Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 13 October, 2009

P. Adamska
A. Wierzbicki
T. Kaszuba
PJIIT

May 18, 2009

# Publish-subscribe over the generic Peer-to-Peer Protocols
**draft-paulina-p2psip-pubsub-01**

## Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on October, 2009.

## Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

## Abstract

This document introduces a generic publish-subscribe protocol, that can be built on top of RELOAD or P2PP. It works both for the unstructured and DHT-based Peer-to-Peer networks. Moreover it is highly customizable to address the optimization issues and support different topology structures. It can be used to implement P2P-SIP services such as presence or event notification.

**Table of contents**

1. Introduction..............................................2
2. Terminology...............................................2
3. Architecture..............................................3
3.1 Overview.................................................3
3.2 Publish-subscribe transport..............................4
3.3 Core Algorithm...........................................4
3.4 Customizable Algorithm...................................6
3.5 Topology Manager.........................................6
3.6 Publish-Subscribe API....................................8
3.6.1 Publish-Subscribe Interface............................8
3.6.2 Publish-Subscribe Callbacks............................9
4. Publish-Subscribe Protocol...............................10
4.1 General information.....................................10
4.1.1 Topic.................................................10
4.1.2 Subscriber............................................11
4.1.3 Subscription..........................................11
4.1.4 Event.................................................12
4.1.5 Interest conditions...................................12
4.1.6 Access control rules..................................13
4.2 Default Customizable Algorithm..........................14
4.2.1 Create topic..........................................14
4.2.2 Transfer topic........................................16
4.2.3 Remove topic..........................................19
4.2.4 Subscribe.............................................19
4.2.5 Unsubscribe...........................................20
4.2.6 Publish...............................................22
4.2.7 Notify................................................22
4.2.8 Maintenance...........................................23
4.2.9 Reliability...........................................26
4.3 Message formats.........................................26
4.3.1 Standard header.......................................27
4.3.2 Standard request header...............................28
4.3.3 Create topic..........................................29
4.3.4 Subscribe.............................................29
4.3.5 Unsubscribe...........................................30
4.3.6 Publish...............................................30
4.3.7 Notify................................................30
4.3.8 Standard response header..............................31
4.3.9 Subscribe response....................................31
4.3.10 Keep-alive...........................................32
4.4 Objects formats.........................................32
4.4.1 AC rules..............................................32
4.4.2 Rule..................................................32
4.4.3 Operation.............................................33
4.4.4 User..................................................33
4.4.5 Subscriber............................................34
4.5 Message types...........................................34
4.6 Event types.............................................35
4.7 Response codes..........................................35
4.8 Security................................................35

5. Integrating publish-subscribe with P2PP/RELOAD................35
5.1 Extending P2PP/RELOAD interfaces..........................35
5.1.1 Callbacks...........................................36
5.1.2 Objects.............................................43
5.1.3 API.................................................44
5.2 Usage of the extension....................................44
5.2.1 Objects.............................................44
6. Future work....................................................45
7. IANA Considerations............................................45
8. References.....................................................45
8.1 Normative references......................................45
8.2 Informative references....................................45
Authors' Addresses................................................46

## 1. Introduction

The publish-subscribe protocol described in this paper is built on top of the generic P2P protocols such as RELOAD[1] and P2PP[2]. Both of them make the differences between peer-to-peer routing algorithms transparent to the higher-layer applications. Our main concern while designing the publish-subscribe protocol was to preserve this transparency in the presence of a great variety of overlay types. The proposed protocol may exchange messages both directly between the specified peers or by encapsulating them in the resource objects and sending inside the P2P insert request. A set of callback methods has been defined to let the publish-subscribe layer capture the incoming P2P requests, process them and modify the default P2P protocol behavior if it is necessary. In this document we describe the generic protocol, that works both for the unstructured and DHT-based P2P networks. It can be easily configured to optimize its behavior using the overlay-specific features or simply replaced by a set of different algorithms for the various P2P networks. Both operations are completely transparent for the higher-layer applications and topology-independent. Additionally we also define the Access Control Rules (AC), which allows higher-layer application to let some of the users subscribe for the specified topic without granting them permission to generate events. These rules are stored by all the topic subscribers, as they can be responsible for accepting other nodes' requests. The proposed publish-subscribe protocol also provides so called Interest Conditions (IC) which can be used as a simple filter for the received events. Each node can define the group of users generating interesting events. Each node also stores a certain number of events, that have recently been published for the topic. The new subscriber may ask for sending some or all of them after the successful subscription process. Nodes may also store a configurable number of the 'backup nodes' in the special caches, to use them in case of the direct parent's failure. The proposed protocol allows replacement of the default publish-subscribe algorithm with some other ones. This means that it may be integrated with a set of different algorithms to be dynamically loaded depending on the underlying overlay type (for instance Scribe-like for Pastry

and some other for different ones). Moreover such operation is completely transparent for the higher-layer applications and topology-independent.

Due to the terminology differences between P2PP and RELOAD, in this document we use 'insert' for the P2PP publish object and RELOAD store request, and 'lookup' for the P2PP lookup and RELOAD fetch request where distinguishing one from another is not essential.

This document is organized as follows. Section 3 describes the core components of the protocol and briefly explains, how the proposed mechanism may be extended. The proposed protocol MAY be integrated with many different publish-subscribe algorithms. Section 4 describes the elements common for all of them, introduces the default algorithm and shows how it can be configured to optimize its behavior for the DHT-based overlays. Detailed information about integration of the publish-subscribe protocol with P2PP and RELOAD are presented in section 5.

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [3].

Some of the terminology has been borrowed from the P2PP[1] and RELOAD[2] drafts.

Topic owner: The node which has created the topic.

Topic root: A non-leaf node in the topology structure - indirect parent for all the other nodes. It MAY be a topic subscriber or not. In the DHT-based overlays, it will be the peer with id numerically closest to the topic identifier.

Topic history: Every node participating in the topology structure, that is able to accept new subscribers (which means it is a peer, not client) stores the list of the events that have recently been published for the specified topic. During the subscription process node can ask its parent to send some, or all of them. History only refers to the custom events, not the predefined ones, such as AC_MODIFIED.

Topic cache: The backup list of nodes participating in the topology structure which is used in case of detecting the direct parent's failure. There may be several types of caches stored be a single node.

## 3. Architecture

### 3.1 Overview

To provide a generic and highly extensible publish-subscribe mechanism we decided to split it, as it is shown in the figure 3.1.1. The arrows illustrate the communication between the various components of the publish-subscribe layer.

```
+---------------------------------------------------------------------+
|                        Publish-Subscribe API                        |
| +-----------------------------+     +-----------------------------+ |
| | Publish-Subscribe Callbacks |     | Publish-Subscribe Interface | |
| +-----------------------------+     +-----------------------------+ |
|           ^         ^                                 |             |
+-----------|---------|---------------------------------|-------------+
            |         +------------------------+        |
            +----------------------+           |        |
                                   |           |        v
+-------------------------------+  |  +-------------------------------+
|      Topology Manager         |  |  |     Customizable Algorithm    |
+-------------------------------+  |  +-------------------------------+
                ^                  |                  ^
                |                  |                  |
                v                  |                  v
+---------------------------------------------------------------------+
|                            Core Algorithm                           |
| +-----------------------------------------------------------------+ |
| |                      Algorithm Configurator                     | |
| +-----------------------------------------------------------------+ |
+---------------------------------------------------------------------+
                                  ^
                                  |
                                  v
+---------------------------------------------------------------------+
|                      Publish-Subscribe Transport                    |
| +-----------------------------+     +-----------------------------+ |
| |    RELOAD/P2PP Transport    |     |       Direct Transport      | |
| +-----------------------------+     +-----------------------------+ |
+---------------------------------------------------------------------+
```

Figure 3.1.1: Publish-subscribe layer architecture

The following sections provide the detailed description of the most important components of the publish-subscribe layer.

### 3.2 Publish-subscribe transport

This component is directly responsible for the communication. It passes the incoming publish-subscribe messages to the Core Algorithm and sends the outgoing ones. It hides the details of the sending procedure - for example encapsulates a publish-subscribe message inside the network resource object if it is to be sent within the P2P insert request. All the publish-subscribe requests may be routed either directly to some specified node, or using the overlay-specific algorithm. The second method is chosen, when the exact destination is not defined - for example to determine the root during the topic creation procedure in a DHT-based network. Indications and responses are always passed directly to a specified node.

[TODO: Perhaps for the direct connections publish-subscribe use RELOAD's Attach() - as it was described in the SIP Usage]

### 3.3 Core Algorithm

This component provides the core features of the publish-subscribe layer which are to be common for all the Customizable Algorithms. All the incoming messages are passed directly to this component. It checks whether it stores information about the specified topic (or does not store - for the create topic request) and whether the message originator is allowed to perform a certain operation. If both requirements are fulfilled, it passes the message to the Customizable Algorithm or the Topology Manager. Otherwise it sends the response with an appropriate error code to the request originator itself. It is also responsible for forwarding the events notifications to the node's children and checking IC, before passing them to the higher-layer application. Probably the most interesting feature of this component is its ability to query the P2P layer for the overlay type and dynamically load the appropriate algorithm with the suitable configuration. This can be achieved by defining a separate Algorithm Configurator that is used by the Core Algorithm to create an appropriate Customizable Algorithm object. The choice of the suitable component is made on the basis of the underlying P2P network type. This object is also able to configure such parameters as the size of the caches. Caches store the lists of the 'backup nodes' to be used in case of detecting the direct parent's failure. Each node may store several types of such lists, for example associated with different levels of the multicast tree. The decision on the number of caches depends on the Customizable Algorithm component's policy. Each parent is responsible for filling its children's caches by periodically sending them keep-alive messages containing the appropriate lists of nodes. Storing such information is essential in the unstructured networks. However it may not be crucial in the DHT-based ones, where the topology structure can be repaired at a relatively low cost using the overlay-specific routing algorithm,

without performing the complex lookup procedures. In such case the Algorithm Configurator may set the cache size to 0 and treat event notifications as a heartbeat messages to reduce the additional traffic associated with the structure maintenance. Moreover, if some new Customizable Algorithm requires sending any additional information in a standard publish-subscribe message - it can register this extended message format to make the Core Algorithm load it instead of the standard message object after receiving the specified request. For example the default Customizable Algorithm stores the information about the Interest Conditions locally, but some different algorithm may choose to built the IC-based multicast tree, where the child's IC are the subset of its parent's IC. This requires adding the information about the IC to the standard subscribe request. Such extended message can be registered using the Algorithm Configurator. Figure 3.3.1 illustrates the usage of this component.

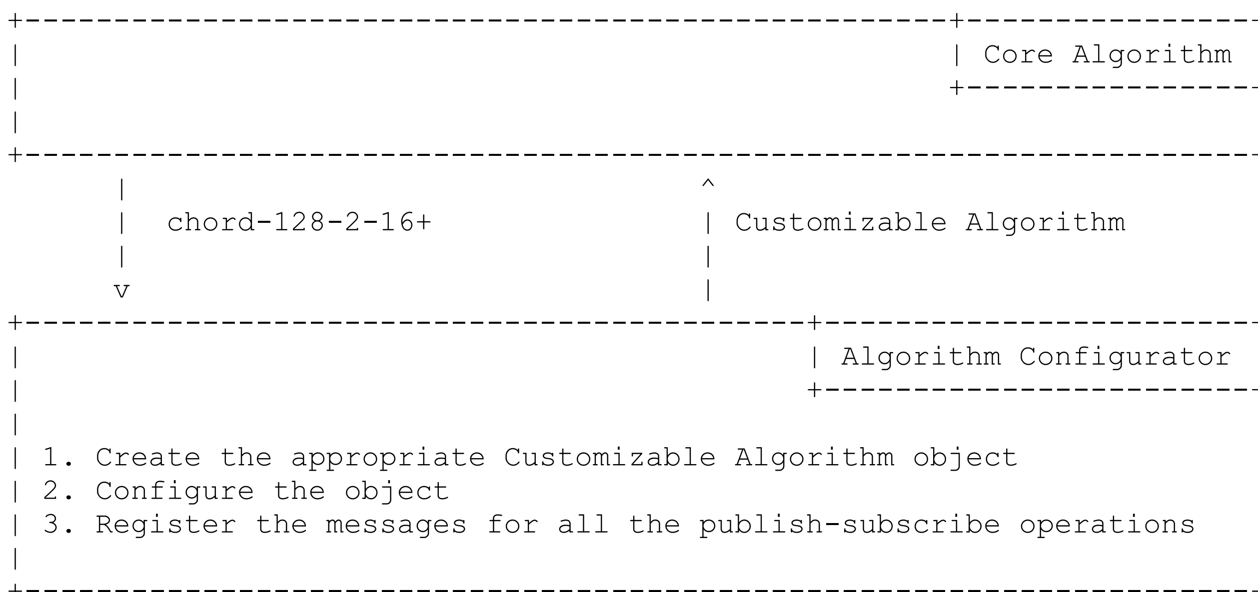



Figure 3.3.1: Usage of the Algorithm Configurator component

**3.4 Customizable Algorithm**

This component implements all the methods from the Publish-Subscribe Interface and takes care of the processing of most of them. For instance the create topic operation requires performing different procedures in the DTH-based and unstructured overlays. In the first case it is enough to encapsulate the create topic request inside the P2P insert request, giving the topic identifier as an object key. The unstructured networks require performing the lookup procedure to ensure that some other node has not created the specified topic before. It is also necessary to place the SUBSCRIPTIONINFO objects in the P2P network to inform other nodes about the existing participants of the topology structure created for the particular

topic. Additionally in the DHT-based networks transferring certain topics to the new root may be necessary, if some new node joins the P2P network and its ID is closer to the topic ID according to the overlay-specific distance metrics. The Customizable Algorithm component also takes part in the topology structure creation process - this feature is described in details in the next section.

### 3.5 Topology Manager

This component is responsible for creating and maintaining the specified topology structure. It handles all the subscribe requests and takes part in the topic transfers. Currently there are two structures defined: star and multicast tree. For the first one, topology manager only ensures, that none of the nodes except the topic root accepts new subscribers. The situation is slightly more complex for the multicast trees. For instance the Customizable Algorithm component may require creating IC-based trees. Then the node, receiving a  subscribe request, must check whether the new subscriber's IC are the subset of its own ones. If it is not - it has to forward the request to the higher level of the multicast tree because it will not be receiving all the events that are interesting to the new subscriber. In such cases Topology Manager must consult the Customizable Algorithm component to identify the next hop for forwarding the subscribe request. Figure 3.5.1 illustrates the subscribe procedure that uses the described mechanism. Figure 3.5.2 shows the structure that does not apply any additional rules for accepting the new subscribers. In this case the marked node B may not be receiving all the interesting events.

```
 (a)                     +------------+     (b)     +------------+
                         |    Root    |             |    Root    |
                         | IC={1,2,3} |-+           | IC={1,2,3} |
                         +------------+ |           +------------+
                           ^    |       |             /          \
                 subscribe |    |       |            /            \
                           |    |       |           /              \
                           |    |       |          /                \
              subscribe  +------------+ |  +------------+ +------------+
             +---------->|     A      | |  |     B      | |     A      |
             |           |  IC={1,2}  | |  |  IC={2,3}  | |  IC={1,2}  |
    +------------+       +------------+ |  +------------+ +------------+
    |     B      |                      |
    |  IC={2,3}  |<---------------------+
    +------------+  subscriber accepted
```

Figure 3.5.1: Subscribe procedure that applies the additional rules defined by the Customizable Algorithm component (a) and the multicast tree after completing the procedure (b)

```
 (a)                       +------------+     (b)         +------------+
                           |    Root    |                 |    Root    |
                           | IC={1,2,3} |                 | IC={1,2,3} |
                           +------------+                 +------------+
                                  |                              |
                                  |                              |
                                  |                       +------------+
                                  |                       |      A     |
               subscribe   +------------+                 |  IC={1,2}  |
             +------------>|      A     |                 +------------+
             |             |  IC={1,2}  |-+                      |
     +------------+        +------------+ |                      |
     |     B      |                       |               +============+
     |  IC={2,3}  |<----------------------+               |     B      |
     +------------+    subscriber accepted                |  IC={2,3}  |
                                                          +============+
```

Figure 3.5.2: Subscribe procedure that does not apply the additional rules defined by the Customizable Algorithm component (a) and the multicast tree after completing the procedure (b)

Figure 3.5.3 briefly describes the exact procedures performed by the node receiving a subscribe request to build an IC-based multicast tree using the proposed mechanism.

```
  +----------------+----------+     +------------------------+----------+
  | Tree Topology |           |     | Customizable Algorithm |          |
  +---------------+           |     +------------------------+          |
  |                          |     |                                   |
  | 2. Query for a better    |     | 3. If the new subscriber's IC are |
  |    node to accept the    |     |    the subset of this node's IC   |
  |    request               |     |    return null                    |
  | 4. If the better node == |     | 3'.Otherwise return this node's   |
  |    null, accept the      |     |    parent                         |
  |    request               |     |                                   |
  | 4'.If the better node    |     |                                   |
  |    != null, forward the  |     |                                   |
  |    request to that node  |     |                                   |
  |                          |     |                                   |
  +--------------------------+     +-----------------------------------+
     ^                 |       ^             |                     ^
     |1.subscribe      |2      |             |3.better             |
     |  request        |       |             |  node               |
     |                 |       |             |                     |
     |                 |       |             |                     |
  +----------------+---|-------|-------------|---------------------|---+
  | Core Algorithm |   |       +-------------+                     |   |
  +----------------+   +-------------------------------------------+   |
  |                                                                    |
  |                                                                    |
```

```
 |  1. If the topic exists and the node is allowed to subscribe for   |
 |     it                                                             |
 +--------------------------------------------------------------------+
```

Figure 3.5.3: Procedure performed by the node receiving a subscribe request

### 3.6 Publish-Subscribe API

#### 3.6.1 Publish-Subscribe Interface

Application issues a publish-subscribe request using one of the methods described below. All of them are implemented by the Customizable Algorithm component. Each operation is asynchronous, as it can take a long time to complete. Parameters enclosed in the square brackets are optional.

```
void createTopic(String topicId, boolean subscribe,
                 [AccessControlRules ac])
```

@param topicId Topic identifier.

@param subscribe Value indicating whether this node wants to automatically subscribe for the specified topic after its creation. In this case the 'create topic' request is specially prepared and there is no need to send a separate 'subscribe' request later on.

@param ac Access control rules defined for the operations associated with this topic.

```
void removeTopic(String topicId)
```

@param topicId Topic identifier.

```
void subscribe(String topicId, [InterestConditions ic],
               [int eventIndex])
```

@param topicId Topic identifier.

@param ic Object representing Interest Conditions defined for the particular topic by the higher-layer application.

@param eventIndex Value indicating which events from the topic history node wants to receive after successfully completing the subscribe operation.

void unsubscribe(String topicId)

@param topicId Topic identifier.

void publish(String topicId, Event e)

@param topicId Topic identifier.

@param e Event to be published.

### 3.6.2 Publish-Subscribe Callbacks

Additionally the publish-subscribe layer defines a set of callback methods, which are invoked after completing asynchronous operations and can be implemented by the higher-layer application. These methods are described below.

public void onTopicSubscribe(String topicId)

@param topicId Topic identifier.

Called after a successful subscription.

public void onTopicCreate(String topicId)

@param topicId Created topic identifier.

Called after a successful topic creation.

public void onTopicNotify(String topicId, byte[] message)

@param topicId Topic identifier.

@param message Message encapsulated in the received event.

Called when node receives a custom event notification.

public void onTopicRemove(String topicId)

@param topicId Removed topic identifier.

Called when node receives the notification informing that the particular topic has been removed.

public void onPubSubError(String topicId, Operation o, int errorCode)

@param topicId Topic identifier.

@param o Operation which failed to complete successfully.

@param errorCode Error code.

Called when an asynchronous operation fails to complete successfully.

# 4. Publish-Subscribe Protocol

## 4.1 General information

### 4.1.1 Topic

Apart from the identifier, there are several information stored for each topic:

1) Owner

   The node which has created the topic

2) Parent

   Node's parent in the topology structure

3) Access Control Rules

   Described in details in the section 4.1.6

4) Interest Conditions

   Described in details in the section 4.1.5

5) Subscription list

   The list of topic subscribers, which are this node's children in the topology structure

6) Caches

   The backup information about other nodes participating in the topology structure

7) Distance

   The distance between the peer id and the topic id (calculated

using the overlay-specific metrics)

8) History

The list of the previously published events for the specified topic

### 4.1.2 Subscriber

Apart from the topic identifier, this object contains information such as:

1) User name

User name in the Peer-to-Peer network

2) Peer id

Peer id in the Peer-to-Peer network

3) IP address

4) Port number

Port used for exchanging the publish-subscribe messages directly

### 4.1.3 Subscription

Subscription contains the following information:

1) Subscriber

2) Expiration time

How long the subscription is valid - after this period node needs to resubscribe

### 4.1.4 Event

Every publish-subscribe event contains information about its publisher and the identifier of the topic it is associated with. The proposed publish-subscribe protocol defines three types of events:

- remove topic - informs all subscribers that the particular topic has been removed

- AC modified - informs all subscribers that the Access Control Rules (section 4.1.6) for the specified topic have been modified

- custom - this event encapsulates the user-defined events

### 4.1.5 Interest conditions

Each subscriber can define its Interest Conditions (IC) for the topic. This means he can declare that he wants to receive information only about the events published by a specified group of users. It can be described as follows:

```
operation:  eventtype(ALL): interestingusers[]
            eventtype(...):interestingusers[]
```

If the 'interestingusers' list for the specified event type is empty, it means that it is always interesting regardless the publisher. The conditions defined for the event type ALL are always checked first. If the 'interestingusers' list for it is empty, the decision on whether the specified event is interesting or not, depends on the 'interestingusers' associated with the particular event type.

If there is no rule defined for some operation or event - it means it is not interesting at all.

By default all the subscribers will be receiving every topic event. It means that each non-leaf node in the topology structure has to pass every event to all of its children. IC are defined locally and checked by the topic subscriber before invoking the higher-layer notify callback. Otherwise the node's parent in the topology structure would have to define similar IC (to receive all the events, that are interesting for its children). Additionally after modifying IC, subscriber would in some cases have to be passed to a different parent in the topology structure, generating extra traffic.

If IC for some topic are defined as follows:

```
NOTIFY: eventtype(ALL):
        eventtype(REMOVETOPIC):
        eventtype(MODIFYAC):
        eventtype(CUSTOM): user1, user2, user3
```

then after receiving a CUSTOM notification from user1, 2 or 3, node will inform higher layer about it. REMOVETOPIC and MODIFYAC events are by default interesting regardless the publisher.

### 4.1.6 Access control rules

Nodes can define a set of rules for the topic, to grant other users different access permissions (AC). These rules can be described as follows:

```
operation:  eventtype(ALL): allowusers[]
            eventtype(...): allowusers[]
```

If the 'allowusers' list for a specified event type is empty, it means that the associated operation can be performed by any subscriber. The permissions defined for the event type ALL are always checked first. If the 'allowusers' list for it is empty, then granting access permission depends exclusively on the 'allowusers' defined for the particular event type.

If there is no rule for some operation or event - it means it is not allowed.

For example if the rules for some topic are:

```
SUBSCRIBE: eventtype(ALL): user1, user2, user3, user4
PUBLISH: eventtype(ALL):
         eventtype(REMOVETOPIC): user1
         eventtype(MODIFYAC): user1
         eventtype(CUSTOM): user1, user2, user3
```

It means that:

- Only user1 is allowed to modify AC rules for the topic. By default this permission is granted exclusively to the topic owner.
- If the topology structure participant receives a 'subscribe' request for example from user5, subscription will fail due to access control restrictions. Only users 1, 2, 3 and 4 are allowed to subscribe for the topic.
- User4 may (according to SUBSCRIBE rules) subscribe for receiving the topic events, but isn't allowed (according to PUBLISH rules) to generate them. Users 1, 2 and 3 are all allowed to publish user-defined events, but only user1 has permission to remove this topic. By default only topic owner has permission to publish the predefined events such as REMOVETOPIC or MODIFYAC.

In case of any collisions in the AC rules - the most important is the one marked with ALL clause. For example if the rules are defined as follows:

```
PUBLISH: eventtype(ALL): user1, user2, user3
         eventtype(REMOVETOPIC): user1
         eventtype(MODIFYAC): user1
         eventtype(CUSTOM): user4
```

User4 will not be able to publish any event, because the first rule to be checked is: eventtype(ALL): user1, user2, user3. It could be repaired either by removing the 'allowusers' list for event type ALL, or adding user4 to it.

The topic owner always retains permission to modify AC rules and remove the topic, but it can also grant it to other users. Rules

defined for the 'notify' operation MAY differ for different nodes. By default each subscriber will only accept event notifications from its parent in the topology structure.

### 4.2 Default Customizable Algorithm

This section describes the default Customizable Algorithm component's behavior and the possibilities of configuring it to address the optimization issues. In some cases it is essential to determine whether the underlying P2P network is a DHT-based or an unstructured one. To fulfill this requirement we define an additional method that asks the generic P2P protocol to calculate the distance between the two given keys according to the overlay-specific metrics. Such calculations are possible only in a DHT-based network. Each publish-subscribe Customizable Algorithm component MUST implement the set of operations defined in this section. In the diagrams shown in this section we use '{PUBSUB}msg' notation to indicate, that the message 'msg' is sent directly to the specified node and '{P2P}insert(msg)' in case of messages encapsulated within an insert request.

#### 4.2.1 Create topic

To perform this operation node encapsulates the 'create topic' message inside the P2P resource object, sets its key to the topic ID and sends it within the P2P insert request. In the DHT-based networks it is enough for the node receiving such request to check, whether it does not store information about the specified topic itself (figure 4.2.1). The unstructured networks require some lookup procedure to determine, whether the topic has not been created by a different node (figure 4.2.2). Such situation may occur, because the object encapsulated inside the P2P resource object will in most cases be stored by the 'create topic' request originator. This is where the idea of the SUBSCRIPTIONINFO objects comes along. They are associated with the particular topic and contain contact information about its subscribers. The status of such object may be PENDING or ACCEPTED. The difference between both states will be explained later on. Each node receiving a 'create topic' request places a pending SUBSCRIPTIONINFO object in the underlying P2P network. Then it looks for other objects associated with the specified topic. If there is at least one with the accepted status - it assures us that the specified topic already exists. If the underlying P2P network stores only the pending SUBSCRIPTIONINFO objects, than the peer with the lowest ID is allowed to become the root for the specified topic. All the others are supposed to remove their SUBSCRIPTIONINFO objects and send the response with an appropriate error code. The 'create topic' request may additionally contain a list of nodes, which are to be added to the topic subscribers after successfully completing the operation. This way its originator does not have to issue a separate 'subscribe' request later on. The 'create topic' message may also be used to

transfer the topic to the new root, in which case a special flag is set to indicate that this request refers to an existing topic. Regardless the purpose, the described request always contains AC rules defined for the topic.

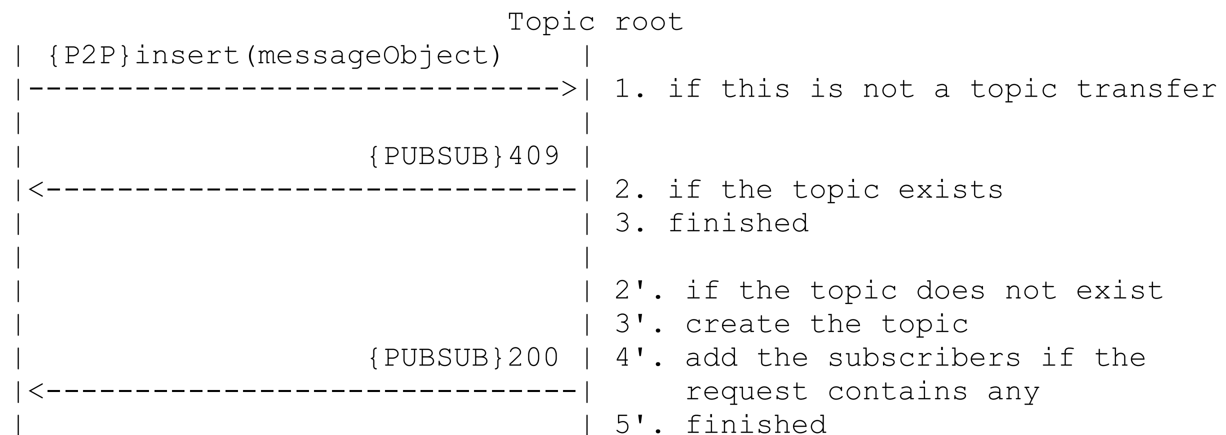
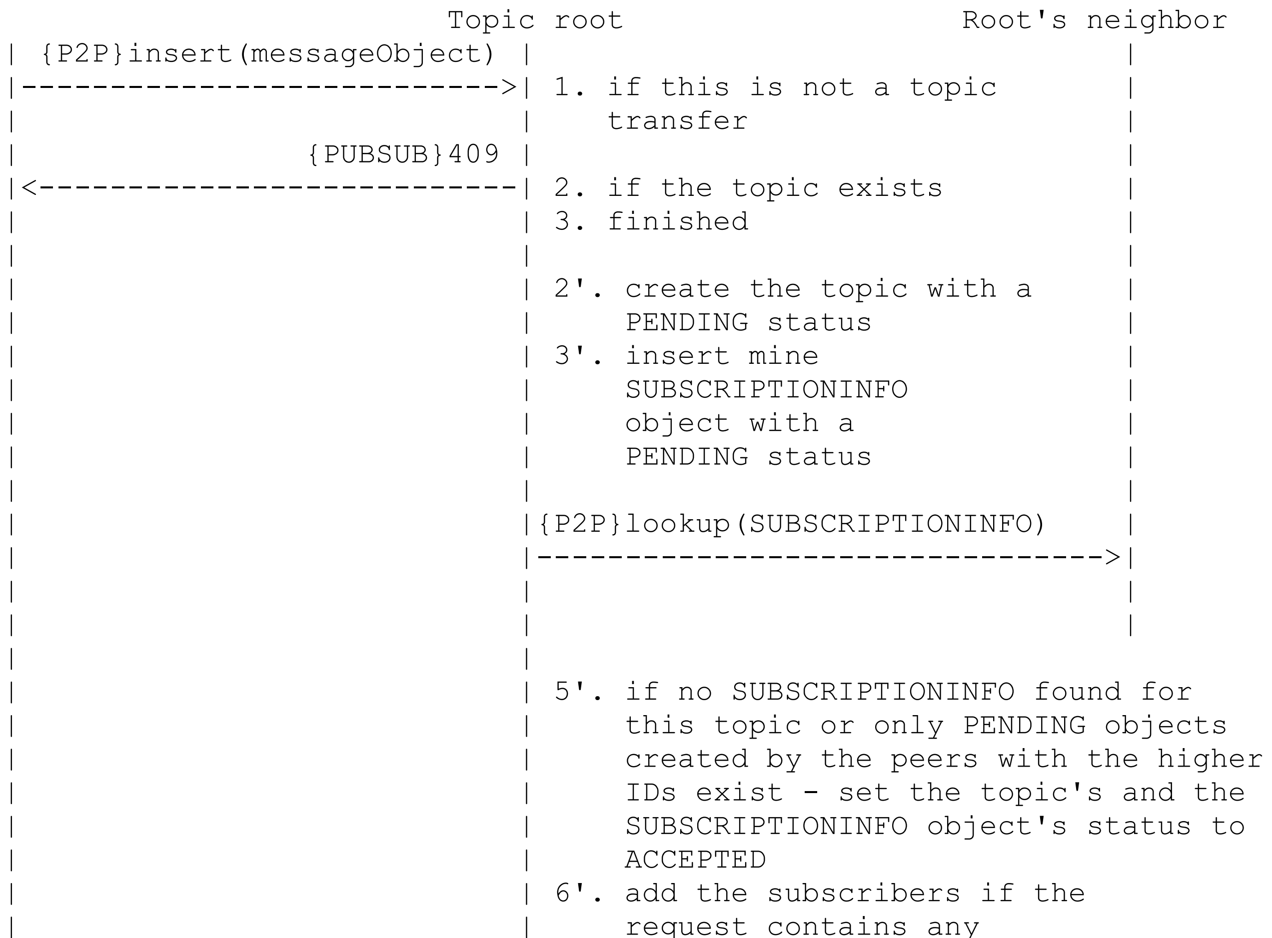



Figure 4.2.1: Create topic procedure in a DHT-based network



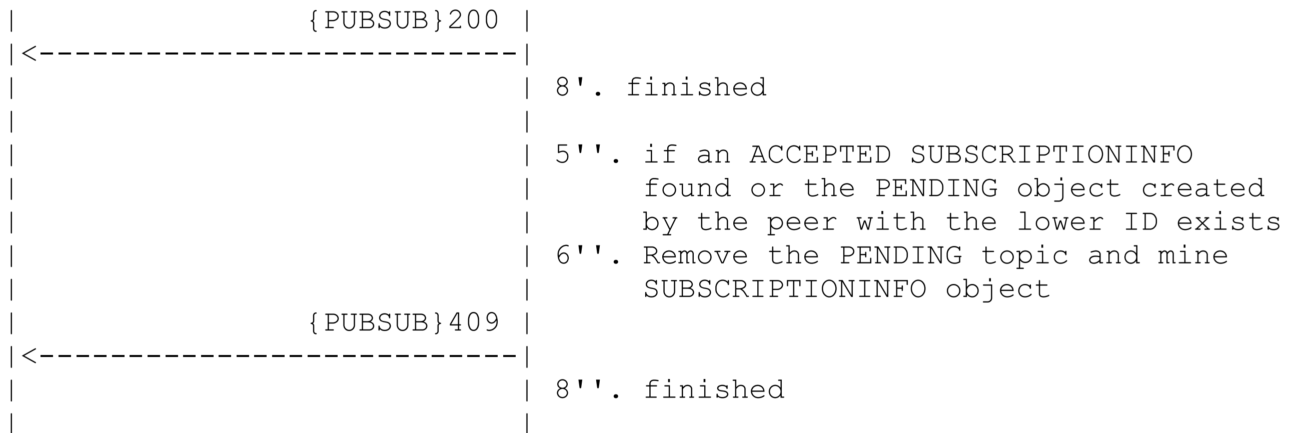
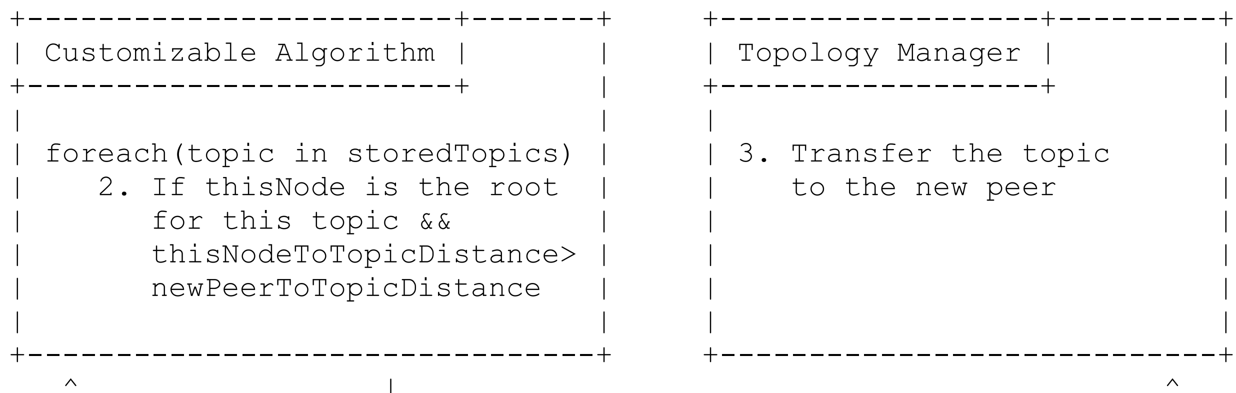



Figure 4.2.2: Create topic procedure in an unstructured network

### 4.2.2 Transfer topic

This operation is essential for the DHT-based networks. It MAY be performed when the publish-subscribe layer receives the information from the P2P layer, that it has accepted the other node's join request. In such case the node iterates through all the topics it stores information about. If it is the root for one of them and the distance between the new peer's ID and the topic ID is smaller than the distance calculated for this node, it sends a create topic request inside the P2P insert message. Before sending the request, the previous root MAY also add the list of its direct children to it. All of the procedures associated with the topic transfer are performed by the Topology Manager component, as it is responsible for retaining the appropriate structure. For instance in the star topology, all the request originator's children must be passed to the new root as well. If some Customizable Algorithm component is designed exclusively for the unstructured networks, it MAY choose not to perform this procedure or use a different measure than the distance between identifiers to determine, whether the specified topic should be transfered. The brief description of the whole procedure is shown in the figure 4.2.2.1.



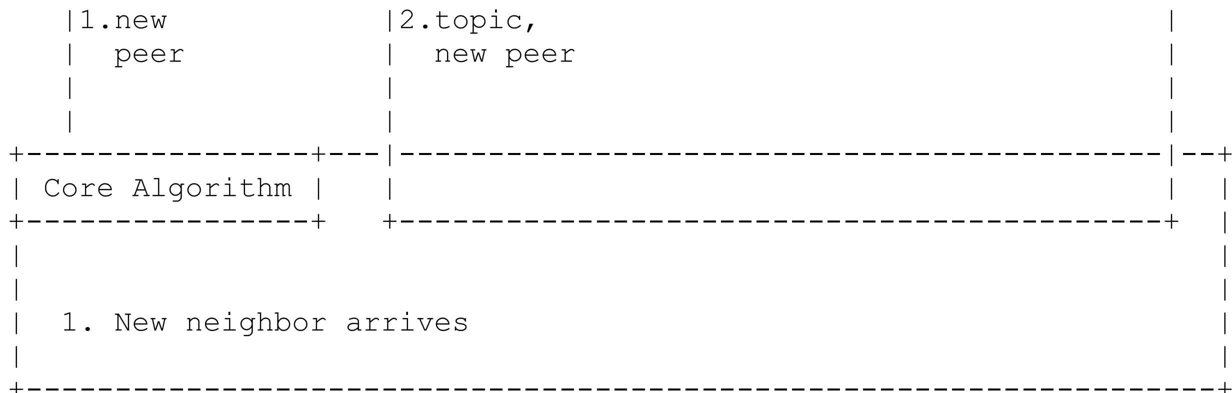



Figure 4.2.2.1: Overview of the topic transfer procedure

The details of the transfer topic procedure performed by the Topology Manager component are described in the figures 4.2.2.2 (for the star topology) and 4.2.2.3 (for the multicast tree).

```
(a)         Previous root                New root
                 |                           |
                 |                           | 1. Add the nodes from
                 |                           |    the list inside the
                 |                           |    transfer request to
                 |                           |    the topic
                 |                           |    subscription list
                 |               {PUBSUB}200 |
                 |<--------------------------|
                 |                           |


(b)         Previous root's
                child                Previous root     New root
                 |                          |               |
                 |                          |  {PUBSUB}200  | Accepting
                 |     1. Set parent to     |<--------------| topic
                 |        the new root      |               | transfer
                 |     2. Send keep-alive   |               |
                 |        indication        |               |
                 |        to inform the     |               |
                 |        child about the   |               |
                 |        parent's change   |               |
                 |                          |               |
                 |      {PUBSUB}keep-alive  |               |
                 |<-------------------------|               |
 1. Check if this |                          |               |
    message was sent|     3. Remove the child |               |
    by my parent  |        from the topic    |               |
    (otherwise    |        subscription list|               |
    discard it)   |     4. If this node is   |               |
```

```
 2. Set parent to   |        not the topic
    the one stated  |        subscriber itself
    in keep-alive   |        - remove the
 3. Update the cache|        topic
 4. Reset keep-alive|     5. Finished
    timer for this  |
    topic           |
                    |
```

Figure 4.2.2.2: Processing the transfer request(a) and response(b) by the star topology

```
  (a)         Previous root                  New root
                    |                           |
                    |                           | 1. Add the request
                    |                           |    originator to the
                    |                           |    topic subscription
                    |                           |    list
                    |               {PUBSUB}200 |
                    |<--------------------------|
                    |                           |


  (b)         Previous root's
                  child                 Previous root     New root
                    |                         |              |
                    |                         |  {PUBSUB}200 | Accepting
                    |     1. Set parent to    |<-------------| topic
                    |        the new root     |              | transfer
                    |     2. Send keep-alive  |              |
                    |        indication       |              |
                    |        to update the    |              |
                    |        child's cache    |              |
                    |                         |              |
                    |      {PUBSUB}keep-alive |              |
                    |<------------------------|              |
 1. Check if this   |                         |              |
    message was sent|     3. Finished         |              |
    by my parent    |
    (otherwise      |
    discard it)     |
 2. Set parent to   |
    the one sent    |
    in keep-alive   |
 3. Update the cache|
 4. Reset keep-alive|
    timer for this  |
    topic           |
                    |
```

Figure 4.2.2.3: Processing the transfer request(a) and response(b) by the multicast tree

### 4.2.3 Remove topic

To perform this operation, the node publishes the predefined REMOVE_TOPIC event among the topic subscribers. After that all the SUBSCRIPTIONINFO objects associated with this topic MUST be removed from the underlying P2P network.

### 4.2.4 Subscribe

To subscribe for a topic, node has to send a 'subscribe' request to the peer that is already participating in the appropriate topology structure. Determining the message destination is crucial for the whole procedure. In the DHT-based networks it is enough to encapsulate the publish-subscribe request in the resource object. Then it can be captured using the P2P protocol callbacks. The unstructured networks require performing the lookup procedure for the SUBSCRIPTIONINFO objects to identify the existing topology structure participants. After that the request may be sent directly to the specified node. Also the multicast trees are built using a different approach in the unstructured and DHT-based environment. In the first case, node accepts only the subscribers with the ID greater than its own one (figure 4.2.4.1). The nodes that do not fulfill these requirements are forwarded to the direct parent. Only the topic root accepts all the requests unless this is forbidden by the AC rules. Such an arrangement of the topology structure participants simplifies the recovery procedures after the direct parent's failure.

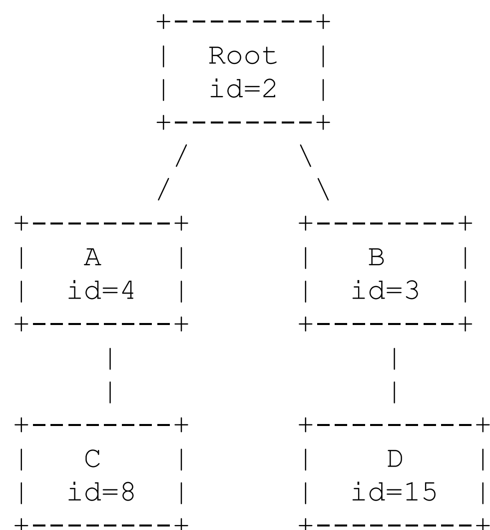



Figure 4.2.4.1: Multicast tree created on top of an unstructured

network





In the DHT-based network the nodes are arranged by the distance between their identifiers and the topic ID (figure 4.2.4.2).

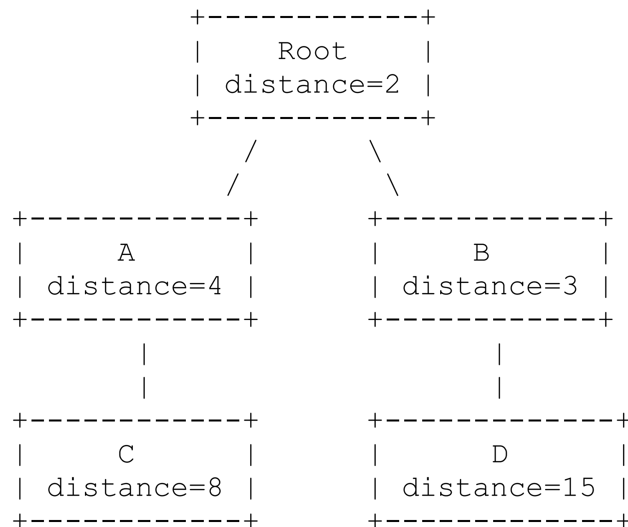



Figure 4.2.4.2: Multicast tree created on top of a DHT-based network

In the unstructured networks, after successfully completing the described procedure, the new subscriber must place its SUBSCRIPTIONINFO object in the underlying network. This is done only if it is a peer, as clients SHOULD NOT be responsible for accepting new subscribers. The node must also locally modify the AC rules for the notify operation, to indicate that only its direct parent is allowed to send event notifications to it. The standard subscribe request also contains the additional information, such as the index of the last received event. After accepting the new subscriber, its parent must send the requested set of the notify messages containing the historical events to it. Each subscription is valid only for a specified time, so the node needs to renew it periodically.

### 4.2.5 Unsubscribe

If the node has no more children in the topology structure and it is not the topic root, it simply sends an 'unsubscribe' request directly to its parent and removes its SUBSCRIPTIONINFO object from the overlay if it is necessary. If the node has more children, it also has to send the keep-alive indication to them. It is depicted in figure 4.2.5.1. The unsubscribe request sent to the node's parent contains the list of its children. Parent node has to add the new children to its own ones. Children nodes get keep-alive indication with the information about the new parent and modify their cache information (figure 4.2.5.2). If necessary, keep-alive indications are propagated down along the multicast tree to update other node's caches (the unsubscribing node can be stored in their grandparents cache). Note that after unsubscribe root node does not notify the

higher-layer about the received events anymore, but it is still able to forward appropriate messages to its children.

```
                            +------+
                            | Root |
                            +------+
                                |
                                |
                            +-------+
                        +-->|   A   |
          unsubscribe   |   +-------+
         children={C,E} |       |
                        |       |          keep-alive
                        |   +--\--/--+      parent=A
                        +---|  B\/   |----------+---+
                            |   /\   |          |   |
                            +--/--\--+          |   |
                            /        \          |   |
                           /          \         |   |
                      +-------+    +-------+    |   |
                      |   C   |    |   E   |<---+   |
                      +-------+    +-------+        |
                          ^                         |
                          |                         |
                          +-------------------------+
```

Figure 4.2.5.1: Multicast tree before the unsubscribe procedure

```
                                +------+
                                | Root |
                                +------+
                                    |
                                    |
                                +-------+
                                |   A   |
                                +-------+
                                  /    \
                                 /      \
   +----------+                 /        \                +----------+
   | Cache C: |         +-------+       +-------+         | Cache E: |
   | p={A}    |<--------|   C   |       |   E   |-------->| p={A}    |
   | g={ROOT} |         +-------+       +-------+         | g={ROOT} |
   | n={C,E}  |                                           | n={C,E}  |
   +----------+                                           +----------+
```

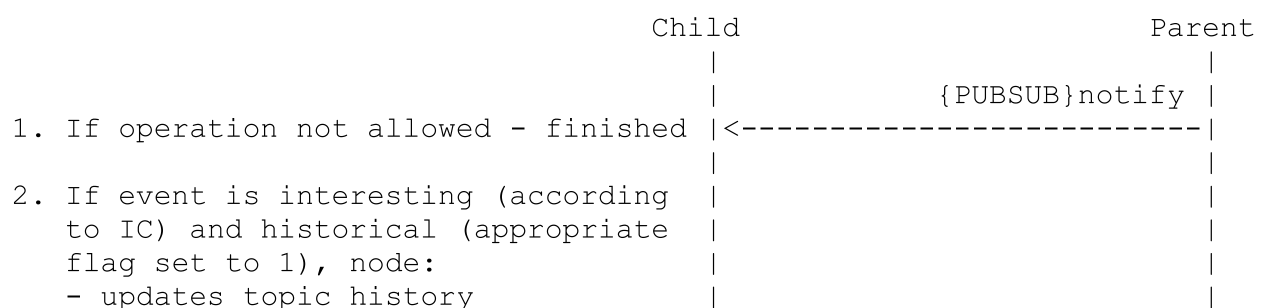
Figure 4.2.5.2: Multicast tree after the unsubscribe procedure

### 4.2.6 Publish

There are three types of the predefined events that can be published among the topic subscribers: AC_MODIFIED, TOPIC_REMOVED and CUSTOM. The first one is generated, when the access control rules are modified. The third one represents a custom event defined by the higher-layer application. The default Customizable Algorithm component requires all the publish messages to be delivered to the topic root, which accepts the request, sends the appropriate response to its originator and the set of notify indications to the direct children. This is done to avoid the situation, when for example AC rules have been modified, but the AC_MODIFIED event has not been propagated among all of the topic subscribers. In such case some of them may accept the events from the node which is not allowed to generate them anymore. In the default Customizable Algorithm component AC rules are defined so, that only the node's parent in the topology structure is allowed to send event notifications to it. Note that there are two situations, in which such indication may be received. The first one is when a new event is generated by some node. In this case the notification SHOULD be forwarded down along the multicast tree. However it might happen that the new subscriber receives the requested historical events after successfully completing the subscribe operation. In such case no further forwarding is needed. The node only updates its own topic history and invokes the higher-layer callback. The default Customizable Algorithm component also requires the node to be a topic subscriber to generate events.

### 4.2.7 Notify

After receiving a publish request, topic root propagates the event among its children using the notify message. Like all indications, notifications are send directly to the specified nodes. Figure 4.2.7.1 briefly describes the procedure performed after receiving such message. By default the Customizable Algorithm component defines AC rules so, that only the node's parent in the topology structure is allowed to send notifications to it. There are three predefined event types sent within the notify message(see section 4.1.4). All the received notifications MUST be forwarded to children unless they are historical.



```
     - if it is topic subscriber, not    |                            |
       just a forwarder invokes          |                            |
       higher-layer application callback |                            |
  3. Finished                            |                            |
                                         |                            |
  2'. If event is interesting and new    |                            |
      (appropriate flag set to 0),       |                            |
      node:                              |                            |
      - updates topic history            |                            |
      - forwards notification to         |                            |
        children (and invokes its own    |                            |
        callback if it is topic          |                            |
        subscriber)                      |                            |
  3'. Finished                           |                            |
                                         |                            |
```

Figure 4.2.7.1: Procedure performed after receiving event notification

### 4.2.8 Maintenance

Apart from the necessity to transfer a topic to the new root in some situations, there is also the nodes' failures problem that has to be handled by the Customizable Algorithm component. To address this issue, we define a keep-alive indication. Every node periodically sends this message to its direct children. If some peer is the other one's parent for several topics, it does not have to send a separate indication for each one of them. Such message contains the information about the current parent and the list of the 'backup nodes' to be placed in the node's cache for each topic. When the child detects its parent's failure, it uses the cache to resubscribe. If the cache size is set to 0 by the Algorithm Configurator component, the default Customizable Algorithm assumes, that the keep-alive message does not carry any additional information and treats the notify indication as a 'keep-alive', like Scribe does. This way, if events are frequently published for the specified topics, no additional traffic associated with the topology structure maintenance is involved. This optimization mechanism can be considered for the DHT-based networks, where the recovery procedure may be performed at a relatively low cost using the overlay-specific routing. In general there are three types of caches stored for each topic: parents cache, grandparents cache and neighbors cache. The capacities of the caches are the parameters k (for the parents and neighbors) and g (for the grandparents). For the node N at the level x, parents cache contains k nodes with the lowest id or the closest to the topic id from the level x-1. It may also contain the node's direct parent. The neighbor cache contains k nodes with the lowest id or the closest to the topic id from the level x, which have the same parent as N. The grandparents cache contains one node from each level between x-2 and x-1-g. This cache is stored in case of the multiple

nodes' failures in the highly unbalanced trees. When all the nodes from the parents cache fail to respond or the cache is empty - the node can try to send 'subscribe' request directly to its grandparent. All the nodes at the same level and in the same subtree have the same caches' contents. Nodes in the different subtrees have different neighbors caches' contents. All the entries in the parents and neighbors caches are sorted from the lowest id (or distance to the topic id) to the highest. In the grandparents cache the node from the highest level is stored at the lowest index. Examples of the topology structures and caches' contents for both the unstructured and DHT-based networks are shown in the figures 4.2.8.1 and 4.2.8.2.

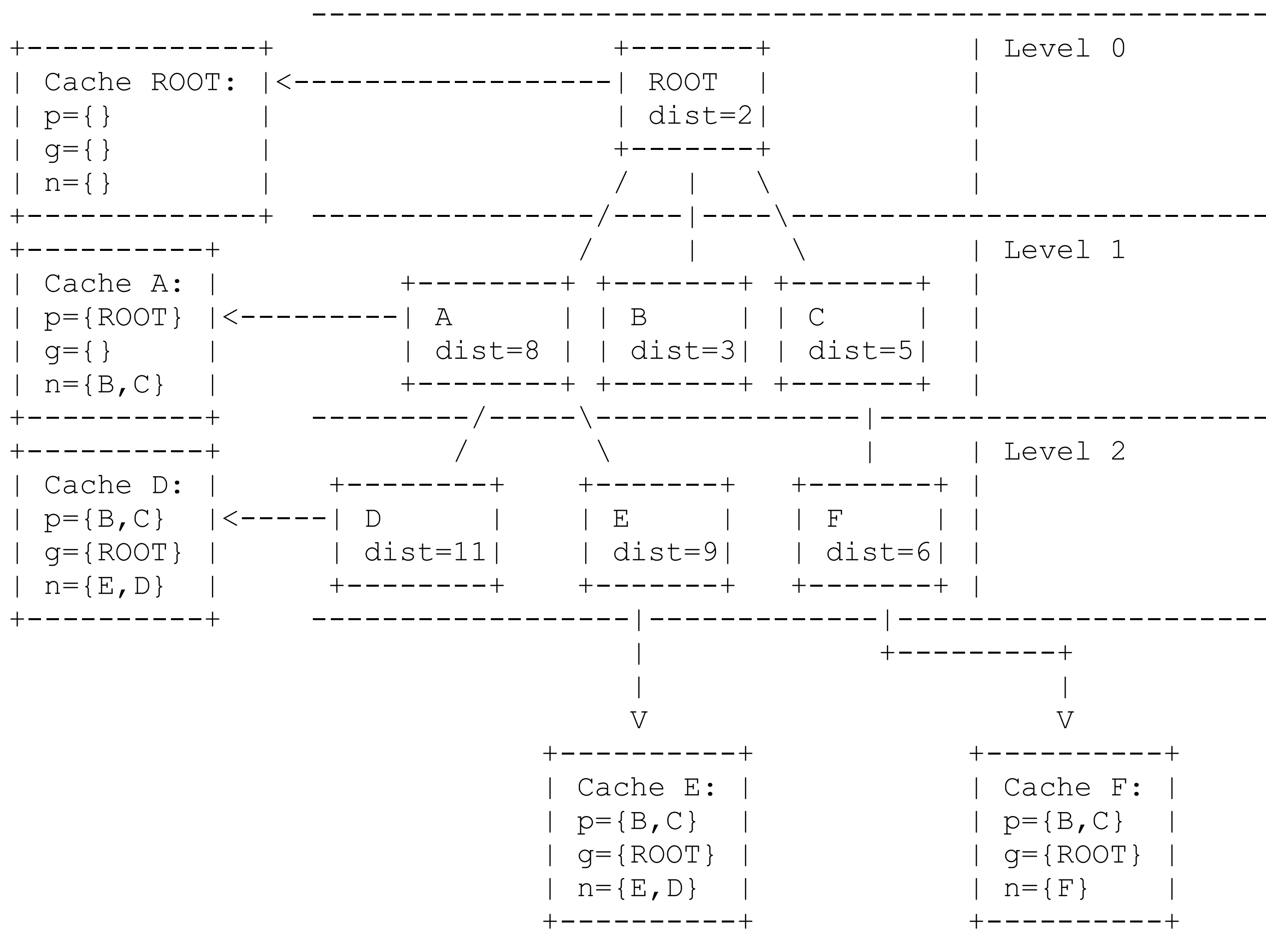
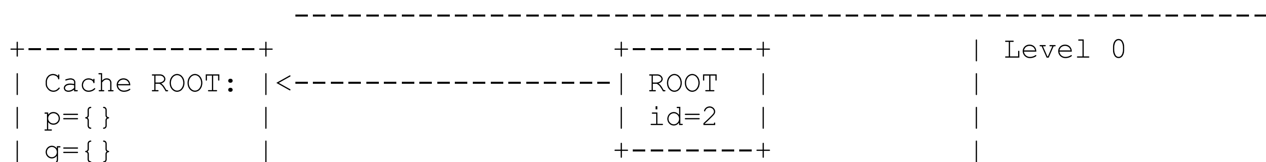



Figure 4.2.8.1: Caches' contents for k=2 and g=1 in the DHT-based network



```
 | n={}        |                   /   |   \           |
 +-------------+  ----------------/----|----\---------------------------
 +----------+                    /     |     \         | Level 1
 | Cache A: |          +--------+ +-------+ +-------+  |
 | p={ROOT} |<---------| A      | | B     | | C     |  |
 | g={}     |          | id=8   | | id=3  | | id=5  |  |
 | n={B,C}  |          +--------+ +-------+ +-------+  |
 +----------+     ---------/-----\---------------|----------------------
 +----------+             /       \              |     | Level 2
 | Cache D: |      +--------+    +-------+   +-------+ |
 | p={B,C}  |<-----| D      |    | E     |   | F     | |
 | g={ROOT} |      | id=11  |    | id=9  |   | id=6  | |
 | n={E,D}  |      +--------+    +-------+   +-------+ |
 +----------+     ------------------|-------------|---------------------
                                    |             +---------+
                                    |                       |
                                    V                       V
                               +----------+            +----------+
                               | Cache E: |            | Cache F: |
                               | p={B,C}  |            | p={B,C}  |
                               | g={ROOT} |            | g={ROOT} |
                               | n={E,D}  |            | n={F}    |
                               +----------+            +----------+
```

Figure 4.2.8.2: Caches' contents for k=2 and g=1 in an unstructured network

Figure 4.2.8.3 briefly describes the usage of the grandparents cache in case of the multiple nodes' failures.

```
                                 +------+
                            +--->| ROOT |
                            |    +------+
                            |       |
                            |       |
                            |    +-\--/-+
                            |    | A\/  |
                            |    |  /\  |
                            |    +-/--\-+
                            |       |    \
          possible path for |       |     \
              resubsciption |    +-\--/-+ +-\--/-+
                            |    | B\/  | | E\/  |
                            |    |  /\  | |  /\  |
                            |    +-/--\-+ +-/--\-+
                            |       |
                            |       |
                            |    +------+        +-------------+
                            +----|  C   |------->| Cache C:    |
```

```
                                  +------+        | p={B,E}      |
                                                  | g={A,ROOT}   |
                                                  | n={C}        |
                                                  +-------------+
```

Figure 4.2.8.3: Example of the grandparents cache usage
(algorithm parameters:  k=2 and g=2)

After discovering the direct parent's failure, node tries to send the 'subscribe' request to the the first node from its parents cache. If it does not respond,  it is removed from the cache, and the 'subscribe' request is sent to the next one. If all the nodes from the parents cache fail to respond, the grandparents cache is used. If the grandparents cache does not contain any useful information and the node's distance to the topic ID is greater than 0, than peer assumes that it participates in the DHT-based network and there may be some node which ID is closer to the topic ID than its own one. In such case it encapsulates the 'transfer topic' request inside the P2P insert message. If this message is received by some other node, it performs a standard topic transfer procedure. If the peer is already participating in the topology structure, than it only omits creating the new topic. If the distance between the topic ID and the peer ID is lower than 0, than node assumes, that it is participating in an unstructured network and cannot rely on the overlay routing algorithm. In such case the neighbor cache's contents are examined. If it does not provide any useful information or this node is the first one on the list, the subscriber performs the lookup procedure for the  SUBSCRIPTIONINFO objects associated with the same topic. If it finds any nodes with the ID lower than its own one, than it must send the subscribe message to the one with the lowest ID. If there are no such nodes in the P2P network, than it assumes that it is the new topic root and waits for the other nodes to send 'subscribe' requests to it. These procedures guarantee the cycles avoidance. Moreover they work both for the multicast tree and star topology.

### 4.2.9 Reliability

[TODO: history of events for temporary out-of-order issues, but not the long term ones - history length as algorithm parameter]

## 4.3 Message formats

All the message formats described in this section are used by the default Customizable Algorithm component. However they MAY be extended by the user-defined ones if it is necessary and registered by a different Algorithm Configurator object. In general there are three types of messages: requests, indications and responses. Node receiving the first one always sends response to the message originator. Both requests and responses contain the unique

identifier of the transaction, they are associated with. It is generated by the request originator, who uses it to find out, which operation is received response related to. Indications are messages which do not require any response as they are not associated with a pending operation.

### 4.3.1 Standard header

Every publish-subscribe message is preceded by the following header:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|IPver|                     Source IP                        //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                         |        Destination IP           //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                         |           Source port           //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                         |        Destination port         //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                         |       Type          |           //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                   Source user name length   |             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                       Source user name                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Source peer id length                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source peer id                       //
+                                                             +
//                                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Destination user name length                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Destination user name                  //
+                                                             +
//                                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Destination peer id length                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination peer id                     //
+                                                             +
//                                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Topic id length                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Topic id                           //
+                                                             +
//                                                            |
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

IP version (3 bits): IP version number, 4 or 6
Source IP (32 or 128 bits): IP address of message sender

Destination IP (32 or 128 bits): IP address of message receiver

Source port (32 bits): Message sender's port number

Destination port (32 bits): Message receiver's port number

Type (8 bits): Publish-subscribe message type

Source user name length (32 bits): Length of the user's unhashed id

Source user name (undefined): User's unhashed id

Source peer id length (32 bits): Length of the peer id

Source peer id (undefined): Peer id

Destination user name length (32 bits): Length of user's unhashed id

Destination user name (undefined): User's unhashed id

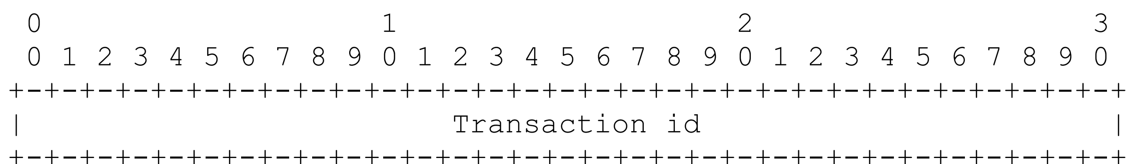
Destination peer id length (32 bits): Length of the peer id

Destination peer id (undefined): Peer id

Topic id length (32 bits): Length of the topic id this message is associated with
Topic id (undefined): Topic id this message is associated with

#### 4.3.2 Standard request header

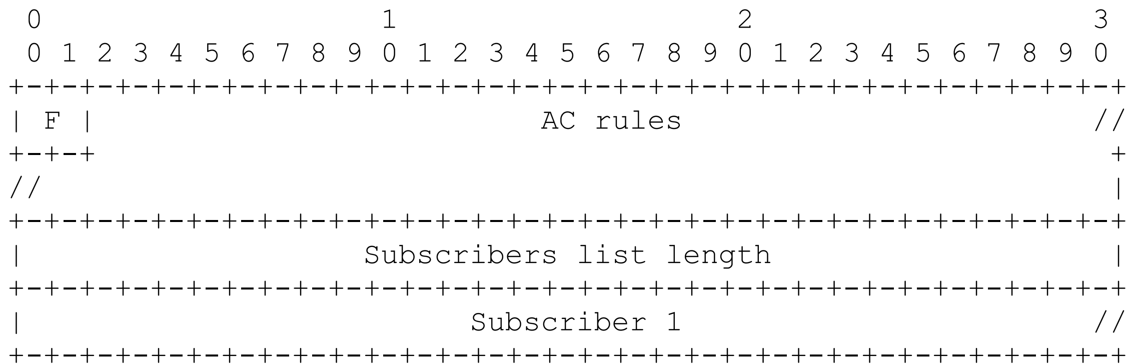
Apart from the type-specific information, every request contains the following header:


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Transaction id                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Transaction id (32 bits): Identifier of the transaction this operation is associated with

**4.3.3 Create topic**

Create topic message can be send to actually create a new topic, or to transfer an existing one to the new root. The node recognizes the requested operation by checking the value of the special flag inside the header.




F (2 bits): Flag representing the requested operation type. Currently only two types are defined (figure 4.3.3.1).

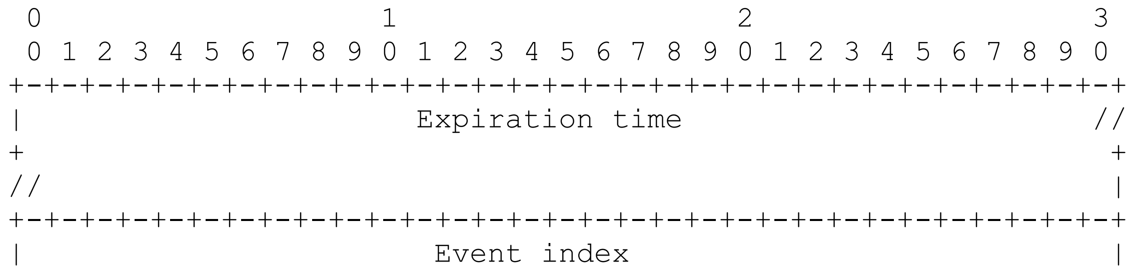
AC rules (undefined): Access control rules defined for the topic (see section 4.4.1 for details).

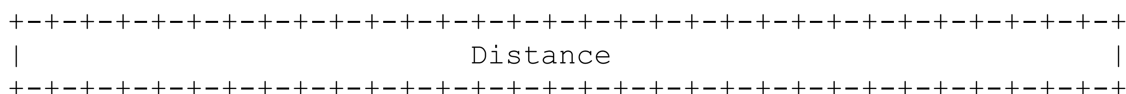
Subscriber list length (32 bits): Length of the list containing the subscribers to be added after the topic creation.

| Value | Description |
|---|---|
| 0 | Creates new topic |
| 1 | Transfers an existing topic to new root |

Figure 4.3.3.1

**4.3.4 Subscribe**






Expiration time (64 bits): Time, after which node will have to resubscribe

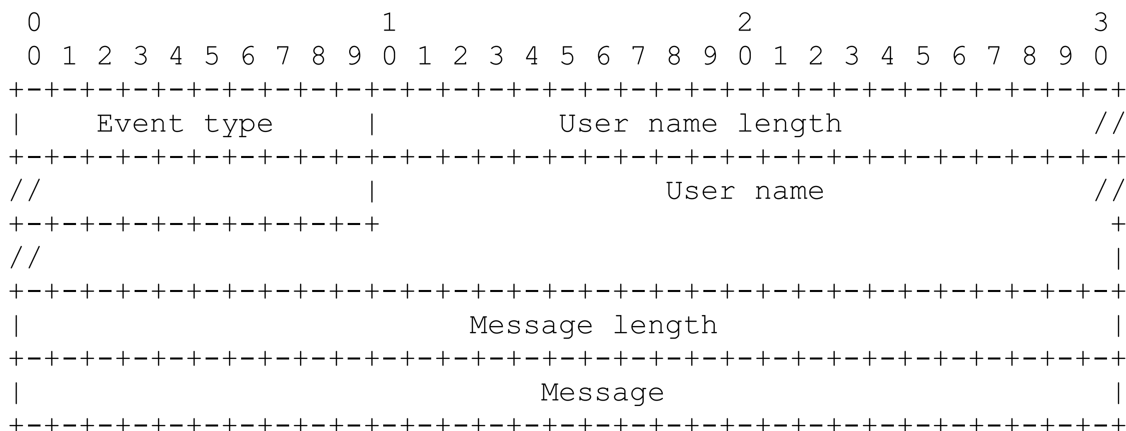
Event index (32 bits): Index of the last received event from the topic history

Distance (32 bits): Distance between the subscriber id and topic id

### 4.3.5 Unsubscribe

Currently there are no type-specific values for this message. It contains only the standard request header.

### 4.3.6 Publish




Event type (8 bits): Field indicating, whether this is the REMOVE_TOPIC, AC_MODIFIED or CUSTOM event

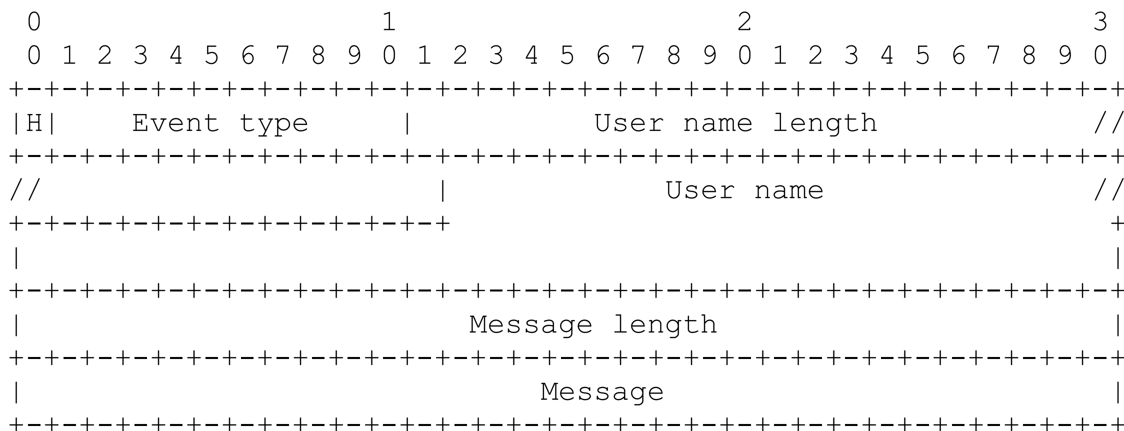
User name length (32 bits): Length of the event publisher's name

User name (undefined): Event publisher's name

Message length (32 bits): Length of the message encapsulated in the CUSTOM event

Message (undefined): User-defined message encapsulated in the CUSTOM event or the modified AC rules

### 4.3.7 Notify



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|H|    Event type     |          User name length           //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                      |           User name               //
+-+-+-+-+-+-+-+-+-+-+-+-+                                     +
|                                                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Message length                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Message                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

H (1 bit): Flag indicating whether this event is the new one (value MUST be 0 in this case), or the old one from the topic history (value MUST be 1)

Event type (8 bits): Field indicating, whether this is a REMOVE_TOPIC, AC_MODIFIED or CUSTOM event

User name length (32 bits): Length of the event publisher's name

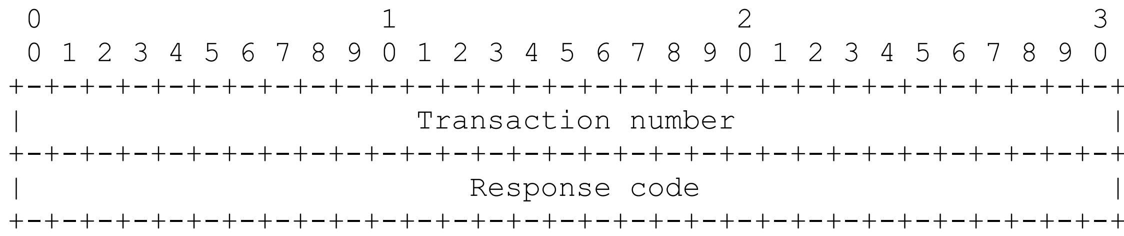
User name (undefined): Event publisher's name

Message length (32 bits): Length of the message encapsulated in the CUSTOM event

Message (undefined): Message encapsulated in the CUSTOM event

### 4.3.8 Standard response header

Every response, except the response code, contains a unique transaction identifier. The request originator uses it, to find out which operation this response is related to.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Transaction number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Response code                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Transaction number (32 bits): Identifier of the transaction, this response is associated with

Response code (32 bits): Response codes are described in section 4.7

**4.3.9 Subscribe response**

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          AC rules                           //
+                                                               +
//                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

AC rules (undefined): AC rules encoding is described in section 4.4.1

**4.3.10 Keep-alive**

Parent (undefined): Subscriber object containing node's parent in the topology structure

Neighbors (undefined): List of the subscriber objects to put into the neighbors cache. Its length depends on the k parameter.

Parents (undefined): List of the subscriber objects to put into the parents cache. Its length depends on the k parameter.

Grandparents (undefined): List of the subscriber objects to put into the grandparents cache. Its length depends on the g parameter.

**4.4 Objects formats**

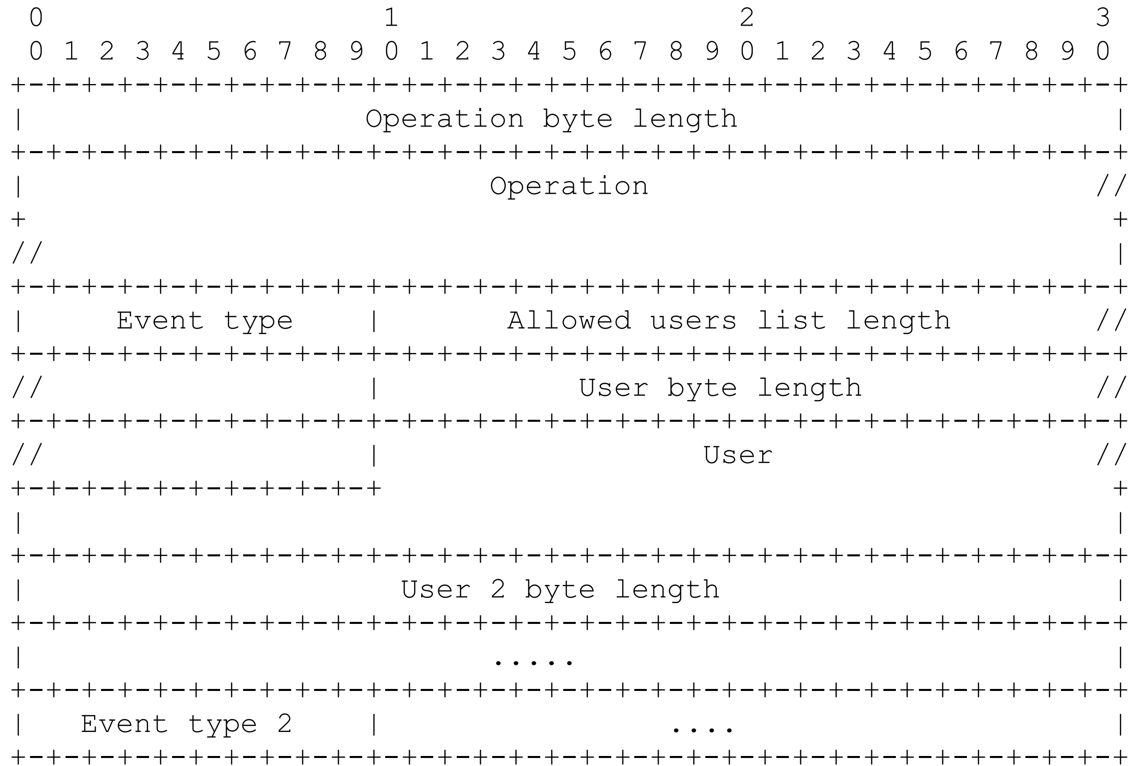
**4.4.1 AC rules**

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     List of rules length                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         List of rules                       //
+                                                               +
//                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

List of rules length (32 bits): Number of the rules on the list

List of rules (undefined): List of the AC rules (see section 4.4.2 for details)

**4.4.2 Rule**



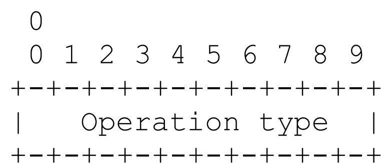
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Operation byte length                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Operation                         //
+                                                             +
//                                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Event type    |      Allowed users list length         //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                  |           User byte length             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                  |                 User                   //
+-+-+-+-+-+-+-+-+-+-+                                         +
|                                                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     User 2 byte length                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           .....                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Event type 2    |                ....                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Operation (undefined): Operation, this rule is associated with

Event type (8 bits): Particular event within an operation (at least one event is mandatory)

User (undefined): User allowed to perform specified operation

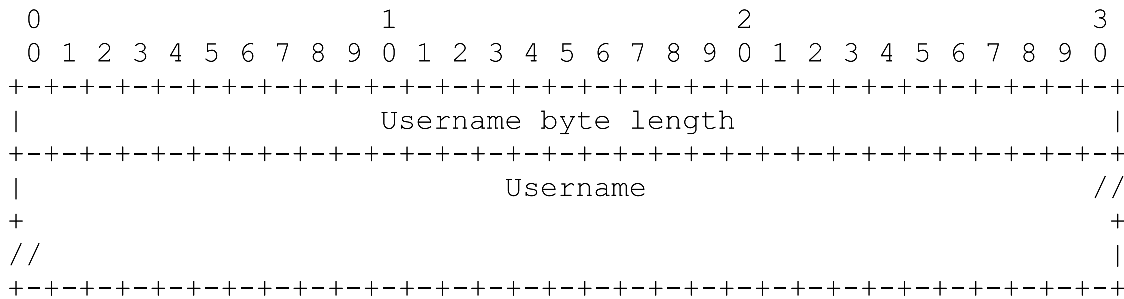
**4.4.3 Operation**



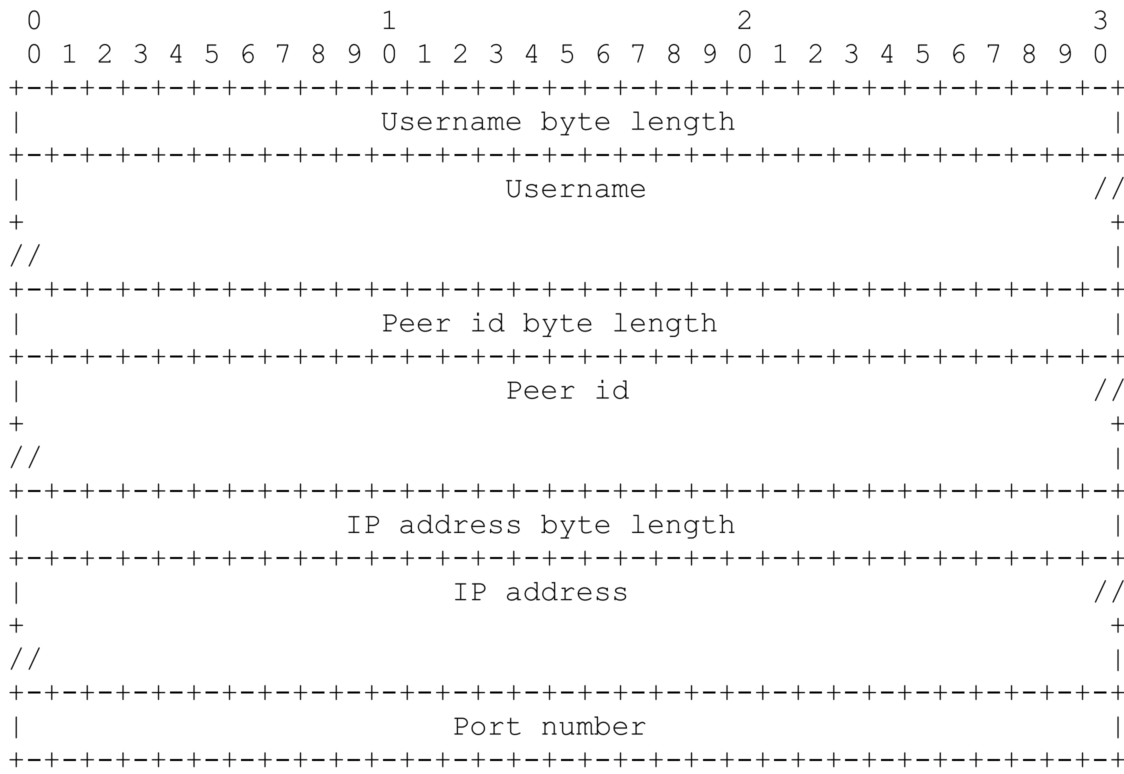
```
 0
 0 1 2 3 4 5 6 7 8 9
+-+-+-+-+-+-+-+-+-+-+
|   Operation type  |
+-+-+-+-+-+-+-+-+-+-+
```

Operation type (8 bits): Type of an operation, this rule is associated with. Currently defined values correspond with the message types (from 1 to 6) described in section 4.5.

**4.4.4 User**




#### 4.4.5 Subscriber




### 4.5 Message types

| Value | Message type | Indication/Request/Response |
|---|---|---|
| 0 | Standard response | Res |
| 1 | Create topic | Req |
| 2 | Subscribe | Req |
| 3 | Unsubscribe | Req |
| 4 | Publish | Req |

| | | |
|---|---|---|
| 5 | Notify | Ind |
| 6 | Keep-alive | Ind |

**4.6 Event types**

| Type | Description |
|---|---|
| 0 | Topic removed |
| 1 | Access Control Rules modified |
| 2 | Custom event defined by higher-layer application |

**4.7 Response codes**

Response codes are inspired from HTTP.

| Code | Description |
|---|---|
| 200 | Operation successful |
| 403 | Operation forbidden due to AC rules |
| 404 | Operation cannot be completed, because the topic or subscriber it is associated with does not exist |
| 409 | Create topic operation cannot be completed because the specified topic already exists |

**4.8 Security**

[TODO: Verifying user identity for AC rules, etc.]

**5. Integrating publish-subscribe with P2PP/RELOAD**

**5.1 Extending P2PP/RELOAD interfaces**

There are several major differences between some of the P2PP and RELOAD requests, we had to consider. P2PP publish allows application to insert only one object per request. Single RELOAD store request MAY contain several objects with the same resource-id, but different kind-id's or data models. It is the same with the P2PP lookup and RELOAD fetch requests. To provide the generic interfaces for both protocols we decided to use object lists, not single objects in all callbacks associated with the requests processing. Due to the terminology differences between both protocols we use 'insert' for the P2PP publish, and RELOAD store request, and 'lookup' for the P2PP lookup and RELOAD fetch request where distinguishing one from another is not essential.

### 5.1.1 Callbacks

We propose adding several callback methods to the P2PP/RELOAD interfaces. They can be used by applications built on top of it not only for gathering information about the received messages, but also for passing data to the overlay layer. Parameters marked as [in] are the ones that are passed to the higher layer by P2PP/RELOAD. These marked as [out] are returned to the overlay by the application layer.

[TODO: Decide what about applications which do not want to use callbacks - default implementation returning true shouldn't be a problem, when performance is considered]

[TODO: Decide, whether signature/certificate checking should be performed before or after the callback invocation (in the second case application layer would have to update the object's signature) and whether to invoke callbacks for example if the RELOAD layer already knows that the signature is invalid]

```
boolean onDeliverRequest([in]Request message,
                         [in/out]List<GeneralObject> objectList)
```

@param message Received P2PP/RELOAD request.

@param objectList Value interpretation is message type dependent.

@return Value interpretation is message type dependent. In general 'true' means 'continue standard processing', and 'false' means 'perform custom operation'.

Usually invoked directly after receiving a request and determining that this node is message destination. In case of lookup requests - after searching for the objects in the peer's resource table (figure 5.1.1.1). If node stores the requested objects there, they are passed to the higher layer using 'objectList' parameter. Higher-layer application may decide to send these objects within the lookup response, or pass a different ones using the same parameter.

After the callback invocation P2PP/RELOAD checks the returned value. If this value is 'true' it simply returns objects found in the resource table in response (or 'NOT FOUND' message if none of the requested objects is stored there). Otherwise response message contents depend on the second callback parameter's value (objects defined by higher layer are returned to the request originator). This MAY be used by the application built on top of P2PP/RELOAD for example to create different 'views' of the node's resource table, depending on the lookup request originator. Higher layer MAY hide object's value from some users or send different values to them.

Similar procedure is performed after receiving an insert request (figure 5.1.1.2) although returned value interpretation is slightly different. When this value is 'true', objects encapsulated inside the insert request are simply put into the peer's resource table. Otherwise either different objects are inserted or only P2PP/RELOAD response is sent.

For both insert and lookup requests application layer MAY modify object list elements' values or replace them with null. It MUST NOT modify resource id, type or data model. For all the request types it also MUST NOT alter the object list size.

```
                                                             Message
                   Receiver                                 Originator
 +-------------------------------------------------+          +------+
 |                                                 |          |      |
 |                                          RELOAD/|          |RELOAD|
 |              Application                   P2PP |  lookup  |/P2PP |
 |                   |                        |    |  request |  |   |
 |                   |                        |<-----------------|   |
 |                   |          1. Fill object |    |          |  |   |
 |                   |             list (if    |    |          |  |   |
 |                   |             some        |    |          |  |   |
 |                   |             requested   |    |          |  |   |
 |                   |             object isn't|    |          |  |   |
 |                   |             stored here |    |          |  |   |
 |                   |             - add null  |    |          |  |   |
 |                   |             at this     |    |          |  |   |
 |                   |             position)   |    |          |  |   |
 |                   |                        |    |          |  |   |
 |                   |        onDeliverRequest |    |          |  |   |
 |                   |       (request, objects)|    |          |  |   |
 |                   |<-----------------------|    |          |  |   |
 |                   |                        |    |          |  |   |
 | 1. continue       |                        |    |          |  |   |
 |    standard       | true, objects          |    |          |  |   |
 |    processing     |----------------------->|    |          |  |   |
 |                   |                        |    |          |  |   |
 |                   |          1. return     |    |          |  |   |
 |                   |             standard   |    |          |  |   |
```

```
  |                   |            response    |   |           |  |   |
  |                   |            containing  |   |           |  |   |
  |                   |            objects     |   |           |  |   |
  |                   |            found in    |   | standard  |  |   |
  |                   |            resource    |   | response  |  |   |
  |                   |            table       |----------------->|   |
  |                   |         2. finished    |   |           |  |   |
  |                   |                        |   |           |  |   |
  |                   |                        |   |           |  |   |
  | 1'. modify object |                        |   |           |  |   |
  |     list (if any  |                        |   |           |  |   |
  |     of stored     |                        |   |           |  |   |
  |     objects needs |                        |   |           |  |   |
  |     to be hidden  |                        |   |           |  |   |
  |     from lookup   |                        |   |           |  |   |
  |     request       |                        |   |           |  |   |
  |     originator    |                        |   |           |  |   |
  |     - set it to   |                        |   |           |  |   |
  |     null)         |                        |   | modified  |  |   |
  |                   | false, modifiedObjects |   | response  |  |   |
  |                   |----------------------->|----------------->|   |
  |                   |                        |   |           |  |   |
  |                   |                        |   |           |  |   |
  |                                                |           |      |
  +------------------------------------------------+           +------+
                Figure 5.1.1.1: Node receives lookup request


                                                               Message
                    Receiver                                  Originator
  +------------------------------------------------+           +------+
  |                                                |           |      |
  |                                         RELOAD/|           |RELOAD|
  |               Application                 P2PP |  insert   |/P2PP |
  |                   |                        |   |  request  |  |   |
  |                   |                        |<-----------------|   |
  |                   |         1. Fill object |   |           |  |   |
  |                   |            list with   |   |           |  |   |
  |                   |            values from |   |           |  |   |
  |                   |            resource    |   |           |  |   |
  |                   |            table (if   |   |           |  |   |
  |                   |            object with |   |           |  |   |
  |                   |            specified   |   |           |  |   |
  |                   |            key and of  |   |           |  |   |
  |                   |            specified   |   |           |  |   |
  |                   |            type isn't  |   |           |  |   |
  |                   |            already     |   |           |  |   |
  |                   |            stored here |   |           |  |   |
  |                   |            - add null  |   |           |  |   |
  |                   |            at this     |   |           |  |   |
```

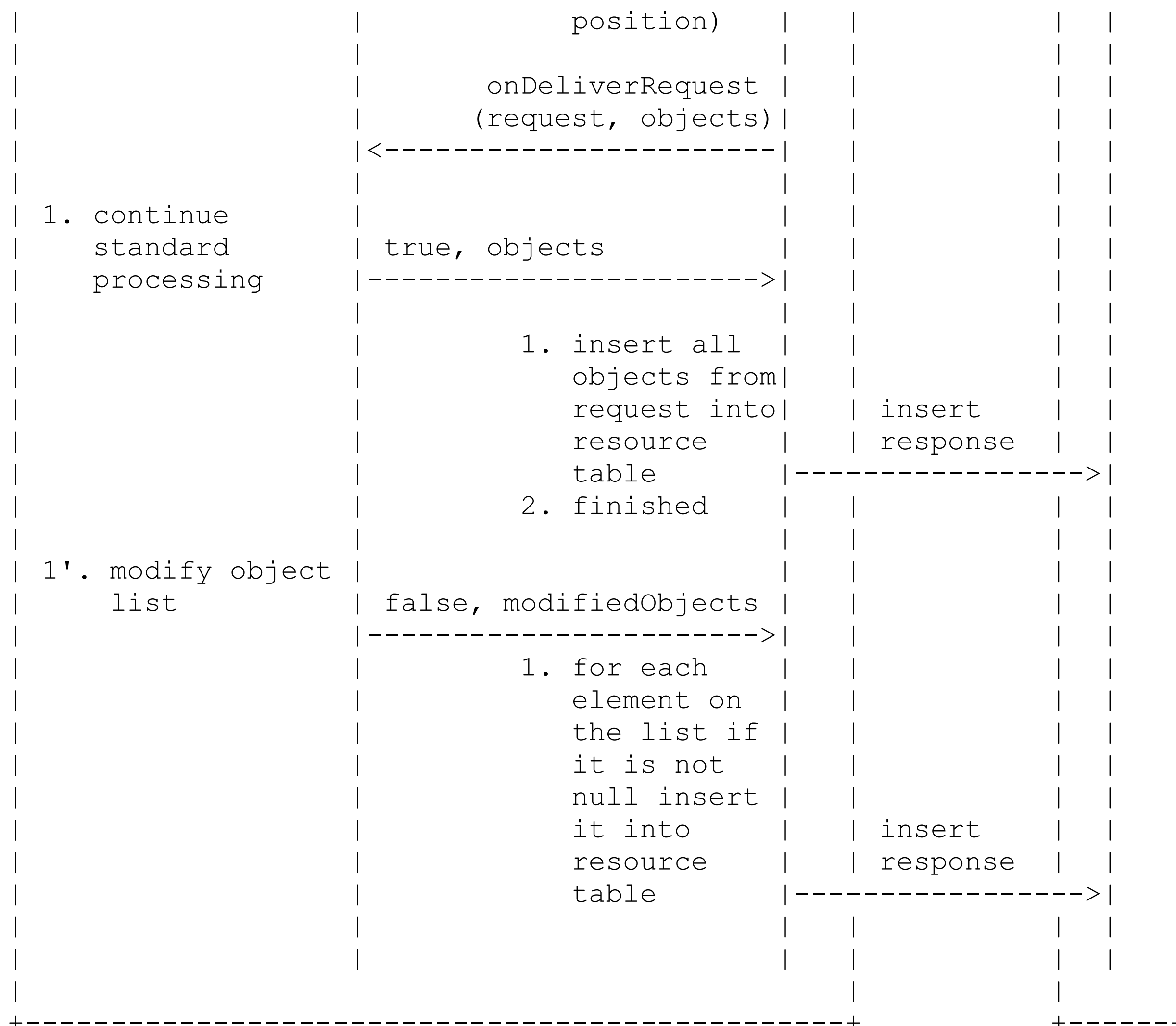



Figure 5.1.1.2: Node receives insert request

```
boolean onForwardingRequest([in]Request message,
                            [in/out]List<GeneralObject> objectList)
```

@param message Received P2PP/RELOAD request.

@param objectList Value interpretation is message type dependent.

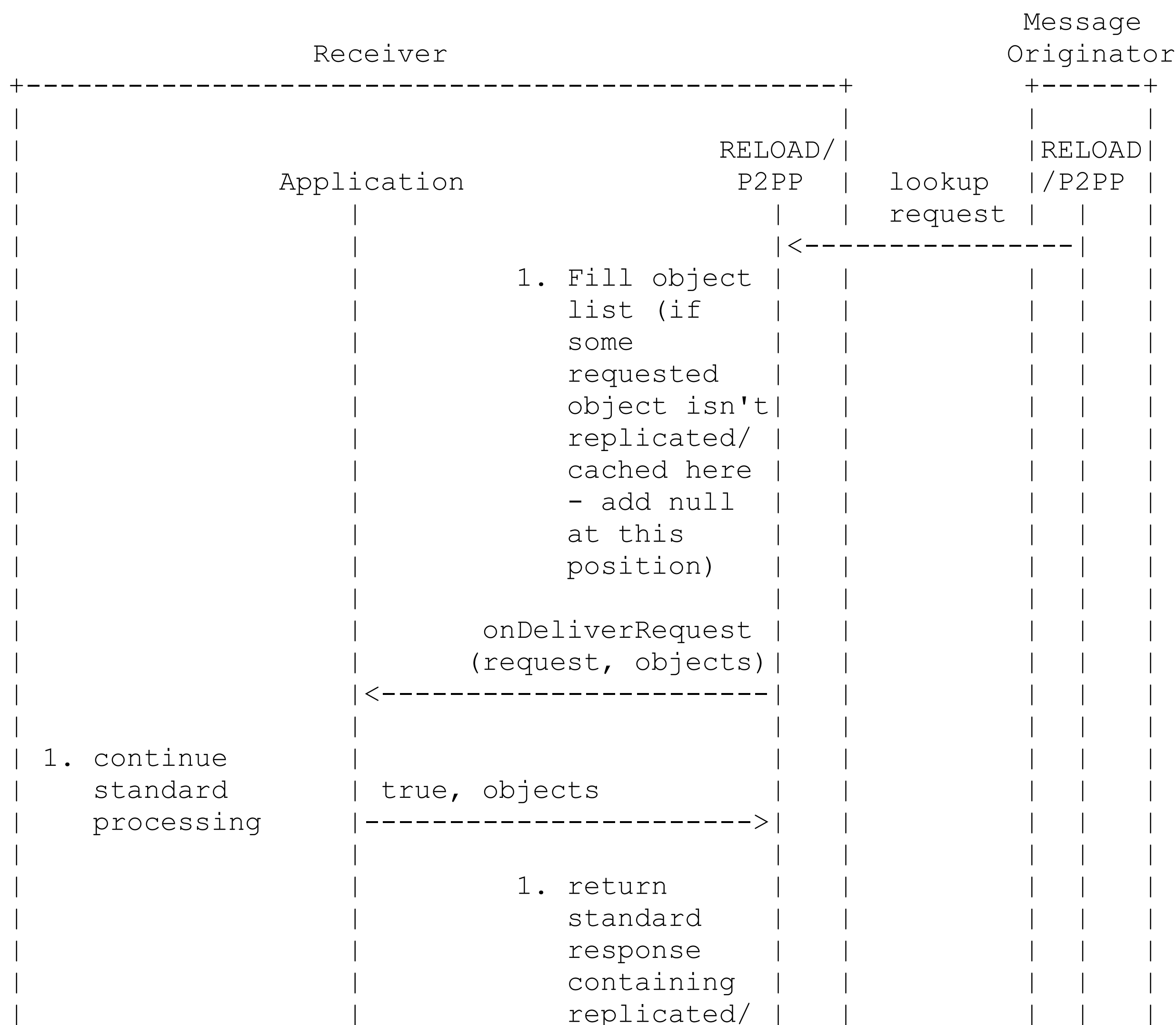
@return Value indicating, whether to forward or discard the message. In case of discarding - node has to send P2PP response. Otherwise request sender will try to retransmit its message and finally assume unresponsive peer's failure.

Usually invoked directly after receiving a P2PP request (and determining that this node is not the message destination), before forwarding it to other node or sending 302 response. In case of lookup requests (figure 5.1.1.3) - after checking, if the requested object is not for example cached (or replicated) here. If such object

is found, P2PP/RELOAD passes it to the higher layer using the 'objectList' parameter. Higher-layer application may decide to send this object within the P2PP lookup response, or pass a different one using the same parameter. After the callback invocation P2PP/RELOAD checks the returned value. If this value is 'true' it simply forwards the message or returns the cached/replicated objects in a standard response. Otherwise the message is discarded, and P2PP/RELOAD response containing objects defined by the higher layer is sent.

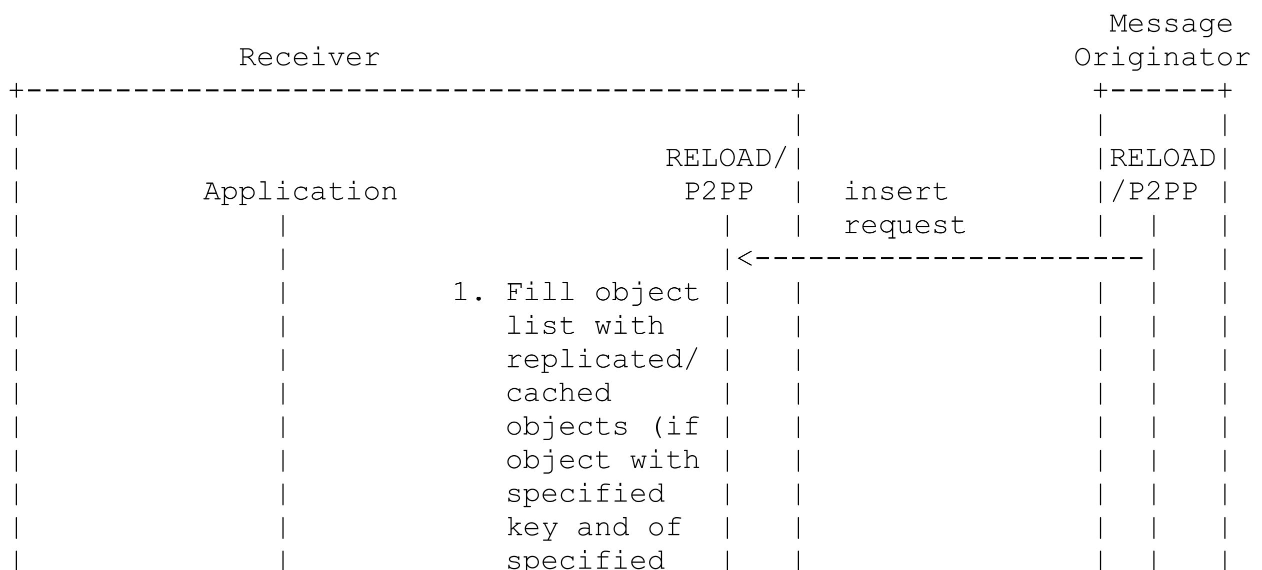
Figure 5.1.1.4 shows the procedure performed in case of receiving insert request. If the returned value is 'true', the message is forwarded. Otherwise node discards the message and sends insert response to its originator.

For both insert and lookup requests application layer MAY modify the object list elements' values or replace them with null. It MUST NOT modify the resource id, type or data model. For all request types it also MUST NOT alter the object list size.



```
  |                   |            cached      |   |          |  |   |
  |                   |            objects,    |   |          |  |   |
  |                   |            forward the |   |          |  |   |
  |                   |            message or  |   |          |  |   |
  |                   |            send        |   | standard |  |   |
  |                   |            redirect    |   | response |  |   |
  |                   |            response    |---------------->|   |
  |                   |         2. finished    |   |          |  |   |
  |                   |                        |   |          |  |   |
  |                   |                        |   |          |  |   |
  | 1'. modify object |                        |   |          |  |   |
  |     list (if any  |                        |   |          |  |   |
  |     of stored     |                        |   |          |  |   |
  |     objects needs |                        |   |          |  |   |
  |     to be hidden  |                        |   |          |  |   |
  |     from lookup   |                        |   |          |  |   |
  |     request       |                        |   |          |  |   |
  |     originator    |                        |   |          |  |   |
  |     - set it to   |                        |   |          |  |   |
  |     null)         |                        |   |          |  |   |
  |                   | false, modifiedObjects |   |          |  |   |
  |                   |----------------------->|   |          |  |   |
  |                   |         1. send        |   |          |  |   |
  |                   |            response    |   |          |  |   |
  |                   |            containing  |   | modified |  |   |
  |                   |            modified    |   | response |  |   |
  |                   |            objects     |---------------->|   |
  |                   |         2. finished    |   |          |  |   |
  |                   |                        |   |          |  |   |
  |                                                |          |      |
  +------------------------------------------------+          +------+
```

Figure 5.1.1.3: Node receives lookup request



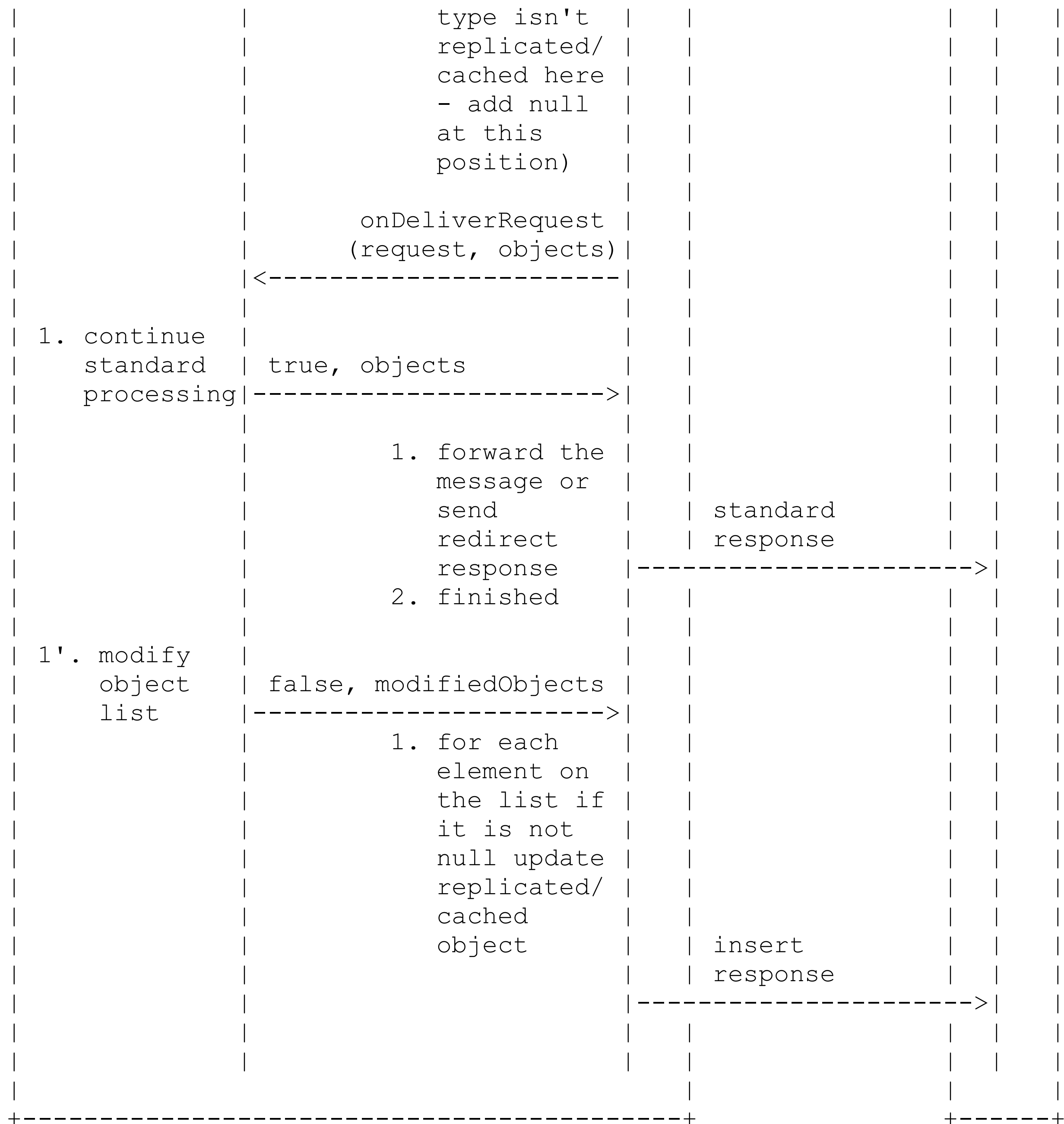



Figure 5.1.1.4: Node receives insert request

boolean onDeliverResponse([in]Response message)

@param message P2PP/RELOAD response.

@return Value indicating whether to continue the standard P2PP/RELOAD processing or not.

Invoked after receiving P2PP/RELOAD response. The returned value interpretation is operation type dependent. For instance if it is

'false' for the insert response, it means, that the specified object MUST NOT be republished. This is useful, if it was only used to encapsulate some higher-layer-defined message.

void onNeighborJoin(PeerInfo newNode, int nodeType)

@param newNode Object containing such information as ID, IP address, and port number.

@param nodeType Value indicating, whether new node is a client, peer, peer acting as bootstrap server, etc.

Invoked after accepting other node's join request and successfully sending OK reply to it (figure 5.1.1.5).

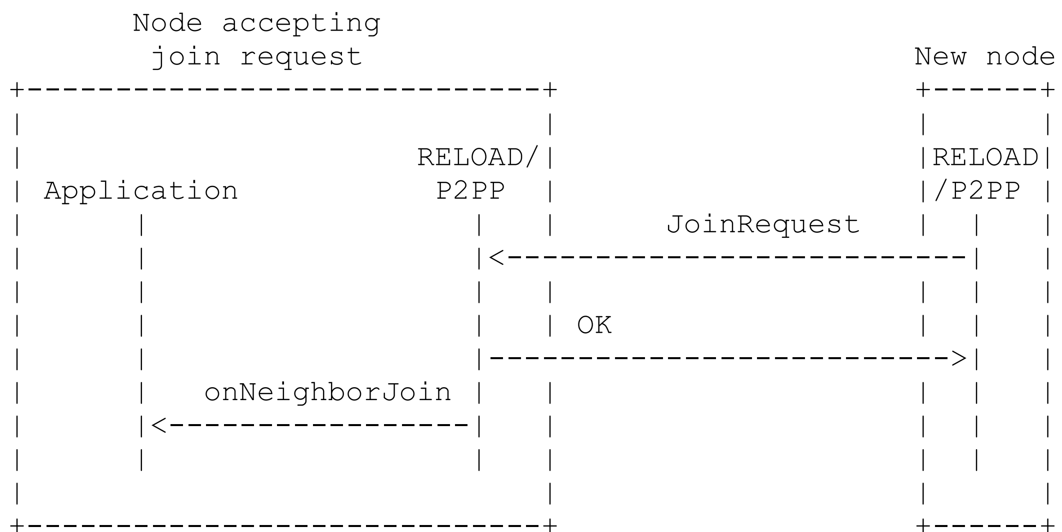



Figure 5.1.1.5

void onNeighborLeave(PeerInfo removedNode, int nodeType)

@param removedNode Object containing such information as ID, IP address, and port number.

@param nodeType Value indicating, whether new node is client, peer, peer acting as bootstrap server, etc.

Invoked after removing a neighbor from the neighbor table.

### 5.1.2 Objects

We propose creating a MESSAGE object type. This object could be used by the higher-layer applications to encapsulate its own messages

within an insert request and make use of the overlay-specific routing. In the DHT-based networks this enables sending a message even if the exact destination's peer ID is unknown (we only know, that it should be the closest one to some given object key). It also guarantees that the message will not be routed to clients (as they are not responsible for storing objects).

This object for P2PP could be described as follows:

Type: MESSAGE
Subtype: value indicating protocol
Message: raw message bytes

Below we propose the MESSAGE object's format using RELOAD semantics:

```
Kind-Id: MESSAGE
Data Model: single value
Value: value indicating protocol
       raw message bytes
```

The received MESSAGE objects do not necessarily have to be stored in the resource tables. Application built on top of P2PP/RELOAD MAY prevent it by capturing insert messages containing such objects using previously described callbacks and making overlay layer discard them. After that, the request originator SHOULD also use the previously described onDeliverResponse callback to prevent P2PP/RELOAD layer from republishing the object and thus resending the higher-layer message encapsulated in it.

### 5.1.3 API

We propose adding a method for calculating distance between the two given keys according to the overlay-specific metrics and hash algorithm. In our algorithm we use it for creating and maintaining the topology structure. This method could be defined as follows:

```
int getDistance(String key1, String key2)
```

@param key1, key2 Method calculates the distance between these keys.

@return In the DHT-based networks returned value is greater or equal
        to 0. In unstructured ones method always returns -1.

Publish-subscribe protocols built on top of the unstructured overlays, based for instance on random walks, may also require access to the routing information. This is why we propose two methods that provide it:

RoutingTable getRoutingTable()

@return Node's routing table.

NeighborTable getNeighborTable()

@return Node's neighbor table.

### 5.2 Usage of the extension

#### 5.2.1 Objects

Apart from the generic MESSAGE object we also propose to add one publish-subscribe-specific type called  SUBSCRIPTIONINFO. It will be placed in network to inform other nodes about the topic existence, and help them join the multicast tree. This is essential for the unstructured overlays.

For P2PP this object can be defined as follows:

```
Type = SUBSCRIPTIONINFO
Status = PENDING/ACCEPTED
Peer = PeerInfo object containing peer ID, IP address and port number
       for the incoming publish-subscribe messages.
```

Below we propose the SUBSCRIPTIONINFO object's format using RELOAD semantics:

```
Kind-Id: SUBSCRIPTIONINFO
Data Model: single value
Value: status = PENDING/ACCEPTED
       peer = PeerInfo object containing peer ID, IP address and port
              number for the incoming publish-subscribe messages.
```

When node wants to subscribe for some topic, it performs lookup for the SUBSCRIPTIONINFO object (giving the topic ID as a key).

## 6. Future work

We plan to provide event metadata to extend the default Interest Conditions and support advanced filters for the content-based multicast.

## 7. IANA Considerations

This document has no actions for IANA.

**8. References**

**8.1 Normative references**

[1] S. Baset, H. Schulzrinne and M. Matuszewski, "Peer-to-Peer Protocol(P2PP)", draft-baset-p2psip-p2pp-01 (work in progress), November 19, 2007.

[2] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD)", draft-ietf-p2psip-reload-00 (work in progress), July 11, 2008.

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

**8.2 Informative references**

[4] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002: Revised Papers, 2002.

[5] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-level multicast using content-addressable networks." in Networked Group Communication, ser. Lecture Notes inComputer Science, J. Crowcroft and M. Hofmann, Eds., vol.2233. Springer, 2001, pp. 14-29.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in Proc. Of the International Conference on Distributed Systems platforms(Middleware), 2001.

[7] A.I.T. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure." vol. 2233, pp. 30-43, 2001.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in SIGCOMM'01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. New York, NY, USA: ACM, 2001, pp. 149-160.

[9] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like p2p systems scalable," in SIGCOMM'03: Proceedings of the 2003 conference on Applications, technologies,

architectures, and protocols for computer communications.
New York, NY, USA: ACM, 2003, pp. 407-418

## Authors' Addresses

Paulina Adamska
Dept. of Computer Networks
Polish-Japanese Institute of Information Technology
Koszykowa 86,
02-008 Warsaw
Poland

Email: tiia@pjwstk.edu.pl

Adam Wierzbicki
Dept. of Computer Networks
Polish-Japanese Institute of Information Technology
Koszykowa 86,
02-008 Warsaw
Poland

Email: adamw@pjwstk.edu.pl

Tomasz Kaszuba
Dept. of Computer Networks
Polish-Japanese Institute of Information Technology
Koszykowa 86,
02-008 Warsaw
Poland

Email: kaszubat@pjwstk.edu.pl