Peer-to-peer API – abstract specification

Gabor Paller, Nokia Siemens Networks (gabor.paller@nsn.com)
Heikki Kokkinen, Nokia Research Center (heikki.kokkinen@nokia.com)

Table of contents

1. TERMS AND ABBREVIATIONS

API	Application Programming Interface
DHT	Distributed Hashtable
IMEI	International Mobile Equipment Identity
NAT	Network Address Translator
MSISDN	Mobile Subscriber ISDN Number
P2P	Peer-to-peer. Refers to a technology where there are no designated clients and servers but any node can act as client or server as the situation requires it.

2. INTRODUCTION

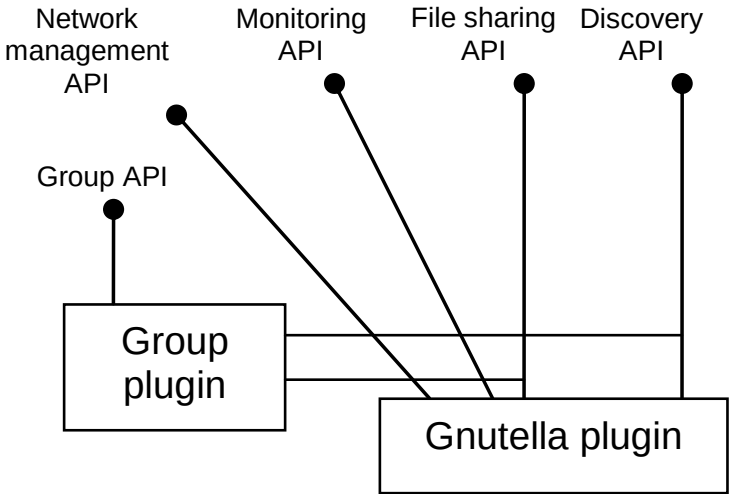
This document is the abstract specification of the P2P API. The purpose of the API is to provide applications with a simple and comprehensive programming interface to a wide variety of P2P technologies. The API described in this document is still in abstract format so that it can be mapped to any implementation technology easily. Therefore the description format used in this document is OMG IDL [1]. This is not a technology selection of any sort; IDL is just a convenient way of describing interfaces in technology-independent way.[1] In the next step the abstract API will be mapped to an interface technology selected for implementation (e.g. Symbian C++, Java, etc.).

[1] As IDL is not planned to be used during implementation in any way, the IDL in the document may not be strictly correct syntactically in every detail. This is considered to be an error only if it makes the API ambiguious for the reader.

3. COMPONENTS AND INTERFACES

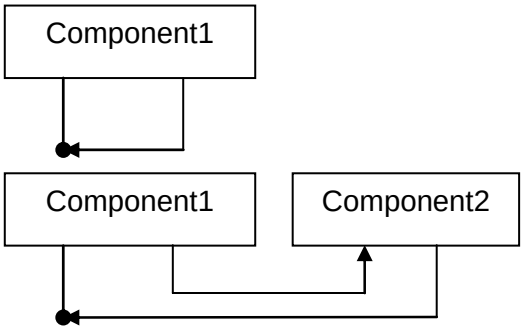
3.1 Relationships of API interfaces and components

The remaining part of the document contains interface specifications written in OMG IDL. It is important to understand the relationship of these interfaces with middleware components that the P2P middleware is composed of. The P2P middleware is component-based due to the large number of possible combination of middleware options installed (or not installed). Each middleware component may expose a number of interfaces of which we concentrate on public P2P API interfaces. One component may provide (and in fact, is expected to provide) several public P2P API interfaces. For example a Gnutella plugin provides discovery, file sharing, network management and monitoring interfaces and does not rely on any other component. Other components may rely on interfaces exposed by other components. For example a Group API component built for Gnutella networks exposes the Group API and uses the Discovery and File Sharing API exposed by the Gnutella component to publish and search for group descriptors. This configuration is depicted in the figure below.



3.2 Hooks

Other concept used by this document is a *hook*. Hook is an interface designed to connect a component implementing cross-cutting concern. The hook is not supposed to be used by applications. In its initial state the hook is connected back to the component providing the hook forming “short-circuit”. When a component implementing cross-cutting concern is installed, that component may intercept the hook and implement e.g. encryption features in the message flow. The figure below demonstrates this effect. In this figure Component1 can be seen with no hook originally then Component2 implementing a cross-cutting concern and connecting to the hook of Component1.



Cross-cutting components may be chained on hook interfaces. In order to allow the chain to be executed correctly, hooks are expected to be implemented like this (the method signature is just an example):[2]

```
RCType hookFunction( in long hookInput, out long hookOutput ) {
  // do own processing
  long myOutput = ownProcessing( hookInput );
  // call outgoing hook interface
  long chainOutput;
  callOutgoingHookInterface( hookInput, chainOutput );
  // Combine chain output and my output in some meaningful way …
  hookOutput = myOutput + chainOutput;
}
```

See section 5 for example message sequence charts for examples of hook usage.

3.3 Events

Key property of P2P networks is that it is very hard to forecast execution times therefore most of the operations need to be asynchronous. Calling the API function just initiates the procedure; the caller is notified by an event about the outcome. The event delivery procedure may be very different depending on the actual API implementation technology, e.g. each asynchronous function may receive the callback function as parameter, listeners could be registered or there can even be an asynchronous event delivery middleware in place. These issues won't be handled in this document because they are very hard to address on this abstract interface level. Instead, they will be described as IDL event types and it is supposed that there is a mechanism to transfer these events to the initiator of the operation.

4. INTERFACES

4.1 Basic types

These types are used throughout the API specification.

[2] The hook mechanism differs from the Strategy pattern (e.g. http://en.wikipedia.org/wiki/Strategy_pattern) in two important ways. First, hook components are not pluggable algorithms but part of the processing chain. Very often, the hook is expected to call the outgoing hook interface else the processing in the component will not continue (e.g. an outgoing packet is not generated). The other difference is that hooks are chained, if a hook does not call its outgoing interface then other hook components in the chain will not be executed.

4.1.1 Result type and result codes

Each API call returns a result code. The result code type and values are defined as follows.

```
typedef short RCType;

const short RC_OK = 0;        /* Succesful invocation */
const short RC_NA = 1;        /* Not available, this function is
not implemented by this interface */
const short RC_AUTHERROR = 2; /* Authentication error */
const short RC_INPROGRESS = 3; /* The operation is in progress. The
caller must wait for notification about completing the operation */
const short RC_PARAMNA = 4; /* Parameter not supported */
const short RC_FILTER = 5; /* Filter string not supported */
const short RC_SUBS = 6; /* Subscription not supported */
const short RC_NOSUCHID = 7; /* Invalid ID (e.g. search task ID) */
const short RC_NOGROUP = 8; /* Group ID does not exist */
const short RC_NOPIPEPROP = 9; /* No such pipe property */
const short RC_AUTHFAILED = 10; /* Authentication failed */
const short RC_AUTHUNKNOWN = 11; /* Can't handle these
authentication parameters */
const short RC_VOTEDDOWN = 12; /* Voting was initiated but was not
successful */
const short RC_NOMEMBER = 13; /* No more members in the group for
this iterator */
const short RC_NOVALUE = 14; /* No value stored under this key in
the DHT */
const short RC_NOSUBS = 15; /* This plugin supports no subscription
in search */
const short RC_ONLYSUBS = 16; /* This plugin supports only
subscription in search */
const short RC_NOTASKID = 17; /* The task ID is unknown */
const short RC_VALUEERROR = 18; /* The value passed to the network
technology plugin cannot be handled, e.g. it is too long */
const short RC_ITERATOR = 19; /* Multiple results are available so
an iterator ID was returned */
```

4.1.2 Hashtable

```
struct {
  string key;
  string value;
}  keyValue;

typedef sequence<keyValue> Hashtable;
```

4.2 Authorization hook

Authorization hook is supported by many components. The hook always has the same signature; the components specify only the parameter usage.

Authorization decisions always express the following statement: *subject* is allowed to make *action* on *resource.* It is just the meaning of the parameters that change in certain

component contexts. This section will define the hook's general format, the concrete interpretation of the parameters will be provided at the components.

Note that this specification assumes that there is a mechanism for the hook to find out the types of subject and resource parameters, e.g. the always the appropriate plugin is attached to the hook that knows the parameter types. In many cases, subject and resource are string but e.g. the messaging API (see section 4.7) passes binary parameters to the hook.

```
interface P2PAPI_AuthHook {
  RCType doAuthorization( in long action, in any subject, in any
resource, out boolean result );
};
```

doAuthorization

Asks the authorization plugin to make authorization decision.

action – number code of the action

subject – subject making the action

resource – resource the action is affecting

result – true if the plugin authorizes the action. The default implementation authorizes everything.

4.3 Power management hook

Power management hook is used by components to provide the central power manager with estimates how much CPU load they will cause in the foreseeable future. When the components start or stop some activity, they call this hook to update the power manager with their CPU utilization estimates. The power manager then uses these estimates to regulate CPU speed.

The power manager and the component may be involved in power negotiation using this hook. The component may call the hook with the request to use 10% of CPU power for 10 seconds. The power manager on the other side of the hook may recommend just 5%. In this situation the component may call the power manager again with 5% accepting its recommendation, may go ahead with 10% and face the consequences or may abort the operation altogether.

In order to facilitate component writers' work, CPU load is specified not in CPU load percentage on the host processor but in kiloinstruction/sec on a reference processor. This allows the component to return constant numbers for certain processing tasks with known CPU load. The power manager calculates the actual CPU load requirement from these reference numbers.

```
interface P2PAPI_PowerHook {
  RCType powerEstimate( in long cpuload, in long time, out long
recommendedCpuLoad );
};
```

powerEstimate

Notifies the power manager that this component estimates to use a certain amount of CPU time for a certain duration of time.

cpuload – CPU load required by the component.

time – estimated time the given CPU load is expected to last.

recommendedCpuLoad – CPU load recommended by the power manager. If this is smaller than the required load, the component must call the hook again with the recommended value if it accepted the recommendation.

4.4 Network management

4.4.1 Main API

```
enum SupernodeOps { SET, QUERY };

interface P2PAPI_NetworkManagement {
    RCType initNetwork( in Hashtable bootstrap );
    RCType startNetwork( out taskid );
    RCType stopNetwork();
    RCType cancel( in taskid );
    RCType supernodeCntl( in SupernodeOps op, inout boolean
supernodeStatus, inout Hashtable supernodeParms );
};
```

initNetwork

Initializes the network.

bootstrap - Network initialization parameters, including authentication and security parameters are passed as key-value pairs.

startNetwork

Actually connects the network. If returns successfully, the node has become member of the P2P network. If the operation is asynchronous (startNetwork returns RC_INPROGRESS), ID number is returned in *taskID* that allows canceling the operation with the cancel method.

stopNetwork

Disconnects from the network. If returns successfully, the node has disconnected from the P2P network.

cancel

Cancels an ongoing asynchronous startNetwork operation.

supernodeCntl

Sets and queries supernode status and parameters. Certain network technology plugins may work in supernode mode when they support other nodes by e.g. caching resource index. This mode involves more memory, CPU and disk space consumption and increased traffic.

op – Operation to exectute. (set or query supernode status)

supernodeStatus – If SET operation, supernode operation is enabled if this parameter is true. If QUERY operation, set to true if the component is in suprenode mode.

supernodeParms – any parameter associated with the supernode operation. Used to set supernode parameters when in SET mode, used to retrieve supernode parameters when in QUERY mode. May be NULL if supernode parameters are not used.

4.4.2 Events

```
eventtype networkStarted {
  RCType resultCode;
};

eventtype networkStopped{
  RCType resultCode;
};

eventtype supernodeStatus {
  RCType resultCode;
  boolean supernodeState;
};
```

networkStarted

P2PAPI_NetworkManagement::startNetwork operation has finished. The outcome of the operation is in resultCode. This event is delivered only if startNetwork returned RC_INPROGRESS result code.

networkStopped

P2PAPI_NetworkManagement::stopNetwork operation has finished. The outcome of the operation is in resultCode. This event is delivered only if stopNetwork returned RC_INPROGRESS result code.

supernodeStatus

SET operation of P2PAPI_NetworkManagement:: supernodeCntl operation has finished. The outcome of the operation is in resultCode and the current supernode status is in supernodeState (true if in supernode state). This event is delivered only if SET operation of supernodeCntl returned RC_INPROGRESS result code.

4.4.3 Authentication hook

```
interface P2PAPI_NetworkManagement_AuthHook {
  RCType authParms( inout Hashtable initParms );
};
```

initParms

This hook is called if the component needs authentication before joining the network (either during initNetwork or startNetwork processing). The hook is called with a copy of initialization parameters plus any initialization parameter the component may have

retrieved/calculated related to authentication. The hook evaluates authentication information, may try to retrieve more (e.g. by consulting a directory service or prompting the user) and updates the input hashtable with the authentication data that is used by the component to authenticate with the network.

4.4.4 Authorization hook

See general description of the hook at section 4.2.

Requires authorization of a node trying to attach to the network.

```
const long AUTH_NETWORK_ATTACH = 1;
```

subject – principal that tries to attach to the network (node or user)

resource – not defined, null.

4.5 Monitoring

4.5.1 Main API

```
interface P2PAPI_Monitoring {
  RCType getMonitoringValue( in string key, out any result );
};
```

getMonitoringValue

key – key of the monitoring value to query. See below for possible values.

result – value of monitoring variable. If the operation results RC_INPROGRESS, the value is delivered as an event.

The following key values are defined now. The plugin returns RC_PARAMNA for any monitoring variable not supported by this network technology plugin.

roundtrip:<nodeid>

Measures the roundtrip to a node identified by <nodeid>. Node ID is represented in a convenient textual format supported by the network technology plugin. The result is in long format and is in milliseconds.

battery:<nodeid>

Obtains the battery status of the node identified by <nodeid>. Node ID is represented in a convenient textual format supported by the network technology plugin. The result is in long format and is in percentage of maximum battery power (e.g. 64 -> 64%).

throughput:<nodeid>

Obtains the throughput toward the node idenrified by <nodeid>. Node ID is represented in a convenient textual format supported by the network technology plugin. The result is in long format and is the estimated throughput to the specified node in KBit/sec.

hopcount:<nodeid>

Obtains the number of the hops from this host to the host identified by <nodeid>. The result is in long format and is the number of hops to the specified node, -1 if the target node is unreachable. It is up to the implementation to define the precision of the measurement result, e.g. a specification where -1 means unreachable node, 0 means immediate connection and 1 means one or more hops is valid.

4.5.2 Events

```
eventtype monitoringValue {
   string key;
   RCType resultCode;
   any result;
};
```

monitoringValue

Result of a getMonitoringValue operation if the API call returned RC_INPROGRESS.

4.5.3 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_MONITOR = 2;
```

Requires authorization of incoming monitoring request.

subject – node that initiated the measurement.

resource – type of measurement, same as monitoring key.

4.6 Discovery

4.6.1 Filters

Discovery operations accept filter string in a simple format. Filter strings resemble and are encoded similary to HTTP query requests. See [2] for the encoding used. The filter consists of meta-information query expressions separated by & characters. Simple wildcards (*.?) may be used although the plugin may refuse handling them.

Example:

type=file&name=pricelist.xls

The following filter tags are available for every filter expression. Note again that the plugin has right to discard any filter expression based on its discovery capabilities.

domain

The domain of the discovery process. The following values are supported.

- proximity – discover in the proximity using any proximity bearer.
- subnet – discover in a network neighborhood which is considered “close” in network terms (fast and cheap connection is available)

- organization – discover in a network neighborhood which is behind the organization's firewalls.
- global – discover in the reach of P2P network.

type

Type of the item to discover. Note that domain subfilter is always available if the network plugin supports it. The following item types are supported.

- peer – discover nodes according to the additional filter conditions. Supported additional filters:
    - name – filter for the name of the node to discover. Supports wildcards if the network plugin does.
    - addressX – physical network address of the node in string format. X is an integer ranging from 1 to the number of interfaces with distinct interfaces that the node supports. The property is expected to be used in search result string, node name alone should be enough to locate a node in peer discovery operations. Value of the property is physical network address of the particular interface in string format. Prefixes are used to differentiate address types. At this stage only IPv4 prefix is specified.
        - IPv4:x.x.x.x – IPv4 address in string format. X values are decimal numbers.
- pipe – discover pipe endpoints. Supported additional filters:
    - name - pipe name. Supports wildcards if the network plugin does.
    - transport – Allowed values are "reliable" or "unreliable".
    - pipe_xxx – xxx is a plugin-specified property for describing any plugin-specific search property. These are expected to be mostly related to unreliable pipes used for real-time data transfer.
- service – discover services according to subfilter expressions. Supported additional filters:
    - name – service name. Supports wildcards if the network plugin does.
    - id – service ID. This is used only for transports that distinguish between technical service IDs and user-readable service names. If there is no such distinction, only name filter should be used.
    - invocation – Allowed values are "reliable" and "unreliable".
    - service_xxx – xxx is a plugin-specified property for describing any plugin-specific search property. These are expected to be filters associated with business properties of the service, e.g. free of charge, authentication needed, etc.
- file – discover files according to subfilter expressions. Supported additional filters:

  - name – file name. Supports wildcards if the network plugin does.
  - file_xxx – any additional meta-information that may be associated with the file.

- group – discover groups according to subfilter expressions. Supported additional filters:
  - name – group name. Supports wildcards if the network plugin does.
- component_implementation: discovers downloadable modules that are necessary to consume or provide a service.
  - service_name – name of the service that the module implements or is necessary to consume the service.
  - module_name – name of the downloadable module. By convention, this is equal to the service name with _client (the module needed to consume the service) or _server postfix (the module needed to provide the service). If the module is a servent (able to work both in client and server mode) then the suffix is an empty string.
  - language – name of the implementation language in lowercase. Currently specified values are: series60_symbian_c++_1_0, series60_symbian_c++_2_0, series60_symbian_c++_1_0, midp_java_1_0 and midp_java_2_0.
  - device – name of the device on which the downloadable module is expected to run. E.g. Nokia_9500.
  - signer – user-readable name of the entity that signed the downloadable module.
- database: discovers synchronizable databases.
  - name – name of the database. Supports wildcards if the network plugin does.
  - syncmethod – name of the synchronization method that this node is able to use to synchronize with the database. It can be a comma-separated list. Currently only SyncML Datasync (syncmlds) is specified.
- traversalserver: discover traversal servers used to cross NATs.
  - type – type of the traversal method the server is able to support. Currently “stun” and “turn” are supported.

4.6.2 Search results

Search results are returned one by one, in smaller packages or in one set depending on the search technology used. Each search result is a string, formatted according to URI escaping rules. The format is:

<object ID>?<prop1>=<value1>[&<prop2>=<value2>] …

Object ID is any transport-specific ID that can be used to locate the object (e.g. file, service, etc.) on the network. Object IDs returned as search results can be directly passed to API calls that consume the object (e.g. pipe access, service invocation or file download API calls).

The name-value pairs after the object ID is meta-information associated with the object. Type meta-information is always returned, otherwise network technology plugins are free to choose how much meta-information they return in the search result but they are encouraged to return as much meta-information as possible to facilitate selection of the best object source from the multiple sources available.

Meta-information names for the search results are the same as in search filters.

This compact representation is optimized for efficiency; in large P2P networks thousands of search results may be received from the peer nodes.

4.6.3 Main API

```
interface P2PAPI_Discovery {
    RCType searchItem( in string filter,in boolean background, in
boolean subscription, in Hashtable secproperties, out
sequence<string> results,out long taskid );
    RCType cancelSearch( in long taskid );
};
```

searchItem

Searches for discoverable items on the network.

filter – search filter string. If the network technology plugin doesn't support the search filter, RC_FILTER must be returned.

background – if true, the call is expected to return RC_INPROGRESS and return all the search results as search events. If false, the call *may* suspend the execution of the caller as long as the search operation is in progress. Note that as search operations are inherently long-running, the network technology plugin may always choose to return just some results when the call returns and return the remaining items as search events. RC_INPROGRESS is always returned if not all the search results were returned by the method call (e.g. additional search results may be received in the form of search events).

subscription – if true, the search operation is not a one-time search but a subscription. Subscriptions continue the search as long as they are not cancelled explicitly. The network technology plugin may refuse subscription operation with RC_NOSUBS error code. If the plugin supports only subscription and this parameter is false, RC_ONLYSUBS error code is returned.

secproperties – Security properties of the an authenticated entity (see identity API, section 4.10). This is needed if the search initiator is impersonated and identifies itself to search targets. Null if the search initiator is anonymous. This feature allows search targets to decide on generating search results based on search initiator's identity.

results – the search result array. It is correct behaviour for network technology plugins to return no result strings here and return all the search results as events although they

should try to return as many search results as possible here if *background* input parameter is false.

taskid – Task ID of the search task that is used in events and can be used as input to cancelSearch call. Returns a unique positive integer that can be used to identify this particular search operation or -1 if the *results* array contains all the search results and it is guaranteed that no search events are generated in relation to this search operation.

cancelSearch

Cancels a running search task.

taskid - ID of the search operation returned by the searchItem call. Returns RC_NOSUCHID if the task ID is not recognized by the system.

4.6.4 Events

```
eventtype searchResult {
   RCType resultCode;
   long taskid;
   sequence<string> results;
};

eventtype searchFinished {
   RCType resultCode;
   long taskid;
};
```

searchResult

Search result event of a long-running search task. Taskid is the ID returned by searchItem call when the search was initiated and results contains one or more search result string.

searchFinished

Notifies the caller that a certain search task has finished. This event is generated only if the termination of the task was not requested explicitly by the initiator of the search operation, e.g. it was not a foreground search task that did not generate any search events or it was not cancelled by the API user explicitly. This event is not guaranteed to be generated as in case of certain networks the end of the search operation cannot be determined. For example in case of Gnutella network, the end of the search operation cannot be determined.

4.6.5 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_DISCOVERY  = 3;
```

Requires authorization of incoming search request.

subject – node and search initator (e.g. user) that initiated an incoming search request. If both are given, the node and search initiator names are separated by a newline (\n) character.

resource – the search filter.

4.6.6 Dynamic result hook

This hook allows installing plugins that respond to search requests dynamically. These plugins receive the search filter and generate matching object IDs. It is the responsibiity of the plugin to generate object IDs that can be referred to, e.g. return meaningful results when referenced.

Dynamic result hooks do not affect the result entries that are returned as result of other publishing operation, e.g. pipe publication.

The P2P middleware is free to choose the method of returning search results returned by the plugin to the search initators. For example partial search results may be returned to the search initiator when the plugin returns or all the returns can be collected and sent back in one batch.

```
interface P2PAPI_Discovery _DynamicResultHook {
  RCType search( in string filter, out sequence<string>
searchResult );
};
```

search

Invoked when this node receives a search request from another node. The method receives the search filter and generates 0 or more search results that matches the request.

filter – the search filter that the node received in the search request.

searchResult – 0 or more result strings that matches the filter.

4.7 Messaging

Messaging API is the lowest level of data transfer. The purpose is just passing messages composed of byte arrays to nodes identified by node ID. The messages sent and received by this function can be distinguished from other message traffic based on a flag. The two options are:

- Receive any traffic conforming to the message filter, including P2P middleware network management messages and messages belonging to other data transfer mechanism (including pipes).
- Receive only messages sent by the functions of this API (e.g. sendMessage).

4.7.1 Addresses and filters

The node ID address format depends on the network technology plugin. Because this interface potentially handles large amount of data, network technology plugin may choose to represent addresses in its convenient binary format (e.g. packed IPv4) and may not use the string representation of the node ID. Address format specifiers are used throughout this API to handle the address format issue.

There is a string format to handle groups: group:<group ID> refers to all the members of the group.

Simple address filters may be used in the message receiver side. Each address format defines its own address filter format (e.g. IPv4 address mask). This specification deals only with filters in string format. Filters in string format are addresses in string representation potentially containing usual wildcards (*,?).

4.7.2 Main API

```
enum addressTypes{ STRING, IPv4_PACKED, CAST_ID }; /* Initial list,
to be extended */

interface P2PAPI_Messaging {
  RCType sendMessage( in boolean all, in short addressType, in any
address, in sequence<octet> message, in Hashtable secparams );
  RCType sendBroadcast( in boolean all, in short addressType, in
any broadcast_address, in sequence<octet> message );
  RCType subscribeMessage( in boolean all, in short addressType, in
any filter, out long taskid, in Hashtable secparams );
  RCType cancelSubscribeMessage( in long taskid );
};
```

sendMessage

Sends a message to an address or to a group (if the address is in group format).

all – if true, this is a raw message that cannot be distinguished from any other traffic (see section 4.7, introduction part).
addressType – Address type of the *address* parameter. Its value is one of the *addressTypes* enumeration.

address – address to send the message to. It may be a group ID if the group address format is used.

message – the message in byte array format.

secparams – security parameters to be used with the sending operation. These are parameters specified by the encryption and authentication provider (see section 4.7.5).

sendBroadcast

Sends a broadcast message using network technology-specific broadcast address format. The purpose of the call is to exploit broadcast mechanisms that the network technology may directly offer gaining performance advantage over group addressing.

all – if true, this is a raw message that cannot be distinguished from any other traffic (see section 4.7, introduction part).

addressType – Address type of the *address* parameter. Its value is one of the *addressTypes* enumeration.

address – address to send the message to. This is a network technology-specific broadcast address. Group addressing is not allowed here.

message – the message in byte array format.

subscribeMessage

Subscribes for message receive events. The API user specifies an address filter and the plugin delivers the messages as messageReceiveEvent events.

all – if this flag is true, all the messages are received that match the address filter, irrespectively whether they were sent by the sendMessage calls or are any other form of traffic.

addressType - Address type of the *filter* parameter. Its value is one of the *addressTypes* enumeration. Note that filter and address types use the same format sets.

filter – address filter in format according to *addressType*.

taskid – Unique task ID associated with the subscription that may be used to cancel the subscription.

secparams – security parameters associated with the encryption and authentication provider (see section 4.7.5).

cancelSubscribeMessage

Cancels a previous message subscription request.

taskid – Task ID of the subscribe operation returned by subscribeMessage.

Returns RC_NOTASKID if this taks ID is not recognized by the middleware layer.

4.7.3 Events

```
eventtype messageReceiveEvent {
   long taskid;
   short addressType;
   any sourceAddress;
   Hashtable secproperties;
   sequence<octet> message;
};
```

messageReceiveEvent

This event is generated when a message is received as a result of subscribeMessage call. Note that message boundaries of sendMessage calls and messageReceiveEvent events are not necessarily the same, message fragmentation of lower layers may result in multiple messageReceiveEvent events generated from a single sendMessage call.

taskid – Subscription task ID (see subscribeMessage) this event is related to.

addressType – type of the sourceAddress field.

sourceAddress – address of the node that sent the message.

secproperties – any security properties associated with the message, e.g. authenticated identity of the sender.

message – the message itself.

4.7.4 Group hook

This hook is called by the message API provider when it is requested to send messages to a group. The hook receives a group ID and returns addresses of group members. If the hook is closed, it always returns RC_NOGROUP error code.

```
interface P2PAPI_GroupHook {
  RCType members( in string groupID, out sequence<short>
addressType, out sequence<any> members );
}
```

members

groupID – ID of the group to query.

addressType – address type array. There is one type data for each address in the *members* array.

members – addresses of group members

4.7.5 Encryption and authentication hook

This hook allows authentication, encryption/decryption and message authentication to be applied at message level. The hook is called at the following points in the message flow.

- SEND – called before the message is passed to the actual message sending layer. The message is already complete, e.g. it contains the user message plus all the headers that may be added by the messaging layer. It is also fragmented if the messaging layer fragments the user message.
- RECEIVE – called immediately after the message is received from the message receiving layer. The message may still be fragmented and containing headers necessary for the messaging layer.

```
enum MessagingOps { SEND, RECEIVE };

interface P2PAPI_Messaging_EncryptionHook {
  RCType handleMessage( in MessagingOps op, inout Hashtable
secparams, in long taskid, in sequence<octet> inmessage, out
sequence<octet> outmessage );
}
```

handleMessage

Handles an incoming or outgoing message. Incoming messages are decrypted, their sender and integrity may be checked. Outgoing messages are encrypted, sender and message authentication information is added. Decryption/authentication errors are signaled in the return value to the caller of the hook.

op – operation that resulted in the call of the hook (see MessagingOps).

secparams – security parameters passed to the sendMessage or to the subscribeMessage call or null if they are not present. In case of subscribeMessage, the security parameters are saved and are passed to this hook only if a message arrives that satisfies the subscription. The hook may process the message and add additional security parameters (e.g. it can authenticate the message source based on the credentials in the message and produce identification parameters in secparams).

taskid – task ID if the message is associated with a message subscription.

inmessage – the message that the plugin receives. The plugin takes this message, decrypts it if it is an incoming message and encrypts it if it is an outgoing message.

outmessage – the encrypted/decrypted message.

4.7.6 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_INMESSAGE  = 4;
```

Requires authorization of incoming message.

subject – node that sent the message (node name in string format if available or address in network format). If the hook is called after authentication hook, the hook is called with the identity of the authenticated sender.

resource – the message in byte array format.

4.8 Pipes

Pipe API provides bidirectional socket-like capabilities. The pipe API may be implemented on top of message API or may be based on a proprietary messaging layer.

4.8.1 Pipe specification

Pipe specification is a string consisting of name-value pairs. The encoding conforms to the URI encoding [2] and the format is the following:

<prop1>=<value1>[&<prop2>=<value2>]

The properties are the same as defines in section 4.6.1. Name property is mandatory; this should be a globally unique name for the pipe. Transport property is optional, it defaults to “reliable” if not specified.

4.8.2 Pipe ID

Pipe ID is conforms to the search result format presented in section 4.6.2. The most important part is object ID. The object ID is in network technology-specific format but it must allow the location of the pipe endpoint over the entire network that the network technology plugin supports. Pipe methods accept the entire search result format or just the object ID, the API user does not have to deal with search result parsing.

4.8.3 Main API

```
enum PipeOps{ GET, SET };

interface P2PAPI_Pipe {
  RCType publishPipe( in string pipespec, out string objectid, in
Hashtable secproperties );
  RCType revokePipe( in string objectid );
  RCType connectPipe( in string objectid, out long pipeid, in
Hashtable secproperties );
  RCType writePipe( in long pipeid, sequence<octet> buffer );
  RCType closePipe( in long pipeid );
  RCType pipeCtl( in pipeops op, in string key, in any value, out
any result );
};
```

publishPipe

Publishes a pipe according to pipe specification. The result of the publishing is that the pipe can be discovered by discovery methods.

pipespec – specification of the pipe to be published.

objectid – object ID of the created pipe if the pipe was published successfully

secproperties – Key-value pairs needed for the encryption and authentication plugin (see section 4.8.5). The interpretation of this hashtable depends on the encryption and authentication technogy used.

revokePipe

Unpublishes a previously published pipe.

objectid – Object ID of the previously created pipe.

connectPipe

Connects to a pipe identified by its object ID. The object ID was most possibly obtained by means of a discovery call. Pipe can be written using writePipe call, incoming messages are delivered as events.

objectid – object ID of the pipe to connect to

pipeid – pipe ID if the pipe connection was successful.

secproperties – Key-value pairs needed for the encryption and authentication plugin (see section 4.8.5). The interpretation of this hashtable depends on the encryption and authentication technogy used.

writePipe

Writes a specified amount of bytes into the pipe. Reliable pipes guarantee that the bytes written into the pipe will arrive in the order they were written into the pipe and no byte will be lost (or error is generated). There is no such guarantee with unreliable pipes.

pipeid – ID of the pipe connection previously returned by connectPipe call.

buffer – byte array containing the message to be written into the pipe. All the bytes in the buffer will be written into the pipe.

closePipe

Closes a pipe.

pipeid – ID of the pipe to close.

pipeCtl

Controls and retrieves pipe properties. These properties are expected to be used by unreliable pipes used for real-time operations. For example the pipe property that this is a real-time pipe and packets sent over this pipe can be discarded in case of network congestion can be set or the jitter measured from received real-time packets can be retrieved.

This method is expected to be invoked immediately after the pipe is opened. Implementations may allow invoking this method during data transfer as well but they have to state explicitly, what properties can be set and at what stage. If there is no such explicit statement, any invocation of this method after the data transfer was started on the pipe is expected to be ignored.

Pipe properties are not specified here, it is expected that network technology plugins providing real-time pipe service will define these.

op – pipe control operation, SET or GET

key – name of the property to set

value – value to be set if it is a SET operation. Otherwise ignored.

result – result of the GET operation.

4.8.4 Events

```
eventtype pipeReceiveEvent {
   long pipeid;
   sequence<octet> message;
};
```

pipeReceiveEvent

This event is generated when incoming message arrives on an open pipe.

pipeid – ID of the of the pipe on which the message is coming in

message – the message as a byte array.

4.8.5 Encryption and authentication hook

This hook allows authentication, encryption/decryption and message authentication to be applied at pipe (session) level.

```
enum PipeHookOps { PUBLISH, UNPUBLISH, CONNECT, READ,  WRITE, CLOSE
};

interface P2PAPI_Messaging_EncryptionHook {
  RCType handlePipeOp( in PipeHookOps op, in string objectid, in
long pipeid, inout Hashtable properties, in sequence<octet>
inmessage, out sequence<octet> outmessage );
}
```

handlePipeOp

Handles a pipe operation from authentication, encryption and message authentication point of view. The hook is called for every pipe operation and the hook receives the parameters of the pipe operation. The hook is called at the following places.

- PUBLISH – the object ID is already known but the pipe has not been published yet.
- UNPUBLISH – the pipe has not been unpublished yet.
- CONNECT – the pipe ID is already known but the pipe has not been connected yet.
- READ – incoming packet has arrived and it is known to be associated with the particular pipe ID but the message has not been passed to pipe API users.
- WRITE – the message has been received from the pipe API user but it has not been extended with any transport-related data.

The default implementation of the hook simply copies incoming message buffer to outgoing message buffer.

op – pipe operation to handle.

objectid – object ID of the pipe in question.

pipeid – pipe ID if known else -1.

properties – security properties (e.g. for PUBLISH and CONNECT) or null if securtity properties are not present. It is expected that in case of CONNECT calls, the middleware will provide enough information in security properties so that the authentication hook can perform the authentication. Alternatively, there may be no security properties available at connect but authentication plugin may produce authentication information gradually. This requires that properties returned from this call (this is an inout parameter) are preserved by the pipe API implementation and are passed to the hook when the next packet belonging to the hook arrives.

inmessage – message passed to the hook in case of READ or WRITE operation.

outmessage – message produced by the hook in case of READ or WRITE operation.

4.8.6 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_PIPECONNECT  = 5;
```

Requires authorization of incoming pipe connection request.

subject – node and principal (e.g. user) that initiated the incoming pipe connection request. If both are given, the node and principal are separated by a newline (\n) character. Principal parameter is passed only if the connection initiator defined secparams that contained an authenticated principal.

resource –pipe ID in long format.

4.9 Services

Service API provides reliable or unreliable service invocation schemes. It is layered on top of the messaging or pipe API and serializes/deserializes unidirectional or bidirectional service invocations to and from remote services. Note that as the actual service invocation scheme depens on the service middleware used, the P2P API takes care only of service publication, security/authentication processing and discovery. Sending and receiving service invocations is accomplished by a plugin-specific proprietary API.

4.9.1 Main API

```
interface P2PAPI_Service {
  RCType publishService( in Hashtable serviceproperties, in
Hashtable secproperties, out string objectid );
  RCType revokeService( in string objectid );
};
```

publishService

Publishes the service so that it can be discovered by the Discovery API.

serviceproperties – properties needed to publish the service. The plugin may specify a number of properties that identify the service handling logic, binding, etc. This specification describes only the *invocation* and *name* properties as described at section 4.6.1.

secproperties – security properties associated with the service to be published. These include encryption level required to connect to the service, authentication required, etc.

objectid – the ID under which the service was published.

revokeService

Revokes a previously published service.

objectid – Object ID of a previously published service.

4.9.2 Encryption and authentication hook

This hook allows authentication, encryption/decryption and message authentication to be applied at service level.

enum ServiceHookOps { SEND, RECEIVE };

```
interface P2PAPI_Service_EncryptionHook {
  RCType handleServiceOp( in ServiceHookOps op, in string objectid,
```

```
in Hashtable properties, in sequence<octet> inmessage, out
sequence<octet> outmessage );
}
```

handleServiceOp

Handles a service invocation operation or incoming service request from authentication, encryption and message authentication point of view.

- SEND – The API user sends a service invocation.
- RECEIVE – The middleware has received service invocation.

The default implementation of the hook simply copies incoming message buffer to outgoing message buffer.

op – service operation to handle.

objectid – object ID of the service in question.

properties – security properties for SEND (service invocation API is out of scope of this specification, presumably there is a way to define security properties for service invocation).

inmessage – entire service invocation message (clear text from the service layer in case of SEND, encrypted/authenticated message in case of RECEIVE).

outmessage – message produced by the hook. (clear text from the service layer in case of RECEIVE, encrypted/authenticated message in case of SEND).

4.9.3 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_SERVICEINVOCATION  = 6;
```

Requires authorization of incoming service invocation request.

subject – node and principal (e.g. user) that initiated the incoming service invocation request. If both are given, the node and principal are separated by a newline (\n) character. Principal parameter is passed only if the service was invoked with security properties specified.

resource –Object ID of the service that was invoked.

4.10 Identity

The identity provider behind the Identity API allows identification and authentication of a principal. Principal in our application domain is user or host, in some special cases it can be an application. When the identity provider is called, the aggregator behind the API calls the installed identity plugins one by one using the security parameters that were specified by the caller. During this process, one such the identity plugin checks the parameters, decides whether it is able to do such authentication (e.g. only the Liberty plugin would be able to

handle Liberty authentication parameters) then performs the identification/authentication task.

The security parameters depend on the identity plugin but one key is predefined. The *principal* key is the identity with which the user is to be authenticated. The value of the key is in a format that refers to the plugin, e.g. x509cert:<certificate> refers to the X.509 certificate-based identification plugin. In order to support multiple current identities, the plugin removes the *principal* key and replaces is with another key in <pluginname>:principal format. For example if the X.509 certificate was correct then the *principal* key with x509cert:<certificate> value is replaced with x509cert:principal key with <certificate> as value.

Identites exist in the form of security parameters which are name-value pairs. The API user provides some initial parameters when the API is called (e.g. user name and the preference to use Yahoo account) and the plugin updates the security parameters with additional name-values that can be used to access services protected by authentication procedures (e.g. it prompts the user for password, checks the username/password with the network-based provider and updates the security parameters with a session key that may be used to access services protected by that identity technology).

It is out of the scope of this specification to describe these name-value pairs because they can be very different based on the identity technology used. This specification only requires that plugins define their parameters so that their plugin-specific keys are in <plugin-name>.<key-name> format, e.g. liberty.username may be the user name for a Liberty identity plugin.

The API may provide current identity or identites. For example when a user uses the mobile phone, the identity associated with the device (IMEI number) or its subscriber identity (MSISDN number) are available without any login process. Other use case for current identity may be a network-based identity provider (e.g. Yahoo). The user logs in once and starts to use another application also requiring the network-based account. The security parameters presented by that application are good enough for the plugin so that it immediately returns the network-based identity parameters (e.g. the user of the same mobile device logged in 5 minutes before and the application is trusted).

4.10.1 Main API

```
interface P2PAPI_Identity {
  RCType doAuthentication( inout Hashtable secproperties );
  RCType currentIdentity( out Hashtable secproperties );
};
```

doAuthentication

Authenticate with the given security properties. If the security properties were recognized but the authentication failed, RC_AUTHFAILED is the response, if the security properties were not recognized (there is no plugin installed to handle these properties), RC_AUTHUNKNOWN is the response code.

secproperties – Security properties that are expected to be processed by the identity plug-in. The plugin updates the table with additional key-value pairs that may be used to access services protected by this authentication technology.

currentIdentity

Returns the security parameters of the identities that the user can use without any further authentication. This may include for example identities associated with the mobile equipment or its subscriber. If the API technology is able to provide caller trust check (e.g. based by the signatures of calling applications) the currently active network-based identities may also be returned (care must be taken that these security parameters are not returned to e.g. a virus or other untrusted code).

secproperties – security parameters associated with all current the identities.

4.11 Group

Group API provides group management features. Groups affect other functionality groups by means of group hooks (e.g. section 4.7.4). Group API is also used by authorization plugins, making the group support available for almost all the other components.

4.11.1 Voting plugin interface

Voting plugins are associated with the group join and destoy operations. When a new member wants to join the group or to destroy the group, members vote on the new member based on the voting policy. The voting policy is determined by the group's voting plugin; e.g. there can be a central node that approves new members or any member can let new members in, etc. When the new node joins, the group votes according to the voting policy and if the vote is successful, the new member becomes part of the group. Same applies to group destroy operations.

A group can never be empty. The creator of the group is also added as its first member. The last member of the group is not allowed to leave the group, it must explicitly destroy it.

Voting may be performed in turns. For example the policy may be that every group member has to agree on joining or destroying the group but some members will agree only if a group member they trust votes yes. This policy can be implemented in two turns: first only some members vote yes, others abstain. In the second round the members are asked to vote again and they receive the result of the first round. In this second round abstaining is equal to no vote. The API supports voting in rounds by allowing voting function to require another round. Voting policy may limit the number of the rounds to e.g. 2 rounds so that the voting process surely finishes.

In order to prevent uncontrolled right elevation by becoming member of the group then changing the voting plugin to a more liberal one, voting plugin updates (see setVotingPlugin, section 4.11.2) are also controlled by the active voting plugin. When the voting plugin is to be updated, setVotingPlugin is called on the voting plugin. The voting plugin may refuse to allow voting plugin updates or may allow only to update to a particular voting plugin, etc.

```
enum GroupOps { JOIN, DESTROY };

enum VoteResult { YES, NO, ABSTAIN, ONE_MORE_TURN };

interface P2PAPI_VotingPlugin {
   RCType vote( in GroupOps op, in string objectid, in string
principal, in Hashtable secparams, in string contextHandle, out
VoteResult result );
```

```
   RCType setVotingPlugin( in string objectid, P2PAPI_VotingPlugin
newplugin );
};
```

vote

Votes on joining a new member or destroying the group.

op - the operation the members vote on.

objectid – ID of the group that the new member wants to join.

principal – principal of the new member (see section 4.10 on principals).

secparams – security parameters that may have been processed by the identity plugin (see section 4.10). In case of secure groups, the plugin may want to check whether the principal was indeed identified and authenticated. If the group is not secure, this parameter is null.

contextHandle – Handle of the voting context that can be used in multi-turn voting. This handle can be used with P2PAPI_Group_Iterator interface (see section 4.11.2) to obtain the members that voted “yes” in the previous group.

result – result of voting. This can be YES, NO, ABSTAIN and ONE_MORE_TURN (see VoteResult enumeration).

setVotingPlugin

This method is called when the API user wants to set a new voting plugin. The current voting plugin may decide whether it allows updating the voting plugin thus preventing uncontrolled right elevation.

objectid – ID of the group for which new voting plugin is to be set on this node.

newplugin – reference to the new voting plugin instance

4.11.2 Main API

```
interface P2PAPI_Group_Iterator {
  RCType iterateGroupMembers( in string objectid, out long
iteratorid );
  RCType nextGroupMember( in long iteratorid, out string principal
);
  RCType closeGroupIterator( in long iteratorid );
};
```

This API allows iteration of group or voting context members.

iterateGroupMembers

Creates an iterator so that all the group members can be obtained.

objectid – ID of the group

iteratorid – the handle that can be used to iterate over group members.

nextGroupMember

Retrieves the next member from the iterator. Returns RC_NOMEMBER if there are no more members in the iterator, in this case it also closes the iterator handle.

iteratorid – Iterator handle created by iterateGroupMembers method.

principal – next group member returned by the iterator.

closeGroupIterator

Closes the iterator handle and frees all resources associated with it. Note that nextGroupMember method does this automatically when it reaches end of the group member list.

iteratorid – iterator handle to close.

```
interface P2PAPI_Group {
  RCType createGroup( in string groupname, out string objectid );
  RCType destroyGroup( in string objectid );
  RCType joinGroup( in string objectid, in Hashtable secparams );
  RCType leaveGroup( in string objectid, in string principal );
  RCType belongsToGroup( in string objectid, in string principal,
out boolean result );
  RCType setVotingPlugin( in string objectid,P2PAPI_VotingPlugin
plugin );
};
```

createGroup

Creates a group. The new group is published so that discovery functions find it (see section 4.6.1). The default voting plugin is such that it allows the update of the voting plugin for this node. Creating a group automatically creates a synchronizable data store for the group that is automatically kept in sync by the synchronization middleware (see section 4.16) with the replicas at group members.

groupname – name of the group to create. This should be a unique name. Note that the group API is not able to guarantee uniqueness because any node may emerge having the group with the same name at any moment.

objectid – ID of the group that can be used to reference it locally or over the network.

destroyGroup

Destroys the group if the voting policy allows it. Returns RC_VOTEDDOWN if the members did not allow destroying the group. Destroying the group also destroys the data store associated with the group. Destroying group can be a lengthy process; groupDestroy event with the outcome of the operation is delivered if the method returns RC_INPROGRESS.

objectid – ID of the group to destroy.

joinGroup

Joins a group if the voting policy allows is. Returns RC_VOTEDDOWN if the members did not allow the new member to join the group. Joining group can be a lengthy process; groupJoin event with the outcome of the operation is delivered if the method returns RC_INPROGRESS.

Joining a group creates synchronization association with the group's data store (see section 4.16).

objectid – ID of the group to join.

secparams – Security parameters to identify the caller for the rest of the group. Handling of this table depends on the group plugin but is expected to be a security parameter table authenticated by the identity plugin (see section 4.10).

leaveGroup

Leaves the group.

objectid – ID of the group to leave.

principal – Identifies the identity that wants to leave the group.

belongsToGroup

Checks whether a certain principal belongs to group.

objectid – ID of the group.

principal – the principal to check.

result – true if the principal belongs to the group, false otherwise.

setVotingPlugin

Sets the voting plugin for a certain group. See section 4.11.1 on the voting plugins and their update process.

objectid – ID of the group to set voting plugin for.

plugin – the new plugin instance to set.

4.11.3 Events

```
eventtype joinGroup {
   RCType status;
   string principal;
   string objectID;
};

eventtype destroyGroup {
   RCType status;
   string objectID;
};
```

joinGroup

This event is delivered if P2PAPI_Group ::joinGroup method returned RC_INPROGRESS due to long execution time of the method.

status – status (mainly RC_OK or RC_VOTEDDOWN)

principal – Principal whose join request was voted upon.

objectID – ID of the group to join.

destroyGroup

This event is delivered if P2PAPI_Group ::destroyGroup method returned RC_INPROGRESS due to long execution time of the method.

status – status (mainly RC_OK or RC_VOTEDDOWN)

objectID – ID of the group to destroy.

4.12 P2P directory service (Distributed Hashtable, DHT)

Distributed Hashtable (DHT) is an abstraction provided by structured networks. Compared to the basic DHT operations, this API provides two additional features.

- Secrets can be associated to keys and the API user can delete keys if it knows the secret (or the hash of the secret).
- Timeout values can be associated to keys and the keys expire automatically after the timeout expires.

This specification is based on the stable features of OpenDHT [3].

4.12.1 Main API

```
interface P2PAPI_DHT {
   RCType put( in string key, in any value, in string secrethash,
in long long timeout );
   RCType get( in string key, out any value, out string secrethash,
out long long timeout );
   RCType remove( in string key, in string secret, in long long
timeout );
   RCType nextItem( in string iteratorid, out any value, out string
secrethash, out long long timeout );
   RCType closeIterator( in string iteratorid );
};
```

put

Stores value under a given key in the DHT.

key – key to store the value under.

value .- value to store.

secrethash – hash of secret that must be provided to remove or replace the value (if the key existed before), null, if not used.

timeout – timeout in seconds after which the key-value pair is removed automatically.

Returns RC_VALUEERROR if the network plugin is not able to handle the value, e.g. it is too long. Returns RC_AUTHFAILED if the value cannot be replaced because the key already exists and no appropriate secret was specified.

get

Retrieves the value, the hash of the secret and timeout value stored under the specific key. If there are multiple values stored under the key, iterator ID is returned as string value and the method returns RC_ITERATOR instead of RC_OK. Returns RC_NOVALUE if the key does not exist in the DHT.

key – key to retrieve

value – value stored under the key.

secrethash – hash of the secret associated with the key.

timeout – remaining timeout for the key-value pair.

remove

Removes a key from the DHT. Returns RC_AUTHFAILED if the secret does not match with the secret stored in the DHT.

key – the key to remove.

secret – the secret associated with the key. The hash of this secret must match that of provided in the put operation.

timeout – timeout of the remove operation. This must be longer than the remaining timeout for the key-value pair else the removed key-value pair reappears in the DHT.

nextItem

Retrieves the next value from the iterator. This method can be used if get returned with RC_ITERATOR error code. In this case get method returned iterator ID in its *value* return value. The values stored under the particular key can be obtained by calling this method using the iterator ID. Returns RC_NOVALUE if there are no more items in the iterator, in this case the iterator is also automatically closed.

iteratorid – Iterator ID returned by get method.

value – value stored under the key.

secrethash – hash of the secret associated with the key.

timeout – remaining timeout for the key-value pair.

closeIterator

Closes the iterator if, for some reason, the API user does not want to read through the operator using the nextItem method (nextItem closes the iterator automatically when attempt is made to read past the last item in the iterator).

iteratorid – ID of the iterator to close.

4.13 Decentralized Object Location and Routing (DOLR)

DOLR allows sending messages to objects published under given object ID (OID). DOLR OIDs are normally of the same length as the key length of the carrier structured network, e.g. 128 or 160 bits.

4.13.1 Main API

```
interface P2PAPI_DOLR {
   RCType publish( in sequence<octet> oid );
   RCType unpublish( in sequence<octet> oid );
   RCType sendToObj(, in sequence<octet> oid, in sequence<octet>
message);
};
```

publish

Publishes object with a given ID on this node.

oid – Object ID to publish on this node.

unpublish

Unpublishes object with given ID on this node.

oid - Object ID to unpublish on this node.

sendToObj

Sends a message to an object with given OID.

oid – Object ID to send the message to.

message – message to send.

4.13.2 Events

```
eventtype dolrReceiveEvent {
   sequence<octet> oid;
   sequence<octet> message;
};
```

dolrReceiveEvent

This event is generated when incoming message arrives from the DOLR module addressed to a given OID.

oid – Object ID the message was addressed to.

message – the message as a byte array.

4.14 CAST

Earlier versions of this document included a CAST API (see [4] for basic introduction to CAST). This support has been integrated into the group support of messaging API (see sendMessage, section 4.7.2). If there is CAST support implemented, CAST plugin can be discovered as a normal group registered under CAST_MULTICAST_<GID> and CAST_ANYCAST_<GID> group name (see section 4.6 on the discovery of the group by its group name). <GID> represents the group ID of the CAST group and according to the CAST mechanism, it has to be a node ID on the structured network. CAST group IDs are normally of the same length as the key length of the carrier structured network, e.g. 128 or 160 bits. Members can then “join” this group but the group member’s name to join must conform to the node ID used by the structured network.

CAST_ID address type was defined on the messaging interface so that sending messages can be handled by the CAST middleware and incoming CAST messages can be distinguished by the address type.

Multicast message can then be sent by invoking sendMessage with the group ID of CAST_MULTICAST_<GID> group. Similarly, anycast message can be sent by invoking sendMessage with the group ID of CAST_ANYCAST_<GID>.

CAST middleware attaches to the group hook interface of the messaging API (see section 4.7.4). When the hook interface is called with a group ID of a CAST group, the hook returns a single CAST address with CAST_ID address type. The messaging API then recognizes the message as CAST message and routes it to the CAST middleware for sending.

Incoming CAST messages are delivered as messageReceiveEvent messages (see section 4.7.3). Source address of the incoming CAST message is the group ID the message was sent to.

4.15 File sharing

File sharing API is able to publish files and file collections to download. Download is also part of this API. Downloader network technology plugins (e.g. Bittorrent) expose file sharing API but implement only file downloader method.

The API does not offer file search, that functionality is available on the discovery interface.

Components exposing file sharing API are encouraged to publish monitoring API (see section 4.5) as well. Keys in the format of fsstatus:<Object ID> are download status of files identified by <Object ID>. The value of the key is the download status in percentage as an integer value.

4.15.1 Main API

```
interface P2PAPI_FS {
   RCType publishFile( in string localfileid, in Hashtable
metainfo, in Hashtable secparams, out string oid );
   RCType unpublishFile( in string oid );
   RCType publishCollection( in string localcollectionid, in
Hashtable metainfo, in Hashtable secparams, out string oid );
   RCType unpublishCollection( in string oid );
```

```
   RCType download( in string oid, out long taskid );
   RCType suspendDownload( in long taskid );
   RCType resumeDownload( in long taskid );
};
```

publishFile

Publishes a single file identified by local file ID and meta-information supplied by the API user so that it can be found by the discovery API (see section 4.6). The file may be referenced throughout the network by its object ID also returned by the call.

localfileid – local file path of the file.

metainfo – meta-information in key-value pair format associated with the file.

secparams – Properties of a principal (see section 4.10) and encryption/message authentication properties for file publication. These may specify who published the file and what encryption/message authentication parameters are required to access it. Null if not used.

oid – Object ID assigned to the file.

unpublishFile

Unpublishes a single file identified by its object ID.

oid – Object ID of the file to unpublish.

publishCollection

Publishes a collection of files (typically a directory of files). Note that the network technology plugin is expected to publish all the files in the collection one by one but may not be able to publish the collection itself. In this case the object ID associated with the collection is a local one and may not be discovered using the discovery API.

localcollectionid – local access path of the collection, e.g. directory path.

metainfo – meta-information in key-value format associated with the collection.

secparams – Properties of a principal (see section 4.10) and encryption/message authentication properties for file collection publication. These may specify who published the collection and what encryption/message authentication parameters are required to access it. Null if not used.

oid – Object ID associated with the collection.

unpublishCollection

Unpublishes a previously published collection.

oid – Object ID associated with the collection.

download

Starts downloading a resource identified by its object ID. The actual data fragments are delivered as events.

oid – Object ID of the resource to download.

secparams – Security parameters (see section 4.10) of the principal that initiated the download.

taskid – Task ID of the download task that the task may be referenced with.

suspendDownload

Suspends the downloading task identified by its task ID.

taskid – Task ID of the download task to suspend.

resumeDownload

Resumes the downloading task identified by its task ID.

taskid – Task ID of the downloading task to resume.

4.15.2 Events

```
eventtype downloadFragment {
   long taskid;
   short percentage;
   long long offset;
   boolean lastFragment;
   sequence<octet> message;
};
```

This event is generated when a fragment from the file being downloaded is delivered to the application.

taskid – ID of the task that the fragment is related to. See download method.

percentage – estimation of the percentage of the data downloaded compared to the full length of the resource.

offset – offset of the fragment in the file.

lastFragment – true if this was the last fragment of the file.

message – the fragment itself.

4.15.3 Encryption and authentication hook

This hook allows authentication, encryption/decryption and message authentication to be applied for the file sharing service.

```
enum FSHookOps {  PUBLISHFILE, PUBLISHCOLLECTION, DOWNLOAD_IN,
DOWNLOAD_OUT };
```

```
interface P2PAPI_FS_EncryptionHook {
  RCType handleFSOp( in ServiceHookOps op, in string objectid, in
taskid, in Hashtable properties, in sequence<octet> inmessage, out
sequence<octet> outmessage );
}
```

handleFSOp

Handles a file sharing operation from authentication, encryption and message authentication point of view.

- PUBLISHFILE – Set publisher and expected encryption parameters for a single file. The file cannot be downloaded with lower security than the one specified and its origin may be verified. Note that the actual origin verification mechanism is not specified, it may be as simple as the downloader receiving the name of the principal that claims to have published the file with no signing involved.
- PUBLISHCOLLECTION – Set publisher and expected encryption parameters for a collection of files. No file from the collection can be downloaded with lower security than the one specified and every file in the collection is claimed to have the specified originator.
- DOWNLOAD_IN and DOWNLOAD_OUT – Incoming and outgoing download packages

The default implementation of the hook simply copies incoming message buffer to outgoing message buffer.

op – file sharing operation to handle.

objectid – object ID of the file/collection in question.

taskid –Task ID in case of download, else -1.

properties – security properties

inmessage – clear text outgoing download message in case of DOWNLOAD_OUT, encrypted incoming download messages in case of DOWNLOAD_IN.

outmessage –encrypted outgoing download message in case of DOWNLOAD_OUT, clear text incoming download messages in case of DOWNLOAD_IN. These are produced by the plugin.

4.15.4 Authorization hook

See general description of the hook at section 4.2.

```
const long AUTH_FSDOWNLOAD  = 7;
```

Requires authorization of incoming download operation.

subject – node and principal (e.g. user) that initiated the incoming download request. If both are given, the node and principal are separated by a newline (\n) character. Principal is

received only if the download initiator specified security parameters for downloading principal.

resource –Object ID of the file to be downloaded.

4.16 Synchronization

P2P synchronization middleware is controlled by means of this API. The API allows synchronizable data stores to be bound with each other and controlling their synchronization process. If two data stores are bound to each other, the synchronization middleware keeps them in sync if there is a connection between the nodes hosting the replicas. This means that when the nodes are connected, synchronization of the differences is performed and the changes are constantly communicated between the nodes while the nodes are connected.

Synchronization may be controlled by the application or may be initiated by the middleware if specified event is received. Currently there is one such event defined, the CONNECTED event. This event is generated when connection is available between the two nodes hosting replicas of the database.

Components exposing file sharing API are encouraged to publish monitoring API (see section 4.5) as well. Keys in the format of syncstatus:<Object ID> are synchronization status of database replicas identified by local <Object ID>. The value of the key is the synchronization status in percentage as an integer value.

4.16.1 Main API

```
interface P2PAPI_Sync {
   RCType bindDB( in string localoid, in string remoteoid );
   RCType startSync( in string localoid, out long taskid );
   RCType stopSync( in long taskid );
   RCType suspendSync( in long taskid );
   RCType resumeSync ( in long taskid );
   RCType specifyTrigger( in string localoid, in string triggerspec
);
};
```

bindDB

Binds two synchronizable data stores so that the synchronization middleware keeps them in sync.

localoid – Object ID of the local replica.

remoteoid – Object ID of the remote replica.

startSync

Starts synchronizing a previously bound data store. Synchronization will be performed with all the bound replicas that are accessible at the starting of the synchronization.

localoid – Local object ID of the data store to synchronize.

taskid – Task ID that may be used to refer to the synchronization task in progress.

stopSync

Stops a currently running synchronization task. According to the transaction mechanisms used by the synchronization middleware, the result may be no change to the original database or partially synchronized databases.

taskid – Task ID of the currently running synchronization task.

suspendsSync

Suspends a currently running synchronization task.

taskid – Task ID of the currently running synchronization task.

resumeSync

Resumes a previously suspended synchronization task.

taskid – Task ID of the previously suspended synchronization task.

specifyTrigger

Specifies trigger when the middleware should autonomously start synchronization sessions of bound databases. After binding databases, there is no automatic trigger by default, this method should be explicitly called to define such a trigger.

localoid – Object ID of the local replica to define trigger for.

triggerspec – Comma-separated list of trigger specifications. Currently only the CONNECTED trigger is specified which means that connection to a node hosting a replica was detected.

4.16.2 Events

```
eventtype startSync {
   long taskid;
   string localoid;
   sequence<string> remoteoid;
   string triggerspec;
};
```

Notifies applications that automatically triggered synchronization has been started.

taskid – Task ID of the synchronization task.

localoid – Object ID of the local replica.

remoteoid – Array of object IDs of all the remote replicas.

triggerspec – Trigger specification that started the sync task.

```
eventtype stopSync {
   long taskid;
   string localoid;
```

```
   sequence<string> remoteoid;
};
```

Notifies applications that a running synchronization task was finished.

taskid – Task ID of the synchronization task.

localoid – Object ID of the local replica.

remoteoid – Array of object IDs of all the remote replicas.

4.17 Resource management

Resource management API provides cooperative resource management for P2P components. The platform manager asks the components to restrict their resource utilization to a certain level and the components are expected to do their best to comply. Note that the platform may use additional, brute-force resource management mechanisms that forcibly prevent non-behaving components to limit their resource usage but the brute-force method may cause decreased service quality. Therefore the use of cooperative resource management and thus the implementation of this API by components with significant resource usage is encouraged.

Resources are specified in key-value format. The aggregator for this API invokes the resource management API on appropriate component by using the first part of the key, the second part is the actual resource to be set. The resource values are normally in long format. The component names are equal to the second section of the API interface names (after the P2PAPI) in lowecase, e.g. P2PAPI_Sync->sync. The key names are in <component name>:<resource name> format, e.g. sync:cpu refers to the maximum CPU utilization of the sync component.

The following resource names are defined.

cpu – maximum allowed CPU utilization. Given in kiloinstruction/second on a reference hardware (see section 4.3). The middleware uses its knowledge about the current hardware to calculate CPU load allowed on the current hardware.

memory – maximum allowed RAM utilization. Given in Kbytes.

filesystem – maximum allowed file system utilization. Given in Kbytes.

4.17.1 Main API

```
interface P2PAPI_Resman {
  RCType setResourceLimits( in Hashtable reslimits );
};
```

setResourceLimits

Sets the resource limits for a given component.

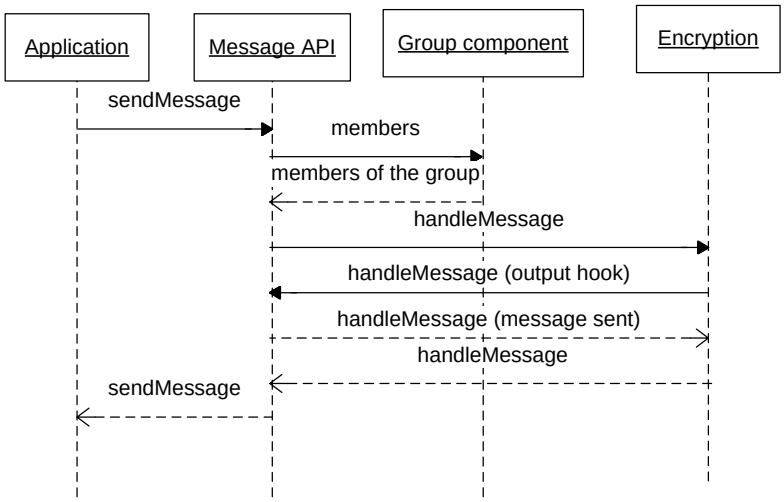
reslimits – Resource utilization key-value pairs.

5. EXAMPLE MESSAGE SEQUENCE CHARTS

In order to facilitate understanding the usage of hook interfaces, example message sequence chart diagrams are presented in this section. Note that the message sequences presented here are informative and implementation-specific. Other implementations may also be conforming to the interface specification.

5.1 Messaging API with group support added

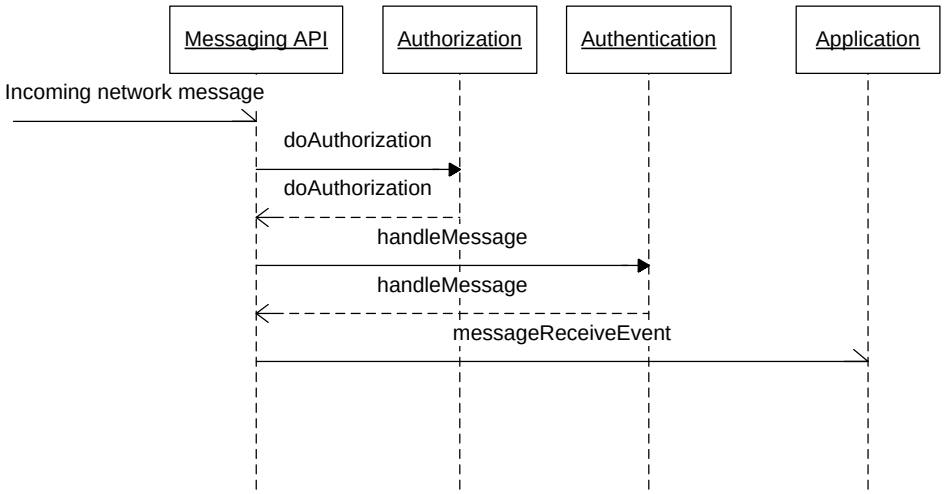
The following message sequence chart demonstrates a case when the Messaging API (see section 4.7) is called to send a message (sendMessage call) and group is specified to contain all the message destinations. The API calls the group component to obtain all the destinations then sends the messages one by one. For sending one message, the messaging API implementation invokes the Encryption hook that encrypts the message then calls back into messaging API implementation where the message is sent. Note that an alternative implementation exists when the encryption hook is called only once (the output hook of the Encryption hook does not send the message but just returns) and the messaging API distributes the same encrypted message after the encryption hook returns.



5.2 Incoming message with authorization and authentication

In this example an incoming message is subject to message source authorization and sender authentication before being delivered to the application that subscribed to this message source. The messaging API implementation receives the incoming message and hands it over to the authorization hook. The authorization hook examines the sender address and finds that messages from that host are not blocked. The message is now handed to the authentication hook. There is an username/password pair in the message so the hook checks the credentials and finds that they match. The identity of the sender and the fact that password authentication was successful are returned in the secparams hashtable. At this point the messaging API implementation sends the message to the subscribing application along with the security parameters.

Note that the messaging API implementation could call again the authorization hook; this time with the sender's authenticated identity. This is an implementation option.



6. REFERENCES

[1] Common Object Request Broker Architecture: Core Specification, V3.0.2, December 2002

[2] Uniform Resource Identifiers (URI), RFC 2396

[3] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu: OpenDHT: A Public DHT Service and Its Uses, *UC Berkeley and Intel Research.*

[4] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz and Ion Stoica: Towards a Common API for Structured Peer-to-Peer Overlays. *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003 Berkeley, CA, USA, February 21-22, 2003*