GWD-R.94
SAGA-RG

Andre Merzky
Vrije Universiteit, Amsterdam

Version: 1.0 RC.5

October 11, 2009

SAGA API Extension: Message API

Status of This Document

This document provides information to the grid community, proposing a standard for an extension to the Simple API for Grid Applications (SAGA). As such it depends upon the SAGA Core API Specification [?]. This document is supposed to be used as input to the definition of language specific bindings for this API extension, and as reference for implementors of these language bindings. Distribution of this document is unlimited.

Copyright Notice

Abstract

This document specifies a Message API extension to the Simple API for Grid Applications (SAGA), a high level, application-oriented API for grid application development. This Message API is motivated by a number of use cases collected by the OGF SAGA Research Group in GFD.70 [?], and by requirements derived from these use cases, as specified in GFD.71 [?]). The API provides a wide set of communication pattern, and targets widely distributed, loosely coupled, heterogenous applications.

Contents

1 Introduction

A significant number of SAGA use cases [?] cover data visualization systems. The common communication mechanism for this set of use cases seems to be the exchange of large messages between different applications. These applications are thereby often demand driven, i.e. require asynchronous notification of incoming messages, and react on these messages independent from their origin. Also, these use cases often include some form of pulish-subscriber mechanism, where a server provides data messages to any number of interested consumers.

This API extension is tailored to provide exactly this functionality, at the same time keeping coherence with the SAGA Core API Look-&-Feel, and keeping other Grid related boundary conditions (in particular middleware abstraction and authentication/authorization) in mind. The applicability of this package is, however, not at all limited to visualization use cases. Instead, the goal is to define a general purpose and easy to use API for event driven exchange of potentially large binary blobs of data.

It is important to note that this API is *not* intented to replace MPI [?]: where MPI is explicitely targeting tightly coupled parallel (as in 'distributed, but co-located, mostly SIMD') applications, the SAGA Message API targets loosely coupled (as in 'widely distributed, heterogeneous, mostly MIMD') applications, and is thus targeting a completely different set of communication patterns.

1.1 Notational Conventions

In structure, notation and conventions, this documents follows those of the SAGA Core API specification [?], unless noted otherwise.

1.2 Security Considerations

As the SAGA API is to be implemented on different types of Grid (and non-Grid) middleware, it does not specify a single security model, but rather provides hooks to interface to various security models – see the documentation of the `saga::context` class in the SAGA Core API specification [?] for details.

A SAGA implementation is considered secure if and only if it fully supports (i.e. implements) the security models of the middleware layers it builds upon, and neither provides any (intentional or unintentional) means to by-pass these security models, nor weakens these security models' policies in any way.

2 Requirements

The SAGA Core API specification defines a stream API package, whose purpose is to facilitat inter-process communication for distributed applications. The paradigm provided is basically that of BSD sockets: a `stream_server` instance can be created to accept incoming client connections, by calling `serve()`. The connection themself are represented by `stream` instances, which can `connect()` to `stream_servers`. The `stream` instances then allow to `read()` and `write()` binary data.

That scheme is very general, and unversally implementable on most middlewares. Experience shows, however, that most application scenarios build additional layers on top of BSD stream like APIs. Those layers usually provide

- protocols,
- simplified bootstrapping,
- higher level communication patterns,
- message encapsulation,
- message ordering,
- message verification,
- reliability,
- atomicity,
- error recovery,

or some subset thereof. Providing these features is non trivial and error prone, and results in large amount of duplicated application code. For that reason, most applications actually rely on third party implementations, like readily available p2p libraries, COM systems, etc. There exists, however, no commonly available infrastructure which covers multiple of the above properties, *and* is available for Grid environments, or other widely distributed infrastructures.

The goal of this API specification is thus to

- provide a uniform API to a wide variety of communication systems, to simplify their usage with applications;
- define a general purpose communication API which fosters the implementation and deployment of communication libraries on Grid environments;
- define communication patterns beyond MPI and P2P, the two dominant distributed message exchange systems in use today;
- do all that in the scope of the SAGA Look-&-Feel, so as to easy application integration, application portability, and semmless integration with other distributed API packages, such as security (`saga::session` and `saga::context`).

According to these goals, and in refererence to the SAGA use cases [?], the SAGA Message API should provide

1. diverse communication patterns;
2. diverse channel options: reliability, ordering, verification, atomicity, ...;
3. message abstraction (with arbitrary sized messages);
4. asynchronous communication and notification; and
5. extremely simple application bootstrapping.

It seems obvious that no single existing communication library will be able to provide the complete scope of the SAGA API. Implementations of this API are thus encouraged, or even required, to bind against different communication libraries – but that again is a declared goal of this API specification. Also, as discussed in detail in section 2.4 of the SAGA Core API specification [?], and also in the SAGA Core Experience Document [?], the design of the SAGA API enables and encourages implementations with multiple backend bindings, and in particular with late bindings.

A second inspection of the enumerated list of requirements above shows that a number of requirements is immediately solved by applying the SAGA Look-&-Feel to the Message API: in particular item (3) and (4) (message abstraction, and asynchronous communincation and notification) are intrinsically provided by SAGA, with `saga::buffer` representing messages, `saga::task` instance representing asynchronous operations, and `saga::metric` and `saga::callback` presenting means for asynchronous notification. We also would like to refer to the SAGA Advert API Extension **??**, which allows for simple bootstrapping of distributed applications, and may be of use for the purposes discussed in this document, too. The advert API will, however, not be able to provide all means for boostrapping communication patterns, and thus is not discussed in more detail here [1].

2.1 Use Case derived Requirements

More specific requirements come from the relatively large set of use cases within the SAGA group. In particular, those use cases allow to more specifically specify the scope of the required API properties listed above. Table 1 lists specific property examples to be covered by the Message API.

[1] We would like to encourage both implementors and users of the Message API to check the Advert API, as it should seemlessly integrate with the Message API, and solve bootstrapping and application coordination in many communication related use cases.

Use Case	API Properties	Requirements
#2: Cyber Infrastructure	• message encapsulation	◦ ordered messages ◦ large binary data
	• channel options	◦ secure end-to-end
#3: DIVA	• message encapsulation	◦ message encryption ◦ ordered messages ◦ async delivery ◦ low latency delivery ◦ fault tolerance ◦ typed messages ◦ large binary data
	• channel options	◦ QoS negotiation ◦ secure end-to-end ◦ low latency delivery ◦ protocol transparency
	• communication pattern	◦ dynamic node migration ◦ group bootstrapping
#13: RoboGrid	• channel options	◦ secure end-to-end
#15: Hybrid Monte Carlo Molecular Dynamics	• message encapsulation	◦ async delivery ◦ typed messages
	• channel options	◦ QoS ensurance ◦ secure end-to-end
	• communication pattern	◦ dynamic node addition
#16: Collaborative Visualization	• message encapsulation	◦ message encryption ◦ ordered messages ◦ async delivery ◦ low latency delivery ◦ typed messages ◦ large binary data
	• channel options	◦ QoS negotiation

Use Case requirements (cont.)

Use Case	API Properties	Requirements
		◦ secure end-to-end ◦ low latency delivery ◦ protocol transparency
	• communication pattern	◦ dynamic node addition ◦ node scalability ◦ group bootstrapping
#17: UCoMS Project	• message encapsulation	◦ message encryption ◦ low latency delivery ◦ large binary data
	• channel options	◦ secure end-to-end ◦ protocol transparency
	• communication pattern	◦ group bootstrapping
#18: Interactive Visualization	• message encapsulation	◦ ordered messages ◦ reliable delivery ◦ async delivery ◦ low latency delivery ◦ large binary data
	• channel options	◦ QoS negotiation ◦ low latency delivery ◦ protocol transparency
	• communication pattern	◦ group bootstrapping
#19: Interactive Image Reconstruction	• message encapsulation	◦ message encryption ◦ message signatures ◦ typed messages ◦ large binary data
	• channel options	◦ QoS negotiation ◦ secure end-to-end ◦ protocol transparency
	• communication pattern	◦ group bootstrapping

Use Case requirements (cont.)

Use Case	API Properties	Requirements
#20: Reality Grid	• message encapsulation	◦ ordered messages ◦ unordered messages ◦ async delivery ◦ low latency delivery ◦ typed messages ◦ large binary data
	• channel options	◦ secure end-to-end ◦ low latency delivery ◦ protocol transparency
	• communication pattern	◦ dynamic node addition ◦ node scalability ◦ group bootstrapping
#22: Computational Steering of Ground Water Pollution Simulations	• message encapsulation	◦ ordered messages ◦ unordered messages ◦ async delivery ◦ low latency delivery ◦ typed messages ◦ large binary data
	• channel options	◦ secure end-to-end ◦ low latency delivery ◦ protocol transparency
	• communication pattern	◦ dynamic node addition ◦ group bootstrapping
#23: Visualization Service for the Grid	• message encapsulation	◦ message encryption ◦ message signatures ◦ ordered messages ◦ unordered messages ◦ async delivery ◦ low latency delivery ◦ typed messages ◦ large binary data
	• channel options	◦ secure end-to-end ◦ low latency delivery ◦ protocol transparency

Use Case requirements (cont.)

Use Case	API Properties	Requirements
	• communication pattern	○ dynamic node addition ○ group bootstrapping

Table 1: Use Case driven requirements to the Message API. Use cases are from [**?**].

Table 1 confirms our earlier impression that the set of requirements varies widely. While we discussed earlier that no single backend will be able to cover the whole scope of requirements, the table also suggests that no single application will make use of all features to be rovided by the message API. The expected overlap both between bckend properties and application requirements is, however, so large, that it seems unwise to try to split the API package into significantly smaller units. Instead, we decided to design the API such that its components can be configured, and are inherently flexible enough, so that they are able to function well in the wide variety of use cases at hand.

3 SAGA Message API

The SAGA Message API provides a mechanism to communicate opaque messages between applications. The intent of the API package is to provide a higher level abstraction on top of the SAGA Stream API: while the exchange of opaque messages is in fact the main motivation for the SAGA Stream API, it still requires a considerable amount of user level code[2] in order to implement this use case. In contrast, this message API extension guarantees that message blocks of arbitrary size are delivered completely and intact, without the need for additional application level coordination, synchronization, or protocol.

Any compliant implementation of the SAGA Message API will imply the utilization of a communication protocol – that may, in reality, limit the interoperability of implementations of this API. This document will, however, not address protocol level interoperability – other documents outside the SAGA API scope may address it separately. [3]

This SAGA API extension inherits the `object`, `async` and `monitorable` interfaces from the SAGA Core API [?]. It CAN be implemented on top of the SAGA Stream API [ibidem].

3.1 General API Structure

Communication channels are not directly visible on API level, but their endpoints are represented by stateful instances of the `endpoint` class and its derivates. Those endpoints allow to connect to a communication channel, to accept connections from a communication channel, and to test for, send and receive messages on that communication channel. What exact type of channel the endpoint interfaces to is determined by

- the URL used to open the channel; and
- the channel properties (attributes) requested by the endpoint instances.

The type of channel behind the endpoint determines

- the set of connected endpoints on the channel (one or more);
- the properties of messages recieved on the channel; and

[2]Code is needed to run a protocol on the base SAGA stream, and to manage messages to be sent/received.

[3]**DISCUSSION (AM): This is very similar to, say, `saga::job`, where we also assume a specific backend which will in practice limit interop: jobs submitted to one bckend are unlikely to be manageable by an application binding to another backend. That is what we habe URLs for, right?**

- the availability of additional actions for operating and controling the connection (only in derivated endpoint classes, see below).

In particular the channel properties mentioned above allow the API to span the range of communication patterns targeted by this API. For example, those properties determine if the channel is reliable/unreliable, if message arrive ordered/unordered, verified/unverified, exactly-once/at-least-once/at-most-once, etc. Obviously, some combinations of channel properties will not be implementable[4] (e.g. `UnReliable AND ExactlyOnce`), but should otherwise allow to specify the required communication characteristics.

The most important property of communication channels is its `Topology`: it determines the overall communication pattern, such as the number of endpoints connected to one channel, the policy of message forwarding to multiple other endpoints, etc. Intuitive examples values of the `Topology` property are 'Peert-to-Peer', 'Point-to-Point', 'Multicast', and 'Publish-Subscriber'.[5]

Messages are encapsulated in instances of the `msg` class – a derivate of `saga::buffer` which adds some additional inspection properties (like message origin)[6]. As those message instances manage pure byte buffers (see `saga::buffer` specification in [?]), applications may usually want to derive that class further to add structure to that byte buffer, as needed. This API specification stays, however, clear of defining data models or data formats, as that would most certainly blow the this API well out of scope. Instead, domain specific data models and data formats can easily be added on application level, as described.

3.2 Endpoint URLs

The endpoint URLs used in the SAGA Message API follow the conventions layed out for the SAGA Stream API [?]: the URL's schema should allow the application to pick interoperable backends, but any backend MUST perform semantically exactly as specified in this document.

[4] or at least will not make much sense

[5] **DISCUSSION (AM): Well, those are all we have right now, really. We should check carefully if we want to support more patterns explicitely, or if we leave the rest to implicit specification via the other properties – but then we could also consider to add properties like 'NumberOfEndpoints', 'MessageForwardingPolicy', etc, to be able to really fully specify, for example, the difference between PublishSubscriber and PeerToPeer.**

[6] **DISCUSSION (AM): Should we predefine some message properties which SHOULD be available for inspection, like TTL, ID (for ordering), SendTimeStamp, RcvTimeStamp or CreationTimeStamp? What to do if the backend does not provide those? Are SAGA-impl estimates acceptable? Probably too many constraints on the backend...**

3.3 State Model

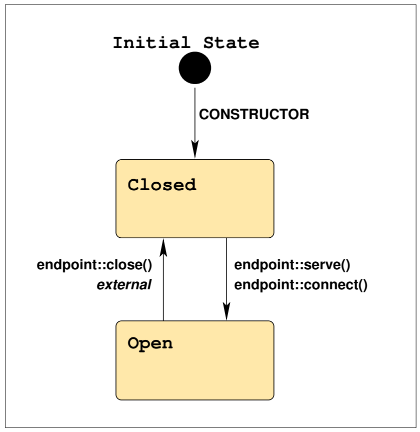

Figure 1: The SAGA Message `endpoint` state model

The state model for message `endpoint` instances is very simple: an endpoint gets constructed in `Closed` state. A successful call to `serve()` or `connect()` moves it into `Open` state, where it can send and receive messages. The endpoint stays in `Open` state as long as the backend is accepting and delivering messages – otherwise (e.g. if the peer disconnects on a Point-to-Point connction, or if a Pub-channel closes on a Publish-Subscriber backend), the endpoint is being moved back into the 'Closed' state. An explicit call to `close()` does also move the endpoint back into the `Closed` state.

Note that an `get_state()` check on an endpoint, which returns `Open`, is no guarantee that the folllowing message can be successfully transmitted: there is always a race condition of checking the state versus actually sending the message. Thus, the `test()`, `send()` and `recv()` operations can always throw an `IncorrectState` exception.[7] [8]

[7] **DISCUSSION (AM): Should there be versions of these calls which do not throw, but return errors? Try/Catch can be costly, and send/recv is all about performance. Also, we do that for file I/O!**

[8] **DISCUSSION (AM): One could imagine additional states, such as 'Serving' or 'Dropped'. 'Serving' would not really make sense though: one could not move a server endpoint out of that state – that only happens if a client connects. Similar to 'Dropped' – any check for dropped is automatically a race condition:**

```
if ( ep.state() != "Dropped" ) ep.send (msg);
```

The connection could get dropped after the test, before the send. So, we need to recover on send anyway... Also, a more detailed state model gets really complicated if multiple clients can connect, or connect/disconnect/reconnect.

3.4 Endpoint Properties

As described above: the exact backend channel which is serving a specific endpoint instance is determined by the endpoint's URL on creation time, and by the properties set on the endpoint via the SAGA Messaga API. It thus seems obvious that either (a) changes of endpoint properties lead to a disconnect of the existing backend, and move the endpoint into the `Closed` state, or (b) changes of endpoint properties are only evaluated when `connect()` or `serve()` is called (which makes inspection of endpoint properties slightly more difficult[9]). This API follows the semantic described in (b).[10]

Two endpoints which communicate with each other MUST have compatible properties [11] – otherwise the connection setup with `connect()` MUST fail.

The individual endpoint properties and their respective values are described below.

3.4.1 Connection Topology

The message API as presented here allows for four different connection topologies: `PointToPoint`, `Multicast`, `PublishSubscriber`, and `PeerToPeer`. **FIXME: check for more. Should that be extensible? How?**

- `PointToPoint` Topology:

  two parties can interchange messages in both directions (both `endpoints` can `send()` and `recv()` messages). An `PointToPoint` endpoint can only have *one* remote connection at any time. All additional connection attempts via `connect()` MUST fail with an `IncorrectState` exception. All additional incoming connections on a `serve()` MUST be declined.

- `Multicast` Topology:

  The initiating endpoint calls `serve()` – that endpoint is called 'Server'. 'Client' endpoints can `connect()` to that server. Messages sent by the Server endpoint are received by all Client endpoints. Messages sent by any Client endpoint are received *only* by the Server endpoint. A single

[9] The application has to take care of race conditions: for example, if a new endpoint gets the property 'Topology' set to 'Peer-to-Peer', and is moved into `Open` state, and the application then sets the 'Topology' to 'Point-to-Point', inspection will show 'Point-to-Point', although that value is actually only getting evaluated after reconnect, i.e after calling `'close()'` and `'connect()'`.

[10] **DISCUSSION (AM): Alternative text: All properties of `endpoint` instances are specified at the creation time of that instance: reliability level, connection topology, message ordering etc. are thus constant for the lifetime of an endpoint, and apply to all connections on that endpoint.**

[11] **DISCUSSION (AM): define 'compatible properties'! Should that be 'the same' properties'?**

endpoint can simultaneously act as a Server and as a Client, bu calling both `connect()` and `serve()` on the same endpoint instance.

- `PublishSubscriber` Topology:

  `PublishSubscriber` stands for Publish-Subscriber topology, and means that participating parties can interchange messages in both directions (all `endpoints` can `send()` and `recv()` messages). Messages sent by *any* `endpoint` are always received by *all* other clients connected to that channel. Note that a `PublishSubscriber` endpoints connected to some channel remain `Open` even if no other endpoints are subscribed (i.e. connected) to that channel.

  Calling `serve()` on a `PublishSubscriber` endpoint implies the creation of a publishing channel. If `close()` is called on that endpoint, all other endoints subscribed to that channel are disconnected.[12]

- `PeerToPeer` Topology:

  On `PeerToPeer` networks, connectivity is transitive. That means that, for example, if an endpoint **A** is connected to an endpoint **B**, which in turn is connected to an endpoint **C**, then messages from **A** will also arrive at **C**. Multiple endpoints can call `serve()` and `connect()`, in any order. PeerToPeer networks can get disconnected (in our example: if **B** fails): the backend MAY be able to continue to deliver messages from **A** to **C** and vice versa.

In either topology, the number of clients connecting *to* an applications endpoint (which calls `serve()`) can be limited by an integer argument to `serve()`. This argument is optional and defaults to `-1` (unlimited). `PointToPoint` endpoints can, however, only connect to one client at any given time. A `connect()` always implies the setup of a single connection.

Client Addressing:
In all topologies, senders can uniquely identify receivers by their id. If they do so, only that specific receiver will receive the respective message, regardless of the topology used by the endpoints (i.e. also in the Multicast, PeerToPeer and PublishSubscriber cases). A message always carries an identifier of the originating endpoint, thus messages can be answered (i.e. sent back) to the originating endpoint.

3.4.2 Reliability

The use cases addressed by the SAGA Message API cover a variety of reliable and unreliable message transfers. The level of reliability required for the message transfer is specified by an `endpoint` property. It defaults to `Reliable`.

[12] **DISCUSSION (AM): Ensure that, semantically, there can only be one publisher. For multiple publishers either use PeerToPeer, or create more endpoints.**

The available realiability levels are:

`UnReliable`:	messages MAY (or may not) reach the remote clients.
`Consistent`:	`UnReliable`, but if a message arrives at one client it MUST arrive at all clients.
`SemiReliable`:	messages MUST arrive at at least one client.
`Reliable`:	all messages MUST arrive at all clients.

Note that, for `PointToPoint` Topology, and in fact in all cases where exactly two endpoints are interconnected, `SemiReliable` degenerates to `Reliable`, and `Consistent` degenerates to `Unreliable`.

A `Reliable` implementation can obviously provide all use cases. `SemiReliable` or `Consistent` implementations also cover the `Unreliable` use case.

`Consistent` and `SemiReliable`, and more so `Reliable` semantics, do often imply a significant protocol overhead, which in particular may affect message latencies. An application should carefully evaluate what reliability requirements it actually has.

3.4.3 Atomicity

Many transport protocols guarantee that messages arrive exactly once. There are, however, many use cases where that is not strictly required. The `Atomicity` flag specifies that, and allows for more efficient policies.

The available atomicity levels are:

`AtMostOnce`:	messages arrive exactly once, or not at all.
`AtLeastOnce`:	messages are guaranteed to arrive, but may arrive more than once.
`ExactlyOnce`:	message arrive exactly once.

Obviously, an implementation which serves messages `ExactlyOnce` can serve all three use cases.

There are seemingly incompatible combinations of `Reliability` and `Atomicity`, such as for example `'UnReliable & ExactlyOnce'`. Although such a property set makes not much sense semantically, it can be provided by a `'Reliable & ExactlyOnce'` implementation.

`AtLeastOnce`, and more so `ExactlyOnce` semantics, do often imply a significant protocol overhead, which in particular may affect message latencies. An application should carefully evaluate what atomicity requirements it actually has.

3.4.4 Correctness and Completeness

The SAGA Message use cases are partly able to handle incorrect and incomplete messages (e.g. for MPEG streams). The level of correctness required for the message transfer can be specified by the `Correctness` proporty. It defaults to `Verified`.

The available correctnes levels are:

`Unverified`:	no correctness nor completeness of messages is guaranteed.
`Verified`:	Any message that is received is guaranteed to be correct and complete.

Correctness and completeness is usually be provided by adding a checksum to the message, and by verifying that checksum before delivery. That procedure usually implies significant memory, compute and latency overheads. An application should careful evaluate what correctness requirements it actually has.

3.4.5 Message Ordering

Many applications will be able to handle out-of-order messages without problems; other applications will require messages to arrive in order. The `Ordering` property allows to specify that requirement. It defaults to `Ordered`.

The available ordering levels are:

`Unordered`:	messages arrive in any order.
`Ordered`:	messages send from one client to another client arrive in the same order as they have been sent.
`GloballyOrdered`:	messages send from any client to any other client arrive in the same order as they have been sent.

In `Ordered` mode, the order of sent messages is only preserved locally – global ordering is not guaranteed to be preserved:

> *Assume three endpoints `A`, `B` and `C`, all connected to each other with `PublishSubscriber`, `Reliable`, `EactlyOnce`, `Verified`, `Ordered`. If `A`*

> *sends two messages `[a1, a2]`, in this order, it is guaranteed that both `B` and `C` receive the messages in this order `[a1, a2]`. If, however, `A` sends a message `[a1]` and then `B` sends a message `[b1]`, `C` may receive the messages in either order, `[a1, b1]` or `[b1, a1]`.*

If `GloballyOrdered`, that order is preserved, which implies either a global synchronization mechanism, or exact global timestamps.

Ordering, and in particular global ordering, usually implies significant memory, compute and latency overheads. An application should careful evaluate what ordering requirements it actually has.

3.5 Memory Management

13

Sending Messages
On sending messages, memory management (allocation and deallocation) is always performed on application level. Depending on the actual language bindings, message data will be passed by-reference (preferred) or by-value. If passed by-reference, the *implementation* MUST NOT access the memory block, and the *application* MUST NOT change the size of a message nor the content of a message while a `send()` operation with this message is in progress – the methods MAY cause an `IncorrectState` exception otherwise. If the message data block is larger than the specified size of the given `msg` instance, the message is truncated, and no error is returned. The application MUST ensure that the given message size is indeed the accessible size of the given message data block, otherwise the behavior of the `send()` is undefined.

Receiving Messages
When receiving messages, the application can choose to perform memory management for the messages itself, or to leave memory management to the implementation.

Application level memory management holds similar restrictions as listed above for sending: the *implementation* MUST NOT access the memory block, and the application MUST NOT change size or content of the message data block while the `receive()` operation is active. If the received message is larger than the size of the given `msg` instance, the message is truncated, and no error is returned. Unless the backend is able to handle that situation, there is no way to receive the remainder of the message. The application MUST ensure that

[13] **DISCUSSION (AM): This section needs to be synced with the `saga::buffer` syntax and semantics!**

the given message size is indeed the accessible size of the given message block – otherwise the behaviour of the `recv()`

Memory is managed by the API *implementation* if the `msg` instance is created with a negative `size` argument (e.g. `-1`). If the message is under implementation management, the data block of the `msg` instance gets allocated by the implementation, and MUST NOT be accessed by the application before the `receive()` operation completed successfully, nor after the `msg` instance has been deleted (e.g. went out of scope).**FIXME: check with buffer semantics!**

An implementation managed `msg` instance MUST refuse to perform a `set_size()` or `set_data()` operation, throwing an `IncorrectState` exception. A message put under implementation memory management always remains under implementation memory management, and cannot be used for application level memory management anymore. Also, a message under application memory management cannot be put under implementation management later, i.e. `set_size()` cannot be called with negative arguments – that would raise a `BadParameter` exception.

If an implementation runs out of memory while receiving a message into a implementation managed `msg` instance, a `NoSuccess` exception with the error message `''insufficient memory''` MUST be thrown.

3.6 Asynchronous Notification and Connection Management

Event driven applications are a major use case for the SAGA Message API – asynchronous notification is thus very important for this API extension. That feature is, in general, provided via the monitoring interface defined in the SAGA Core API Specification [**?**].

The available metrics on the `endpoint` class allow to monitor the `endpoint` instance for connecting, disconnecting and dropping client connections, for state changes, and of course for incoming messages. All metrics will allow to identify the respective remote party by its connection URL, which will be stored in the `RemoteID` field of the context associated with a metric change – that context is only available when using callbacks though. Alternatively, that remote party is also identifyable via the `msg` instance itseld, which can expected for sender and receiver URL (the receiver URL will usually be the endpoint URL which received the message).

Native remote endoint URLs are not always available – the implementation SHOULD in this case assign an internal URL for each client, to allow to identify clients uniquely. If the implementation can not reliably distinguish client endpoints (e.g. on some Peer-to-Peer or Publish-Subscriber backends), then it

MUST leave the respective context attribute empty, and throw a DoesNotExist exception on the message excpection.

3.7 Specification

```
package saga.message
{
  enum state
  {
    Open              = 1,
    Closed            = 2
  }

  enum topology
  {
    PointToPoint      = 1,
    Multicast         = 3,
    PublishSubscriber = 2,
    PeerToPeer        = 4
  }

  enum reliability
  {
    UnReliable        = 1,
    Consistent        = 3,
    SemiReliable      = 2,
    Reliable          = 4
  }

  enum atomicity
  {
    AtMostOnce        = 1,
    AtLeastOnce       = 2,
    ExactlyOnce       = 3
  }

  enum correctness
  {
    Unverified        = 1,
    Verified          = 2
  }

  enum ordering
```

```
{
  Unordered         = 1,
  Ordered           = 2,
  GloballyOrdered   = 3
}


class msg : implements   saga::buffer
         // from buffer  saga::object
         // from object  saga::error_handler
{
  CONSTRUCTOR  (in    array<byte>   data = 0,
                in    int           size = 0,
                out   msg           obj);
  DESTRUCTOR   (in    msg           obj);

  set_receiver (in    int           receiver_id);
  get_sender   (out   int           sender_id);
}


interface endpoint : implements   saga::object
                     implements   saga::async
                     implements   saga::monitorable
                  // from object  saga::error_handler
{
  // inspection methods
  get_url       (out   string        url);
  get_receivers (out   array<string> urls);

  // management methods
  serve         (in    int           n        = -1  );
  connect       (in    string        url      = "",
                 in    float         timeout  = -1.0);
  close         (void);

  // I/O methods
  send          (in    msg           msg,
                 in    url           receiver = "",
                 in    float         timeout  = -1.0);
  test          (out   int           size,
                 in    url           sender   = "",
                 in    float         timeout  = -1.0);
  recv          (inout msg           msg,
                 in    url           sender   = "",
                 in    float         timeout  = -1.0);
```

```
// Attributes:
//   name:  State
//   desc:  endpoint state in respect to the state diagram
//   mode:  ReadOnly
//   type:  Enum
//   value: -
//   notes: possible values: 'Open' or 'Closed'
//
//   name:  Topology
//   desc:  informs about the connection topology
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "PointToPoint"
//
//   name:  Reliability
//   desc:  informs about the reliability level
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Reliable"
//
//   name:  Atomicity
//   desc:  informs about the atomicity level
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "ExactlyOnce"
//
//   name:  Correctness
//   desc:  informs about the message correctness
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Verified"
//
//   name:  Ordering
//   desc:  informs about the message ordering
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Ordered"
//
//
// Metrics:
```

```
    //   name:  State
    //   desc:  fires if the endpoint state changes
    //   mode:  Read
    //   unit:  1
    //   type:  Enum
    //   value: ""
    //   notes: - has the literal value of the endpoints
    //          state attribute
    //
    //   name:  Connect
    //   desc:  fires if a receiver connects
    //   mode:  Read
    //   unit:  1
    //   type:  String
    //   value: ""
    //   notes: - this metric can be used to perform
    //            authorization on the connecting receivers.
    //          - the value is the endpoint URL of the
    //            remote party, if known.
    //
    //   name:  Closed
    //   desc:  fires if the connection gets closed by
    //          the remote endpoint
    //   mode:  Read
    //   unit:  1
    //   type:  String
    //   value: ""
    //   notes: - the value is the endpoint id of the
    //            remote party, if known.
    //
    //   name:  Message
    //   desc:  fires if a message arrives
    //   mode:  Read
    //   unit:  1
    //   type:  String
    //   value: ""
    //   notes: - the value is the endpoint id of the
    //            sending party, if known.
  }
-
-   class endpoint_simple : implements   saga::endpoint
-                     // from endpoint  saga::object
-                     // from endpoint  saga::async
-                     // from endpoint  saga::monitorable
-                     // from object    saga::error_handler
-   {
```

```
-     CONSTRUCTOR   (in    session       session,
-                    in    string        url          = "",
-                    in    int           topology     = PointToPoint,
-                    in    int           reliablility = Reliable,
-                    in    int           atomicity    = ExactlyOnce,
-                    in    int           ordering     = Ordered,
-                    in    int           correctness  = Verified,
-                    out   sender        obj);
-     DESTRUCTOR    (in    sender        obj);
-   }
-
-   class endpoint_multicast : implements   saga::endpoint
-                          // from endpoint  saga::object
-                          // from endpoint  saga::async
-                          // from endpoint  saga::monitorable
-                          // from object    saga::error_handler
-   {
-     CONSTRUCTOR   (in    session       session,
-                    in    string        url          = "",
-                    in    int           topology     = Multicast,
-                    in    int           reliablility = Reliable,
-                    in    int           atomicity    = ExactlyOnce,
-                    in    int           ordering     = Ordered,
-                    in    int           correctness  = Verified,
-                    out   sender        obj);
-     DESTRUCTOR    (in    sender        obj);
-   }
-
-   class endpoint_pub_sub : implements   saga::endpoint
-                        // from endpoint  saga::object
-                        // from endpoint  saga::async
-                        // from endpoint  saga::monitorable
-                        // from object    saga::error_handler
-   {
-     CONSTRUCTOR   (in    session       session,
-                    in    string        url          = "",
-                    in    int           topology     = PublishSubscriber,
-                    in    int           reliablility = Reliable,
-                    in    int           atomicity    = ExactlyOnce,
-                    in    int           ordering     = Ordered,
-                    in    int           correctness  = Verified,
-                    out   sender        obj);
-     DESTRUCTOR    (in    sender        obj);
-
-     list_channels (out  array<std::string> channels);
-
```

```
-     join          (in    string        channel);
-     leave         (in    string        channel);
-
-     // I/O methods
-     send          (in    string        channel,
-                    in    float         timeout  = -1.0,
-                    in    msg           msg);
-     test          (in    string        channel,
-                    in    float         timeout  = -1.0,
-                    out   int           size);
-     recv          (in    string        channel,
-                    in    float         timeout  = -1.0,
-                    inout msg           msg);
-   }
-
-   class endpoint_peer_to_peer : implements   saga::endpoint
-                           // from endpoint  saga::object
-                           // from endpoint  saga::async
-                           // from endpoint  saga::monitorable
-                           // from object    saga::error_handler
-   {
-     CONSTRUCTOR   (in    session       session,
-                    in    string        url          = "",
-                    in    int           topology     = PeerToPeer,
-                    in    int           reliablility = UnReliable,
-                    in    int           atomicity    = Unknown,
-                    in    int           ordering     = UnOrdered,
-                    in    int           correctness  = Verified,
-                    out   sender        obj);
-     DESTRUCTOR    (in    sender        obj);
-   }
  }
```

3.8 Specification Details

class msg

The msg object encapsulates a sequence of bytes to be communicated between applications. A msg instance can be sent (by an endpoint calling send()), or received (by an endpoint calling recv()). A message does not belong to a session, and a msg object instance can thus be used in multiple sessions, for multiple endpoints.

```
- CONSTRUCTOR
  Purpose:  create a new message object
  Format:   CONSTRUCTOR          (in  int      size = 0,
                                  out sender   obj);
  Inputs:   size:                 the size of the message
  Outputs:  obj:                  new message object
  Throws:   NotImplemented
            NoSuccess
  Notes:    - see notes on memory management


- DESTRUCTOR
  Purpose:  Destructor for sender object.
  Format:   DESTRUCTOR           (in  sender obj)
  Inputs:   sender:               object to be destroyed
  Outputs:  -
  Throws:   -
  PostCond: - the connection is closed..
  Notes:    - see notes on memory management.


- set_size
  Purpose:  set the size of the message data buffer
  Format:   set_size             (in  int  size);
  Inputs:   size:                 size of data buffer
  Outputs:  -
  Throws:   NotImplemented
            BadParameter
            IncorrectState
            NoSuccess
  Notes:    - see notes on memory management.
            - size must be positive, otherwise a
              'BadParameter' exception is thrown.
            - set_size() cannot be called on an
              implementation managed msg instance.
              That raises a 'IncorrectState' exception.
            - the method does not cause a memory resize etc,
              but merely informs the implementation on the
              size to be used for the data buffer on send()
              or recv().


- get_size
  Purpose:  get the size of the message data buffer
```

```
  Format:   get_size             (out  int  size);
  Inputs:   -
  Outputs:  size:                 size of data buffer
  Throws:   NotImplemented
            NoSuccess
  Notes:    - see notes on memory management.
            - on application managed messages, the call
              returns exactly the value which was set during
              construction, or via set_size().
            - on implementation managed buffers, the call
              returns the currently allocated buffer size.
              That size can reliably be used to access the
              data buffer.


- set_data
  Purpose:  set the data buffer for the message
  Format:   set_data             (inout array<byte> buffer);
  Inputs:   -
  InOuts:   buffer                data buffer for message
  Outputs:  -
  Throws:   NotImplemented
            IncorrectState
            NoSuccess
  Notes:    - see notes on memory management.
            - set_data() cannot be called on an
              implementation managed msg instance.
              That raises a 'IncorrectState' exception.
            - the given data buffer will not be resized, or
              reallocated, or deallocated by the
              implementation, but only read from or written
              to.  In can thus be, for example, a mmapped
              memory segment.


- get_data
  Purpose:  get the data buffer for the message
  Format:   get_data             (out array<byte> buffer);
  Inputs:   -
  Outputs:  buffer                data buffer for message
  Throws:   NotImplemented
            NoSuccess
  Notes:    - see notes on memory management.
            - get_data() returns the current message buffer.
              Depending on the language binding, that can be
              a reference to the actual buffer (which avoids
```

```
                memcopies, preferred), or a copy of the
                message buffer.
              - if a reference is returned for a implementation
                managed msg instance, that reference MUST NOT
                be changed by the application, and MUST NOT be
                accessed after the msg instance is destroyed,
                e.g. goes out of scope.
              - the returned buffer may be empty or NULL.
```

---

class endpoint

The endpoint object represents a connection endpoint for the message exchange, and can `send()` and `recv()` messages. It can be connected to other endpoints (`connect()`), and can be contacted by other endpoints (`serve()`). All other endpoints connected to the `endpoint` instance will receive the messages sent on that `endpoint` instance. The `endpoint` instance will also receive all messages sent by any of the other endpoints (global order is not guaranteed to be preserved!).

```
  - CONSTRUCTOR
    Purpose:  create a new endpoint object
    Format:   CONSTRUCTOR      (in  session  session,
                                in  string   url         = "",
                                in  int      reliable    = 1,
                                in  int      topology    = 1,
                                in  int      ordering    = 1,
                                in  int      correctness = 1,
                                    out endpoint obj);
    Inputs:   session:              session to be used for
                                    object creation
              url:                  specification for
                                    connection setup (serving)
              reliable:             flag defining transfer
                                    reliability
              topology:             flag defining connection
                                    topology
              ordering:             flag defining message
                                    ordering
    Outputs:  obj:                  new endpoint object
```

```
  Throws:   NotImplemented
            IncorrectURL
            AuthorizationFailed
            AuthenticationFailed
            PermissionDenied
            NoSuccess
  PostCond: - the endpoint is in 'New' state, and can now
              serve client connections (see serve()), or
              connect to other endpoints (see connect()).
            - the given URL can be used to specify the
              protocol, network interface, port number etc
              which are to be used for the serve() method.
              The URL can be empty - the implementation
              will then use default values.  These defaults
              MUST be documented by the implementation.
            - the URL error semantics as defined in the SAGA
              Core API specification applies.


- DESTRUCTOR
  Purpose:  Destructor for sender object.
  Format:   DESTRUCTOR           (in  sender obj)
  Inputs:   sender:               object to be destroyed
  Outputs:  -
  Notes:    -


inspection methods:
-------------------

- get_url
  Purpose:  get URL to be used to connect to this server
  Format:   get_url              (out string url);
  Inputs:   -
  Outputs:  url:                  string containing the
                                  contact URL of this
                                  endpoint.
  Throws:   NotImplemented
            IncorrectState
  Notes:    - returns a URL which can be passed to the
              receiver constructor to create a client
              connection to this endpoint.
            - this method can only be called after serve()
              has been called - otherwise an
              'IncorrectState' exception is thrown.  The
              return of a URL does not imply a guarantee
```

```
                that a endpoint can successfully connect with
                this URL (e.g.  the URL may be outdated on
                'Closed' endpoints).

    - get_receivers
      Purpose:  get the endpoint URLs of connected clients
      Format:   get_url              (out array<string> urls);
      Inputs:   -
      Outputs:  urls:                 endpoint URLs of connected
                                      clients.
      PreCond:  - the sender is in 'Open' state.
      Throws:   NotImplemented
                IncorrectState
      Notes:    - the method causes an 'IncorrectState'
                  exception if the sender instance is not in
                  'Open' state.
                - the returned list can be empty
                - if a remote endpoint does not has a URL (e.g.
                  if it did not yet call serve()), the
                  returned array element is an empty string.
                  That allows to count the connected clients.


    management methods:
    -------------------

    - serve
      Purpose:  start to serve incoming client connections
      Format:   serve                (in  int    n   = -1);
      Inputs:   n:                    number of clients to
                                      accept
      Outputs:  -
      Throws:   IncorrectState
                NoSuccess
      PreCond:  - the endpoint is in 'New' or 'Open' state, but
                  did not yet call serve().
      PostCond: - the endpoint is in 'Open' state, and accepts
                  client connections.
      Notes:    - if the endpoint is not in 'New' or 'Open' state
                  when this method is called, or if serve() was
                  called on this instance before, an
                  'IncorrectState' exception is thrown.
                - a diconnect()'ed endpoints cannot serve() again
                  (it is in 'Closed' state).
                - 'n' defines the number of clients to accept.
                  If that many clients have been accepted
```

```
                 successfully (e.g. messages could have been
                 sent to / received from these clients), the
                 serve call finishes.
               - if 'n' is set tp '-1', the default, no limit
                 on the accepted clients is applied.  The call
                 then blocks indefinitely.


  - connect
    Purpose:  connect to another endpoint
    Format:   connect              (in  float  timeout = -1.0,
                                    in  string url);
    Inputs:   timeout:              seconds to wait
              url:                  specification for
                                    connection setup
    Outputs:  -
    Throws:   IncorrectState
              IncorrectURL
              AuthorizationFailed
              AuthenticationFailed
              PermissionDenied
              Timeout
              NoSuccess
    PreCond:  - the endpoint is in 'New' or 'Open' state.
    PostCond: - the endpoint is in 'Open' state, and can
                send and receive messages.
    Notes:    - if the endpoint is not in 'New' or 'Open'
                state when this method is called, an
                'IncorrectState' exception is thrown.
              - a close()'ed endpoint cannot be connect()'ed
                again (it is in 'Closed' state).
              - if reliability level, connection topology
                or message ordering of the connecting
                and connected endpoint do not match, the
                method fails with a 'NoSuccess' exception,
                and a descriptive error message.
              - the URL error semantics as defined in the
                SAGA Core API specification applies.
              - the timeout semantics as defined in the
                SAGA Core API specification applies.


  - close
    Purpose:  disconnect from all backend channels
    Format:   close                (in  float timeout = -1.0);
    Inputs:   timeout:              seconds to wait
```

```
  Outputs:  -
  Throws:   NotImplemented
            Timeout
            NoSuccess
  PreCond:  -
  PostCond: - the endpoint is in 'Closed' state.
  Notes:    - it is no error to call close() on a 'Closed'
              endpoint.
            - a close()'ed endpoint can serve() or
              connect() again.
            - the timeout semantics as defined in the
              SAGA Core API specification applies.


I/O methods:
------------

- send
  Purpose:  send a message to all connected endpoints
  Format:   serve                (in  float timeout = -1.0,
                                  in  msg msg);
  Inputs:   timeout:              seconds to wait
            msg:                  message to send
  Outputs:  -
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  Notes:    - if the endpoint is not in 'Open' state when
              this method is called, an 'IncorrectState'
              exception is thrown.
            - error reporting is non-trivial, as some
              message transfer may succeed for some clients,
              and not for others.  For reliable transfers,
              or 'Verified' correctness, the method MUST
              raise a 'NoSuccess' exception with detailed
              information about the clients the transport
              failed for.  For unreliable transfer, the
              method MAY raise such an exception if the
              implementation deems the error condition
              severe enough to disrupt the communication
              altogether (i.e. future messages are unlikely
              to get through).  Again, the exception must
              then give detailed information on the
              client(s) which failed.  For 'Unverified'
              Correctness, such an exception MUST NOT be
```

```
                raised.
              - a timeout can happen for all or for one
                client - the returned error MUST indicate
                which is the case, and which clients failed.
              - the implementation MUST carefully document its
                possible error conditions.
              - if the endpoint reached the 'Open' state by
                calling serve(), and did not call connect(),
                no client endpoint may be connected to this
                endpoint instance.  That does not cause an
                error, but the message is silently discarded.
              - the timeout semantics as defined in the
                SAGA Core API specification applies.

  - test
    Purpose:  test if a message is available for receive
    Format:   test                 (in  float timeout = -1.0,
                                    out int   size);
    Inputs:   timeout:              seconds to wait
              size:                 size of incoming message
    Outputs:  -
    Throws:   NotImplemented
              IncorrectState
              NoSuccess
    Notes:    - if the endpoint is not in 'Open' state when
                this method is called, an 'IncorrectState'
                exception is thrown.
              - if the endpoint reached the 'Open' state by
                calling serve(), and did not call connect(),
                no client endpoint may be connected to this
                endpoint instance.  That does not cause an
                error -- the method will wait for the
                specified timeout.  The implementation MUST
                respect messages originating from connections
                which have been established during the timeout
                waiting time.
              - if no message is available for recv() after
                the timeout, the method returns (it does not
                throw a 'Timeout' exception).  The returned
                size is set to -1.
              - if a message is available for recv(), the
                returned size is set to the size of the
                incoming messages data buffer.  The size MUST
                be a valid value to be used to construct a new
                msg object instance.  The message for which
                the size was returned MUST be the message
```

```
                  which is returned on the next initiated recv()
                  call.
                - if any (synchronous or asynchronous) recv()
                  calls are in operation while test is called,
                  they MUST NOT be served with the incoming
                  message if size is returned as positive value.
                  Instead, the next initiated recv() call get
                  served.
                - the timeout semantics as defined in the
                  SAGA Core API specification applies.

    - recv
      Purpose:  receive a message from remote endpoints
      Format:   test                 (in    float timeout = -1.0,
                                      inout msg   msg);
      Inputs:   timeout:              seconds to wait
      InOuts:   msg:                  received message
      Outputs:  -
      Throws:   NotImplemented
                IncorrectState
                Timeout
                NoSuccess
      Notes:    - if the endpoint is not in 'Open' state when
                  this method is called, an 'IncorrectState'
                  exception is thrown.
                - if the endpoint reached the 'Open' state by
                  calling serve(), and did not call connect(),
                  no client endpoint may be connected to this
                  endpoint instance.  That does not cause an
                  error -- the method will wait for the
                  specified timeout.  The implementation MUST
                  respect messages originating from connections
                  which have been established during the timeout
                  waiting time.
                - error reporting is non-trivial, as some
                  message transfer may succeed for some clients,
                  and not for others.  For reliable transfers,
                  or 'Verified' correctness, the method MUST
                  raise a 'NoSuccess' exception with detailed
                  information about the clients the transport
                  failed for.  For unreliable transfer, the
                  method MAY raise such an exception if the
                  implementation deems the error condition
                  severe enough to disrupt the communication
                  altogether (i.e. future messages are unlikely
                  to get through).  Again, the exception must
```

```
              then give detailed information on the
              client(s) which failed.  For 'Unverified'
              Correctness, such an exception MUST NOT be
              raised.
            - if no message is available for recv() after
              the timeout, the method throws a 'Timeout'
              exception.  The application must use test() to
              avoid this.
            - the timeout semantics as defined in the
              SAGA Core API specification applies.
```

3.9 Examples

TO BE DONE

4 Intellectual Property Issues

4.1 Contributors

This document is the result of the joint efforts of several contributors. The authors listed here and on the title page are those committed to taking permanent stewardship for this document. They can be contacted in the future for inquiries about this document.

Andre Merzky
andre@merzky.net
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1083
1081HV Amsterdam
The Netherlands

The initial version of the presented SAGA API was drafted by members of the SAGA Research Group. Members of this group did not necessarily contribute text to the document, but did contribute to its current state. Additional to the authors listed above, we acknowledge the contribution of the following people, in alphabetical order:

Andrei Hutanu (LSU), Hartmut Kaiser (LSU), Pascal Kleijer (NEC), Thilo Kielmann (VU), Gregor von Laszewski (ANL), Shantenu Jha (LSU), and John Shalf (LBNL).

4.2 Intellectual Property Statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

4.3 Disclaimer

This document and the information contained herein is provided on an "As Is" basis and the OGF disclaims all warranties, express or implied, including but not limited to any warranty that the use of the information herein will not infringe any rights or any implied warranties of merchantability or fitness for a particular purpose.

4.4 Full Copyright Notice

FIXME: clarify data format/data model/byte ordering etc. issues
FIXME: Check with WS-Notification, WS-Eventing, WS-Relaibility and WS-ReliabaleMessaging.
FIXME: point out the saga core sections used (task, attrib, . . .)
FIXME: add examples, also for async and monitoring
FIXME: recv $->$ receive