GWD-R.94
SAGA-RG

Version: 1.0 RC.5

Andre Merzky
Vrije Universiteit, Amsterdam

January 15, 2008

SAGA API Extension: Message Bus 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 [1]. 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 Bus API extension to the Simple API for Grid Applications (SAGA), a high level, application-oriented API for grid application development. This Message Bus API is motivated by a number of use cases collected by the OGF SAGA Research Group in GFD.70 [2], and by requirements derived from these use cases, as specified in GFD.71 [3]). It adds an additional layer of abstraction to the SAGA Stream API, which is described in the SAGA Core API specification [1].

Contents

1 Introduction

A significant number of SAGA use cases [2] 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 (publish/subscribe).

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.

1.1 Notational Conventions

In structure, notation and conventions, this documents follows those of the SAGA Core API specification [1], 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 [1] 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

3 SAGA Message API

3.1 Introduction

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: the exchange of opaque messages is in fact the main motivation for the SAGA Stream API, but it requires a considerable amount of user level code in order to implement this use case with the current SAGA Stream API. 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 or synchronization.

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.

This SAGA API extension inherits the `object`, `async` and `monitorable` interfaces from the SAGA Core API [1]. It CAN be implemented on top of the SAGA Stream API [ibidem].

3.1.1 Endpoint URLs

The endpoint URLs used in the SAGA Message API follow the conventions layed out for the SAGA Stream API [1].

3.1.2 State Model

The state model for message `endpoint` instances is very simple: an endpoint gets constructed in `New` state. A successful call to `serve()` or `connect()` moves it into `Open` state, where it can send and receive messages. A call to `close()` moves it into the only final state, `Closed`.

Note that the `Open` state does not imply any active connection. E.g., no client may have connected yet after `serve()` has been called. Or a connection which has been established with `connect()` may have been dropped by the remote side. The `Open` state only signals that the methods `send()` and `recv()` can be called on the endpoint instance. These methods will fail gracefully of no connection is active: `send()` will silently discard the message to send, and `recv()` will block

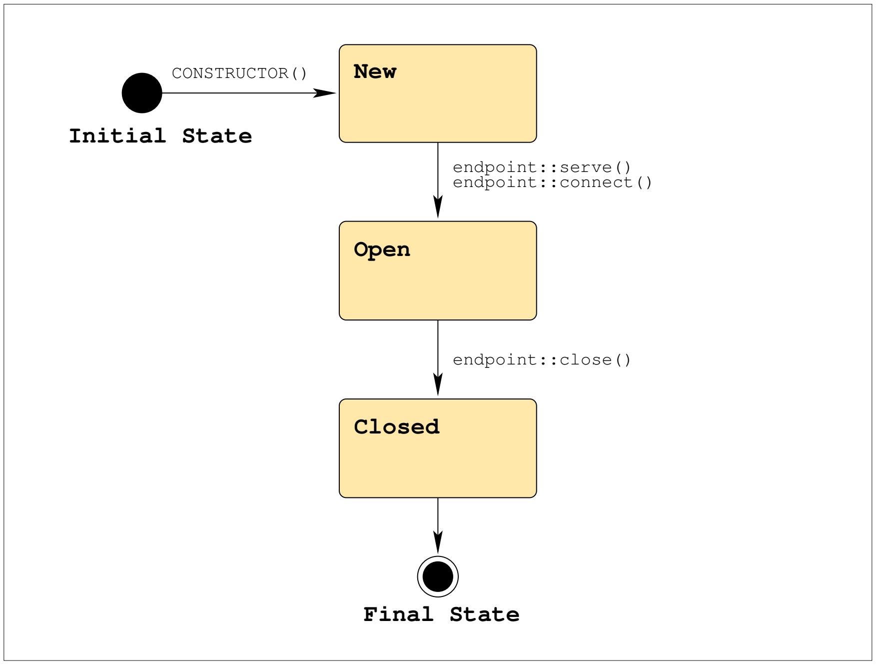

Figure 1: The SAGA Message `endpoint` state model

until a connection is (re-)established, and a new message arrives.

3.1.3 Classes

The SAGA Message API consists of two classes: a `msg` class, encapsulating an opaque message to sent, or an opaque message received; and a `endpoint` class, representing the sending and receiving end for a sequence of opaque messages.

A message sent by an `endpoint` is received by all `endpoint`s which `connect()`ed to that sending `endpoint`. A `endpoint` can `test()` for the availability of a message, and can `receive()` it. A `endpoint` can also be notified of incoming messages, by using the asynchronous notification mechanisms of the `monitorable` interface, as described in [1].

3.1.4 Memory Management

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 message data memory block before a `send()` operations starts, nor after the `send()` operation finishes. 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 would cause an `IncorrectState` exception then. If the message data block is larger than the 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 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.

For application level memory management hold similar restrictions as listed above for sending: the implementation MUST NOT access the memory block before or after the `recv()` operation is active, 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. The Application MUST ensure that the given message size is indeed the accessible size of the given message block.

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).

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.

3.1.5 Asynchronous Notification and Connection Management

Event driven applications are a major use case for the SAGA Message API – asynchronous notification is thus of some importance for this API extension. It is, in general, provided via the monitoring interface defined in the SAGA Core API Specification [1].

The available metrics on the `endpoint` class allow to monitor the `endpoint` instance for connecting, disconnecting and dropping client connections, for state changes, and for incoming messages. The last is probably the most important metric, and allows to receive messages asynchronously.

The connection inspection metrics, `RemoteConnect`, `RemoteDisconnect`, and `RemoteDropped` try to identify the respective remote party by its connection URL. That URL is, however, not always always available, and the notification mechanism may not allow the application to distinguish which client failed. That is, at the moment, intentional: we imagine the main use case to be the publisher/subscriber model, where a server serves any number of interested clients, and where clients receive data from usually one service. Also, we think that it is, in most use cases, unimportant from where a message originates.

Harder requirements on connection management would imply, in our opinion, either (a) a much more complex API, or (b) a point-to-point connection paradigm (such as the SAGA Streams, i.e. without support for publish/subscriber).

3.1.6 Endpoint Properties

All properties of `endpoint` instances are specified at the creation time of that instance: reliability level, connection topology, and message ordering are thus constant for the lifetime of an endpoint, and for all connections on that endpoint. Two endpoints which communicate with each other MUST have the same properties – otherwise the connection setup with `connect()` will fail with an `NoSuccess` exception.

3.1.7 Connection Topology

The message API as presented here allows for two different connection topologies: `PointToPoint` and `MessageBus`. It defaults to `PointToPoint`.

In either topology, the number of clients connecting to a server (which called `serve()` can be limited by an integer argument to `serve()`. This argument is optional and defaults to `-1` (no limit). A `connect()` always implies the setup of a single connection.

`PointToPoint` **Topology:** `PointToPoint` topology means that two participating parties can interchange messages in both directions (both `endpoint`s can `send()` and `recv()` messages). At the same time, an `endpoint` can be connected to multiple remote parties, which all `recv()` the messages sent by this `endpoint`, and which can all `send()` messages to this `endpoint`. However, messages sent to a remote party are received *only* by that party, and are not received by any other clients connected to that party.

`MessageBus` **Topology:** `MessageBus` topology means that two participating parties can interchange messages in both directions (both `endpoint`s can `send()` and `recv()` messages). However, messages sent to an `endpoint` are also received by *all* other clients connected to that `endpoint` (this property is transitive).

In this topology, all endpoints which are (directly or indirectly) connected to each other receive all messages sent from any of the connected `endpoint`s to any other one.

3.1.8 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 can be specified by a flag on the creation of an `endpoint` instance. It defaults to `Reliable`.

The available realiability levels are:

`Unreliable`:	messages MAY (or may not) reach the remote clients, but at-most-once.
`Atomic`:	`Unreliable`, but a message received by one client is guaranteed to (MUST) arrive at all clients, but at-most-once.
`SemiReliable`:	messages are guaranteed to (MUST) arrive at all clients, but may arrive more than once.
`Reliable`:	all messages are guaranteed to (MUST) arrive at all clients, but at-most-once.

If a connection setup requires `unreliable` message transfer, the implementation CAN be `unreliable`, `atomic` or `reliable`. If it requires `atomic` transfer, the implementation CAN be `atomic` or `reliable`. If it requires `reliable` transfer, the implementation MUST be `reliable`.

3.1.9 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 a flag on the creation of an `endpoint` instance. It defaults to `Verified`.

The available realiability levels are:

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

3.1.10 Message Ordering

The mode of the message ordering can be specified by a flag on the creation of an `endpoint` instance. It defaults to `Ordered`.

The available modes are:

`Ordered`:	messages arrive in the same order as they have been sent.
`Unordered`:	messages arrive in any order.

In `Ordered` mode, the order of sent messages is noly preserved locally – global ordering is never guaranteed to be preserved:

> *Assume three endpoints `A`, `B` and `C`, all connected to each other. 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]`.*

3.2 Specification

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

```
    Closed         =  3
  }

  enum reliability
  {
    Reliable       =  1,
    Atomic         =  2,
    SemiReliable   =  3,
    Unreliable     =  4
  }

  enum topology
  {
    PointToPoint   =  1,
    MessageBus     =  2
  }

  enum ordering
  {
    Ordered        =  1,
    Unordered      =  2
  }

  enum correctness
  {
    Verified       =  1,
    Unverified     =  2
  }

  enum managed
  {
    Application    =  1,
    Implementation =  2
  }


  class msg : implements saga::error_handler
              implements saga::attribute
  {
    CONSTRUCTOR  (in    int           size = 0,
                  in    int           managed = 1;
                  out   msg           obj);
    DESTRUCTOR   (in    msg           obj);

    set_size     (in    int           size);
    get_size     (out   int           size);
```

```
  set_data     (inout array<byte>   buffer);
  get_data     (out   array<byte>   buffer);

  // Attributes:
  //   name:  Managed
  //   desc:  informs about the memory management
  //          mode
  //   mode:  ReadOnly
  //   type:  Enum
  //   value: "Application"
}

class endpoint : implements   saga::object
                 implements   saga::async
                 implements   saga::monitorable
              // from object  saga::error_handler
{
  CONSTRUCTOR   (in    session       session,
                 in    string        url         = "",
                 in    int           reliable    = 1,
                 in    int           topology    = 1,
                 in    int           ordering    = 1,
                 in    int           correctness = 1,
                 out   sender        obj);
  DESTRUCTOR    (in    sender        obj);

  // inspection methods
  get_url       (out   string        url);
  get_receivers (out   array<string> urls);

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

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

  // Attributes:
```

```
//   name:  Reliability
//   desc:  informs about the reliability level
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Reliable"
//
//   name:  Topology
//   desc:  informs about the connection topology
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "PointToPoint"
//
//   name:  Ordering
//   desc:  informs about the message ordering
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Ordered"
//
//   name:  Correctness
//   desc:  informs about the message correctness
//          of the endpoint
//   mode:  ReadOnly
//   type:  Enum
//   value: "Verified"
//
//
// Metrics:
//   name:  State
//   desc:  fires if the sender state changes
//   mode:  Read
//   unit:  1
//   type:  Enum
//   value: "New"
//
//   name:  RemoteConnect
//   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:  RemoteDisconnect
    //   desc:  fires if a receiver disconnects or the
    //          connection dropped
    //   mode:  Read
    //   unit:  1
    //   type:  String
    //   value: ""
    //   notes: - the value is the endpoint URL of the
    //            remote party, if known.
    //
    //   name:  RemoteDropped
    //   desc:  fires if the connection gets dropped by
    //          the remote sender
    //   mode:  Read
    //   unit:  1
    //   type:  String
    //   value: ""
    //   notes: - the value is the endpoint URL 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 URL of the
    //            sending party, if known.
  }
}
```

3.3 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 `endpoint`s.

```
- 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 close()'d 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()'d 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:  close the endpoint, and release all
                resources
      Format:   close                (in  float timeout = -1.0);
```

```
  Inputs:   timeout:             seconds to wait
  Outputs:  -
  Throws:   NotImplemented
            IncorrectState
            Timeout
            NoSuccess
  PreCond:  - the endpoint is in 'Open' state.
  PostCond: - the endpoint is in 'Closed' state.
  Notes:    - if the endpoint is not in 'Open' state when
              this method is called, an 'IncorrectState'
              exception is thrown.
            - the timeout semantics as defined in the
              SAGA Core API specification applies.
            - a close()'d endpoint cannot serve() or
              connect() again.


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.
```

3.4 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
FIXME: /

References

[1] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). Grid Forum Document GFD.xx, 2007. Global Grid Forum.

[2] A. Merzky and S. Jha. A Collection of Use Cases for a Simple API for Grid Applications. Grid Forum Document GFD.70, 2006. Global Grid Forum.

[3] A. Merzky and S. Jha. A Requirements Analysis for a Simple API for Grid Applications. Grid Forum Document GFD.71, 2006. Global Grid Forum.