Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Patrick Wienecke

Konzeption und Realisierung einer “Common Java API“ für eine transparente hybride Multicastkommunikation

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Patrick Wienecke

Konzeption und Realisierung einer “Common Java API“ für eine transparente hybride Multicastkommunikation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Thomas C. Schmidt
Zweitgutachter : Prof. Dr.-Ing. Franz Korf

Abgegeben am 4. Mai 2011

Patrick Wienecke

Thema der Bachelorarbeit

Konzeption und Realisierung einer “Common Java API“ für eine transparente hybride Multicastkommunikation

Stichworte

IP-Multicast, Overlay-Multicast, hybrider Multicast, Hamcast, H∀Mcast , Multicast API, Java

Kurzzusammenfassung

Diese Bachelorarbeit befasst sich mit der Konzeption und Realisierung einer Common Java-API für eine transparente hybride Multicastkommunikation. Die Konzeption orientiert sich dabei am Internet Draft “A Common API for Transparent Hybrid Multicast“ von Matthias Wählisch und bindet die allgemeine API in der Realisierungsphase an den hybriden Multicastdienst H∀Mcast . Dieser wird neben Multicastgrundlagen in einem separaten Kapitel vorgestellt und analysiert. Eine abschließende Evaluierung untersucht Performancewerte der Java-API und vergleicht diese mit einer bereits implementierten C++ API.

Patrick Wienecke

Title of the paper

Design and implementation of a "Common Java API" for a transparent, hybrid multicast communication.

Keywords

IP-Multicast, Overlay-Multicast, hybrider Multicast, Hamcast, H∀Mcast , Multicast API, Java

Abstract

This bachelor thesis deals with the design and implementation of a “Common Java API“ for a transparent, hybrid multicast communication. The design is guided by the Internet Draft “A Common API for Transparent Hybrid Multicast“by Matthias Wählisch. In the implementation phase it connects the common API to the hybrid multicast service Hamcast. Hamcast and multicast fundamentals are presented and analysed in a separate chapter. A final evaluation examines the performance of the Java-API and compares it with an already implemented C++ API.

Danksagung

Sebastian Meilling für die Betreuung während der gesamten Entwicklungzeit.

Dominik Charrousset für die nütlzichen Programmiertipps zur Verbesserung der Performance.

Dirk Schulz fürs kurzfristige korrekturlesen.

Gunnar Weitschat und Stefan Hübner (Firma Base2It Consult) für die Beurlaubung während der Programmier- und Schreibphase der Bachelorarbeit.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einführung

1.1. Motivation

Das Internet hat im Laufe der letzten 3 Jahrzehnte stark an Präsenz in unserem täglichen Leben gewonnen. Diente es früher noch überwiegend der Unicastkommunikation, gewinnt es heute im Bereich der Gruppenkommunikation, auch bekannt unter dem Namen Multicast, immer mehr an Bedeutung. Die Berkley-Socket-API genoss in dieser vergangenen Zeit eine weite Akzeptanz und sorgte somit für eine homogene Struktur auf dem Layer 4, der sogenannten Transportschicht im Osi-Schichtenmodell (ver. J.D.Day und Zimmermann (1983)) . Das IPv4 Protokoll stellte in dieser Zeit eine ähnliche homogene Struktur auf der Vermittlungsschicht, dem Layer 3 des OSI Models dar. Dieses einheitliche, weit verbreitete Struktur der Vermittlungs- und Transportschicht ermöglichte Programmierern die Entwicklung von IP-Unicast Kommunikationssoftware, die quasi auf jeder Maschine und in jedem Netzwerk lauffähig war. Untersucht man das Kommunikationsverhalten der Internetteilnehmer, so hat sich insbesondere im letzten Jahrzehnt ein großer Wandel vollzogen. Sozial Netzwerke, Online Multiplayerspiele, Videokonferenzsoftware und IPTV erfreuen sich täglich wachsender Nutzerkreise, bei denen die Hauptkommunikation über verschiedenste Formen von Gruppenkommunikation realisiert wird.

1.2. Problemstellung

Multicast gibt es in einer Vielzahl an Varianten, deren technische Umsetzungen auf verschiedensten Ebenen des OSI Schichtenmodells bereits realisiert worden sind. Diese Vielzahl an Gruppenkommunikationsvarianten könnte Entwickler dazu zwingen, in ihren Netzwerkprogrammen einen adaptiven Service zu realisieren, der das Programm zur Laufzeit an einen verfügbaren Multicastdienst bindet. Zumeist wird allerdings aufgrund fehlender Unterstützung der ISP's[1](ver. Diot u. a. (2000)) und fehlender Standardisierung der meisten Multicastverfahren auf diese Art der dynamischen Wahl eines Dienstes verzichtet. Für eine gesicherte Gruppenkommunikation werden meist eigene Meshinfrastrukturen aufgebaut

[1] Internet Service Provider

oder der Einsatz von Replikationsservern für die erfolgreiche Gruppenkommunikation genutzt. Eine Kommunikationsanwendung zu schreiben, die überall lauffähig ist und den Anforderungen der Verfügbarkeit, Robustheit und Transparenz ähnlich der Berkley-Socket-API für IP-Unicast gerecht wird, zwingt Programmierer eigene Overlay[2] Multicastkommunikationsdienste zu entwickeln. Nachteil dieser Implementierung ist die nicht vorhandene Interoperabilität der verschiedensten Multicastverfahren und so kann es bei Multimedia- oder Kommunikationscomputern vorkommen, dass diese mehr als einen Multicastdienst installiert haben und diese Dienste auch alle zeitgleich genutzt werden. Dieses sorgt im heutigen Internet für eine enorme Netzwerklast und zwingt die ISP kontinuierliche ihre Netze und Rechenzentren auszubauen . Eine standardisierte Schnittstelle für Multicast würde neben der Transparenz, Verfügbarkeit und Robustheit auch für einen Abflachung der Entwicklungskurve des Traffic sorgen, denn dass die Netzwerklast in den nächsten Jahren noch steigen wird, daran besteht kein Zweifel. „Das Internet, Server und Rechenzentren gehören zu den besonders stark wachsenden Stromverbrauchern innerhalb des Informationstechnologie-Sektors. Eine bessere Energieeffizienz dieser Schaltstellen ist ein entscheidender Beitrag zum Klimaschutz, der auch noch Geld spart “ so Sigmar Gabriel[3]. Zwischen 2000 und 2006 hat sich der Energiebedarf von Rechenzentren verdoppelt und lag 2006 bei 8,7 Milliarden . Aufgrund des kontinuierlich wachsenden Strompreises hat sich der Preis in dieser Zeit mit 867 Millionen Euro mehr als verdreifacht. Dieser Verbrauch entspricht etwa drei Kohlekraftwerken. (ver. BMU und Bitkom (2008)) Ein erster Schritt für eine effizientere Kommunikation im Internet ist die Spezifizierung einer Common API[4] für Multicast.

1.3. Lösungsansatz

Matthias Wählisch hat die Probleme von Multicast im Draft “A Common API for Transparent Hybrid Multicast“ (ver. M.Wählisch u. a. (2011)) aufgegriffen und eine API für transparentes hybrides Multicast konzeptioniert und diskutiert. Hybride Multicastdienste sorgen für technologieunabhängige Gruppenkommunikation, die Programmierschnittstelle für Kompatibilität auf Anwendungsebene. Ziel der API ist eine Entkopplung der Gruppenanwendungssoftware von dem genutzten Multicastdienst und seiner verwendeten Technologie. Primitiven wie Join und Leave sollen technologieunabhängig Gruppenidentifikation und Adressierung von Mitgliedern ermöglichen. Die konkrete Realisierung der Multicastkommunikation geschieht im hybriden Multicastdienst, der nach Wählisch's Vorschlag in der Middleware platziert wird. Die Common API soll jedem Programmierer auf einfachste Weise eine Implementierung von Gruppenkommunikation ermöglichen. Multicasttransportvarianten, Kommunikation von verschiedenen Netzen, Routing und Adressierung werden vom Programmierer gekapselt und

[2] Multicast oberhalb der Transportschicht
[3] http://www.bmu.de/files/pdfs/allgemein/application/pdf/broschuere_rechenzentren_bf.pdf
[4] Programmierschnittstelle

durch den hybriden Multicastdienst hinsichtlich des zur Verfügung stehenden Verfahrens effizient bereitgestellt. Die Primitiven des Multicastdienstes werden über die Common API angesprochen und bedient.

1.4. Aufbau der Arbeit

Aufgabe dieser Arbeit ist die Implementierung einer Programmierschnittstelle kz. API nach Vorgaben des Draft “A Common API for Transparent Hybrid Multicast“ in der objektorientierten Sprache Java. Neben der konzeptionellen Entwicklung Kapitel 3 wird die Programmierschnittstelle in Kapitel 4 an einen bereits realisierten hybriden Multicastdienst H∀Mcast gebunden, der in Kapitel 2 neben Multicast Grundlagen vorgestellt wird. Bei der konzeptionellen Entwicklung werden alle Primitiven des Draft untersucht und die Realisierung unter Java diskutiert. In Kapitel 5 werden einige Test durchgeführt und diese mit den Ergebnissen einer bereits realisierten Programmierschnittelle unter C++ verglichen.

2. Multicast

2.1. IP-Multicast

IP-Multicast (ver. Deering (1989)), bekannt auch unter dem Namen nativer Multicast, ist eine Gruppenkommunikationslösung, die fest im IP Stack integriert ist. Bei genauerer Betrachtung finden wir die Implementation des nativen Multicast auf der Vermittlungsebene (Layer3) des OSI-Schichten-Modells wieder. IP-Multicast ist ein verbindungsloser Übertragungsdienst und nutzt auf dem Layer 4, der Transportschicht, das UDP Protokoll(ver. Postel (1980)). TCP, ein verbindungsorientiertes Protokoll, welches auf Layer 4 durch Ack's den erfolgreichen Empfang eines Paketes bestätigt, ist für Gruppenkommunikation aus zweierlei Sicht ungeeignet. Bei steigender Empfängerzahl käme es bei dem Absender zu einer Ack-Implosion (ver. Wittmann und Zitterbart (2001)), eine Verzögerung des Paketversands aufgrund der Abarbeitung der Acksflut wäre die Folge. Eine bewusste Paketwiederholung würde den Datenstrom mindern und bei Anwendungen mit Echtzeitverhalten, IPTV oder Radiostream zu unerwünschten großen Zeitverzögerungen führen. Einige Anwendungen, die UDP nutzen, lösen das Problem des Paketverlustes durch Paritätsmechanismen im Kommunikationsprotokoll der Anwendung oberhalb von Layer 4.

Für Multicast IPv4 Gruppen sind die 32 Bit Adressen von 224.0.0.0 bis 239.255.255.255 reserviert. Dieser Adressraum wird als Class D Netz bezeichnet und hat im Gegensatz zu den Class A, B und C Adressen keine Aufteilung in Host- und Net ID. Die ersten vier Bits, das sogenannte Multicastadress Präfix (1110), sind für die Identifizierung einer Multicastadresse, die restlich 28 bieten 2^{28}= 268.435.456 gültige Adressen. (ver. Albanna u. a.). Der IPv4-Multicast Adressraum ist wiederum unterteilt in permanente und dynamische Adressgruppen. Eine kleine Auswahl permanente Adressen sind in Tabelle 2.1 aufgelistet.

Multicastkommunikation erfolgt empfängerbasierend. Damit ein Router in einem IPv4 Netzwerk IP-Multicast Nachrichten empfangen und zustellen kann, müssen die Hostrechner des Netzwerkes den Router über deren Bedarf an IP-Multicastnachrichten informieren. Das Internet Group Managment Protocol (IGMP), welches in seiner aktuellen Version im RFC 3376 (ver. B.Cain (2002)) definiert ist, stellt hierfür die nötigen Funktionen. Aufgabe des Hosts ist es, den Router über seine Gruppenmitgliedschaft über IGMP Nachrichten zu informieren, mit dem Ziel, IP-Multicast-Nachrichten zu empfangen. IGMP Multicastnachrichten werden

Adresse	Beschreibung
224.0.0.0	reserviert
224.0.0.1	Alle Nodes innerhalb des Subnetzwerk
224.0.0.2	Alle Router innerhalb des Subnetzwerk
224.0.0.4	Alle DVMRP Router in einem Subnetzwerk
224.0.0.9	Alle RIPv2 Router in einem Subnetzwerk
224.0.0.9	Alle NTP Empfänger

Tabelle 2.1.: Permanente IPv4 Multicast Adressen

auch als Reports bezeichnet. Das TTL Feld hat den Wert 1, damit der Host nur den Router in seinem Netzwerk über seine Gruppenmitgliedschaft informiert. Der Router selbst merkt sich nur die Gruppenmitgliedschaften von Netzwerken und nicht, welcher Host im Netzwerk Mitglied einer Gruppe ist. Ein Report über eine Gruppenmitgliedschaft ist zeitlich begrenzt (default:260 Sekunden) und muss deshalb in regelmäßigen Abständen wiederholt werden.

Auch in IPv6 (ver.Deering und Hinden (1998)) ist IP Multicast vorgesehen. IPv6 Multicastadressen kennzeichnen sich durch 8 führende Einsen. Anschließend sind 4 Bits für Flags und weitere vier Bits für den Gültigkeitsbereich.(ver. Hinden und Deering (2006)). Die verbleibenden 112 Bits bilden die Group-ID und ermöglichen somit 2^{112} gültige Multicastadressen. Auch IPv6 definiert permanente Multicastadressen, diese werden allerdings durch das Setzen des T-Bit[1] gekennzeichnet und gehen somit dem Adressbereich von 2^{112} nicht verloren. Gruppenmanagement wird unter IPv6 über das Multicast-Listener-Discovery (MLD) geregelt, welches in der aktuellen Version Nr.2 in RFC 3810 (ver. Vida und Costa (2004) definiert ist. MLD ist ein Subprotokoll von ICMPv6 (ver.Conta und Deering (2006)) und wird dadurch wie alle ICMPv6-Nachrichten durch einen Next-Header-Wert von 58 identifiziert.

Um Pakete einer Multicastgruppe zu empfangen, stellt die Socketnetwork API Multicast-Group-Membership Socketoptionen bereit (siehe Tabelle 2.2 (Stevens u. a. (2009))), die den Empfang und das Versenden von Multicastpaketen über den IP-Stack ermöglichen. Hierbei unterscheidet man zwischen API Calls unter IPv4 und IPv6. Mit der Einführung der unabhängigen IP-Group-Membership-Socketoptionen (Abschnitt 3 der Tabelle 2.2), ist die transparente Entwicklung zur Vermittlungsschicht deutlich erleichtert worden. Die aufgeführten Codebeispiele 2.1, 2.2, 2.3 zeigen für einen Gruppenabonnement unter IP-Multicast die benötigten API Calls. Unter IPv4 wird implizit beim Abonnement einer Gruppe ein IGMP Report ausgelöst Zeile 55, bei IPV6 ein MLD Report Zeile 79. Zusammengefasst ergeben die Codebeispiele eine transparente mcast_join Funktion. Diese prüft vor Nutzung des IPv4 Befehls IP_ADD_MEMBERSHIP oder des IPv6 Befehls IPV6_JOIN_GROUP ob auf diesem System die neueren transparenten Multicast-Group-Membership-Socketoptionen definiert sind und nutzt im positiven Falle den transparenten Befehl MCAST_JOIN_GROUP Zeile 26. Zwar lö-

[1] Transient/vorübergehend

sen die transparenten Socketoptionen das Festlegen auf eine IP-Version im Quellcode, die fehlende DNS Unterstützung von IP-Multicast zwingt die Anwendung allerdings trotz transparenter Entwicklung vor Kommunikationsbeginn, sich auf eine IP Version einzuschränken. Die unterschiedlichen Multicastverfahren schränken die Entwickler zur Programmierzeit, durch die Wahl eines Verfahrens in der Interoperabilität zu anderen Programmen ein: ASM[2], jeder kann Pakete an die Gruppe schicken oder SSM[3], nur Pakete vom Sender S herstammend sind zulässig und deren verschiedenen API-Calls (siehe Tabelle 2.2 *SSM Befehle sind kursiv dargestellt*),. Diese Probleme und die meist fehlende native Unterstützung von ISP haben in den letzten Jahren Multicast in seiner Verbreitung stark gebremst und Entwickler gezwungen eigene Multicastprotokolle auf Basis eines Unicast-Underlays zu entwickeln.

Command	Datatype	Description
IP_ADD_MEMBERSHIP	struct ip_mreq	Join a multicast group
IP_DROP_MEMBERSHIP	struct ip_mreq	Leave a multicast group
IP_BLOCK_SOURCE	*struct ip_mreq_source*	*Block a source from a joined group*
IP_UNBLOCK_SOURCE	*struct ip_mreq_source*	*Unblock a previously blocked source*
IPV6_JOIN_GROUP	struct group_mreq	Join a multicast group
IPv6_LEAVE_GROUP	struct group_mreq	Leave a multicast group
MCAST_JOIN_GROUP	struct group_req	Join a multicast group
MCAST_LEAVE_GROUP	struct group_req	Leave a multicast group
MCAST_BLOCK_SOURCE	struct group_source_req	Block a source from a joined group
MCAST_UNBLOCK_SOURCE	struct group_source_req	Unblock a previously blocked source
MCAST_JOIN_SOURCE_GROUP	*struct group_source_req*	*Join a source-specific group*
MCAST_LEAVE_SOURCE_GROUP	*struct group_source_req*	*Leave a source-specific group*

Tabelle 2.2.: Multicast group membership socket options (Stevens u. a. (2009))

Listing 2.1: Join a multicast group:IP version-independent (Stevens u. a. (2009))

```
1
2  /* include mcast_join1 */
3  #include        "unp.h"
4  #include        <net/if.h>
5
6  int
7  mcast_join(int sockfd, const SA *grp, socklen_t grplen,
8                  const char *ifname, u_int ifindex)
9  {
10 #ifdef MCAST_JOIN_GROUP
11         struct group_req req;
12         if (ifindex > 0) {
13                 req.gr_interface = ifindex;
14         } else if (ifname != NULL) {
15                 if ( (req.gr_interface = if_nametoindex(ifname)) == 0) {
16                         errno = ENXIO;  /* i/f name not found */
17                         return(-1);
```

[2]ASM Any Source Multicast
[3]Source Specific Multicast

```
18                  }
19          } else
20                  req.gr_interface = 0;
21          if (grplen > sizeof(req.gr_group)) {
22                  errno = EINVAL;
23                  return -1;
24          }
25          memcpy(&req.gr_group, grp, grplen);
26          return (setsockopt(sockfd, family_to_level(grp->sa_family),
27                          MCAST_JOIN_GROUP, &req, sizeof(req)));
28  #else
```

Listing 2.2: Join a multicast group:IPv4 (Stevens u. a. (2009))

```
29          switch (grp->sa_family) {
30          case AF_INET: {
31                  struct ip_mreq          mreq;
32                  struct ifreq            ifreq;
33
34                  memcpy(&mreq.imr_multiaddr,
35                              &((const struct sockaddr_in *) grp)->sin_addr,
36                              sizeof(struct in_addr));
37
38                  if (ifindex > 0) {
39                          if (if_indextoname(ifindex, ifreq.ifr_name) == NULL) {
40                                  errno = ENXIO;  /* i/f index not found */
41                                  return(-1);
42                          }
43                          goto doioctl;
44                  } else if (ifname != NULL) {
45                          strncpy(ifreq.ifr_name, ifname, IFNAMSIZ);
46  doioctl:
47                          if (ioctl(sockfd, SIOCGIFADDR, &ifreq) < 0)
48                                  return(-1);
49                          memcpy(&mreq.imr_interface,
50                                     &((struct sockaddr_in *) &ifreq.ifr_addr)->sin_addr,
51                                     sizeof(struct in_addr));
52                  } else
53                          mreq.imr_interface.s_addr = htonl(INADDR_ANY);
54
55                  return(setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
56                                                  &mreq, sizeof(mreq)));
57          }
```

Listing 2.3: Join a multicast group:IPv6 (Stevens u. a. (2009))

```
58  #ifdef  IPV6
59  #ifndef IPV6_JOIN_GROUP         /* APIv0 compatibility */
60  #define IPV6_JOIN_GROUP         IPV6_ADD_MEMBERSHIP
61  #endif
62          case AF_INET6: {
63                  struct ipv6_mreq        mreq6;
64
65                  memcpy(&mreq6.ipv6mr_multiaddr,
66                              &((const struct sockaddr_in6 *) grp)->sin6_addr,
67                              sizeof(struct in6_addr));
68                  if (ifindex > 0) {
69                          mreq6.ipv6mr_interface = ifindex;
70                  } else if (ifname != NULL) {
```

```
71                          if ( (mreq6.ipv6mr_interface = if_nametoindex(ifname)) == 0) {
72                                  errno = ENXIO;  /* i/f name not found */
73                                  return(-1);
74                          }
75                  } else
76                          mreq6.ipv6mr_interface = 0;
77
78                  return(setsockopt(sockfd, IPPROTO_IPV6, IPV6_JOIN_GROUP,
79                                                  &mreq6, sizeof(mreq6)));
80          }
81  #endif
82          default:
83                  errno = EAFNOSUPPORT;
84                  return(-1);
85          }
86  #endif
87  }
```

2.2. Application-Layer Multicast

Beim Application-Layer-Multicast ALM werden die drei Aufgabenbereiche Gruppenkommunikation, Gruppenmanagement und Routing nicht wie beim IP-Multicast im IP-Stack des Host und der Router des Netzwerkes realisiert, sondern auf Anwendungsebene der Kommunikationsprogramme. Die am ALM beteiligten Knoten bauen gemeinsam ein virtuelles Netzwerk (Overlay) auf, abstrahiert vom darunterliegenden nativen Netzwerk(Underlay) (ver. Steinmetz und Wehrle (2005)). Auf dem Underlay wird per Unicast kommuniziert. Dies löst zwar einige Probleme von Multicast, wie z.B. die fehlende Unterstützung eines DNS Services und die meist nur geringfügige Unterstützung von IP-Multicast der ISP. Die Interoperabilität der Overlaymulticastverfahren ist allerdings durch die große Auswahl an unterschiedlichsten Verfahren und deren fehlende Standardisierung nicht verbessert. Die Fehlende einheitliche Definition in einem RFC und deren unterschiedlichen Architekturansätze, unstrukturierte oder strukturierte Peer-to-Peer Netzwerke fordern bei den API-Calls für Multicastkommunikation unterschiedlichste Abläufe, die eine allgemeine API quasi unmöglich machen. Weiterer Nachteile gegenüber dem IP-Multicast sind die Replikation und das Routing auf Anwendungsebene, was zu zeitlichen Verzögerungen und nicht optimalen Routen führen kann.

2.3. Hybrider Multicast

Hybrider Multicast kombiniert die Vorteile von nativem Multicast auf Netzwerkschicht, hinsichtlich der Netzlast und Overlaymulticast auf Anwendungsschicht, hinsichtlich der Verfügbarkeit bei der Bereitstellung eines Multicastdienstes. Die Grundidee sind IP-Multicastinseln, sogenannte walled gardens, mittels eines geeigneten Overlay-Verfahrens zu verbinden.

Denkbar wäre beim hybriden Multicast auch, Kombinationen aus Overlaymulticastinseln und IP-Multicastinseln zu kombinieren. Das Overlay hat im hybriden Multicast die Funktion eines routing Backbones. Grenzen der Inseln werden durch Administration und Technologie bestimmt. Für die korrekte Funktionalität benötigt ein hybrider Multicastdienst lediglich einen Unicast-Routing-Dienst und entkoppelt so Netzwerkinnovation von Netzwerkkompetenz der Anwendungsentwickler sowie vom Kooperationswillen einzelner ISPs und Netzbetreiber. IM [4](ver.Jin u. a. (2009)), SHM [5] (ver.Lu u. a. (2007) und UM[6] (ver.Zhang u. a. (2006)) sind alle drei Multicastverfahren, die einen hybriden Multicastansatz verfolgen. IM ist allerdings hinsichtlich fehlender Spezifikationen der domänenübergreifenden Gruppenverwaltung, SHM aufgrund von Skalierbarkeitsproblemen durch eine zentrale Verwaltung ungeeignet für eine zukünftige forcierte Verbreitung des Architekturansatzes. UM löst die Probleme von IM und SHM, wurde sogar als Prototyp entwickelt und evaluiert. UM spezifiziert aber keine Lösung für die Kommunikation von unterschiedlichen Overlaymulticastverfahren und geht ebenfalls nicht näher auf Kommunikation zwischen IPv4 und IPv6 ein.

2.4. Die H∀Mcast-System Architektur

H∀Mcast (ver.Meiling u. a. (2010),Schmidt und Wählisch (2010), Wählisch und Schmidt (2010)) ist ein weiterer hybrider Architekturansatz, bei dessen Entwicklung neben dem universellen Multicast-Dienst auch eine Anbindung an eine standardisierte, technologietransparente Anwendungsschnittstelle zur Gruppenkommunikation im Internet berücksichtigt worden ist. Im Gegensatz zu anderen hybriden Multicastansätzen verfolgt H∀Mcast einen generischen Ansatz. Jeder Teilnehmer wählt anhand der verfügbaren Multicastkommunikationsform unter H∀Mcast die entsprechende Kommunikationsdomäne. Domänen fassen unter H∀Mcast alle Teilnehmer zusammen, die über die gleiche Kommunikationstechnologie verfügen. Die Grenzen der Multicast Domänen können administrativ bei unterschiedlichen ISPs, oder auch technologisch sein. Abbildung 2.1 zeigt ein Beispielszenario mit drei Multicast-Domänen, zwei unterschiedlichen Multicasttechnologien (A und B) und mehreren Nutzern in den beiden Gruppen G bzw. F. In diesem Szenario könnten Technologie B z.B. nativer Multicast und Technologie A ein Scribe-Multicast-Overlay sein, um die beiden Domänen zu verbinden. Die einzelnen Domains sind über Gateways IMGs[7] miteinander verbunden und ermöglichen somit eine weitreichende Gruppenkommunikation. Damit H∀Mcast in seinen API-Calls von Technologie(z.B. IPv4/IPv6) und Protokoll(IP-Multicast/Overlay Multicast) unabhängig ist, wird eine allgemeine Multicast-API als Programmierschnittstelle genutzt. H∀Mcast realisiert die Idee, hybriden Multicast in vorhandene Endsysteme als Middlewaredienst zu inte-

[4] Island-Multicast
[5] Scalable Hybrid Multicast
[6] Universal Multicast
[7] Inter-domain Multicast Gateway

grieren und über die API vielen Anwendungen zeitgleich zur Verfügung zu stehen. Hauptaufgabe des Middlewaredienstes ist Service Discovery zur Erkennung und Konfiguration lokaler Ressourcen sowie die Möglichkeit, Interdomain-Multicast-Gatewaysfunktionalität zur Paketvermittlung über Domaingrenzen hinweg zu ermöglichen. Beobachtungen der heutigen Netzwerkstruktur haben sogar gezeigt, dass Endsysteme, Router oder auch Servicegateways bereits ausreichend Rechenleistung sowie Speicher für einen weiteren komplexeren Paketvermittlungsdienst zur freien Verfügung haben. H∀Mcast braucht zur weiteren Paketvermittlung einen reinen Unicastdienst, der im heutigen Internet überall zur Verfügung steht. Der Ansatz H∀Mcast als Middlewaredienst zu realisieren und über die API möglichst vielen Programmen zur Verfügung zu stehen sowie die niedrigen Ansprüche an vorhandener Infrastruktur machen H∀Mcast zu einem bis Dato nie dagewesenen Multicastkommunikationsdienst.

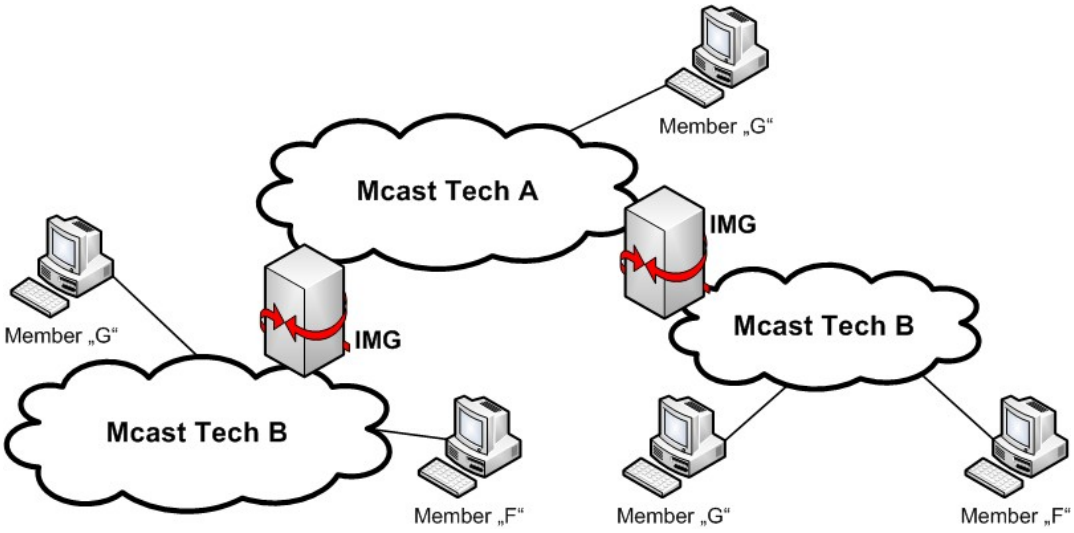

Abbildung 2.1.: Beispiel einer H∀Mcast Infrastruktur

2.4.1. Adressierung unter H∀Mcast

Ein Grund für das Deployment-Dilemma von IP-Multicast ist die bestehende Abhängigkeit der Namen und Adressen einer Gruppe zur Technologie. Overlaytechnologien lösen zwar diese Abhängigkeit, allerdings jedes Protokoll auf seine eigene Weise. Ein Name repräsentiert unter H∀Mcast innerhalb einer Technologie eindeutig eine Multicastgruppe und die Adresse ermöglicht eine Lokalisierung der Gruppenteilnehmer über die verwendete Technologie. Eine Gruppe ist eine virtuelle Entität, die eine Indirektion zwischen Sender und Empfänger darstellt. Unter IP-Multicast und Overlaymulticast gibt es keine Unterscheidung zwischen Gruppen-Namen(ID) und Adressen (Locator). Eine URI, so wie sie auch im Draft

für eine Multicast-Common-Api vorgeschlagen wird, ermöglicht hingegen eine eindeutige domänenübergreifende Identifizierung einer Gruppe. Eine Multicast Uri ist unter H∀Mcast wie folgt aufgebaut:

scheme ’://’ group ’@’ instantiation ’:’ port ’/’ sec-credentials

Das Schema spezifiziert den Namensraum (Ipv4,scripe...), in dem Gruppennamen global eindeutig sind. So darf zum Beispiel die Gruppe snoopy (sip://snoopy@peanuts.com:1234) im Namensraum sip nicht ein weiteres Mal definiert werden, allerdings im Namensraum scribe (scripe://snoopy@peanuts.com:1234) durchaus. Die Instantiation referenziert einen dedizierten Endhost oder ein Overlay, welche die angegebene Gruppe instantiiert. Diese Form ist analog zu SSM mit <S,G> und ermöglicht Pakete nur von einem dedizierten Endhost anzunehmen. Der Port adressiert einen gewünschte Anwendung und über die credentials kann der Zugriff auf eine Multicastgruppen kryptografisch eingeschränkt und kontrolliert werden. Teilnehmer- Einschränkungen sucht man unter IP-Multicast vergebens, was ein Grund für ISPs ist, die Unterstützung zu verweigern.

2.4.2. Interdomain Multicast Gateways

H∀Mcast ermöglicht durch verschiedene Technologiedomains eine effiziente Multicastkommunikation zu realisieren. Teilnehmer in den verschiedenen Domains können innerhalb der Domain hinsichtlich ihres Protokolls effizient kommunizieren. IMG ermöglichen dabei die Adressierung von Empfängern in anderen Domains. Ein IMG ist ein dedizierter Knoten, der Zugriff auf zwei oder mehr Multicastdomänen hat und in der Lage ist, Daten zwischen diesen zu vermitteln. Bei der Vermittlung muss der IMG allerdings berücksichtigen, dass eine URI zwar global eindeutig ist, allerdings innerhalb der Domäne über die Adresse der Multicasttechnologie angesprochen werden muss. Das zwingt einen IMG einen Mappingdienst zu implementieren, der zur Laufzeit eine Bindung einer Uri auf die Technologieadresse des Empfängers in der Domäne ermöglicht.

2.4.3. Middleware

H∀Mcast wird als Middlewaredienst auf den Hosts installiert, um eine gewisse Grundfunktionalität des hybriden Multicastdienstes bereitzustellen. Jedes Modul unter H∀Mcast realisiert implizit für eine erfolgreiche Multicastkommunikation neben Service-Discovery und Service-Selektion einen Name-Adress -Mapping-Dienst, um technologiespezifische Adressen auf Uri’s zu mappen. Um auf Anwendungsebene eine möglichst große Transparenz zu bieten,

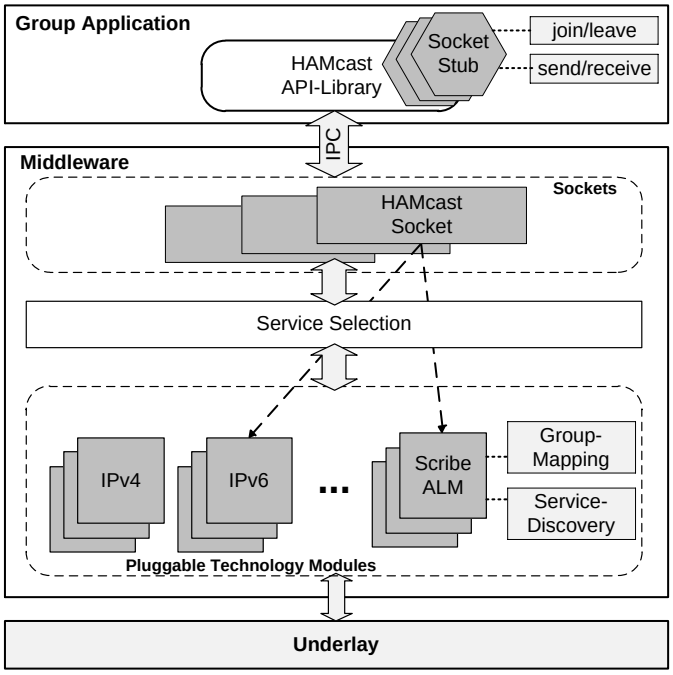

Abbildung 2.2.: Aufbau der H∀Mcast Architektur

läuft die Kommunikation mit der Middleware auf Anwendungsebene programmiersprachenunabhängig über ein eigens realisiertes IPC Protokoll ab. Dies ermöglicht neben der vorhandenen Programmierschnittstelle von H∀Mcast die unter C++ geschrieben ist, auch Netzwerkkommunikationsprogramme, die z.B. unter Java geschrieben sind, nachträglich an die Middleware zu binden. Eine entsprechende API, die sich an die Spezifikation des Draft hält und das IPC Protokoll der H∀Mcast Middleware in der Sprache Java implementiert, wird in Kapitel**??** konzeptioniert und realisiert. Die Interprozess-Kommunikation wird vom späteren Multicastentwickler gekapselt- um diesem möglichst einfache API-Calls zu ermöglichen.

2.4.4. IPC Kommunikationsmodul

Damit H∀Mcast möglichst unabhängig auf Anwendungsebene angesprochen werden kann, geschieht die Anbindung des Middlewaredienstes über IPC[8]. Das eigens entwickelte RMI[9] Verfahren(ver.Coulouris (2002),S.204-244) ist speziell auf den hybriden Multicastdienst H∀Mcast abgestimmt und bedient sich keiner bestehenden Implementierung, wie z.B. Corba oder Java-RMI. Die eigene, schlanke IDL Sprache[10] (siehe Listing2.4) und eine auf die

[8]Interprozesskommunikation
[9]Remote Methode Invocation
[10]Interface Definition Language

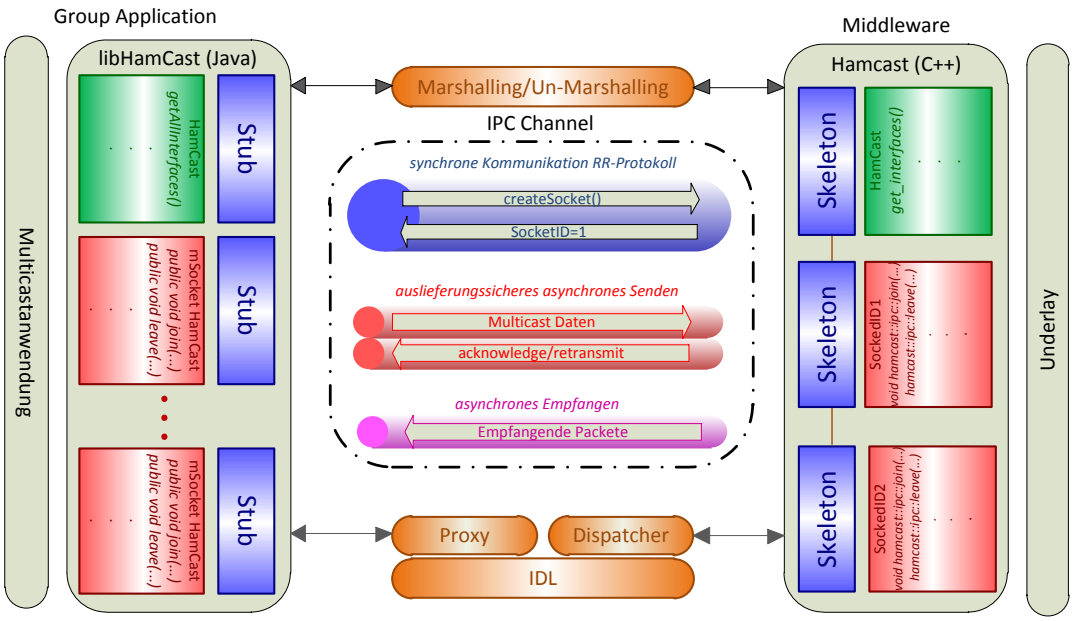

Abbildung 2.3.: IPC Kommunikation in H∀Mcast

Kommunikation optimierte Anforderung/Antwort Nachricht (siehe Abb. 2.4) ermöglichen im Gegensatz zu Java RMI oder Corba einen entfernten Methodenaufruf mit einem sehr geringen Overhead durch Informations- und Steuerungsdaten in der RR-Nachrichtenstruktur. Ein Proxy im IPC-Modul auf Anwendungsebene und ein Dispatcher im IPC-Modul von H∀Mcast in der Middleware sorgen für lokale Transparenz und ermöglichen dem Anwender einfache lokale Funktionsaufrufe, die in der Middleware ausgeführt werden (siehe Abb. 2.3).Durch Marshalling- und Un-Marshalling-Verfahren werden Probleme von Hardware- und Programmiersprachenabhängigkeiten gelöst. Die Middleware erwartet und verschickt die Daten in der Networkorder, dem sogenannten Big Endian Format. Das most signifikant Byte steht an der kleinsten Adresse, bzw. wird zuerst auf den Stream geschrieben.

2.4.5. Anforderung-/Antwort-Nachrichtenstruktur

Bei der Entwicklung des Prototypen H∀Mcast wurde als Kommunikationsprotokoll für den entfernten Methodenaufruf auf TCP gesetzt. Durch das verbindungsorientierte Protokoll TCP (ver. Tanenbaum (2003), Seite 580-604) ist sichergestellt, dass Anforderungs- und Antwortnachrichten sicher ausgestellt werden und extra Implementierungen zur wiederholten Übertragung, dem Filtern von Duplikaten und der Flusssteuerung, nicht extra realisiert werden müssen. Ein späterer Umstieg von TCP auf das verbindungslose Protokoll UDP (ver. Postel (1980)) ist allerdings einfach zu realisieren, weil der überwiegende Teil an Funktionsaufru-

Listing 2.4: C++ IDL der Middleware

```
enum message_type
{
        sync_request            = 0x00,
        sync_response           = 0x01,
        async_event             = 0x02,
        async_send              = 0x03,
        async_recv              = 0x04,
        cumulative_ack          = 0x05,
        retransmit              = 0x06,
};
enum function_id
{
        fid_create_socket       = 0x0001,
        fid_delete_socket       = 0x0002,
        fid_create_send_stream  = 0x0003,
        fid_join                = 0x0004,
        fid_leave               = 0x0005,
        fid_set_ttl             = 0x0006,
        fid_get_sock_interfaces = 0x0007,
        fid_add_sock_interface  = 0x0008,
        fid_del_sock_interface  = 0x0009,
        fid_set_sock_interfaces = 0x000A,
        // service calls
        fid_get_interfaces      = 0x0100,
        fid_group_set           = 0x0101,
        fid_neighbor_set        = 0x0102,
        fid_children_set        = 0x0103,
        fid_parent_set          = 0x0104,
        fid_designated_host     = 0x0105
};

enum exception_id
{
        eid_none                    = 0x0000,
        eid_requirement_failed      = 0x0001,
        eid_internal_interface_error= 0x0002
};
```

fen nach dem RR-Protokoll[11] ablaufen und somit ein leichtes Erkennen von Paketverlusten bereits implementiert ist. Senden und Empfangen laufen nicht nach dem Schema des RR-Protokoll ab und werden über ein asynchrones Verfahren realisiert. Das Sendeprotokoll allerdings garantiert durch Acknowlege- und Retransmitnachrichten eine garantierte Auslieferung in der Middleware. Ein Umstieg auf UDP ist auch hier einfach zu realisieren. Eine Acknowledge Nachricht hat allerdings keinen Informationsgehalt darüber, ob die Nachricht bei einem Empfänger angekommen ist, da Multicast UDP als Übertragungsprotokoll nutzt. Asynchrone empfangende Nachrichten benötigen zur Zeit noch TCP, da eine Auslieferung in der Anwendungssoftware bei UDP nicht garantiert werden kann.

Eine IPC Nachricht unter H∀Mcast (siehe. Abb. 2.4) besitzt einen 16 Byte großen Messa-

[11] Request-Replay

gehead, der Information und Kontrolldaten für die IPC Kommunikation enthält, gefolgt vom Messagebody, dem Multicastdatabereich, der bei synchronen Nachrichten Über- und Rückgabeparamter von entfernten Methodenaufrufen enthält oder bei asynchronen Nachrichten Sende- oder Empfangsdaten.

1. message_type: H∀Mcast unterscheidet zwischen fünf verschieden Nachrichtentypen (siehe IDL Code 2.4 Zeile 3-9).
2. field1: 2 Byte großes Feld
3. field2: 4 Byte großes Feld
4. field3: 4 Byte großes Feld
5. cs: größe des content
6. content: serialisierte Nutzdaten

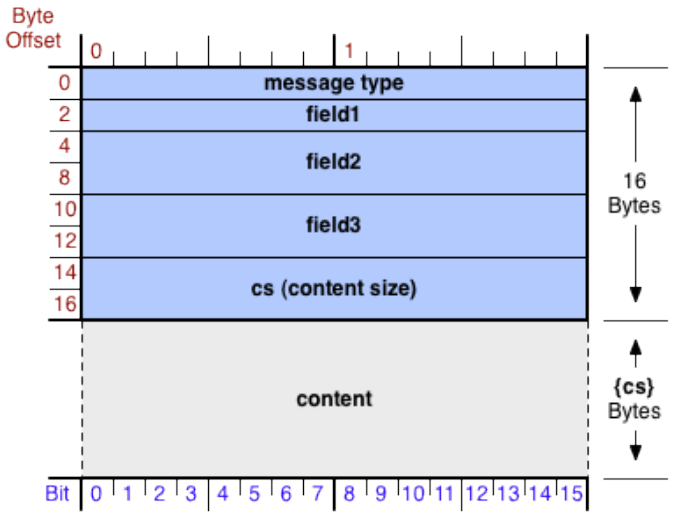

Abbildung 2.4.: IPC Nachricht unter H∀Mcast (ver.Charousset (2011))

2.4.5.1. Synchrone Anfrage und Antwort

Eine im Stub erstellte transparente synchrone IPC Anforderungsnachricht (siehe Abb.2.5), enthält im Feld message_type die Konstante SYNC_REQUEST(siehe Listing 2.4). Im Feld 1 eine FunktionID der IDL, die einen entfernten Methodenaufruf ermöglicht. Der Proxy im IPC-Modul auf Anwendungsebene ist für das Mapping von lokalen Methoden auf FunktionIDs zuständig. Der Dispatcher im IPC-Modul der Middleware wählt durch Reservemapping die passende Funktion der empfangenden FunktionsID aus. Feld 2 enthält eine eindeutige,

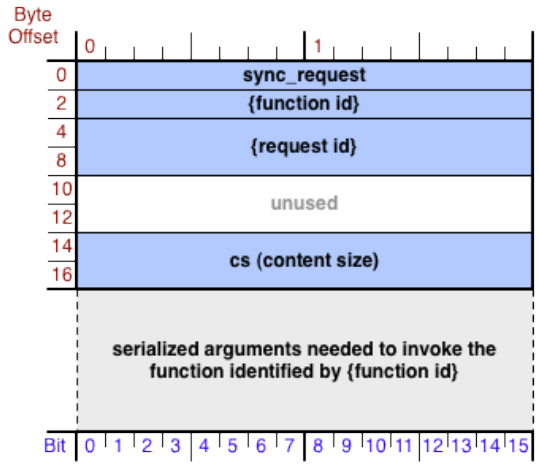

Abbildung 2.5.: Synchrone Anfrage (ver.Charousset (2011))

4 Byte große RequestID, die in der Middleware als unsigned int interpretiert wird und somit 4294967296 Requestnachrichten ermöglicht. Nach 2^{32} verschickten synchronen Nachrichten beginnt die RequestID wieder von 0. Feld 3 bleibt bei einer synchronen Anfrage ungenutzt. Das content size Feld enthält die Größe in Byte des Multicastdata Bereiches in dem Übergabeparameter der entfernten Funktion durch Marshalling, programmiersprachen- und systemplattformunabhängig verpackt werden. Eine synchrone Anfrage bekommt immer eine synchrone Antwort der Middleware (siehe Abb. 2.6), auch im Falle eines Fehlers. Die Empfangende IPC-Nachricht enthält im message_type Feld die Konstante SYNC_RESPONSE und informiert über Feld 1, ob die Anfrage mit der Request ID im Feld 2 erfolgreich ausgeführt werden konnte. Feld 3 ist wie schon bei der Anfrage ungenutzt und kann für spätere Erweiterungen genutzt werden. Das content size Feld beschreibt die Größe des Multicast-Databereiches. Bei einer erfolgreichen Anfrage enthält der Databereich die Rückgabewerte des entfernten Methodenaufrufes, im Fehlerfall eine Fehlercodebeschreibung. Die Rückgabewerte werden ebenfalls von der Middleware durch Marshalling Maschinen und programmiersprachenunabhängig dargestellt. Auf alle synchronen Anfragen gibt es von der Middleware auch eine Antwort. Diese Form hat Spector(1982) als RR Request-Replay Protokoll für RPC definiert. Ein Vorteil dieser Kommunikationsform ist der Verzicht auf Acks für die Paketbestätigung oberhalb von Layer 4, der quasi automatisch durch die Antwort passiert. Auch ein Austausch von TCP auf das deutlich schlankere verbindungslose Layer 4 Protokoll UDP wäre möglich, wird allerdings in der ersten Version des Prototypen von H∀Mcast nicht unterstützt.

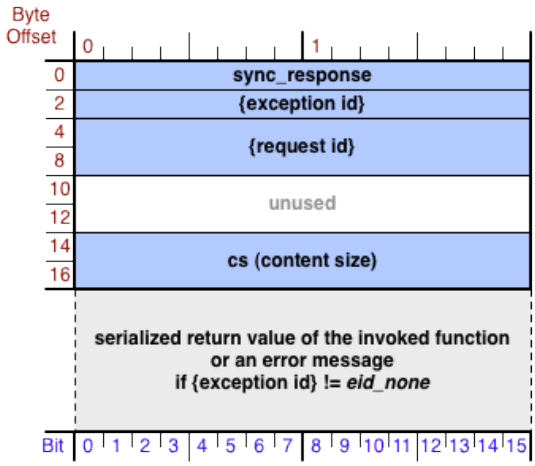

Abbildung 2.6.: Synchrone Antwort (ver.Charousset (2011))

2.4.5.2. Asynchrones Senden, Bestätigen und Wiederholen

Eine im SocketStub erstellte transparente asynchrone IPC Sendenachricht (siehe Abb. 2.7), enthält im Feld message_type die Konstante ASYNC_SEND(siehe Listing 2.4). Feld 1 enthält eine Stream ID, die für jede Socket/Uri Kombination eindeutig ist. Feld 2 enthält die Socketinformationen und Feld 3 eine fortlaufende Sequenznummer, die für jede StreamID mit 0 beginnt und bei 2^{32} wieder auf 0 gesetzt wird. Das Content-Size-Feld beschreibt die Größe des Databereiches, in dem die Daten des verschickten Paketes stehen. Diese werden 1 zu 1 weitergereicht. Um einen deutlich höheren Datendurchsatz zu erreichen, werden die Pakete nicht einzeln bestätigt, sondern in zeitlichen Abständen durch eine empfangende Acknowledgenachricht 2.8. Diese enthält im Feld message_type CUMULATIVE_ACK (siehe Listing 2.4). Feld 1 und Feld 2 enthalten zur Lokalisierung Stream und Socket ID. Die Sequenznummer in Feld 3 bestätigt alle erfolgreich versendeten Pakete aus der Middleware kleiner gleich der Sequenznummer. Diese Nachricht hilft, die lokale History in regelmäßigen Abständen zu löschen. Eine Acknowlege Nachricht besagt allerdings nur, dass eine Nachricht aus der Middleware erfolgreich über ein Multicastverfahren verschickt worden ist, da Multicast UDP ist, kann eine erfolgreiche Ankunft beim Empfänger nicht bestätigt werden. Die Middleware speichert empfangende Pakete zwischen, um diese für das gewählte Multicastverfahren vorzubereiten. In Stoßzeiten, wenn viele Anwendungen die Middleware nutzen, kann es passieren, dass der lokale Zwischenspeicher in der Middleware überläuft. H∀Mcast garantiert aber einen Versand in richtiger Reihenfolge der Sequenznummern und muss somit ein verloren gegangenes Paket neu anfordern. Dabei wird davon ausgegangen, dass wenn ein Paket verloren gegangen ist, alle weiteren Pakete der Sequenz, die verschickt worden sind, ungültig sind. In diesem Falle löst die Middleware eine Retransmit Nachricht an das IPC Modul der Anwendungs API aus. Die message_type enthält bei dieser Nachricht (sie-

he Abb. 2.9) die Konstante RETRANSMIT (siehe Listing 2.4). Feld 1 und Feld 2 enthalten zur Lokalisierung Stream und Socket ID. Die Sequenznummer in Feld 3 informiert über das fehlende Paket und stößt die Wiederholung aller Pakete beginnend ab der Sequenznummer an. Ein Beispielhafter Ablauf einer Kommunikation zwischen der Middleware von H∀Mcast und einer Anwendung ist in Abb.2.10 ersichtlich. Diese Form der asynchronen Kommunikation setzt voraus, dass alle Pakete, die noch nicht von der Middleware bestätigt worden sind, zwischengespeichert werden müssen um diese ggf. bei einer Retransmit-Nachricht zu wiederholen.

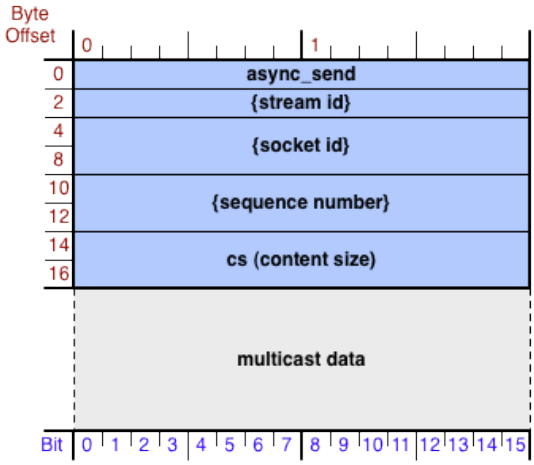

Abbildung 2.7.: Asynchrones senden (ver.Charousset (2011))

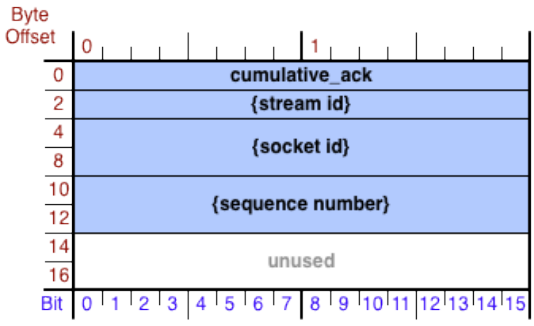

Abbildung 2.8.: Acknowledge Message (ver.Charousset (2011))

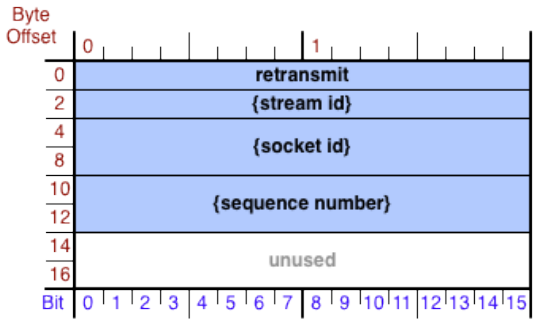

Abbildung 2.9.: Retransmit Message (ver.Charousset (2011))

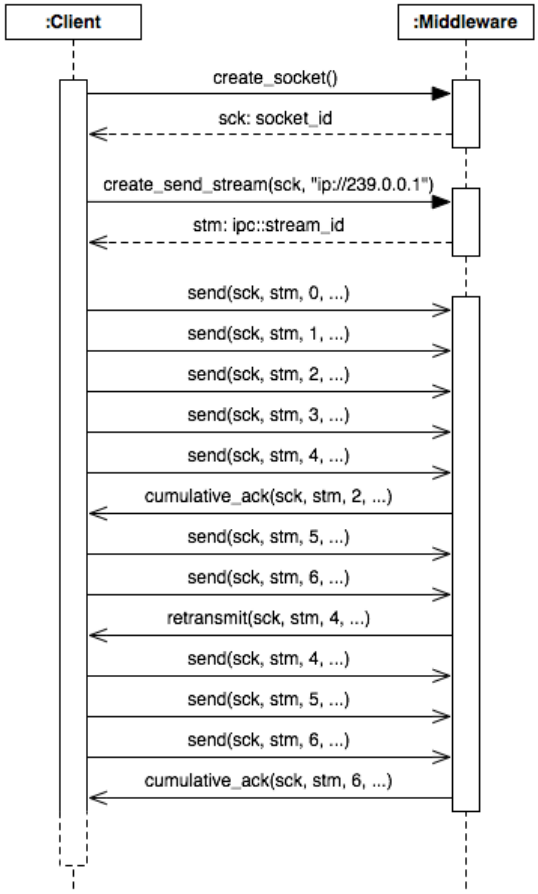

Abbildung 2.10.: IPC Kommunikation (ver.Charousset (2011))

3. Konzeption einer Java Multicast API

3.1. Motivation

Der hybride Multicastdienst H∀Mcast der im Kapitel 2.4 ausführlich vorgestellt worden ist, löst die Probleme von IP-Multicast hinsichtlich der Verfügbarkeit und reduziert die Probleme von reinem Overlay-Multicast hinsichtlich des Netwerklast. Bei der konzeptionellen Entwicklung von H∀Mcast wurde versucht einen Multicastdienst zu entwickeln, der zeitgleich mehreren Anwendungen auf einer Hardware zur Verfügung steht. Eine standardisierte Schnittstelle auf Anwendungsebene ist quasi unabdingbar, um möglichst vielen Softwareentwicklern den Zugang zu H∀Mcast zu ermöglichen. Im folgenden Kapitel wird eine Programmierschnittstelle kz. API unter Java konzeptioniert und realisiert, die sich an den Vorgaben des Draft "A Common Api for hybrid Multicast" (ver. M.Wählisch u. a. (2011)) orientiert und über ein IPC-Modul an den hybriden Multicastdienst H∀Mcast gebunden wird.

3.2. Die objektorientierte Sprache Java

Als Implementierungssprache des Draft wurde die objektorientierte Sprache Java gewählt. Die Plattformunabhängigkeit hat die Programmiersprache in nur kurzer Zeit zu einer mächtigen objektorientierten Sprache wachsen lassen und schenkt über 6 Millionen Softwareentwicklern (ver.Ullenboom (2011)) einen sicheren Arbeitsplatz. Im Gegensatz zu vielen anderen Programmiersprachen, die speziellen Maschinencode für Windows oder Linux und einem bestimmten Prozessor zum Beispiel für x86er Mikroprozessoren erstellen, erzeugt die Java Plattform unabhängig immer denselben Bytecode. Eine virtuelle Maschine, unter Java trägt sie den Namen JVM, interpretiert den compilierten Bytecode und führt diesen aus. Die virtuelle Maschine ist selber in C++ geschrieben und umfasst ca. 900 Zeilen C++ Code (ver. Ullenboom (2011), Seite 51), der Javacode Compiler hingegen basiert auf Java Sourcecode.

Das reine Interpretieren hat in den Anfängen durch das sequentielle Erkennen, Dekodieren und Ausführen der Befehle zu großen Performanceproblemen geführt. Erste Javaprogramme waren im Gegensatz zu C++ Programmen deutlich langsamer. Die Idee des Just-in-Time

Compiler, HP hatte 1970 schon ähnliche Lösungen für Basic Maschinen, brachte der Sprache die nötige Steigerung der Geschwindigkeitsperformance.

In den Anfängen der Sprachverbreitung von Java wurde unter anderem verstärkt in die Vermarktung von Hardwarechips investiert. Die Entwicklung ging von Sun selbst aus und einer der ersten Prozessoren war PicoJava. Nach längerer Flaute lässt allerdings der Samsung Prozessor S5L8900 hoffen. Dieser Prozessor bedient sich einer Armtechnologie namens Jazelle DBX und ermöglicht eine direkte Ausführung von Java Bytecode. Mit einer Geschwindigkeit von 667 MHZ und den zusätzlichen Multimediamöglichkeiten ist er einoptimaler Prozessor für mobile Einheiten. Auch Apple hat dieses für sich erkannt und den Prozessor in sein iPhone integriert. Ironie des Schicksals allerdings , dass Apple im Betriebssystem des iPhone bis Dato keine Unterstützung von Java vorsieht.

Java ist nicht bis zur letzten Konsequenz objektorientiert, so wie Smalltalk es vorbildlich demonstriert, sondern auch nach dem Vorbild der Programmiersprache C eine imperative Programmiersprache. Primitive Datentypen, wie int, float, long erlauben eine Verwaltung, ohne dass explizite Erzeugen eines Objektes, Performance-Verbesserung durch Compileroptimierung stand bei dem Eingriff des Designs mit der Einführung der primitiven Datentypen an oberster Stelle. Microsoft hingen bewies im Nachhinein mit seiner virtuellen Maschine für die .net Plattform, dass auch eine einhundertprozentig objektorientierte Sprache ohne Trennung dieser Datentypen nicht unter Performanceeinbrüchen leiden muss.

3.3. Java als Draftimplementierungssprache

Die Funktionsdeklarationen des Draft “A Common API for Transparent hybrid Multicast“ sind sprachneutral, berücksichtigen und beantworten keine spezifische Implementierungsfragen für Hochsprachen. Im folgenden Kapitel werden im Allgemeinen alle Pseudocode-Funktionsdeklarationen des Draft erläutern und im Speziellen die Implementierung unter Java diskutieren. Bei der Entwicklung des Java Quellcodes verfolgen wir den objektorientierten Ansatz dieser Sprache. Dieses Programmierparadigma soll dem späteren Java-Programmierer einen vertrauten Umgang mit der Java-API schaffen. Daten und Methoden werden sukzessiv nach außen gekapselt und schließen eine versehentliche Manipulation von außen aus.

Um möglichst keine implementierungsspezifischen Abhängigkeiten auf Anwendungsebene durch spezielle Parameter und Rückgabewerte zu generieren, werden Rückgabeobjekte sehr allgemein gehalten. Datenstrukturen werden aufgrund der Vielzahl an bereits implementierten Formen nicht konkretisiert, sondern über eine Instanz eines Iteratorobjekts realisiert. Die gekapselten Datenstrukturen können somit jederzeit angepasst, optimiert oder

gegen andere Bibliotheksdatenbankstrukturen ausgetauscht werden. Ebenfalls können spezielle auf den hybriden Multicastdienst programmierte Datenstrukturen genutzt werden, vorausgesetzt diese implementieren das Interface Iterable aus dem Packet java.lang.*. Die Objekte und deren Methodenaufrufe der Multicast-API sehen einen vollständigen nebenläufigen Zugriff für Gruppenanwendungen vor, so dass Programmierer der Anwendungssoftware keine expliziten Synchronisationsmechanismen implementieren müssen. Der weiterverbreitete Synchronisationsmechanismus unter Java über das Schlüsselwort synchronized findet bei Methoden eines Multicastsocketobjekts allerdings keine Anwendung, da dem Anwender ein echter nebenläufiger Zugriff der beispielhaften Methoden Recieve und Send eines Socketobjekts ermöglicht werden soll. Das Schlüsselwort synchronized realisiert unter Java das Konzept des Monitors. Monitore kapseln kritische Bereiche und Regeln den exklusiven Zugriff auf diesen. Unter Java wird das “this“ Objekt einer synchronized Methode gesperrt. Die Methoden send() und recieve(), die über eine Instanz einer MulticastSocket-Klasse angesprochen werden, könnten demnach bei Benutzung des Schlüsselwortes synchronized nicht parallel stattfinden. Diese Anforderungen müssen bei einer konkreten Anbindung an einen hybriden Multicastdienst berücksichtigt werden.

Der Entwicklungsprozess der Javamulticast API wird nach den grundlegenden fünf Phasen der Softwareentwicklung (ver. Kecher (2009)) gesteuert. Die Analyse und Definition der 1ten Phase wurde bereits im Draft getätigt (ver. M.Wählisch u. a. (2011)) und an einigen Stellen dieser Arbeit wird aufgrund der Komplexität auf diesen verwiesen. Die 2te Phase Entwurf/Design wird im aktuellen Kapitel diskutiert. Programmiersprachenspezifische Implementierung wird in diesem und nächsten Kapitel 4 besprochen und entwickelt. Klassendiagramme der UML geben uns einen groben Überblick der Konstruktoren und Methoden der Javamulticast API. Phase 4: Test sowie Phase 5: Einsatz und Wartung werden erst im Kapitel 5 besprochen.

Die abstrakte Javaklasse MultiCastSocket.java (siehe Abb. 3.1) ermöglicht eine “ist-eine-Art von “ Beziehung für Unterklassen auszudrücken. MultiCastSocket.java gibt eine Signatur vor, die eine Unterklasse bei Bindung an einen hybriden Multicastdienst konkret implementieren muss. Im folgenden Kapitel werden sukzessive die abstrakten Methoden diskutiert und dabei zum Teil schon auf Implementierungsdetails bei Bindung an den hybriden Multicastdienst H∀Mcast (siehe Abb. 3.2) eingegangen.

Die Vorgabe der abstrakten Klasse MulticastSocket.java ermöglicht somit ein einfaches Austauschen der konkreten Implementierungsklasse ohne große Anpassung der Anwendungssoftware. Voraussetzung für den Austausch ist allerdings, dass die neue Klasse auch ein “ist-ein-Art “ Beziehungzu MulticastSocket hat . Diese dynamische Bindung ermöglicht Polymorphie und ist in der objektorientierten Programmierung ein mächtiges Werkzeug (siehe Listing 3.1).

Vergleicht man nun die abstrakte Klasse mit den Vorgaben des Draft “A Common API for

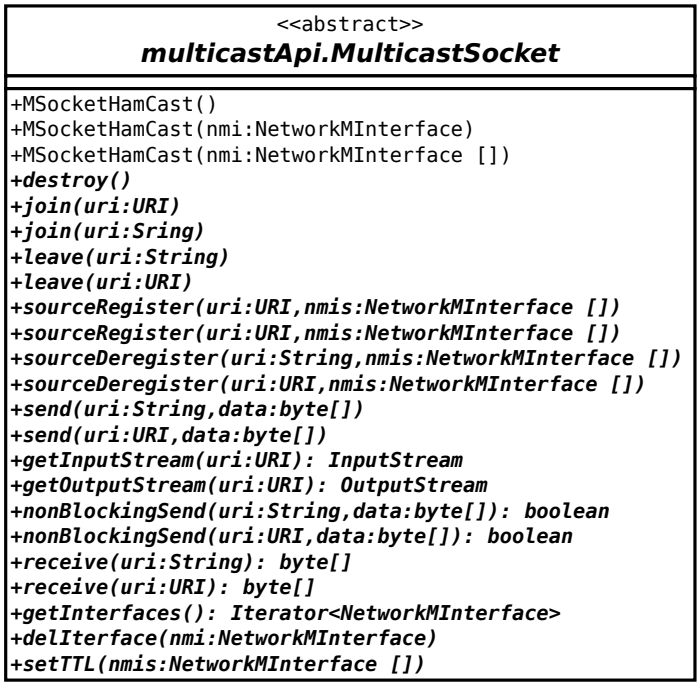

Abbildung 3.1.: Klassendiagramm:abstract MulticastSocket

Transparent Hybrid Multicast“, fällt auf, dass ein Teil der Methoden nicht in der abstrakten Klasse definiert wird. Diese Methoden müssen statisch in der konkreten Unterklasse realisiert und an den genutzten hybriden Multicastdienst gebunden werden. Grund für die statische Realisierung ist, dass die Funktionen in keinem Zusammenhang mit einem Socket stehen und auch aufrufbar sein sollten, ohne die konkrete Erzeugung eines Socketobjekts. Statische Methoden werden unter Java über den Klassennamen angesprochen. Anwendungsimplementierungen, die auf statische Methoden zugreifen, zumeist sind es IMG Funktionalitäten im hybriden Multicast, können nicht so einfach den konkreten Multicastdienst der API austauschen, da über die Namensbindung eine Abhängigkeit zur Klasse besteht. Um Anpassungen minimal zu halten, ermöglicht der statische Import unter Java das Ansprechen statischer Methoden ohne Klassenangabe. Ein einfaches Austauschen der statischen Importzeile (siehe Beispielcode 3.2 reicht, um einen Wechsel des hybriden Multicastdienst für statische Methoden zu erzwingen.

Listing 3.2: statischer Import unter Java

```
1 import static hamCastApi.HybridMulticastSocket.*;
2 import static hamCastApi.NewHybridMulticastSocket.*;
```

Diese Möglichkeit würde Anpassungen durch Austausch der Klasse des konkreten Multicastdiensts, minimal halten. Leider werden die statischen Methoden, die sich im Methodenaufruf von lokalen nicht mehr unterscheiden, nur in speziellen Editoren und Entwicklungsprogram-

Abbildung 3.2.: Klassendiagramm:libmulticastapi.MSocketHamCast

Listing 3.1: Polymorphie Javabeispiel

```
1  // Erzeugung eines Socket für einen hybriden MulticastDienst
2  MulticastSocket mSocket= new HybridMulticastSocket();
3  // versendet eines Packetes über den hybriden Dienst
4  mSocket.send(uri, packet)
5  /**
6  * Polymorphie sorgt für minimale Quellcodeanpssung
7  * beim Umsteig auf einen anderen hybriden Dienst
8  **/
9  mSocket= new NewHybridMulticastSocket();
10 // versendet eines Packetes über den hybriden Dienst
11 mSocket.send(uri, packet)
```

men, wie z.B. Eclipse gekennzeichnet, was die Lesbarkeit des Quellcode minimiert. Bei der Implementierung der Draftspezifikation in Java wurde versucht, möglichst ähnliche Klassennamen, Funktionsaufrufe und Rückgabewerte zu realisieren, wie Java Programmierer sie bereits von Netzwerkprogrammen gewohnt sind.

3.4. Realisierung einer Java Multicast API

3.4.1. Uniform Resource Identifier

Spezifikation einer Multicast URI

Einen DNS Service, ähnlich wie er bereits unter IP-Unicast Kommunikation im Internet bekannt ist, bietet Multicast in seiner nativen Form nicht. Dies zwingt den Programmierer schon in der Phase der Codeentwicklung, Adressen für die Kommunikation zu definieren und sich damit auf der Vermittlungsschicht auf eine Technologie IPv4 oder IPv6 festzulegen. Um einen technologieunabhängigen Dienst zur Verfügung zu stellen, muss die Adressierung und Namensgebung von der verwendeten Technologie abstrahiert werden. Die Spezifikation des Draft für eine hybride Multicast definiert eine URI[1] als Identifier für Multicast Gruppe und Adressen. Das Namensschema einer URI ist im RFC 3986 Berners-Lee (2006) spezifiziert. Eine Anleitung und Empfehlung zur Nutzung von Uniform-Resource-Identifier findet man im RFC 4395 Hansen u. a. (2005).

Eine Multicast Uri wird im Draft folgendermaßen definiert:

scheme ’://’ group ’@’ instantiation ’:’ port ’/’ sec-credentials

1. scheme:Definiert den Namensraum z.B. ip,sip oder scribe

[1] Uniform Resource Identifier

2. group: Identifiziert die Multicastgruppe
3. instantiation: identifiziert die Entität die eine Instanz der Gruppe generiert. z.B. SSM oder eine SIP Domain
4. port: Identifiziert eine bestimmte Anwendung für eine Instanz der Gruppe
5. sec-credentials: Ist für die Implementierung von Sicherheitsmechanismen

URI Implementierung unter Java

Unter Java bedienen wir uns der Klasse URI aus dem Packet java.net [2] (siehe Abb. 3.3).Mit einer großen Anzahl an überladenen Konstruktoren und einer großen Methodenauswahl stellen wir dem Gruppenanwendungsprogrammierer eine sehr mächtige Klasse bereit, die eine flexible Erzeugung der URI erlaubt und spezielle Parsingabfragen auf Komponenten ermöglicht. Das Namensschema der Variablen und deren Methoden aus der Klasse java.net.URI weicht von der aus dem dem Draft vorgegeben Multicast URI ab und muss berücksichtigt werden (siehe Tabelle. 3.1). Eine Übersicht der Befehle und deren Konsolenausgabe ist im Codebeispiel (siehe Listing 3.3) ersichtlich.

Draft	java.net.URI	Java Methodenaufruf
scheme	Scheme	getScheme()
group	user-info	getUserInfo()
instantiation	host	getHost
port	port	getPort
sec-credentials	path	getPath()

Tabelle 3.1.: java.net.URI vergl. Draft Uri

[2] http://download.oracle.com/javase/1.4.2/docs/api/java/net/URI.html

Listing 3.3: Uri.*();

```
1 URI uri = new URI("scribe://foobar@hallowelt:1234/test");
2 System.out.println(uri.getScheme()); //scribe
3 System.out.println(uri.getSchemeSpecificPart()); //foobar@hallowelt:1234/test
4 System.out.println(uri.getUserInfo()); //foobar
5 System.out.println(uri.getHost()); //hallowelt
6 System.out.println(uri.getPort()); //1234
7 System.out.println(uri.getPath()); ///test
```

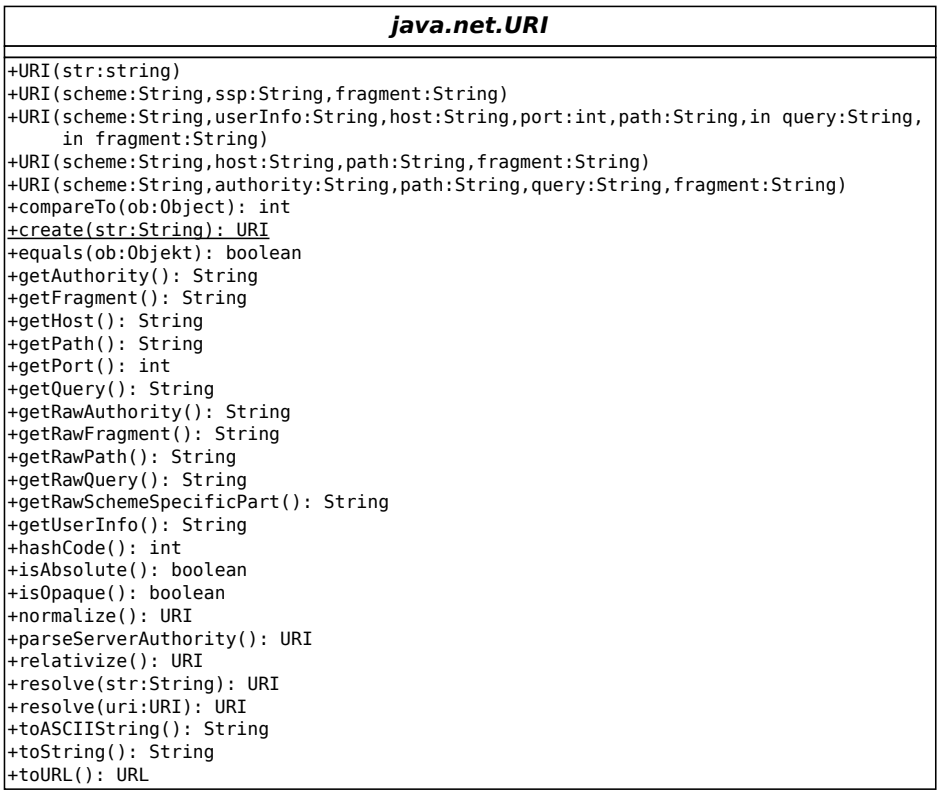

Abbildung 3.3.: Klassendigramm: java.net.URI

3.4.2. Multicastinterfaces für transparentes hybrides Multicast

Spezifikation eines Interfaces

Die aufgeführte Datenstruktur “if_prop“, hält Informationen über ein verfügbares Multicastinterface und berücksichtigt bei der Identifizierung die Vorgaben aus dem RFC 3493 (vgl. Gilligan (2003) S.16), die einen Index “if_index“ aus dem Bereich der ganzen positiven Zahlen

vorschlägt, beginnend mit dem Wert 1. Neben dem Namen des logischen Interface, enthält die Struktur noch die Adresse sowie Information über das Multicasttransportverfahren.

```
struct if_prop {
        unsigned int if_index; /* 1, 2, ... */
        char        *if_name;  /* "eth0", "eth1:1", "lo", ... */
        char        *if_addr;  /* "1.2.3.4", "abc123", ... */
        char        *if_tech;  /* "ip", "overlay", ... */
      };
```

- if_index: Beschreibt den Identifier des Interface.
- if_name: Beschreibt den Namen des Interface.
- if_addr: Beschreibt die Adresse des lokalen Interface.
- if_tech: Beschreibt die Technologie des Interfaces.

Interface implementierung unter Java

Unter Java wird ein Multicastinterface durch eine Instanz der Klasse NetworkMInterface repräsentiert(siehe Abb. 3.4). Die für die API speziell programmierte Klasse ist Teil des Pakets MulticastApi und besitzt viele Ähnlichkeiten zur Klasse NetworkInterface aus dem Packet java.net [3], welches versierten Java-Netzwerk-Programmierern bekannt sein dürfte.

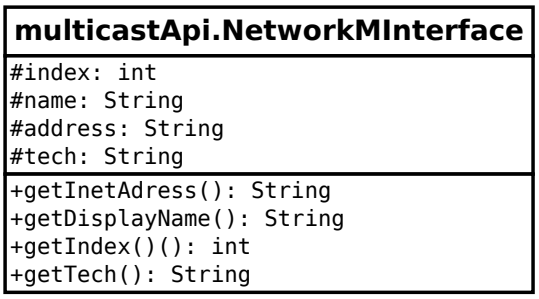

Abbildung 3.4.: Klassendiagramm: multicastApi.NetworkMInterface

[3] http://download.oracle.com/javase/1.4.2/docs/api/java/net/NetworkInterface.html

3.4.3. getAllInterface()

Spezifikation von getAllInterface()

```
getInterfaces(out Int num_ifs, out Interface <if>);
```

- num_ifs: Beschreibt die Anzahl der Listenelemente.
- Interface<if>: Enthält eine Liste mit if Elementen.

Das Ergebnis gibt eine parametisierte Liste “Interface“ aller aktiven verfügbaren Interfaces zurück. Die Größe der Liste wird durch den Parameter “num_ifs “ angegeben.

getInterface() implementierung unter Java

Listing 3.4: static Iterator<NetworkMInterface> getInterfaces()

```
1 public static Iterator<NetworkMInterface> getInterfaces(){}
```

Eine konkrete Unterklasse von MulticastSocket aus dem Paket multicastApi, in unserem Beispiel MSocketHamCast, sollte die statische Methode getInterfaces() bereitstellen. Dieser Methodenaufruf liefert dem Programmierer einen generischen Iterator vom Typ NetworkMInterface, der eine individuelle Untersuchung der Datenstruktur ermöglicht (siehe Listing 3.5).

Listing 3.5: MSocketHamCast.getInterface()

```
1 Iterator<NetworkMInterface> netMInter =MSocketHamCast.getInterfaces();
2 while(netMInter.hasNext()){
3   NetworkMInterface nmi = netMInter.next();
4
5   System.out.println("index:␣"+ nmi.getIndex());
6   System.out.println("name:␣"+ nmi.getDisplayName() );
7   System.out.println("adress:␣" + nmi.getInetAdress());
8   System.out.println("technology:␣" + nmi.getTech() + "\n");
9 }
```

Betrachtet man nun einmal die klassische getInterface() Methode der Javanetzwerk-API aus dem Packet java.net, ist kaum ein Unterschied im Aufruf der getInterface() Methode zu erkennen (vergleiche Listing 3.5 mit 3.6). NetworkInterface.getNetworkInterfaces() gibt historisch bedingt ein Enumeration zurück. In der Version Java 1.0, seit dem es auch schon die Javanetzwerk-API gibt, war Enumeration, der Standarditerator für Datenstrukturen. Erst mit Einführung der Collection-Api in Java 1.2 wurden Datenstrukturen deutlich handlicher und skalierbarer(vgl. Ullenboom (2011),S.633). Neu für diese Datenstrukturen, ist der Iterator, der heutzutage deutlich verbreiteter als Enumeration ist und deshalb auch als Rückgabewert der getnetworkInterfaces() Methode der Klasse MSocketHamCast dient. Seit Java 5.0 mit

Einführung der Generics ist dieser auch typgebunden und minimiert dadurch das Risiko von RunTimeException [4] durch Programmierfehler.

Listing 3.6: NetworkInterface.getNetworkInterfaces()

```
1  Enumeration<NetworkInterface> nie = NetworkInterface.getNetworkInterfaces();
2  int n=0;
3
4  while ( nie.hasMoreElements() ) {
5    NetworkInterface ni = nie.nextElement();
6
7    System.out.println("index:␣"+ ni.getIndex(); //Nur im Packet möglich
8    System.out.println("name:␣"+ ni.getDisplayName() );
9    System.out.println("adress:␣" + ni.getInetAddresses();
10 }
```

3.4.4. Group Management

3.4.4.1. createMSocket()

Spezifikation von createMSocket()

```
createMSocket(in Interface <if>,out Socket s);
```

- in if: Übergabe einer Liste von Interfacekennung zur Bindung an das Socket(optional).
- out s: Identifiziert das Socket durch ein eindeutige SocketID

Die Funktion createMSocket() erstellt ein Multicastsocket. Der Funktionsaufruf ermöglicht optional die Übergabe einer Liste von Interfacekennungen, um das Socket an ein bestimmtes Interfaces zu binden. Bei Übergabe eines leeren Parameters wird das default Interface gewählt. Als Rückgabeparameter erhält man eine eindeutige Socket ID, über die sich das Socket identifizieren lässt. Ein Fehler im Funktionsaufruf wird im Rückgabewert null gekennzeichnet.

createMSocket() implementierung unter Java

Listing 3.7: MSocketHamCast() überladene Konstruktoren

```
1  public MSocketHamCast() throws IOException
2  public MSocketHamCast(NetworkMInterface nmi) throws IOException
3  public MSocketHamCast(NetworkMInterface [] nmis) throws IOException
```

[4] http://download.oracle.com/javase/1.4.2/docs/api/java/lang/RuntimeException.html

Für die Erstellung eines MulticastSocket wird in der objektorientierten Sprache Java ein Objekt erstellt. Dieses Objekt ermöglicht den direkten Zugriff auf Socketoperationen. Bei der Initialisierung des Objekts wird im ausgewählten Konstruktor eine Verbindung zum hybriden Multicastdienst aufgebaut. Die abstrakte Klasse MulticastSocket gibt eine Grundstruktur an abstrakten Methoden vor, die in einer konkreten Klasse (siehe Abb. 3.2), mit Inhalt gefüllt werden muss. Neben den abstrakten Methoden werden in der Mutterklasse ebenfalls Empfehlungen für Konstruktoren durch leere Deklarationen angegeben, die bei Bedarf in der Unterklasse überschrieben werden können. Diese konkrete Implementierung der Klassemethoden ist abhängig vom hybriden Multicastdienst sowie dem genutzten RMI [5] Verfahren bei Ansiedlung des hybriden Multicastdienstes in der Middleware. Möglichkeiten wären hier Java-RMI, Corba oder ein eigenes RMI-Verfahren(vgl. Coulouris (2002)). Polymorphismus ermöglicht ein einfaches Austauschen des hybriden Multicastdienstes (siehe Listing 3.8). Der überladene Konstruktor ermöglicht ebenfalls die Übergabe eines einzelnen NetworkMInterface Objekts oder eines ObjektArrays vom Typ NetworkMInterfaces. Diese Interfaces werden bei der Erstellung des Socket als Empfangs.- oder Verteilungskanal genutzt. Bei leerer oder fehlerhafter Übergabe wird das Socket an das default Interface gebunden. Bei Kommunikationsproblemen mit dem Middlewaredienst wird eine IOException aus dem Java Packet java.io [6] ausgelöst.

3.4.4.2. deleteMSocket()

Spezifikation von deleteMSocket()

```
deleteMSocket(in Socket s, out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID.
- out error:Im Fehlerfall enthält der Parameter eine -1.

Der deleteMSocket() Funktionsaufruf entfernt das Multicastsocket. Zur Destruktion des Sockets wird die eindeutige SocketID “s “ übergeben. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter “error “ mit einer -1 gekennzeichnet.

deleteMSocket() implementierung unter Java

Listing 3.8: abstract void destroy() throws IOException

```
1 public abstract void destroy() throws IOException;
```

[5] Remote Method Invocation Entfernter Methodenaufruf
[6] http://download.oracle.com/javase/1.4.2/docs/api/java/io/IOException.html

Klassen, die eine “Ist-eine-Art-von-Beziehung“ [7] zu der abstrakten Klasse HamcastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. Destroy() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Die Funktion ist für die saubere Destruktion des Multicastsocket-Objekts zuständig. Neben der Mitteilung an den hybriden Multicastdienst über die Socket-Zerstörung sollte diese Funktion die Socketreferenz des Objekt aus allen globalen Listen der API austragen, damit der Garbage Collector dieses Objekt zerstören kann. Bei Kommunikationsproblemen mit dem Middlewaredienst wird eine IOException aus dem Java Packet java.io ausgelöst.

Listing 3.9: MulticastSocket.destroy()

```
1 //Polymorhismus
2 MulticastSocket mSocket = new MSocketHamCast();
3 //  ...Socketoperationen
4 msocket.destroy();
5 //...löscht die Referenz damit der GC zuschalgen kann
6 msocket=null;
```

Der Garbage Collector ist unter Java für das Zerstören von Objekten zuständig. Entscheidungsgrundlage für das am Leben lassen eines Objektes ist eine gültige Referenzierung. Wenn sich allerdings in der konkreten Klassenimplementierung von MulticastSocket das Objekt beim Erzeugen in globale Listen einträgt, z.B. durch den Konstruktor, wird auch eine Dereferenzierung (siehe Zeile 6 Listing 3.8) keine Speicherfreigabe nach Durchlauf des GC zur Folge haben, da die globalen Listen gültige Referenzen auf das Objekt halten. Ein Aufruf von destroy() ist demnach zwingend notwendig und sollte in der Applikation berücksichtigt werden. Auch in einem Ausnahmefall, der zum Beenden des Threads führen kann, sollte das Hauptprogramm einen Aufruf von destroy() im Catchblock der Ausnahme initiieren, um das Socketobjekt sauber aus der Middleware zu entfernen. Der hybride Multicastdienst in der Middleware sollte auch weiterhin aktiv bleiben und anderen Javaprogrammen performant zur Verfügung stehen. Rudimente von abgestürzten Programmen, die ihre Socket nicht sauber aus dem hybriden Multicastdienst ausgetragen, stören allerdings diese Vorgabe. Im Falle einer RuntimeExeption, die ggf. auf Fehler im Programmcode der Multicastkommunikationsapplikation schließen lässt, wird empfohlen, die finalize() Methode des Client zu überschrieben. (siehe Codebeispiel 3.10)

Listing 3.10: MulticastSocket.finalize()

```
1 protected void finalize(){
2         mSocket.destroy();
3 }
```

Der GC ruft diese Methode auf, wenn er ein Objekt findet, auf welches keine Referenzierung mehr besteht. Tatsächlich garantiert die Sprachspezifikation nicht, dass ein Destruktor überhaupt aufgerufen wird. (vgl. Krüger (2009) S.175). Wenn er aber aufgerufen wird, so erfolgt

[7] public class MSocketHamCast extends MulticastSocket

dies nicht, wenn die Lebensdauer des Objekts endet, sondern dann, wenn der Garbagecollector den für das Objekt reservierten Speicher zurückgibt. Dieses kann unter Umständen sehr lange dauern, da der GC ein asynchroner Hintergrundprozess mit einer niedrigen Priorität ist. Wenn das Hauptprogramm beendet wird, bevor der GC angelaufen ist, RuntimeException im Mainthread, werden auch keine Destruktoren aufgerufen. Das Hauptprogramm gibt den HeapSpeicher an die MMU[8] des Betriebssystem zurücks. (vgl. **?**)

3.4.5. Abonnieren und Aufkündigen von Packeten

3.4.5.1. join()

Spezifikation von join()

```
join(in Socket s, in Uri group_name, out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in group_name: Multicast-Gruppenadresse deren Packete Empfangen werden sollen.
- out error:Im Fehlerfahl enthält der Parameter eine -1

Der Funktionsaufruf join() abonniert die Pakete der Multicastgruppe “group_name“ . Die Adressierung dieser Gruppe erfolgt durch eine Uri . Das Socket, welches Pakete, die an die Uri adressiert sind, empfangen soll, wird durch den Übergabeparameter “s“ identifiziert. Der Funktionsaufruf initiiert im hybriden Multicastdienst für natives Multicast, einen IGMP Report für IPv4 (vgl. B.Cain (2002), RFC 3376), einen MLD Report für IPv6(vgl. Vida und Costa (2004), RFC 3810). Bei Overlaymulticast ein Overlayabonnement. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter “error“ mit einer -1 gekennzeichnet.

join() implementierung unter Java

Listing 3.11: MulticastSocket.join()

```
1 public void join(URI uri) throws IOException
2 public void join(URI uri []) throws IOException
3 public void join(String uri_s) throws IOException, URISyntaxException
```

[8]Memory Management Unit

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MulticastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. Join() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Der Methodenaufruf abonniert die Multicastpakete der Gruppe “uri“ . Als gültige Übergabeparameter(siehe Listing 3.11) wird ein Objekt der Klasse URI aus dem Packet java.net.URI gefordert. Alternativ ist auch die Parameterübergabe eines Strings möglich, der nach Vorgaben des RFC 3986(ver. Berners-Lee (2006)) aufgebaut ist. Neben der möglichen IOException kann dieser Methodenaufruf noch eine URISyntaxException werfen (siehe Listing 3.12). Eine Array Übergabe mit beinhalteten URI’s ist ebenfalls möglich. In diesem Fall wird impliziert für jede URI ein Join für das Socket initiiert.

Implementierungsbesonderheiten von join() unter H∀Mcast

Der Methodenaufruf von join() ist bei Instantiierung der Klasse MsocketHamCast, die den hybriden Multicastdienst H∀Mcast *aus Kapitel 2.4 nutzt, nicht explizit nötig, da ein Receive() auf eine bestimmte URI implizit ein join() veranlasst, falls noch nicht geschehen.*

Listing 3.12: Beispiel MulticastSocket.join()

```
1 try {
2   mSocket.join(new URI("ip//[FF02::3]:1234"));
3 } catch (IOException e1) {
4   System.err.println("Fehler in der Middleware beim abonnieren der Gruppe");
5   e1.printStackTrace();
6 } catch (URISyntaxException e1) {
7   System.err.println("Syntaxfehler der URI");
8   e1.printStackTrace();
9 }
```

3.4.5.2. leave()

Spezifikation von leave()

```
leave(in Socket s, in Uri group_name, out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in group_name: Multicast Gruppenadresse, von der keine Packete mehr empfangen werden sollen.
- out error:Im Fehlerfahl enthält der Parameter eine -1

Der Funktionsaufruf leave() kündigt das Abonnement einer Multicastgruppe. Die Identifizierung der zu kündigenden Gruppe geschieht über “group_name“. Das Socket, welches die an die Uri adressierten Pakete in Zukunft nicht mehr empfangen will, wird durch die eindeutige Socket-ID “s“ identifiziert. Der Aufruf löst in der Middleware einen IGMP/MLD Report für natives Multicast aus, bei Overlaymulticast eine Overlay-Benachrichtigung über die Kündigung des Multicastgruppen- Abonnements. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

leave() implementierung unter Java

Listing 3.13: MulticastSocket.leave()

```
1 public void leave(String uri) throws IOException, URISyntaxException
2 public void leave(URI uri) throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MulticastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. Leave() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Der Methodenaufruf leave() ermöglicht einer konkret implementierten Unterklasse von MulticastSocket, ein Abonnement einer Gruppe aufzukündigen. Parameterübergabe einer URI aus dem Packet java.net oder eines Strings sind möglich. Neben der möglichen IOException aus dem Packet java.IO , kann bei Übergabe eines Strings auch eine URISyntaxException aus dem Packet java.net ausgelöst werden.

Implementierungsbesonderheiten von leave() unter HamCast

In der Klasse MsocketHamCast, die den hybriden Multicastdienst H∀Mcast *aus Kapitel 2.4 nutzt, sorgt ein leave() Aufruf spezifikationsgemäß für eine Kündigung der Gruppe und löscht dabei alle bereits empfangenden Pakete, die noch nicht von der Anwendung aus dem Kommunikationsmodul abgeholt worden sind.*

3.4.5.3. sourceRegister()

Spezifikation von sourceRegister()

```
srcRegister(in Socket s, in Uri group_name,
                    out Int num_ifs, out Interface <ifs>,
                    out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in group_name: Multicastadresse an die Packete gesendet werden sollen.
- out num_ifs: Anzahl der Interfaceelemente in der Liste(optional)
- out Interface: Interfacesliste (optional)
- out error:Im Fehlerfahl enthält der Parameter eine -1

Die Funktion sourceRegister() ist nur für einige Multicastverfahren geeignet und aktiviert alle Interfaces des Socket “s“ für den Versand von Multicastnarichten an die Adresse “group_name“. Optional kann eine Liste “Interface“ zurückgeliefert werden, deren Indizes über den Erfolg der Registrierung des zugehörigen Interface informieren. “num_ifs“ gibt dabei die Größe der Liste an und im Falle des Wertes 0 für “num_ifs“ wird als Liste null zurückgegeben.

sourceRegister() implementierung unter Java

Listing 3.14: MulticastSocket.sourceRegister()

```
1 public NetworkInterface [] sourceRegister(String uri_s, NetworkInterface [] nmi) throws
      URISyntaxException, IOException
2 public NetworkInterface [] sourceRegister(URI uri, NetworkInterface [] nmi) throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klassen MulticastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. sourceRegister() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Der Methodenaufruf sourceRegister() aktiviert alle NetworkMInterface des Sockets s für den Versand von Multicastpacketen an die URI. Parameterübergabe einer URI aus dem Packet java.net oder eines Strings sind möglich. Neben der möglichen IOException aus dem Packet java.IO kann bei Übergabe eines Strings auch eine URISyntaxException aus dem Packet java.net ausgelöst werden.

3.4.5.4. sourceDeregister()

Spezifikation von sourceDeregister()

```
srcDeregister(in Socket s, in Uri group_name,
                    out Int num_ifs, out Interface <ifs>,
                    out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in group_name: Multicastadresse, deren Packete die nicht mehr über alle Interface verschickt werden sollen.
- out num_ifs: Anzahl der Interfaceelemente, für die eine Abmeldung fehlgeschlagen ist(optional)
- out Interface: Interfacesliste (optional)
- out error:Im Fehlerfahl enthält der Parameter eine -1

Die Funktion sourceDeregister() ist nur für einige Multicastverfahren geeignet und meldet alle Interfaces des Socket “s“ für den Versand von Multicastnachrichten an die Adresse “group_name“ ab. Optional kann eine Liste “Interface“ zurückgeliefert werden, deren Indizes über den Erfolg der Abmeldung des zugehörigen Interface informieren. “num_ifs“ gibt dabei die Größe der Liste an und im Falle des Wertes 0 für “num_ifs“ wird als Liste null zurückgegeben .

sourceDeregister() implementierung unter Java

Listing 3.15: MulticastSocket.sourceDeregister()

```
1 public NetworkInterface [] sourceDeregister(String uri_s, NetworkInterface [] nmi) throws
      URISyntaxException, IOException
2 public NetworkInterface [] sourceDeregister(URI uri, NetworkInterface [] nmi) throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klassen MulticastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. sourceDeregister() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Der Methodenaufruf sourceDeregister() deaktiviert auf allen NetworkMInterface des Sockets s den Versand für Multicastpakete, die an die URI adressiert sind. Parameterübergabe einer URI aus dem Packet java.net oder eines Strings sind möglich. Neben der möglichen IOException aus dem Paket java.IO , kann bei Übergabe eines Strings auch eine URISyntaxException aus dem Paket java.net ausgelöst werden.

3.4.6. Send and Receive Primitiven

3.4.6.1. send()

Spezifikation von send()

```
send(in Socket s, in Uri group_name,out Size msg_len,
          out Msg msg_buf,out Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in msg_len: Gibt die Länge des Paketes an
- in msg_buf: Das zuversendene Packet
- out error:Im Fehlerfahl enthält der Parameter eine -1

Der Funktionsaufruf send() ermöglicht das Versenden des Paketes “msg_buf“ an eine Multicastgruppe “group_name“. Über die Socked ID “s“ wird das Socket gewählt. Ein erfolgreicheres Senden an eine Multicastgruppe wird mit einer 0 , ein Fehler mit einer -1 bestätigt. Da Multicast allerdings UDP auf der Transportschicht nutzt, kann keine Aussage darüber getroffen werden, ob das Paket beim Client wirklich angekommen ist.

send() implementierung unter Java

Listing 3.16: MulticastSocket.send()

```
1 public void send(URI uri, byte [] data) throws IOException
2 public void send(String uri_s, byte [] data) throws IOException, URISyntaxException
```

Klassen, die eine Ïst-eine-Art-von-Beziehungßu der abstrakten Klasse MulticastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. send() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Der Aufruf der send() Methode über eine Instanz einer konkret implementierten Unterklasse von MulticastSocket ermöglicht das Versenden von Paketen an eine Multicastgruppe. Adressiert wird die Empfängergruppe über eine URI. Entweder wird eine Instanz der Klasse Java.net.URI übergeben oder ein String, der nach Vorgaben des RFC 3986 Berners-Lee (2006) aufgebaut ist. Neben der möglichen IOException aus dem Packet java.IO, kann bei Übergabe eines Strings auch eine URISyntaxException aus dem Packet java.net geworfen werden.

Implementierungsbesonderheiten von send() unter H∀Mcast

Listing 3.17: boolean nonBlockingSend()

```
1 public boolean nonBlockingsend(URI uri, byte [] data) throws IOException
2 public boolean nonBlockingsend(String uri_s, byte [] data) throws IOException, URISyntaxException
```

Die void-Funktion send() kann bei gefülltem Sendepuffer im IPC Modul, welches die Daten an die Middleware schickt, zeitweilig blockieren, bis die Methode die Daten in den Puffer ablegen kann. Wenn diese Funktionalität nicht gewünscht ist und der Anwendungsthread bei gefülltem Puffer nicht blockieren soll, kann die Anwendung den Methodenaufruf nonBlockingSend() nutzen. Dieser informiert über einen booleschen Rückgabeparameter über den Erfolg des Senders. Diese Methode ist in der abstrakten Klasse MulticastSocket nicht als abstrakt gekennzeichnet. Wünscht eine spezielle API allerdings diese Funktionalität, muss diese Methode in der konkreten Implementierung überschrieben werden. Wird diese Methode nicht überschrieben, besitzt Sie die gleichen Eigenschaften wie send() und kann zeitweilig blockieren (siehe Listing 3.18)

Listing 3.18: nonBlockingSend() default Implementation

```
1  public boolean nonBlockingSend(String uri_s, byte[] data) throws IOException, URISyntaxException{
2
3          send(uri_s, data);
4
5          return true;
6  }
7
8  public boolean nonBlockingSend(URI uri, byte[] data) throws IOException {
9
10         send(uri, data);
11
12         return true;
13 };
```

3.4.6.2. receive()

Spezifikation von receive()

```
receive(in Socket s, in Uri group_name,
in Size msg_len, in Msg msg_buf,
                 in Int error);
```

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in group_name: Multicast-Gruppenadresse deren Packete Empfangen werden sollen.

- out msg_len: Länge des empfangenden Multicast-Packet.
- out msg_buf: Multicast-Packet.
- out error:Im Fehlerfahl enthält der Parameter eine -1

Der Funktionsaufruf receive() empfängt Daten der URI “group_name“ auf dem Socket “s“, welches durch die Socket ID identifiziert wird. Das empfangende Paket befindet sich im Array “msg_buf“, deren Länge über den Parameter “msg_len“ empfangen wird. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

Seit der Version 01 des Draft wird vorgeschlagen, “group_name“ als out Parameter zu implementieren. Logik der automatischen Paketaufteilung nach Empfänger würde aus der API in die Anwendung verlegt werden. Die Api Version dieser Bachelorarbeit berücksichtig dieses noch nicht, da die neuere Draftversion zum Ende der Entwicklungszeit publiziert worden ist.

receive() implementierung unter Java

Listing 3.19: MulticastSocket.recieve()

```
1 public byte[] recieve(String uri_s) throws IOException, URISyntaxException
2 public byte[] receive(URI uri) throws IOException
3 //Nicht blockierende Funktionen
4 public byte[] recieve(String uri_s,int wait) throws IOException, URISyntaxException,TimeOutException
5 public byte[] receive(URI uri, int wait) throws IOException, TimeOutException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MultiCastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. Receive() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden . Eine Instanz einer konkret implementierten Klasse, z.B. MSocketHamCast bietet die Methode receive() an. Der blockierende Aufruf dieser Methode ruft empfangende Pakete, die als Byte Array übergeben werde und an die Multicast Uri adressiert worden sind, ab. Wenn noch keine Pakete empfangen worden sind, blockiert diese Methode. Als Parameter wird eine URI oder ein String, der nach den Spezifikation des RFC 3986 (ver. Berners-Lee (2006)) aufgebaut ist, übergeben. Neben der möglichen IOException aus dem Packet java.IO , kann bei Übergabe eines Strings auch eine URI SyntaxException aus dem Packet java.net geworfen werden.

Die überladenen Konstruktoren ermöglichen auch einen zeitlich gesteuerten blockierenden Leseaufruf. Diese Methoden sind nicht abstrakt und bereits konkret in der Mutterklasse MulticastSocket implementiert. Der Parameter wait des primitiven Datentyps int gibt die Wartezeit

in Millisekunden an. Die Blockierung wird durch einen Exceptionwurf des Typs TimeOutException aus dem Paket java.lang.Exception ausgelöst, wenn in der Zeit “wait“, kein Paket empfangen worden ist.

3.4.6.3. Besonderheiten von receive() unter H∀Mcast

Wenn im Vorwege kein explizites join() für die Multicastgruppe URI aufgerufen worden ist, wird implizit beim ersten receive() Aufruf ein Join auf diese Uri ausgeführt. Für einen sauberen Coding Style wird allerdings der explizite Aufruf von join() empfohlen, um einen anschließenden Methodenaufruf von leave() nicht zu vergessen.

3.4.6.4. Java spezifische Erweiterungen für send() und receive()

Betrachten wir eine kleine Unicast Serverroutine (siehe Listing 3.20) unter Java, die in regelmäßigen Abständen Socketoperationen durchführt, so fällt uns auf, dass eine Hilfsklasse PrintWriter() (siehe Abb. 3.5) bei Schreibprozessen unter anderem ein direktes Schreiben von Strings ermöglicht, ohne dabei spezielle offensichtliche I/O Schreibebefehle zu nutzen. Beim genaueren Untersuchen des Klassendiagrammes von PrintWriter 3.5 entdeckt man viele bereits bekannte I/O Methoden, die darauf schließen lassen, dass die wohl bekannteste Klasse System.out auch eine Subklasse von Printwriter ist. Somit hat jeder Javaprogrammierer, der in seinen ersten Versuchen mit System.out.println("Hallo Welt") eine Konsolenausgabe programmiert hat, indirekt schon einmal Kontakt mit einer PrintWriter Instanz gehabt. Der Konstruktor von PrintWriter erwartet ein OutputStream Objekt, welches die Methode getOutputStream()(siehe Listing 3.20, Zeile 3) der Klasse Socket für Unicastkommunikation liefert.

Listing 3.20: getOutputStream()

```
1  private static void sending( Socket client) throws IOException
2  {
3    PrintWriter out = new PrintWriter( client.getOutputStream(), true );
4    out.println("Hallo Welt");
5  }
```

Betrachtet man einmal die abstrakte Basisklasse OutputStream (siehe Abb. 3.6), die PrintWriter bei der Erzeugung eines Objekts fordert, erkennt man schnell, welche Ersparnis Vererbung unter Java bringen kann. Alle API’s die I/O Kommunikation realisieren , sind unter Java Unterklassen von OutputStream oder sie stellen eine zusätzliche Klasse bereit, welche diese Eigenschaft besitzt. Jede dieser Unterklassen hat ihre eigene Implementierung,

[8] http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/TimeoutException.html

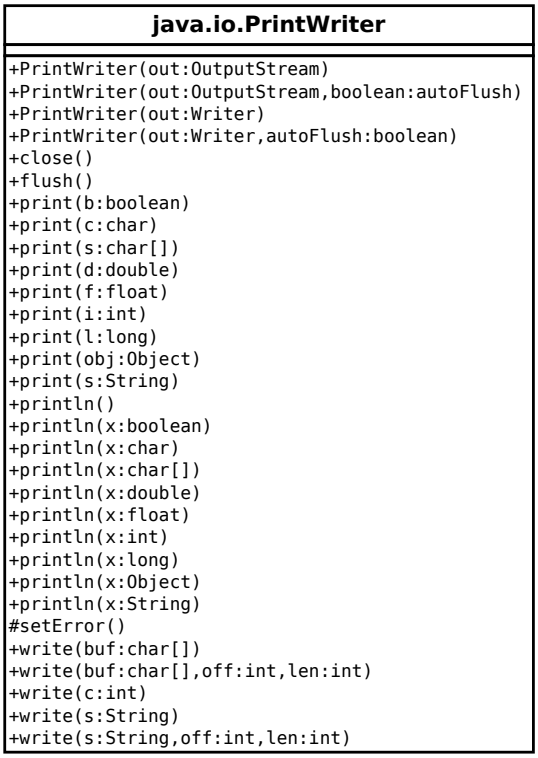

Abbildung 3.5.: Klassendigramm: java.io.PrintWriter

um I/O Kommunikation mit der Gegenstelle zu realisieren. Die Gegenstelle kann eine Datei sein, oder wie in unserem Fall auch ein entfernter Rechner, der über Socketoperationen angesprochen wird. Um eine konkrete Ausgabe der Methoden der Klasse PrintWriter zu gewährleisten, muss mindestens die abstrakte Methode write(int b) (siehe Beispielcode 3.21) implementiert werden und an die I/O Kommunikation angepasst werden.

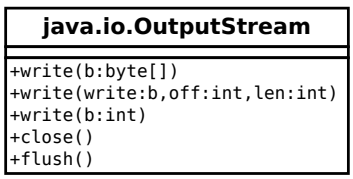

Abbildung 3.6.: Klassendigramm: java.io.OutputStream

Listing 3.21: write(int arg0)

```
1 @Override
2 public void write(int arg0) throws IOException {
3   byte[] sign={(byte)arg0};
4   mSocket.send(uri,sign);
5 }
```

Der Methodenaufruf println("Hallo Welt") eines PrintWriters nutzt die Funktion void write(byte[] b, int off, int len). Hier stellt sich natürlich die Frage, wie eine Kommunikation mit unserem MulticastSocket Objekt zustande kommt, wenn doch nur die abstrakte write Methode 3.21, angepasst worden ist. Die Betrachtung des Quellcode (siehe Beispielcode 3.22) gibt Aufschlüsse darüber und wirft Probleme auf.

Listing 3.22: write(...)

```
1  public void write(byte<span class="listing">[]</span> b, int off, int len) throws IOException {
2  if (b == null)
3      throw new NullPointerException();
4  else if ((off < 0) || (off > b.length) || (len < 0) ||((off + len) > b.length) || ((off + len) < 0))
       {
5      throw new IndexOutOfBoundsException();}
6   else if (len == 0)
7      return;
8  for (int i = 0 ; i < len ; i++)
9              write(b[off + i]);
10   }
```

Der Methodenaufruf println("Hallo Welt") des PrintWriter Objekt reicht den String Parameter weiter an die Methode write(byte[] b, int off, int len) des OutputStream Objekt. Konkret ist allerdings nur die abstrakte Methode write(int arg0) an das MulticastSocket gebunden. Beim Betrachten des Quellcode der write(byte[] b, int off, int len) Methode wird offensichtlich, dass diese Methode implizit die abstrakte Methode write() nutzt. Der Methodenaufruf out.println("Hallo Welt") würde in dieser einfachen Form jeden Buchstaben als einzelnes Paket verschicken und die Übertragungseffizienz durch den Packetoverhead stark minimieren. Ein Überschreiben der Methoden write(byte[] b, int off, int len) und public void write(byte[] b) wird von der Java-Vererbungshierarchie unterstützt und gestaltet die Übertragung deutlich effizienter (siehe Beispielcode 3.23). Bei der Benutzung der Methoden der PrintWriter-Klasse printf(), println() sollte natürlich beachtet werden, dass Java diese Aufrufe puffert und ein flush() erst für die endgültige Ausgabe sorgt. Der Konstruktor von PrintWriter ermögliche eine Aktivierung von Autoflush.(siehe Beispielcode 3.24, Zeile:11) Somit wird bei jedem Aufruf von printf... der puffer geleert.

Listing 3.23: overrite write()

```
1  @Override
2  public void write(byte[] arg0, int arg1, int arg2) throws IOException {
3    mSocket.send(uri,Arrays.copyOfRange(arg0, 0, arg2));
4  }
5
6  @Override
7  public void write(byte[] b) throws IOException {
8    mSocket.send(uri,b);
9  }
```

Listing 3.24: new PrintWriter()

```
1  public static void main(String[] args) throws  URISyntaxException, InterruptedException, IOException
       {
2
3  MSocketHamCast mSocket=new MSocketHamCast();
4
5  PrintWriter out=null;
6  URI uri= new URI("ip://239.0.1.1:1234");
7
8  out = new PrintWriter(mSocket.getOutputStream(uri),true);
9
10 for (int i=0;i<=1000000;i++) {
11   String hello_world = "Hallo Welt " + "Nr.:"+i;
12   out.println(hello_world);
13 }
14
15 Thread.sleep(3000);
16 mSocket.destroy();
17 System.out.println("Done");
18         }
19 }
```

Das Gegenstück zu OutputStream ist die abstrakte Klasse InputStream (siehe Abb.3.7). Jeder binäre Eingangsstrom unter Java repräsentiert sich als eine Unterklasse von InputStream. Beispielhaft ist die Konsoleneingabe System.in eine Unterklasse von InputStream. Die verschiedenen Protokolle zum Lesen aus den Eingangströmen werden in den konkreten Subklassen von Inputstream implementiert. Die read() Methode der Klasse ist abstrakt und fordert eine konkrete Implementierung an einen Datenstrom. Für ein effizientes Lesen wird allerdings empfohlen, alle weiteren Methoden zu überschreiben und diese explizit an das Socket zu binden. Das Gegenstück zur Klasse PrintWriter, welche Schreibeoperationen mit

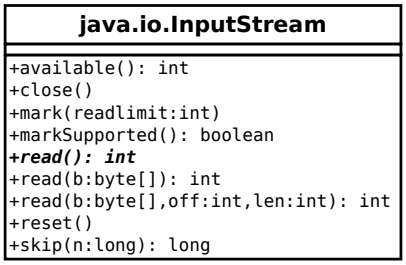

Abbildung 3.7.: Klassendigramm: java.io.InputStream

Hilfe der Klasse OutputStream für Datenströme unter Java ermöglicht, ist die Klasse Scanner. [9],die im Konstruktor ein InputStream Objekt fordert. Eine Instanz der Klasse Scanner ermöglicht durch die Vielzahl an Methoden neben zeilenweisem auch wortweises Lesen aus Datenströmen (siehe Listing 3.26).

[9]http://download.oracle.com/javase/1.5.0/docs/api/java/util/Scanner.html

Um diese bereits implementierten Methoden zu nutzen, stellt die abstrakte klasse MulticastSocket eine Methode public InputStream getInputStream() bereit. Um die Funktionalität des Scanners für einfache Leseoperationen aus dem hybriden Multicastdienst HamCast zu ermöglichen, wurden die read() Methoden implementiert und überschrieben (siehe Listing 3.25) . Bei Applikationen, die ihr eigenes Protokoll nutzen und die auch eigene Parser integriert haben, macht die Nutzung von Scanner und PrintWriter wenig Sinn. Hier wird natürlich empfohlen, die direkte Socket Methoden send() und receive() zu nutzen, um erstellte Bytepakete 1:1 an den Multicastdatenstrom weiterzugeben.

Listing 3.25: override read()

```
@Override
public int read(byte[] arg0, int arg1, int arg2) throws IOException {

byte[] tmp= mSocket.receive(uri);
System.arraycopy(tmp,0,arg0,arg1,tmp.length);
return tmp.length;

}
```

Listing 3.26: new Scanner()

```
public static void main(String[] args) throws Exception {

MSocketHamCast mSocket = null;
mSocket = new MSocketHamCast();
URI uri= new URI("ip://239.0.1.1:1234");
mSocket.join(uri);
Scanner in = new Scanner(mSocket.getInputStream(uri));
while (true) {
  String str=in.nextLine();
  System.out.println(str);
}
}
```

3.4.7. Socket Options

3.4.7.1. getInterfaces()

Spezifikation von getInterfaces()

```
getInterfaces(in Socket s, out Int num_ifs,
                    out Interface <ifs>, out Int error);;
```

Der Funktionsaufruf getInterfaces(), gibt eine Liste aller aktiven Interfaces für das Socket “s“ zurück . Der Rückgabeparameter “num_ifs“ gibt dabei die Anzahl der Interfaces “ifs“ im Array an. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

- in s: Identifiziert das Socket durch ein eindeutige SocketID
- in num_ifs: Anzahl der Element in Interface.
- in Interface<ifs>: Interfaceliste.
- out error:Im Fehlerfahl enthält der Parameter eine -1

getInterfaces() implementierung unter Java

Listing 3.27: MulticastSocket.getInterfaces()

```
1 public Iterator<NetworkMInterface> getInterfaces() throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MultiCastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. getInterface() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Eine Instanz einer konkret implementierten Klasse, z.B. MSocketHamCast bietet die Methode getInterfaces() an. Im Gegensatz zur statischen Methode getAllInterfaces(), werden bei diesem Methodenaufruf Interfaces des aktiven Socket ausgegeben. Als Rückgabewert bekommt der Programmierer einen Iterator, der es ihm ermöglicht, über die Liste von Objekten des Typs NetworkMInterface zu iterieren. Im Fehlerfall löst der Methodenaufruf eine IOExcetion aus.

3.4.7.2. delInterfaces()

Spezifikation von delInterfaces()

```
delInterface(in Socket s, Interface if,out Int error);
```

Der Aufruf der Funktion delInterface(), löst das Interface “if“ vom Socket “s“. Ein Fehler im Funktionsaufruf wird im Rückgabewertparameter “error“ mit einer -1 gekennzeichnet.

- in s: Identifiziert das Socket durch eine eindeutige SocketID
- in Interface if: Identifiziert eine Verteilungskanal.
- out error:Im Fehlerfahl enthält der Parameter eine -1

delInterfaces() implementierung unter Java

Listing 3.28: MulticastSocket.delInterfaces()

```
1 public void delInterfaces(NetworkMInterface itf) throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MultiCastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. delInterface() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden . Der Methodenaufruf löst das Interface, welches über den Parameter identifiziert und übergeben wird, von der Socketinstanz.

3.4.7.3. setTTL()

Spezifikation von setTTL()

setTTL(in Socket s, in Int h, in Int num_ifs, in Interface <ifs>, out Int error);

Der Funktionsaufruf setTTL konfiguriert die maximale Anzahl “h“ an Hops[10] für die Auswahl an Interfaces “ifs“, die an das Socket “s“ gebunden sind. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

[10] Der Weg zwischen Netwerkknoten
Router

- in s: Identifiziert das Socket durch eine eindeutige SocketID
- in h: Gibt die maximale Anzahl an Hops an.
- in num_ifs: Anzahl der Element in Interface.
- in Interface<ifs>: Interfaceliste.
- out error:Im Fehlerfahl enthält der Parameter eine -1

setTTL() implementierung unter Java

Listing 3.29: MulticastSocket.setTTL()

```
1 public void setTTL(NetworkMInterface [] nmi) throws IOException
```

Klassen, die eine “Ist-eine-Art-von-Beziehung“ zu der abstrakten Klasse MultiCastSocket aufbauen wollen, müssen deren abstrakte Methoden implementieren. getInterface() ist eine dieser abstrakten Methoden und muss in der konkreten Klasse speziell für den hybriden Multicastdienst implementiert werden. Dieser Methodenaufruf ermöglicht einer Instanz von MulticastSocket, die maximale Anzahl an Hops für eine Teilmenge seiner gebundenen Interfaces zu setzen. Als Übergabeparameter dient ein Objektarray, welches die NetworkMInterfaces identifiziert. Im Fehlerfall wird eine IOException ausgelöst.

3.4.8. GroupSet Options

3.4.8.1. groupSet()

Spezifikation von groupSet()

```
int groupSet(in Interface if, out Int num_groups,
                    out GroupSet <groupSet>, out Int error);

struct GroupSet {
         uri group_name; /* registered multicast group */
         int type;       /* 0 = listener state, 1 = sender state,
                            2 = sender & listener state */
       }
```

- in if: Identifiziert das Interface
- in num_groups: Anzahl der Element in GroupSet.
- in GroupSet<groupSet>: Groupsetliste.
- out error:Im Fehlerfahl enthält der Parameter eine -1

Der Funktionsaufruf groupSet() gibt eine Liste “groupSet“ aller aktiven registrierten MulticastAdressen auf den Interfaces “if“ zurück. Die Liste enthält je Element eine Multicastadresse Uri sowie einen Statuswert, der beschreibt, ob die Adresse zum Senden, Empfangen oder für beide Funktionen genutzt wird. Diese Informationen können für Gruppenmanagement und Routingprotokolle hilfreich sein. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

groupSet() implementierung unter Java

Listing 3.30: static void setTTL()

```
1 public static Iterator<GroupSet> groupSet(NetworkMInterface itf) throws IOException
```

Die statische Methode groupSet,() die wir über die konkret implementierte Unterklasse von MulticastSocket ansprechen, erwartet als Übergabe ein NetworkMInterface. Eine Auswahl aller Interface-Objekte bekommen wir über die statische Methode getAllInterface()(siehe Kapitel 3.4.3). Als Rückgabewert liefert der Methodenaufruf eine Instanz auf ein Iteratorobjekt. Diese ermöglicht ein einfaches Auslesen der parametisierten Listenelemente vom Typ

GroupSet. Jedes Groupsetobjekt enthält eine Uri und eine Typendefinition (siehe Klassendiagramm 3.8. Die private deklarierte Variable “type“ kann die Werte 0 für Empfänger, 1 für Sender sowie 2 für Empfänger und Sender annehmen. Um die Informationen aus dem Objekt auszulesen, enthält es get() Methoden. Im Fehlerfall wird eine IO Exception ausgelöst.

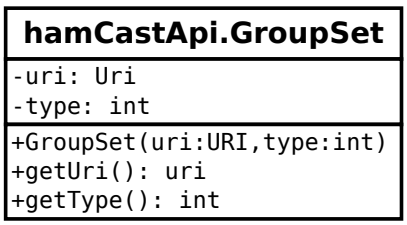

Abbildung 3.8.: Klassendigramm: hamcastApi.GroupSet

3.4.8.2. neighborSet()

Spezifikation von neighborSet()

```
neighborSet(in Interface if, out Int num_neighbors,
                    out Uri <neighbor_address>, out Int error);
```

Die Funktion neigborSet() liefert eine Liste bekannter Nachbarn “Uri<neighbor_adress>“, für das Interface “if“. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

- in Interface if: Identifiziert eine Verteilungskanal.
- out num_neighbors:Anzahl der Listenelemente.
- out neighbor_address:Liste mit allen über eine Uri adressierten Nachbarnknoten.
- out error:Im Fehlerfahl enthält der Parameter eine -1

neighborSet() implementierung unter Java

Listing 3.31: static Iterator<URI> neigborSet()

```
1 public static Iterator<URI> neighborSet(NetworkMInterface nmi) throws IOException
```

Die statische Methode neigborSet,() die wir über die konkret implementierte Unterklasse von MulticastSocket ansprechen, erwartet als Übergabe ein NetworkMInterface. Die Instanz aufs Iteratorobjekt, welche als Rückgabeparameter geliefert wird, ermöglicht ein leichtes Durchlaufen der Liste. Diese Liste enthält alle Nachbarknoten, welche durch ein URI Objekt identifiziert werden. Im Fehlerfall wirft der Methodenaufruf eine IOException.

childrenSet()

Spezifikation von childrenSet()

```
childrenSet(in Interface if, in Uri group_name,
                  out Int num_children, out Uri <child_address>,
                  out Int error);
```

- in if: Identifiziert das Interface
- in Uri group_name: Identifiziert eine Multicastgruppe.
- out num_children: Anzahl der Kinderknoten in der Liste.
- out Uri<child_adress>:Liste mit Kinderknoten, die durch eine Uri addressiert werden.
- out error:Im Fehlerfahl enthält der Parameter eine -1 .

Die Funktion childrenSet() liefert für das ausgewählte Netzwerk “if“ eine Liste aller Kinderknoten, die ein join auf die Multicastgruppe “group_name“ durchgeführt haben. Dieser Funktionsaufruf informiert bei einem Host, der als Multicastrouter arbeitet, über die Routingtabellen des Interfaces. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet.

childrenSet() implementierung unter Java

Listing 3.32: static Iterator<URI> childrenSet()

```
1 public static Iterator<URI> childrenSet(NetworkMInterface nmi, String uri) throws IOException,
      URISyntaxException
2 public static  Iterator<URI> childrenSet(NetworkMInterface nmi, URI uri) throws IOException
```

Die statische Methode childrenSet, (NetworkMInterface itf, URI uri) die wir über die konkret implementierte Unterklasse von MulticastSocket ansprechen, erwartet als Übergabe ein NetworkMInterface und eine Instanz der Klasse java.net.URI. Alternativ ist auch nach den

Vorgaben des RFC3986 (ver. Berners-Lee (2006)) ein String erlaubt, der eine URI repräsentiert. Die Instanz auf ein Iteratorobjekt, die als Rückgabe geliefert wird, ermöglicht ein leichtes Durchlaufen einer Adressliste aller Kinderknoten für das NetworkMInterafce “itf“. Im Fehlerfall wirft der Methodenaufruf eine IOException. Bei Übergabe eines Strings kann ebenfalls eine URISyntaxException auftreten.

3.4.8.3. parentSet()

Spezifikation von parentSet()

```
parentSet(in Interface if, in Uri group_name,
                  out Int num_parents, out Uri parent_address,
                  out Int error)
```

- in if: Identifiziert das Interface
- in Uri group_name: Identifiziert eine Multicastgruppe.
- out num_children: Anzahl der Elternknoten in der Liste.
- out Uri<child_adress>:Liste mit Elternknoten, die durch eine Uri addressiert werden.
- out error:Im Fehlerfahl enthält der Parameter eine -1 .

Der Funktionsaufruf parentSet() liefert für das ausgewählte Netzwerk “if“ eine Liste aller Elternknoten “parent_address“, von denen Multicastpackete mit der Adresse “group_name “ empfangen werden. Ein Fehler im Funktionsaufruf wird im Rückgabeparameter error mit einer -1 gekennzeichnet. Dieser Funktionsaufruf ist für Monitoringsoftware, wie Sebastian Zagaria (ver. Zagaria (2010)) sie in seiner Bachelorarbeit vorstellt, vonnöten.

parentSet() implementierung unter Java

Listing 3.33: static Iterator<URI> parentSet()

```
1 public static Iterator<URI> parentSet(NetworkMInterface nmi, String uri) throws IOException,
      URISyntaxException
2 public static  Iterator<URI> parentSet(NetworkMInterface nmi, URI uri) throws IOException
```

Die statische Methode parentSet, (NetworkMInterface itf, URI uri) die wir über die konkret implementierte Unterklasse von MulticastSocket ansprechen, erwartet als Übergabe ein NetworkMInterface und eine Instanz der Klasse java.net.URI. Alternativ ist auch nach den Vorgaben des RFC3986 (ver. Berners-Lee (2006)) ein String erlaubt, der eine URI repräsentiert. Die Instanz auf ein Iteratorobjekt, die als Rückgabe geliefert wird, ermöglicht ein leichtes

Durchlaufen einer Adressliste aller Elternknoten für das NetworkMInterafce “nmi“. Im Fehlerfall wirft der Methodenaufruf eine IOException. Bei Übergabe eines Strings kann desweiteren ein URISyntaxException auftreten.

3.4.8.4. designatedHost()

Spezifikation von designatedHost()

`designatedHost(in Interface if, in Uri group_name, out Int return)`

- in if: Identifiziert das Interface
- in Uri group_name: Identifiziert eine Multicastgruppe.
- out return: Eine 1 für Forwarder oder Querier, eine 0 für keine spezielle Funktion.

Die Funktion designatedHost() ermittelt, ob ein Host “group_name“ für ein Netwerk “if“ die Rolle eines designated forwarder oder querier eingenommen hat. Solch eine Information wird beinahe von allen Multicastprotokollen zur Vermeidung von Paketduplikation genutzt. Ein Funktionsaufruf wird bestätigt mit dem Rückgabewert 1 für die Rolle als Forwarder oder Querier. Sollte der Host keine dieser Funktion im Netzwerk haben, gibt der Aufruf eine 0 zurück. Eine -1 als Rückgabewert lässt auf einen Fehler schließen.

designatedHost() implemtierung unter Java

Listing 3.34: static Iterator<URI> parentSet()

```
1 public static boolean designatedHost(NetworkMInterface nmi, String uri) throws IOException,
      URISyntaxException
2 public static boolean designatedHost(NetworkMInterface nmi, URI uri) throws IOException
```

Die statische Methode designatedHost (NetworkMInterface nmi, URI uri), die wir über die konkret implementierte Unterklasse von MulticastSocket ansprechen, erwartet als Übergabe ein NetworkMInterface “nmi“ und eine Instanz der Klasse java.net.URI. Alterantiv ist auch nach den Vorgaben des RFC ein String erlaubt. Ein “true“bestätigt die Funktion eines Host “uri“ im Netwerk “nmi“ als Forwarder oder Querier. Im Fehlerfall wirft der Methodenaufruf eine IOException. Bei Übergabe eines Strings kann desweiteren ein URISyntaxException auftreten.

3.4.8.5. enableEvents()

enableEvents()

```
enableEvents();
```

Die Funktionsaufruf enableEvents() informiert die Anwendung über zukünftige Gruppenänderungen im H∀Mcast Netz, Die Funktion ist für den Aufbau und Steuerung eines IMGs nötig.

3.4.8.6. disableEvents()

disableEvents()

Die Funktionsaufruf disableEvents() reicht zukünftige Gruppenänderungen nicht mehr auf Anwendungsebene weiter.

```
disableEvents();
```

4. Realisierung

4.1. Kommunikationsaufbau

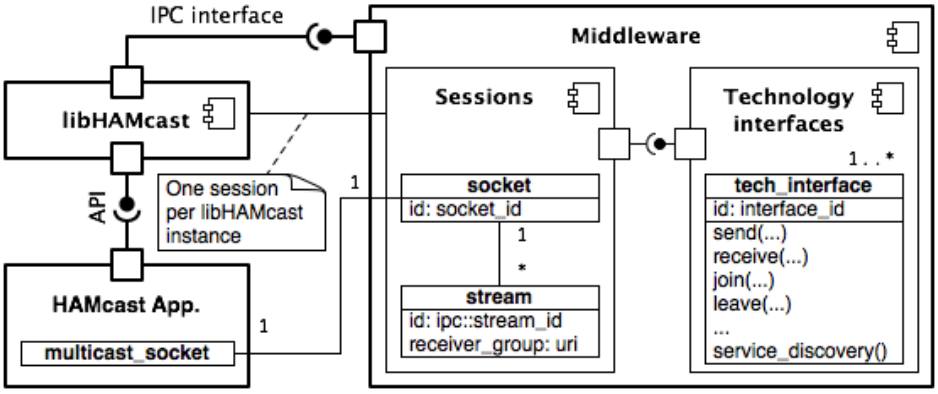

Abbildung 4.1.: Komponentendiagramm H∀Mcast Architektur

Unter Java bietet die Klasse MSocketHamcast (siehe Abb. 3.2) aus dem Paket libHamCast eine konkrete Implementierung der abstrakten Klasse MulticastSocket (siehe Abb. 3.1) aus dem Paket multicastApi. Die Klasse MSocketHamCast enthält einen statischen Initialisierer, der beim Laden der Klasse das IPC-Modul initialisiert und eine Verbindung mit dem hybriden Multicastdienst H∀Mcast aufbaut, der in der Middleware platziert ist,.H∀Mcast ist so konzipiert, dass seine Funktionalität unabhängig mehreren Programmen auf Anwendungsebene zur Verfügung stehen kann. Jede Anwendungsverbindung wird unter H∀Mcast in einer Sessionkomponente separiert. Für einen erfolgreichen Kommunikationsaufbau wird im Konstruktor des IPC-Modul aus dem File “/tmp/hamcast/meeting_point/middleware/middleware.config_file“ der Port des Middlewaredienst ausgelesen und anschließend an den Konstruktor der Klasse MiddlewareStream übergeben. Diese Klasse enthält die nativen Lese- und Schreiboperation für die Kommunikation mit der Middleware. Im Konstruktor wird mit dem Port und der Middlewareadresse, in unserem Falle localhost, ein TCP-Channel eröffnet. Für die Authentifizierung und die Absicherung, dass kompatible Versionen API und H∀Mcast eine Kommunikation aufbauen wollen, wird zuerst eine Magic Number und im Anschluss eine Major und Minor Nummer an die Middleware geschickt. Mit

einer 0 bestätigt die Middleware die Authentifizierung und eine kompatible Programmversion. SO_SNDBUF, SO_RCVBUF und TCP_NODELAY aus der Klasse Parameter werden ebenfalls im Konstruktor abgerufen und setzen die Größe des Empfangs und Sendebuffer und aktivieren, bzw. deaktivieren TCP_NODELAY im TCP/IP Stack. Alle drei Parameter sind als public static deklariert und ermöglichen dem Entwickler einen Eingriff in die Performance der IPC Kommunikation. Sinnvoll könnte zum Beispiel für einen Entwickler das Aktivieren von TCPNODELAY sein, damit große Pakete direkt verschickt werden und nicht über den Nagel-Algorithmus gebündelt werden. Die beste Auslastung des IPC-Channel bekommt der Entwickler, wenn er seine Pakete auf die Größe TCP_MAXSEG-16Byte (IPC Nachrichten Header) fragmentiert und TCPNODELY aktiviert. Für kleine Pakete wird das Deaktivieren von TCP NODelay empfohlen, da hier der Nagel-Algorithmus für eine gute Auslastung auf dem TCP-Channel sorgt. Für die Schreibe- und Leseoperationen auf dem Channel stellt die Klasse Middlewarstream read und write Methoden bereit. Die Methoden sind als protected deklariert und kapseln diese so vor möglichen Zugriffen aus der Anwendungsebene.

Listing 4.1: static initializer Ipc

```
protected static Ipc ipc = null;
   static {
       try {
           ipc = new Ipc("localhost");
       } catch (IOException e) {
           if (Debugging.ACTIVE) {
               Debugging.log(Debugging.FATAL, "Middleware connection refused");
               Debugging.log(Debugging.FATAL, e.toString());
            }
        }
 }
```

Listing 4.2: MiddlewareStream

```
public MiddlewareStream(String adress, int port, long magicNumber,long majorNumber, long minorNumber)
      throws IOException {
middlewareSocket = new Socket(adress, port);
middlewareSocket.setReceiveBufferSize(Parameter.SO_RCVBUF);
if (Debugging.ACTIVE) {
  Debugging.log(Log.INFO, "TCP Receivebuffer: "+middlewareSocket.getReceiveBufferSize());
}
middlewareSocket.setSendBufferSize(Parameter.SO_SNDBUF);
if (Debugging.ACTIVE) {
  Debugging.log(Log.INFO, "TCP Sendbuffer: "+middlewareSocket.getSendBufferSize());
}
middlewareSocket.setTcpNoDelay(Parameter.TCP_NODELAY);
if (Debugging.ACTIVE) {
  Debugging.log(Log.INFO, "TCP NoDelay:"+Parameter.TCP_NODELAY);
}
this.out = middlewareSocket.getOutputStream();
this.in = middlewareSocket.getInputStream();

write(Marshalling.uint32ToByteArray(Parameter.m_magic_number));
write(Marshalling.uint32ToByteArray(Parameter.m_major_version));
write(Marshalling.uint32ToByteArray(Parameter.m_minor_version));
if (read() != 1) {
```

```
22    if (Debugging.ACTIVE) {
23      Debugging.log(Log.FATAL, "Your API-Version ist incompatible with Hamcast");
24    }
25    throw new IOException("incompatible Api-Version");
26  }
27  }
```

4.2. Multicast Socketojekt erstellen

Ein MSocketHamCast Objekt wird unter Java über den new Operator erzeugt. Der Konstruktor ruft über das Ipc-Modul eine entsprechende Methode der Proxyklasse auf. Diese ermöglicht durch RMI einen Methodenaufruf von createSocket() in der Middleware. Jedes Socketobjekt wird über eine ID in der Middleware referenziert und der entfernte Methodenaufruf createSocket() liefert diese ID zurück. Die gelieferte SocketID wird allerdings im MSocketHamCast Objekt als private deklariert, um IPC-Informationen von der Anwendung zu kapseln. Methodenaufrufe des entfernten Objekts geschehen nun über das MSocketHamCast Objekt, beim RMI Verfahren spricht man auch von sogenannten Stub-Objekten. Jedes Socket-Objekt enthält eine ConcurrentHashMap<String,Sendbuf>() , die für jeden Sendstream einen Buffer erstellt. Als Schlüsselwert wird die URI in Stringrepräsentation gewählt. Auf IPC Ebene wird allerdings ein SendStream nicht über eine URI identifiziert, sondern über eine StreamID, die in Kombination mit einer SocketID eindeutig ist. Für ein schnelles Mapping von einer StreamID auf einen SendBuf erzeugt der Konstruktor eine weitere ConcurrentHashMap<Integer, Sendbuf>(), die beispielhaft von der Methode für die Bestätigung von Paketen genutzt wird. Ebenfalls wird für den Empfang von Multicastnachrichten eine ConcurrentHashMap<String, ConcurrentLinkedQueue<byte[]» initialisiert. ASYN_RCVBUF aus der Parameterklasse enthält die Größe des Empfangsbuffers default:16MB. Diese Variable ist als public deklariert und ermöglicht so dem Anwendungsbenutzer den Zugriff auf diese Parameter. Als letztes wird die Socketreferenz (this) über eine globale Liste eingetragen, damit das IPC Modul auf Sende- und Empfangsbuffer zugreifen kann.

Listing 4.3: new MSocketHamCast()

```
1   private int socketID;...
2
3   public MSocketHamCast() throws IOException {
4
5       // Entfernter Methodenaufruf
6       socketID = ipc.proxy.createSocket();
7
8       socksSendMap = new ConcurrentHashMap<String, Sendbuf>();
9       // für ein schnelles Mapping beim Empfangen eines Ack
10      socksSendMap_StreamID = new ConcurrentHashMap<Integer, Sendbuf>();
11
12      receiveBufferSize=Parameter.ASYN_RCVBUF;
13      receiveBuffer = new ConcurrentHashMap<String, ConcurrentLinkedQueue<byte[]>>();
```

```
14
15      MSocketHamCast.socketList.put(socketID, this);
16
17
18  }
```

4.3. Der Proxy des IPC Moduls

Alle Methoden der Klasse MSocketHamCast nutzen den Proxy, um einen entfernten Methodenaufruf transparent einzuleiten. Innerhalb des Proxys werden durch Marshallingverfahren synchrone IPC Nachrichten erstellt und versendet (siehe Listing 4.4 u. 4.5). Der entfernte Methodenaufruf über die Methode sync_request() blockie rt, bis die entsprechende Antwort auf die Anfrage als synchrone IPC-Nachricht empfangen wird. Die Daten der Antwort werden durch Un-Marshalling-Verfahren interpretiert und geben entsprechende Parameter an die Methode im MSocketHamCast Objekt zurück.

Listing 4.4: Proxy.createStream()

```
1  protected synchronized int createStream(int socketID, String uri) throws IOException {
2    int sendStream;
3    Message msg = sync_request(IDL.FID_CREATE_SEND_STREAM, Marshalling.serialize(socketID, uri.toString
         ()));
4    if (msg.field1 != 0) {
5      checkErr(msg.field1, "createStream()", UnMarshalling.exception(msg.content));
6    }
7    sendStream = UnMarshalling.desCreateStream(msg.content);
8    if (Debugging.ACTIVE) {
9      Debugging.log(Debugging.INFO, new String("Proxy::createStream()::Succesfull␣create␣SendStream␣" +
           sendStream + "␣on␣Socket␣" + socketID
10           + "␣for␣Uri" + uri));
11   }
12   return (int) sendStream;
13 }
```

Listing 4.5: Proxy.sync_request()

```
1  private Message sync_request(int field1, byte[] data) {
2  comModul.ipcSyncBuffer.addRequest(new Message(IDL.SYNC_REQUEST, field1, requestID++, 0, data));
3    Message syn_msg;
4    do {
5      syn_msg = comModul.ipcSyncBuffer.getResponse();
6      if (syn_msg == null)
7        Thread.yield();
8     } while (syn_msg == null);
9   return syn_msg;
10 }
```

Listing 4.6: Java IDL

```
1  package libHamCast;
2  public class IDL {
```

```
3
4     // FunctionID
5     protected static final int FID_CREATE_SOCKET = 0x0001;
6     protected static final int FID_DELETE_SOCKET = 0x0002;
7     protected static final int FID_CREATE_SEND_STREAM = 0x0003;
8     protected static final int FID_JOIN = 0x0004;
9     protected static final int FID_LEAVE = 0x0005;
10    protected static final int FID_SET_TTL = 0x0006;
11    protected static final int FID_GET_SOCK_INTERFACE = 0x0007;
12    protected static final int FID_ADD_SOCK_INTERFACE = 0x0008;
13    protected static final int FID_DEL_SOCK_INTERFACE = 0x0009;
14
15    // Service Calls
16    protected static final int FID_GET_INTERFACE = 0x0100;
17    protected static final int FID_GROUP_SET = 0x0101;
18    protected static final int FID_NEIGHBOR_SET = 0x0102;
19    protected static final int FID_PARENT_SET = 0x0103;
20    protected static final int FID_CHILDREN_SET = 0x0104;
21    protected static final int FID_DESIGNATED_HOST = 0x0105;
22
23    // IPC MessageID
24    protected static final int SYNC_REQUEST = 0x00;
25    protected static final int SYNC_RESPONSE = 0x01;
26    protected static final int ASYNC_EVENT = 0x02;
27    protected static final int ASYNC_SEND = 0x03;
28    protected static final int ASYNC_RECV = 0x04;
29    protected static final int CUMULATIVE_ACK = 0x05;
30    protected static final int RETRANSMIT = 0x06;
31
32    //Exeptions
33    protected static final int EID_NONE = 0x0001;
34    protected static final int EID_REQUIREMENT_FAILED = 0x0002;
35    protected static final int EID_INTERNAL_INTERFACE_ERROR = 0x003;
36 }
```

4.4. Marshalling und Un-Marshalling in H∀Mcast

Um Daten erfolgreich maschinen- und programmiersprachenunabhängig auszutauschen, müssen die Daten in ein separates, externes Datenformat gebracht werden. Ein Verpacken der Daten bezeichnet man mit Marshalling ein entpacken als Un-Marshalling. Elementare Datentypen können auf Maschinen unterschiedlich dargestellt werden, so werden Zeichensätze auf Unixbasierten Betriebssystemen im ASCII Zeichensatz gespeichert, 1Byte pro Zeichen, auf Dos basierten Betriebssystemen im Unicode-Standard, der 2Byte pro Zeichen benötigt. Zahlendarstellungen werden in x86 Systemen nach Little-Endian gespeichert, das niederwertigste Byte an die kleinsten Speicheradresse , im PowerPC hingegen wird die Datendarstellung des Big-Endian-Formates genutzt. Der Middlewaredienst H∀Mcast und das Gruppenkommunikationsprogramm laufen im Prototypen auf der gleichen Maschine und eine explizite Berücksichtigung der aufgeführten Hardwareabhängigkeitsprobleme wäre nicht

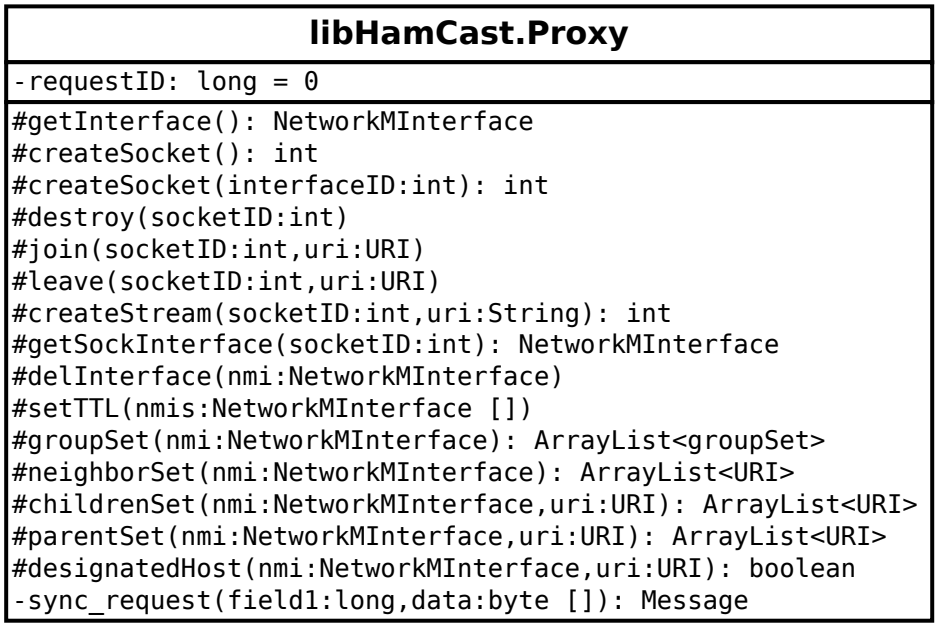

Abbildung 4.2.: Klassendiagramm libHamCast.Proxy

C++	Marshalling Unmarshalling	Java
unsigned int	4 byte(Big-Endian)	long
unsigned short	2 byte(Big-Endian)	int
char [x]	[Länge]4 Byte(Big-Endian) + x byte Ascii Codiert	String
struct	[Anzahl Structelemente]4 byte(little Endian) Marshalling der Elemente nach Deklarationreinfolge	Objekt

Tabelle 4.1.: Marshalling/Unmarshalling

von Nöten. Vorstellbar wäre allerdings für die Zukunft, die Middleware in einen kleinen NAT[1] Router zu integrieren, der für ein Heimnetzwerk den Dienst fürs hybride Multicast bereitstellt. Für diese Infrastruktur ist eine hardwareunabhängige externe Datendarstellung unabdingbar. In H∀Mcast werden alle Daten in Network-Byte-Order (Big-Endian), auf den TCP-Stream geschrieben (siehe Listing 4.7). Ein weiteres Problem, welches die externe Datendarstellung berücksichtigen muss, sind die unterschiedlichen Datentypen in den Programmiersprachen. Ein unsigned Int in C++, mit dem Wert 2^{32}, wird beim Empfang unter Java den Wert -1 ergeben, da Java keine unsigned Interpretation von Integern Werten zulässt. Beim Un-Marshalling-Verfahren muss dieses für die genutzte Programmiersprache berücksichtigt werden, um Zahlen im Grenzbereich korrekt darzustellen. (siehe Tabelle 4.1).

[1] network address translation

Listing 4.7: Marshalling.convertArrayToLong()

```
1  public static int unsignedByteToInt(byte b) {
2          return (int) b & 0xFF;
3      }
4
5  public static int convertArrayToLong(byte[] b, int start, int size) {
6          int value = 0;
7          for (int i = start; i < start + size; i++) {
8              //Network Byte Order
9              value += unsignedByteToInt(b[i]) << (8 * (i - start));
10
11         }
12         return value;
13 }
```

4.5. Senden mit MSocketHamCast

Senden von Multicastdaten wird über asynchrone IPC-Nachrichten realisiert (siehe Kapitel 2.4.5.2). Beim ersten send() Methodenaufruf für eine Uri wird implizit ein Socketstream in der Middleware über eine synchrone Ipc-Nachricht angefordert. Bei erfolgreichem Methodenaufruf wird ein Objekt erstellt, dessen Methoden die Daten für den Versand vorbereiten und in einer lokalen History zwischenspeichern. Die Referenz des Objekts wird in zwei Mappingdatenstrukturen hinterlegt, um auf einfache Weise den Zugriff über Uri oder Stream ID zu gewähren. Die add() Funktion von Sendbuf versucht die Daten in den lokalen Buffer abzulegen und informiert über den Rückgabeparameter. Wenn kein Ablegen der Daten möglich ist, da der Sendbuffer im IPC Modul gefüllt ist, legt sich der Thread schlafen . Acknowlege-Nachrichten, die dafür sorgen, dass die lokale History in regelmäßigen Abständen gelöscht wird, informieren gleichzeitig den Thread über freien Platz im Sendbuf. Die Größe des Sendbuffer bestimmt die statische Varaible ASYN_SNDBUF aus der Klasse Parameter. Sie besitzt getter und setter Methoden kann somit von den Anwendungssoftware gesteuert werden.

Listing 4.8: MSocketHamCast send()

```
1  public void send(URI uri, byte[] data) throws IOException {
2
3    String uri_s = uri.toString();
4
5    if (!socksSendMap.containsKey(uri_s)) {
6
7      int streamID =ipc.proxy.createStream(socketID, uri_s);
8
9      Sendbuf sendbuf = new Sendbuf(uri_s, streamID, socketID, this);
10
11     //mapping for sendoperation
12     socksSendMap.put(uri_s, sendbuf);
13
14     //fast mapping over streamid
15     socksSendMap_StreamID.put(streamID, sendbuf);
16   }
```

```
17
18    Sendbuf sendbuf = socksSendMap.get(uri_s);
19
20    while(!sendbuf.add(data)) {
21      // kein Senden möglich! warten bis der Ack Thread ein signal gibt
22      try {
23        sendlock.lock();
24          sendCondition.await();
25        sendlock.unlock();
26      } catch (InterruptedException e) {e.printStackTrace();}
27    }
28  }
```

4.6. Empfangen mit MSocketHamCast

Empfang von Multicastnachrichten geschieht wie das Senden dieser Nachrichten über asynchrone IPC-Nachrichten. In Kapitel 3.4.6.3 wurde bereits erwähnt, dass eine receive Methode implizit einen join Methodenaufruf enthält, soweit dieser im Vorwege nicht bereits geschehen ist. Wenn noch keine Daten für diese URI eingetroffen sind, legt sich der Empfänger schlafen und wird beim Eintreffen von Daten aus dem IPC Modul über diese informiert, ein klassisches Producer-Consumer Konzept. Das IPC Modul enthält einen Thread, der für das Empfangen von Nachrichten aus der Middleware zuständig ist. Dieser Thread liest kontinuierlich vom TCP-Stream und parst im Anschluss die gelesenen Daten, um die nötigen Operationen für die verschieden IPC Nachrichten durchzuführen.

Listing 4.9: MSocketHamCast send()

```
1   public byte[] receive(URI uri) throws IOException {
2
3     byte[] receive;
4
5     if (!receiveBuffer.containsKey(uri.toString())) {
6       join(uri);
7     }
8     do {
9       receive = receiveBuffer.get(uri.toString()).poll();
10      // Keine Packet zu empfangen
11      if (receive == null) {
12        try {
13          // consumer wait for producer
14            receivelock.lock();
15            receiveCondition.await();
16            receivelock.unlock();
17          } catch (InterruptedException e) {e.printStackTrace();}
18        } else {
19          //atomar operation
20          writeLock.lock();
21          receiveBufferSize+= receive.length;
22          writeLock.unlock();
23        }
24      } while (receive == null);
```

```
25
26    return receive;
27    }
```

4.7. Debugging

Informationen aus der API können bei der Entwicklung von Gruppensoftware entscheidend zum Erfolg beitragen, die Codeentwicklungszeit minimieren und damit Kosten senken. Die konkrete Realisierung der Java API enthält in ihrem Paket libHamCast eine zusätzliche Debugging-Klasse, die für die Ausgabe von Trace-Informationen zuständig ist. Hierbei unterscheidet die Klasse 6 Stufen, die sich in Fatal, Error, Warn, Info, Debug und Trace unterteilen. Fatal ist die niedrigste Loggingstufe und gibt nur Informationen aus, die unabdingbar zum Programmabsturz führen. Jede höhere Stufe gibt neben den neuen Informationen auch alle Informationen unterliegender Stufen aus. Das Debuglevel kann über die statische Variable debuglevel aus der Klasse Debugging gesetzt werden. Ebenfalls enthält diese Klasse die beiden boolschen Variablen printConsole und printFile, die die Ausgabe auf den verschiedenen Medien steuern.

Listing 4.10: Debugging()

```
1  public static final int TRACE = 5;
2  public static final int DEBUG = 4;
3  public static final int INFO = 3;
4  public static final int WARN = 2;
5  public static final int ERROR = 1;
6  public static final int FATAL = 0;
7  //Variable zum Einstellen des Debuglevel's
8  public static int debugLevel = FATAL;
```

Umfangreiche Debugfunktionen können allerdings die API in ihrer Geschwindigkeit bremsen und den Java Bytecode unnötig aufblähen. Eine Release-Software sollte demnach möglichst eine API ohne Debugfunktionalität nutzen. Unter C gibt es die Möglichkeit, mit ifdef den Inhalt, der kompiliert werden soll, über den Präprozessor zu selektieren (ver. Wolf (2005)). Java besitzt keine ifdef, allerdings können durch finale Boolsche Variablen ähnliche Codeeinsparungen geschaffen werden. Eine if-Anweisung, die zur Kompilierungszeit false beträgt und durch final Deklaration ihren Wert nicht nachträglich ändern kann, wird vom Compiler nicht verarbeitet (ver. Darwin (2004)). ACTIVE ist als statisch protected final in der Kasse Debugging deklariert und ermöglicht über seine boolschen Parameter True/False die Debugfunktionalität beim Kompilieren ein- oder auszuschließen.

Listing 4.11: ifdef unter Java

```
1 if (Debugging.ACTIVE) {
2       Debugging.log(Debugging.INFO, new String(("Proxy::createSocket(int)::succesfull_socketID:" +
            socketID)));
3 }
```

1. TRACE: Enthält alle Logging Informationen
2. DEBUG: Informiert unter anderem über jedes Empfangende und versandte Packet
3. INFO: Informiert über alle Methodenaufrufe
4. WARN: Informiert über Warnung, z.B. wenn ein Buffer voll ist.
5. ERROR: Informiert über Fehler, die nicht zum Programmabbruch führen.
6. FATAL: Informiert über Fehler, die zum Programmabbruch führen.

5. Test und Evaluierung

5.1. Evaluierung

Um die Leistung der Java-API in Kombination mit dem hybriden Middlewaredienst H∀Mcast zu evaluieren, wurden einige allgemeine Performancetests der prototypischen Implementierung untersucht. Als grundlegende Metrik werden in den Diagrammen 5.1 u. 5.2 der Paketdurchsatz pro Sekunde ausgegeben. Hierbei wurden verschiedene Paketgrößen getestet und mit der bereits implementierten C++ API verglichen. Für den Versuchsaufbau wurden ähnliche Rechner gewählt, die über einen 1Gbit Switch der Firma Linksys SRW-2016 verbunden sind. Als Sender wurde ein Dell Optiplex 986 Computer mit einem Intel i5(750) Prozessor (4Cores/2,67Mhz) und 8MB Arbeitsspeicher eingesetzt, als Empfänger ein Dell Optiplex 986 Intel i7(870) Prozessor (4Cores/2,93GHz) mit 8MB Arbeitsspeicher. Ein eigens unter Java entwickeltes Benchmark-Tool (siehe Listing Appendix A.3) erlaubt die Betrachtung des Paketdurchsatzes bei unterschiedlichster Paketgröße. Um Störungen in Testergebnissen durch Starten von gleich oder höher priorisierten Prozessen im Linuxkernel zu minimieren, wurde ein Skript geschrieben, welches jede Paketgröße 50 mal 60 Sekunden testet und somit externe Fehlereinflüsse minimiert. Als hybrider Middlewaredienst wurde H∀Mcast in der Version 4.0 verwandt und als Empfänger und Sender die Java Implementation der API. Denkbar wäre auch, in Zukunft Kombinationen von C++ und Java zu untersuchen.

5.2. Analyse und Bewertung

Im Diagramm 5.1 ist der Paketdurchsatz der C++ und der Java API dargestellt, die beide den Middlewaredienst H∀Mcast nutzen. Um die Daten besser interpretieren zu können, wurden auch Ergebnisse eines nativen IP-Multicast, also ohne Middlewaredienst H∀Mcast untersucht und ausgegeben. Als Messreihe dienen Paketgrößen von 100-1500 Byte. Um die Bruttoleistung auf dem Netzwerk auswerten zu können, wurde im Benchmarktool die getestete Packetgröße-28 Byte gewählt (Nettogröße). 28Byte Steuerdaten bestehend aus 20Byte ipv4 Header und 8 Byte UDP Header ist der Overhead des Layer 3 und 4. Bei kleineren Paketen unterliegt H∀Mcast deutlich dem IP-Multicast (siehe Diagramm 5.1 u. 5.2) und nutzt die Bandbreite des Netzwerkes längst nicht aus. Der Java API Sender unterliegt ebenfalls

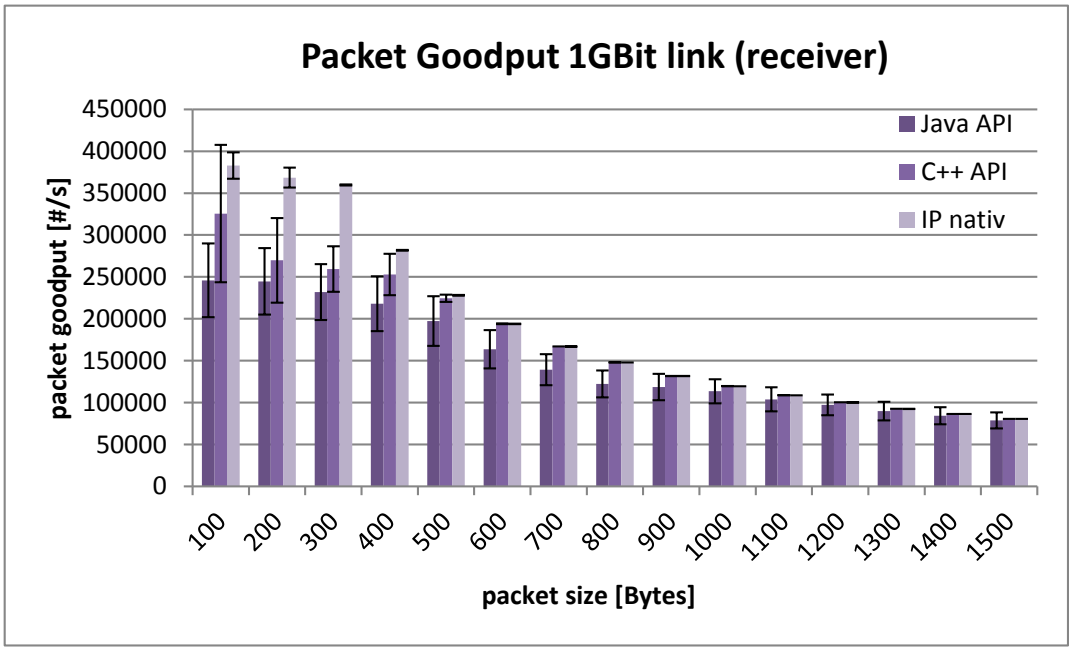

Abbildung 5.1.: Packet Goodput 1GB Receiver

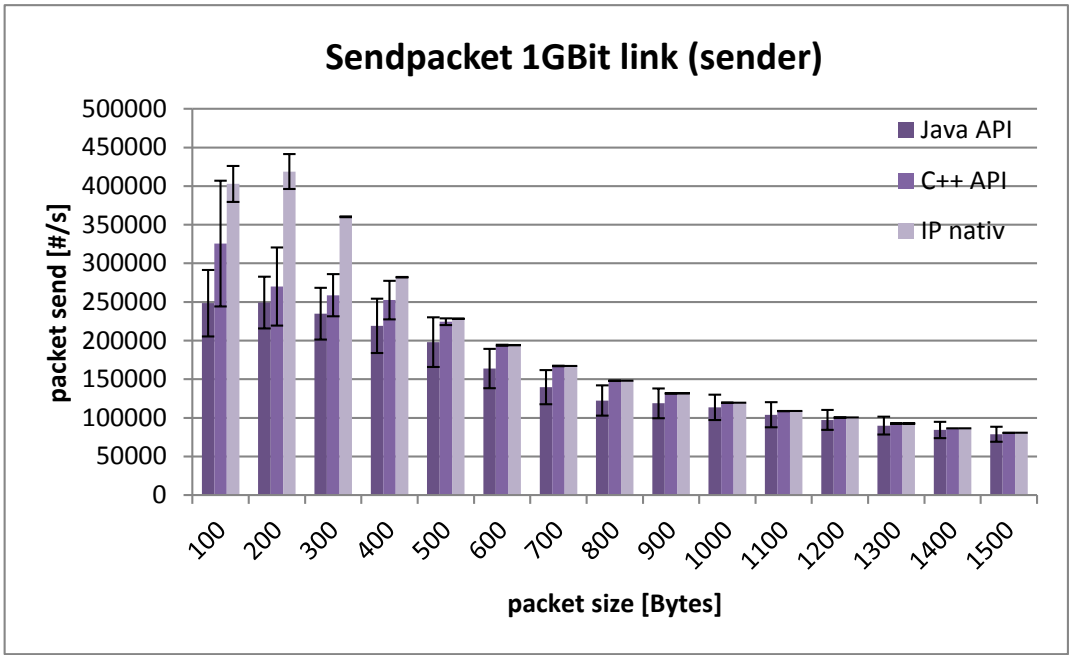

Abbildung 5.2.: Packet flow 1GB Sender

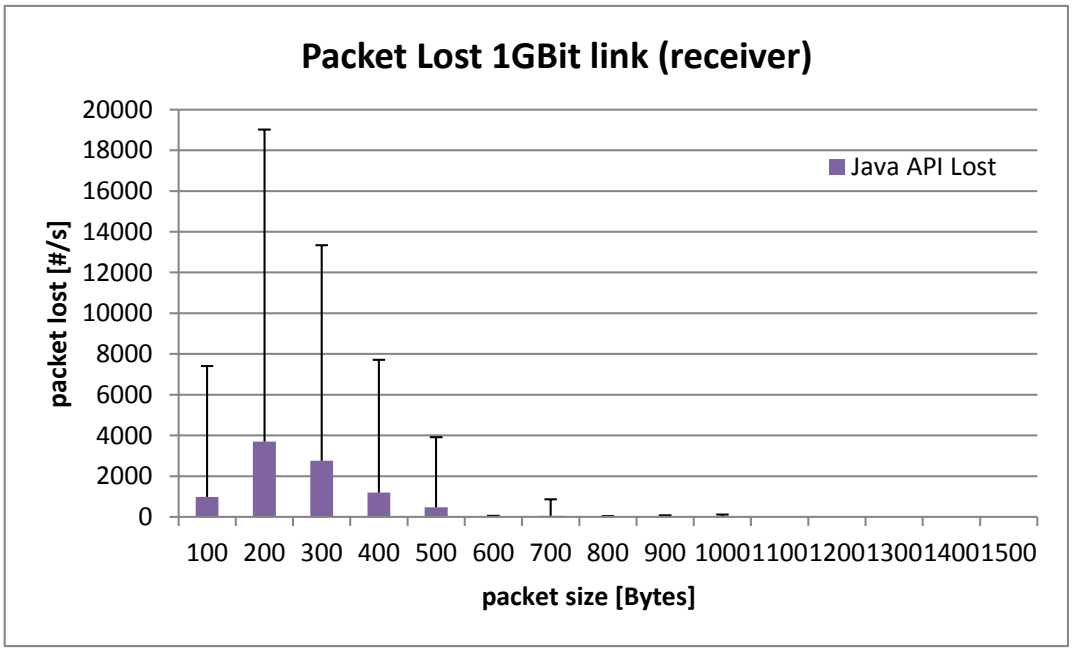

Abbildung 5.3.: Packet lost 1GB Receiver

in niedrigen Paketgrößen der C++ API und eine systematische Untersuchung und Verbesserung sollte ggf. Abhilfe schaffen. Ab einer Paketgröße von 1000Byte haben Java API und C++ API etwa gleiche Ergebnisse und nähern sich beide den Werten des nativen IP. Bei dieser Paketgröße werden auch erstmalig die 900MBit überschritten, 1000Byte x 97329 Pakete/S ≙ 900MBit und es kann von einer guten Netzwerkauslastung gesprochen werden. Zur Entwicklungszeit traten bei niedrigen Paketgrößen für kurze Zeit unregelmäßig große Paketverluste auf Seiten des Empfängers auf (siehe Diagramm 5.1). Innerhalb der Arbeit konnte eine vollständige systematische Analyse dieses Fehlers nicht durchgeführt werden. Erste Untersuchungen haben allerdings gezeigt, dass dieses Problem nur bei kleinen Paketen <700 Byte auftritt. Der große Paketverlust basiert auf einem Überlauf des SO_RCVBUF, der nicht ausreichend schnell von der Java API ausgelesen wird. Die unregelmäßigen Paketverluste wurden durch Einsatz von deutlich schnelleren Receiver-Computern oder durchs downgraden des Netzwerks auf 100MBit vollständig behoben und somit wird ausgeschlossen, dass es sich um einen Bug im Code handelt. Es wird empfohlen, den Garbage Collector auf Receiverseite zu untersuchen und die Auslesegeschwindigkeit des SO_RCVBUF zu analysieren und ggf. zu optimieren. Auch eine Untersuchung des Prozessscheduling könnte Aufschluss über Engpässe im Reciever geben.

5.3. Zusammenfassung und Ausblick

In dieser Arbeit haben wir neben einer kurzen Einführung in Multicastgrundlagen die Common Api for hybrid Multicast des Internet Draft von Matthias Wählisch analysiert und die Konzeptionierung in der Programmiersprache Java diskutiert. Des weiteren wurde ein Prototyp dieser API entwickelt und an den hybriden Multicast Dienst H∀Mcast gebunden. Eine kleine Evaluierung am Ende dieser Arbeit rundet die Entwicklung des Java Prototypen ab, lässt allerdings noch Fragen bezüglich desPacketlost offen.

H∀Mcast wurde bis zum Abschluss dieser Arbeit noch keiner Gesamtperformance unter Berücksichtigung komplexer hyprider Anwendungsszenarien und Netzwerktopologien unterzogen. Ein Test in naher Zukunft im G-LAB Testbeds[1] ist geplant und wird weitere Ergebnisse liefern. Auch sind weitere Module geplant, um den Prototypen H∀Mcast zu erweitern. Eine systematische Analyse der Javatests, vorstellbar auch als Java Empfänger und C++ Sender, werden weitere Aufschlüsse über Paketverlust in Extremsituationen geben.

[1] www.german-lab.de

Literaturverzeichnis

[Albanna u. a.] ALBANNA, Z. ; ALMEROTH, K. ; MEYER, D. ; SCHIPPER, M.: IANA Guidelines for IPv4 Multicast Address Assignments / IETF (3171). – RFC

[B.Cain 2002] B.CAIN: Internet Group Management Protocol, Version 3 / Network Working Group. URL http://tools.ietf.org/html/rfc3376, Oktober 2002 (3376). – RFC. – 53 S

[Berners-Lee 2006] BERNERS-LEE, T.: Uniform Resource Identifier (URI): Generic Syntax / Network Working Group. URL http://tools.ietf.org/html/rfc4395, Februar 2006 (3986). – RFC. – 61 S

[BMU und Bitkom 2008] BMU ; BITKOM: Energieverbrauch in Rechenzentren senken. In: *16. Legislaturperiode* 08 (2008), Nr. 167. – URL http://www.bitkom.org/60376.aspx?url=Energieeinsparpotenziale_von_Rechenzentren_in_Deutschland.pdf

[Charousset 2011] CHAROUSSET, D.: *Hamcast Interprocess Communication*. Januar 2011. – URL http://www.realmv6.org/hamcast_dev_libhamcast_doxygen.html

[Conta und Deering 2006] CONTA, A. ; DEERING, S.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification / Network Working Group. URL http://tools.ietf.org/html/rfc4443, März 2006 (4443). – RFC

[Coulouris 2002] COULOURIS, George: *Verteilte Systeme Konzepte und Design*. Addison-Wesley, 2002. – ISBN 3-8273-7022-1

[Darwin 2004] DARWIN, I. F.: *Java Cookbook Second Edition*. O Reilly Media, 2004. – ISBN 0-596-00701-9

[Deering 1989] DEERING, S.: Host extensions for IP multicasting / IETF. URL http://www.ietf.org/rfc/rfc1112.txt, August 1989 (1112). – RFC

[Deering und Hinden 1998] DEERING, S. ; HINDEN, R.: Internet Protocol, Version 6 (IPv6) / Network Working Group. URL http://tools.ietf.org/html/rfc2460, Dezember 1998 (2460). – RFC. – 39 S

[Diot u. a. 2000] DIOT, Christophe ; LEVINE, Brian N. ; LYLES, Bryn ; KASSEM, Hassan ; BALENSIEFEN, Doug: Deployment Issues for the IP Multicast Service and Architecture. In: *IEEE Network Magazine* 14 (2000), Nr. 1, S. 78–88

[Gilligan 2003] GILLIGAN, R.: Basic Socket Interface Extensions for IPv6 / Network Working Group. URL http://tools.ietf.org/html/rfc3493, Februar 2003 (3493). – RFC. – 39 S

[Hansen u. a. 2005] HANSEN, T. ; DEERING, S. ; KOUVELAS, I. ; FENNER, B. ; THYAGARAJAN, A.: Guidelines and Registration Procedures for New URI Schemes / Network Working Group. URL http://www.ietf.org/rfc/rfc3986.txt, Januar 2005 (4395). – RFC. – 13 S

[Hinden und Deering 2006] HINDEN, R. ; DEERING, S.: IP Version 6 Addressing Architecture / IETF. URL http://www.ietf.org/rfc/rfc4291.txt, Februar 2006 (4291). – RFC

[J.D.Day und Zimmermann 1983] J.D.DAY ; ZIMMERMANN, H.: *The OSI reference model*. 1983. – URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1457043

[Jin u. a. 2009] JIN, Xing ; CHENG, Kan-Leung ; CHAN, S.-H. G.: Island multicast: combining IP multicast with overlay data distribution. In: *Trans. Multi.* 11 (2009), Nr. 5, S. 1024–1036. – ISSN 1520-9210

[Kecher 2009] KECHER, Christoph: *UML 2*. Galileo Computering, August 2009. – ISBN 978-3-8362-1419-3

[Krüger 2009] KRÜGER, Guido: *Handbuch der Java Programmierung*. Addison-Wesley, 2009. – ISBN 978-3-82732874-8

[Lu u. a. 2007] LU, Shaofei ; WANG, Jianxin ; YANG, Guanzhong ; GUO, Chao: SHM: Scalable and Backbone Topology-Aware Hybrid Multicast. In: *16th Intern. Conf. on Computer Communications and Networks (ICCCN'07)*, August 2007, S. 699–703. – ISSN 1095-2055

[Meiling u. a. 2010] MEILING, Sebastian ; CHAROUSSET, Dominik ; SCHMIDT, Thomas C. ; WÄHLISCH, Matthias: System-assisted Service Evolution for a Future Internet – The HAMcast Approach to Pervasive Multicast. In: *Proc. of IEEE GLOBECOM 2010, Workshop MCS 2010*. Piscataway, NJ, USA : IEEE Press, 2010

[M.Wählisch u. a. 2011] M.WÄHLISCH ; SCHMIDT, T.C. ; S.VENAAS: A Common API for Transparent Hybrid Multicast / SAM Research Group. URL http://tools.ietf.org/html/draft-irtf-samrg-common-api-01, März 2011. – DRAFT. – 31 S

[Postel 1980] POSTEL, J.: User Datagram Protocol. URL http://tools.ietf.org/rfc/rfc768.txt, Januar 1980 (768). – RFC. – 3 S

[Schmidt und Wählisch 2010] SCHMIDT, Thomas C. ; WÄHLISCH, Matthias: System-level Service Assurance – The HAMcast Approach to Global Multicast. In: *Proc. of 6th TridentCom* Bd. 46. Berlin Heidelberg : Springer-Verlag, 2010, S. 635–639

[Steinmetz und Wehrle 2005] STEINMETZ, R. ; WEHRLE, K.: *Peer to Peer Systeme and Applications*. Springer, 2005. – ISBN 978-3-540-29192-3

[Stevens u. a. 2009] STEVENS, W. R. ; FENNER, B. ; RUDOLFF, A.M.: *Network Programming The Sockets Networking API*. Addison-Wesley, November 2009. – ISBN 0-13-141155-1

[Tanenbaum 2003] TANENBAUM, A.S.: *Computernetzwerke*. Person Studium, 2003. – ISBN 3-8273-7046-9

[Ullenboom 2011] ULLENBOOM, C.: *Java ist auch eine Insel*. Galileo Computing, August 2011. – ISBN 978-3-8362-1506-0

[Vida und Costa 2004] VIDA, R. ; COSTA, L.: Multicast Listener Discovery Version 2 (MLDv2) for IPv6 / Network Working Group. URL http://tools.ietf.org/html/rfc3810, Juni 2004 (3810). – RFC. – 62 S

[Wählisch und Schmidt 2010] WÄHLISCH, Matthias ; SCHMIDT, Thomas C.: Hybrid Adaptive Mobile Multicast – The HAMcast Approach of Unified Access to Future Services. In: GUIGNER, Jean-Paul L. (Hrsg.) u. a.: *Proceedings of the TERENA Networking Conference (TNC 2010)*. Amsterdam : TERENA, Juni 2010. – Poster

[Wittmann und Zitterbart 2001] WITTMANN, R. ; ZITTERBART, M.: *Multicast Communication*. Morgan Kaufmann, August 2001. – ISBN 1-55860-645-9

[Wolf 2005] WOLF, Jürgen: *C von A bis Z*. Galileo Computing, 2005. – ISBN 3-89842-392-1

[Zagaria 2010] ZAGARIA, Sebastian: *Visualisierungs- und Monitoring-Software fuer eine hybride Multicast Architektur*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeiten, 2010. – URL http://inet.cpt.haw-hamburg.de/thesis/completed/sebastian_zagaria.pdf

[Zhang u. a. 2006] ZHANG, Beichuan ; WANG, Wenjie ; JAMIN, Sugih ; MASSEY, Daniel ; ZHANG, Lixia: Universal IP multicast delivery. In: *Computer Networks* 50 (2006), Nr. 6, S. 781–806. – ISSN 1389-1286

A. Quellcode

A.1. SimpleSender.java

Listing A.1: testing.SimpleSender.java

```
1  package testing;
2
3  import java.io.IOException;
4  import java.net.URI;
5  import java.net.URISyntaxException;
6  import libHamCast.Debugging;
7  import libHamCast.MSocketHamCast;
8
9  /**
10  * Ein einfacher Sender, der in einer Endlossschleife HalloWelt verschickt
11  *
12  * @author Patrick Wienecke
13  * @version 1.0
14  */
15
16 public class SimpleSender {
17
18         /**
19          *
20          * MSocketHamCastSender
21          *
22          * @param args
23          * @throws IOException
24          * @throws URISyntaxException
25          *
26          */
27         public static void main(String[] args) throws IOException, URISyntaxException  {
28
29                 //Debuginformation
30             Debugging.debugLevel = Debugging.TRACE;
31                 Debugging.printFile=true;
32
33                 //Multicastsocket erstellen
34                 MSocketHamCast mSocket=new MSocketHamCast();
35
36                 //Uri erstellen
37                 URI uri=new URI("ip://239.0.0.1:1234");
38
39                 for (int i=0;true;i++) {
40
41                         String hello_world = "Hello World " + "Nr. "+i;
42
```

```
43                          mSocket.send(uri, hello_world.getBytes());
44                          System.out.println(hello_world);
45
46                  }
47
48
49
50          }
51
52  }
```

A.2. SimpleReceiver.java

Listing A.2: testing.SimpleReceiver.java

```
1   package testing;
2
3   import java.io.IOException;
4   import java.net.URI;
5   import java.net.URISyntaxException;
6   import java.util.Iterator;
7
8   import libHamCast.Debugging;
9   import libHamCast.MSocketHamCast;
10  import multicastApi.NetworkMInterface;
11
12  /**
13   * Ein einfacher Empfänger, der in einer Endlossschleife empfängt. Beim Starten
14   * ruft er alle geladenen Interfaces unter Hamcast ab!
15   *
16   * @author Patrick Wienecke
17   * @version 1.0
18   */
19
20  public class SimpleReceiver {
21
22      /**
23       *
24       * MSocketHamCastReceiver
25       *
26       * @param args
27       * @throws IOException
28       * @throws URISyntaxException
29       *
30       */
31
32      public static void main(String[] args) throws Exception {
33
34          // Setzt Debuglevel und Ausgabemedium
35          Debugging.debugLevel = Debugging.TRACE;
36          Debugging.printFile = true;
37          Debugging.printConsole = true;
38
39          // Ruft Interfaces unter HamCast ab
40          Iterator<NetworkMInterface> netMInter = MSocketHamCast.getAllInterfaces();
```

```
41
42          while (netMInter.hasNext()) {
43
44              NetworkMInterface nmi = netMInter.next();
45
46              System.out.println("index: " + nmi.getIndex());
47              System.out.println("load: " + nmi.getDisplayName());
48              System.out.println("adress: " + nmi.getInetAdress());
49              System.out.println("technology: " + nmi.getTech() + "\n");
50
51          }
52          // Erstellt ein Socketobjekt
53          MSocketHamCast mSocket = new MSocketHamCast();
54
55          URI uri = new URI("ip://239.0.0.1:1234");
56
57          //Join auf eine Uri(optional) da receive impliziet ein join durchführt
58          mSocket.join(uri);
59
60
61          while (true) {
62
63              System.out.println(new String(mSocket.receive(uri)));
64
65          }
66
67      }
68
69  }
```

A.3. Benchmark.java

Listing A.3: testing.Benchmark.java

```
1   package testing;
2
3   import java.io.IOException;
4   import java.net.URI;
5   import java.net.URISyntaxException;
6
7   import libHamCast.Debugging;
8   import libHamCast.MSocketHamCast;
9
10  /**
11   * Dieses Benchmarktool ermoeglicht detaillierte Sende und Empfangstest mit
12   * einer Instanz von libHamCast.MSocketHamCast durchzufuehren
13   *
14   * @author Patrick Wienecke
15   * @version 1.0
16   */
17
18  public class Benchmark {
19
20      /**
21       * Ein Byte was unter Java auf int gecasted wird und einen Wert über 127
```

```
22      * besitzt wird in einen negativen Wert umgerechnet(MSB=1). Diese Methode
23      * loest das Problem und ermoeglicht eine unsigned Interpretation.
24      *
25      * @param b
26      *            byte
27      * @return int
28      *
29      */
30     public static int unsignedByteToInt(byte b) {
31
32         return (int) b & 0xFF;
33     }
34
35     /**
36      * Rechnet eine Folge von bytes in einem Array in ein int um.
37      *
38      * @param b
39      *            bytearray
40      * @param start
41      *            start
42      * @param size
43      *            size
44      * @return int
45      *
46      */
47
48     public static int convertArrayToInt(byte[] b, int start, int size) {
49
50         int value = 0;
51
52         if (size <= 4) {
53             for (int i = start; i < start + size; i++) {
54                 value += unsignedByteToInt(b[i]) << (8 * (i - start));
55             }
56         }
57         return value;
58     }
59
60     public static byte[] uint32ToByteArray(long value) {
61
62         byte[] int_byte = new byte[4];
63         int_byte[0] = (byte) (value);
64         int_byte[1] = (byte) (value >>> 8);
65         int_byte[2] = (byte) (value >>> 16);
66         int_byte[3] = (byte) (value >>> 24);
67         return int_byte;
68     }
69
70     /**
71      *
72      * Hauptprogramm
73      *
74      * @param args
75      *            [] args
76      * @throws URISyntaxException
77      * @throws InterruptedException
78      * @throws IOException
79      * @throws Exception
80      *
81      *
```

```
82      **/
83
84      public static void main(String[] args) throws URISyntaxException,
85              InterruptedException, IOException {
86
87          Debugging.debugLevel = Debugging.FATAL;
88          final int ITERATIONTIME = 1000;
89
90          // Default Werte:
91          int ii = 0;
92          int packetsize = 0;
93          long id = 0;
94          int delay = 0;
95          long stopp = 0;
96          URI uri = new URI("ip://239.0.0.1:1234");
97          boolean start = true;
98          byte[] packet = new byte[1200];
99          boolean mode = true; // Empfänger oder Sender
100         boolean endless = true; // Endlossschleife
101
102         // Analysieren der Parameter
103         while (ii < args.length) {
104
105             String arg = args[ii++];
106
107             if (arg.equals("-b") || arg.equals("--byte")) {
108                 packetsize = new Integer(args[ii++]);
109
110             } else if (arg.equals("-h") || arg.equals("--help")) {
111                 System.out
112                         .println("-b|--byte␣packetsize[Byte]␣default=1200\n"
113                                 + "-i|--id␣first␣packet␣start␣with␣id␣default:0␣\n"
114                                 + "-d|--delay␣set␣the␣delay[ms]␣time␣between␣two␣packets␣default:0\n"
115                                 + "-c|--count␣␣count␣of␣sendpacket␣default:␣endless(0)\n"
116                                 + "-u|--uri␣set␣the␣listener␣adress␣default:ip://239.0.0.1:1234\n"
117                                 + "-l|--log␣set␣the␣apilog␣level␣0:fatal␣1:error␣2:warning␣3:info␣4:
                                        debug␣5:trace␣default:fatal␣\n"
118                                 + "-t|--time␣[ms]␣after␣Stop␣endless:0␣default:0␣\n"
119                                 + "-r|--receiver␣Receiver␣mode␣␣default:␣receiver␣\n"
120                                 + "-s|--sender␣Sendermode␣␣default:␣receiver␣\n");
121                 start = false;
122             } else if (arg.equals("-i") || arg.equals("--id")) {
123                 id = new Integer(args[ii++]);
124
125             } else if (arg.equals("-d") || arg.equals("--delay")) {
126                 delay = new Integer(args[ii++]);
127
128             } else if (arg.equals("-u") || arg.equals("--uri")) {
129                 uri = new URI(args[ii++]);
130
131             } else if (arg.equals("-l") || arg.equals("--log")) {
132                 // Debugginginformationen werden in in File geschrieben
133                 Debugging.printFile = true;
134                 Debugging.debugLevel = new Integer(args[ii++]);
135
136             } else if (arg.equals("-t") || arg.equals("--time")) {
137                 stopp = new Integer(args[ii++]);
138                 if (stopp == 0) {
139                     endless = true;
140                 } else {
```

```
141                     endless = false;
142                 }
143
144             } else if (arg.equals("-r") || arg.equals("--receiver")) {
145                 mode = true;
146
147             } else if (arg.equals("-s") || arg.equals("--sender")) {
148                 mode = false;
149             }
150         }
151         //Bei -h wird das Programm nicht gestartet
152         if (start) {
153
154             // Sendermodus
155             if (!mode) {
156
157
158                 long lost = 0;
159                 byte[] tmp = new byte[4];
160                 long cycleTime = 0;
161
162                 MSocketHamCast mSocket = new MSocketHamCast();
163                 System.out.println("#runtime[ms]\tpacket\tloss\tmemory");
164
165                 // speichert die Startzeit für Option -t
166                 long totaltime = System.currentTimeMillis() + stopp;
167
168                 while ((System.currentTimeMillis() < totaltime || endless)) {
169
170                     //speicher Startzeit für neuen Durchlauf
171                     long starttime = System.currentTimeMillis();
172                     long endtime = starttime + ITERATIONTIME;
173
174                     int i = 0; // Paketsendecounter
175
176                     for (i = 0; endtime > System.currentTimeMillis(); i++) {
177                         // Erzeugt das Paket
178                         packet = new byte[packetsize];
179                         tmp = uint32ToByteArray(id++);
180                         // Kopiert in die ersten 4 Bytes eine fortlaufende ID
181                         System.arraycopy(tmp, 0, packet, 0, 4);
182                         // sendet das Packet
183                         mSocket.send(uri, packet);
184                         // für spezille test, wenn der Sender zwischen den Packet schlafen soll
185                         Thread.sleep(delay);
186                     }
187                     //Berechnung der effektiven Sendezeit
188                     cycleTime += System.currentTimeMillis() - starttime;
189                     System.out.println(cycleTime + "\t" + i + "\t" + lost
190                             + "\t");
191
192                 }
193
194             } else {
195                 // Receiver Modus
196                 MSocketHamCast mSocket = new MSocketHamCast();
197                 System.out.println("runtime[ms]\tpacket\tloss");
198
199                 // Blockiert damit t erst mit dem ersten Packet anfängt zu zählen
200                 mSocket.receive(uri);
```

```
201
202                 byte[] tmp = new byte[4];
203
204                 long lastid = 0;
205                 long cycleTime = 0;
206                 long totaltime = System.currentTimeMillis() + stopp;
207
208                 while ((System.currentTimeMillis() < totaltime || endless)) {
209
210                     // Initialisierung
211                     long starttime = System.currentTimeMillis();
212                     long endtime = starttime + ITERATIONTIME;
213                     int i = 0;
214                     int lost = 0;
215
216                     for (i = 0; endtime > System.currentTimeMillis(); i++) {
217                         // Empfängt Packete von der Uri! Achtung kann blockieren!
218                         byte[] rPacket = mSocket.receive(uri);
219                         System.arraycopy(rPacket, 0, tmp, 0, 4);
220                         id = convertArrayToInt(tmp, 0, 4);
221                         rPacket = null;
222
223                         //Berechnung der verlorenen Pakete
224                         lost += id - (lastid + 1);
225                         lastid = id;
226                     }
227                     //Berechnung der effektiven Empfangszeit
228                     cycleTime += System.currentTimeMillis() - starttime;
229                     System.out.println(cycleTime + "\t" + i + "\t" + lost
230                             + "\t");
231
232                 }
233             }
234         }
235     }
236 }
```

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 4. Mai 2011	
Ort, Datum	Unterschrift