – auf andere Middlewaretechnologien als UPnP zu. So würde z. B. eine Unterstützung der Jini-Technologie eine vollwertige *Java Virtual Machine (JVM)* erfordern [38]. Die JVM der *Java Standard Edition* belegt z. B. bei Ausführung eines Programms, das eine einfache Schleife durchläuft, bereits 7 MB RAM[3]. Für die virtuelle Maschine der Java Micro Edition, Connected Device Configuration sind immer noch mindestens 2 MB RAM erforderlich [71].

Zusammenfassend kann gesagt werden, dass limitierte Geräte die genannten Anforderungen nicht oder nur unzureichend erfüllen können, da sie über zu geringe Ressourcen wie Rechenleistung oder Arbeitsspeicher verfügen.

3.3. Verwendung von Geräte-Proxies

Um das im vorherigen Abschnitt betrachtete Problem zu lösen – also limitierte Geräte trotz ihrer Beschränkungen in Middleware-basierte Netzwerkgemeinschaften aufzunehmen – können folgende grundlegende Ansätze verfolgt werden:

- **Optimierte Implementierung**
  Die Architektur und Algorithmik der Software des limitierten Gerätes kann bis zu einem gewissen Grad auf einen geringen Ressourcenverbrauch optimiert werden. Dies erfolgt i. d. R. durch eine gezielte Abstimmung der Implementierung auf den Verwendungszweck und die Hardwareplattform des Gerätes. Hierzu werden oft Codegenerierungsverfahren [57, 59] eingesetzt. Diese Vorgehensweise schränkt jedoch die Vielseitigkeit der Applikation ein und kann einen erhöhten Implementierungsaufwand nach sich ziehen.

- **Verzicht auf Funktionalität**
  Ein weiterer möglicher Ansatz ist, die Funktionalität des Systems zu reduzieren, indem etwa alternative oder speziell angepasste Middleware-Protokolle mit geringeren Anforderungen systemweit eingesetzt werden. Dies ist jedoch i. A. keine angemessene Lösung, da im System einzusetzende Protokolle oft äußere Vorgaben sind und durch mögliche proprietäre Veränderungen die Anforderung der Standard-Konformität (siehe Abschnitt 3.4) verletzt wird.

[1] http://www.sics.se/~adam/uip
[2] http://savannah.nongnu.org/projects/lwip
[3] Getestet auf einem herkömmlichen PC.

- **Verlagerung von Funktionalität**
  Der dritte mögliche Ansatz besteht in der Verlagerung von Funktionalität vom limitierten Gerät auf ein oder mehrere entfernte Geräte, die über entsprechend mehr Ressourcen verfügen und dem Aufbau einer Kommunikationsverbindung zwischen den Geräten. Somit wird aus dem limitierten Gerät und dem entfernten Host, der Funktionalität des limitierten Gerätes aufnimmt, ein verteiltes System.

Im Rahmen der vorliegenden Arbeit liegt der Fokus auf dem dritten Ansatz, also der Verlagerung von Funktionalität. Dies wird durch die Verwendung von *Geräte-Proxies* erreicht. Allgemein ist ein Proxy ein Stellvertreter, der eine Aufgabe anstelle einer Instanz durchführt, die er vertritt. Ein Geräte-Proxy hat die Aufgabe, ein (limitiertes) Gerät im Netzwerk zu vertreten: Er wird auf einem entfernten, i. d. R. leistungsfähigeren Gerät (Proxy-Host) ausgeführt und hat im Prinzip die Funktion eines Treibers (siehe Abbildung 3.1). Er übernimmt die Anbindung an die Middleware und kommuniziert mit dem Gerät über ein schlankes, auf die Fähigkeiten des Gerätes zugeschnittenes Protokoll. Das Protokoll zwischen Proxy und limitiertem Gerät kann vollkommen proprietär sein, da es zur Middleware hin durch den Proxy gekapselt und somit im System nicht sichtbar wird.

> Die Instanz, die die Lebenszyklusverwaltung der Proxies durchführt, wird im Folgenden *Application Manager (AM)* genannt. Als Bezeichnung für ein limitiertes Gerät, das durch einen Proxy vertreten wird, wird im Folgenden der Begriff *Transceiver* (in Anlehnung an den Sindrion-Transceiver) verwendet.

Das im Sindrion-System verwendete binäre Protokoll zwischen Proxy und Sindrion-Transceiver wird *Sindrion Control* genannt [86]. Es setzt direkt auf einer TCP-Verbindung auf und stellt einfache, auf die Funktionen des Sindrion-Transceiver zugeschnittene Kommandos bereit. Ein vergleichbares Verfahren kommt auch im Rahmen der *Jini Surrogate Architecture* zum Einsatz [89].

Abbildung 3.2 stellt ein Beispielszenarium mit Geräte-Proxies dar: Die limitierten Geräte (Transceiver) können in einem einfachen Beispiel Lichtschalter und Lampen sein. Sie werden von ihren Proxies im Middleware-System vertreten. Wird der Schalter betätigt, so erfolgt die Kommunikation zwischen Schalter und Proxy bzw. Lampe und Proxy über jeweils private, schlanke Protokolle (wie Sindrion Control), während die beiden Proxies über das Middleware-Protokoll (z. B. UPnP) kommunizieren. Eine native Middleware-basierte Applikation wie eine intelligente Haussteuerung, kann den Dienst der Lampe ebenfalls nutzen. Die Umsetzung auf das schlanke Protokoll der Lampe ist dabei vollkommen transparent.

3.4. Allgemeine Systemanforderungen

Im Folgenden werden typische allgemeine Anforderungen an Ubiquitous-Computing-Systeme [26, 76] diskutiert. Es wird dabei gesondert darauf eingegangen wie diese Anforderungen trotz oder gerade *aufgrund* der Anwendung des Proxy-Konzeptes erfüllt werden können.

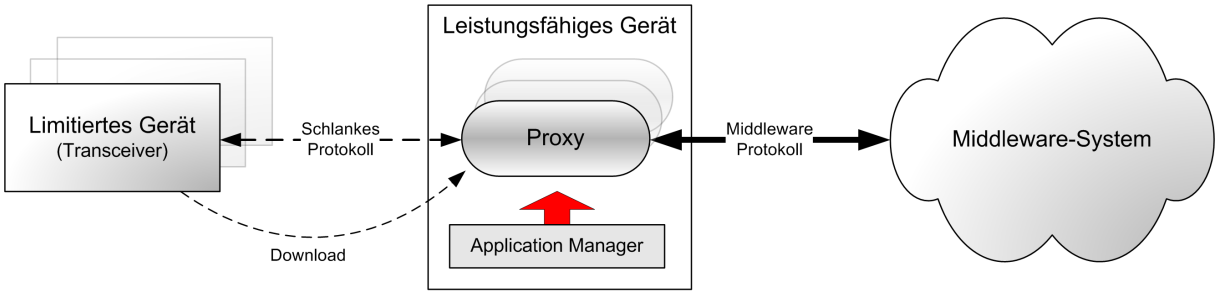


Abb. 3.1.: Der Proxy vertritt das limitierte Gerät im Middleware-basierten Netzwerk.

Selbstkonfiguration und prompte Interoperabilität

Ubiquitous-Computing-Systeme müssen in der Lage sein, sich selbst ohne Zutun eines Systemadministrators zu konfigurieren. Für Proxy-basierte Systeme bedeutet das insbesondere, dass ein Proxy transparent für den Nutzer gestartet werden muss, wenn ein Transceiver dem System beitritt und beendet, wenn der Transceiver nicht mehr zur Verfügung steht. Sofern der Code des Proxys auf dem Host des Application Managers noch nicht vorhanden ist, muss er automatisch dort bereitgestellt werden.

Geräte, die unterschiedlichen Standards entsprechen, sollten in der Lage sein, *Standard-übergreifend* zu interagieren. Das heißt, Geräte sollten auch „fremde“ Standards unterstützen. Solch *prompte Interoperabilität* kann durch die Verwendung Proxy-basierter Systeme unterstützt werden, da ein Proxy durchaus mehrere Schnittstellen zu unterschiedlichen Middleware-Standards besitzen kann. So ist es denkbar, dass ein Proxy die Funktionen seines Transceivers gleichzeitig auf einen UPnP- und einen Jini-Dienst abbildet oder ein Application Manager mehrere Typen von Transceivern, die unterschiedlichen Discovery-Protokollen entsprechen, unterstützt.

Ökonomischer Umgang mit Systemressourcen

Um Systemressourcen nicht unnötig zu verbrauchen, sollte ein einzelner Application Manager in der Lage sein, Proxies für möglichst viele Transceiver auszuführen. Neben der Bereitstellung entsprechender Verwaltungsfunktionalität zur Ausführung mehrerer Proxies ist zu berücksichtigen, dass die Ausführung eines Proxys in einem eigenständigen Prozess im Allgemeinen mehr Arbeitsspeicher benötigt, als in einem Thread, der im Prozess des Application Managers ausgeführt wird.

Weiterhin kann der Proxy Funktionen des Transceivers übernehmen, die über die Middleware-Anbindung hinaus gehen: So kann der Proxy rechenintensive Aufgaben wie die Aufbereitung der vom Transceiver gelieferten Rohdaten übernehmen. Als Beispiel sei eine Bildverarbeitung genannt, die nicht auf einer Kamera, sondern auf dem zugehörigen Proxy durchgeführt wird. Ferner kann der Proxy in vielen Anwendungsfällen Anfragen von Clients eigenständig beantworten, während sich das limitierte Gerät im Ruhezustand befindet, um Energie zu sparen. So ist z. B. denkbar, dass das limitierte Gerät seinen Proxy periodisch

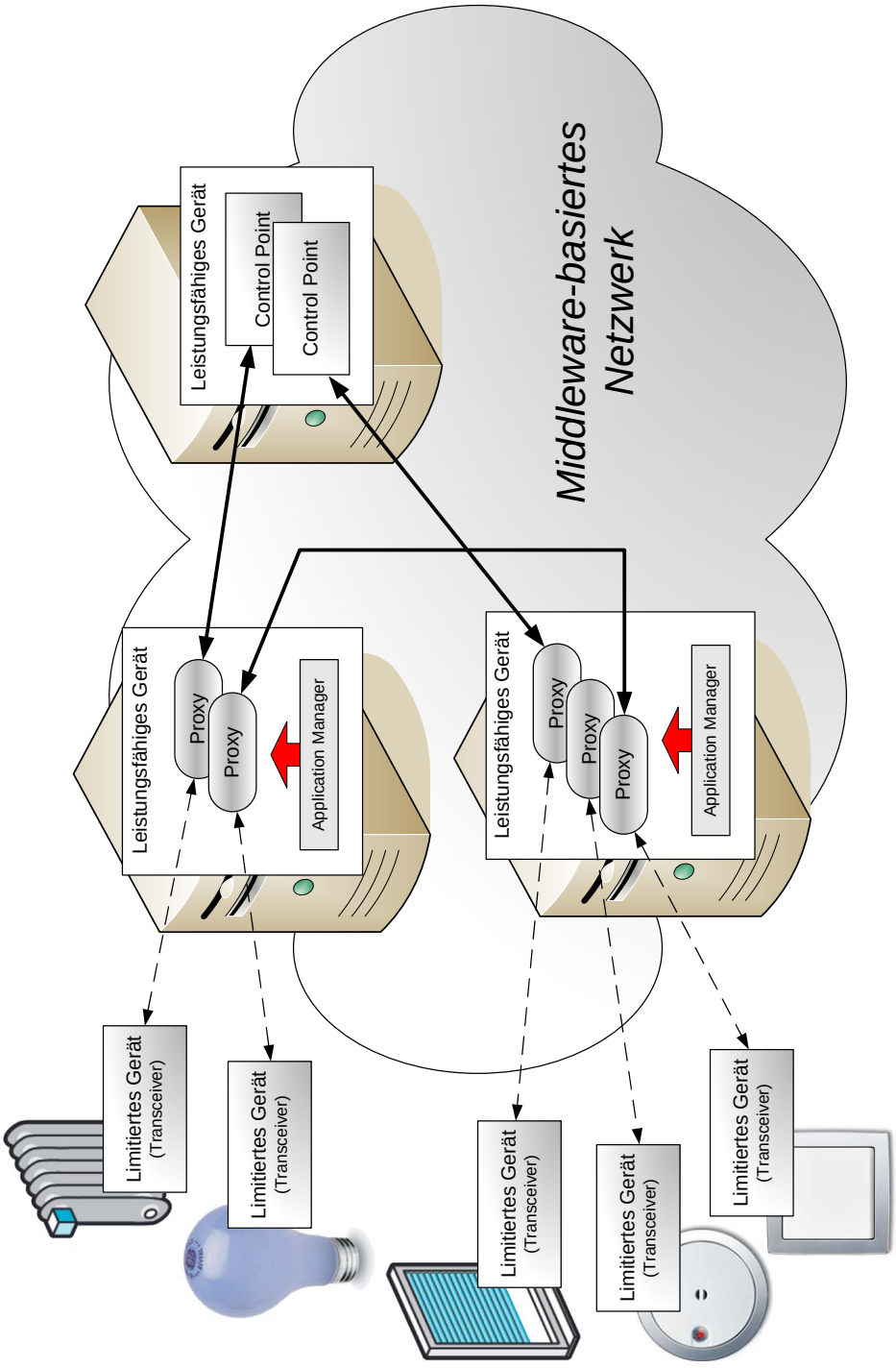


Abb. 3.2.: Beispielszenarium.

mit Messdaten versorgt und zwischendurch in den Standby-Modus wechselt, während der Proxy jederzeit auf Anfrage von Clients den zuletzt aktualisierten Wert liefert. Ein alternativer Ansatz ist, das limitierte Gerät bis auf rudimentäre Kommunikationsfähigkeiten herunter zu fahren, sodass das Gerät auch *asynchron* von seinem Proxy benachrichtigt werden kann, wie es beim Sindrion-Transceiver der Fall ist (siehe Anhang A.2).

Sowohl die Geräte-Proxies selbst, als auch die Application Manager benötigen Ressourcen und müssen an geeigneter Stelle im System ausgeführt werden. Grundsätzlich ist die Verwendung herkömmlicher PCs möglich, um Application Manager und Geräte-Proxies auszuführen. Allerdings macht es Sinn, auch andere bereits im System vorhandene Geräte für die Ausführung von Proxies zu verwenden. Beispiele für derartige Geräte sind *Residential Gateways*, *WIFI-Accesspoints* oder *SetTop Boxen*.

Zuverlässigkeit

Zuverlässigkeit spielt in verteilten Systemen allgemein eine zentrale Rolle, da durch die Verzahnung der verteilten Komponenten bei einem Ausfall schnell Teilbereiche oder gar das gesamte System betroffen sein können. Dieser Effekt wird bei Anwendung des Proxy-Konzeptes noch verstärkt: Da ein limitiertes Gerät und sein Proxy ein Seriensystem bilden (siehe Kapitel 5), reicht der Ausfall einer der beiden Komponenten, um den Ausfall des gesamten Dienstes zu erwirken. Werden mehrere Proxies auf einem einzelnen Host ausgeführt, so sind von einem Ausfall des Hosts gleich mehrere Proxies betroffen. Allgemein müssen Zuverlässigkeitsanforderungen bereits beim Systemdesign berücksichtigt werden – so ist eine automatische Lebenszyklusverwaltung bereits ein Zuverlässigkeitsmerkmal, da beendete Proxies bei Bedarf automatisch neu gestartet werden. Zusätzlich kann die Zuverlässigkeit des Systems durch den Einsatz dedizierter Fehlertoleranzmechanismen, wie sie im zweiten Teil dieser Arbeit betrachtet werden, erhöht werden. Letztendlich ist anzustreben, dass die Zuverlässigkeit eines Proxy-basierten Systems der eines Standalone-Systems nicht nachsteht.

Vertretbarer Implementierungsaufwand

Bei Anwendung von Proxies ist aufgrund der Verzahnung von Proxy und Transceiver ein erhöhter Implementierungsaufwand im Vergleich zur Realisierung von Standalone-Geräten zu erwarten. Der erhöhte Implementierungsaufwand kann allerdings durch geeignete Entwicklungsumgebungen sowie Codegenerierungsverfahren [59, 39, 9] kompensiert werden.

Standard-Konformität

Die verwendeten Technologien und Komponenten sollten möglichst Standard-konform sein oder zumindest keine Inkompatibilitäten mit Standard-konformen Komponenten hervorrufen. Für Proxy-basierte Systeme heißt das, dass Schnittstellen, die im Gesamtsystem sichtbar sind, entweder gängigen Standards entsprechen oder zumindest möglichst nah an gängige Standards angelehnt sein sollten.

4. Lebenszyklusverwaltung für Geräte-Proxies

Geräte-Proxies können nur dann effizient eingesetzt werden, wenn ihre Lebenszyklen automatisch und transparent verwaltet werden. Diese Aufgabe wird durch den Application Manager erfüllt und gliedert sich in vier wesentliche Schritte, die in den folgenden Abschnitten näher erläutert werden:

1. **Discovery der Transceiver**
   Der Application Manager muss in der Lage sein, Transceiver im System zu entdecken, für die noch kein Proxy ausgeführt wird.
2. **Aushandlung der Proxy-Ausführung**
   Es muss bestimmt werden, welcher Application Manager den Proxy des Transceivers ausführt.
3. **Bereitstellung des Proxy-Codes**
   Der Code des Proxys muss von einer Quelle bezogen werden.
4. **Ausführen bzw. Beenden der Proxies**
   Der Proxy muss automatisch gestartet werden bzw. beendet, wenn der Transceiver nicht mehr zur Verfügung steht.

Daraus ergeben sich spezielle Anforderungen, die Transceiver erfüllen müssen, um an einer Lebenszyklusverwaltung für Geräte-Proxies teilnehmen zu können. Neben der grundlegenden Fähigkeit zur Kommunikation mit anderen Geräten sind dies:

- **Unterstützung eines gemeinsamen Discovery-Protokolls**
  Damit der Transceiver von den Application Managern entdeckt werden kann, muss er an einem Discovery-Protokoll teilnehmen können, das auch von den Application Managern unterstützt wird.
- **Übermittlung der URL des Proxy-Codes**
  Da der Proxy von dem Transceiver selbst bereit gestellt werden soll, muss der Transceiver Kenntnis haben, unter welcher URL der Code des zugehörigen Proxys bezogen werden kann. Ferner muss er in der Lage sein, den Application Managern den URL mitzuteilen.
- **Unterstützung eines vom Proxy verwendeten Bind-Protokolls**
  Sobald der zugehörige Proxy gestartet ist, bindet er sich über ein eigenes Protokoll an den Transceiver. Dieses Protokoll kann privat sein, muss jedoch vom Transceiver und seinem Proxy unterstützt werden.

Um einem Transceiver eindeutig einen Proxy zuordnen zu können, ist ein *Pairing-Mechanismus* erforderlich, wie er in Abbildung 4.1 dargestellt ist. Ist der Transceiver im Zustand *paired*, signalisiert er, dass er an einen Proxy gebunden ist und damit kein neuer Proxy gestartet werden darf. Der Transceiver wechselt in den Zustand *paired*, sobald sein Proxy ausgeführt wird und eine Verbindung zu ihm aufbaut. Sobald der Proxy beendet wird, muss der Transceiver wieder in den Zustand *unpaired* wechseln und signalisiert damit, dass er den Start eines Proxys wünscht. Eine wichtige Voraussetzung für den ordnungsgemäßen Ablauf ist, dass der Proxy sich beim Beenden bei seinem Transceiver abmeldet. Für den Fall, dass er dies versäumt, muss der Transceiver die Aktivität des Proxys überwachen (*Heartbeat-Messages*) und bei Inaktivität die Verbindung lösen und in den Zustand *unpaired* übergehen.

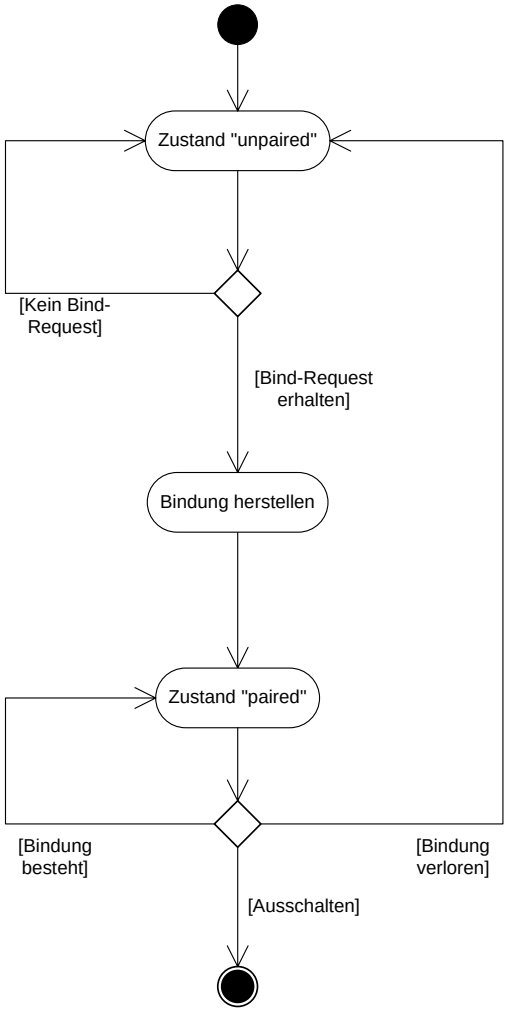


Abb. 4.1.: Pairing-Vorgang aus Sicht des Transceivers.

4.1. Discovery der Transceiver

Application Manager und Transceiver müssen ein gemeinsames Discovery-Protokoll verwenden, damit die Transceiver von den Application Managern entdeckt werden können. Dabei ist es erforderlich, dass das verwendete Discovery-Protokoll die Transceiver nicht überfordert. Entsprechend den in Abschnitt 3.4 gestellten Anforderungen ist es weiterhin wünschenswert, ein Standard-konformes Discovery-Protokoll einzusetzen. Die im Folgenden genannten Standards für Discovery sind bzgl. des Einsatzes im betrachteten System denkbar [12]. Eine Übersicht ist darüber hinaus in Tab. 4.1 gegeben (siehe auch Abschnitt 1.1).

- **Jini Multicast Request Protocol**
  Das von der Jini-Technologie verwendete Multicast Request Protocol ermöglicht das Finden eine *Jini Lookup Services* [90]. Das eigentliche Service Discovery bzw. die Anmeldung eines Dienstes wird anschließend über diesen Lookup Service abgewickelt. Zum Anmelden eines Dienstes lädt der Dienstanbieter ein sog. *Proxy-Dienstobjekt* auf den Lookup Server, das nach Abgleich mit einem Template vom Dienstnehmer bezogen werden kann.
- **Simple Service Discovery Protcol (SSDP)**
  SSDP ist das von UPnP verwendete Discovery-Protokoll [98]. Es setzt auf HTTPU bzw. HTTPMU (siehe Kapitel 2) auf, je nachdem, ob ein Gerät sich im Netz bekannt macht (*Notify*) oder auf eine Suchanfrage (*Find*) reagiert.
- **Peer Discovery Protocol**
  Das von JXTA verwendete Peer Discovery Protocol [91] definiert XML-basierte *Discovery Query Messages* und *Discovery Response Messages*. Ähnlich wie bei UPnP werden Advertisements verwendet, um Geräte im Netzwerk bekannt zu machen.
- **Service Location Protocol (SLP)**
  SLP ist ein Discovery-Protokoll, das ähnlich wie das Jini Multicast Request Protocol auf einem dreistufigen Konzept basiert und zentrale Lookup-Dienste, sog *Service Agents* voraussetzt.
- **Web Service Discovery**
  Web Service Discovery ist ein auf SOAP bzw. SOAP-over-UDP basierendes Discovery Protokoll [63]. Es wird im Rahmen von DPWS verwendet und besitzt Ähnlichkeiten zu dem von UPnP verwendeten SSDP.

Das Jini Multicast Request Protocol sowie SLP setzen die Verfügbarkeit von zentralen Komponenten in Form von Lookup Servern bzw. Directory Agents voraus. Dies stellt bzgl. des Discovery-Vorgangs limitierter Geräte prinzipiell kein Problem dar, da diese Funktion von den Application Managern übernommen werden kann. Allerdings setzt Jini die Verwendung von Java/RMI voraus[1], was das Multicast Request Protocol für limitierte Geräte

[1] Die Referenzimplementierung der Firma Sun verwendet zum Anmelden eines Dienstes an den Lookup Server das RMI-Protokoll.

	Jini	SSDP	JXTA	SLP	DPWS
Netzwerktransport	unabh.	TCP/IP	unabh.	unabh.	TCP/IP
Programmier-sprache	Java	unabh.	unabh.	unabh.	unabh.
Verwendung zentraler Komp.	ja	nein	teilw.	ja	nein
Eignung für limitierte Geräte	–	✓	✓	✓	✓

Tab. 4.1.: Gegenüberstellung der Discovery-Protokolle.

ungeeignet macht. Sofern ein TCP/IP-Stack (erforderlich für SSDP/DPWS) verfügbar ist, sind die anderen Protokolle prinzipiell für die Verwendung auf limitierten Geräten geeignet. Da UPnP jedoch bereits als Referenztechnologie dient, wurde im Rahmen der vorliegenden Arbeit SSDP als Discovery-Protokoll ausgewählt.

Wie aus Abbildung 4.2 deutlich wird, werden für eine ausschließliche Unterstützung von UPnP-Discovery die Protokolle SOAP und Teile von GENA, sowie die darunterliegenden Protokolle HTTP bzw. TCP nicht benötigt (vgl. Abbildung 2.3). Damit entfällt die aufwendige Verarbeitung von XML-Daten. Da lediglich ein UDP/IP-Stack benötigt wird, kann die notwendige Funktionalität für den Discovery-Vorgang im Transceiver bereitgestellt werden, ohne die Ressourcen des Gerätes über Maßen zu strapazieren.[2] Um Discovery zu unterstützen, muss ein limitiertes Gerät damit lediglich *SSDP Notify* Nachrichten (Advertisements) versenden, wie in Listing 4.1 dargestellt. Ein solches Advertisement ist nicht XML-basiert und kann in Teilen vorgefertigt im Speicher des Transceivers vorgehalten werden. Es muss daher nicht dynamisch (Ressourcen-intensiv) erzeugt werden. Lediglich individuelle Angaben wie die IP-Adresse des Gerätes müssen an entsprechender Stelle (Zeile 8 im Listing) in den vorbereiteten Code-Block eingesetzt werden.

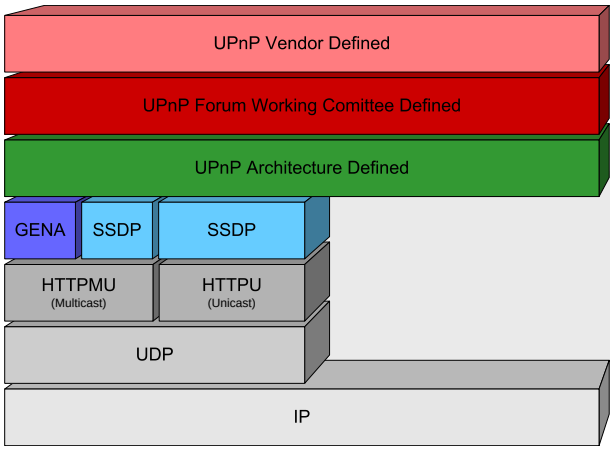


Abb. 4.2.: Für den Discoveryvorgang relevante Schichten des UPnP-Stacks.

[2] Es existieren eine Reihe schlanker UDP/IP-Stacks (siehe Abschnitt 3.2), die für die Verwendung in einem Transceiver geeignet sind.

Lst. 4.1: UPnP-Advertisement.

```
NOTIFY * HTTP/1.1
USN: uuid:10e761be09c+26584556+12cf041f+2395b7::urn:schemas-upnp-org:device:
    DimmableLight:1
HOST: 239.255.255.250:1900
NTS: ssdp:alive
NT: urn:schemas-upnp-org:device:DimmableLight:1
CACHE-CONTROL: max-age = 1800
SERVER: Windows XP/5.1, UPnP/1.0, Infineon LightBulb 1.0
LOCATION: http://129.217.184.3:3350/
```

Ein UPnP-Gerät, das keinerlei Funktionalität für Control und Event Notification bereit stellt, wird *UPnP Basic Device* genannt [98]. Damit stellt der Transceiver ein UPnP Basic Device dar. Allerdings muss ein Application Manager in der Lage sein, Transceiver, die durch Proxies vertreten werden sollen, von herkömmlichen UPnP Basic Devices zu unterscheiden. Hierzu ist ein Erkennungsmerkmal notwendig. Im Beispiel des Sindrion-Systems liegt dieses Erkennungsmerkmal in der Benennung der UPnP Presentation Page in der Device Description, die den Dateinamen *SindrionTX.html* besitzt. Ein weiteres Merkmal ist das Vorhandensein der Datei *Sindrion.zip*, die die Sindrion-spezifischen Beschreibungsdateien (s. u.) enthält. Details zu den Erkennungsmerkmalen befinden sich in Anhang A.1.1.

4.2. Aushandlung der Proxy-Ausführung

Zu einem Zeitpunkt darf stets nur ein aktiver Proxy pro Transceiver vorhanden sein, da ansonsten nicht eindeutig ist, welcher Proxy für den Transceiver zuständig ist. In diesem Abschnitt werden geeignete Verfahren diskutiert, wie die Zuständigkeiten für den Start eines Proxys ausgehandelt werden können.

Im einfachsten Ansatz initiiert jeder Application Manager den Start eines Proxys, sobald er einen neuen Transceiver entdeckt (*Proxy-Race*). Der Nachteil dieser Vorgehensweise ist die unnötige Beanspruchung von Ressourcen, wenn mehrere Proxy-Instanzen von unterschiedlichen Application Managern parallel gestartet werden, jedoch nur eine Instanz schließlich eine Bindung mit dem Transceiver eingehen kann. Dies ist insbesondere ein Problem, wenn mehrere Application Manager gleichzeitig den Code des Proxys direkt vom eingebetteten Webserver des Transceivers beziehen.

Um zu vermeiden, dass mehrere Application Manager gleichzeitig versuchen, einen Proxy zu starten und damit unnötige Systembelastung verursachen, können die Application Manager einen Transceiver z. B. nach dem *First-Come-First-Serve-Prinzip* reservieren: Ein Application Manager sendet ein Signal zur Sperrung an den Transceiver, sobald er ihn entdeckt hat. Der Application Manager, der sich zuerst bei dem Transceiver meldet, bekommt den Zuschlag und darf den Proxy starten. Parallel startet der Transceiver einen Timer und löst die Sperrung nach einem Timeout auf, falls sich kein Proxy meldet, um ein Pairing erneut zuzulassen.

Alternativ kann die Zuweisung des für den Start eines Proxys zuständigen Application Managers entsprechend der jeweiligen Situation erfolgen: So ist z. B. eine Zuweisung in Abhängigkeit der verfügbaren Ressourcen des Application Manager denkbar, um ein *Load-Balancing* im System zu realisieren. Für den Aushandlungsvorgang sind *Bidding-Protokolle* [37, 79] ein gangbarer Weg. Listing 4.2 zeigt den Algorithmus eines einfachen Bidding-Verfahrens in Pseudo-Code. Eine Voraussetzung ist, dass ein Application Manager in der Lage ist, die verfügbaren Systemressourcen seines Hosts, wie Energiereserven, verfügbarer Arbeits- und Festspeicher, sowie aktuelle Prozessorlast, zu ermitteln (s. u.).

Lst. 4.2: Einfacher Bidding-Algorithmus.

```
T = ID des entdeckten Transceivers;
L = Mein aktueller Lastwert;
MEIN_GEBOT = (L,T);

Sende MEIN_GEBOT an alle AMs;
Starte Timer;

wait until ((Gebote für T von allen anderen AMs empfangen) or (Timer
    abgelaufen));

if (MEIN_GEBOT ≤ min(EMPFANGENE_GEBOTE) or |EMPFANGENE_GEBOTE|
    = 0)
{
  Reserviere T;
}

else if (T nicht gebunden)
{
  Wiederhole gesamten Vorgang;
}
```

Entsprechend dem Algorithmus sendet jeder Application Manager seinen aktuellen Lastwert an alle anderen Application Manager. Dieser Lastwert wird als *Gebot* für die Ausführung eines Transceivers T betrachtet, wobei das Gebot mit dem geringsten Wert gewinnt. Jeder Application Manager prüft, ob sein Gebot alle anderen Gebote unterbietet oder gleichwertig ist. Ist dies der Fall oder wurden keine Gebote von anderen Application Managern empfangen, sendet er eine Pairing-Anforderung an den Transceiver. Falls mehrere Application Manager das gleiche Angebot abgegeben haben, so werden sie alle versuchen, den Transceiver zu reservieren. In diesem Fall greift die o. g. Regel, wonach lediglich die erste Pairing-Anforderung nach dem First-Come-First-Serve-Prinzip angenommen wird. Beobachtet der Application Manager anschließend keine Bindung des Transceivers, wird der gesamte Vorgang wiederholt.

Der aktuelle Lastwert spiegelt die Menge an verfügbaren Ressourcen auf dem Host des Application Managers wider. Diese Ressourcen sind der verfügbare Arbeits- und Festspeicher, Energiereserven, sowie die momentane Prozessorauslastung und die verfügbare Übertragungsbandbreite der Netzwerkschnittstelle. Der Lastwert sagt etwas darüber aus, wie gut ein Application Manager in der Lage ist, einen weiteren Proxy auszuführen. Die Schwierigkeit ist hier, Lastwerte auf ein vergleichbares Maß zu normieren. Bei Anwendung

des o. g. einfachen Aushandlungsverfahrens hat sich allerdings gezeigt, dass eine Einstufung in wenige Bereiche wie *Low*, *Medium* und *High* ausreicht, um Application Manager entsprechend ihrer Lastsituation zu klassifizieren [58, 107]. Schließlich geht es bei den Aushandlungsverfahren im Wesentlichen darum, Application Manager, die ihrer eigenen Einschätzung nach ausgelastet sind, entsprechend auszuschließen.

Generell empfiehlt es sich, Geräte, die nicht über eine stationäre Energieversorgung verfügen, für die Ausführung von Proxies als nur *bedingt geeignet* einzustufen, um Hosts mit unbeschränkten Energiereserven den Vortritt zu lassen. Als Erweiterung ist denkbar, dass limitierte Geräte die Anforderungen ihres Proxys beschreiben. So kann ein limitiertes Gerät direkt mitteilen, dass der Proxy rechenintensive Operationen ausführt und damit entsprechende Prozessorleistung verlangt.

Die eigentliche Messung der verfügbaren Ressourcen gestaltet sich wegen der verwendeten Laufzeitumgebungen allerdings als problematisch. Der Grund liegt darin, dass Laufzeitumgebungen wie die *Java Virtual Machine* oder das *.Net Runtime Environment* Applikationen vom Betriebssystem abschotten und nur begrenzt Möglichkeiten bieten, den lokalen Systemstatus abzufragen. Dies trifft vor allem auf die Java Virtual Machine zu, da entsprechend der Plattformunabhängigkeit von Java die Abstraktion des darunterliegenden Systems ausgeprägter ist, als es bei der .Net Laufzeitumgebung der Fall ist. Unter Berücksichtigung dieser Einschränkungen bieten sich folgende Ansätze für die Abfrage von Systeminformationen an:

1. **Nutzung von Monitoring-Schnittstellen der Laufzeitumgebung**
   Die Laufzeitumgebungen bieten zum Teil Monitoring-Schnittstellen, um in begrenztem Umfang Statusinformationen der Hostplattform abzufragen.

2. **Nutzung des Native Interface**
   Sofern die Laufzeitumgebung Zugriff auf native Systembibliotheken bietet (z. B. *Java Native Interface (JNI)*), können prinzipiell sämtliche Informationen abgefragt werden, die auch vom Betriebssystem ermittelt werden können. Allerdings ist diese Lösung mit einem erhöhten Implementierungsaufwand verbunden, da native Komponenten entsprechend bereitgestellt werden müssen.

3. **Nutzung impliziter Indikatoren**
   Kommen die ersten beiden genannte Ansätze nicht in Frage, so können in manchen Fällen „implizite Indikatoren“ verwendet werden, um auf den Status des Systems zu schließen. So kann z. B. anhand des Zeitraums, den ein Thread vom Scheduler Rechenzeit zugewiesen bekommt, darauf geschlossen werden, wie stark das System ausgelastet ist [95].[3]

[3] Dies ist jedoch eine relativ ungenaue Methode, die zudem auf ein Einmessen der Last im Ruhezustand angewiesen ist.

4.3. Bereitstellung des Proxy-Codes

Damit der Code eines Proxys vom Application Manager automatisch bezogen werden kann, müssen Informationen darüber geliefert werden, woher der Code zu beziehen ist. Dies kann z. B. durch die Angabe eines URLs geschehen. Zu diesem Zweck wurde das Modell der UPnP Descriptions durch sog. *Sindrion Descriptions* erweitert [87]. Die Sindrion Description ist in ihrer Notation an die UPnP Descriptions angelehnt. Sie gibt an, in welchen Archiven sich die Komponenten von Proxy, Specific Control Point (SCP) und Sindrion-Applet (siehe Kapitel 1) befinden. Ein Beispiel einer Sindrion Description befindet sich in Anhang B.

Die Komponenten des Proxys können zum einen vom Transceiver selbst herunter geladen werden, zum anderen ist es denkbar, dass die Description auf einen entfernten HTTP-Server (Webserver) verweist. Das Ablegen des Codes auf einem entfernten HTTP-Server würde den Transceiver entlasten, da im Transceiver kein entsprechender Speicherplatz bereitgestellt werden müsste. Hinzu kommt, dass der Download eines Proxys direkt vom Transceiver je nach Datenrate entsprechend viel Zeit in Anspruch nehmen kann: Selbst wenn die volle Bandbreite von 50 kbit/s des Sindrion-Transceivers (siehe Anhang A.2) zur Verfügung stünde, würden immer noch knapp 20 s für den Download eines 100 kB Proxy anfallen. Bei einer Netto-Datenrate von 30 kbit/s, würden bereits knapp 30 s für den Download des Codes anfallen (siehe Abbildung 4.3). Allerdings ist bei Verwendung eines entfernten HTTP-Servers Internet-Konnektivität des Application Managers erforderlich, sofern der HTTP-Server sich nicht im lokalen Netzwerk befindet.

Neben dem einfachen Ansatz, den Code vom HTTP-Server stets direkt in den Speicher des Application Manager zu laden, wurde im Rahmen des Sindrion-Projektes vorgesehen, Code in einem lokalen *Code-Repository* auf jedem Application Manager abzulegen und im Sinne eines Caches vorzuhalten. Somit kann der Code des Proxys ggf. direkt aus dem jeweilig lokalen Repository bezogen werden, anstatt ihn über die im Vergleich zum Zugriff auf das Repository schmalbandige Netzwerkanbindung des Transceivers zu beziehen.

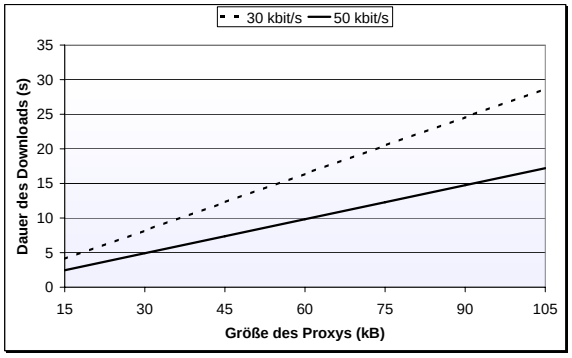


Abb. 4.3.: Download-Dauer des Proxys vom Transceiver.

Jeder Application Manager kann sein eigenes Repository anderen Application Managern zur Verfügung stellen. Umgekehrt können Application Manager, die den Code eines Proxys nicht in ihrem lokalen Repository vorfinden, ihn ggf. vom Repository eines anderen Application Managers herunter laden. Um jeweils ihr lokales Repository zu exportieren, müssen die Application Manager einen Service zur Veröffentlichung ihres Repositorys anbieten.

Der Ablauf des Beziehens von Code sieht wie in Abbildung 4.4 dargestellt aus:

1. Ein Application Manager prüft, ob der Code des Proxys bereits lokal verfügbar ist. Falls ja, lädt er ihn aus seinem eigenen Repository.

2. Ist der Code nicht lokal verfügbar, prüft der Application Manager die Repositories anderer in Reichweite befindlicher Application Manager.

3. Ist der Code auch dort nicht verfügbar, wird der Code vom Transceiver direkt heruntergeladen.

Um die Möglichkeit zu schaffen, trotz einer Verteilung des Codes in den Repositories die Implementierungen von Komponenten zu pflegen, wurde im Rahmen des Sindrion-Projektes eine Versionierung eingeführt. So wird jeweils eine *Haupt-* und *Nebenversionsnummer* in den Sindrion Descriptions (siehe Anhang B) zu jeder einzelnen Komponente vergeben. Haben zwei Komponenten die selbe Hauptversionsnummer, jedoch unterschiedliche Nebenversionsnummern, so ist die Komponente mit der höheren Nebenversionnummer zu der anderen abwärtskompatibel. Komponenten mit unterschiedlichen Hauptversionsnummern sind nicht notwendigerweise zueinander kompatibel.

Wird der Code von einem anderen Application Manager oder direkt vom Transceiver bezogen, wird er in das lokale Repository eingepflegt und mit einem Zeitstempel versehen. Vorhandene Komponenten mit gleicher Haupt- jedoch kleinerer Nebenversionsnummer werden dabei überschrieben. Vorhandene Komponenten mit unterschiedlichen Hauptversionsnummern werden entsprechend verwahrt. Jedesmal, wenn eine Code-Komponente aus dem lokalen Repository verwendet wird, wird der zugehörige Zeitstempel aktualisiert. Somit können anhand der Zeitstempel Elemente des Repositorys nach dem LRU-Verfahren[4] entfernt werden, um die Größe des Repositorys zu beschränken.

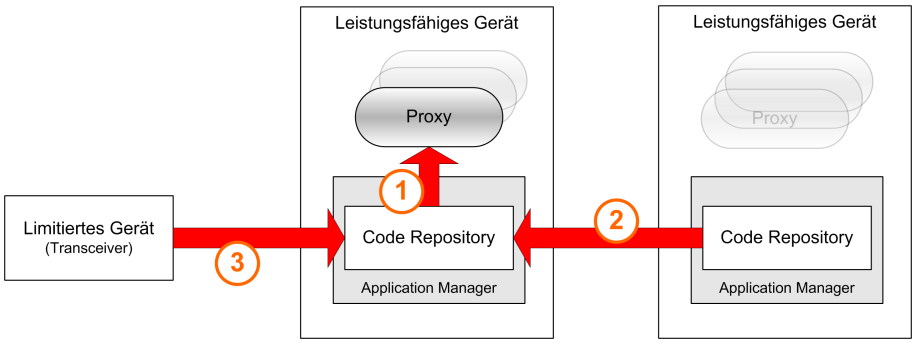


Abb. 4.4.: Bezugsreihenfolge des Proxy-Codes.

[4]LRU: Least Recently Used

4.4. Ausführen und Beenden der Proxies

Sobald der Code des Proxys auf der lokalen Plattform verfügbar ist, kann er ausgeführt werden. Da ein Application Manager selbst auf unterschiedlichen Betriebssystemen ausgeführt werden kann, spielt Plattformunabhängigkeit bei der Ausführung des Proxys eine zentrale Rolle. Daher bieten sich insbesondere Laufzeitumgebungen wie die *Java Virtual Machine (JVM)* oder auch die *.Net Common Language Runtime* zur Ausführung von Proxies an. Ferner ist die Fähigkeit zum *dynamischen Binden* von Code wichtig, wenn ein Proxy zur Laufzeit in eine bereits bestehende virtuelle Maschine geladen wird. Allgemein kann die Ausführung des Proxy-Codes auf zwei Arten erfolgen:

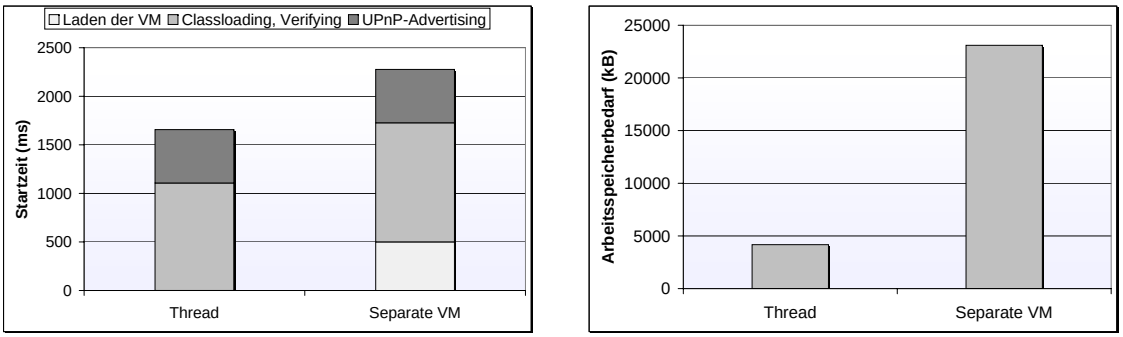
- **Ausführung als Thread**
  Bei einer Ausführung als Thread wird das Objekt, das die Initialisierungsmethoden des Proxys enthält, von einem *Classloader* in die selbe virtuelle Maschine geladen, die auch den Application Manager ausführt. Der Vorteil dieser Vorgehensweise liegt in der kurzen Startzeit des Proxys sowie im vergleichsweise geringen Ressourcenverbrauch. Allerdings birgt dieser Ansatz auch potentielle Risiken, da ein fehlerhafter Proxy bei Ausführung als Thread nicht einfach beendet werden kann.[5]

- **Ausführung als Prozess**
  Bei einer Ausführung als Prozess wird der Proxy in einer separaten virtuellen Maschine (VM) gestartet. Diese VM wird direkt aus der virtuellen Maschine des Application Managers heraus erzeugt und bekommt die Klassen des Proxys über einen *Classpath Parameter* übergeben. Der Vorteil dieser Vorgehensweise liegt in der strikten Trennung zwischen Application Manager und Proxies. So kann der Application Manager bei Bedarf den Prozess des Proxys – im Unterschied zur oben beschriebenen Ausführung als Thread – einfach beenden. Von Nachteil sind der erhöhte Ressourcenbedarf und die aufwendigere Interaktion mit dem Proxy, da eine Kommunikation zwischen Application Manager und Proxy nunmehr über die Grenzen virtueller Maschinen, z. B. über RPC-Mechanismen wie Java RMI erfolgen muss.

Abbildung 4.5(a) zeigt die durchschnittlichen Startzeiten eines Proxys, der beispielhaft den Dienst eines *UPnP Binary Light* bereit stellt. Wird der Proxy als Thread gestartet, so fallen durchschnittlich ca. 1,5 s Gesamtzeit für den Start an. Wie in der Abbildung dargestellt, werden ca. 0,5 s für die Anmeldung des UPnP-Dienstes benötigt. Wird der Proxy als Prozess (also in einer eigenen virtuellen Maschine) gestartet, so erhöht sich der Zeitaufwand nochmals um ca. 0,5 s für den Start der separaten VM.[6] Hierbei wird allerdings davon ausgegangen, dass der Code der VM sich bereits im Memory-Cache der Maschine befand. Auch bzgl. des Ressourcenbedarfs zeigen sich große Unterschiede bei den beiden Startvarianten, wie in Abbildung 4.5(b) dargestellt: Während ein Proxy, der als Thread in der virtuellen Maschine des Application Managers ausgeführt wird, lediglich rund 4 MB Arbeitsspeichder benötigt, fallen bei Ausführung in einer separaten VM über 20 MB an erforderlichem Arbeitsspeicher an.

[5] Die Methoden `Thread.stop()` bzw. `Thread.abort()` zum Abbrechen laufender Threads unter Java bzw. .Net arbeiten unzuverlässig und gelten daher als *deprecated*.

[6] Für die Messungen wurde die VM der Java Standard Edition 6 verwendet.



(a) Startzeit eines Proxys.

(b) Arbeitsspeicherbedarf eines Proxys.

Abb. 4.5.: Startzeit und Ressourcenbedarf bei Ausführung eines Proxys als Thread und in einer separaten virtuellen Maschine.

4.5. Beispielablauf

Im Folgenden wird beispielhaft die Ausführung eines Proxys anhand eines Setups bestehend aus einem Transceiver und zwei Application Managern beschrieben. Dabei wird das Sindrion-System als konkretes Beispiel herangezogen. Der entsprechende Ablauf ist in Abbildung 4.6 illustriert.

1. Sobald der Sindrion-Transceiver mit Energie versorgt wird, bezieht er eine IP-Adresse über DHCP bzw. über Auto IP.
2. Der Transceiver sendet Advertisements über Multicast-Nachrichten, die von den Application Managern empfangen werden.
3. Die Application Manager laden die UPnP Device Description herunter und schließen aus dem URL der Presentation Page, dass es sich bei dem entdeckten UPnP Basic Device um einen Sindrion-Transceiver handelt.
4. Jeder der beiden Application Manager sendet eine Sperranforderung an den Transceiver. Der Transceiver gibt dem Application Manager, der zuerst die Sperranforderung gesendet hat, den Zuschlag und teilt dem anderen Application Manager mit, dass er bereits gesperrt wurde.
5. Der Application Manager, der den Zuschlag bekommen hat, lädt die Sindrion Description-Datei vom Transceiver herunter und wertet sie aus. Hierdurch erfährt er, welche Dateien in welchen Versionen er zur Ausführung des Proxys beziehen muss.
6. Der Application Manager sucht den Proxy-Code zuerst in seinem lokalen Code-Repository, liegt der Code dort nicht, vor fragt er bei dem anderen Application Manager an, ob er den Code in seinem Repository hat. Ist das auch nicht der Fall, lädt der er den Code direkt vom Transceiver herunter.

7. Der Application Manager führt den Proxy aus – entweder als Thread oder in einer separaten Java Virtual Machine. Sobald der Proxy gestartet ist, bindet sich sich dieser mit dem Transceiver und macht sich über eigene Advertisements im System bekannt.

4.6. Der Sindrion Application Manager

Die in den bisherigen Abschnitten beschriebenen Konzepte und Verfahren wurden im Rahmen der vorliegenden Arbeit in der Implementierung eines Application Managers für das Sindrion-Projekt umgesetzt. Ein besonderer Wert wurde dabei auf hohe Konfigurierbarkeit sowie eine möglichst nahtlose Integration in das Betriebssystem gelegt. Im folgenden wird kurz auf einige Merkmale der entstandenen Software eingegangen. Eine detaillierte Beschreibung der Architektur des Sindrion Application Managers befindet sich in Anhang A.1.

Implementiert wurde der Sindrion Application Manager unter Verwendung der Java Standard Edition 6 und lässt sich damit auf gängigen Plattformen wie Microsoft Windows oder auch Linux ausführen. Neben einer Ausführung als eigenständige Applikation ist die Integration als Bundle in ein OSGi-Framework möglich. Als UPnP-Stack kam der Java-basierte *Infineon UPnP/1.0*, der von Infineon Technologies zur Verfügung gestellt wurde, zum Einsatz.

Abb. 4.7.: Tray-Integration.

Nach dem Start läuft das Programm im Hintergrund und reagiert automatisch, sobald es Sindrion-Transceiver im Netzwerk entdeckt. Neben dem vollautomatischen Modus ist auch ein manueller Modus konfigurierbar. Bei Verwendung der Windows-Plattform fügt sich der Application Manager in System-konforme Benutzerschnittstellen wie die *Systemsteuerung* oder das *System Tray* ein (Abbildung 4.7). Alle grafischen Benutzerschnittstellen sind, wie in Abbildung 4.8 dargestellt, an das *Look and Feel* der Oberfläche von Windows XP angelehnt, um dem Benutzer ein vertrautes Arbeitsumfeld zu bieten. Die beiden wesentlichen GUI-Komponenten des Sindrion Application Managers sind:

- **Der Sindrion Explorer**
  Der Sindrion Explorer ist ein an den Windows-Explorer angelehntes Kontrollfenster, das im System vorhandene Sindrion-Transceiver sowie herkömmliche UPnP-Geräte anzeigt. Er erlaubt zum einen den manuellen Start von Proxies, als auch eine einfache regelbasierte Konfiguration, für welche Transceiver Proxies automatisch auszuführen sind.

- **Das Sindrion Application Center**
  Über das Application Center lässt sich das Verhalten des Application Managers konfigurieren. Dies betrifft z. B. die automatische Ausführung von Proxies sowie die

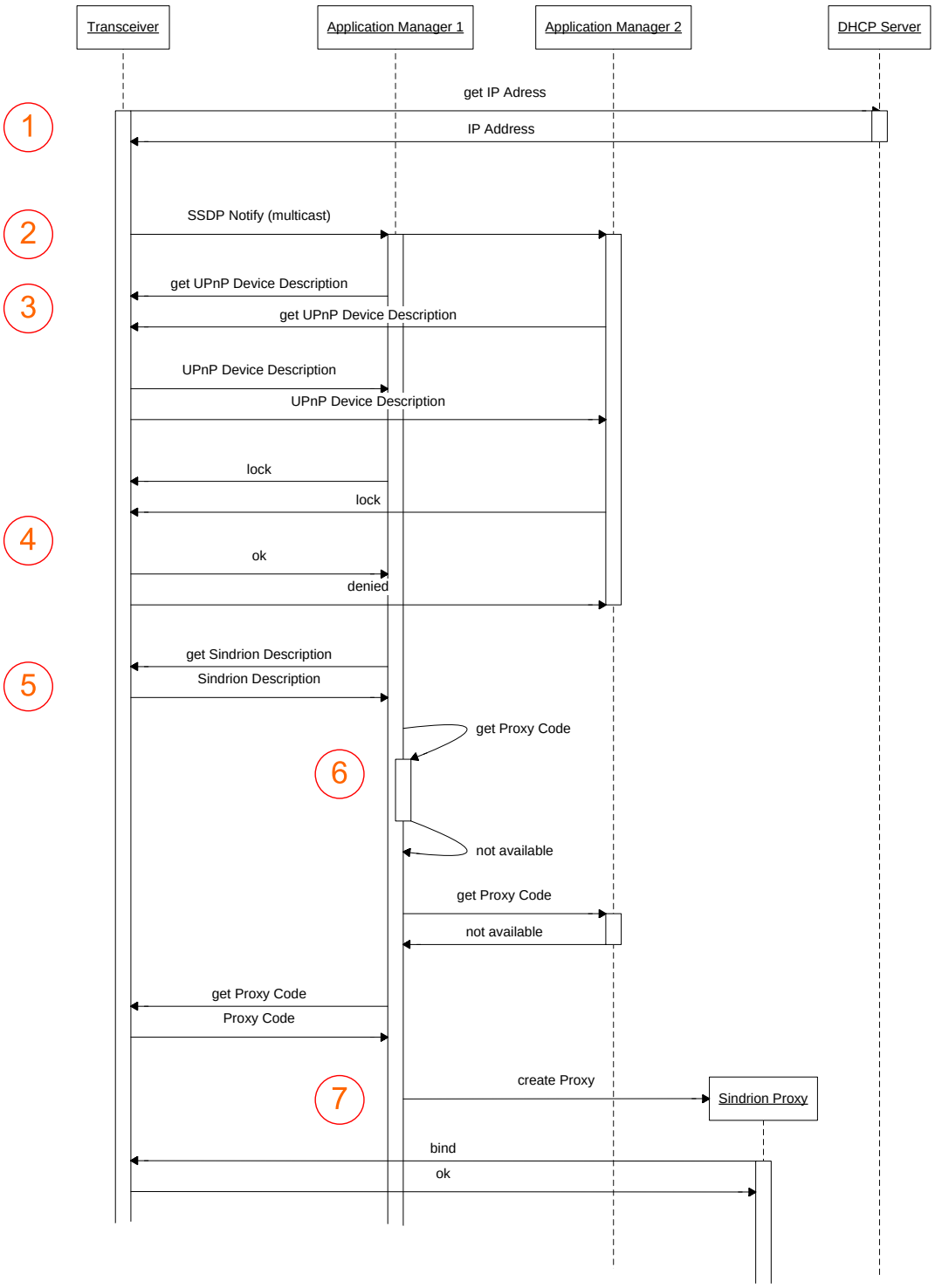


Abb. 4.6.: Sequenzdiagramm der Proxy-Ausführung.

Art der Ausführung (Thread oder Prozess). Weiterhin lässt sich über das Application Center das Code Repository des Application Managers bei Bedarf leeren.

Die Implementierung des Sindrion Application Managers diente im Rahmen der vorliegenden Arbeit als Grundlage für weitere Untersuchungen. So wurden Messungen am realen System unter Verwendung dieser Software durchgeführt.

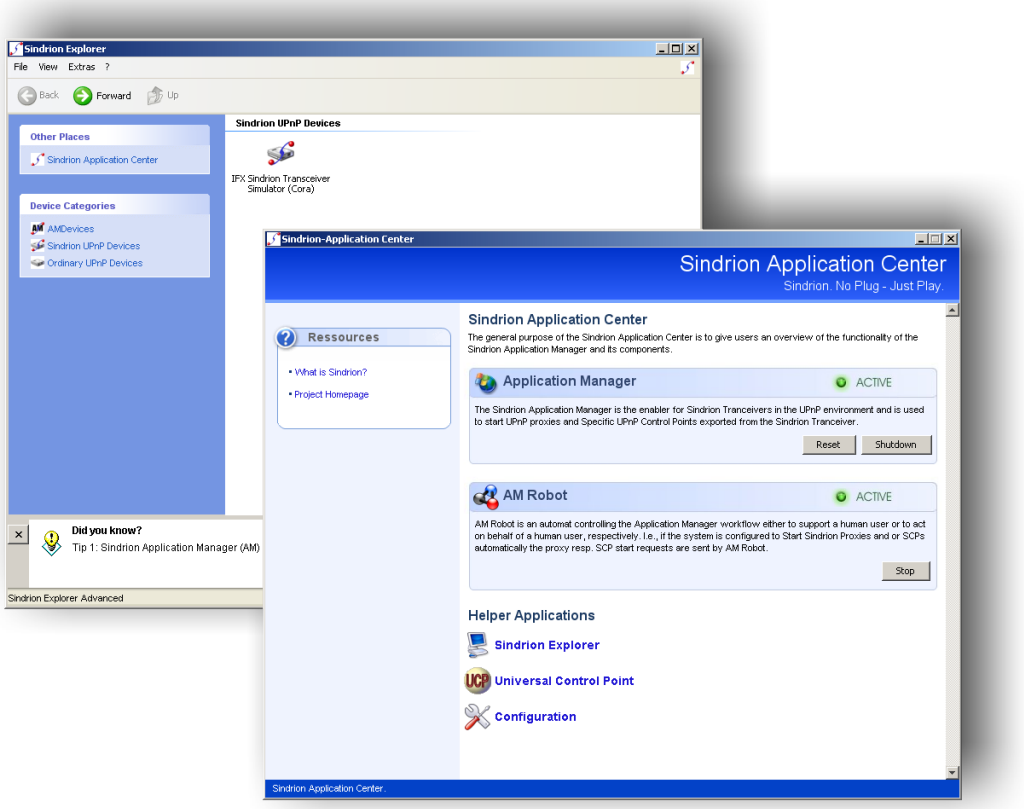


Abb. 4.8.: Grafische Sindrion-Benutzerschnittstellen.

Teil II.

Fehlertoleranz

5. Grundlagen der Fehlertoleranz

Ein *System* ist ein identifizierbarer Mechanismus, der sich durch sein Verhalten sowie durch seine Schnittstellen zur Außenwelt auszeichnet [6]. Es enthält eine oder mehrere *Komponenten*, wobei eine Komponente auch wieder ein System sein kann (in diesem Fall wird auch von einem Sub-System gesprochen). So ist das Gespann aus Transceiver und Proxy ein (Sub-)System, während Transceiver und Proxy einzeln betrachtet jeweils als Komponenten betrachtet werden.[1]

Keine Komponente eines Systems hat eine unendliche Lebenserwartung. Alterungserscheinungen, überschrittene Fertigungstoleranzen, Programmierfehler, Kommunikations-Übertragungsfehler und auch aufgezehrte Energiereserven sind mögliche Ursachen, die zu einem Versagen bzw. zum Ausfall von Komponenten führen können. Mit der Komplexität von Systemen steigt die Wahrscheinlichkeit, dass Fehler vorhanden sind, die Störungen hervorrufen. Je mehr funktionale Abhängigkeiten zwischen den Komponenten eines Systems bestehen, desto größer ist die Wahrscheinlichkeit, dass Störungen größere Bereiche eines Systems – bis hin zum gesamten System – beeinflussen oder gar zum Versagen führen. Dies wird insbesondere bei *Seriensystemen* deutlich: Ist $R_i(t)$ die Zuverlässigkeit einer Komponente eines Systems, das aus n Komponenten besteht, so ergibt sich die Gesamtzuverlässigkeit des Systems durch das Produkt der Zuverlässigkeiten der einzelnen Komponenten [47]:

$$R_{series}(t) = \prod_{i=1}^{n} R_i(t)$$

Damit ist die Zuverlässigkeit bzw. Zeit bis zum Ausfall eines Seriensystems entsprechend geringer als die Zuverlässigkeiten der Einzelkomponenten. Dieser Zusammenhang ist insbesondere wichtig, da es sich bei einem System bestehend aus einem Transceiver und dem zugehörigen Proxy um ein Seriensystem handelt. Aus diesem Grund ist *Fehlertoleranz* ein wichtiger Aspekt bei der Entwicklung Proxy-basierter Systeme. Viele Fehlertoleranzmerkmale eines Systems sind bereits *Design-inhärent* – so wird am Beispiel des Sindrion-Systems bei Ausfall eines Proxys automatisch ein neuer Proxy gestartet – doch kann auf diese Weise keine umfassende Fehlertoleranz erreicht werden, da der Ausfall eines Proxys sich unmittelbar auf weitere Komponenten auswirken und *Inkonsistenzen* hervorrufen kann. Daher sind weitere Protokolle und Verfahren notwendig, die als *dedizierte Fehlertoleranz-Erweiterungen* betrachtet werden können und in diesem Teil der vorliegenden Arbeit diskutiert werden.

[1] Natürlich ist ein Transceiver bzw. ein Proxy für sich auch wieder ein System, jedoch ist die *innere* Architektur dieser Systeme für die Betrachtung nicht von Bedeutung.

5.1. Fehler, Störung und Versagen

Im Folgenden werden im Zusammenhang mit Fehlertoleranz die Begriffe *Fehler*, *Störung* und *Versagen* in Anlehnung an die englischen Begriffen *Fault*, *Error* und *Failure* verwendet [47]. Die jeweiligen Bedeutungen der Begriffe werden in Abbildung 5.1 veranschaulicht: Von Versagen (Failure) sprechen wir dann, wenn ein System oder eine Komponente sich nicht mehr spezifikationsgemäß verhält. Ein extremer Fall des Versagens einer Komponente ist ihr kompletter Ausfall. Man kann allerdings nur dann unmittelbar auf das bevorstehende Versagen einer Komponente oder eines Systems schließen, wenn zuvor eine Störung (Error) bemerkt wurde. Die Störung ist damit der Teil des Systemzustandes, der zum Versagen des Systems bzw. der Komponente führen kann (aber nicht muss). Das Versagen selbst ist erst observierbar, nachdem es eingetreten ist und auch nur, sofern eine Instanz das Verhalten der Komponente oder des Systems überwacht. Das Versagen eines Subsystems oder einer Komponente kann sich wiederum als Störung im übergeordneten System bemerkbar machen. Unmittelbare Ursache von Störungen sind Fehler (Faults).

Zwei Beispiele: Ein Computerprogramm kann Fehler enthalten („Bugs“ in der Dateizugriffsroutine), die sich im Betrieb in einer Störung manifestieren (Datei nicht zugreifbar). Wird die Störung nicht angemessen behandelt (z. B. Laden von Default-Werten), kann sie zum Versagen des Programms (Ausfall bzw. Absturz) führen. Aber auch, wenn ein Mobiltelefon ausfällt (Versagen), weil sein Akku keine Energie mehr liefert (Störung), ist im weitesten Sinne ein Fehler die Ursache, da der Akku überbeansprucht bzw. nicht rechtzeitig geladen wurde (menschliches Versagen).

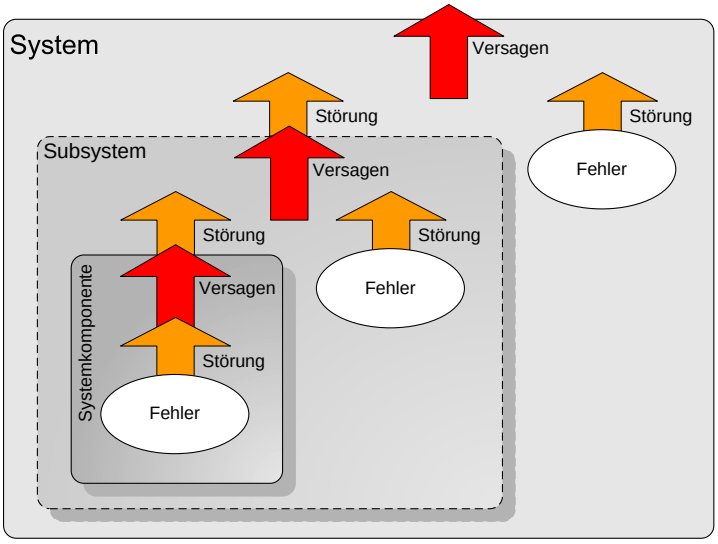


Abb. 5.1.: Zusammenhang von *Fehler*, *Störung* und *Versagen*.

Damit liegt die Ursache von Störungen bzw. vom Versagen in der Präsenz von Fehlern. Generell sind Fehler nicht gänzlich zu vermeiden, auch wenn man stets im Vorfeld bemüht ist, ihre Anzahl gering zu halten. Hinzu kommt, dass Fehler mitunter erst nach einer gewissen Zeit z. B. aufgrund von Alterungserscheinungen auftreten. Die Auftrittswahrscheinlichkeit dieser Art von Fehlern kann z. B. durch *präventiven Austausch* von Komponenten vor Ablauf der erwarteten Lebensdauer reduziert werden.

Doch selbst wenn Fehler nicht gänzlich vermieden werden können, ist es jedoch möglich, mit den Auswirkungen der Fehler, also den Störungen, umzugehen, um Versagen zu vermeiden. Da man Störungen erst observieren kann, *nachdem* sie aufgetreten sind, ist die einzige Reaktionsmöglichkeit, sie zu *kaschieren*. Dies bedeutet, dass eine Störung für die übergeordneten Systemstrukturen transparent gemacht wird, indem ihre Auswirkungen abgefangen bzw. eingedämmt werden. Natürlich kann (und sollte) dies auch ein Beheben der zugrunde liegenden Fehler beinhalten.

5.2. Grundbausteine fehlertoleranter Systeme

Verfahren zum Aufbau fehlertoleranter Systeme können in ein Ebenenmodell eingeordnet werden, wie es in Abbildung 5.2 dargestellt ist. Fehlertoleranzverfahren sind oberhalb der Infrastrukturebene angeordnet und erweitern diese um entsprechende Fehlertoleranzmerkmale. Im Wesentlichen können drei allgemeine Ansätze als Beitrag zur Fehlertoleranz unterschieden werden:

- **Migration von Code und Daten**
  Wird ein drohendes Versagen eines Gerätes erkannt, so wird die Funktionalität des Gerätes (wenn möglich zur Laufzeit) auf ein alternatives Gerät migriert.
- **Anwendung von Rollback-Recovery-Verfahren**
  Das System wird befähigt, beim Versagen einzelner Komponenten einen gültigen Zustand „vor“ Auftreten der Störung wieder anzufahren.
- **Redundante Haltung von Code und Daten**
  Sowohl Code als auch Daten können redundant im System vorgehalten werden, um bei Ausfall einer Komponente auf eine bereitstehende *Schattenkopie* umzuschalten.

Die genannten Fehlertoleranzverfahren sind auf die Verfügbarkeit bestimmter Systemmerkmale angewiesen – im Folgenden als *Grundbausteine* bezeichnet [47]. Diese sind:

- **Fail-Stop-Prozesse**
  Typischerweise stellt eine Komponente bei Auftritt einer Störung ihre Arbeit nicht unmittelbar ein (Fail-Stop). Eher verhält sie sich auf eine arbitrate Weise, die der Spezifikation der Komponente zwar eindeutig widerspricht, jedoch mitunter nur schwer als Versagen zu erkennen ist. Fehlertoleranzverfahren sind allerdings darauf angewiesen, dass ein Versagen zuverlässig und in endlicher Zeit (d.h. möglichst unmittelbar) erkannt wird. Um ein arbitrates Verhalten von Komponenten auf ein Fail-Stop-Verhalten abzubilden, existieren sog. *Byzantine-Agreement-Protokolle* [47].

Dabei werden die Aussagen verschiedener möglicherweise gestörter Komponenten miteinander verglichen, um das Versagen einer dieser Komponenten zu erkennen. Ein weiterer gängiger Ansatz zur Realisierung von Fail-Stop-Prozessen sind *Exception-Mechanismen*, wie sie von modernen Programmiersprachen wie Java oder den Sprachen der .Net-Familie verwendet werden.

- **Stable Storage**
  Rollback-Recovery-Verfahren sind auf die Verfügbarkeit eines ausfallsicheren Datenspeichers (Stable Storage) angewiesen. Derartige Speichermedien stehen typischerweise in Form von RAID-Systemen zur Verfügung. Je nach Applikation kann aber auch eine einfache Festplatte oder ein anderer permanenter Datenspeicher als Stable Storage betrachtet werden.

- **Zuverlässige Kommunikation**
  Fehlertoleranzverfahren sind i. d. R. darauf angewiesen, dass Kommunikationspartner zuverlässig erreicht werden und keinerlei Daten beim Austausch von Nachrichten unbemerkt verloren gehen. Zwar können physikalische Übertragungskanäle keine zuverlässige Übertragung garantieren, doch können hierzu entsprechende Protokolle eingesetzt werden.[2]

- **Synchrone Uhren**
  Für viele Verfahren ist eine Synchronisation der Uhren der Komponenten eines verteilten Systems erforderlich. Hierfür existieren eine Reihe von Verfahren [29, 80]. Da allgemein keine absolute Uhrensynchronisation erreicht werden kann, werden i. d. R. Verfahren angewendet, die gewährleisten, dass die Uhren des Systems maximal um einen definierten Wert „auseinanderlaufen“.

[2] Einer der bekanntesten Vertreter ist das *Transmission Control Protocol (TCP)*, das zuverlässige Übertragungen in IP-basierten Netzwerken bereitstellt.

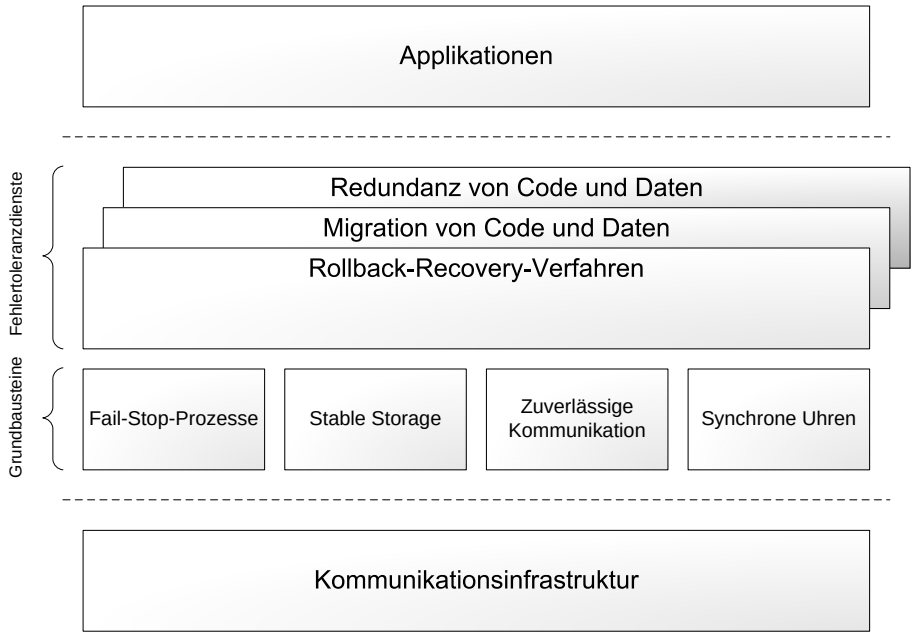


Abb. 5.2.: Ebenen der Fehlertoleranz.

6. Konzeptionelle Überlegungen zur Fehlertoleranz

In diesem Kapitel wird die Beschaffenheit des im ersten Teil der Arbeit entwickelten Systems untersucht und modelliert. Basierend auf den gewonnenen Erkenntnissen werden schließlich Ansätze aufgezeigt, mit denen das System um dedizierte Fehlertoleranzmerkmale erweitert werden kann.

6.1. Systemmodell

Bevor konkrete Fehlertoleranzkonzepte für das im ersten Teil der Arbeit vorgestellte System entwickelt werden können, müssen Modelle definiert werden, die den folgenden Überlegungen als Grundlage dienen. Diese Modelle sind ein *Infrastrukturmodell*, das die Topologie des Systems beschreibt, sowie ein *Fehlermodell*, das definiert, welche Ausprägungen von Fehlern bzw. Störungen im System erwartet werden.

Im Folgenden betrachten wir das verteilte System $S = (T, P, C)$, das sich zusammensetzt aus den Transceivern $T = \{t_1, \ldots, t_n\}$, den Proxies $P = \{p_1, \ldots, p_n\}$, sowie den Control Points $C = \{c_1, \ldots, c_n\}$. Die Topologie dieses Systems ist in Abbildung 6.1 dargestellt: Zu jedem Transceiver t existiert ein Proxy p, während Control Points an sich in beliebiger Anzahl vorhanden sein können. Die Proxies werden von den Application Managern $AM = \{am_1, \ldots, am_m\}$ ausgeführt und sind untereinander verbunden durch Kommunikationsverbindungen $V_{pp} \subseteq P^2$, die durch die Middleware Universal Plug and Play bereitgestellt werden. Weiterhin existieren Kommunikationsverbindungen $V_{cp} \subseteq C \times P$ zwischen Control Points und Proxies, die ebenfalls Middleware-basiert sind. Zuletzt existieren Kommunikationsverbindungen $V_{pt} \subseteq \{(p_i, l_i)\}$ zwischen jeweils einem Proxy und dem jeweils zugehörigen Transceiver.[1]

Bei dem betrachteten System handelt es sich um ein *Message-Passing System*. In jedem Transceiver t_i, jedem Proxy p_i, sowie jedem Control Point c_i können zwei Arten von Ereignissen auftreten: *Sendeereignisse* und *Empfangsereignisse*. Diese Ereignisse können den jeweiligen Zustand der Komponenten t_i, p_i und c_i verändern. Damit ist der Zustand einer Komponente definiert durch ihren Anfangszustand und der Sequenz aus Kommunikationsereignissen. Sende- und Empfangsereignisse haben eine Reihenfolge, die auf Grundlage von Lamport's *Happened-Before Relation* [54] bestimmt wird. Zwischen einem Empfangs-

[1] Diese direkten Verbindungen zwischen Transceivern und Proxies sind *logische* Verbindungen. Es ist durchaus denkbar, dass bzgl. der Vermittlungsschicht eine vermaschte Netztopologie zugrunde liegt.

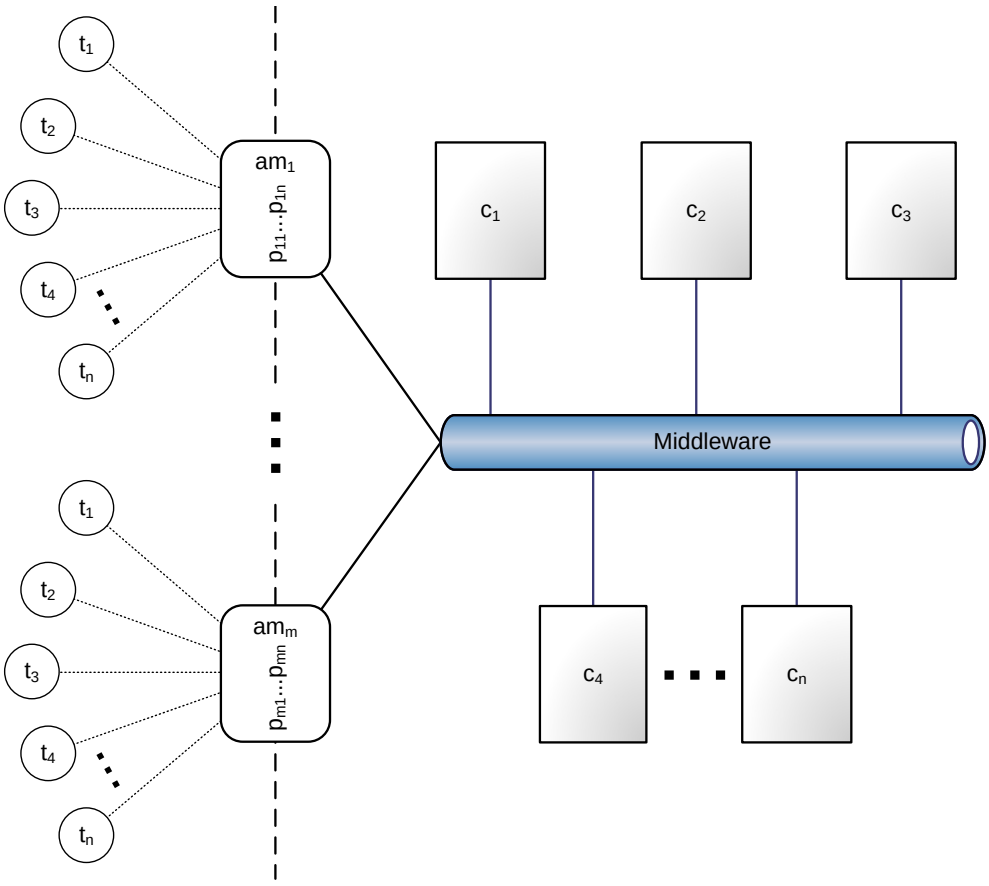


Abb. 6.1.: Darstellung des Systemmodells.

und einem Sendeereignis ist das Verhalten der Komponenten deterministisch – es gilt damit die *Piecewise Deterministic Assumption (PWD)* [27].

Jeder Transceiver stellt neben seinem Proxy einen auf die Applikation des Transceivers zugeschnittenen *Specific Control Point* (siehe Kapitel 1) bereit. Neben einer Anpassung an die Applikation kann der Specific Control Point auch spezielle Erweiterungen, wie eine Unterstützung für Rollback-Recovery-Protokolle, enthalten. Transceiver können das System spontan verlassen und einen Standby-Modus betreten, um Energie zu sparen. In dieser Zeit bleibt der Proxy für seinen Transceiver aktiv und beantwortet Anfragen von Control Points sofern möglich.

Ein weiteres Charakteristikum des betrachteten Systems ist seine typische Verwendung in *In-Haus-Netzwerken*. Daraus ergibt sich, dass Nachrichten, die an Transceiver geschickt werden, üblicher Weise einfach aufgebaute Kommandonachrichten zur Steuerung von Geräten sind. Eine derartige Nachricht hat eine durchschnittliche Länge von 20 - 40 Bytes und besteht aus einem Kommando und einem einfachen Satz an Parametern.

6.2. Verfügbarkeit der Grundbausteine

Bezüglich der in Abschnitt 5.2 vorgestellten Bausteine für fehlertolerante Systeme gelten folgende Annahmen, die in Tabelle 6.1 zusammengefasst werden:

- Es wird davon ausgegangen, dass sämtliche Komponenten des betrachteten Systems dem *Fail-Stop-Modell* entsprechen. Daher wird im Folgenden der Begriff „Ausfall“ als Synonym für das Versagen einer Komponente verwendet.
- In der Regel werden Application Manager auf leistungsfähigen Hosts wie z. B. PCs ausgeführt. Es wird daher vorausgesetzt, dass mindestens ein Application Manager im System ein *Stable Storage* bereit stellt, das jeder Komponente des Systems zur Ablage von Daten zur Verfügung steht.[2]
- Die Kommunikationsverbindungen V_{pp}, V_{cp} und V_{pt} werden als prinzipiell zuverlässig angenommen, da sie alle auf dem *Transmission Control Protocol (TCP)* basieren. Zwar setzt die für V_{pp} und V_{cp} verwendete UPnP-Middleware auch das unzuverlässige *User Datagram Protocol (UDP)* ein, doch nutzen die für den Austausch von Kontrollnachrichten und Ereignissen verwendeten Komponenten des UPnP-Stacks das TCP-Protokoll, wie in Abbildung 6.2 illustriert.
- Eine Synchronisation von Uhren ist i. d. R. nur mit viel Aufwand zu realisieren (Hardwareanforderungen an die Zeitgeber, Unterstützung von Synchronisationsprotokollen). Daher wird im Folgenden davon ausgegangen, dass synchrone Uhren im betrachteten System *nicht* zur Verfügung stehen.

[2] Hier wird ein Datenspeicher auf einer Festplatte als Stable Storage betrachtet. In der Praxis kann auch ein RAID-System verwendet werden.

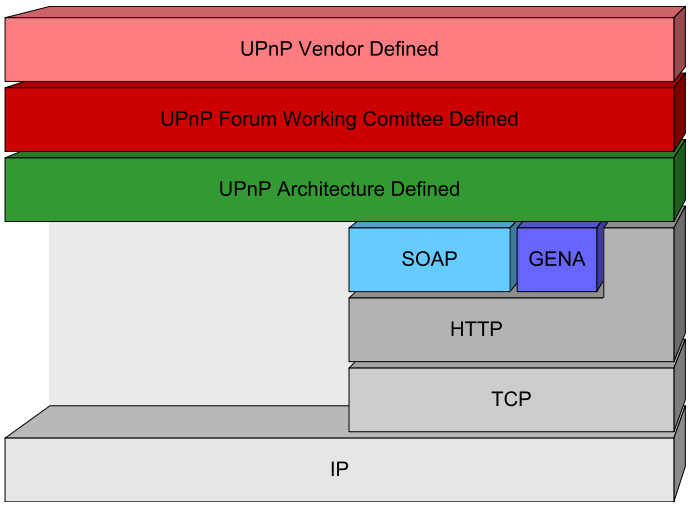


Abb. 6.2.: Für die Kommunikation auf Applikationsebene über SOAP und GENA verwendet UPnP das zuverlässige TCP-Protokoll.

Grundbaustein	Verfügbarkeit	Anmerkung
Fail-stop-Verhalten	verfügbar	Wird als gegeben angenommen. Durch Java Exception-Modell unterstützt.
Stable Storage	verfügbar	Zum Beispiel UPnP-basierter Stable Storage-Service denkbar.
Zuverlässige Kommunikation	verfügbar	UPnP verwendet für die wesentliche Kommunikation das TCP-Protokoll.
Synchrone Uhren	nicht verfügbar	Nur aufwendig zu realisieren.

Tab. 6.1.: Verfügbarkeit der Fehlertoleranz-Grundbausteine.

6.3. Fehlermodell

Im Folgenden wird diskutiert, welche Arten von Fehlern im betrachteten System auftreten können und welche Auswirkungen diese nach sich ziehen können. Folgende Ausfallszenarien werden im Fehlermodell berücksichtigt:

- **Ausfall des Transceivers**
  Der Ausfall des Transceivers muss prinzipiell akzeptiert werden, da der Transceiver selbst kein fehlertolerantes System ist. Entsprechend dem *Fail-Stop-Modell* wird bei Ausfall des Transceivers t_i der zugehörige Proxy p_i automatisch beendet. Dieses Merkmal ist bereits Design-inhärent in der entwickelten Infrastruktur verankert und muss daher nicht durch dedizierte Verfahren erreicht werden.

- **Ausfall des Proxys**
  Ein Proxy p_i kann als Folge des Ausfalls von t_i beendet werden oder auch unabhängig von t_i ausfallen. Wird p_i als Folge des Ausfalls von t_i beendet, so handelt es sich *nicht* um eine Störung. Im anderen Fall wird der Proxy von dem selben oder einem anderen Application Manager neu gestartet, sobald der Ausfall des Proxys detektiert wurde und der Transceiver seinen Zustand als *ungebunden* signalisiert hat. Auch dieses Merkmal ist bereits Design-inhärent in der Infrastruktur vorhanden.
  Es ist jedoch möglich, dass mit dem Ausfall des Proxys Zustandsinformationen, die nur in p_i und nicht in t_i enthalten waren, verloren gehen. Sofern die Zustände von t_i und p_i abgeglichen wurden, bevor p_i ausfällt, wird bei einem Neustart von p_i der aktuelle Zustand automatisch von t_i auf p_i übertragen und ist somit mit dem Rest des Systems konsistent. Falls nicht, entstehen Inkonsistenzen zwischen den Zuständen der Kommunikationspartner von p_i, wie andere Proxies oder Control Points.

- **Ausfall des Application Managers**
  Wenn ein Application Manager bzw. der Host, der den Application Manager ausführt, ausfällt, sind davon sämtliche Proxies betroffen, die von diesem Application Manager verwaltet bzw. auf dem Host ausgeführt werden. Im Prinzip handelt es sich also beim Ausfall eines Application Managers um einen mehrfachen Proxy-Ausfall. In einer solchen Situation greift derselbe Ansatz wie oben, wonach die betroffenen Proxies automatisch von anderen Application Managern neu gestartet werden. Natürlich besteht hier ebenfalls das Problem möglicher Inkonsistenzen beim Neustart der Proxies, sofern die Zustände mit den jeweiligen Transceivern nicht abgeglichen wurden.

- **Ausfall des Control Points**
  Ein Control Point kann prinzipiell jede Applikation sein, die den Dienst eines Proxys bzw. Transceivers nutzt. Jede dieser Applikationen kann nach dem Fail-Stop-Modell ausfallen. Da Control Points jedoch per Definition lediglich Nutzer eines Dienstes sind, erzeugt ihr Ausfall keine weiteren Abhängigkeiten, wie es bei Ausfall eines Proxys der Fall wäre. Allerdings ist der Control Point neben dem Transceiver in vielen Fällen eine Schnittstelle zur Außenwelt bzw. zum Benutzer. Daher ist es wichtig, dass ein nach einem Ausfall neu gestarteter Control Point mit dem Rest des Systems

konsistent ist. Es ist allerdings davon auszugehen, dass die Herstellung dieser Konsistenz von der verwendeten Middleware gewährleistet wird, da diese dafür sorgt, dass der Zustand der Dienstanbieter beim Start entsprechend abgefragt wird. Inkonsistenzen können jedoch dann entstehen, wenn die Applikation, die den Control Point enthält, auch selbst wieder Dienstanbieter ist und applikationsspezifische Statusinformationen existieren, die bei einem Ausfall der Komponente verloren gehen.

- **Ausfall von Kommunikationsverbindungen**
  Im Prinzip kann jede Kommunikationsverbindung im System ausfallen. Im Folgenden gilt auch hier das Fail-Stop-Modell. Ferner wird der Ausfall einer Kommunikationsverbindung zwischen zwei Komponenten als Ausfall der sendenden Komponente betrachtet, wodurch der Ausfall von Kommunikationsverbindungen nicht gesondert behandelt werden muss.

Wenn eine Komponente des Systems eine Nachricht an eine andere Komponente sendet, diese andere Komponente jedoch aufgrund einer Störung nicht antwortet, so wird der Verbindungsversuch nach einem Timeout abgebrochen. Erst in diesem Moment wird die Störung bemerkt. UPnP-basierte Komponenten des Systems, wie Transceiver und Proxies, realisieren über das Senden von Advertisements einen *Heartbeat-Mechanismus*. Das heißt, eine ausgefallene UPnP-Komponente wird nach Ablauf ihres zuletzt gesendeten Advertisement automatisch bemerkt.

6.4. Realisierungsansätze

Im Prinzip drehen sich die im vorherigen Abschnitt betrachteten Ausfallszenarien neben dem Ausfall von Control Points um den Ausfall von Proxies, da Proxies entweder eigenständig oder als Folge des Ausfalls ihres Transceivers oder als Folge des Ausfalls ihres Application Managers ausfallen können. In manchen Fällen ist der drohende Ausfall von Proxies vorhersehbar und kann u. U. vermieden werden. In vielen Fällen sind Ausfälle jedoch unvorhersehbar bzw. müssen hingenommen werden. Allerdings ist in diesen Fällen ein angemessener Umgang mit den Konsequenzen notwendig. Hierfür sind folgende Ansätze denkbar, die im Rahmen der vorliegenden Arbeit verfolgt wurden:

- **Migration von Proxies**
  Sofern ein drohender Ausfall eines Proxys erkannt wird – z. B., wenn die Energiereserven des Hosts beinahe aufgebraucht sind oder die CPU- bzw. Speicherauslastung an eine vorgegebene Grenze stößt – kann der Proxy zur Laufzeit auf einen anderen Host bzw. einen anderen Application Manager *migriert* werden.

- **Rollback-Recovery von Proxies und Control Points**
  Sofern ein Proxy nicht durch Migration „gerettet“ werden konnte, muss er im System neu gestartet werden. Hierbei ist essentiell, dass sich das Gesamtsystem nach den Neustart des Proxys wieder in einem konsistenten Zustand befindet. Um dies zu gewährleisten, können Rollback-Recovery Protokolle verwendet werden. Allerdings ist es wünschenswert, die Transceiver durch Anwendung von Rollback-Reco-

very-Verfahren nicht zusätzlich zu belasten. Im Idealfall braucht ein Transceiver das angewendete Rollback-Recovery-Protokoll selbst gar nicht unterstützen.

Die Migration von Proxies wird im folgenden Kapitel 7 behandelt, während Rollback-Recovery-Verfahren in Kapitel 8 untersucht werden. Darüber hinaus stehen weitere Ansätze zur Verfügung, die jedoch über den Rahmen der vorliegenden Arbeit hinaus gehen und nicht eingehend untersucht wurden:

- **Transparenter Austausch von Transceivern**
  Wenn ein Transceiver ausfällt, kann der Ausfall u. U. vom dahinter liegenden Proxy vor dem Rest des Systems für einen gewissen Zeitraum verborgen werden. Es ist denkbar, in der Zwischenzeit den Transceiver gegen ein gleichwertiges Gerät auszutauschen, bzw. auf ein alternatives Gerät umzuschalten. Anschließend muss der „neue“ Transceiver mit dem Status des Proxys synchronisiert werden. Diese Möglichkeit ist allerdings stark applikationsabhängig. Weiterhin ist der Vorgang des Austauschs des Transceiver ein interner Vorgang im Subsystem bestehend aus Proxy und Transceiver. Daher kann dieser Vorgang nur schwer im Rahmen einer allgemeinen Lebenszyklusverwaltung von Proxies unterstützt werden.

- **Cloning von Proxies**
  Eine zusätzliche bzw. zur Migration von Proxies alternative Möglichkeit besteht darin, redundante Kopien von Proxies auf anderen Hosts vorzuhalten. Allerdings ist hier speziell zu berücksichtigen, dass stets nur ein aktiver Proxy für die Kommunikation mit dem Transceiver zuständig sein kann. Ferner müssen geeignete Synchronisationsmechanismen angewendet werden, um den aktiven Proxy mit seinen Klonen abzugleichen [67]. Weiterhin ist der Ansatz des Cloning zu den anderen vorgestellten Verfahren orthogonal und kann als mögliche Erweiterung der vorliegenden Arbeit betrachtet werden.

7. Migration von Geräte-Proxies

Im Rahmen der vorliegenden Arbeit wird die Migration von Geräte-Proxies als Mittel zur *Fehlervermeidung* diskutiert. In diesem Kapitel werden die der Migration zugrundeliegenden Konzepte und Technologien vorgestellt. Es ist ferner eine Java-basierte Referenzimplementierung entstanden, deren Leistungsfähigkeit anhand von Messungen evaluiert wurde.

Für eine erfolgreiche Anwendung von Proxy-Migration ist die Früherkennung von Ausfällen essentiell, da Proxies migriert werden müssen, *bevor* ihr Host ausfällt. In vielen Fällen deutet sich ein drohender Ausfall frühzeitig an:

- Die Ressourcen des Hosts des Application Managers erreichen eine kritische Stufe. So kann z. B. die durchschnittliche CPU-Auslastung oder der zur Verfügung stehende Arbeitsspeicher in einem vorgegebenen Zeitfenster einen kritischen Wert erreichen. Ferner kann im Application Manager eine Störung auftreten, die durch den Exception-Mechanismus abgefangen werden konnte, der Application Manager sich jedoch in einem potentiell instabilen Zustand befindet.
- Der Benutzer fährt den Host des Application Managers ordnungsgemäß herunter. Sobald der Application Manager vom Betriebssystem das Signal zum Beenden erhält, kann er über einen *Shutdown Hook*[1] einen Vorgang ausführen, der die momentan ausgeführten Proxies auf einen anderen Host „evakuiert“. Dieser Vorgang kann z. B. auch in Gang gesetzt werden, wenn ein Host mit unterbrechungsfreier Stromversorgung (USV) bei einem Stromausfall das Signal zum Herunterfahren des Systems erhält.[2]

Neben diesen einfachen Indizien bevorstehender Ausfälle sind auch komplexere Verfahren zur Früherkennung denkbar. So ist der Einsatz statistischer Verfahren möglich (Host wird regelmäßig zu einer bestimmten Uhrzeit herunter gefahren, die Belastung eines Hosts steigt zu bestimmten Zeiten aufgrund einer durchgeführten Datensicherung stark an, etc.). Darüber hinaus ist auch der Einsatz von *Byzantine-Agreement-Protokollen* denkbar (siehe Abschnitt 5.2), um ein arbitrates Verhalten eines fehlerhaften Application Managers zu erkennen. Allerdings müssen hierzu externe Überwachungsinstanzen eingesetzt werden, was den Implementierungsaufwand deutlich erhöht.

[1] Ein *Shutdown Hook* ist ein Thread, der beim Beenden einer virtuellen Maschine automatisch ausgeführt wird und i. d. R. Cleanup-Funktionen enthält.

[2] Dies setzt natürlich voraus, dass die Netzwerkinfrastruktur nicht von dem Stromausfall betroffen oder selbst durch eine USV abgesichert ist.

Sofern ein bevorstehender Ausfall des Application Managers erkannt wurde, können Proxies *zur Laufzeit* auf einen anderen Host bzw. Application Manager *migriert* werden. Bei diesem Vorgang wird die Ausführung eines Proxys kurz unterbrochen, während *Code und Daten* des Proxys auf einen anderen Host bzw. in eine andere virtuelle Maschine verschoben und dort die Ausführung fortgesetzt wird. Dieser Vorgang wird in Abbildung 7.1 veranschaulicht. Wie in der Abbildung dargestellt, ist allgemein zu beachten, dass Verbindungen zwischen den Proxies und ihren Kommunikationspartnern entsprechend „umgebogen“ werden müssen. Somit sind an die Migration von Proxies folgende konkrete Anforderungen zu stellen:

1. **Detektion des aktuellen Gerätezustandes**
   Es müssen Verfahren verfügbar sein, um aus dem aktuellen Zustand des Gerätes auf sein bevorstehendes Versagen zu schließen. Dazu muss der aktuelle Zustand ermittelt werden können. Mögliche Ansätze hierzu wurden bereits in Abschnitt 4.2 diskutiert. Um einen drohenden Ausfall im Systemmodell zu berücksichtigen, wird das Fail-Stop-Modell zum sog. *Fail-stutter Fault Modell* erweitert [96]. Dieses Modell berücksichtigt einen Zustand, in dem eine Komponente zwar noch funktioniert, jedoch z. B. aufgrund mangelnder Ressourcen eine unzureichende Performance aufweist.

2. **Signalisierung des Migrationsvorgangs**
   Es muss möglich sein, Kommunikationpartner eines zu migrierenden Proxys (andere Proxies oder Control Points) über den Vorgang der Migration zu informieren, damit diese den Vorgang nicht mit einer Störung des Proxys verwechseln.

3. **Verschieben von Code und Daten des Proxys**
   Das Programm des Proxys muss jederzeit unterbrochen werden können. Weiterhin müssen Code und Daten des Proxys auf einen anderen Host verschoben und dort an der Stelle, an der das Programm unterbrochen wurde, erneut zur Ausführung gebracht werden können.

4. **Umgang mit lokalen Referenzen**
   Sofern lokale Komponenten über Referenzen direkt mit dem zu migrierenden Proxy kommunizieren, müssen diese Referenzen geeignet behandelt werden, sodass sie nach erfolgter Migration nicht „ins Leere“ zeigen und nach Möglichkeit sogar weiter verwendbar bleiben.

Im Folgenden wird im Detail vorgestellt, auf welche Weise diesen Anforderungen entsprochen werden kann.

7.1. Signalisierung des Migrationsvorgangs

Die Durchführung der Migration nimmt eine gewisse Zeit in Anspruch, während dieser der Proxy bzw. der Dienst des Proxys den Control Points nicht zur Verfügung steht. Dieser Zustand kann von einem Control Point fälschlicher Weise als Ausfall des Proxys gewertet

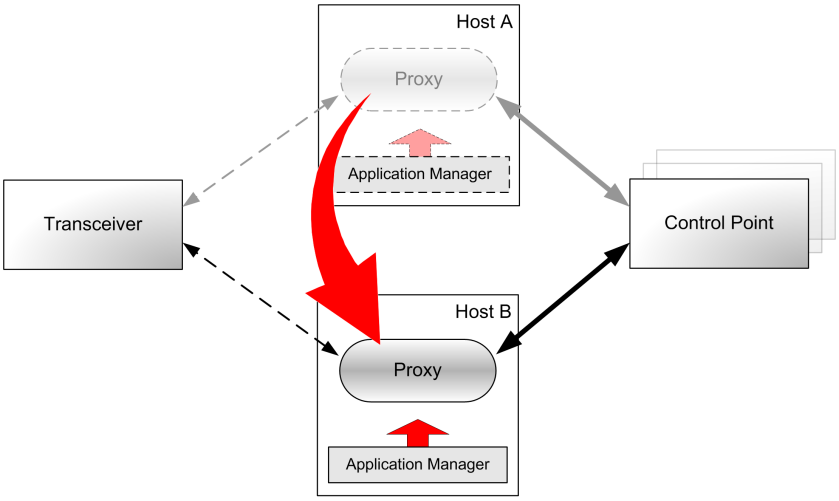


Abb. 7.1.: Migrationvorgang eines Proxys.

werden. Daher kann ein Control Point eine Störung melden, obwohl es sich bei dem Vorgang lediglich um eine Migration handelt. Hinzu kommt, dass sich bei einer Migration die Netzwerk-Adresse des Proxys ändert. So kann ein Proxy zwar seine UUID[3] beibehalten, doch wird sich auf jeden Fall die IP-Adresse des Proxys ändern. Laut Spezifikation darf ein Standard-konformes UPnP-Device allerdings nicht spontan seine IP-Adresse ändern, sondern muss seinen Dienst vor Änderung der IP ordnungsgemäß abmelden (*Byebye-Message*) und ihn mit geänderter IP erneut anmelden [98]. Insbesondere hierdurch würde eine Migration von den Control Points fälschlicher Weise für ein Abmelden des Proxys aus dem System gehalten.

Um Fehlinterpretationen des Migrationsvorganges zu vermeiden, ist eine *Migrations-Signalisierung* erforderlich. Das heißt, ein Proxy teilt seinen Kommunikationspartnern vor der Migration mit, dass er das System nicht permanent verlassen wird, sondern seine folgende Ab- und Anmeldung zu dem ordnungsgemäßen Migrationsvorgang gehört. In der im Rahmen der vorliegenden Arbeit entstandenen Referenzimplementierung sieht der Signalisierungsprozess so aus, dass der Proxy eine Ankündigungsnachricht an seine Kommunikationpartner schickt. Diese werten die darauffolgende Byebye-Message als Beginn der Migration und erwarten, dass sich der selbe Proxy in Kürze unter Beibehaltung seiner UUID wieder anmelden wird. Abgebrochen wird dieser Vorgang über einen Timeout.

Neben der Signalisierung der bevorstehenden Migration ist eine Abstimmung mit dem Application Manager erforderlich, der den Proxy in Empfang nehmen soll. Mitunter muss ein geeigneter Application Manager erst gefunden werden. Hierbei können die selben Mechanismen verwendet werden, die bereits bei der Aushandlung der Proxy-Ausführung zum Einsatz kamen (siehe Abschnitt 4.2).

[3]Universally Unique Identifier

7.2. Verschieben von Code und Daten des Proxys

Bei der eigentlichen Migration wird die Programmausführung des Proxys gestoppt, Programmcode und Daten des Proxys auf einen anderen Host verschoben und die Programmausführung wird auf dem neuen Host fortgesetzt. Dieser Vorgang stellt spezielle Anforderungen an die Laufzeitumgebung:

- Die Laufzeitumgebung sollte *plattformunabhängig* sein, da es durchaus denkbar ist, dass es sich bei der Zielplattform der Migration um einen anderen Typ als bei der Quellplattform handelt.
- Die Laufzeitumgebung muss *Serialisationsmechanismen* unterstützen, um Objekte in eine geordnete, transportable Form überführen zu können.

Diese Anforderungen werden von der *Java Virtual Machine (JVM)* erfüllt, während das *Common Runtime Environment* der *.Net-Plattform* nur eingeschränkte Plattformunabhängigkeit bietet.

Bei der Serialisation werden Memberdaten mit Hilfe von *Introspektionsmechanismen*[4] aus Objekten extrahiert und zusammen mit Strukturinformationen in einem Datencontainer abgelegt [88]. Dieser Container kann nun leicht auf einen anderen Host bzw. in eine andere virtuelle Maschine übertragen werden. Am Zielort werden die Memberdaten aus dem Container ausgelesen und mit Hilfe der gespeicherten Strukturinformationen in neu erzeugte Objekte eingepflegt. Sofern zu serialisierende Objekte auf andere Objekte verweisen, werden diese anderen Objekte ebenfalls serialisiert und übertragen. Hier ist jedoch zu beachten, dass ausschließlich *plattformunabhängige* Objekte serialisierbar sind. Beispielsweise ist ein *Socket-Objekt*, dass von einem Proxy zur Kommunikation verwendet wird, nicht serialisierbar.

Da ein Proxy aufgrund seiner Verbindung zum Transceiver auch nicht-serialisierbare Objekte enthält, muss der Proxy vor der eigentlichen Migration in einen serialisierbaren Zustand überführt werden.[5] Das heißt, er muss eigenständig alle in ihm enthaltenen nicht-serialisierbaren Objekte terminieren, bzw. Referenzen zu anderen nicht-serialisierbaren Strukturen zurückziehen. Dies beinhaltet auch einen Abbau der Kommunikationsverbindung zum Transceiver sowie die Abmeldung von der Middleware-basierten Netzgemeinschaft.

Generell muss bei der Serialisation beachtet werden, dass serialisierte Daten ausschließlich Memberdaten von Objekten sind, jedoch keinerlei Code der entsprechenden Klassen enthalten ist. Folglich muss der Code der jeweiligen Klassen auf dem Zielhost verfügbar

[4] Introspektion ist die Fähigkeit einer VM, Objekte zur Laufzeit zu analysieren. Diese Funktion wird nicht von allen Laufzeitumgebungen unterstützt, da sie einen relativ hohen Ressourcenbedarf hat. So ist die VM der *Connected Limited Device Configuration (CLDC)* der Java 2 Micro Edition nicht introspektionsfähig.

[5] Allgemein ist ein kooperatives Verhalten der Proxies notwendig, da der Thread eines Proxys nicht einfach angehalten werden kann. So ist die Methode Thread.stop() in der Java API als *deprecated* markiert, da ihr Aufruf nur schwer überschaubare Folgen hat. Die Verwendung der Methode Thread.Abort() wird unter .Net aus dem selben Grund nicht empfohlen.

sein, damit die Objekte wiederhergestellt werden können. Java kennt einen *Codebase-Mechanismus*, der den normalen Klassenpfad um Remote-Adressen erweitert. Im Rahmen des betrachteten Systems bietet sich hierzu jedoch die Verwendung des verteilten Code-Repositorys an (siehe Abschnitt 4.3). Das heißt, der Code eines zu migrierenden Proxys wird mit Hilfe des verteilten Code-Repositorys auf dem Zielhost verfügbar gemacht.

7.3. Beispielablauf

Die Migration eines Proxys gliedert sich in folgende Schritte, die im Sequenzdiagramm in Abbildung 7.2 dargestellt sind:

1. **Abstimmung mit dem Zielhost**
   Der Quell-AM ermittelt entsprechend der in Abschnitt 4.2 vorgestellten Verfahren einen geeigneten Zielhost und vereinbart mit dem Ziel-AM eine Übernahme des Proxys.

2. **Stoppen des Proxys**
   Der laufende Proxy erhält ein Signal, sämtliche Aktivitäten einzustellen, sodass sich sein Zustand nicht mehr verändert. Ab diesem Zeitpunkt nimmt er keine Anfragen von seinem Transceiver bzw. von Control Points mehr entgegen.

3. **Signallisierung der Migration und Abmelden des Proxys**
   Der Proxy meldet seinen Dienst im Netzwerk ab. Zuvor signalisiert er seinen Kommunikationspartnern, dass er die Netzgemeinschaft nur kurzzeitig (d. h. für die Dauer der Migration) verlassen wird.

4. **Serialisation des Proxys**
   Der Proxy überführt sich selbst in einen serialisierbaren Zustand. Anschließend wird er serialisiert.

5. **Übertragen der serialisierten Daten**
   Die serialisierten Daten des Proxys werden zum Zielhost übertragen. Der Zielhost fordert vor der folgenden De-Serialisierung den Klassencode des Proxys vom Quell-Host an. Hierzu wird das Repository des sendenden Application Managers verwendet.

6. **Deserialisierung des Proxys**
   Auf dem Zielhost wird der empfangene Code deserialisiert und ein neuer, lauffähiger Proxy erzeugt, dessen Zustand dem direkt vor der Serialisierung entspricht.

7. **Erneutes Anmelden des Dienstes**
   Der Proxy meldet seinen Dienst durch das Versenden von Notify-Messages erneut an. Dabei behält er seine UUID, während sich die IP-Adresse durch den gewechselten Host geändert hat.

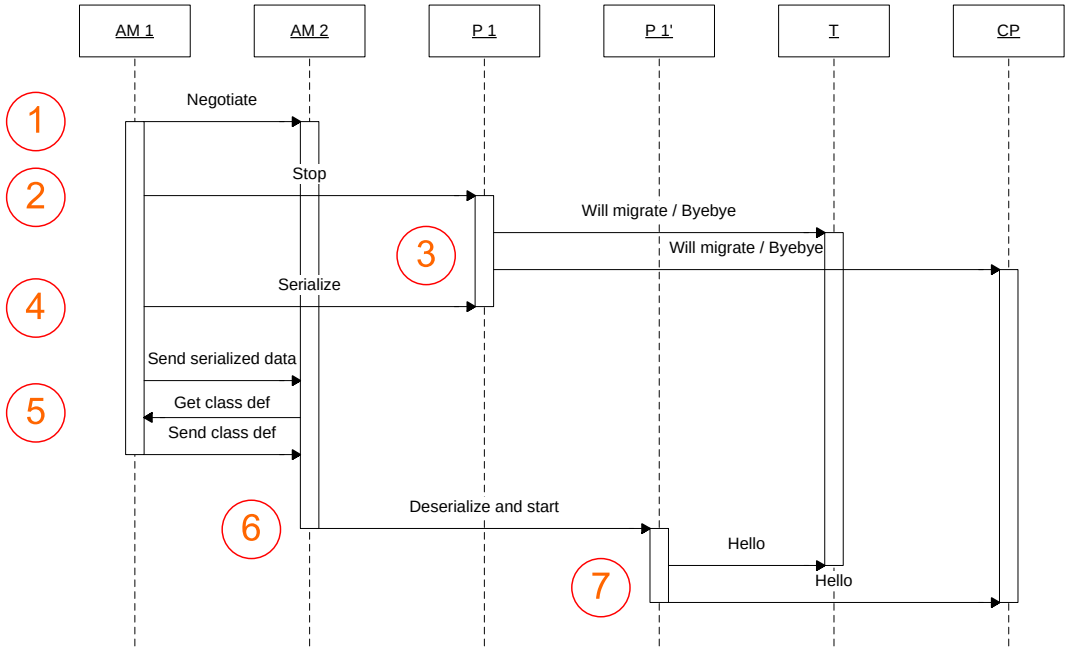


Abb. 7.2.: Sequenzdiagramm der Migration eines Proxys.

7.4. Umgang mit lokalen Referenzen

Sofern Proxies als Threads mit anderen Komponenten in einer gemeinsamen virtuellen Maschine ausgeführt werden, können lokale Referenzen zwischen den anderen Komponenten und den Proxies existieren. Wenn ein Proxy auf einen anderen Host migriert wird, werden Referenzen, die vom Proxy ausgehen, automatisch abgebaut, wenn er auf die Serialisation vorbereitet wird. Referenzen, die von anderen Komponenten auf den Proxy weisen, führen jedoch ins Leere, wenn der Proxy ohne weiteres entfernt wird. Nun bestünde ein Lösungsansatz darin, die äußeren Komponenten vor der Migration zu informieren, damit sie ihre Referenzen zurückziehen können. Es sind jedoch Situationen denkbar, in denen eine äußere Komponente auf eine Zugriffsmöglichkeit auf den Proxy angewiesen ist und nicht mehr funktionieren könnte, würde ihr diese Zugriffsmöglichkeit entzogen.

Dieses Problem kann gelöst werden, indem eine „Weiterleitung“ für migrierte Objekte verwendet wird. Im Rahmen der vorliegenden Arbeit wurden hierzu *Wrapper* eingeführt, die die zu migrierenden Objekte (also die Proxies) umschließen (siehe Abbildung 7.3). Äußere Referenzen weisen ausschließlich auf den Wrapper, der dieselbe Schnittstelle besitzt, wie das in ihm enthaltene Objekt. Der Wrapper reicht einen Methodenaufruf direkt weiter an das in ihm enthaltene Objekt. Wird das Objekt auf einen anderen Host verschoben, so führt der Wrapper statt eines lokalen Aufrufs einen für den ursprünglichen Aufrufer transparenten Remote-Aufruf auf dem nunmehr entfernten Objekt durch. Entsprechend kann der Wrapper etwaige Rückgabewerte an die ursprünglichen Aufrufer vermitteln.

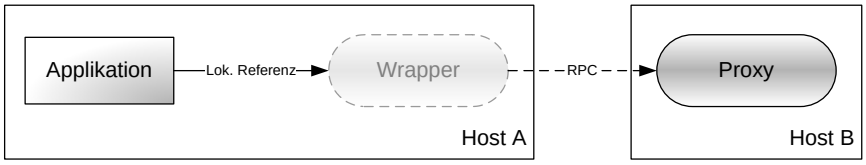


Abb. 7.3.: Weiterleitung lokaler Aufrufe durch den Wrapper.

Für einen entfernten Aufruf sind RPC-Mechanismen notwendig. Bei diesen Mechanismen wird ein Stub-Skeleton-Schema verwendet, um Aufrufe transparent über ein Netzwerk zu senden. Bei Java-basierten Systemen bietet sich hier die Verwendung von *Remote Method Invocation (RMI)* an. Zwar könnte auch UPnP für den Remote-Aufruf verwendet werden, jedoch ist das binäre RMI-Protokoll leistungsfähiger als ein Aufruf über das textbasierte SOAP-Protokoll. Hinzu kommt, dass die RMI-Bibliotheken unter Java den gesamten Vorgang des *Parameter Marshalling* auf transparente Weise übernehmen. Ein letzter wichtiger Punkt ist, dass bei Verwendung von UPnP die Weiterleitungsfunktionen als Dienste im UPnP-Netz sichtbar wären, während eine Weiterleitung per RMI für den Benutzer unsichtbar „Out-of-Band“ geschieht.

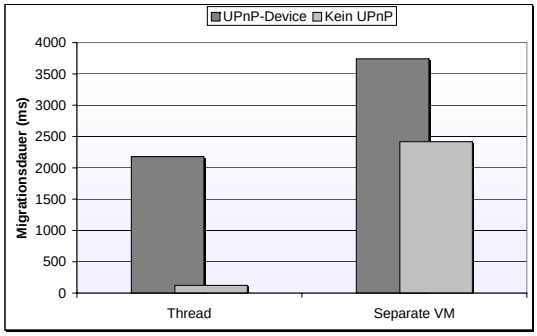
Sobald keine Referenzen von Aufrufern auf den Wrapper mehr existieren, kann der Wrapper entfernt werden. Virtuelle Maschinen wie die JVM nutzen hierzu einen *Garbage Collector*. Der Garbage Collector entfernt Objekte aus dem Speicher, die nicht mehr referenziert werden. Ein Problem stellen hier die Referenzen dar, die der Application Manager zu Verwaltungszwecken auf den Wrapper hält. Solange diese Referenzen bestehen, kann der Wrapper nicht entfernt werden und der Application Manager kann seine Referenzen auf den Wrapper nicht entfernen, da er dann den Kontakt zum Wrapper endgültig verlieren würde. Eine Lösung stellt der Einsatz sog. *schwacher Referenzen*[6] dar. Schwache Referenzen werden vom Garbage Collector ignoriert. Das heißt, der Garbage Collector entfernt alle Objekte aus dem Speicher, auf die keine oder ausschließlich schwache Referenzen existieren. Solange der Application Manager also schwache Referenzen zur Verwaltung des Wrappers verwendet, kann der Garbage Collector den Wrapper entfernen, sobald keine weiteren „echten“ Referenzen mehr auf ihn existieren.

7.5. Migrationsdauer

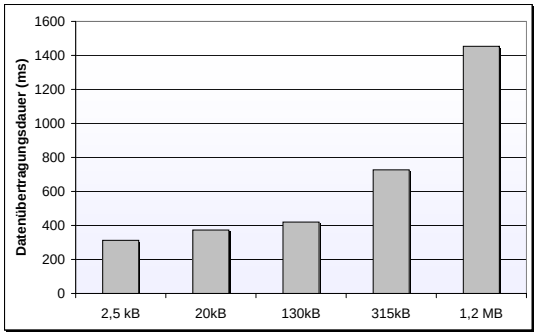
Abbildung 7.4(a) veranschaulicht die Dauer der Migration einer einfachen Java-Anwendung sowie eines aktiven UPnP-Proxys. Für die Messung wurde ein Setup bestehend aus zwei herkömmlichen PCs (1,6 und 2,5 GHz) verwendet, die über ein 100 Mbit/s LAN verbunden wurden. Als Softwareplattform wurde das Java Runtime Environment 6, sowie der im gesamten Rahmen dieser Arbeit eingesetzte Infineon UPnP-Stack verwendet.

[6] Engl.: *Weak References*

Wie sich schon bei der Startzeit von Proxies gezeigt hat (siehe Abbildung 4.5(a) in Abschnitt 4.4), wird die meiste Zeit für die Bereitstellung des UPnP-Dienstes sowie, bei Ausführung als Prozess, für den Start der virtuellen Maschine benötigt. Sofern der Proxy als Thread in der virtuellen Maschine des Application Managers ausgeführt wird, werden für eine Migration ca. 2 Sekunden benötigt. Bei einer Ausführung in einer separaten VM (als Prozess) fallen knapp 4 Sekunden für die Migration an. Aufgrund der hohen Übertragungsgeschwindigkeit des Netzes spielt die Größe des Proxy Codes eine untergeordnete Rolle, wie Abbildung 7.4(b) zu entnehmen ist.



(a) Einfluss von UPnP und Ausführungsart auf die Dauer des Migrationsvorgangs.



(b) Dauer der reinen Datenübertragung.

Abb. 7.4.: Proxy-Migrationsdauer.

Der in diesem Kapitel vorgestellte, recht geradlinige Ansatz der Migration bietet noch Raum zur Optimierung. So ist es denkbar, an in Frage kommenden Zielorten der Migration Klone des Proxys bereit zu halten und bei Bedarf einfach auf einen der Klone „umzuschalten“ [24]. Auf diese Weise wird der Vorgang der Migration beschleunigt, allerdings bedeutet dieser Ansatz auch einen erheblich größeren Verwaltungsaufwand.

8. Rollback-Recovery in Proxy-basierten Systemen

Rollback-Recovery-Protokolle sind ein Mittel zur Wiedererlangung eines integren Gesamtsystems, nachdem einzelne Komponenten versagt und das System damit beeinträchtigt haben. Wie der Name sagt, wird das System durch die Anwendung der Verfahren befähigt, nach dem Auftreten von Störungen in einen gültigen Zustand „zurückzurollen“. Die Hauptaufgabe von Rollback-Recovery-Protokollen ist es dabei, das Auftreten von *Inkonsistenzen* zu vermeiden.

Ein Zustand eines Systems wird als *konsistent* bezeichnet, wenn er bei einem fehlerfreien Betrieb des Systems erreicht werden *kann*[1] [47]. Ein inkonsistenter (und damit im fehlerfreien Betrieb unmöglicher) Systemzustand zeichnet sich durch das Auftreten *verwaister Nachrichten* aus. Eine verwaiste Nachricht ist dadurch gekennzeichnet, dass eine Komponente j eine Nachricht m empfangen hat, obwohl m von keiner anderen Komponente i gesendet wurde. Ein solcher Zustand ist nicht denkbar und darf im System, nachdem ein Rollback-Recovery-Vorgang durchgeführt wurde, keinesfalls auftreten. Empfänger verwaister Nachrichten werden als *verwaiste Prozesse* bezeichnet. *Verlorene Nachrichten* kennzeichnen den umgekehrten Fall: Eine Nachricht m wurde von einer Komponente i gesendet, die von keiner anderen Komponente j empfangen wurde. Verlorene Nachrichten charakterisieren allerdings keinen inkonsistenten Systemzustand, da es theoretisch denkbar ist, dass die Nachricht m den Empfänger *noch nicht* erreicht hat. In diesem Fall wird von einer *In-Transit-Message* gesprochen [27].

Inkonsistenzen in dem in dieser Arbeit betrachteten Proxy-basierten System können dann auftreten, wenn ein Proxy p_i nach einem Ausfall als p_i' neu gestartet wird. Sofern der Transceiver t_i eine aktuelle Kopie des Zustandes von p_i enthält, stellt p_i seinen letzten Zustand mit Hilfe von t_i wieder her. Allerdings kann es sein, dass der auf t_i gespeicherte Zustand nicht dem Zustand von p_i direkt vor seinem Ausfall entspricht. So kann p_i einem anderen Proxy p_j kurz vor seinem Ausfall eine Nachricht gesendet haben. Später wird der neu gestartete p_i' „im Glauben“ sein, diese Nachricht niemals gesendet zu haben – was einen inkonsistenten Systemzustand darstellt.

Prinzipiell ergibt sich bei Rollback-Recovery-Verfahren der Umstand, dass alle Teilnehmer das selbe Protokoll unterstützen müssen. Dies ist ein Problem, da Rollback-Recovery-Protokolle bisher nicht standardisiert sind. Im Umfeld des betrachteten Systems lässt sich jedoch ausnutzen, dass neben den Sindrion-Proxies auch die *Specific Control Points*

[1] Das heißt nicht, dass der Zustand schon einmal aufgetreten sein muss. Es geht lediglich darum, dass er im fehlerfreien Betrieb möglich ist.

(SCPs) von den Transceivern bereitgestellt werden (siehe Kapitel 1) und somit erwartet werden kann, dass die SCPs das selbe Rollback-Recovery-Verfahren einsetzen, wie die Proxies.

> Für Sub-Systeme bzw. Komponenten, die als Einheit am Rollback-Recovery teilnehmen, werden im Folgenden die beiden in der Literatur üblichen Begriffe *Recovery Unit (RU)* bzw. *Prozess* verwendet.
> Für die Darstellung der Abläufe beim Rollback-Recovery wird im Folgenden eine in der Literatur übliche Art von Sequenzdiagrammen verwendet, die sich jedoch von der in der vorliegenden Arbeit bisher verwendeten Art von Sequenzdiagrammen unterscheidet.

8.1. Übersicht gängiger Rollback-Recovery-Protokolle

In diesem Abschnitt wird eine Übersicht gängiger Rollback-Recovery-Protokolle gegeben. Anschließend wird in Abschnitt 8.2 die Entwicklung eines geeigneten Rollback-Recovery-Verfahrens für das im ersten Teil der Arbeit entstandene System beschrieben. Für weitergehende Informationen sei insbesondere auf das Buch von Pankaj Jalote „Fault Tolerance in Distributed Systems“ [47] sowie den Survey von Elnohazy et. al. „A Survey of Rollback-Recovery Protocols in Message-Passing Systems“ [27] verwiesen.

Rollback-Recovery-Protokolle lassen sich in die Klassen der *Checkpoint-basierten* Protokolle und der *Log-basierten* Protokolle gliedern, wie in der Baumübersicht in Abbildung 8.1 gezeigt. Diese beiden Familien werden in den folgenden Abschnitten 8.1.1 und 8.1.2 vorgestellt. In Abschnitt 8.1.3 findet sich schließlich eine Gegenüberstellung der vorgestellten Verfahren. Der Unterschied zwischen den beiden Protokollfamilien liegt darin, dass Checkpoint-basierte Verfahren Zustandsabbilder (Checkpoints) erzeugen, um Komponenten später in zuvor gespeicherte Zustände zurückzuführen, während Log-basierte Verfahren Protokolle der Systemhistorie anlegen und im Fehlerfall auswerten, um das System wieder in einen konsistenten Zustand zu fahren.

Ein Vorgang, der bei allen Verfahren eine zentrale Rolle spielt, ist der sog. *Output Commit*. Von einem Output Commit sprechen wir immer dann, wenn ein Rollback-Recovery-System mit seiner Außenwelt interagiert und diese damit beeinflusst. Das Problem ist, dass die zumeist zustandsbehaftete Außenwelt i. d. R. nicht in der Lage ist, an einem Rollback-Vorgang teilzunehmen. Für Rollback-Recovery-Protokolle bedeutet dies, dass sie in der Lage sein müssen, den Systemzustand unmittelbar nach Beeinflussung der Außenwelt bei einem Rollback wieder anfahren zu können, da es ansonsten zu Inkonsistenzen mit der Außenwelt kommen würde.

8.1.1. Checkpoint-basierte Protokolle

Allgemein gibt es zwei Ansätze für Checkpointing: *Asynchronous Checkpointing* und *Synchronous Checkpointing* (oft auch als *Distributed Checkpointing* bezeichnet). Beim Asyn-

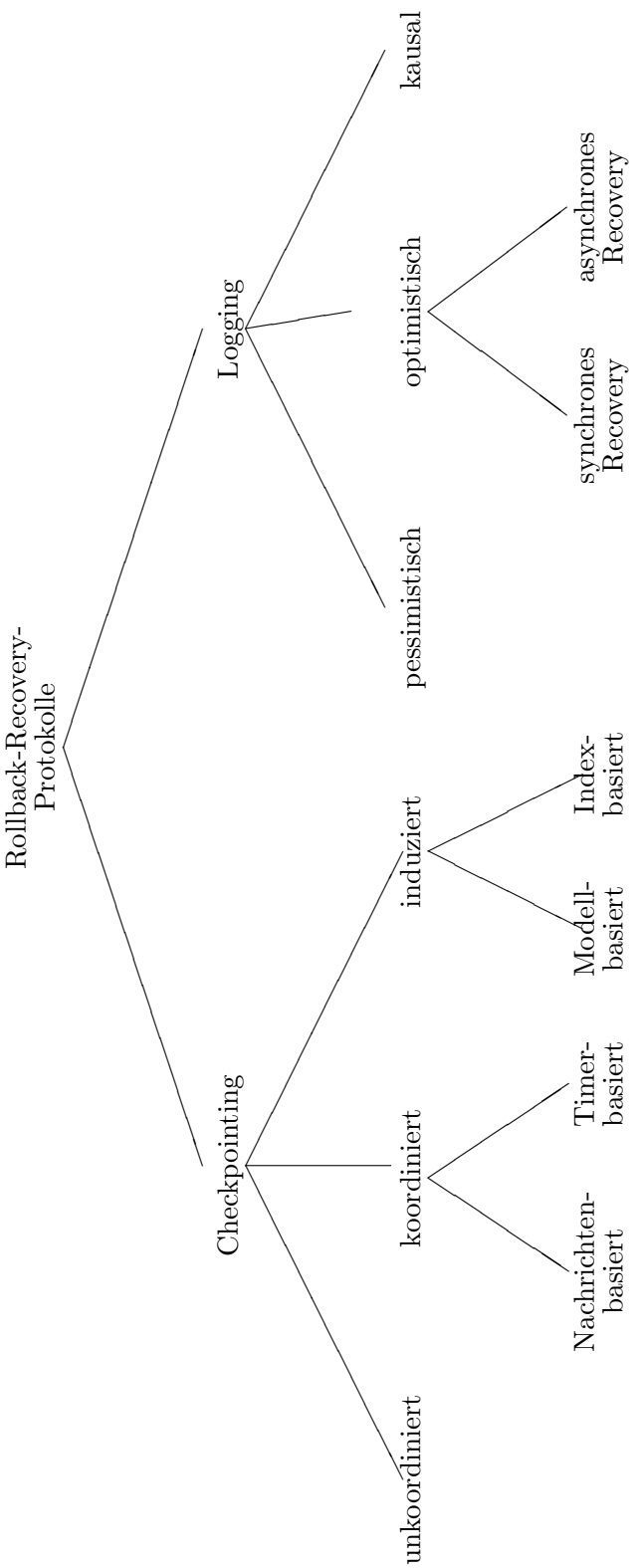


Abb. 8.1.: Übersicht der Rollback-Recovery-Protokolle

chronous Checkpointing erstellen Komponenten Checkpoints unabhängig voneinander, während beim Synchronous Checkpointing die Erstellung der Checkpoints *systemweit* durch einen Zeitpunkt oder ein Ereignis eingeleitet wird.

8.1.1.1. Asynchronous Checkpointing

Beim Asynchronous Checkpointing erstellen Recovery Units ihre Checkpoints vollkommen unabhängig voneinander (asynchron) und legen sie auf einem Stable Storage ab. Dabei wird in Kauf genommen, dass zu dem Zeitpunkt, an dem die Checkpoints erstellt werden, nicht gewährleistet werden kann, dass die Menge der aktuellsten Checkpoints in ihrer Gesamtheit einen konsistenten Systemzustand ergibt. Daher muss bei Einleitung eines *Rollbacks* untersucht werden, *welche* Checkpoints zusammen einen gültigen Systemzustand ergeben und damit eine sog. *Recovery Line* bilden. Hierzu können z. B. *Journale* geführt werden, mit denen verfolgt werden kann, in welcher Weise die im System vorhandenen Recovery Units sich gegenseitig durch den Austausch von Nachrichten beeinflusst haben [47].

Der Hauptnachteil des Asynchronous Checkpointing ist das mögliche Auftreten des *Domino-Effekts*, bei dem unkontrollierte Rollbacks einzelner Prozesse auftreten und das System im ungünstigsten Fall lawinenartig bis in den Anfangszustand „zurückfällt“. Ein weiterer Nachteil ist, dass die Journale relativ groß werden können, wenn weit zurückliegende Systemzustände angefahren werden sollen. Das in Bezug auf die vorliegende Arbeit wohl größte Manko ist jedoch die Unfähigkeit des Asynchronous Checkpointing – aufgrund der vollkommenen Unabhängigkeit der Recovery Units – einen *Output Commit* durchzuführen.

Der Vorteil von Asynchronous Checkpointing liegt in seiner Einfachheit, da keinerlei Koordinierung zwischen den Prozessen erforderlich ist. Allgemein bietet sich Asynchronous Checkpointing an, wenn mit einem seltenen Auftreten von Fehlern zu rechnen ist und zudem relativ selten kommuniziert wird.

Anwendungsbeispiel

Um ein allgemeines Verständnis für die Problematik des Rollback-Recovery zu schaffen, wird im Folgenden ein einfaches Beispiel betrachtet, um den Vorgang des Recovery beim Asynchronous Checkpointing näher zu beleuchten. Wir gehen von einem Setup bestehend aus den Transceivern t_1, t_2 und t_3, sowie den zugehörigen Proxies p_1, p_2 und p_3 aus. Die Transceiver t_1 und t_2 repräsentieren zwei Lichtschalter mit den zugehörigen Proxies p_1 und p_2. t_3 repräsentiert eine Lampe, die von dem Proxy p_3 vertreten wird. Im Beispiel schaltet t_1 die Lampe t_3 ein, während t_2 den veränderten Status von t_3 automatisch anzeigen soll. In diesem Beispiel sei jede Komponente eine eigenständige Recovery Unit. Wir gehen weiterhin davon aus, dass zum Startzeitpunkt jede Komponente einen Initial-Checkpoint C_0 erstellt hat. Jeweils nach Erhalt einer Nachricht wird ein weiterer Checkpoint erstellt. Beim Rollback wird das von Wang et al. vorgestellte Verfahren verwendet [104].

Folgende (vorerst fehlerfreie) Sequenz wird nun, wie in Abbildung 8.2 dargestellt, durchlaufen:

1. Der Benutzer betätigt Schalter 1, worauf t_1 eine Nachricht an seinen Proxy p_1 sendet.
2. Der Proxy p_1 schickt eine Nachricht zum Einschalten der Lampe an p_3.
3. p_3 schaltet die Lampe durch Senden einer Nachricht an t_3 ein.
4. t_3 bestätigt das Einschalten der Lampe gegenüber p_3.
5. p_3 informiert p_1 und p_2 über die Statusänderung per Event.
6. p_1 und p_2 informieren ihre Transceiver über die Statusänderung.

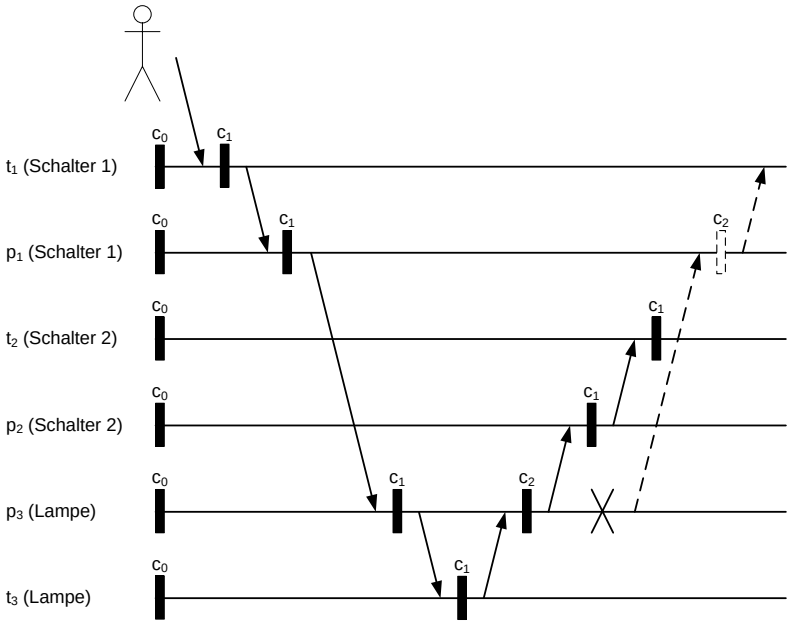
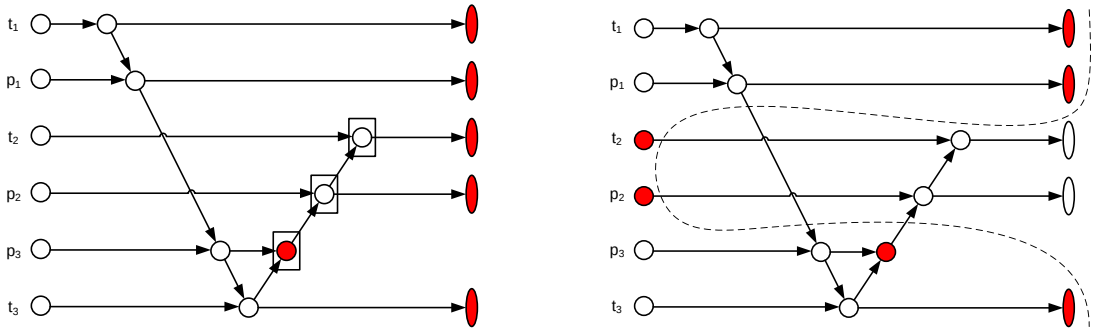


Abb. 8.2.: Sequenzdiagramm für das Anwendungsbeispiel.

Wir nehmen nun an, dass bei p_3 eine Störung Auftritt ("X" in Abbildung 8.2), nachdem p_2 bereits per Event über die Statusänderung informiert wurde, jedoch bevor p_1 informiert werden konnte. Um das System nach dem Neustart von p_3 in einen konsistenten Zustand zu bringen, muss eine Recovery Line gefunden werden, die einen möglichst aktuellen Systemzustand widerspiegelt. Um die Recovery Line zu ermitteln, befolgen wir den in Listing 8.1 dargestellten *Propagationsalgorithmus*. Abbildung 8.3(a) zeigt einen sog. *Checkpoint Graphen* [104], der die Vorgehensweise des Propagationsalgorithmus widerspiegelt. Anschießend ergibt sich die in Abbildung 8.3(b) dargestellte Recovery Line.

Lst. 8.1: Propagationsalgorithmus nach Wang et al.

```
Füge den jeweils letzten Checkpoint von jedem fehlerhaften
Prozess der Menge 'RootSet' zu;

Füge den aktuellen Status von jedem fehlerfreien Prozess
der Menge 'RootSet' zu;

Markiere alle Checkpoints, die von den Elementen der Menge
'RootSet' erreichbar sind;

WHILE (Mindestens ein Element der Menge 'RootSet' ist
       markiert)
{
   Ersetze jedes markierte Element von 'RootSet' durch den
   letzten nicht markierten Checkpoint des selben Prozesses;

   Markiere alle Checkpoints, die von den Elementen der Menge
   'RootSet' erreichbar sind;
}

'RootSet' entspricht der Recovery Line;
```



(a) Ausgangszustand.

(b) Erreichte Recovery Line.

Abb. 8.3.: Checkpoint Graph und erreichte Recovery Line des Anwendungsbeispiels.

8.1.1.2. Synchronous Checkpointing

Beim Synchronous Checkpointing stimmen sich die beteiligten Prozesse bei der Erstellung ihrer Checkpoints ab, um zu gewährleisten, dass der resultierende Systemzustand konsistent ist. Hierzu existieren im Wesentlichen zwei Ansätze:

- **Timer-basierte Synchronisation**
  Die gemeinsame Erstellung eines Checkpoints findet zu festgelegten Zeitpunkten

statt. Dieser Ansatz ist allerdings auf die Verfügbarkeit synchroner Uhren angewiesen. Synchrone Uhren erfordern wiederum Zeitsynchronisationsprotokolle, die garantieren, dass die Uhren im System maximal um einen Zeitraum Δt auseinander laufen. Wenn Recovery Units ihre Checkpoints erstellt haben, so müssen sie damit noch um Δt abwarten, bevor sie ihre Arbeit wieder aufnehmen, um sicher zu stellen, dass alle Recovery Units des Systems ihre Checkpoints erstellt haben. Damit hängt die Performance von synchronem Checkpointing direkt von der Genauigkeit der verwendeten Uhren ab.
Ein Output Commit ist bei diesem Ansatz generell nicht möglich, da die Zeitpunkte, an denen Checkpoints erstellt werden, unabhängig von der jeweiligen Situation im Vorhinein feststehen.

- **Nachrichten-basierte Synchronisation**
  Eine alternative Möglichkeit der Abstimmung besteht darin, dass eine Recovery Unit allen anderen Recovery Units die Erstellung ihres Checkpoints über eine Nachricht signalisiert und damit eine systemweite Checkpoint-Erstellung forciert. Insbesondere wenn ein Output Commit durchgeführt werden soll, ist dieser Ansatz jedoch ungünstig, da jedesmal, wenn eine Recovery Unit mit der Außenwelt kommuniziert, alle anderen Recovery Units im System gezwungen werden, einen Checkpoint zu erstellen.

Der eigentliche Recovery-Vorgang ist beim Synchronous Checkpointing relativ einfach, da lediglich zum letzten globalen Checkpoint zurückgesprungen werden muss und kein Domino-Effekt wie beim Asynchronous Checkpointing auftreten kann.

8.1.1.3. Communication-induced Checkpointing

Communication-induced (induziertes) oder auch *quasi-asynchronous* Checkpointing ist eine Kombination aus Asynchronous Checkpointing und Synchronous Checkpointing. Es vermeidet den Domino-Effekt und gestattet es Prozessen *einige* ihrer Checkpoints (sog. *Local Checkpoints*) unabhängig voneinander zu erstellen [96]. Allerdings können bzw. *müssen* Prozesse gezwungen werden, zusätzliche Checkpoints (sog. *Forced Checkpoints*) zu erzeugen, um eine gültige Recovery Line zu erzielen. Um erkennen zu können, wann die Erstellung von Checkpoints erzwungen werden muss, werden Metadaten auf gesendete Nachrichten aufgeprägt und von den Empfängern entsprechend ausgewertet. Communication-induced Checkpointing existiert im Wesentlichen in zwei Ausprägungen:

- Beim *Model-based Checkpointing* wird bei Erkennung bestimmter Kommunikations-Pattern die Erstellung von Checkpoints erzwungen. Entsprechend erfolgt der Nachrichtenaustausch nach einem vorgegebenen Schema, wodurch diese Kommunikations-Pattern eingehalten werden [103].
- Verfahren nach dem *Index-based Checkpointing* prägen Indizes auf Nachrichten auf und erlauben anhand dieser Indizes eine Erkennung, wann die Erstellung von Checkpoints erzwungen werden muss [18].

Die meisten Verfahren gehen bei der Erzeugung erzwungener Checkpoints prinzipiell konservativ vor und erzeugen mehr Checkpoints, als eigentlich erforderlich wäre. Daher wird allgemein angestrebt, die Zahl der Forced Checkpoints so gering wie möglich zu halten [42].

8.1.2. Log-basierte Protokolle

Log-basierte Protokolle nutzen neben dem bereits beschriebenen Checkpointing sog. *Message Logs*, um Prozesse zu befähigen, einen gültigen Systemzustand vor Auftreten einer Störung wieder „anzufahren“ [4, 3]. Der Vorgang des Anfahrens wird auch *Replay* genannt. Die Anwendung Log-basierter Verfahren hat zwei wesentliche Vorteile:

- **Effizienter Output Commit**
  Die Interaktion mit der Außenwelt (Output Commit) kann effizienter durchgeführt werden, da ein Prozess nach einem Rollback den Zustand wieder anfahren kann, der konsistent mit der Außenwelt ist. Bei einer Anwendung rein Checkpoint-basierter Verfahren müsste vor einem Output Commit jedesmal ein neuer Checkpoint erzeugt werden, was sich aufwendig gestaltet bzw. je nach Verfahren gar nicht möglich ist (siehe Abschnitt 8.1.1).
- **Vermeidung des Domino-Effektes**
  Der Domino-Effekt kann bei Log-basierten Verfahren prinzipiell nicht auftreten. Daher können die Verfahren sehr gut mit dem einfach zu realisierenden Asynchronous Checkpointing ergänzt werden. Ein neu gestarteter Prozess beginnt stets bei seinem letzten Checkpoint und führt dann einen Replay entsprechend der Einträge in seinem Message Log durch.

Voraussetzung für die Anwendung Log-basierter Verfahren ist die Modellierbarkeit des zugrundeliegenden Systems als Folge von *Zustandsintervallen* – deterministischer Abläufe, die jeweils durch nicht-deterministische Ereignisse eingeleitet werden.[2] Der Empfang einer Nachricht ist solch ein nicht-deterministisches Ereignis, das ein neues Zustandsintervall einleitet. Ein Datensatz, der ein Zustandsintervall komplett beschreibt, wird *Determinate* des Zustandsintervalls genannt. Es ist Aufgabe der Message Logs, die Determinanten sämtlicher Zustandsintervalle (und damit sämtlicher empfangener Nachrichten der Prozesse) aufzuzeichnen. Message Logs werden i. A. im flüchtigen Speicher der Prozesse gehalten und gehen bei Ausfall der Prozesse verloren. Determinanten, die hingegen auf einem Stable Storage abgelegt wurden, werden als *stable* bezeichnet.

Es soll nun betrachtet werden, unter welchen Bedingungen verwaiste Prozesse (Inkonsistenzen) im System durch Anwendung von Logging-Verfahren vermieden werden können: Es sei e ein nicht-deterministisches Ereignis eines Prozesses p. Ferner gelte:

- $Depend(e)$ sei die Menge der Prozesse, die nach *Lamport's Happened Before Relation* [54] von e beeinflusst werden. In dieser Menge ist auch p selbst enthalten.

[2] Die Annahme, dass dies zutrifft, wird *Piecewise Deterministic Assumption (PWD)* genannt (siehe Abschnitt 6.1).

- $Log(e)$ sei die Menge der Prozesse, die die Determinante von e in ihrem *flüchtigen* Speicher vorhalten.
- $isStable(e)$ sei ein Prädikat, das dann wahr ist, wenn die Determinante von e auf dem Stable Storage festgehalten wurde.

In einem System sind keine verwaisten Prozesse vorhanden, wenn gilt:

$$\forall e : \neg isStable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

Dies bedeutet: *Wenn im System Ereignisse aufgetreten sind, deren Determinanten nicht auf einem Stable Storage abgelegt sind, dann ist die Menge der Prozesse, die von den Ereignissen abhängig sind, eine Teilmenge der Prozesse, die die Determinanten in ihren Message Logs vorhalten.* Solange diese Eigenschaft zutrifft, sind verwaiste Prozesse im System vermeidbar. Daher wird diese Eigenschaft in der Literatur auch als *Always-No-Orphans Condition* bezeichnet [27]. Anschaulicher ausgedrückt bedeutet sie, dass verwaiste Prozesse vermieden werden können, solange die Determinanten aufgetretener Ereignisse entweder auf dem Stable Storage oder in den Message Logs der Prozesse verfügbar sind.

8.1.2.1. Pessimistic Logging

Pessimistic Logging trägt seinen Namen aufgrund der Annahme, dass Fehler prinzipiell in *jedem* Zustandsintervall auftreten können. Diese Annahme ist pessimistisch, da Fehler in der Regel viel seltener auftreten. Beim Pessimistic Logging wird die Determinante jedes nicht-deterministischen Ereignisses auf dem Stable Storage abgelegt *bevor* das Ereignis den Zustand der Komponente beeinflussen kann. Daher wird Pessimistic Logging oft auch als *Synchronous Logging* bezeichnet. Formal betrachtet wird beim Pessimistic Logging eine striktere Fassung der Always-No-Orphans Condition umgesetzt, da sämtliche Determinanten unmittelbar auf dem Stable Storage abgelegt werden:

$$\forall e : \neg isStable(e) \Rightarrow |Depend(e)| = 0$$

Pessimistic Logging hat den Vorteil, dass es ein vergleichsweise einfach umzusetzendes Verfahren ist. Hinzu kommt, dass für den Output Commit kein spezielles Protokoll eingehalten werden muss. Als Nachteil sind die Latenzzeiten zu nennen, die durch die häufigen synchronen Zugriffe auf das Stable Storage entstehen. Pessimistic Logging wird daher zumeist in Systemen favorisiert, in denen ein geringer Recovery-Overhead wichtiger ist als der normale Laufzeit-Overhead [46].

Beim Recovery stellt ein neu gestarteter Prozess über seinen aktuellsten Checkpoint einen Ausgangszustand her. Anschließend arbeitet er die aufgetretenen nicht-deterministischen Ereignisse mit Hilfe seines Message Logs ab (Replay). Hierbei ist es erforderlich, die nach dem letzten nicht-deterministischen Event zu sendenden Nachrichten *physikalisch* neu zu senden. Alle anderen Sendevorgänge können „intern“ abgehandelt werden,

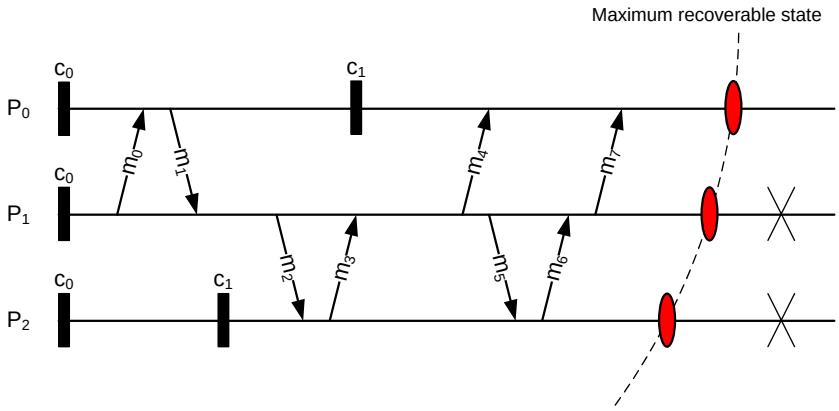


Abb. 8.4.: Beispiel-Sequenzdiagramm für Pessimistic Logging.

ohne sie wirklich zu verschicken. Betrachten wir hierzu Abbildung 8.4 für ein Beispiel: Prozess P_2 stellt nach seinem Neustart den Ausgangszustand C_1 mit Hilfe des letzten Checkpoints wieder her. Anschließend fängt er an, sein Message Log zu durchlaufen, das er vom Stable Storage bezogen hat. Das Log enthält die Determinanten der Nachrichten-Empfangsereignisse von m_2 und m_5. Da die Determinante von m_5 im Log enthalten ist, kann sich der Prozess sicher sein, dass m_3 gesendet wurde, da das Senden von m_3 zum deterministischen Ablauf gehört, der durch das nicht-deterministische Ereignis m_2 eingeleitet wurde. Der Empfang der Nachricht m_5 ist jedoch das letzte Ereignis im Log. Daher kann P_2 nicht wissen, ob m_6 bereits gesendet wurde oder nicht. Folglich muss m_6 physikalisch erneut gesendet werden – selbst auf die Gefahr hin, dass es sich um eine Wiederholung handelt. Mit dieser möglichen Wiederholung muss der Empfänger der Nachricht umgehen können. Das heißt, er muss selbst nachhalten, welche Nachrichten er bereits empfangen hat und welche nicht. Hierzu können z. B. Nachrichten-IDs bzw. Sequenznummern verwendet werden.

8.1.2.2. Optimistic Logging

Im Unterschied zu Pessimistic Logging geht *Optimistic Logging* davon aus, dass Fehler *selten* auftreten bzw., dass der Logging-Vorgang jeweils vor Auftreten eines Fehlers abgeschlossen ist. Daher speichern Optimistic Logging Verfahren ihre Log-Einträge *asynchron* – also während das Ereignis den Prozess bereits verändert – auf dem Stable Storage. Determinanten werden im Message Log im flüchtigen Speicher gehalten, das von Zeit zu Zeit auf das Stable Storage geleert wird. Optimistic Logging nimmt damit in Kauf, dass temporär verwaiste Prozesse entstehen können, stellt jedoch sicher, dass nach einem abgeschlossenen Rollback keine verwaisten Prozesse vorhanden sind.

Der Vorteil des Optimistic Logging ist, dass Applikationen nicht bis zum Abschluss des jeweiligen Logging-Vorganges blockieren und damit keine Latenzzeiten wie beim Pessimistic Logging entstehen. Allerdings ist der Recovery-Vorgang wesentlich komplizierter –

schließlich ist es möglich, dass die in den Message Logs gespeicherte Determinanten beim Ausfall eines Prozesses verloren gehen, *bevor* sie auf dem Stable Storage gesichert werden konnten. Somit müssen Abhängigkeiten zwischen Prozessen, die durch den Austausch von Nachrichten zu Stande kommen, ständig nachgehalten werden und beim Rollback der maximal erreichbare Systemzustand zuerst ermittelt werden, bevor er angefahren werden kann [50, 85].

Ein asynchroner Zugriff auf das Stable Storage ist beim Output Commit allerdings nicht möglich: Hier erfolgt der Zugriff synchron, um Konsistenz mit der Außenwelt garantieren zu können. Hinzu kommt, dass beim Output Commit mitunter die Abstimmung mit anderen Prozessen notwendig ist: Ein Prozess, der die Außenwelt beeinflusst, muss nachvollziehen, welche anderen Prozesse *indirekt* an dem Output Commit beteiligt sind. Diese Prozesse müssen benachrichtigt werden, damit sie ebenfalls eine Sicherung auf dem Stable Storage durchführen. So muss in dem in Abbildung 8.5 dargestellten Beispiel der Prozess P_0, der zum Zeitpunkt X einen Output Commit durchführt, nicht nur die Nachrichten m_4 und m_7 selbst auf dem Stable Storage sichern – er muss auch Prozess P_2 auffordern, die Nachrichten m_2 und m_5 zu sichern, um den globalen Systemzustand garantiert wiederherstellbar zu machen.

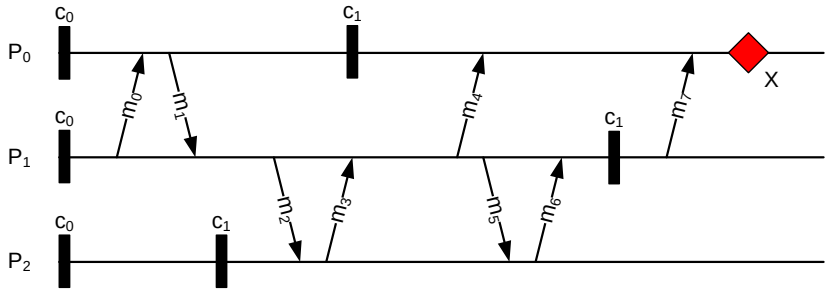


Abb. 8.5.: Beispiel-Sequenzdiagramm für Optimistic Logging.

8.1.2.3. Causal Logging

Bezogen auf seine Merkmale ist *Causal Logging* eine Mischform aus Pessimistic Logging und Optimistic Logging: Es gestattet wie auch Optimistic Logging einen asynchronen Zugriff auf das Stable Storage (ausgenommen beim Output Commit – da ist der Zugriff synchron). Wie auch Pessimistic Logging gestattet es den Prozessen, Output Commits unabhängig voneinander durchzuführen. Dabei werden niemals verwaiste Prozesse erzeugt. Causal Logging trägt den Namen, weil jeder Prozess im System die durch den Nachrichtenaustausch verursachten *kausalen* Abhängigkeiten mitverfolgt, die *kumulativ* den aktuellen Zustand des Prozesses bewirkt haben. Zwei bekannte Vertreter des Causal Logging sind das *Manetho*-Protokoll [28] sowie *Family-based Logging* [5]. Family-based Logging ist schlanker zu implementieren als Manetho, allerdings toleriert es nur einen Ausfall zu einer Zeit im System. Die folgenden Betrachtungen beziehen sich daher auf das Manetho-Protokoll.

Um die kausalen Abhängigkeiten im System zu verfolgen, benutzt das Manetho-Protokoll sog. *Antezedenzgraphen* [28]. Der Antezedenzgraph AG des i-ten Zustandsintervalls σ eines Prozesses p, $AG(\sigma_{p,i})$, ist ein gerichteter azyklischer Graph. Ein Knoten im Antezedenzgraph repräsentiert ein Zustandsintervall $\sigma_{p,i}$, das durch ein nicht-deterministisches Ereignis (Empfang einer Nachricht) eingeleitet wird (siehe Abbildung 8.6). Er hat stets zwei auf ihn zulaufende Kanten: eine von dem Knoten, der das vorherige Zustandsintervall des *selben* Prozesses repräsentiert und eine von dem Knoten der das Zustandsintervall des *sendenden* Prozesses repräsentiert. Abbildung 8.7 zeigt eine Beispielsequenz mit zugehörigem Antezedenzgraph. Ein Knoten eines Antezedenzgraphen enthält folgende Informationen:

- Den eindeutigen Identifier (ID) der Recovery Unit,
- die ID des Senders der Nachricht,
- den Index des eingeleiteten Zustandsintervalls und
- eine eindeutige ID der empfangenen Nachricht.[3]

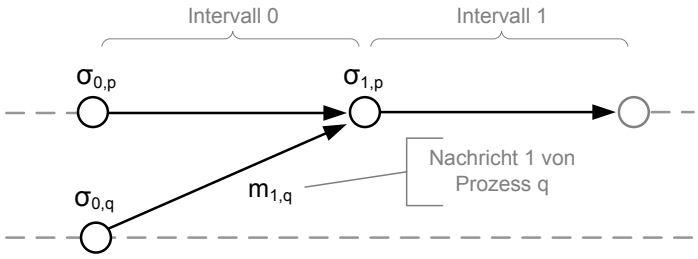


Abb. 8.6.: Ein Zustandsintervall i wird in der verwendeten Art der Darstellung jeweils mit dem Bezeichner σ_i eingeleitet.

Neben dem Antezedenzgraphen des aktuellen Zustandsintervalls hält eine Recovery Unit ein Message Log in ihrem flüchtigen Speicher, das Inhalte und Identifikationsmerkmale jeder *gesendeten* Nachricht enthält. Wird eine Nachricht gesendet, so wird der aktuelle Antezedenzgraph mit der Nachricht gesendet. Zu willkürlichen Zeitpunkten legt eine Recovery Unit einen Checkpoint ihres aktuellen Zustands auf einem Stable Storage ab. Das Erstellen dieser Checkpoints wird nicht mit anderen RUs koordiniert. Das aktuelle Message Log sowie der aktuelle Antezedenzgraph AG werden als Teil des Zustandes mit gespeichert. Antezedenzgraphen, die bereits auf dem Stable Storage abgelegt sind, müssen nicht mehr mit den Nachrichten der Recovery Unit verschickt werden. Abbildung 8.8 veranschaulicht diese Vorgänge. Bevor ein Output Commit durchgeführt wird, legt eine Recovery Unit den Antezedenzgraphen des aktuellen Zustandsintervalls auf dem Stable Storage ab. Auch hier ist keinerlei Koordination mit anderen RUs notwendig – allerdings erfolgt der Zugriff auf das Stable Storage synchron.

[3] Der Antezedenzgraph enthält *keine* Kopien der empfangenen Nachrichten, da diese im Log des *Senders* gespeichert werden.

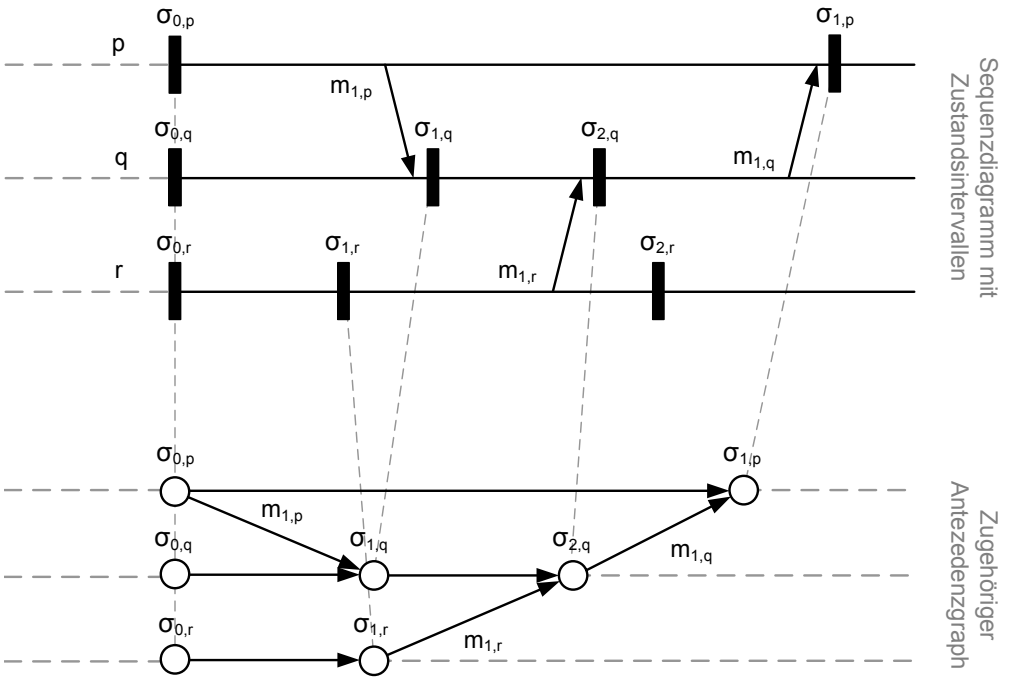


Abb. 8.7.: Manetho-Beispielsequenz mit zugehörigem Antezedenzgraphen.

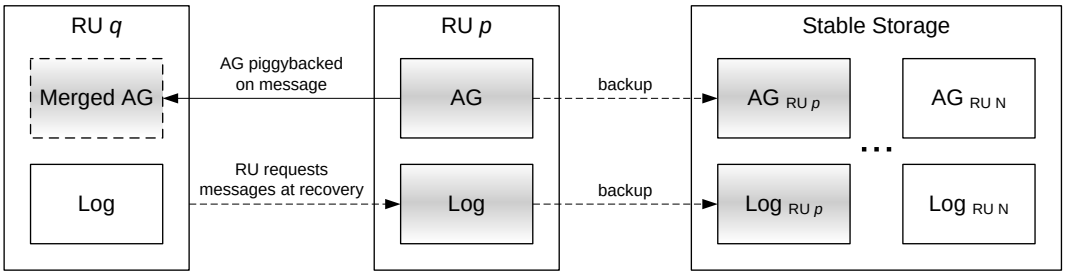


Abb. 8.8.: Recovery Units (RUs) legen Antezedenzgraphen (AGs) und Logs auf dem Stable Storage ab. AGs werden zusätzlich mit jeder Nachricht geschickt und beim Empfänger zusammengeführt.

Aufgrund allgemeiner Netzlatenzen ist es theoretisch denkbar, dass eine Nachricht ihren Empfänger erreicht, nachdem der Sender der Nachricht ausgefallen ist, der Empfänger jedoch bereits über den Ausfall des Senders informiert wurde. In diesem Fall hätte der Empfänger keine Möglichkeit zu ermitteln, ob die Nachricht vor dem Ausfall oder nach einem Recovery-Vorgang des Senders verschickt wurde. Aus diesem Grund führt das Manetho-Protokoll sog. *Inkarnationsnummern* ein, die bei jedem Recovery-Vorgang inkrementiert und mit jeder Nachricht verschickt werden. Auf diese Weise kann der Empfänger einer Nachricht ermitteln, ob die Nachricht vor oder nach dem letzten Recovery-Vorgang gesendet wurde. In Anhang C befindet sich eine detaillierte Beschreibung des Recovery-Protokolls, das von Manetho verwendet wird.

8.1.3. Gegenüberstellung der vorgestellten Protokolle

Die im Wesentlichen der Arbeit von Elnohazy et. al. [27] entnommene Tabelle 8.1 stellt die in den Abschnitten 8.1.1 und 8.1.2 beschriebenen Protokolle gegenüber:

- Sämtliche Log-basierten Protokolle setzen architekturbedingt stückweisen Determinismus (PWD) voraus (siehe Abschnitt 8.1.2). Checkpoint-basierte Protokolle sind auf stückweisen Determinismus hingegen nicht angewiesen.
- *Garbage Collection* – das Aufräumen von Checkpoint- bzw. Log-Daten – ist i. d. R. ein komplexer Vorgang[4], lediglich im Falle des Synchronous Checkpointing und des Pessimistic Logging ist der Vorgang trivial, da die Prozesse lediglich ihren letzten Checkpoint bzw. das Message Log ab dem letzten Checkpoint verwahren müssen.
- Asynchronous Checkpointing, Communication-induced Checkpointing und Optimistic Logging erfordern die Aufbewahrung mehrerer Checkpoints pro Prozess. Bei Synchronous Checkpointing, Pessimistic Logging sowie beim Causal Logging muss lediglich der letzte Checkpoint aufbewahrt werden.
- Das einzige Protokoll, bei dem der Domino-Effekt auftreten kann, ist Asynchronous Checkpointing, alle anderen Protokolle lassen den Domino-Effekt nicht zu.
- Verwaiste Prozesse sind bei Asynchronous Checkpointing, bei Communication-induced Checkpointing, sowie beim Optimistic Logging temporär möglich. Alle Protokolle gewährleisten jedoch, dass nach Abschluss des Rollbacks keine verwaisten Prozesse mehr vorhanden sind.
- Die Protokolle, die lediglich einen Checkpoint pro Prozess zulassen (Synchronous Checkpointing, Pessimistic Logging, sowie Causal Logging), erlauben natürlich auch nur ein Rollback zum letzten Checkpoint. Communication-induced Checkpointing, und Optimistic Logging erfordern mitunter einen Rollback über mehrere Checkpoints, während Asynchronous Checkpointing aufgrund des Domino-Effektes im Extremfall über alle Checkpoints zurückrollen kann.

[4] Je nach Verfahren ist nicht unmittelbar ersichtlich, welche Checkpoint- bzw. Log-Daten zu einem bestimmten Zeitpunkt obsolet sind und damit verworfen werden können.

- Bei Synchronous Checkpointing sowie Pessimistic Logging ist der Recovery-Vorgang vergleichsweise simpel. Bei den anderen Protokollen müssen zum Teil komplexe Abhängigkeiten zwischen den Prozessen berücksichtigt werden, was den Vorgang des Recovery aufwendiger macht.
- Im Prinzip erlauben alle Protokolle bis auf Family-based Logging beliebig viele Ausfälle zur selben Zeit. Family-based logging erlaubt hingegen nur einen einzigen Ausfall zu einer Zeit, was das Verfahren für viele Anwendungszwecke ungeeignet macht.
- Der im Fokus der vorliegenden Arbeit relevanteste Gesichtspunkt ist der Output Commit. Checkpoint-basierte Protokolle sind hier gänzlich ungeeignet, da sie entweder keinen Output Commit durchführen können (Asynchronous Checkpointing) oder, wie im Falle von Synchronous Checkpointing oder Communication-induced Checkpointing nur sehr ineffizient arbeiten, da sie erfordern, dass mehrere Prozesse Checkpoints erstellen, obwohl lediglich ein einzelner Prozess den Output Commit durchführt.
  Log-basierte Protokolle sind hier generell geeigneter, unterscheiden sich allerdings auch hinsichtlich der Performance beim Output Commit. Das schnellste Verfahren ist hier Pessimistic Logging, da es kein spezielles Protokoll für den Output Commit erfordert. Optimistic Logging ist aufgrund der notwendigen Multi-Host-Abstimmung beim Output Commit vergleichsweise langsam, während kausales Logging einen guten Kompromiss darstellt.

8.2. Realisierung eines Rollback-Recovery-Verfahrens für das betrachtete System

Im Folgenden wird ein geeignetes Rollback-Recovery-Verfahren für das im ersten Teil der Arbeit entwickelte System konzipiert. Mit Bezug auf die Eigenschaften dieses Systems (siehe Abschnitt 6.1), sind die folgenden Anforderungen an ein Rollback-Recovery-Verfahren zu stellen:

- **Eignung für häufigen Output Commit**
  Interaktion mit der Außenwelt erfolgt i. d. R. über den Transceiver. Durch den typischen Einsatz von Transceivern als Aktuatoren findet eine rege Interaktion mit der Außenwelt statt. Das angewendete Rollback-Recovery-Protokoll muss daher für einen häufigen Output Commit geeignet sein.
- **Zeitnahes Ansprechverhalten der Geräte**
  Der Zugriff auf einen Transceiver darf durch das Rollback-Recovery-Protokoll nicht zu stark verlangsamt werden. Mehr als 200 ms Latenz bei Durchführung eines Schaltvorgangs wirken sich als störend für den Benutzer aus.[5]
- **Möglichst geringer Kommunikationsoverhead**
  Um die teilnehmenden Geräte nicht unnötig zu belasten, sollte ein Rollback-Recove-

[5] Wert empirisch ermittelt.

	CHECKPOINTING			LOGGING		
	unkoordiniert	**koordiniert**	**induziert**	**pessimistisch**	**optimistisch**	**kausal**
PWD vorausgesetzt	nein	nein	nein	ja	ja	ja
Garbage Collection	komplex	einfach	komplex	einfach	komplex	komplex (einfacher bei FBL)
Checkpoints pro Prozess	mehrere	1	mehrere	1	mehrere	1
Domino-Effekt	möglich	nein	nein	nein	nein	nein
Verwaiste Prozesse	möglich	nein	möglich	nein	möglich	nein
Umfang des Rollbacks	beliebig	letzter globaler Checkpoint	mehrere Checkpoints möglich	letzter Checkpoint	mehrere Checkpoints möglich	letzter Checkpoint
Recovery-Vorgang	komplex	einfach	komplex	einfach	komplex	komplex (einfacher bei FBL)
Gleichzeitige Ausfälle	beliebig	beliebig	beliebig	beliebig	beliebig	beliebig (1 bei FBL)
Output Commit	nicht möglich	sehr langsam	sehr langsam	sehr schnell	langsam	schnell

Tab. 8.1.: Vergleichsübersicht der Rollback-Recovery-Protokolle.

ry-Protokoll ein möglichst geringes zusätzliches Datenaufkommen verursachen. Dies gilt besonders in Hinblick die Transceiver, deren Energiereserven und Bandbreiten sehr gering sind.

- **Umgang mit temporär abwesenden Transceivern**
  Transceiver können aufgrund ihrer *Betriebszyklen* zeitweise im Standby-Betrieb sein und sind daher vorübergehend nicht erreichbar. Diese Eigenschaft darf dem Einsatz des Rollback-Recovery-Protokolls nicht im Wege stehen.

8.2.1. Transceiver und Proxy als gemeinsame Recovery Unit

Für Systeme, die Geräte unterschiedlicher Leistungsklassen enthalten, werden mitunter *hybride Verfahren* für Rollback-Recovery vorgeschlagen [41]. Dabei verwenden leistungsarme Geräte ein anderes Rollback-Recovery-Verfahren als Geräte höherer Leistungsklasse. Generell wäre eine Anwendung solch hybrider Verfahren auch auf das im Rahmen der vorliegenden Arbeit betrachtete System denkbar. Dies würde jedoch immer noch bedeuten, dass die Transceiver ein Rollback-Recovery Protokoll direkt unterstützen müssten, was ihre ohnehin knappen Ressourcen zusätzlich beanspruchen würde.

Allerdings kann bzgl. des betrachteten Systems ausgenutzt werden, dass pro Transceiver jeweils nur *ein einzelner* Proxy existiert. Damit kann das Paar bestehend aus Proxy und Transceiver als Einheit bzw. in sich geschlossene Recovery Unit betrachtet werden, die sich intern synchronisiert (siehe Abbildung 8.9).

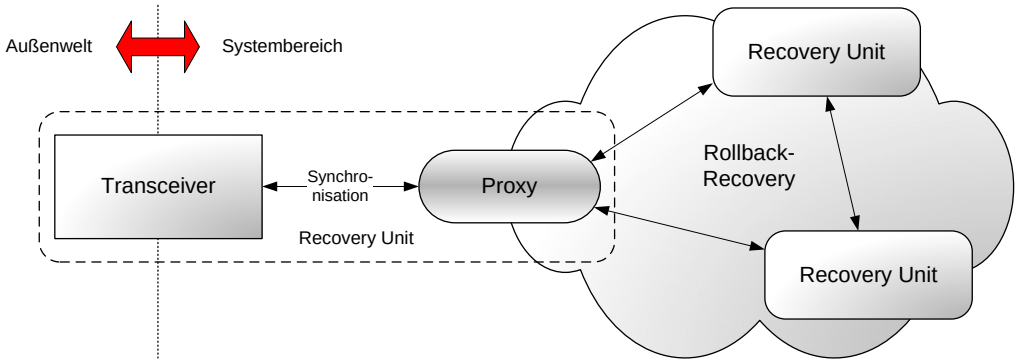


Abb. 8.9.: Der Transceiver synchronisiert sich mit seinem Proxy, während der Proxy am eigentlichen Rollback-Recovery teilnimmt.

Durch diesen Ansatz wird der Transceiver entlastet, da er für ein Rollback-Recovery Protokoll selbst keinerlei Unterstützung bieten muss. Ferner kann er seine „gewohnten“ Betriebszyklen beibehalten und somit Energie einsparen.

8.2.2. Auswahl eines Rollback-Recovery Protokolls als Basis

Im Folgenden wird untersucht, welche Rollback-Recovery Protokolle sich unter den o. g. Anforderungen auf das betrachtete System anwenden lassen. Dabei werden zuerst alle Protokolle ausgeschlossen, die die genannten Anforderungen nicht bzw. nur unzureichend erfüllen. Die in Frage kommenden Kandidaten werden schließlich einer weiteren Evaluierung unterzogen.

Ein System, wie es im Rahmen der vorliegenden Arbeit behandelt wird, interagiert regelmäßig und oft mit der Außenwelt. Da Checkpointing-basierte Verfahren in Bezug auf den Output Commit nur sehr ineffizient arbeiten oder ihn, wie im Falle von Asynchronous Checkpointing, überhaupt nicht unterstützen, beschränkt sich die folgende Betrachtung daher auf Log-basierte Verfahren. Zwei mögliche Extreme stellen hier Pessimistic Logging und Optimistic Logging dar, während sich Causal Logging als interessanter Kompromiss dieser Ansätze darstellt.

Während Pessimistic Logging Log-Einträge unverzüglich auf einem Stable Storage sichert und damit eine größere Latenz im fehlerfreien Betrieb in Kauf nimmt, sichert Optimistic Logging Einträge asynchron auf dem Stable Storage. Dies hat den Vorteil, dass der Kommunikationsoverhead im fehlerfreien Betrieb i. d. R. geringer ist als beim Pessimistic Logging. Allerdings lässt Optimistic Logging die Entstehung temporär verwaister Prozesse zu, wodurch ein größerer Aufwand im Recovery-Fall hingenommen werden muss und der Output Commit aufgrund des synchronen Zugriffs auf das Stable Storage und der mitunter hinzukommenden Multi-Node-Abstimmung vergleichsweise langsam abläuft (siehe Abschnitt 8.1.2.2). Weiterhin ist es möglich, dass eine vom Output Commit indirekt betroffene Recovery Unit ausgefallen ist, ihr Ausfall jedoch noch nicht bemerkt wurde. Der Kontaktierungsversuch würde erst nach einem Timeout abgebrochen. Allerdings würde der *gesamte* laufende Output Commit derweil blockieren. Da im betrachteten System sehr häufig Output Commits durchgeführt werden, wird Optimistic Logging daher aus der weiteren Betrachtung ausgeschlossen.

Tabelle 8.2 fasst die in diesem Abschnitt getroffenen Überlegungen nochmals zusammen. Aufgrund der gewonnenen Erkenntnisse beschränkt sich die weitere Betrachtung auf den möglichen Einsatz von Pessimistic Logging oder Causal Logging als Alternative. Da jedoch auch für Causal Logging gilt, dass ein asynchroner Zugriff auf das Stable Storage nur möglich ist, wenn kein Output Commit durchgeführt wird, ist es fraglich ob sich der Einsatz des aufwendigeren Causal Logging im Vergleich zum einfachen Pessimistic Logging im betrachteten System lohnt. Diese Frage soll im Folgenden durch eine nähere Evaluierung der beiden Verfahren beantwortet werden.

8.2.3. Lazy Output Commit

In diesem Abschnitt wird mit dem *Lazy Output Commit* eine im Rahmen der vorliegenden Arbeit entstandene Anpassung von Rollback-Recovery-Protokollen an Proxy-basierte Systeme vorgestellt. Der Lazy Output Commit stellt eine Verbesserung für Protokolle dar,

Verfahren	Eignung	Anmerkung
Asynchronous Checkpointing	ungeeignet	Kein Output Commit möglich.
Synchronous Checkpointing	ungeeignet	Output Commit sehr ineffizient.
Communication-induced Checkpointing	ungeeignet	Output Commit sehr ineffizient.
Pessimistic Logging	geeignet	Logging erfolgt stets synchron. Entstehung von Latenzzeiten.
Optimistic Logging	bedingt geeignet	Output Commit vergleichsweise ineffizient. Node-übergreifende Abstimmung beim Output Commit erforderlich.
Manetho (Causal Logging)	geeignet	Vergleichsweise aufwendig. Asynchroner Zugriff auf Stable Storage nur möglich, wenn kein Output Commit durchgeführt wird.
FBL (Causal Logging)	ungeeignet	Funktioniert nicht bei mehreren gleichzeitigen Ausfällen.

Tab. 8.2.: Eignung der Protokolle.

die beim Output Commit über einen Transceiver einen synchronen Zugriff auf das Stable Storage erfordern, wie es z. B. beim Pessimistic Logging der Fall ist. Hierbei wird ausgenutzt, dass Zustandsinformationen einer Recovery Unit bestehend aus Transceiver und Proxy *redundant* sowohl im Transceiver, als auch im Proxy zur Verfügung stehen. Der sich ergebende Vorteil ist, dass unter bestimmten Umständen die Zeit für den Zugriff auf das Stable Storage eingespart werden kann, da dieser *asynchron* zum Zugriff auf den Transceiver durchgeführt werden kann.

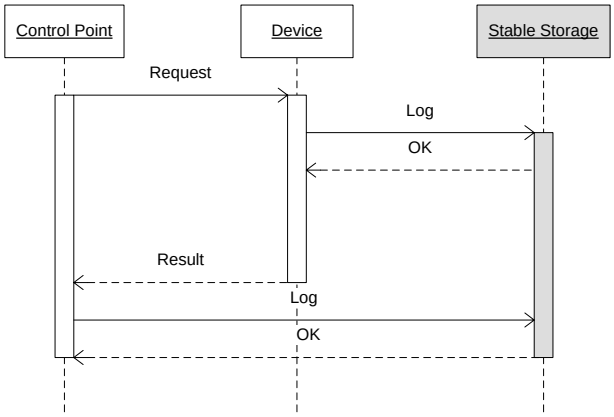
Um den Lazy Output Commit zu verstehen, muss zuerst beleuchtet werden, wie ein herkömmlicher Output Commit über einen Transceiver abläuft: Bei Verwendung von Protokollen wie SOAP wird jeder *Action Request* durch einen *Action Response* quittiert. Abbildung 8.10 zeigt die Verarbeitung einer SOAP-Action unter Anwendung von Pessimistic Logging. Betrachten wir zunächst Abbildung 8.10(a). Der Ablauf entspricht einem Aufruf einer Action von einem Control Point auf einem herkömmlichen UPnP Device. Entsprechend dem Grundgedanken des Pessimistic Logging speichern sowohl das Device als auch der Control Point die Determinanten jeder empfangenen Nachricht auf dem Stable Storage, bevor sie sie verarbeiten. Abbildung 8.10(b) veranschaulicht diese Zusammenhänge unter Einbeziehung eines Transceivers. Hier zeigt sich, dass durch die Verarbeitung der Antwort des Transceivers ein *weiterer* Logging-Vorgang des Proxys hinzukommt. Damit führt ein Proxy bei einem einzelnen Aufruf einer Action durch einen Control Point aufgrund der Transceiver-Interaktion *zwei* synchrone Zugriffe auf das Stable Storage durch.

Wie in Abbildung 8.11 dargestellt, erfolgt der Zugriff auf den Transceiver beim Lazy Output Commit asynchron zum Zugriff auf das Stable Storage. Weiterhin wird nach Erhalt der Antwort des Transceivers kein weiterer Logging-Vorgang durchgeführt. Solange der Zugriff auf den Transceiver gleich viel oder mehr Zeit in Anspruch nimmt, als der Zugriff auf das Stable Storage, fällt der Logging-Vorgang zeitlich nicht mehr ins Gewicht. In diesem Fall wird die Zeit für einen Zugriff auf das Stable Storage also komplett eingespart. Tabelle 8.3 betrachtet alle möglichen Ausfallszenarien beim Lazy Output Commit und zeigt, dass in keinem Fall die Konsistenz des Systems beeinträchtigt wird. Die einzige Bedingung, die an das Verfahren gestellt werden muss, ist, dass der Transceiver in der Lage sein muss, den Zustand des Proxys komplett zu sichern, da sonst unter bestimmten Umständen (*Fall 3*) keine Konsistenz mit der Außenwelt garantiert werden kann.

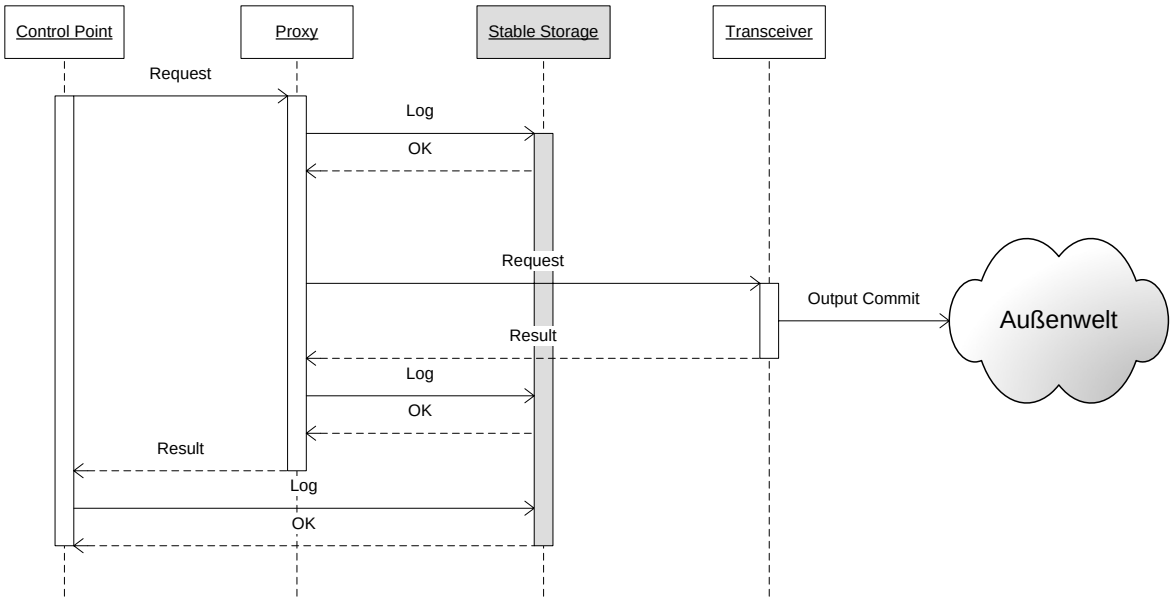
Es bleibt zu klären, warum davon ausgegangen werden kann, dass der Zugriff auf den Transceiver i. d. R. mehr Zeit in Anspruch nimmt, als der Zugriff auf das Stable Storage: Ein Action Request r_{proxy}, der von einem Control Point an den Proxy gesendet wird, enthält neben dem Namen der aufzurufenden Action n_{action} auch einen Payload p. Der Payload kann aufgeschlüsselt werden in einen Teil, der lediglich dem Proxy zugestellt wird p_{proxy} und einen Teil p_{trans}, der an den Transceiver durchgereicht wird. Formal:

$$r_{proxy} = n_{action} + p_{trans} + p_{proxy}$$

In der Regel gilt $p_{proxy} = 0$. Damit entspricht die Größe der Determinate, die vom Proxy auf das Stable Storage gelogged wird, in etwa der Größe der Nachricht an den Transceiver. Da davon auszugehen ist, dass bei der *schmalbandigen* Anbindung des Transceivers der



(a) Ohne Einbeziehung des Transceivers.



(b) Mit Einbeziehung des Transceivers.

Abb. 8.10.: Verarbeitung eines SOAP-Requests bei Verwendung von Pessimistic Logging.

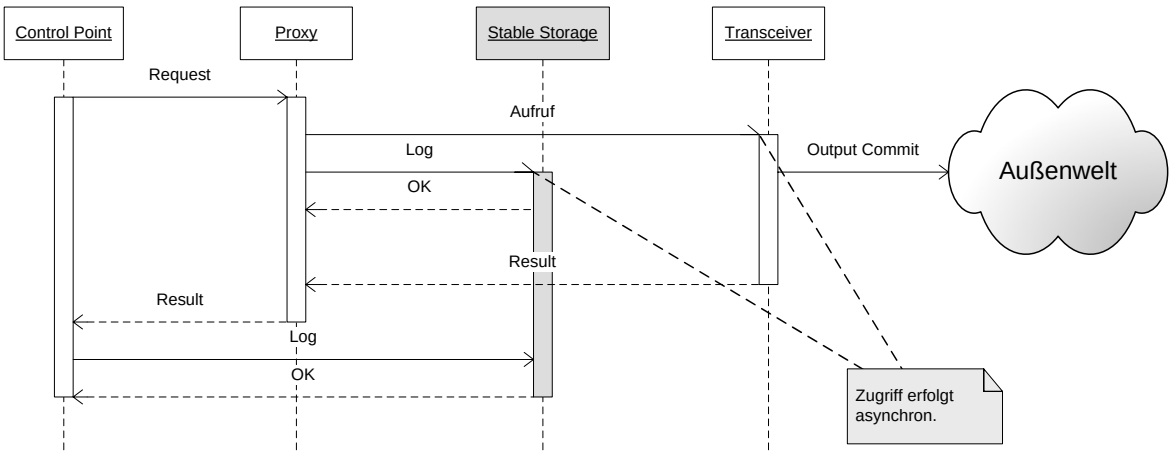


Abb. 8.11.: Lazy Output Commit.

Zugriff auf den Transceiver $t_{transcomm}$ zuzüglich der Zeit für den Output Commit t_{commit} mehr Zeit in Anspruch nimmt, als der Logging-Vorgang des Proxy, fällt dieser zeitlich nicht ins Gewicht:

$$t_{logging} < t_{transcomm} + t_{commit}$$

In einigen Fällen mag dies nicht zutreffen: Nämlich dann, wenn ein exklusiver Payload an den Proxy p_{proxy} existiert, der gelogged werden muss oder, falls das Stable Storage so langsam ist, dass der Logging-Vorgang entsprechend mehr Zeit in Anspruch nimmt. Allerdings macht dieser Fall die Anwendung des *Lazy Output Commit* nicht unmöglich, sondern reduziert lediglich seine Wirkung. Schließlich wird auch in diesem Fall immer noch Zeit gewonnen, da der Zugriff auf den Transceiver und der Logging-Vorgang asynchron ablaufen.

8.2.4. Aufbau eines UPnP-basierten Storage-Dienstes

Recovery Units sind auf ein *Storage* angewiesen, mit dessen Hilfe sie Checkpoint- bzw. Log-Daten ablegen können. Von einem *Stable Storage* sprechen wir dann, wenn sichergestellt ist (oder davon ausgegangen wird), dass die abgelegten Daten nicht verloren gehen können. In diesem Abschnitt wird der Aufbau eines Storage-Dienstes für das im Rahmen der vorliegenden Arbeit entstandene System beschrieben. Dabei wird ausschließlich die Problematik des Zugriffs auf das Storage betrachtet, jedoch nicht, auf welche Weise Daten ausfallsicher verwahrt werden können. Da jedoch davon auszugehen ist, dass ein Storage auf einem leistungsfähigen Host im Netzwerk realisiert wird, können gängige Verfahren wie z. B. RAID-Festplattenarrays eingesetzt werden, um Log-Daten bzw. Checkpoints sicher zu verwahren. Im Folgenden werden daher die Begriffe „Storage“ und „Stable Storage“ als Synonyme verwendet.

Fall Nr.	Ausfall	Committed	Gelogged	Auswirkung	Zulässig
1	Nur Proxy			Entspricht einer verlorenen Nachricht des Control Points an den Proxy. Nach dem Neustart des Proxys muss der Control Point den Aufruf erneut schicken.	✓
2	Nur Proxy		✓	Der Proxy startet neu und führt einen Replay mit Hilfe des Stable Storage durch. Danach ist sein Status zum Rest des Systems konsistent.	✓
3	Nur Proxy	✓		Der Proxy startet neu und führt einen Replay mit Hilfe des Transceivers durch. Dies ist allerdings nur möglich, sofern der Transceiver den kompletten Status des Proxys sichern kann.	(✓)
4	Nur Proxy	✓	✓	Entspricht *Fall 2*.	✓
5	Transceiver			Als Folge des Transceiver-Ausfalls wird der Proxy beendet. Entspricht damit *Fall 1* mit dem Unterschied, dass Transceiver und Proxy das System endgültig verlassen.	✓
6	Transceiver		✓	Entspricht einer verlorenen Nachricht an die Außenwelt. Der Proxy würde als Folge des ausgefallenen Transceivers beendet und zuvor eine Fehlermeldung an den Control Point senden.	✓
7	Transceiver	✓		Der Ausfall des Transceivers geschieht *nach* Abschluss der Operation. Der Proxy wird aufgrund des Transceiver-Ausfalls beendet.	✓
8	Transceiver	✓	✓	Entspricht *Fall 7*.	✓
9	Beide			Entspricht einer verlorenen Nachricht des Control Points an den Proxy. Transceiver und Proxy verlassen das System allerdings endgültig.	✓
10	Beide		✓	Entspricht *Fall 9*.	✓
11	Beide	✓		Der Ausfall der Komponenten geschieht *nach* Abschluss der Operation.	✓
12	Beide	✓	✓	Entspricht *Fall 11*.	✓

Tab. 8.3.: Mögliche Ausfallszenarien beim Lazy Output Commit. (*Committed*: Output Commit durchgeführt. *Gelogged*: Determinante der Nachricht auf dem Stable Storage gesichert.)

Das Storage muss von allen Geräten in der Netzgemeinschaft gefunden werden können. Aus diesem Grund liegt es nahe, Discovery-Mechanismen einzusetzen, um das Storage im System bekannt zu machen. Im Rahmen der vorliegenden Arbeit wurde hierfür, wie im gesamten System, UPnP verwendet. Geräte können damit gezielt nach einem Storage-Device im System suchen. Sobald ein Gerät eine Verbindung zu dem Storage geöffnet hat, kann es seine Checkpointing- bzw. Log-Daten dort ablegen. Dabei wird jedem Gerät ein „privater“ Storage-Dienst zur Verfügung gestellt. Die Architektur des Storage ist dabei unabhängig von der Art der Daten, die dort abgelegt werden. Damit ist es Sache des jeweiligen Gerätes, welche Daten es in welcher Form dort ablegt.

Sofern die verwendete Middleware keine binären Protokolle verwendet, so wie es bei UPnP der Fall ist, ist sie nur bedingt für den Transport der eigentlichen Log-Daten geeignet. Würde z. B. für jeden Ablagevorgang von Log-Daten eine neue UPnP-Verbindung aufgebaut, so würden pro Vorgang Latenzzeiten in Höhe von ca. 70 Millisekunden anfallen – wird hingegen ein binäres Protokoll direkt auf einer TCP-Verbindung verwendet, so fallen Latenzzeiten von unter zehn Millisekunden an, wie in Abbildung 8.12 gezeigt.[6] Dieses Problem lässt sich jedoch mit Hilfe von *Out-of-Band*-Verbindungen[7] umgehen, wie es z. B. bei *UPnP-AV* [99] im Rahmen von Multimedia-Anwendungen praktiziert wird.

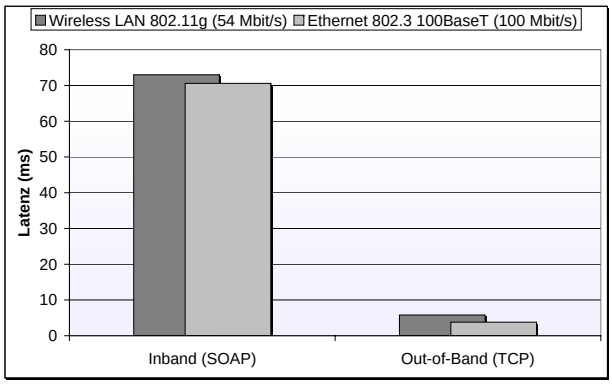


Abb. 8.12.: Latenzzeiten der UPnP Actions.

Damit wird UPnP lediglich dazu verwendet, das Storage im Netzwerk zu finden und eine Recovery Unit an-, bzw. abzumelden. Der genaue Ablauf ist in Abbildung 8.13 dargestellt: Nachdem eine RU den Storage-Dienst im Netzwerk gefunden hat, ruft sie die Action *Register* auf und übergibt ihre eigene UUID als Kennung. Das Storage liefert der RU anschließend die Nummer eines der RU exklusiv bereitgestellten TCP-Ports zurück. Über

[6] Gemessen wurde in einem herkömmlichen 100 Mb/s LAN und in einem 54 Mb/s WLAN. Wie man an den geringen Zeitunterschieden zwischen der LAN- und der WLAN-Verbindung sehen kann (Abbildung 8.12), wird bei Anwendung von UPnP die meiste Zeit für die Aufbereitung der XML-Daten benötigt, während die eigentliche Datenübertragung zeitlich kaum ins Gewicht fällt. Die Messung fand auf Grundlage des Java-basierten Infineon UPnP-Stacks statt.

[7] Mit einer *Out-of-Band*-Verbindung ist im Zusammenhang mit UPnP eine reine TCP-Verbindung ohne Verwendung des SOAP-Protokolls gemeint.

diesen Port kann die RU eine TCP-Verbindung aufbauen und beliebige Datensätze auf dem Storage ablegen. Auf welche Weise die Nutzdaten strukturiert werden, ist allein Sache der Recovery Unit.[8] Über einen Aufruf der Action *Goodbye* kann sich die RU ordnungsgemäß vom Storage abmelden.

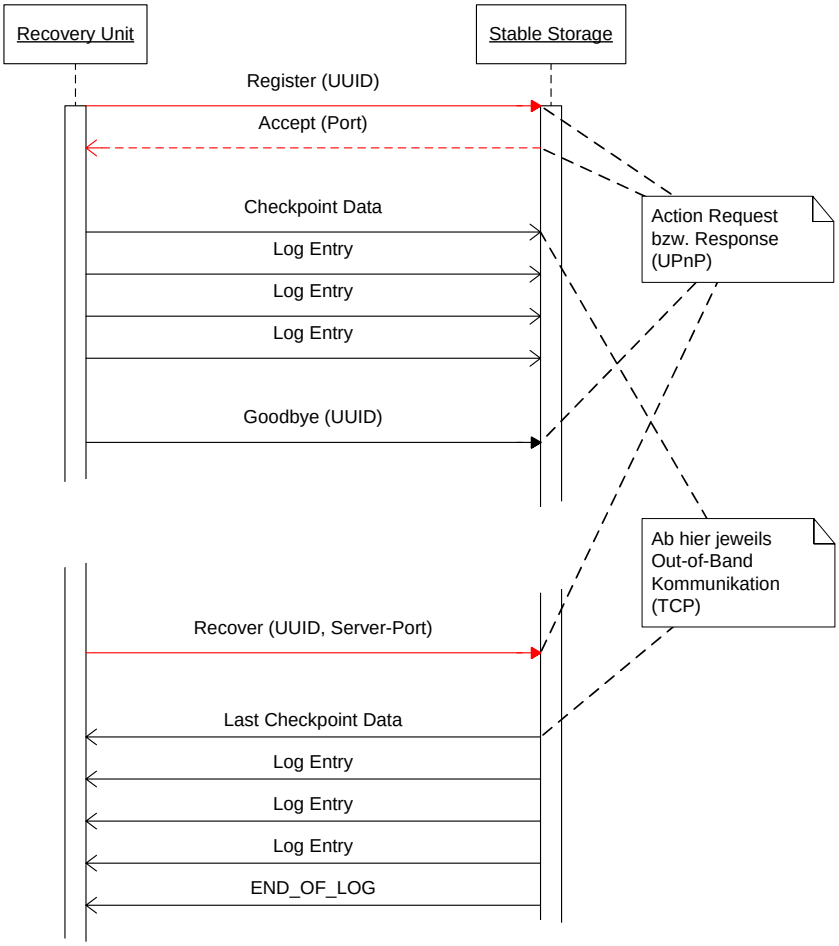


Abb. 8.13.: Zugriff auf das Storage im fehlerfreien Betrieb sowie beim Recovery.

Um nach einem Neustart einen Recovery-Vorgang durchführen zu können, meldet sich die Recovery Unit über die Action *Recover* am Storage an. Als Parameter werden die UUID der RU sowie die Nummer eines Server-Ports angegeben, auf dem die RU die Daten ihres letzten Checkpoints bzw. ihre Log-Daten erwartet. Der erste Datensatz, den die RU vom Storage empfängt, ist der zuletzt abgelegte Checkpoint. Sobald sie ihren Status entsprechend den Informationen im Checkpoint wiederhergestellt hat, ruft die RU die

[8] Allerdings muss die RU signalisieren, ob es sich bei einem Datensatz um einen Checkpoint oder einen Log-Eintrag handelt. Hierzu kann z. B. das erste Bit einer jeweiligen Übertragung verwendet werden.

Action *Get_Next* unter Angabe ihrer UUID auf dem Storage auf und bekommt den ersten Eintrag des Logs. Mit jedem Aufruf von *Get_Next* wird ein weiterer Log-Eintrag gesendet. Ist kein weiterer Log-Eintrag vorhanden, wird der String `END_OF_LOG` zurückgeliefert und der Vorgang ist abgeschlossen. Danach meldet sich die RU wieder über die Action *Register* am Storage an und nimmt ihre Arbeit wieder auf.

Abbildung 8.14 zeigt für die Erstellung von Checkpoints sicherungsrelevante Komponenten eines UPnP-Gerätes. Diese Komponenten sind zum einen die Statusvariablen, die für andere Komponenten sichtbar sind. Zum anderen kann jedes Gerät eine beliebige Menge an weiteren applikationsspezifischen Zustandsinformationen besitzen, die gesichert werden müssen. Ausgehend von einem *UPnP Root Device* ist die Sicherung dieser Daten für jedes eingebettete Gerät vorzunehmen.

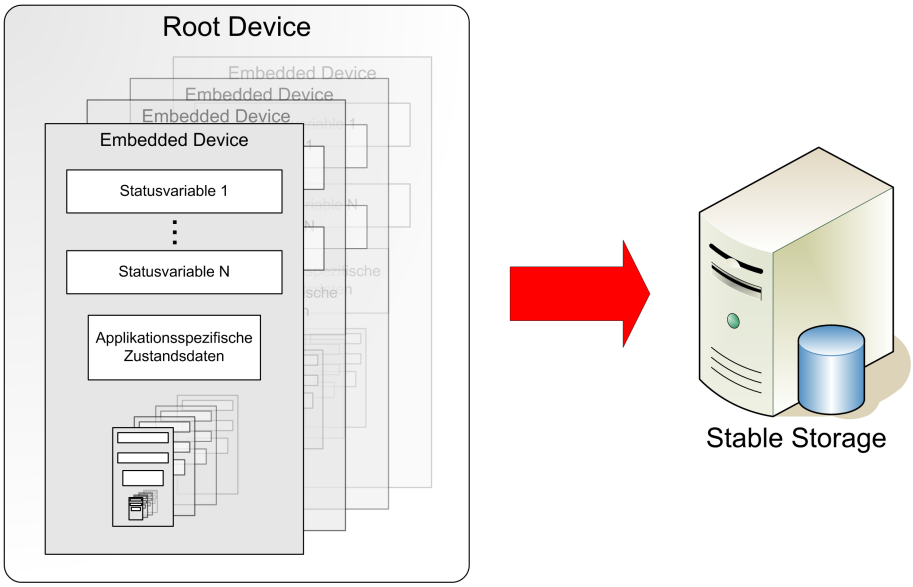


Abb. 8.14.: Zu sichernde Komponenten eines UPnP-Gerätes.

8.3. Evaluierung von Pessimistic Logging und Causal Logging

Tabelle 8.4 zeigt, welche Datenmengen im Rahmen der folgenden Betrachtungen für Log-Einträge und Checkpoints angenommen wurden. Für einen typischen Aufruf eines Kommandos mit Parametern werden ca. 20 - 40 Byte angenommen. Eine im Log abzulegende Determinante enthält neben dem Inhalt der Nachricht die 28 Byte lange UUID des Senders sowie 8 Byte für eine zusätzliche Event-ID. Die Größe der Determinate wird daher mit 80 Byte überschlagen. Für einen Checkpoint, der die vollständigen Statusinformation einer Recovery Unit zu einem Zeitpunkt enthält, wurden großzügig 20 kB angesetzt.

Bezeichnung	Beschreibung	Größe
SenderID	ID des Senders der Nachricht (UPnP UUID)	28 Byte
EventID	ID, die die Reihenfolge der Events angibt	8 Byte
MessageContent	Funktionenname + Aufrufparameter	20 - 40 Byte
CheckpointData	Kompletter Status-Record einer RU	20 kB

Tab. 8.4.: Datenvolumen für Log-Einträge und Checkpoints.

Zugriffslatenzen

Für die Messungen der Latenzzeiten beim Zugriff auf das im letzten Abschnitt vorgestellte Stable Storage wurde ein Setup bestehend aus zwei herkömmlichen PCs (1,6 und 2,5 GHz) verwendet, die über ein 100 Mbit/s LAN verbunden wurden. Die Referenzimplementierung des Storage-Dienstes wurde unter Verwendung der Java Standard Edition 6 vorgenommen. Wie in Abschnitt 8.2.4 beschrieben, erfolgt ein Zugriff auf das Stable Storage direkt über eine TCP-Verbindung ohne Verwendung des SOAP-Protokolls. Ein einzelner Logging-Vorgang (80 Byte Log-Daten) nimmt ca. 5 ms in Anspruch. Ein Checkpoint mit einen Umfang von 20 kB wird in unter 15 ms verarbeitet.[9] Der Einfachheit halber wird im Folgenden eine Zugriffsdauer von rund 10 ms auf das Stable Storage veranschlagt.

Da im zeitlichen Rahmen der vorliegenden Arbeit keine Referenz-Hardware eines Transceivers verfügbar war, kann die Dauer für einen Zugriff auf den Transceiver lediglich überschlagen werden: Wenn eine einfache Kommandonachricht von 20 Bytes an den Transceiver geschickt wird, so werden für die einfache Übertragung bei einer Nutzdatenrate von 30 kbit/s ca. 5 ms (ohne Latenz) für die reine Datenübertragung in eine Richtung benötigt. Für die Gesamtdauer, bis eine Bestätigung vom Transceiver erhalten wird, werden daher 10 ms veranschlagt:

$$t_{trans} = 10\ ms$$

Bei Verwendung von *Pessimistic Logging* und dem *Lazy Output Commit* wird pro Kommunikationszyklus insgesamt zweimal auf das Stable Storage zugegriffen: Einmal, beim eigentlichen Output Commit durch den Proxy und einmal durch den Control Point nach Erhalt der Bestätigungsnachricht (siehe Abbildung 8.11). Der Zugriff auf das Stable Storage durch den Proxy wird allerdings zeitlich überlagert durch den zeitintensiveren asynchronen Zugriff auf den Transceiver und wirkt sich auf den Gesamtzeitbedarf nicht aus. Damit ergibt sich insgesamt eine Latenz von 10 ms für den Logging-Vorgang:

$$t_{logging-PL} = 10\ ms$$

[9] Hier wird der Einfluss der Netzlatenz deutlich, da die Übermittlung des kleineren Datensatzes nur unverhältnismäßig weniger Zeit in Anspruch nimmt. Bei der Messung wurde nicht die Zeit zum Ablegen der Daten auf einem permanenten Datenspeicher wie einer Festplatte berücksichtigt.

Beim Output Commit im Falle des *Causal Logging* muss der Antezedenzgraph auf dem Stable Storage abgelegt werden. Hierfür werden wieder rund 10 ms angesetzt. Damit entsteht bei Durchführung des Output Commit über den Transceiver eine durch das Causal Logging hervorgerufene Gesamtlatenz von ca. 10 ms, während in den Fällen, in denen der Proxy eine Anfrage selbst beantworten kann und damit kein Output Commit durchgeführt wird, keine Latenz durch Anwendung des Causal Logging entsteht. Unter der Annahme, dass der Proxy die Hälfte aller Anfragen eigenständig beantworten kann, entsteht eine gemittelte Gesamtlatenz bei Anwendung von Causal Logging von 5 ms:

$$\bar{t}_{logging-CL} = 5\ ms$$

Die Gesamtzugriffszeit beim Aufruf einer Aktion von einem Control Point an einen Transceiver kann nun wie folgt überschlagen werden, wobei für den Zugriff über UPnP $t_{upnp} = 70\ ms$ angenommen werden (siehe Abbildung 8.12):

$$t_{gesamt} = t_{upnp} + t_{trans} + t_{logging}$$

Bei Pessimistic Logging:

$$70\ ms + 10\ ms + 10\ ms = 90\ ms$$

Bei Causal Logging:

$$70\ ms + 10\ ms + 5\ ms = 85\ ms$$

Bei dieser Überlegung wird die Verarbeitungszeit des Transceivers nicht berücksichtigt. Setzt man für die maximal zulässige Gesamtdauer einer Aktion 200 ms an[10], so verbleiben in beiden Fällen gut 100 ms für die Verarbeitung des Kommandos durch den Transceiver. Insgesamt zeigt sich, dass unter den gegebenen Anforderungen und Systemeigenschaften bzgl. der Zugriffslatenz kein nennenswerter Vorteil durch Einsatz des Causal Logging Protokolls gegenüber Pessimistic Logging zu erzielen ist.

Kommunikationsoverhead

Es wurde weiterhin untersucht, welcher Overhead bei der Anwendung von Pessimistic Logging im Vergleich zum Manetho-Protokoll (Causal Logging) entsteht. Hierzu wurde eine Simulationsumgebung auf Basis der *Diskreten-Event-Simulation* eingesetzt, die im Rahmen der vorliegenden Arbeit entwickelt wurde.

Es wurde angenommen, dass hinter jedem Proxy, der eine Nachricht empfängt, ein Transceiver steht und in 50 % der Fälle die Nachricht an den Transceiver weitergeleitet und ein Output Commit durchgeführt wird.[11] Die Simulation wurde für 2 bis 10 Proxies durchgeführt, die entsprechend einem zufälligen Muster miteinander kommunizieren. Dabei wurde angenommen, dass jeweils doppelt so viele Nachrichten ausgetauscht werden, wie

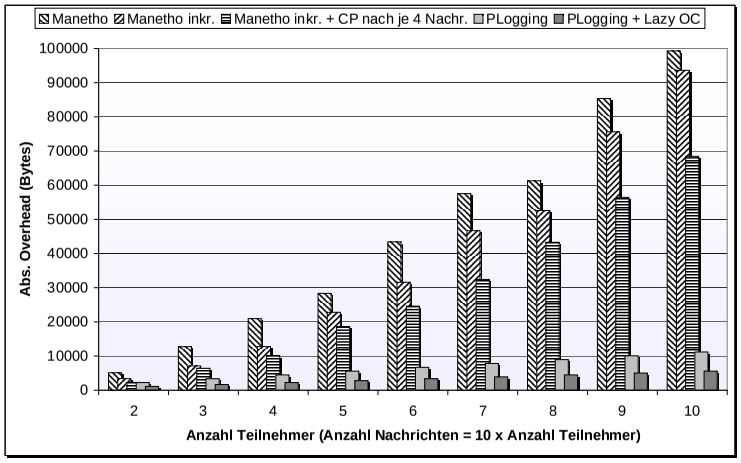
[10] Bis zu diesem empirisch ermittelten Wert empfindet ein Benutzer die Antwort einer Aktion als *unmittelbar*.

[11] Das heißt, bei Anwendung des Manetho-Protokolls muss der aktuelle Antezedenzgraph extra auf dem Stable Storage gesichert werden.

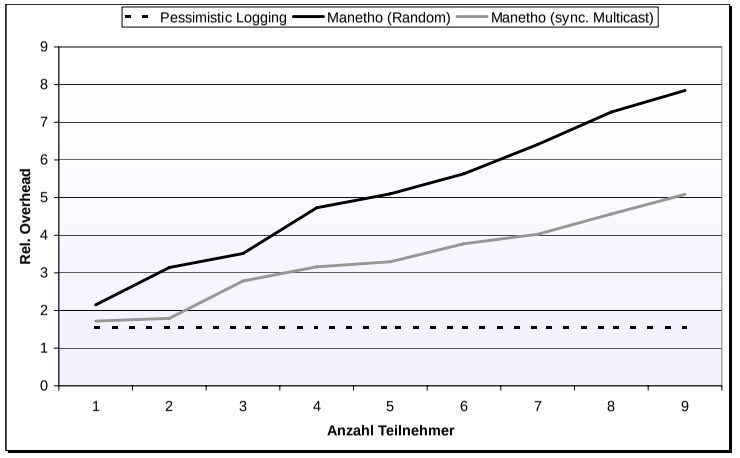
Proxies im System vorhanden sind. Bei 10 Proxies werden also 20 Kommandonachrichten ausgetauscht. Für die Bestimmung der anfallenden Datenmenge beim Manetho-Protokoll wurde jeweils über fünf Messungen gemittelt, da diese von dem jeweiligen Kommunikationsmuster abhängt. Neben der Betrachtung des einfachen Manetho-Protokolls wurden verschiedene Verbesserungsmöglichkeiten, wie ein inkrementelles Senden der Antezedenzgraphen sowie periodisches Checkpointing untersucht [28]. Beim Pessimistic Logging wurde entsprechend den Abbildungen 8.10(b) und 8.11 der normale Output Commit mit dem Lazy Output Commit verglichen. Für die jeweiligen Nachrichtenlängen wurden die Werte gewählt, die in Tabelle 8.4 vorgestellt wurden.

Die Ergebnisse der Untersuchungen sind in den Abbildungen 8.15(a) bis 8.15(d) dargestellt. Abbildung 8.15(a) zeigt den absoluten Overhead des Manetho-Protokolls im Vergleich zu dem Overhead, der bei Pessimistic Logging entsteht. Es ist zu erkennen, dass der Overhead bei Manetho mit der Komplexität des zugrundeliegenden *Kommunikationsmusters* steigt (nicht-linearer Anstieg): Wenn häufiger mit unterschiedlichen Partnern kommuniziert wird, entstehen mehr Abhängigkeiten zwischen den Kommunikationspartnern. Dies wirkt sich beim Manetho-Protokoll unmittelbar auf die Größe der Antezedenzgraphen aus. Wie zu erkennen ist, liegt der Overhead bei Manetho deutlich über dem von Pessimistic Logging. Selbst bei Anwendung verbesserter Verfahren wie inkrementelles Senden der Antezedenzgraphen und periodisches Checkpointing (über Checkpoints gesicherte Informationen müssen nicht mehr mit den Antezedenzgraphen gesendet werden) bleibt der Overhead des Manetho-Protokolls unter den gegebenen Randbedingungen deutlich größer als der von Pessimistic Logging – zumal Pessimistic Logging durch Anwendung des Lazy Output Commit nochmals deutlich verbessert wird. Um eine geeignete Erstellungsrate der Checkpoints für das gegebene System bei Anwendung des Manetho-Protokolls zu ermitteln, wurde das Datenaufkommen bei verschiedenen Erstellungsraten ermittelt, wie in Abbildung 8.15(c) dargestellt. Schließlich wurde eine Rate von 4 (Erstellung des Checkpoints nach jeder 4. empfangenen Nachricht) für die weiteren Betrachtungen angenommen. Auch in Abbildung 8.15(b) (relativer Overhead) lässt sich erkennen, dass die Größe der Antezedenzgraphen beim Causal Logging stark abhängig ist vom Kommunikationsverhalten der Teilnehmer: Sofern die Teilnehmer in zufälligen Mustern (schwarze Linie) kommunizieren, entsteht mehr Overhead als bei geordneten Mustern (graue Linie).

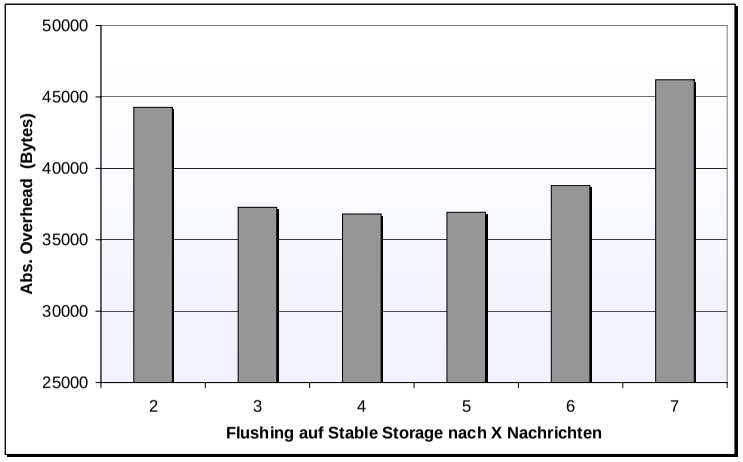
Wie schon bei der Betrachtung der Zugriffslatenz zeigt sich auch hier, dass der Einsatz von Causal Logging im Vergleich zum einfachen Pessimistic Logging unter den gegebenen Randbedingungen keinen Vorteil bietet – zumal das Protokoll wesentlich komplexer ist als Pessimistic Logging und sein Einsatz damit mit entsprechend großem Aufwand verbunden ist. In der Literatur findet sich die Aussage, dass Causal Logging einen geringeren Overhead erzeuge als Pessimistic Logging [28]. Diese Aussage ist richtig ab einer bestimmten Größe der übermittelten Nutzdaten: Erst wenn die Größe der Nachrichten bei Pessimistic Logging der Größe der Antezedengraphen des Causal Logging im Mittel entspricht, erzeugen die Protokolle gleich viel Overhead. Dies ist in Abbildung 8.15(d) veranschaulicht. So würde bei dem gegebenen System der Einsatz des Manetho-Protokolls erst ab einer Nachrichtengröße von ca. 350 Byte lohnen. Unter den gestellten Annahmen liegt die Größe der Nachrichten jedoch im Bereich von unter 100 Byte.



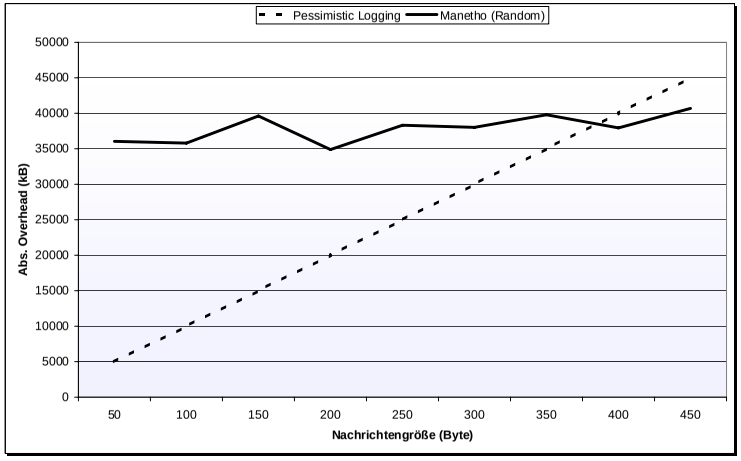
(a) Absoluter Overhead.



(b) Relativer Overhead.



(c) Optimale Checkpoint-Rate.



(d) Overhead vs. Nachrichtengröße.

Abb. 8.15.: Vergleich des Overheads von Manetho (Causal Logging) und Pessimistic Logging.

9. Systemintegration

Auf Basis des im ersten Teil der vorliegenden Arbeit entwickelten Systems wurden im zweiten Teil dedizierte Fehlertoleranzerweiterungen entwickelt. Es bleibt zu klären, wie die drei entstandenen Komponenten, die Proxy-basierte Infrastruktur, die Mechanismen zur Migration von Proxies sowie die Unterstützung von Rollback-Recovery-Protokollen zwischen Proxies und ihren Control Points, möglichst nahtlos in ein Gesamtsystem integriert werden können.

Bei der Integration kann ausgenutzt werden, dass die Durchführung der Rollback-Recovery-Verfahren intern zwischen den Proxies (bzw. den Transceivern) und den Control Points geschieht und von den Proxies und Control Points *aktiv* durchgeführt wird. Das heißt, für die Application Manager findet die Durchführung der Rollback-Recovery-Verfahren transparent statt. Die Migration hingegen wird von den Application Managern bestimmt. Hier verhalten sich die Proxies passiv bzw. agieren erst auf Anweisung durch die Application Manager. Dadurch sind beide Erweiterungen, die Migration und die Rollback-Recovery-Verfahren, klar voneinander abgegrenzte, modulare Komponenten, die der im ersten Teil der Arbeit entstandenen Infrastruktur bedarfsgerecht und in beliebiger Kombination hinzugefügt werden können.

Das entstandene Gesamtsystem mit Proxy-Migration und Rollback-Recovery-Unterstützung ist in Abbildung 9.1 dargestellt. Im Unterschied zu Abbildung 3.2 (Basissystem) ist ein Gerät hinzugekommen, das das in Abschnitt 8.2.4 vorgestellte Stable Storage zur Verfügung stellt. Dieses Gerät kann einer der Application Manager des Systems sein. Um zu entscheiden, welcher Application Manager das Stable Storage bereit stellt, können (abgesehen von einer manuellen Konfiguration) die in Abschnitt 4.2 vorgestellten Aushandlungsverfahren eingesetzt werden. Alternativ können auch mehrere Application Manager ein Stable Storage bereit stellen.

Es bleibt weiterhin zu klären, in welchen Fällen sich die Migration eines Proxys lohnt. Eine Alternative zur Migration liegt darin, den Ausfall eines Proxys einfach zuzulassen und seinen Zustand nach dem Neustart auf einem anderen Host durch den Transceiver oder ein angewendetes Rollback-Recovery-Verfahren wieder herzustellen. Durch den Transceiver kann der Zustand des Proxys generell nur dann wieder hergestellt werden, wenn der komplette Zustand des Proxys im Speicher des Transceivers Platz findet. Doch selbst, wenn das der Fall ist, ist eine Übertragung der Zustandsdaten über das LAN im Rahmen der Migration schneller, als die Übertragung des Zustandes über die schmalbandige Verbindung zwischen Transceiver und Proxy – zumal die Ressourcen des Transceivers auf diese Weise nicht durch die Datenübertragung belastet werden. Würde ein Rollback-Recovery-Verfahren angewendet, um den Zustand des Proxys wieder herzustellen, so müsste

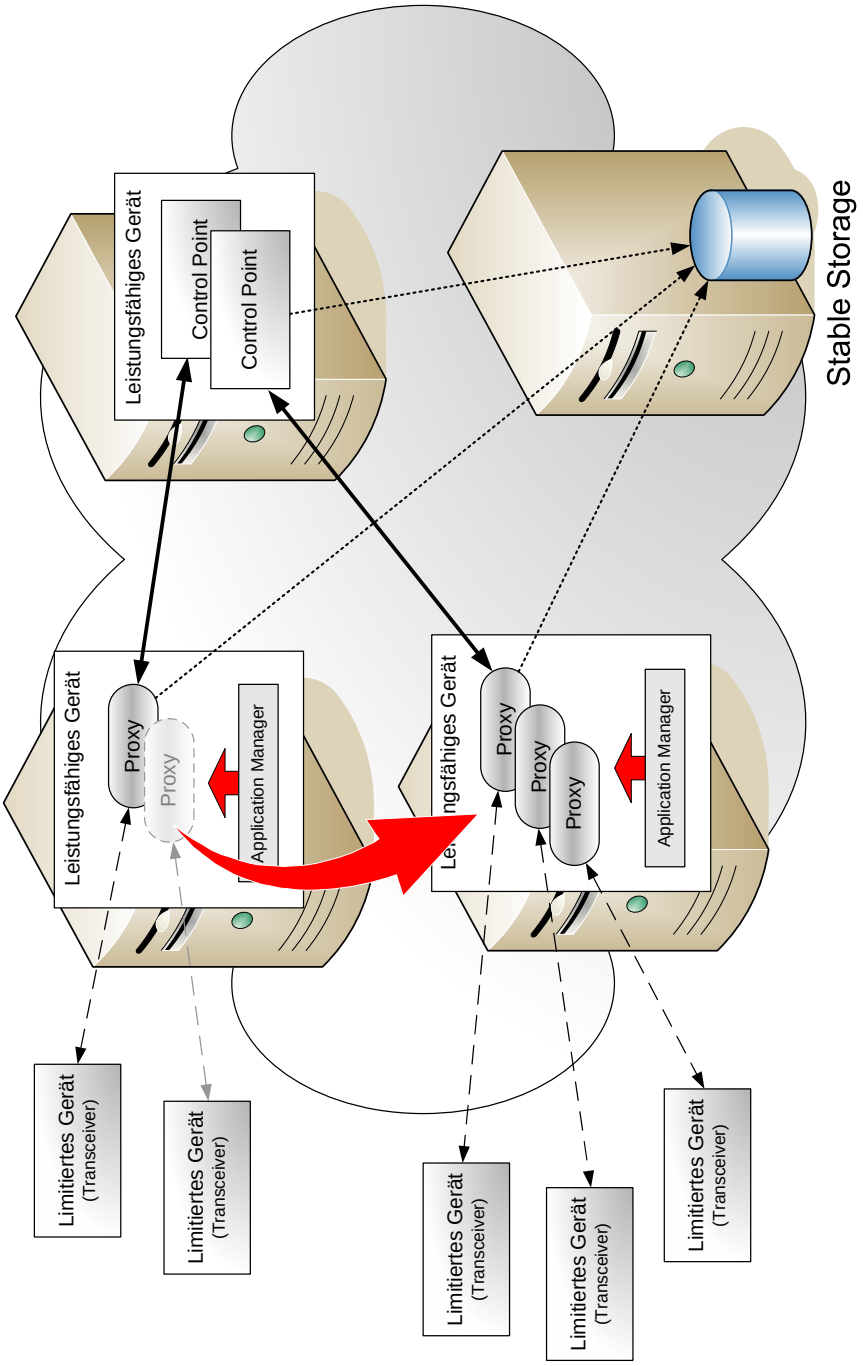


Abb. 9.1.: Gesamtszenarium mit Migration und Rollback-Recovery.

der letzte Zustand in mehreren Schritten wieder angefahren werden, anstatt ihn in einem einzigen Vorgang wieder herzustellen, wie es bei einer Migration der Fall wäre. Dieses „Anfahren“ ist in jedem Fall mit weiterem Kommunikationsoverhead verbunden wie dem Bezug der Antezedengraphen bei Anwendung des Manetho-Protokolls oder dem Neu-Senden von Nachrichten bei Einsatz von Pessimistic Logging. Zuletzt muss berücksichtigt werden, dass die Migration eines Proxys als Mittel zur Störungsvermeidung ein geordneter Vorgang ist, bei dem für die Control Points keine Störung sichtbar wird, während ein hingenommener Ausfall des Proxys unmittelbare Auswirkungen auf seine Kommunikationspartner haben kann.

10. Fazit

Der Fokus der vorliegenden Arbeit lag auf der Integration limitierter Geräte (sog. *Transceiver*) in Service-orientierte Architekturen (SOA), die auf Middleware-Systemen aufbauen. Nachdem die Problematik aufgezeigt wurde, die zumeist Mikrocontroller-basierten Transceiver mit Middleware-Unterstützung auszustatten, wurde als gangbarer Weg die Verwendung von Geräte-Proxies gewählt: Jeder Transceiver bekommt einen Proxy zur Seite gestellt, der zum einen die Middleware-Anbindung für den Transceiver übernimmt, zum anderen aber auch applikationsspezifische Aufgaben des Transceivers übernehmen kann. Um die Verwendung der Geräte-Proxies transparent zu gestalten, wurde im Rahmen der Arbeit ein System zur automatischen Lebenszyklusverwaltung von Geräte-Proxies entwickelt und für UPnP-basierte SOA implementiert. Die Kernkomponente dieses Systems, der *Application Manager*, ist Teil des *Sindrion*-Systems, das mittlerweile in verschiedenen weiteren Forschungsprojekten eingesetzt wird.

Generell wird die Ausfallwahrscheinlichkeit eines Systems durch den Einsatz von Geräte-Proxies ohne weitere Maßnahmen erhöht. Dies hat zwei Gründe: Zum einen fällt der Dienst eines Transceivers nicht nur aus, wenn der Transceiver selbst ausfällt, sondern auch, wenn der zugehörige Proxy ausfällt. Zum anderen können beim Neustart eines Proxys Inkonsistenzen entstehen, die die System-Integrität beeinträchtigen. Um diesen Problemen zu begegnen wurden im Rahmen der Arbeit zwei Ansätze für die Vermeidung von bzw. den Umgang mit Störungen untersucht: Wenn der drohende Ausfall eines Proxys erkannt wird (z. B. knappe Ressourcen des Hosts), so kann der Proxy zur Laufzeit auf einen anderen Host *migriert* werden. Sofern der Ausfall eines Proxys sich nicht durch Migration vermeiden lässt oder spontan geschieht, wird er von einem Application Manager automatisch neu gestartet. Um dabei mögliche Inkonsistenzen zu vermeiden, wurden *Rollback-Recovery*-Verfahren eingesetzt. Hierzu wurden zuerst gängige Rollback-Recovery-Verfahren beleuchtet. Anhand einer Anforderungsanalyse wurden schließlich *Pessimistic Logging* und *Causal Logging* als mögliche Kandidaten für den Einsatz als Rollback-Recovery-Protokolle für das betrachtete System ermittelt. Diese beiden Verfahren wurden im Anschluss eingehend evaluiert. Bei keinem der Verfahren wird der Transceiver weder zusätzlich belastet, noch muss er eine spezielle Unterstützung für die Verfahren mitbringen. Allerdings zeigte sich, dass das vermeintlich leistungsfähigere Causal Logging seine Vorteile aufgrund der angenommenen häufigen Kommunikation mit der Außenwelt (*Output Commit*) und den vergleichsweise kleinen Längen der übermittelten Nachrichten, nicht ausspielen kann. Ferner wurde gezeigt, dass der Overhead von Pessimistic Logging durch eine Anpassung an Proxy-basierte Systeme, die *Lazy Output Commit* genannt wurde, erheblich verbessert werden konnte.

Damit können die in Abschnitt 3.4 gestellten Anforderungen als erfüllt betrachtet werden: So ist das entstandene System fähig zur Selbstkonfiguration, da limitierte Geräte automatisch erkannt, sowie Proxies automatisch bezogen und ausgeführt werden. Zuverlässigkeit ist zum einen Design-inhärent vorhanden, da Proxies z. B. automatisch neu gestartet werden. Zum anderen wird Zuverlässigkeit durch dedizierte Fehlertoleranzmaßnahmen, wie die Migration von Proxies oder Rollback-Recovery erreicht. Zuletzt ist das entstandene System sehr nah an bestehende Standards angelehnt und in weiten Teilen sogar Standard-konform.

Anhang

A. Architektur des Sindrion-Systems

A.1. Der Sindrion Application Manager

Im Rahmen der vorliegenden Arbeit wurde eine Software für die Lebenszyklusverwaltung von Proxies für das Sindrion-Projekt – der *Sindrion Application Manager* – spezifiziert [77] und implementiert. Die Implementierung erfolgte in Java unter Verwendung der Java Standard Edition 6. Der Grund für die Wahl von Java liegt primär in der Fähigkeit von Java, Klassen zur Laufzeit in die JVM zu laden und dynamisch zu binden. Dies ist eine wichtige Anforderung an die Ausführung von Proxies (siehe Abschnitt 4.4). Die Architektur des Application Managers folgt dem *Observer Design Pattern* [32] – die einzelnen Komponenten des Application Managers kommunizieren über Ereignisnachrichten. Die Architektur des Application Managers ist in Abbildung A.1 dargestellt. Die einzelnen Komponenten der Architektur werden in den folgenden Abschnitten näher erläutert. Für eine nähere Beschreibung von Abläufen innerhalb des Application Managers sei auf Abschnitt A.1.1 verwiesen.

Discovery Manager

Aufgabe des Discovery Managers ist es, UPnP-Geräte und speziell Sindrion-Geräte in der Netzwerkumgebung zu finden. Im Falle von herkömmlichen UPnP-Geräten wird ein Discovery-Event einfach an den Device Item Manager (s. u.) durchgereicht. Weiterhin wird jedes UPnP-Gerät auf Vorhandensein der Sindrion-Merkmale (siehe Abschnitt 4.1) geprüft. Sofern diese Merkmale vorhanden sind, wird der Device Item Manager entsprechend benachrichtigt. Allgemein ist der Discovery Manager in der Lage, folgende Gerätetypen zu unterscheiden:

- Herkömmliche UPnP-Geräte
- Sindrion-Transceiver
- Sindrion-Proxies
- Andere Application Manager

Device Item Manager

Vom Discovery Manager entdeckte Geräte werden in einem „Weltmodell“ innerhalb des Device Item Managers abgebildet. Die Abbildung eines realen, *äußeren* Gerätes ist ein *Device Item*. Jedes Device Item hat seinen eigenen Typ, der dem Typ des äußeren Gerätes entspricht. Darüber hinaus enthält jedes Device Item einen Zustandsautomaten, dessen

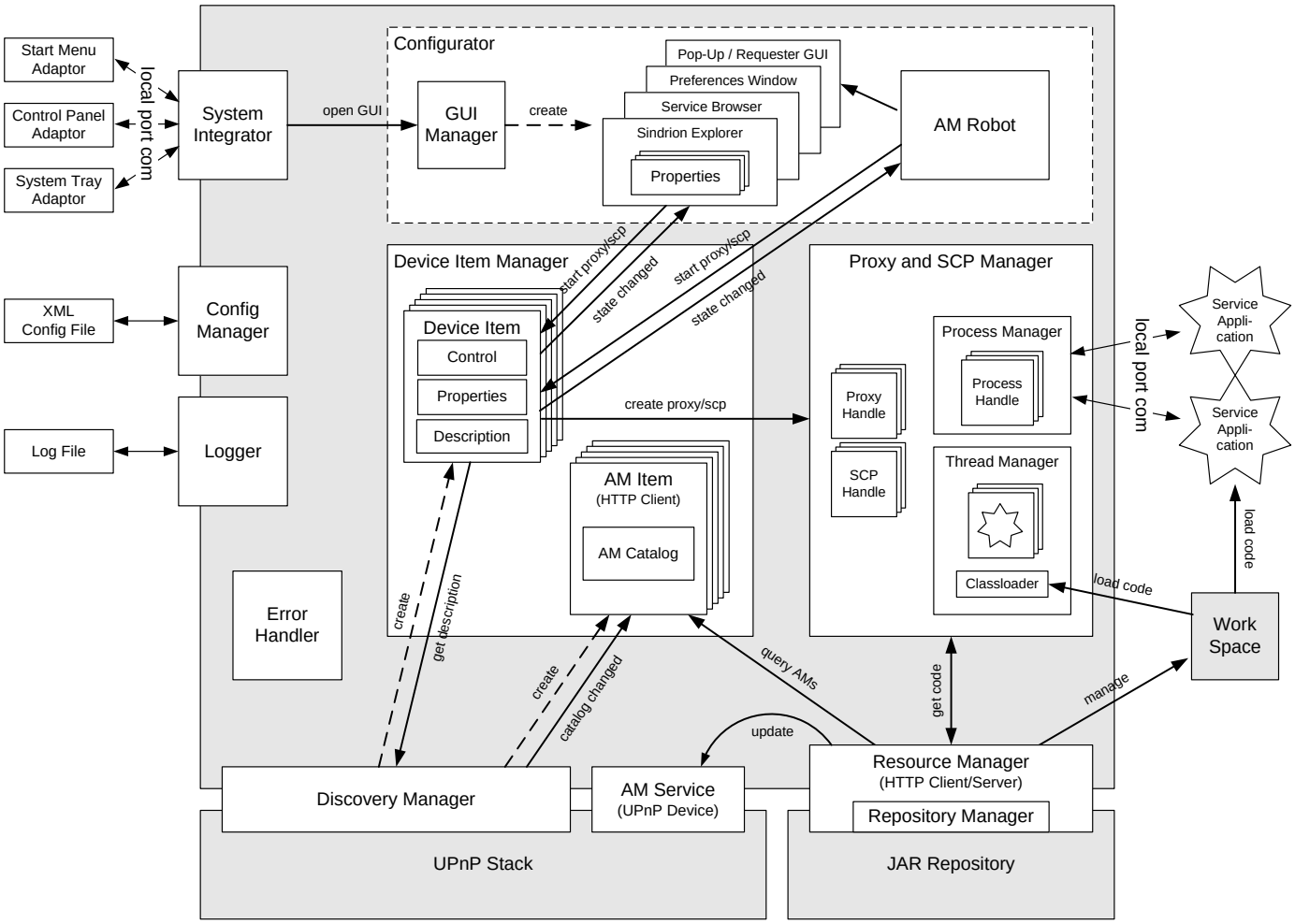


Abb. A.1.: Architektur des Sindrion Application Managers.

Übergänge durch äußere Events angeregt werden und der in der Lage ist, Kontrollnachrichten an das System abzusetzen. Als Beispiel sei eine Nachricht genannt, die von einem Device Item gesendet wird, um einen Proxy zu starten. Die Zustandsautomaten der Device Items werden in Abschnitt A.1.1 näher beschrieben.

Proxy/SCP Manager

Events zum Starten bzw. Beenden von Proxies und Specific Control Points (SCPs) werden von den jeweiligen Device Items an den Proxy/SCP Manager gesendet. Es ist Aufgabe des Proxy/SCP Managers Proxies bzw. SCPs auszuführen. Die Ausführung erfolgt entweder als Thread oder als Prozess – also in einer separaten VM. Doch bevor eine Komponente ausgeführt werden kann, muss der entsprechende Code vom Resource Manager bereitgestellt werden.

Resource Manager

Es ist Aufgabe des Resource Managers, den Code für Proxies bzw. SCPs auf Anfrage bereitzustellen. Dies kann aus drei unterschiedlichen Quellen erfolgen:

1. Aus dem lokalen Repository
2. Aus einem Remote-Repository
3. Direkt vom Transceiver

Im lokalen Repository wird der Code zusammen mit Zeit-, Namens- und Versionsinformationen vorgehalten. Über die Zeitangaben lassen sich die ältesten Komponenten bei Bedarf entfernen, um den Speicherbedarf zu reduzieren.

Configurator

Im Package *Configurator* befinden sich sämtliche GUI-Komponenten, die zum Betrieb bzw. zur Konfiguration des Application Managers erforderlich sind. Abbildung 4.8 zeigt als Beispiel den *Sindrion Explorer* sowie das *Sindrion Application Center*.

Supplemental

Das Package *Supplemental* enthält Hilfskomponenten, die für den Betrieb des Application Managers notwendig sind, wie z. B. eine Konfigurationsverwaltung, Logger oder eine Schnittstelle zum Windows System Tray (siehe Abschnitt 4.6).

A.1.1. Beschreibung der Abläufe in den Zustandsautomaten

Abbildungen A.2 bis A.5 zeigen die Zustandsdiagramme der Automaten, die mit Ausnahme der Gerätetyperkennung im *Device Item Manager* ausgeführt werden. Alle Automaten durchlaufen Zustände, die den Lebenszyklen der jeweils repräsentierten Geräte entsprechen.

Discovery und Detektion des Gerätetyps

Der Zustandsautomat für die Erkennung des Gerätetyps ist im *Discovery Manager* realisiert. Abbildung A.2 zeigt den entsprechenden Zustandsautomaten. Der Discovery Manager setzt auf den UPnP-Stack auf und empfängt eine Nachricht, sobald ein neues UPnP Device entdeckt wurde. Anschließend prüft er das Device auf Merkmale, die auf den genauen Typ des Gerätes schließen lassen. Handelt es sich um ein UPnP Basic Device und verweist der Eintrag der Presentation Page in der UPnP Description auf die Datei *SindrionTX.html*, so wird das Gerät als Sindrion Transceiver eingestuft. Anschließend wird ein entsprechendes *Device Item* im *Device Item Manager* erzeugt und initialisiert. Für eine Einstufung als Sindrion Application Manager muss die Datei der Presentation Page den Namen *SindrionAM.html* tragen, wobei das Gerät ein vollständiges UPnP Device (also kein Basic Device) sein muss. Ein Sindrion Proxy ist ein vollständiges UPnP Device, dessen Presentation Page den Dateinamen *SindrionProxy.html* hat. Trifft keines dieser Erkennungsmerkmale zu, wird das Gerät als *Plain UPnP Device* eingestuft. Falls bei der Bearbeitung eines Device Items ein Fehler auftritt, kann das Gerät auch nachträglich als Plain UPnP Device eingestuft werden. Hier liegt die Überlegung zu Grunde, dass ein unvollständiges bzw. fehlerhaftes Sindrion UPnP Device nicht mehr als ein herkömmliches UPnP Device ist.

Sindrion Transceiver Device Item

Der Zustandsautomat des *Sindrion Transceiver Device Item* ist in Abbildung A.3 dargestellt. Es ist Aufgabe des Sindrion Transceiver Device Items, den Transceiver zu repräsentieren und die Ausführung des Proxys zu überwachen. Im ersten Schritt lädt das Sindrion Transceiver Device Item die *Sindrion Description* vom Transceiver herunter. Die entsprechende Adresse wurde zuvor vom Discovery Manager übergeben. Anschließend erfolgt die Auswertung der Sindrion Description. Sofern bei diesen Schritten kein Fehler (fehlende oder korrupte XML-Datei) auftritt, geht der Automat anschließend in den *No Proxy State* über. Dieser repräsentiert den Zustand, dass für den Transceiver noch kein Proxy vorhanden ist. Der Start eines Proxys kann nun entweder auf dem lokalen Application Manager oder aber auf einem entfernten Application Manager erfolgen. Erfolgt der Start entfernt, so geht der Automat in den Zustand *External Proxy State* über. Der AM bemerkt den Start des entfernten Proxys, da ein UPnP Device mit der UUID des Proxys entdeckt wurde, das nicht vom lokalen AM selbst ausgeführt wird. Wird der Start des Proxys vom lokalen AM selbst initiiert, nimmt der Automat den Zustand *Negotiating Proxy State* ein. In dieser Phase wird der Code des Proxys vom *Proxy/SCP Manager* bezogen und ausgeführt. Sobald der Proxy gestartet ist, nimmt der Automat den Zustand *Local Proxy State* ein.

Sindrion Proxy Device Item

Die linke Seite von Abbildung A.4 illustriert den Ablauf innerhalb des *Sindrion Proxy Device Item*. Der Zustandsautomat durchläuft ebenfalls, wie im obrigen Fall, die Zustände *Downloading Description* und *Parsing Description*. Anschließend geht der Automat über in den Zustand *Proxy Active*.

Plain UPnP Device Item

Der Zustandsautomat des *Plain UPnP Device Item* (Abbildung A.4 rechts) repräsentiert den Lebenszyklus eines herkömmlichen UPnP-Gerätes. Dabei kann es sich auch um ein Gerät handeln, das zuerst als Sindrion-Gerät eingestuft und aufgrund eines Fehlers „umgestuft“ wurde.

Sindrion AM Device Item

Das *Sindrion AM Device Item* repräsentiert einen entfernten Application Manager. Sein Zustandsautomat ist in Abbildung A.5 dargestellt. Normalerweise befindet sich der Automat im Zustand *Idle*. Zu Beginn oder wenn der Inhalt des Repositorys des entfernten AMs sich geändert hat, spring der Automat in den Zustand *Downloading Catalog State*. Nach erfolgreichem Download wird der Katalog ausgewertet (*Parsing Catalog State*). Nach erfolgreicher Auswertung geht der Automat wieder in den Zustand *Idle* über.

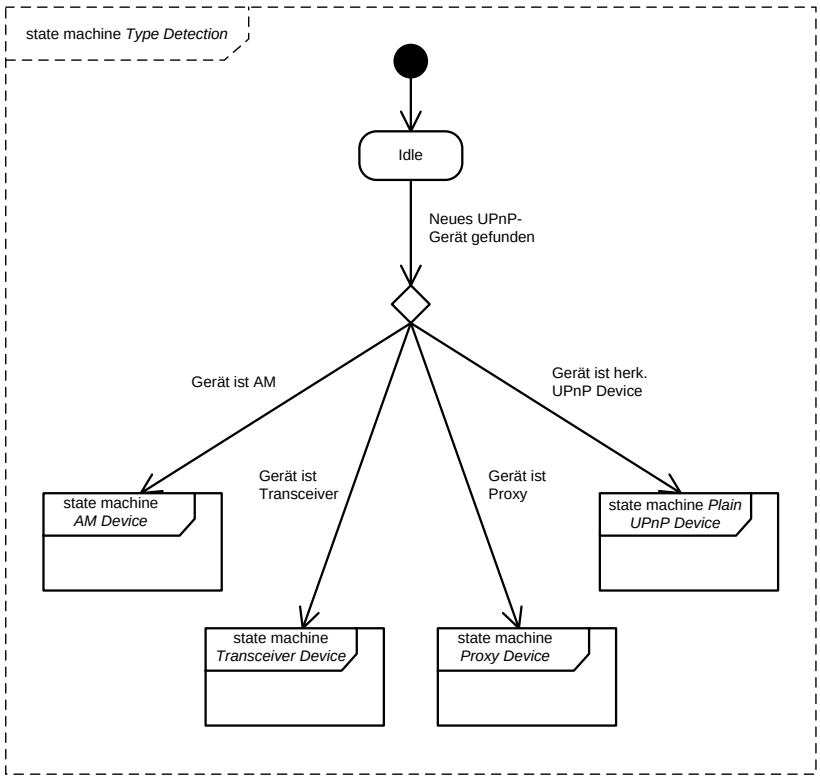


Abb. A.2.: Zustandsdiagramm der Gerätetyperkennung des Discovery Managers.

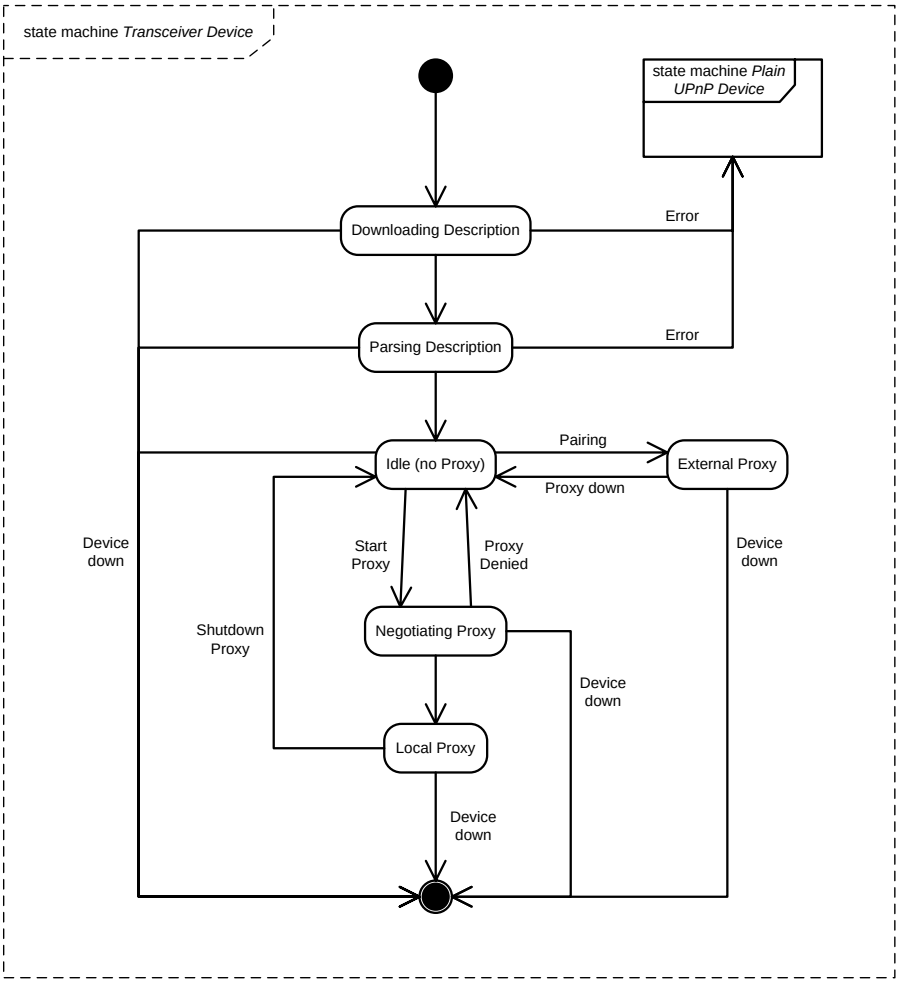


Abb. A.3.: Zustandsdiagramm des *Transceiver Device*.

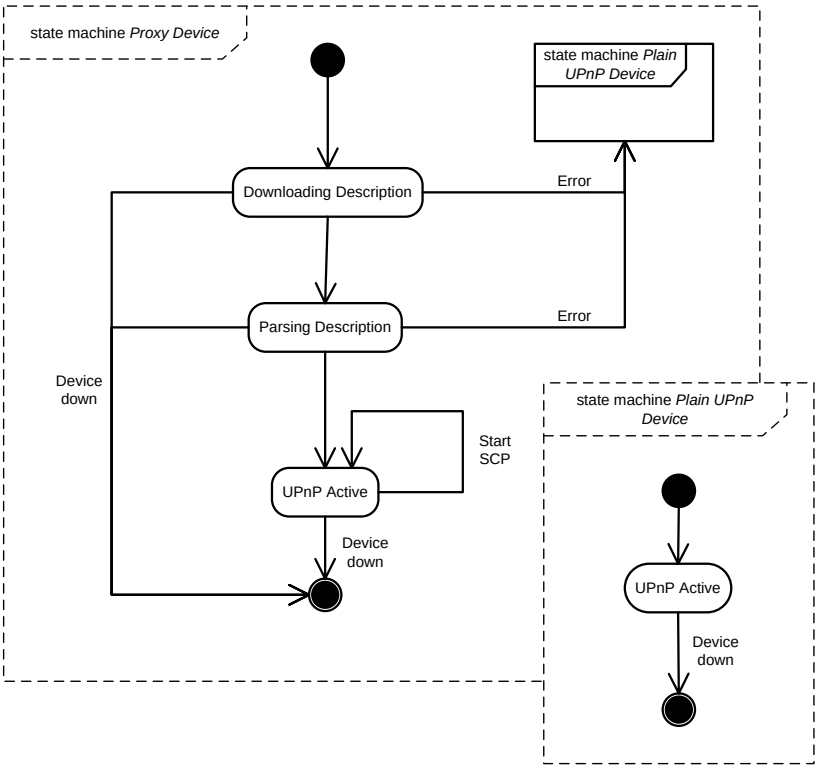


Abb. A.4.: Zustandsdiagramme von *Proxy Device* sowie *Plain UPnP Device*.

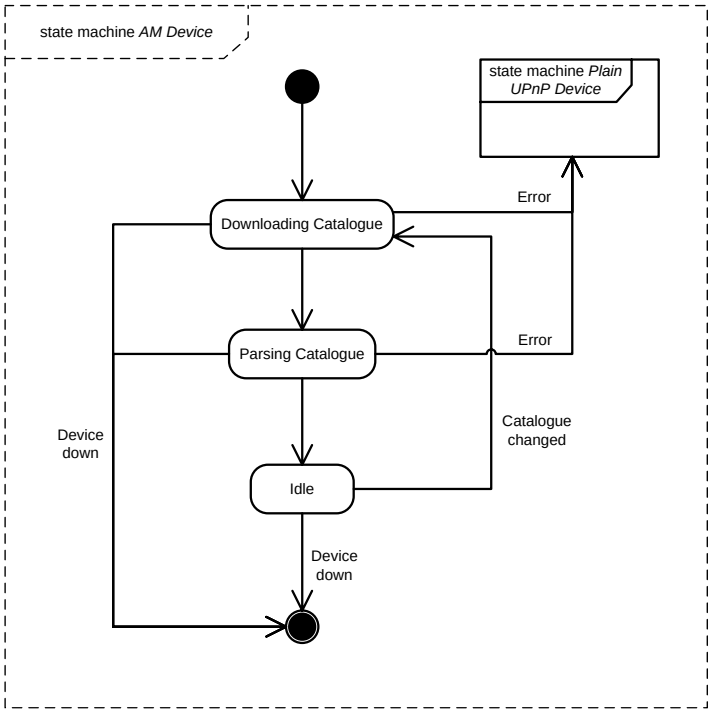


Abb. A.5.: Zustandsdiagramm des *AM Device*.

A.1.2. Beispiel eines Proxy-Lebenszyklus

Im Folgenden wird beispielhaft der Lebenszyklus eines im Sindrion Application Manager ausgeführten Proxys beschrieben.

1. Der *Discovery Manager* detektiert ein neues Gerät und stellt fest, dass es sich um einen Sindrion-Transceiver handelt.
2. Im *Device Item Manager* wird ein neues Transceiver Device Item erzeugt. Dieses durchläuft den in Abbildung A.3 dargestellten Zustandsautomaten. Es initiiert die Auswertung der *Sindrion Description*.
3. Entsprechend der Angaben in der Sindrion Description wird der *Resource Manager* beauftragt, den Code für den Proxy zu beziehen. Zuerst überprüft er das lokale Repository. Wir gehen davon aus, dass er den Code dort nicht vorfindet. Nun fragt er die Repositories sämtlicher erkannter Application Manager ab, um herauszufinden, ob einer von ihnen den Code vorhält. Ist auch das nicht der Fall, initiiert er den Download direkt vom Transceiver und fügt den Code in sein lokales Repository ein.
4. Der *Proxy/SCP Manager* führt den Proxy als Thread oder als Prozess (separate VM) aus. Der Proxy verbindet sich mit dem Transceiver. Die Präsenz des Proxys wird von den Application Managern im System detektiert. Die entsprechenden Proxy Device Items werden erzeugt und gehen schließlich über in den Zustand *External Proxy* – das lokale Device Item nimmt hingegen den Zustand *Local Proxy* ein (siehe Abbildung A.4).
5. Sobald der Sindrion Transceiver die Netzgemeinschaft wieder verlässt und der Application Manager, der den Proxy ausführt, eine *Byebye-Message* vom Transceiver empfängt, wird auch der Proxy wieder beendet. Zuvor sendet der Proxy selbst Byebye-Messages, worauf alle entsprechenden *Sindrion Proxy Device Items* aus den Application Managern (lokal und remote) entfernt werden.

A.2. Der Sindrion-Transceiver

Der Prototyp des Sindrion-Transceivers basiert auf einem PCB[1] mit herkömmlichen Bauteilen [8]. Der RF-Part basiert auf einem ISM-Band-Transceiver (433 MHz - 915 MHz), der eine Bruttodatenrate von 50 kbit/s bereit stellt. Kern des Transceivers ist ein 16-Bit Mikrocontroller, der mit 9 MHz betrieben wird. Der Programmspeicher ist 64 kB groß, während 16 kB Arbeitsspeicher zur Verfügung stehen. Die durchschnittliche Leistungsaufnahme liegt bei weniger als 1 mW. Wie in Abbildung A.6 dargestellt, besitzt der Sindrion Transceiver drei *Power Domains*: Im Standby ist lediglich das *RF Front End* (Domain 1) aktiv. Sobald Funksignale empfangen werden, wird der MAC-Layer (Domain 2) aktiviert und versucht, aus den empfangenen Signalen gültige Daten-Frames zu ermitteln. Sobald dies gelingt, wird die eigentliche Service Application und die Unterstützung für das Sindrion Control Protokoll (Domain 3) aktiviert.

[1] Printed Circuit Board

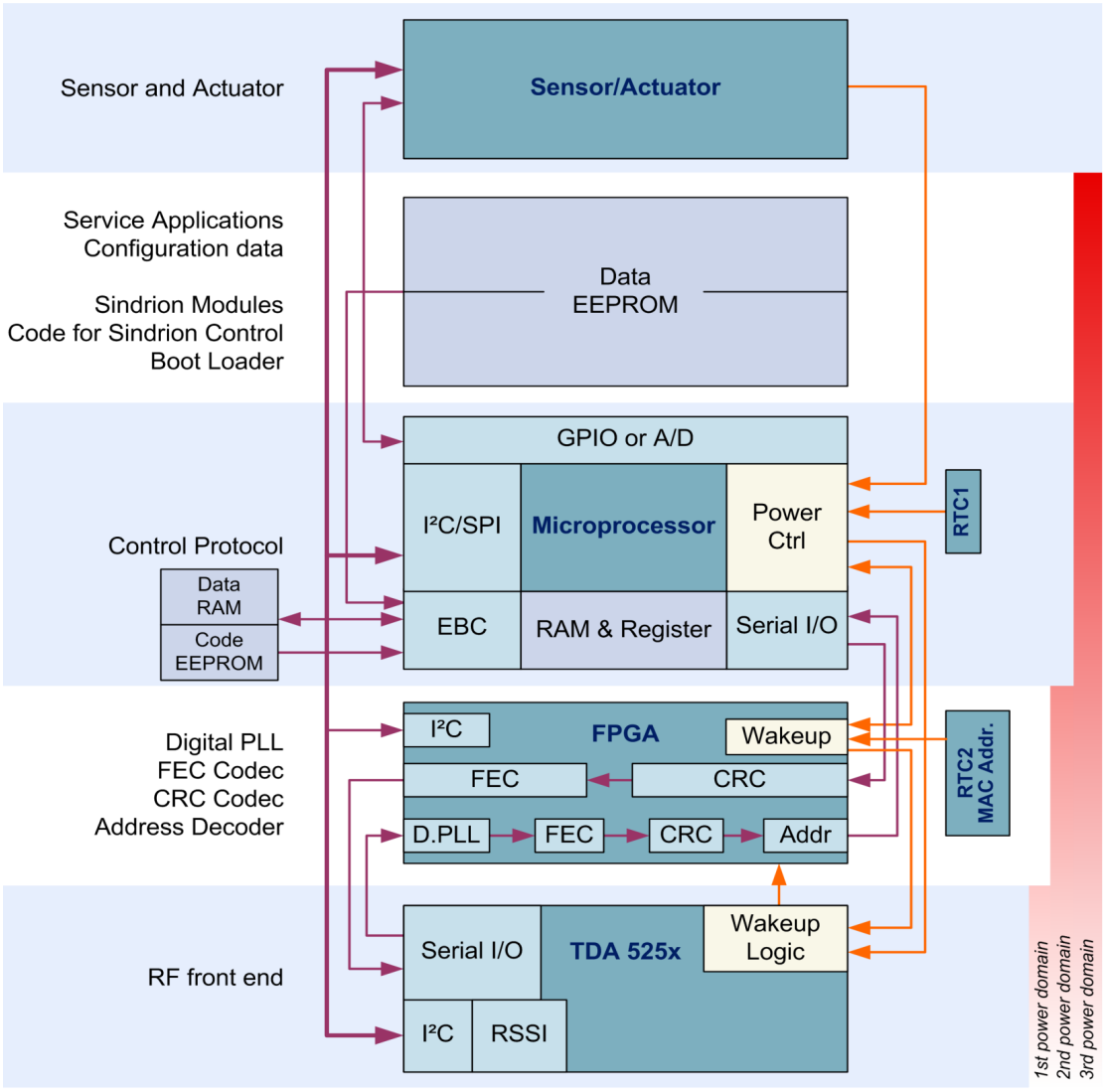


Abb. A.6.: Power Domains des Sindrion Transceivers. (Quelle: Infineon)

B. Beispiele von UPnP und Sindrion Descriptions

Die XML-basierten UPnP Device und Service Descriptions dienen der Beschreibung der Eigenschaften und Schnittstellen von UPnP Devices [98]. Die Dateien werden vom eingebetteten Webserver der UPnP-Geräte zum Download bereit gestellt und von UPnP Control Points bezogen und ausgewertet. Device und Service Descriptions werden in separaten Dateien bereitgestellt, damit ein Control Point gezielt die Service Descriptions beziehen kann, die er benötigt.

Die Sindrion Description ist in Anlehnung an die UPnP Descriptions entstanden. Sie wird vom Sindrion Transceiver bereit gestellt und gibt Auskunft über die Bezugsmöglichkeiten des Sindrion Proxys und des SCPs [87].

B.1. UPnP Device Description

Listing B.1 zeigt die Device Description eines *DimmableLight*. Die Zeilen 2 - 6 erklären, dass es sich um eine standardisierte Device Description handelt, die in der Version 1.0 vorliegt. Die Angabe des Geräte-Typs steht in Zeile 8. Neben allgemeinen Geräteinformationen (Zeilen 9 - 16) und Angaben über bereitgestellte Icon-Dateien zur Anzeige im Control Point (Zeilen 18 - 26), enthält Zeile 17 die UUID des Devices. Die Zeilen 27 - 42 enthalten eine Liste der bereitgestellten Services des Gerätes. Neben einer Angabe des standardisierten Service-Typs (z. B. *DimmingService*, Zeile 36) wird auf die zugehörige Service Description (Zeile 38) verwiesen.

Lst. B.1: Beispiel einer UPnP Device Description.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
   <specVersion>
      <major>1</major>
      <minor>0</minor>
   </specVersion>
   <device>
      <deviceType>urn:schemas-upnp-org:device:DimmableLight:1</deviceType>
      <friendlyName>IFX Light Bulb (Cora)</friendlyName>
      <manufacturer>Infineon Technologies AG</manufacturer>
      <manufacturerURL>http://www.infineon.com</manufacturerURL>
      <modelDescription>A Simulated Test Device</modelDescription>
      <modelName>Infineon CPR ET Emulated Light Bulb</modelName>
      <modelNumber>IFX-CPR-ET-L1</modelNumber>
```

```
        <modelURL>http://intra.muc.infineon.com/cpr/topics/
            Emerging_technologies/</modelURL>
16      <serialNumber>123456780</serialNumber>
        <UDN>uuid:10e9d7945d0+1fa3025a+3c6f79a2+139aeb</UDN>
18      <iconList>
           <icon>
20            <mimetype>image/png</mimetype>
              <width>32</width>
22            <height>32</height>
              <depth>8</depth>
24            <url>icon.png</url>
           </icon>
26      </iconList>
        <serviceList>
28         <service>
              <serviceType>urn:schemas-upnp-org:service:SwitchPower:1</
                  serviceType>
30            <serviceId>urn:upnp-org:serviceId:SwitchPower.1</serviceId>
              <SCPDURL>SwitchPower_1_description.xml</SCPDURL>
32            <controlURL>SwitchPower_1_control</controlURL>
              <eventSubURL>SwitchPower_1_event</eventSubURL>
34         </service>
           <service>
36            <serviceType>urn:schemas-upnp-org:service:DimmingService:1</
                  serviceType>
              <serviceId>urn:upnp-org:serviceId:DimmingService.1</serviceId>
38            <SCPDURL>DimmingService_1_description.xml</SCPDURL>
              <controlURL>DimmingService_1_control</controlURL>
40            <eventSubURL>DimmingService_1_event</eventSubURL>
           </service>
42      </serviceList>
        <presentationURL>presentation.html</presentationURL>
44   </device>
</root>
```

B.2. UPnP Service Description

Die Service Description des *DimmingService* (Listing B.2) gibt Auskunft über die von dem Service bereitgestellten Actions. Die Action *GetLoadLevelTarget* z. B. ist in den Zeilen 18 - 27 beschrieben. Die Action hat als Rückgabewert den aktuellen Lichtwert des *DimmableLight* (siehe Zeilen 22, 23) und verweist auf die zugehörige Zustandsvariable *LoadLevelTarget*, die in den Zeilen 40 - 48 durch ihren Datentyp, ihren Default-Wert und ihren zulässigen Wertebereich beschrieben ist.

Lst. B.2: Beispiel einer UPnP Service Description.

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <scpd xmlns="urn:schemas-upnp-org:service-1-0">
3     <specVersion>
         <major>1</major>
5        <minor>0</minor>
```

```
    </specVersion>
7   <actionList>
      <action>
9       <name>SetLoadLevelTarget</name>
        <argumentList>
11        <argument>
            <name>NewLoadLevelTarget</name>
13          <direction>in</direction>
            <relatedStateVariable>LoadLevelTarget</relatedStateVariable>
15        </argument>
        </argumentList>
17    </action>
      <action>
19      <name>GetLoadLevelTarget</name>
        <argumentList>
21        <argument>
            <name>RetLoadLevelTarget</name>
23          <direction>out</direction>
            <relatedStateVariable>LoadLevelTarget</relatedStateVariable>
25        </argument>
        </argumentList>
27    </action>
      <action>
29      <name>GetLoadLevelStatus</name>
        <argumentList>
31        <argument>
            <name>RetLoadLevelStatus</name>
33          <direction>out</direction>
            <relatedStateVariable>LoadLevelStatus</relatedStateVariable>
35        </argument>
        </argumentList>
37    </action>
    </actionList>
39  <serviceStateTable>
      <stateVariable sendEvents="no">
41      <name>LoadLevelTarget</name>
        <dataType>ui1</dataType>
43      <defaultValue>0</defaultValue>
        <allowedValueRange>
45        <minimum>0</minimum>
          <maximum>100</maximum>
47      </allowedValueRange>
      </stateVariable>
49    <stateVariable sendEvents="yes">
        <name>LoadLevelStatus</name>
51      <dataType>ui1</dataType>
        <defaultValue>0</defaultValue>
53      <allowedValueRange>
          <minimum>0</minimum>
55        <maximum>100</maximum>
        </allowedValueRange>
57    </stateVariable>
    </serviceStateTable>
59 </scpd>
```

B.3. Sindrion Description

Listing B.3 zeigt die Sindrion Description eines Transceivers, der als Thermometer arbeitet. Die Datei enthält Angaben zum Sindrion Proxy (Zeilen 8 - 17), zum Specific Control Point (Zeilen 18 - 25), sowie zu dem ebenfalls vom Transceiver bereit gestellten Java Applet Zeilen 26 - 33. Innerhalb eines Bereichs wird auf die Dateien verwiesen, aus denen sich die jeweilige Komponente zusammen setzt. Dabei ist es durchaus möglich, dass mehrere Komponenten die selben Dateien verwenden. Zum Beispiel die Datei *JAR001*, die sowohl vom Proxy (Zeile 11), als auch vom SCP (Zeile 20) verwendet wird, da es sich um die Kernkomponente des UPnP-Stacks handelt, wie in den Zeilen 36 - 43 zu sehen ist. Dort wird neben Angabe der Haupt- und Nebenversionsnummer der Komponente (Zeilen 39 - 42) auf den Dateinamen verwiesen, über den die Datei vom Transceiver bezogen werden kann (Zeile 38).

Lst. B.3: Beispiel einer Sindrion Description.

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <sind:sindrionDescription xmlns:sind="http://www.infineon.com/Sindrion/
       description">
3    <specVersion>
       <major>1</major>
5      <minor>0</minor>
     </specVersion>
7    <name>Sindrion Thermometer</name>
     <application>
9      <proxy description="Thermometer Proxy">
         <mainClass>com.infineon.sindrion.proxyimpl.thermometer.
             ThermometerProxy</mainClass>
11       <file>JAR001</file>
         <file>JAR003</file>
13       <file>JAR004</file>
         <file>JAR007</file>
15       <file>JAR008</file>
         <file>JAR010</file>
17     </proxy>
       <controlPoint description="Thermometer Control Point">
19       <mainClass>com.infineon.sindrion.proxyimpl.thermometer.ThermometerSCP
             </mainClass>
         <file>JAR001</file>
21       <file>JAR002</file>
         <file>JAR006</file>
23       <file>JAR008</file>
         <file>JAR010</file>
25     </controlPoint>
       <applet description="Thermometer Applet">
27       <mainClass>com.infineon.sindrion.dev.wrapper.WrapperApplet</mainClass>
         <file>JAR004</file>
29       <file>JAR006</file>
         <file>JAR007</file>
31       <file>JAR008</file>
         <file>JAR010</file>
33     </applet>
   </application>
```

```
35  <code>
      <jar id="JAR001">
37      <ref>infineon/upnp/v1x0/base.jar</ref>
        <url>ifx-upnp10-base.jar</url>
39      <version>
          <major>2</major>
41        <minor>0</minor>
        </version>
43      <comment>Base archive of the Infineon UPnP 1.0 protocol stack</comment
            >
      </jar>
45    <jar id="JAR002">
        <ref>infineon/upnp/v1x0/controlPoint.jar</ref>
47      <url>ifx-upnp10-cp.jar</url>
        <version>
49        <major>2</major>
          <minor>0</minor>
51      </version>
        <comment>Control Point archive of the Infineon UPnP 1.0 protocol stack
            </comment>
53    </jar>
      <jar id="JAR003">
55      <ref>infineon/upnp/v1x0/device.jar</ref>
        <url>ifx-upnp10-dev.jar</url>
57      <version>
          <major>2</major>
59        <minor>0</minor>
        </version>
61      <comment>Device archive of the Infineon UPnP 1.0 protocol stack</
            comment>
      </jar>
63    <jar id="JAR004">
        <ref>infineon/sindrion/appman/proxyimpl/thermoproxy.jar</ref>
65      <url>thermoproxy.jar</url>
        <version>
67        <major>2</major>
          <minor>0</minor>
69      </version>
        <comment>UPnP Device proxy for Sindrion Transceiver</comment>
71    </jar>
      <jar id="JAR006">
73      <ref>infineon/sindrion/appman/proxyimpl/thermoscp.jar</ref>
        <url>thermoscp.jar</url>
75      <version>
          <major>2</major>
77        <minor>0</minor>
        </version>
79      <comment>Switch Control Point application (core library)</comment>
      </jar>
81    <jar id="JAR007">
        <ref>infineon/sindrion/control/control.jar</ref>
83      <url>control.jar</url>
        <version>
85        <major>2</major>
          <minor>0</minor>
```

```
87        </version>
          <comment>Control library for Sindrion Transceiver</comment>
89      </jar>
        <jar id="JAR008">
91        <ref>infineon/upnp/v1x0/local.jar</ref>
          <url>ifx-upnp10-local.jar</url>
93        <version>
            <major>2</major>
95          <minor>0</minor>
          </version>
97        <comment>Local archive of the Infineon UPnP 1.0 protocol stack</
              comment>
        </jar>
99      <jar id="JAR010">
          <ref>infineon/sindrion/dev/component/development.jar</ref>
101       <url>development.jar</url>
          <version>
103         <major>2</major>
            <minor>0</minor>
105       </version>
          <comment>Library for Sindrion software components</comment>
107     </jar>
    </code>
109 </sind:sindrionDescription>
```

C. Rollback-Recovery-Protokoll „Manetho“

Listing C.1 zeigt beispielhaft die Prozeduren eines aus der Arbeit von Elnozahy et al. [28] entnommenen Recovery-Algorithmus für das Manetho-Protokoll. Zu Beginn des Recovery-Vorgangs stellt eine RU ihren letzten durch einen Checkpoint gesicherten Zustand mit Hilfe des Stable Storage wieder her. Anschließend ruft sie die lokale Prozedur `RECOVER` auf. Parameter dieser Prozedur sind die Kennung der RU p, der Index des aktuellen (wiederhergestellten) Zustandsintervalls c, die aktuelle Inkarnationsnummer $INCNUM$, sowie die Menge aller im System befindlichen RUs S. Bei Aufruf der Prozedur legt die RU zuerst die Inkarnationsnummer $INCNUM$ auf dem Stable Storage ab. Anschließend wird ein neuer Graph G erzeugt, der dem Antezedenzgraphen $AG\sigma_{p,c}$ entspricht, der bei Wiederherstellung des Checkpoints vom Stable Storage bezogen wurde.

Nun ruft die RU p die Prozedur `GET_RU` auf jeder *anderen* RU im System auf. Bei Aufruf auf einer RU q legt `GET_AG` zuerst den aktuellen AG auf dem Stable Storage ab. Nun bestimmt q das *jüngste* Zustandsintervall k der RU p, $\sigma_{p,k}$, das sich im Antezedenzgraphen AG von q befindet. Anschießend wird k dem Vektor $REJECTVEC$ hinzugefügt. Solange die Prozedur `SEND_INC` nicht aufgerufen wird, verweigert q jede Nachricht von anderen RUs deren beigefügter AG den Knoten eines Zustandsintervalls $\sigma_{p,i}$ enthält mit $i > k$. Am Ende der Prozedur `GET_AG` liefert q den Wert $AG(\sigma_{p,k})$ zurück. Kehrt `GET_AG` nach Aufruf aus `RECOVER` zurück, fügt p den zurückgelieferten Graphen AG in seinen Arbeitsgraphen G ein. Weiterhin wird die Inkarnationummer von q zu dem Vektor $INCVEC$ hinzugefügt.

Nun ruft p auf allen anderen RUs die Prozedur `SEND_INC` auf mit den Argumenten p und $INCVEC$. Auf der RU q aktualisiert `SEND_INC` den lokalen Vektor $INCVEC$ und hebt die o. g. Beschränkungen bzgl. des Nachrichtenempfangs auf. Zuletzt stellt p den Zustand direkt vor dem Ausfall $\sigma_{p,m}$ her. Dabei bezieht p Nachrichten von den Message-Logs der jeweiligen Sender und führt interne Events bei Bedarf erneut aus. RU p sendet keinerlei Nachrichten in die Außenwelt oder an andere RUs. Allerdings legt es die Nachrichten, die es gesendet *hätte* in seinem flüchtigen Message-Log ab.

Lst. C.1: Recoveryalgorithmus für das Manetho-Protokoll.

```
1  procedure RECOVER(p, c, INCNUM, S)
     INCNUM ← INCNUM + 1;
3    save INCNUM on stable storage;
     INCVEC[p] ← INCNUM;
5    G ← AG(σ_{p,c});

7    for all q ∈ S, q ≠ p do
       (INQ, AGQ) ← remote call q: GET_AG(p);
9      G ← G ∪ AGQ;
       INCVEC[q] ← INQ;
11
     for all q ∈ S, q ≠ p do
13     remote call q: SEND_INC(p, INCVEC);

15   m ← max j such that σ_{p,j} ∈ G;
     STATEINDEX ← c;
17
     while STATEINDEX ≤ m do
19     execute up to next event without sending application
       messages;
21     STATEINDEX ← STATEINDEX + 1;

23     if next event is a receive then
         request message from sender's log;
25     else
         re-execute internal event;
27   return;


29
   procedure GET_AG(p)
31   save AG on stable storage;
     k ← max j such that σ_{p,j} ∈ AG;
33   REJECTVEC[p] ← k;
     return (INCNUM, σ_{p,k});
35

37 procedure SEND_INC(p, INCVEC_P)
     for all s ∈ S do
39     INCVEC[s] ← max (INCVEC[s], INCVEC_P[s]);
       REJECTVEC[p] ← ∞;
41     return;
```

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] J. Allard, V. Chinta, S. Gundala, and G. G. Richard III. Jini meets UPnP: An architecture for Jini/UPnP interoperability. In *2003 Symposium on Applications and the Internet*, Orlando, Florida, January 2003.

[2] OSGi Alliance. *OSGi Service Platform, Release 3*. IOS Press, Inc., 2003.

[3] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University, Department of Computer Science, 1996.

[4] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *Proceedings of the 15th International Conference of Distributed Computing Systems*, September 1994.

[5] L. Alvisi and K. Marzullo. Trade-offs in implementing optimal message logging protocols. In *Symposium on Principles of Distributed Computing*, pages 58–67, 1996.

[6] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice Hall, 1981.

[7] F. Bagci, J. Petzold, W. Trumler, and T. Ungerer. Ubiquitous mobile agent system in a P2P-network. In *Proceedings of the Annual Conference on Ubiquitous Computing*, Seattle, USA, October 2003.

[8] D. Barisic, D. Bichler, J. Harnisch, T. Herndl, T. Lentsch, M. Loew, M. Krogmann, I. Karls, G. Stromberg, H. Linde, E. Naroska, J. Platte, P. Resch, and P. Schramm. Market entry strategies for wireless sensor networks. In *Tagungsband der Informationstagung Mikroelektronik (ME 2006)*, 2006.

[9] D. Barisic, M. Krogmann, G. Stromberg, and P. Schramm. Making embedded software development more efficient with SOA. In *2nd IEEE Workshop on Service Oriented Architectures in Converging Networked Environments*, 2007.

[10] F. Baude, D. Caromel, F. Huet, and J. Vayssiere. Communicating mobile active objects in Java. In *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer, May 2000.

[11] G. Bertoni Machado, F. Siqueira, R. Mittmann, and C. A. Vieira e Vieira. Integration of embedded devices through web services: Requirements, challenges and early results. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006.

[12] C. Bettstetter and C. Renner. A comparison of service discovery protocols and

implementation of the service location protocol. In *Proc. EUNICE Open European Summer School*, Twente, Netherlands, September 2000.

[13] D. Bichler, E. Naroska, G. Stromberg, H. Linde, M. Fachberger, M. Huemer, M. Pfaff, P. Schramm, R. Knauseder, S. Sorbely, T. Sturm, W. Weber, X. Shi, and Y. Gsottberger. An embedded system architecture for UPnP compatible wireless control networks. In *Proceedings of the 2004 International Symposium on Signals, Systems, and Electronics (ISSSE'04)*, August 2004.

[14] N. Bieberstein, S. Bose, and M. Fiammante. *Service-Oriented Architecture Compass.* Prentice Hall International, 2005.

[15] G. Bollella and J. Gosling. The real-time specification for Java. *IEEE Computer*, 33(6):47–54, 2000.

[16] G. Borriello and R. Want. Embedded computation meets the World Wide Web. *Commun. ACM*, 43(5):59–66, May 2000.

[17] E. Brewer, N. Borisov, M. Chen, R. von Behren, M. Welsh, D. Culler, J. MacDonald, J. Lau, and S. D. Gribble. Ninja: A framework for network services. In *Proceedings of the 2002 Usenix Technical Conference*, Monterey, CA, USA., June 2002.

[18] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the IEEE International Symposium on Reliability, Distributed Software, and Databases*, pages 207–215, December 1984.

[19] B. Brummit, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easy living: Technologies for intelligent environments. In *Symposium on Handheld and Ubiquitous Computing (huc2k)*, pages 25–27. Springer, 2000.

[20] V. Cepa and M. Mezini. MobCon: A generative middleware framework for Java mobile applications. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05)*, 2005.

[21] R. Cerqueira, C. Hess, M. Romn, and R. Campbell. Gaia: A development infrastructure for active spaces, 2001.

[22] H. Chen and T. Finin. An ontology for a context aware pervasive computing environment. In *IJCAI Workshop on Ontologies and Distributed Systems*, 2003.

[23] S. Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), 2004.

[24] C. Clark, K. Fraser, S. Hand, G. J. Hanseny, E. July, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[25] W. S. Conner, L. Krishnamurty, and R. Want. Making every day life easier using dense sensor networks. In *Ubicomp 2001: Ubiquitous Computing, Third International Conference*, pages 49–55. Springer, 2001.

[26] W. K. Edwards and R. E. Grinter. At home with ubiquitous computing: Seven chal-

lenges. In *Ubicomp 2001: Ubiquitous Computing, Third International Conference*, pages 256–272. Springer, 2001.

[27] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375 – 408, 2002. Siehe auch Version von 1999. Scheinen inhaltlich gleich.

[28] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[29] J. Elson and K. Römer. Wireless sensor networks: a new regime for time synchronization. *ACM SIGCOMM Computer Communication Review*, 33(1):149–154, January 2003.

[30] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 117–129. ACM Press, 2000.

[31] N. Furmento, J. Hau, W. L. Lee, S. Newhouse, and J. Darlington. Implementations of a service-oriented architecture on top of Jini, JXTA and OGSI. In *Lecture Notes in Computer Science*, volume 3165/2004, pages 90–99. Springer Berlin/Heidelberg, 2004.

[32] E. Gamma, R. Helm, and R. E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1995.

[33] M. Golm and J. Kleinoeder. Ubiquitous computing and the need for a new operating system architecture. http://www4.informatik.uni-erlangen.de/Projects/JX/Papers/ubitools01.pdf, 2001.

[34] L. Gong. A software architecture for open service gateways. *IEEE Internet Computing*, pages 64–70, January/February 2001.

[35] Y. Gsottberger, X. Shi, G. Stromberg, W. Weber, T.F. Sturm, H. Linde, E. Naroska, and P. Schramm. Sindrion: A prototype system for low-power wireless control networks. In *Proceedings of the 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, 2004.

[36] T. Gu, H. Keng Pung, and D. Q. Zhang. Toward an OSGi-Based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3(4):66–74, October - December 2004.

[37] W. Guiling, C. Guohong, and T. La Porta. A bidding protocol for deploying mobile sensors. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, 2003.

[38] R. Gupta and S. Talwarand D. P. Agrawal. Jini home networking: A step toward pervasive computing. *IEEE Computer*, 2:34–40, August 2002.

[39] S. Hartwig, C. Heite, S. Jurthe, J. Kemper, P. Resch, P. Schramm, and B. Wilms. EI-

Blue - a self-contained system for EIB device control using Bluetooth. In *Proceedings of the International Symposium on Intelligent Environments*, 2006.

[40] S. Helal, B. Winkler, C. Lee, Y. Kaddourah, L. Ran, C. Giraldo, and W. Mann. Enabling location-aware pervasive computing applications for the elderly. In *Proceedings of the 1st IEEE Pervasive Computing Conference, IEEE CS*, pages 531–538, 2003.

[41] H. Higaki and M. Takizawa. Recovery protocol for mobile checkpointing. In *DEXA Workshop*, pages 520–525, 1998.

[42] J. M. Hélary, A. Mostefaoui, R. H. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 183–190, October 1997.

[43] T. D. Hodes and R. H. Katz. Composable ad hoc location-based services for heterogeneous mobile clients. *ACM Wireless Networks Journal, special issue on mobile computing: selected papers from MobiCom '97*, 5(5):411–427, October 1999.

[44] B. Horowitz, N. Magnusson, and N. Klack. Telia's service delivery solution for the home. *IEEE Communications Magazine*, pages 120–125, April 2002.

[45] P. Huang, V. Lenders, P. Minnig, and M. Widmer. Jini for ubiquitous devices. Technical Report ETH TIK-Nr. 137, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology - Zürich, June 2002.

[46] Y. Huang and Y. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 459–, 1995.

[47] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1998.

[48] F. Jammes and H. Smit. Service oriented architectures for devices - the SIRENA view. In *Proceedings of 2005 3rd IEEE International Conference on Industrial Informatics (INDIN 2005)*, 2005.

[49] M. Jeronimo and J. Weast. *UPnP Design by Example - A Software Designer's Guide to Universal Plug and Play.* Intel Press, 2003.

[50] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462 – 491, September 1990.

[51] D.-S. Kim, J.-M. Lee, W.-H. Kwon, and I.-K. Yuh. Design and implementation of home network systems using UPnP middleware for networked appliances. In *IEEE Transactions on Consumer Electronics*, volume 48, pages 963 – 972, November 2002.

[52] T. Kindberg and J. Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443–456, 2001.

[53] M. Krogmann. Design and implementation of an embedded UPnP stack using code generation techniques. Master's thesis, University of Dortmund, 2006.

[54] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(9):558 – 565, 1978.

[55] V. Lenders, P. Huang, and M. Muheim. Hybrid Jini for limited devices. In *Proceeding of the IEEE International Conference on Wireless LANs and Home Networks*, December 2001.

[56] W. Liu, Z. Chen, S. Tu, and W. Du. Adaptable QOS management in OSGi-Based cooperative gateway middleware. In *Second International Workshop Grid and Cooperative Computing, GCC*, 2003.

[57] C. W. Loftus, A. Olsen, E. Inocencio, and P. Viana. A code generation strategy for CORBA-based internet applications. In *Proceedings of the First InternationalEnterprise Distributed Object Computing Workshop [1997]. EDOC '97.*, pages 160–169, October 1997.

[58] C. Lu and S.-M. Lau. An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes. In *International Conference on Distributed Computing Systems*, pages 629–636, 1996.

[59] D. Maclay. Click and code [automatic code generation]. *IEEE Review*, 46(3):25–28, 2000.

[60] M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. Invisible network: Concepts and architecture. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002.

[61] P. Maniatis, M. Roussopoulos, E. Swierk, K. Lai, G. Appenzeller, X. Zhao, and M. Baker. The mobile people architecture. *ACM Mobile Computing and Communications Review*, 3(3):36–42, July 1999.

[62] D. Marples and P. Kriens. The Open Services Gateway Initiative: An introductory overview. *IEEE Communications Magazine*, pages 110–114, December 2001.

[63] Microsoft Corporation, Inc. Web services dynamic discovery (ws-discovery). http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Discovery.pdf, April 2005.

[64] B. A. Miller, T. Nixon, C. Tai, and M. D. Wood. Home networking with Universal Plug and Play. *IEEE Communications Magazine*, 39(12):104–109, December 2001.

[65] B. A. Myers. Mobile devices for control. In *Proceedings of the Fourth International Symposium on Human Computer Interaction with Mobile Devices, Mobile HCI 2002*, 2002.

[66] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: A P2P networking infrastructure based on RDF. In *WWW2002 Conference Proceedings*, 2002.

[67] E. Pacitti, M. T. Özsu, and C. Coulon. Preventive multi-master replication in a clus-

ter of autonomous databases. *Lecture Notes in Computer Science*, 2790/2004:318–327, 2003.

[68] M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[69] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Ubicomp 2001: Ubiquitous Computing, Third International Conference*, pages 56–75. Springer, 2001.

[70] A. Ranganathan, R. H. Campbell, M. Endler, and Schmidt D. A middleware for context-aware agents in ubiquitous computing environments. In *ACM/IFIP/USENIX international middleware conference*. Springer-Verlag, 2003.

[71] P. Resch, P. Schramm, and F. Uptmoor. Einführung in Bluetooth, die Java 2 Micro Edition und die Java APIs für Bluetooth. Technical report, Robotics Research Institute, University Dortmund, 2006.

[72] P. Rigole, Y. Berbers, and T. Holvoet. A UPnP software gateway towards EIB home automation. In *Proceedings of the IASTED International Conference on Computer Science and Technology*, pages 253–258, 2003.

[73] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 4:74–83, 2002.

[74] M. Roman, H. Ho, and R. H. Campbell. Application mobility in active spaces. In *Proceedings of the 1st International Conference on Mobile and Ubiquitous Multimedia*, 2002.

[75] S. J. Ross, J. L. Hill, M. Y. Chen, A. D. Joseph, D. Culler, and E. Brewer. A composeable framework for secure multi-modal access to internet services from post-pc devices. *Mobile Networks and Applications*, 7(5):389–406, October 2002.

[76] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[77] P. Schramm and E. Naroska. Sindrion Application Manager architecture. Technical Report IFX-SI-036-appmanspec-20031210, Computer Engineering Institute, University of Dortmund, 2003.

[78] P. Schramm, E. Naroska, P. Resch, J. Platte, H. Linde, G. Stromberg, and T. Sturm. A service gateway for networked sensor systems. *IEEE Pervasive Computing*, 3(1):66–74, 2004.

[79] R. Schwartz and S. Kraus. Bidding mechanisms for data allocation in multi-agent environments. In *Agent Theories, Architectures, and Languages*, pages 61–75, 1997.

[80] F. Sivrikaya and B. Yener. Time synchronization in sensor networks: a survey. *IEEE Network*, 18(4):45–50, 2004.

[81] L. Smith, C. Roe, and K. S. Knudsen. A Jini lookup service for resource-constrained

devices. In *4th IEEE International Workshop on Networked Appliances*, January 2002.

[82] H. Song, D. Kim, K. Lee, and J. Sung. UPnP-based sensor network management architecture. In *Proceedings of the Second International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2005)*, 2005.

[83] P. Steggles, P. Webster, and A. Harter. The implementation of a distributed framework to support 'follow me' applications. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Technique and Applications (PDPTA'98)*, volume 3, pages 1381–1388, Las Vegas, NV, July 1998.

[84] M. Strasser, J. Baumann, and F. Hohl. Mole - a java based mobile agent system. *Special Issues in Object Oriented Programming*, pages 301–308, 1997.

[85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204 – 226, August 1985.

[86] G. Stromberg, T. F. Sturm, D. Barisic, M. Krogmann, and S. Reiter. Sindrion Control Protocol v3.0 specification. Technical report, Infineon Technologies AG, 2006.

[87] T. F. Sturm. Sindrion system architecture. Technical Report IFX-SI-017-softarch-20031031, Infineon Technologies, Corporate Research, Emerging Technologies, 2003.

[88] Sun Microsystems Inc. Java object serialization specification revision 1.5.0. http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf, July 2003.

[89] Sun Microsystems Inc. Jini technology surrogate architecture specification version 1.0. http://surrogate.jini.org/sa.pdf, October 2003.

[90] Sun Microsystems Inc. Jini discovery and join specification. http://www.jini.org/wiki/Jini_Discovery_and_Join_Specification, 2006.

[91] Sun Microsystems Inc. JXTA v2.0 Protocols Specification. http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html, January 2007.

[92] S. Supakkul and L. Chung. Virtual OSGi framework and telecommunications, 2001.

[93] K. Takashio and G. Soeda. A mobile agent framework for follow-me applications in ubiquitous computing environment. In *21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01)*, Mesa, Arizona, April 2001.

[94] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless microsensor network models. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(2):28–36, April 2002.

[95] B. Toursel, R. Olejnik, and A. Bouchi. An object observation for a java adaptive distributed application platform. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC02)*, pages 171–176, 2002.

[96] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel

systems, 2005.

[97] J. Trevor, D. M. Hilbert, and B. N. Schilit. Issues in personalizing shared ubiquitous devices. In *Ubicomp 2002: Ubiquitous Computing, Fourth International Conference*, pages 56–72. Springer, 2002.

[98] UPnP Forum. UPnP Device Architecture, version 1.0. http://www.upnp.org/standardizeddcps, June 2000.

[99] UPnP Forum. UPnP AV architecture: 0.83 for UPnP, version 1.0. http://www.upnp.org/standardizeddcps/mediaserver.asp, June 2002.

[100] UPnP Forum. BinaryLight Device Template, version 1.01. http://www.upnp.org/standardizeddcps, November 2003.

[101] D. Valtchev and I. Frankov. Service gateway architecture for a smart home. *IEEE Communications Magazine*, pages 126–132, April 2002.

[102] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[103] Y. M. Wang. Consistent global checkpoints that contain a set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.

[104] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.

[105] M. Weiser. The computer for the 21st century. *Scientific American*, pages 94–104, September 1991.

[106] K. Werthschulte and F. Schneider. Linking devices to the European Installation Bus. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 2001.

[107] S. Zhou and D. Ferrari. An experimental study of load balancing performance. Technical Report CSD-87-336, University of California at Berkeley, 1987.