SAM Research Group
Internet-Draft
Intended status: Informational
Expires: January 17, 2013

M. Waehlisch
link-lab & FU Berlin
T C. Schmidt
HAW Hamburg
S. Venaas
cisco Systems

July 16, 2012

A Common API for Transparent Hybrid Multicast
draft-irtf-samrg-common-api-05

Abstract

Group communication services exist in a large variety of flavors, and technical implementations at different protocol layers. Multicast data distribution is most efficiently performed on the lowest available layer, but a heterogeneous deployment status of multicast technologies throughout the Internet requires an adaptive service binding at runtime. Today, it is difficult to write an application that runs everywhere and at the same time makes use of the most efficient multicast service available in the network. Facing robustness requirements, developers are frequently forced to use a stable, upper layer protocol provided by the application itself. This document describes a common multicast API that is suitable for transparent communication in underlay and overlay, and grants access to the different multicast flavors. It proposes an abstract naming by multicast URIs and discusses mapping mechanisms between different namespaces and distribution technologies. Additionally, it describes the application of this API for building gateways that interconnect current multicast domains throughout the Internet.

Status of this Memo

Table of Contents

1. Introduction

Currently, group application programmers need to make the choice of the distribution technology that the application will require at runtime. There is no common communication interface that abstracts multicast transmission and subscriptions from the deployment state at runtime, nor has been the use of DNS for group addresses established. The standard multicast socket options [RFC3493], [RFC3678] are bound to an IP version by not distinguishing between naming and addressing of multicast identifiers. Group communication, however, is commonly implemented in different flavors such as any source (ASM) vs. source specific multicast (SSM), on different layers (e.g., IP vs. application layer multicast), and may be based on different technologies on the same tier as with IPv4 vs. IPv6. It is the objective of this document to provide for programmers a universal access to group services.

Multicast application development should be decoupled of technological deployment throughout the infrastructure. It requires a common multicast API that offers calls to transmit and receive multicast data independent of the supporting layer and the underlying technological details. For inter-technology transmissions, a consistent view on multicast states is needed, as well. This document describes an abstract group communication API and core functions necessary for transparent operations. Specific implementation guidelines with respect to operating systems or programming languages are out of scope of this document.

In contrast to the standard multicast socket interface, the API introduced in this document abstracts naming from addressing. Using a multicast address in the current socket API predefines the corresponding routing layer. In this specification, the multicast name used for joining a group denotes an application layer data stream that is identified by a multicast URI, independent of its binding to a specific distribution technology. Such a group name can be mapped to variable routing identifiers.

The aim of this common API is twofold:

o Enable any application programmer to implement group-oriented data communication independent of the underlying delivery mechanisms. In particular, allow for a late binding of group applications to multicast technologies that makes applications efficient, but robust with respect to deployment aspects.

o Allow for a flexible namespace support in group addressing, and thereby separate naming and addressing resp. routing schemes from the application design. This abstraction does not only decouple

programs from specific aspects of underlying protocols, but may open application design to extend to specifically flavored group services.

Multicast technologies may be of various peer-to-peer kinds, IPv4 or IPv6 network layer multicast, or implemented by some other application service. Corresponding namespaces may be IP addresses or DNS naming, overlay hashes, or other application layer group identifiers like <sip:*@peanuts.org>, but also names independently defined by the applications. Common namespaces are introduced later in this document, but follow an open concept suitable for further extensions.

This document also discusses mapping mechanisms between different namespaces and forwarding technologies and proposes expressions of defaults for an intended binding. Additionally, the multicast API provides internal Interfaces to access current multicast states at the host. Multiple multicast protocols may run in parallel on a single host. These protocols may interact to provide a gateway function that bridges data between different domains. The application of this API at gateways operating between current multicast instances throughout the Internet is described, as well.

1.1. Use Cases for the Common API

The following generic use cases can be identified that require an abstract common API for multicast services:

Application Programming Independent of Technologies: Application programmers are provided with group primitives that remain independent of multicast technologies and their deployment in target domains. They are thus enabled to develop programs once that run in every deployment scenario. The use of Group Names in the form of abstract meta data types allows applications to remain namespace-agnostic in the sense that the resolution of namespaces and name-to-address mappings may be delegated to a system service at runtime. Thereby, the complexity is minimized as developers need not care about how data is distributed in groups, while the system service can take advantage of extended information of the network environment as acquired at startup.

Global Identification of Groups: Groups can be identified independent of technological instantiations and beyond deployment domains. Taking advantage of the abstract naming, an application is thus enabled to match data received from different Interface technologies (e.g., IPv4, IPv6, or overlays) to belong to the same group. This not only increases flexibility, an application may for instance combine heterogeneous multipath streams, but also

simplifies the design and implementation of gateways and translators.

Uniform Access to Multicast Flavors: The URI naming scheme uniformly supports different flavors of group communication such as any source and source specific multicast, selective broadcast etc., independent of their service instantiation. The traditional SSM model for instance can experience manifold support, either by directly mapping the multicast URI (i.e., "group@instantiation") to an (S,G) state on the IP layer, or by first resolving S for a subsequent group address query, or by transferring this process to any of the various source specific overlay schemes, or by delegating to a plain replication server. The application programmer can invoke any of these underlying mechanisms with the same line of code.

Simplified Service Deployment through Generic Gateways: The common multicast API allows for an implementation of abstract gateway functions with mappings to specific technologies residing at a system level. Such generic gateways may provide a simple bridging service and facilitate an inter-domain deployment of multicast.

Mobility-agnostic Group Communication: Group naming and management as foreseen in the common multicast API remain independent of locators. Naturally, applications stay unaware of any mobility-related address changes. Handover-initiated re-addressing is delegated to the mapping services at the system level and may be designed to smoothly interact with mobility management solutions provided at the network or transport layer (see [RFC5757] for mobility-related aspects).

1.2. Illustrative Examples

1.2.1. Support of Multiple Underlying Technologies

On a very high-level, the common multicast API provides the application programmer with one single interface to manage multicast content independent of the technology underneath. Considering the following simple example in Figure 1: A multicast source S is connected via IPv4 and IPv6. It distributes one piece of multicast content (e.g., a movie). Receivers are connected via IPv4/v6 and overlay multicast respectively.

```
    +-------+       +-------+               /         +-------+
    |   S   |       |  R1   |                        |  R3   |
    +-------+       +-------+                        +-------+
   v6|   v4|           |v4                              |OLM
     |     |          /                                 |
     |  ***| ***  ***/ **                           *** /***  ***  ***
      \*   |*   **  /**   *                        *   /*   **   **   *
      *\   \_______/_______*__v4__+-------+       *   /                *
       *\    IPv4/v6      *       |  R2   |__OLM__ *_/ Overlay Mcast  *
      *  \_________________*__v6__+-------+       *                    *
       *   **   **   **   *                        *    **   **   **  *
        ***  ***  ***  ***                          ***  ***  ***  ***
```

Figure 1: Source S sends the same multicast content to all interfaces

Using the current socket API, the application programmer needs to decide in advance on the IP technologies. Additional distribution techniques, such as overlay multicast, must be integrated specifically to the application. For each technology the application programmer needs to create a separate socket and initiates a dedicated join or send. As the current socket API does not distinguish between group name and group address, the content will be delivered multiple times to the same receiver (cf., R2). As the souce may distribute content via a technology that is not supported by the receivers or its Internet Service Provider (cf., R3) a gateway is required. To build gateway functions a consistent view on the multicast states at the gateway is important.

The common multicast API simplifies programming of multicast applications as it abstracts content distribution from specific technologies. In addition to calls which implement receiving and sending of multicast data, it provides service calls to grant access to internal multicast states at the host.

The API described in this document defines a minimal set of interfaces for the system components at the host to fulfill group communication. It is open to the implementation to provide additional convenience functions for the programmer.

The implementation of content distribution for the example shown in Figure 1 may then look like:

```
//Initialize multicast socket
MulticastSocket m = new MulticastSocket();
//Associate all available interfaces
m.addInterface(getInterfaces());
//Subscribe to multicast group
m.join(URI("opaque://news@cnn.com"));
//Send to multicast group
m.send(URI("opaque://news@cnn.com"),message);
```

Send/receive example using the common multicast API

The gateway function for R2 can be implemented by the service calls similar to:

```
//Initialize multicast socket
MulticastSocket m = new MulticastSocket();
//Check (a) host is designated multicast node for this interface
//      (b) receivers exist
for all this.getInterfaces() {
  if(designatedHost(this.interface) &&
       childrenSet(this.interface,
          URI("opaque://news@cnn.com")) != NULL) {
    m.addInterface(this.interface);
  }
}
while(true) {
  m.send(URI("opaque://news@cnn.com"),message);
}
```

Gateway example using the common multicast API

1.2.2. Support of Multi-Resolution Multicast

Multi-resolution multicast adjusts the multicast stream to consider heterogeneous end devices. The multicast data (e.g., available by different compression levels) are typically be announced under multiple multicast addresses, which are decoupled from each other. Using the common API multi-resolution multicast can be implemented from an operator- as well as subscriber-centric perspective by utilizing the Name-to-Address mapping.

Operator-Centric: An operator deploys a domain-specific mapping. In this case, any multicast receiver (e.g., mobile or DSL user) subscribes to the same multicast name, which will be resolved locally to different multicast addresses. Then, any Group Address describes a different level of data quality.

Subscriber-Centric: In a subscriber-centric example, the multicast receiver chooses the quality in advance based on a pre-defined naming syntax. Consider a layered video stream "blockbuster" available at different qualities Q_i, each of which consist of the base layer plus the sum of EL_j, j <= i enhancement layers. Each individual layer may then be accessible by a name "EL_j.Q_i.blockbuster", j <= i, while a specific quality aggregates the corresponding layers to "Q_i.blockbuster", and the full-size movie may be just called "blockbuster".

2. Terminology

This document uses the terminology as defined for the multicast protocols [RFC2710],[RFC3376],[RFC3810],[RFC4601],[RFC4604]. In addition, the following terms will be used.

Group Address: A Group Address is a routing identifier. It represents a technological specifier and thus reflects the distribution technology in use. Multicast packet forwarding is based on this address.

Group Name: A Group Name is an application identifier that is used by applications to manage communication in a multicast group (e.g., join/leave and send/receive). The Group Name does not predefine any distribution technologies, even if it syntactically corresponds to an address, but represents a logical identifier.

Multicast Namespace: A Multicast Namespace is a collection of designators (i.e., names or addresses) for groups that share a common syntax. Typical instances of namespaces are IPv4 or IPv6 multicast addresses, overlay group IDs, group names defined on the application layer (e.g., SIP or Email), or some human readable strings.

Interface: An Interface is a forwarding instance of a distribution technology on a given node. For example, the IP Interface 192.168.1.1 at an IPv4 host.

Multicast Domain: A Multicast Domain hosts nodes and routers of a common, single multicast forwarding technology and is bound to a single namespace.

Inter-domain Multicast Gateway (IMG): An Inter-domain Multicast Gateway (IMG) is an entity that interconnects different Multicast Domains. Its objective is to forward data between these domains, e.g., between an IP layer and overlay multicast.

3. Overview

3.1. Objectives and Reference Scenarios

The default use case addressed in this document targets at applications that participate in a group by using some common identifier taken from some common namespace. This Group Name is typically learned at runtime from user interaction like the selection of an IPTV channel, from dynamic session negotiations like in the Session Initiation Protocol (SIP), but may as well have been predefined for an application as a common Group Name. Technology-specific system functions then transparently map the Group Name to Group Addresses such that

o programmers are enabled to process group names in their programs without the need to consider technological mappings to designated deployments in target domains;

o applications are enabled to identify packets that belong to a logically named group, independent of the Interface technology used for sending and receiving packets. The latter shall also hold for multicast gateways.

This document considers two reference scenarios that cover the following hybrid deployment cases displayed in Figure 2:

1. Multicast Domains running the same multicast technology but remaining isolated, possibly only connected by network layer unicast.

2. Multicast Domains running different multicast technologies but hosting nodes that are members of the same multicast group.

```
                                        +-------+         +-------+
                                        | Member|         | Member|
                                        |  Foo  |         |   G   |
                                        +-------+         +-------+
                                              \            /
                                            ***  ***  ***  ***
                                           *   **   **   **   *
                                          *                    *
                                           *   MCast Tec A    *
                                          *                    *
                                           *   **   **   **   *
                                            ***  ***  ***  ***
   +-------+          +-------+                              |
   | Member|          | Member|                  +-------+
   |   G   |          |  Foo  |                  |  IMG  |
   +-------+          +-------+                  +-------+
       |                |                            |
       ***  ***  ***  ***                  ***  ***  ***  ***
      *   **   **   **   *                *   **   **   **   *
     *                    *  +-------+   *                    *
      *   MCast Tec A    * --|  IMG  |--  *   MCast Tec B    *    +-------+
     *                    *  +-------+   *                    * - | Member|
      *   **   **   **   *                *   **   **   **   *    |   G   |
       ***  ***  ***  ***                  ***  ***  ***  ***     +-------+
```

Figure 2: Reference scenarios for hybrid multicast, interconnecting group members from isolated homogeneous and heterogeneous domains.

3.2. Group Communication API and Protocol Stack

The group communication API consists of four parts. Two parts combine the essential communication functions, while the remaining two offer optional extensions for an enhanced monitoring and management:

Group Management Calls provide the minimal API to instantiate a multicast socket and to manage group membership.

Send/Receive Calls provide the minimal API to send and receive multicast data in a technology-transparent fashion.

Socket Options provide extension calls for an explicit configuration of the multicast socket such as setting hop limits or associated Interfaces.

Service Calls provide extension calls that grant access to internal multicast states of an Interface such as the multicast groups under subscription or the multicast forwarding information base.

Multicast applications that use the common API require assistance by a group communication stack. This protocol stack serves two needs:

- It provides system-level support to transfer the abstract functions of the common API, including namespace support, into protocol operations at Interfaces.

- It group communication services across different multicast technologies at the local host.

A general initiation of a multicast communication in this setting proceeds as follows:

1. An application opens an abstract multicast socket.

2. The application subscribes/leaves/(de)registers to a group using a Group Name.

3. An intrinsic function of the stack maps the logical group ID (Group Name) to a technical group ID (Group Address). This function may make use of deployment-specific knowledge such as available technologies and group address management in its domain.

4. Packet distribution proceeds to and from one or several multicast-enabled Interfaces.

The abstract multicast socket describes a group communication channel composed of one or multiple Interfaces. A socket may be created without explicit Interface association by the application, which leaves the choice of the underlying forwarding technology to the group communication stack. However, an application may also bind the socket to one or multiple dedicated Interfaces, which predefines the forwarding technology and the Multicast Namespace(s) of the Group Address(es).

Applications are not required to maintain mapping states for Group Addresses. The group communication stack accounts for the mapping of the Group Name to the Group Address(es) and vice versa. Multicast data passed to the application will be augmented by the corresponding Group Name. Multiple multicast subscriptions thus can be conducted on a single multicast socket without the need for Group Name encoding at the application side.

Hosts may support several multicast protocols. The group communication stack discovers available multicast-enabled Interfaces. It provides a minimal hybrid function that bridges data between different Interfaces and Multicast Domains. Details of service discovery are out of scope of this document.

The extended multicast functions can be implemented by a middleware as conceptually visualized in Figure 3.

```
*-------*     *-------*
| App 1 |     | App 2 |
*-------*     *-------*
    |             |
*---------------------*        ---|
|   Middleware        |           |
*---------------------*           |
     |          |                 |
*---------*     |                 |
| Overlay |     |                  \  Group Communication
*---------*     |                  /  Stack
     |          |                 |
     |          |                 |
*---------------------*           |
|   Underlay          |           |
*---------------------*        ---|
```

Figure 3: A middleware for offering uniform access to multicast in underlay and overlay

3.3. Naming and Addressing

Applications use Group Names to identify groups. Names can uniquely determine a group in a global communication context and hide technological deployment for data distribution from the application. In contrast, multicast forwarding operates on Group Addresses. Even though both identifiers may be identical in symbols, they carry different meanings. They may also belong to different Multicast Namespaces. The Namespace of a Group Address reflects a routing technology, while the Namespace of a Group Name represents the context in which the application operates.

URIs [RFC3986] are a common way to represent Namespace-specific identifiers in applications in the form of an abstract meta-data type. Throughout this document, all Group Names follows a URI notation with the syntax defined in Section 4.2.1. Examples are, ip://224.1.2.3:5000 for a canonical IPv4 ASM group, sip://news@cnn.com for an application-specific naming with service instantiator and default port selection.

An implementation of the group communication stack can provide convenience functions that detect the Namespace of a Group Name or further optimize service instantiation. In practice, such a library would provide support for high-level data types to the application, similar to some versions of the current socket API (e.g., InetAddress in Java). Using this data type could implicitly determine the Namespace. Details of automatic Namespace identification or service handling are out of scope of this document.

3.4. Namespaces

Namespace identifiers in URIs are placed in the scheme element and characterize syntax and semantic of the group identifier. They enable the use of convenience functions and high-level data types while processing URIs. When used in names, they may indicate an application context, or facilitate a default mapping and a recovery of names from addresses. They characterize its type, when used in addresses.

Compliant to the URI concept, namespace-schemes can be added. Examples of schemes are generic or inherited from applications.

3.4.1. Generic Namespaces

IP This namespace is comprised of regular IP node naming, i.e., DNS names and addresses taken from any version of the Internet Protocol. A processor dealing with the IP namespace is required to determine the syntax (DNS name, IP address version) of the group expression.

SHA-2 This namespace carries address strings compliant to SHA-2 hash digests. A processor handling those strings is required to determine the length of the group expression and passes appropriate values directly to a corresponding overlay.

Opaque This namespace transparently carries strings without further syntactical information, meanings, or associated resolution mechanism.

3.4.2. Application-centric Namespaces

SIP The SIP namespace is an example of an application-layer scheme that bears inherent group functions (conferencing). SIP conference URIs may be directly exchanged and interpreted at the application, and mapped to group addresses on the system level to generate a corresponding multicast group.

RELOAD This namespace covers address strings immediately valid in a RELOAD [I-D.ietf-p2psip-base] overlay network. A processor handling those strings may pass these values directly to a corresponding overlay.

3.5. Name-to-Address Mapping

The multicast communication paradigm requires all group members to subscribe to the same Group Name, taken from a common Multicast Namespace, and thereby to identify the group in a technology-agnostic way. Following this common API, a sender correspondingly registers a Group Name prior to transmission.

At communication end points, Group Names require a mapping to Group Addresses prior to service instantiation at its Interface(s). Similarly, a mapping is needed at gateways to translate between Group Addresses from different namespaces consistently. Two requirements need to be met by a mapping function that translates between Multicast Names and Addresses.

a. For a given Group Name, identify an Address that is appropriate for a local distribution instance.

b. For a given Group Address, invert the mapping to recover the Group Name.

In general, mapping can be complex and need not be invertible. A mapping can be realized by embedding smaller in larger namespaces or selecting an arbitrary, unused ID in a smaller target namespace. For example, it is not obvious how to map a large identifier space (e.g., IPv6) to a smaller, collision-prone set like IPv4 (see [I-D.venaas-behave-v4v6mc-framework][I-D.venaas-behave-mcast46]). Mapping functions can be stateless in some contexts, but may require states in others. The application of such functions depends on the cardinality of the namespaces, the structure of address spaces, and possible address collisions. However, some namespaces facilitate a canonical, invertible transformation to default address spaces.

3.5.1. Canonical Mapping

Some Multicast Namespaces defined in Section 3.4 can express a canonical default mapping. For example, ip://224.1.2.3:5000 indicates the correspondence to 224.1.2.3 in the default IPv4 multicast address space at port 5000. This default mapping is bound to a technology and may not always be applicable, e.g., in the case of address collisions. Note that under canonical mapping, the multicast URI can be completely recovered from any data message received from this group.

3.5.2. Mapping at End Points

Multicast listeners or senders require a Name-to-Address conversion for all technologies they actively run in a group. Even though a mapping applies to the local Multicast Domain only, end points may need to learn a valid Group Address from neighboring nodes, e.g., from a gateway in the collision-prone IPv4 domain. Once set, an end point will always be aware of the Name-to-Address correspondence and thus can autonomously invert the mapping.

3.5.3. Mapping at Interdomain Multicast Gateways

Multicast data may arrive at an IMG in one technology, requesting the gateway to re-address packets for another distribution system. At initial arrival, the IMG may not have explicit knowledge of the corresponding Multicast Group Name. To perform a consistent mapping, the group name potentially needs to be acquired out of band from a neighboring node.

3.6. A Note on Explicit Multicast (XCAST)

In Explicit Multicast (XCAST) [RFC5058] the multicast source explicitly pre-defines the receivers. From a conceptual perspective, XCAST is another distribution technology (i.e., a new technology-specific interface) for this API. The instantiation part of the Group Name may refer to multiple receivers. However, from an implementation perspective, this specification defines the syntax of the Group Name accordingly to [RFC3986], which does not support a set of (sub-)identifiers. Extending [RFC3986] is out of scope of this document.

Implementing XCAST without a set representation in the Group Name requires a topology-dependent mapping of the Name to a set of subscribers. Defining these details is out of scope of this document.

3.7. MTU Handling

This API considers a multi-technology scenario, in which different technologies may have different Maximum Transmission Unit (MTU) sizes. Even if the MTU size between two hosts has been determined, it may change over time either initiated by the network (e.g., path changes) or by the end hosts (e.g., interface change due to mobility).

The design of this API is based on the objective of robust communication and easy application development. The MTU handling and the place of fragmentation is thus guided by the following

observations:

Application The application programmer needs a simple way to send packets in a technology-agnostic fashion. The programmer needs to know the maximum amount of data that can be sent once per socket, but should not be distracted by changing MTU sizes. The configuration of the maximum message size by the application programmer while creating the socket may disrupt communication when (a) interfaces will be excluded or (b) the path MTU changes during transmission and thus disables the corresponding interfaces.

Middleware A middleware between application and technology interfaces ensures a general ability of packet handling, which prevents the application programmer to implement fragmentation. A maximum message size guaranteed by the group communication stack (e.g., middleware) is not allowed to change during runtime, as this would conflict with technology-agnostic development.

Technology Interfaces Fragmentation depends on the technology in use. The interfaces, thus, need to deal with MTU sizes that may vary between interfaces and along different paths.

This API proposes to guarantee a maximum message size for the application programmer and to handle fragmentation at the interface level. However, the application programmer should be able to determine the technology-specfic atomic message size, e.g., for optimization reasons.

The maximum message size value should be realistic (e.g., following IP clients) but detailed values are out of scope of this document.

4. Common Multicast API

4.1. Notation

The following description of the common multicast API is described in pseudo syntax. Variables that are passed to function calls are declared by "in", return values are declared by "out". A list of elements is denoted by <>. The pseudo syntax assumes that lists include an attribute which represents the number of elements.

The corresponding C signatures are defined in Appendix A.

4.2. Abstract Data Types

4.2.1. Multicast URI

Multicast Names and Multicast Addresses used in this API follow an URI scheme that defines a subset of the generic URI specified in [RFC3986] and is compliant with the guidelines in [RFC4395].

The multicast URI is defined as follows:

```
scheme "://" group "@" instantiation ":" port "/" sec-credentials
```

The parts of the URI are defined as follows:

scheme refers to the specification of the assigned identifier [RFC3986] which takes the role of the Multicast Namespace.

group identifies the group uniquely within the Namespace given in scheme.

instantiation identifies the entity that generates the instance of the group (e.g., a SIP domain or a source in SSM), using syntax and semantic as defined by the Namespace given in scheme. This parameter is optional. Note that ambiguities (e.g., identical node addresses in multiple overlay instances) can be distinguished by ports.

port identifies a specific application at an instance of a group. This parameter is optional.

sec-credentials used to implement optional security credentials (e.g., to authorize a multicast group access). Note that security credentials carry a distinct technical meaning w.r.t. AAA schemes and may differ between group members. Hence the sec-credentials are not considered part of the Group Name.

4.2.2. Interface

The Interface denotes the layer and instance on which the corresponding call will be effective. In agreement with [RFC3493] we identify an Interface by an identifier, which is a positive integer starting at 1.

Properties of an Interface are stored in the following struct:

```
    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", ... */
    };
```

The following function retrieves all available Interfaces from the system:

```
    getInterfaces(out Interface <ifs>);
```

It extends the functions for Interface Identification defined in Section 4 of [RFC3493] and can be implemented by:

```
    struct if_prop *if_prop(void);
```

4.2.3. Membership Events

A membership event is triggered by a multicast state change, which is observed by the current node. It is related to a specific Group Name and may be receiver or source oriented.

```
    event_type {
            join_event;
            leave_event;
            new_source_event;
    };

    event {
           event_type event;
           Uri group_name;
           Interface if;
    };
```

An event will be created by the group communication stack and passed to applications that have registered for events.

4.3. Group Management Calls

4.3.1. Create

The create call initiates a multicast socket and provides the application programmer with a corresponding handle. If no Interfaces will be assigned based on the call, the default Interface will be selected and associated with the socket. The call returns an error code in the case of failures, e.g., due to a non-operational middleware.

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

The ifs argument denotes a list of Interfaces (if_indexes) that will be associated with the multicast socket. This parameter is optional.

On success a multicast socket identifier is returned, otherwise NULL.

4.3.2. Delete

The delete call removes the multicast socket.

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

The s argument identifies the multicast socket for destruction.

On success the out parameter error is 0, otherwise -1.

4.3.3. Join

The join call initiates a subscription for the given Group Name. Depending on the Interfaces that are associated with the socket, this may result in an IGMP/MLD report or overlay subscription, for example.

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

The s argument identifies the multicast socket.

The groupName argument identifies the group.

On success the out parameter error is 0, otherwise -1.

4.3.4. Leave

The leave call results in an unsubscription for the given Group Name.

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

The s argument identifies the multicast socket.

The groupName identifies the group.

On success the out parameter error is 0, otherwise -1.

4.3.5. Source Register

The srcRegister call registers a source for a Group on all active Interfaces of the socket s. This call may assist group distribution in some technologies, for example the creation of sub-overlays. It may remain without effect in some multicast technologies.

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

The s argument identifies the multicast socket.

The groupName argument identifies the multicast group to which a source intends to send data.

The ifs argument points to the list of Interface indexes for which the source registration failed. A NULL pointer is returned, if the list is empty. This parameter is optional.

If source registration succeeded for all Interfaces associated with the socket, the out parameter error is 0, otherwise -1.

4.3.6. Source Deregister

The srcDeregister indicates that a source does no longer intend to send data to the multicast group. This call may remain without effect in some multicast technologies.

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

The s argument identifies the multicast socket.

The group_name argument identifies the multicast group to which a source has stopped to send multicast data.

The ifs argument points to the list of Interfaces for which the source deregistration failed. A NULL pointer is returned, if the list is empty.

If source deregistration succeeded for all Interfaces associated with the socket, the out parameter error is 0, otherwise -1.

4.4. Send and Receive Calls

4.4.1. Send

The send call passes multicast data destined for a Multicast Name from the application to the multicast socket.

It is worth noting that it is the choice of the programmer to send data via one socket per group or to use a single socket for multiple groups.

```
send(in Socket s, in Uri groupName,
     in Size msgLen, in Msg msgBuf,
     out Int error);
```

The s argument identifies the multicast socket.

The groupName argument identifies the group to which data will be sent.

The msgLen argument holds the length of the message to be sent.

The msgBuf argument passes the multicast data to the multicast socket.

On success the out parameter error is 1. If the message is too long the out parameter is 0, otherwise -1.

4.4.2. Receive

The receive call passes multicast data and the corresponding Group Name to the application.

It is worth noting that it is the choice of the programmer to receive data via one socket per group or to use a single socket for multiple groups.

```
receive(in Socket s, out Uri groupName,
        out Size msgLen, out Msg msgBuf,
        out Int error);
```

The s argument identifies the multicast socket.

The group_name argument identifies the multicast group for which data was received.

The msgLen argument holds the length of the received message.

The msgBuf argument points to the payload of the received multicast data.

On success the out parameter error is 1. If the message is too long the out parameter is 0, otherwise -1.

4.5. Socket Options

The following calls configure an existing multicast socket.

4.5.1. Get Interfaces

The getInterface call returns an array of all available multicast communication Interfaces associated with the multicast socket.

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

The s argument identifies the multicast socket.

The ifs argument points to an array of Interface index identifiers.

On success the out parameter error is 0, otherwise -1.

4.5.2. Add Interface

The addInterface call adds a distribution channel to the socket. This may be an overlay or underlay Interface, e.g., IPv6 or DHT. Multiple Interfaces of the same technology may be associated with the socket.

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

The s and if arguments identify a multicast socket and Interface, respectively.

On success the value 0 is returned, otherwise -1.

4.5.3. Delete Interface

The delInterface call removes the Interface if from the multicast socket.

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

The s and if arguments identify a multicast socket and Interface, respectively.

On success the out parameter error is 0, otherwise -1.

4.5.4. Set TTL

The setTTL call configures the maximum hop count for the socket a multicast message is allowed to traverse.

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

The s and h arguments identify a multicast socket and the maximum hop count, respectively.

The ifs argument points to an array of Interface index identifiers. This parameter is optional.

On success the out parameter error is 0, otherwise -1.

4.5.5. Get TTL

The getTTL call returns the maximum hop count a multicast message is allowed to traverse for the socket.

```
getTTL(in Socket s,
       out Int h, out Int error);
```

The s argument identifies a multicast socket.

The h argument holds the maximum number of hops associated with socket s.

On success the out parameter error is 0, otherwise -1.

4.5.6. Atomic Message Size

The getAtomicMsgSize function returns the maximum message size that an application is allowed to transmit per socket at once without fragmentation. This value depends on the interfaces associated with the socket in use and thus may change during runtime.

```
getAtomicMsgSize(in Socket s,
                 out Int return);
```

On success, the function returns a positive value, otherwise -1.

4.6. Service Calls

4.6.1. Group Set

The groupSet call returns all multicast groups registered at a given Interface. This information can be provided by group management states or routing protocols. The return values distinguish between sender and listener states.

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

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

The if argument identifies the Interface for which states are maintained.

The groupSet argument points to a list of group states.

On success the out parameter error is 0, otherwise -1.

4.6.2. Neighbor Set

The neighborSet function returns the set of neighboring nodes for a given Interface as seen by the multicast routing protocol.

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

The if argument identifies the Interface for which neighbors are inquired.

The neighborsAddresses argument points to a list of neighboring nodes on a successful return.

On success the out parameter error is 0, otherwise -1.

4.6.3. Children Set

The childrenSet function returns the set of child nodes that receive multicast data from a specified Interface for a given group. For a common multicast router, this call retrieves the multicast forwarding information base per Interface.

```
childrenSet(in Interface if, in Uri groupName,
            out Uri <childrenAddresses>, out Int error);
```

The if argument identifies the Interface for which children are inquired.

The groupName argument defines the multicast group for which distribution is considered.

The childrenAddresses argument points to a list of neighboring nodes on a successful return.

On success the out parameter error is 0, otherwise -1.

4.6.4. Parent Set

The parentSet function returns the set of neighbors from which the current node receives multicast data at a given Interface for the specified group.

```
parentSet(in Interface if, in Uri groupName,
          out Uri <parentsAddresses>, out Int error);
```

The if argument identifies the Interface for which parents are inquired.

The groupName argument defines the multicast group for which distribution is considered.

The parentsAddresses argument points to a list of neighboring nodes on a successful return.

On success the out parameter error is 0, otherwise -1.

4.6.5. Designated Host

The designatedHost function inquires whether this host has the role of a designated forwarder resp. querier, or not. Such an information is provided by almost all multicast protocols to prevent packet duplication, if multiple multicast instances serve on the same subnet.

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

The if argument identifies the Interface for which designated forwarding is inquired.

The groupName argument specifies the group for which the host may attain the role of designated forwarder.

The function returns 1 if the host is a designated forwarder or querier, otherwise 0. The return value -1 indicates an error.

4.6.6. Enable Membership Events

The enableEvents function registers an application at the group communication stack to receive information about group changes. State changes are the result of new receiver subscriptions or leaves as well as of source changes. Upon receiving an event, the group service may obtain additional information from further service calls.

```
enableEvents();
```

Calling this function, the stack starts to pass membership events to the application. Each event includes an event type identifier and a Group Name (cf., Section 4.2.3).

The multicast protocol has not to support membership tracking to enable this feature. This function can also be implemented at the middelware layer.

4.6.7. Disable Membership Events

The disableEvents function deactivates the information about group state changes.

```
disableEvents();
```

On success the stack will not pass membership events to the application.

4.6.8. Maximum Message Size

The getMaxMsgSize function returns the maximum message size that an application is allowed to transmit per socket at once. This value is guaranteed by the group communication stack.

```
getMaxMsgSize(out Int return);
```

On success, the function returns a positive value, otherwise -1.

5. Implementation

A reference implementation of the Common API for Transparent Hybrid

Multicast is available with the HAMcast stack [hamcast-dev], [GC2010], [LCN2012]. This open-source software supports the multicast API (C++ and Java library) for group application development, the middleware as a userspace system service, and several multicast-technology modules. The middleware is implemented in C++.

This API is verified and adjusted based on the real-world experiences gathered in the HAMcast project.

6. IANA Considerations

This document makes no request of IANA.

7. Security Considerations

This draft does neither introduce additional messages nor novel protocol operations.

8. Acknowledgements

We would like to thank the HAMcast-team, Dominik Charousset, Gabriel Hege, Fabian Holler, Alexander Knauf, Sebastian Meiling, and Sebastian Woelke, at the HAW Hamburg for many fruitful discussions and for their continuous critical feedback while implementing the common multicast API and a hybrid multicast middleware. We gratefully acknowledge WeeSan, Mario Kolberg, and John Buford for their suggestions to improve the document. We would like to thank the Name-based socket BoF (in particular Dave Thaler) for clarifying insights into the question of meta function calls.

This work is partially supported by the German Federal Ministry of Education and Research within the HAMcast project, which is part of G-Lab.

9. Informative References

[GC2010] Meiling, S., Charousset, D., Schmidt, T., and M. Waehlisch, "System-assisted Service Evolution for a Future Internet - The HAMcast Approach to Pervasive Multicast", Proc. of IEEE GLOBECOM 2010 Workshops. MCS 2010, pp. 938-942, Piscataway, NJ, USA: IEEE Press, December 2010.

[I-D.ietf-mboned-auto-multicast]

Bumgardner, G., "Automatic Multicast Tunneling", draft-ietf-mboned-auto-multicast-14 (work in progress), June 2012.

[I-D.ietf-p2psip-base]
Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", draft-ietf-p2psip-base-21 (work in progress), March 2012.

[I-D.venaas-behave-mcast46]
Venaas, S., Asaeda, H., SUZUKI, S., and T. Fujisaki, "An IPv4 - IPv6 multicast translator", draft-venaas-behave-mcast46-02 (work in progress), December 2010.

[I-D.venaas-behave-v4v6mc-framework]
Venaas, S., Li, X., and C. Bao, "Framework for IPv4/IPv6 Multicast Translation", draft-venaas-behave-v4v6mc-framework-03 (work in progress), June 2011.

[LCN2012] Meiling, S., Schmidt, T., and M. Waehlisch, "Large-Scale Measurement and Analysis of One-Way Delay in Hybrid Multicast Networks", Proc. of 37th Annual IEEE Conference on Local Computer Networks (LCN 2012). Piscataway, NJ, USA: IEEE Press, October 2012.

[RFC1075] Waitzman, D., Partridge, C., and S. Deering, "Distance Vector Multicast Routing Protocol", RFC 1075, November 1988.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2710] Deering, S., Fenner, W., and B. Haberman, "Multicast Listener Discovery (MLD) for IPv6", RFC 2710, October 1999.

[RFC3376] Cain, B., Deering, S., Kouvelas, I., Fenner, B., and A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, October 2002.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[RFC3678] Thaler, D., Fenner, B., and B. Quinn, "Socket Interface

Extensions for Multicast Source Filters", RFC 3678, January 2004.

[RFC3810] Vida, R. and L. Costa, "Multicast Listener Discovery Version 2 (MLDv2) for IPv6", RFC 3810, June 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4395] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 35, RFC 4395, February 2006.

[RFC4601] Fenner, B., Handley, M., Holbrook, H., and I. Kouvelas, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)", RFC 4601, August 2006.

[RFC4604] Holbrook, H., Cain, B., and B. Haberman, "Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast", RFC 4604, August 2006.

[RFC5015] Handley, M., Kouvelas, I., Speakman, T., and L. Vicisano, "Bidirectional Protocol Independent Multicast (BIDIR-PIM)", RFC 5015, October 2007.

[RFC5058] Boivie, R., Feldman, N., Imai, Y., Livens, W., and D. Ooms, "Explicit Multicast (Xcast) Concepts and Options", RFC 5058, November 2007.

[RFC5757] Schmidt, T., Waehlisch, M., and G. Fairhurst, "Multicast Mobility in Mobile IP Version 6 (MIPv6): Problem Statement and Brief Survey", RFC 5757, February 2010.

[hamcast-dev] "HAMcast developers", <http://hamcast.realmv6.org/developers>.

Appendix A. C Signatures

This section describes the C signatures of the common multicast API, which are defined in Section 4.

```
int createMSocket(int* result, size_t num_ifs, const uint32_t* ifs);

int deleteMSocket(int s);

int join(int msock, const char* group_uri);

int leave(int msock, const char* group_uri);

int srcRegister(int msock,
                const char* group_uri,
                size_t num_ifs,
                const uint32_t *ifs);

int srcDeregister(int msock,
                  const char* group_uri,
                  size_t num_ifs,
                  const uint32_t *ifs);

int send(int msock,
         const char* group_uri,
         size_t buf_len,
         const void* buf);

int receive(int msock,
            const char* group_uri,
            size_t buf_len,
            void* buf);

int getInterfaces(int msock,
                  size_t* num_ifs,
                  uint32_t** ifs);

int addInterface(int msock, uint32_t iface);

int delInterface(int msock, uint32_t iface);

int setTTL(int msock, uint8_t value,
           size_t num_ifs, uint32_t* ifs);

int getTTL(int msock, uint8_t* result);

int getAtomicMsgSize(int msock);
```

```
typedef struct {
    char* group_uri; /* registered mcast group */
    int type; /* 0: listener state,
                 1: sender state
                 2: sender and listener state */
}
GroupSet;

int groupSet(uint32_t iface,
             size_t* num_groups,
             GroupSet** groups);

int neighborSet(uint32_t iface,
                const char* group_name,
                size_t* num_neighbors,
                char** neighbor_uris);

int childrenSet(uint32_t iface,
                const char* group_name,
                size_t* num_children,
                char** children_uris);

int parentSet(uint32_t iface,
              const char* group_name,
              size_t* num_parents,
              char** parents_uris);

int designatedHost(uint32_t iface,
                   const char* group_name);

typedef void (*MembershipEventCallback)(int,          /* event type   */
                                        uint32_t,     /* interface id
*/
                                        const char*); /* group uri
*/

int registerEventCallback(MembershipEventCallback callback);

int disableEvents();

   int getMaxMsgSize();
```

Appendix B. Practical Example of the API

```
-- Application above middleware:

//Initialize multicast socket;
//the middleware selects all available interfaces
MulticastSocket m = new MulticastSocket();

m.join(URI("ip://224.1.2.3:5000"));
m.join(URI("ip://[FF02:0:0:0:0:0:0:3]:6000"));
m.join(URI("sip://news@cnn.com"));

-- Middleware:

join(URI mcAddress) {
  //Select interfaces in use
  for all this.interfaces {
    switch (interface.type) {
      case "ipv6":
        //... map logical ID to routing address
        Inet6Address rtAddressIPv6 = new Inet6Address();
        mapNametoAddress(mcAddress,rtAddressIPv6);
        interface.join(rtAddressIPv6);
      case "ipv4":
        //... map logical ID to routing address
        Inet4Address rtAddressIPv4 = new Inet4Address();
        mapNametoAddress(mcAddress,rtAddressIPv4);
        interface.join(rtAddressIPv4);
      case "sip-session":
        //... map logical ID to routing address
        SIPAddress rtAddressSIP = new SIPAddress();
        mapNametoAddress(mcAddress,rtAddressSIP);
        interface.join(rtAddressSIP);
      case "dht":
        //... map logical ID to routing address
        DHTAddress rtAddressDHT = new DHTAddress();
        mapNametoAddress(mcAddress,rtAddressDHT);
        interface.join(rtAddressDHT);
       //...
    }
  }
}
```

Appendix C. Deployment Use Cases for Hybrid Multicast

This section describes the application of the defined API to implement an IMG.

C.1. DVMRP

The following procedure describes a transparent mapping of a DVMRP-based any source multicast service to another many-to-many multicast technology.

An arbitrary DVMRP [RFC1075] router will not be informed about new receivers, but will learn about new sources immediately. The concept of DVMRP does not provide any central multicast instance. Thus, the IMG can be placed anywhere inside the multicast region, but requires a DVMRP neighbor connectivity. The group communication stack used by the IMG is enhanced by a DVMRP implementation. New sources in the underlay will be advertised based on the DVMRP flooding mechanism and received by the IMG. Based on this the event "new_source_event" is created and passed to the application. The relay agent initiates a corresponding join in the native network and forwards the received source data towards the overlay routing protocol. Depending on the group states, the data will be distributed to overlay peers.

DVMRP establishes source specific multicast trees. Therefore, a graft message is only visible for DVMRP routers on the path from the new receiver subnet to the source, but in general not for an IMG. To overcome this problem, data of multicast senders will be flooded in the overlay as well as in the underlay. Hence, an IMG has to initiate an all-group join to the overlay using the namespace extension of the API. Each IMG is initially required to forward the received overlay data to the underlay, independent of native multicast receivers. Subsequent prunes may limit unwanted data distribution thereafter.

C.2. PIM-SM

The following procedure describes a transparent mapping of a PIM-SM-based any source multicast service to another many-to-many multicast technology.

The Protocol Independent Multicast Sparse Mode (PIM-SM) [RFC4601] establishes rendezvous points (RP). These entities receive listener and source subscriptions of a domain. To be continuously updated, an IMG has to be co-located with a RP. Whenever PIM register messages are received, the IMG must signal internally a new multicast source using the event "new_source_event". Subsequently, the IMG joins the group and a shared tree between the RP and the sources will be established, which may change to a source specific tree after a sufficient number of data has been delivered. Source traffic will be forwarded to the RP based on the IMG join, even if there are no further receivers in the native multicast domain. Designated routers of a PIM-domain send receiver subscriptions towards the PIM-SM RP.

The reception of such messages initiates the event "join_event" at the IMG, which initiates a join towards the overlay routing protocol. Overlay multicast data arriving at the IMG will then transparently be forwarded in the underlay network and distributed through the RP instance.

C.3. PIM-SSM

The following procedure describes a transparent mapping of a PIM-SSM-based source specific multicast service to another one-to-many multicast technology.

PIM Source Specific Multicast (PIM-SSM) is defined as part of PIM-SM and admits source specific joins (S,G) according to the source specific host group model [RFC4604]. A multicast distribution tree can be established without the assistance of a rendezvous point.

Sources are not advertised within a PIM-SSM domain. Consequently, an IMG cannot anticipate the local join inside a sender domain and deliver a priori the multicast data to the overlay instance. If an IMG of a receiver domain initiates a group subscription via the overlay routing protocol, relaying multicast data fails, as data are not available at the overlay instance. The IMG instance of the receiver domain, thus, has to locate the IMG instance of the source domain to trigger the corresponding join. In the sense of PIM-SSM, the signaling should not be flooded in underlay and overlay.

One solution could be to intercept the subscription at both, source and receiver sites: To monitor multicast receiver subscriptions ("join_event" or "leave_event") in the underlay, the IMG is placed on path towards the source, e.g., at a domain border router. This router intercepts join messages and extracts the unicast source address S, initializing an IMG specific join to S via regular unicast. Multicast data arriving at the IMG of the sender domain can be distributed via the overlay. Discovering the IMG of a multicast sender domain may be implemented analogously to AMT [I-D.ietf-mboned-auto-multicast] by anycast. Consequently, the source address S of the group (S,G) should be built based on an anycast prefix. The corresponding IMG anycast address for a source domain is then derived from the prefix of S.

C.4. BIDIR-PIM

The following procedure describes a transparent mapping of a BIDIR-PIM-based any source multicast service to another many-to-many multicast technology.

Bidirectional PIM [RFC5015] is a variant of PIM-SM. In contrast to

PIM-SM, the protocol pre-establishes bidirectional shared trees per group, connecting multicast sources and receivers. The rendezvous points are virtualized in BIDIR-PIM as an address to identify on-tree directions (up and down). However, routers with the best link towards the (virtualized) rendezvous point address are selected as designated forwarders for a link-local domain and represent the actual distribution tree. The IMG is to be placed at the RP-link, where the rendezvous point address is located. As source data in either cases will be transmitted to the rendezvous point address, the BIDIR-PIM instance of the IMG receives the data and can internally signal new senders towards the stack via the "new_source_event". The first receiver subscription for a new group within a BIDIR-PIM domain needs to be transmitted to the RP to establish the first branching point. Using the "join_event", an IMG will thereby be informed about group requests from its domain, which are then delegated to the overlay.

Appendix D. Change Log

The following changes have been made from draft-irtf-samrg-common-api-04

1. Added section "A Note on Explicit Multicast (XCAST)"

2. Added section "MTU Handling"

3. Added socket option getAtomicMSgSize

4. Added service call getMaxMsgSize

The following changes have been made from draft-irtf-samrg-common-api-03

1. Added section "Illustrative Example"

2. Added section "Implementation"

3. Minor clarifications

The following changes have been made from draft-irtf-samrg-common-api-02

1. Added use case of multicast flavor support

2. Restructured Section 3

3. Major update on namespaces and on mapping

4. C signatures completed

5. Many clarifications and editorial improvements

The following changes have been made from draft-irtf-samrg-common-api-01

1. Pseudo syntax for lists objects changed

2. Editorial improvements

The following changes have been made from draft-irtf-samrg-common-api-00

1. Incorrect pseudo code syntax fixed

2. Minor editorial improvements

The following changes have been made from draft-waehlisch-sam-common-api-06

1. no changes; draft adopted as WG document (previous draft-waehlisch-sam-common-api-06, now draft-irtf-samrg-common-api-00)

The following changes have been made from draft-waehlisch-sam-common-api-05

1. Description of the Common API using pseudo syntax added

2. C signatures of the Comon API moved to appendix

3. updateSender() and updateListener() calls replaced by events

4. Function destroyMSocket renamed as deleteMSocket.

The following changes have been made from draft-waehlisch-sam-common-api-04

1. updateSender() added.

The following changes have been made from draft-waehlisch-sam-common-api-03

1. Use cases added for illustration.

2. Service calls added for inquiring on the multicast distribution system.

3. Namespace examples added.

4. Clarifications and editorial improvements.

The following changes have been made from draft-waehlisch-sam-common-api-02

1. Rename init() in createMSocket().

2. Added calls srcRegister()/srcDeregister().

3. Rephrased API calls in C-style.

4. Cleanup code in "Practical Example of the API".

5. Partial reorganization of the document.

6. Many editorial improvements.

The following changes have been made from draft-waehlisch-sam-common-api-01

1. Document restructured to clarify the realm of document overview and specific contributions s.a. naming and addressing.

2. A clear separation of naming and addressing was drawn. Multicast URIs have been introduced.

3. Clarified and adapted the API calls.

4. Introduced Socket Option calls.

5. Deployment use cases moved to an appendix.

6. Simple programming example added.

7. Many editorial improvements.

Authors' Addresses

Matthias Waehlisch
link-lab & FU Berlin
Hoenower Str. 35
Berlin 10318
Germany

Email: mw@link-lab.net
URI: http://www.inf.fu-berlin.de/~waehl

Thomas C. Schmidt
HAW Hamburg
Berliner Tor 7
Hamburg 20099
Germany

Email: schmidt@informatik.haw-hamburg.de
URI: http://inet.cpt.haw-hamburg.de/members/schmidt

Stig Venaas
cisco Systems
Tasman Drive
San Jose, CA 95134
USA

Email: stig@cisco.com