SAM Research Group	M. Waehlisch
Internet-Draft	link-lab & FU Berlin
Intended status: Informational	T C. Schmidt
Expires: September 08, 2011	HAW Hamburg
	S. Venaas
	cisco Systems
	March 07, 2011

A Common API for Transparent Hybrid Multicast
draft-waehlisch-sam-common-api-06

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 using a stable, upper layer protocol controlled 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

Copyright Notice

Table of Contents

*1. Introduction

*1.1. Use Cases for the Common API

*2. Terminology

*3. Overview

*3.1. Objectives and Reference Scenarios

*3.2. Group Communication API & Protocol Stack

*3.3. Naming and Addressing

*3.4. Mapping

*4. Common Multicast API

*4.1. Notation

*4.2. Abstract Data Types

*4.2.1. Multicast URI

*4.2.2. Interface

*4.2.3. Membership Events

*4.3. Group Management Calls

*4.3.1. Create

*4.3.2. Delete

*4.3.3. Join

*4.3.4. Leave

*4.3.5. Source Register

*4.3.6. Source Deregister

*4.4. Send and Receive Calls

*4.4.1. Send

*4.4.2. Receive

*4.5. Socket Options

*4.5.1. Get Interfaces

*4.5.2. Add Interface

*4.5.3. Delete Interface

*4.5.4. Set TTL

*4.5.5. Get TTL

*4.6. Service Calls

*4.6.1. Group Set

*4.6.2. Neighbor Set

*4.6.3. Children Set

*4.6.4. Parent Set

*4.6.5. Designated Host

*4.6.6. Enable Membership Events

*4.6.7. Disable Membership Events

*5. Functional Details

*5.1. Namespaces

*5.2. Mapping

*6. IANA Considerations

*7. Security Considerations

*8. Acknowledgements

*9. References

*Appendix A. C Signatures

*Appendix B. Practical Example of the API

*Appendix C. Deployment Use Cases for Hybrid Multicast

*Appendix C.1. DVMRP

*Appendix C.2. PIM-SM

*Appendix C.3. PIM-SSM

*Appendix C.4. BIDIR-PIM

*Appendix D. Change Log

*Authors' Addresses

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.

The standard multicast socket options [RFC3493], [RFC3678] are bound to an IP version and do not distinguish between naming and addressing of multicast identifiers. Group communication, however, is commonly implemented in different flavors such as any source (ASM) vs. source specific mutlicast (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 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:

- 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.

- Allow for a flexible namespace support in group addressing, and thereby separate naming and addressing/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 P2P 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 proposes and discusses mapping mechanisms between different namespaces and forwarding technologies. 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

Four 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 its deployment in target domains. They are thus enabled to develop programs once that run in every deployment scenario. The employment 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 simplifies the design and implementation of gateways and translators.

Simplified Service Deployment through Generic Gateways The 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 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.

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 ID.

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.

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

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.

Inter-domain Multicast Gateway: 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 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

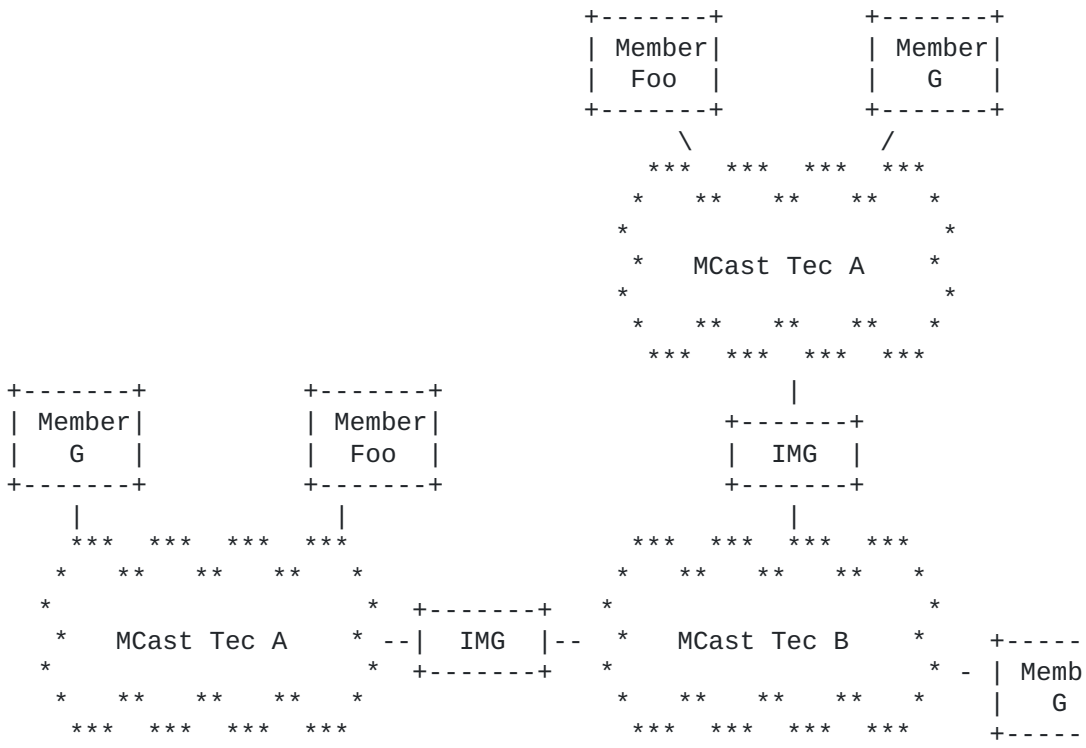
- programmers are enabled to process group names in their programs without the need to consider technological mappings to designated deployments in target domains;

- 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 refers to a reference scenario that covers the following two hybrid deployment cases displayed in Figure 1:

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.



It is assumed throughout the document that the domain composition, as well as the node attachment to a specific technology remain unchanged during a multicast session.

3.2. Group Communication API & 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 management:

Group Management Calls provide the minimal API to instantiate a multicast socket and 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 like 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 bridges data distribution between different multicast technologies.

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

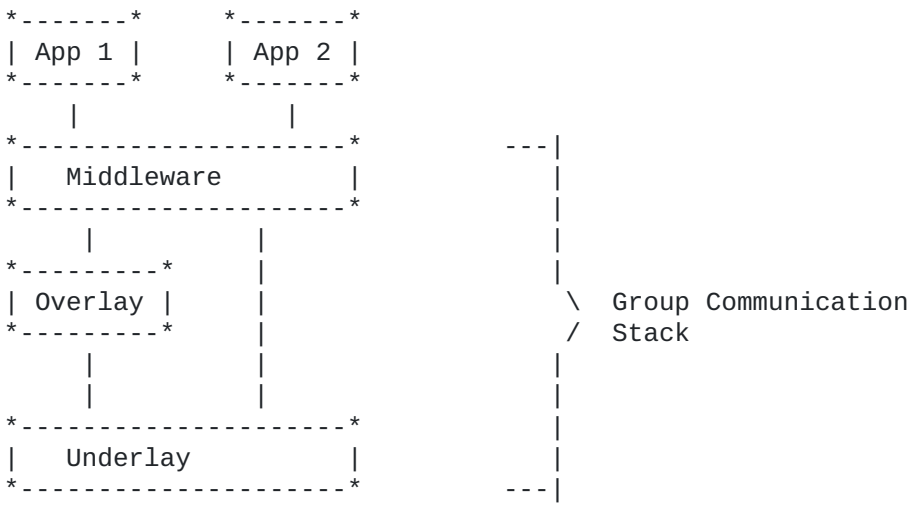
1. An application opens an abstract multicast socket.

2. The application subscribes/leaves/(de)registers to a group using a logical group identifier.

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 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 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 communication 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 2.



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 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, any kind of Group Name 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 middleware can provide convenience functions that detect the namespace of a Group Name and use it to optimize service instantiation. In practice, such a library would provide support for high-level data types to the application, similar to the current socket API (e.g., InetAddress in Java). Using this data type could implicitly determine the namespace. Details of automatic namespace identification is out-of-scope of this document.

3.4. Mapping

All group members subscribe to the same Group Name taken from a common namespace and thereby identify the group in a technology-agnostic way. Group Names require a mapping to addresses prior to service instantiation at an Interface. Similarly, a mapping is needed at gateways to translate between Group Addresses from different namespaces. Some namespaces facilitate a canonical transformation to default address spaces. For example, ip://224.1.2.3:5000 has an obvious correspondence to 224.1.2.3 in the IPv4 multicast address space. Note that in this example the multicast URI can be completely recovered from any data packet received from this group.

However, mapping in general can be more complex and need not be invertible. 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. For example, it is not obvious how to map a large identifier space (e.g., IPv6) to a smaller, collision-prone set like IPv4.

Two (or more) Multicast Addresses from different namespaces may belong to

a. the same logical group (i.e., same Multicast Name)

b. different multicast channels (i.e., different technical IDs).

This decision can be based on invertible mappings. However, the application of such functions depends on the cardinality of the namespaces and thus does not hold in general. It is not obvious how to map a large identifier space (e.g., IPv6) to a smaller set (e.g., IPv4).

A mapping can be realized by embedding smaller in larger namespaces or selecting an arbitrary, unused ID in the target space. The relation between logical and technical ID is maintained by mapping functions which can be stateless or stateful. The middleware thus queries the mapping service first, and creates a new technical group ID only if there is no identifier available for the namespace in use. The Group Name is associated with one or more Group Addresses, which belong to different namespaces. Depending on the scope of the mapping service, it ensures a consistent use of the technical ID in a local or global domain.

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 corresponding C signatures are defined in Appendix 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 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 the namespace given in scheme.

port identifies a specific application at an instance of a group.

sec-credentials used to implement security credentials (e.g., to authorize a multicast group access).

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

It extends the functions for Interface Identification in [RFC3493] (cf., Section 4) and can be implemented by:

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

4.2.3. Membership Events

A membership event is triggered by a mutlicast 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
};
```

An event will be created by the middleware and passed to applcations that are 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 may return an error code in the case of failures, e.g., due to a non-operational middleware.

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

The if 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 value 0 is returned, otherwise -1.

4.3.3. Join

The join call initiates a subscription for the given group. 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 group_name, out Int error);
```

The s argument identifies the multicast socket.
The group_name argument identifies the group.
On success the value 0 is returned, otherwise -1.

4.3.4. Leave

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

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

The s argument identifies the multicast socket.
The group_name identifies the group.
On success the value 0 is returned, 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, the creation of sub-overlays, for example. Not all multicast technologies require his call.

```
    srcRegister(in Socket s, in Uri group_name,
                in Int num_ifs, 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 intends to send data.
The num_ifs argument holds the number of elements in the ifs array. This parameter is optional.
The ifs argument points to the list of interface indexes for which the source registration failed. If num_ifs was 0 on output, a NULL pointer is returned. This parameter is optional.
If source registration succeeded for all interfaces associated with the socket, the value 0 is returned, 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 group_name,
                  in Int num_ifs, 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 num_ifs argument holds the number of elements in the ifs array.
The ifs argument points to the list of interfaces for which the source deregistration failed. If num_ifs was 0 on output, a NULL pointer is returned.
If source deregistration succeeded for all interfaces associated with the socket, the value 0 is returned, otherwise -1.

4.4. Send and Receive Calls

4.4.1. Send

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

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

The s argument identifies the multicast socket.
The group_name argument identifies the group to which data will be sent.
The msg_len argument holds the length of the message to be sent.
The msg_buf argument passes the multicast data to the multicast socket.
On success the value 0 is returned, otherwise -1.

4.4.2. Receive

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

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

The s argument identifies the multicast socket.
The group_name argument identifies the multicast group for which data was received.
The msg_len argument holds the length of the received message.
The msg_buf argument points to the payload of the received multicast data.
On success the value 0 is returned, 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 Int num_ifs,
                  out Interface <ifs>, out Int error);
```

The s argument identifies the multicast socket.

The num_ifs argument holds the number of interfaces in the ifs list.
The ifs argument points to an array of interface index identifiers.
On success the value 0 or lager is returned, 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 delnterface 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 value 0 is returned, 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 Int num_ifs, in Interface <ifs>,
           out Int error);
```

The s and h arguments identify a multicast socket and the maximum hop count, respectively.
The num_ifs argument holds the number of interfaces in the ifs list. This parameter is optional.
The ifs argument points to an array of interface index identifiers. This parameter is optional.
On success the value 0 is returned, 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 value 0 is returned, 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.

```
    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 */
    }
```

The if argument identifies the interface for which states are maintained.
The num_groups argument holds the number of groups in the groupSet array.
The groupSet argument points to a lilst of group states.
On success the value 0 is returned, 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 Int num_neighbors,
                out Uri <neighbor_address>, out Int error);
```

The if argument identifies the interface for which neighbors are inquired.
The num_neighbors argument holds the number of addresses in the neighbor_address array.
The neighbor_address argument points to a list of neighboring nodes on a successful return.
On success the value 0 is returned, 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 group_name,
                out Int num_children, out Uri <child_address>,
                out Int error);
```

The if argument identifies the interface for which children are inquired.
The group_name argument defines the multicast group for which distribution is considered.
The num_children argument holds the number of addresses in the child_address array.

The child_address argument points to a list of neighboring nodes on a successful return.
On success the value 0 is returned, 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 group_name,
          out Int num_parents, out Uri parent_address,
          out Int error);
```

The if argument identifies the interface for which parents are inquired.
The group_name argument defines the multicast group for which distribution is considered.
The num_parents argument holds the number of addresses in the parent_address array.
The parent_address argument points to a list of neighboring nodes on a successful return.
On success the value 0 is returned, otherwise -1.

4.6.5. Designated Host

The designatedHost function inquires whether the 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 group_name
               out Int return);
```

The if argument identifies the interface for which designated forwarding is inquired.
The group_name 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 middleware to inform the application about a group change. This is the result of receiver new subscriptions or leaves as well as the observation of source changes. The group service may call other service calls to get additional information.

```
void enableEvents();
```

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

4.6.7. Disable Membership Events

The disableEvents function deactivates the information about group state changes.

```
    void disableEvents();,
```

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

5. Functional Details

In this section, we describe specific functions of the API and the associated system middleware in detail.

5.1. 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 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 and functions currently foreseen include

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.

OLM This namespace covers address strings immediately valid in an overlay network. A processor handling those strings need not be aware of the address generation mechanism, but may pass these values directly to a corresponding overlay.

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.

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

5.2. Mapping

Group Name to Group Address, SSM/ASM TODO

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 API and a hybrid multicast middleware.
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. References

[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[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.
[RFC1075]	Waitzman, D., Partridge, C. and S. Deering, "Distance Vector Multicast Routing Protocol", RFC 1075, November 1988.
[RFC5015]	Handley, M., Kouvelas, I., Speakman, T. and L. Vicisano, "Bidirectional Protocol Independent Multicast (BIDIR-PIM)", RFC 5015, October 2007.
[RFC3810]	Vida, R. and L. Costa, "Multicast Listener Discovery Version 2 (MLDv2) for IPv6", RFC 3810, June 2004.
[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.
[RFC2710]	Deering, S., Fenner, W. and B. Haberman, "Multicast Listener Discovery (MLD) for IPv6", RFC 2710, October 1999.
[RFC4601]	Fenner, B., Handley, M., Holbrook, H. and I. Kouvelas, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)", RFC 4601, August 2006.
[RFC3376]	Cain, B., Deering, S., Kouvelas, I., Fenner, B. and A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, October 2002.
[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.
[I-D.ietf-mboned-auto-multicast]	Thaler, D, Talwar, M, Aggarwal, A, Vicisano, L, Pusateri, T and T Morin, "Automatic IP Multicast Tunneling", Internet-Draft draft-ietf-mboned-auto-multicast-11, July 2011.

Appendix A. C Signatures

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

```
    int createMSocket(uint32_t *if);

    int deleteMSocket(int s);

    int join(int s, const uri group_name);

    int leave(int s, const uri group_name);

    int srcRegister(int s, const uri group_name,
                    uint_t num_ifs, uint_t *ifs);

    int srcDeregister(int s, const uri group_name,
                      uint_t num_ifs, uint_t *ifs);

    int send(int s, const uri group_name,
             size_t msg_len, const void *buf);

    int receive(int s, const uri group_name,
                size_t msg_len, msg *msg_buf);

    int getInterfaces(int s, uint_t num_ifs, uint_t *ifs);

    int addInterface(int s, uint32_t if);

    int delInterface(int s, uint32_t if);

    int setTTL(int s, int h, uint_t num_ifs, uint_t *ifs);

    int getTTL(int s, int h);

    int groupSet(uint32_t if, uint_t *num_groups,
                 struct groupSet *groupSet);

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

    int neighborSet(uint32_t if, uint_t *num_neighbors,
                    const uri *neighbor_address);

    int childrenSet(uint32_t if, const uri group_name,
                   uint_t *num_children, const uri *child_address);

    int parentSet(uint32_t if, const uri group_name, uint_t *num_parents
                    const uri *parent_address);

    int designatedHost(uint32_t if, const uri *group_name);
```

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.

Appendix 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.

Appendix 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.

Appendix 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.

Appendix 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-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 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 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 Venaas cisco Systems
Tasman Drive San Jose, CA 95134 USA EMail: stig@cisco.com