GFD-E.5
Scheduling Working Group
Category: EXPERIMENTAL

Alain Roy
University of Wisconsin-Madison
Volker Sander
Forschungszentrum Jülich GmbH

May 23rd, 2002

Advance Reservation API

Status of this Draft

This document specifies an experimental grid working draft for the Grid scheduling community. Discussion and suggestions are requested. Distribution of this memo is unlimited.

GGF EDITOR NOTE: This document is EXPERIMENTAL and is not intended to specify a Grid recommendation or standard, nor is this document a “recommendations track” document. It is intended solely to provide details of an experimental API.

Copyright Notice

Table of Contents

1. Introduction 2
  1.1. Scope 2
  1.2. A Context 2
  1.3. Terms 3
2. Reservations 4
3. Using the API 6
  3.1. Initialization 6
  3.2. Describing a Reservation Request 6
  3.3. Passing Authorization Information 8
  3.4. Creating a Reservation 9
  3.5. Modifying a Reservation 9
  3.6. Querying a Reservation 10
  3.7. Reservation Attributes 10
  3.8. Committing a Reservation 11
  3.9. Binding a Reservation 11
  3.10. Using Callbacks 12
  3.11. Canceling a Reservation 13
  3.12. Deactivating the Advance Reservation Module 14
4. Grid Advance Reservation API Reference 14
  4.1. Constants 14
  4.2. Data Structures 18
  4.3. Functions 19
5. Security Considerations 25
6. Author Information 25
7. Copyright Notice 25
8. Intellectual Property Notice 26

1. Introduction

The Grid scheduling architecture should provide programmers with convenient access to end-to-end Quality of Service (QoS) for programs. To do so, mechanisms are required for making advance QoS reservations for different types of resources, including computers, networks, and disks. A reservation is a promise from the system that an application will receive a certain level of service from a resource. For example, a reservation may promise a certain bandwidth on a network or a certain percentage of a CPU.

A Grid resource reservation API should provide two capabilities. First, it should allow users to make reservations either in advance of when the resource is needed or at the time that the user needs it. Second, the same API should be capable of making and manipulating a reservation regardless of the type of the underlying resource, thereby simplifying the programming when an application must work with multiple kinds of resources and multiple simultaneous reservations.

1.1. Scope

This document presents an advance reservation API. It does not present any mechanisms or interfaces for querying the status of previously made reservations, in order to assist users in discovering good times to make reservations. It is expected that another document will describe such mechanisms and interfaces.

This document uses the Resource Specification Language (RSL) to describe a reservation request. While we believe that RSL is a good interim solution, we expect that a longer-term solution will be developed separately in cooperation with the Information Services working group.

Advance reservations for computer resources are just beginning to be used. We expect that as advance reservations become more widely used, our understanding will deepen and grow. During this time, it will be useful for different advance reservation systems to provide the same interface, in order to enable broad experimentation with such systems. This document provides such an interface, but we expect that as our knowledge grows, we will desire an updated interface. We expect this interface to serve us well for two to five years, at which point a new interface will be developed to reflect new understanding.

1.2. A Context

The proposed Grid Advance Reservation API can be considered a remote procedure call mechanism to communication with a reservation manager. A reservation manager controls reservations for a resource: it performs admission control and controls the resource to enforce the reservations. Some resources already can work with advance reservations, so the reservation manager is a simple program. Most resources cannot deal with advance reservations, however, so the reservation manager tracks the reservations and does admission control for new reservation requests.

To create a flexible architecture that supports co-allocation, resource location, and resource acquisition steps, we propose a layered architecture with three levels of APIs and one level of low-level mechanisms.

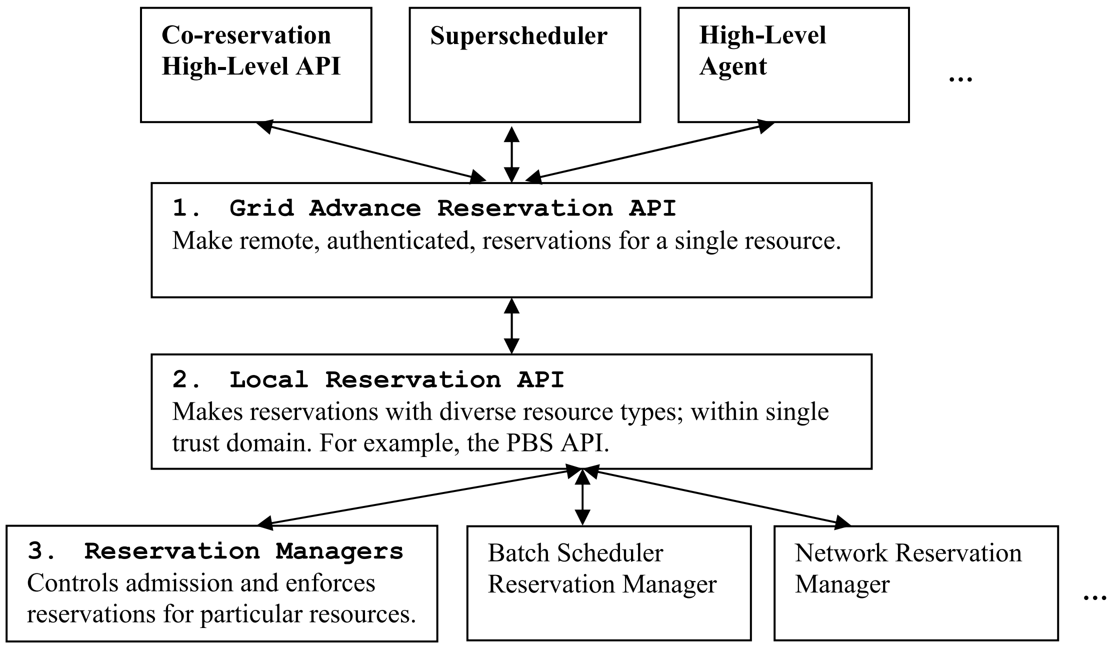
Co-reservation High-Level API
Superscheduler
High-Level Agent
...
1. Grid Advance Reservation API
Make remote, authenticated, reservations for a single resource.
2. Local Reservation API
Makes reservations with diverse resource types; within single trust domain. For example, the PBS API.
3. Reservation Managers
Controls admission and enforces reservations for particular resources.
Batch Scheduler Reservation Manager
Network Reservation Manager
...

Figure 1 – layered advance reservation architecture

The remaining document describes the intermediate API in Figure 1, the Grid Advance Reservation API. We note that there is no requirement that the Grid Advance Reservation API be implemented by using a local reservation API; this is just a conceptual guide to aid in understanding where this API fits in a larger context. This API is immediately useful both to programmers wishing to make advance reservation and to those building higher-level tools, such as superschedulers, co-reservation agents, or other as-yet-unforeseen tools.

We envision that when users desire advance reservations for multiple resources and different kinds of resources at the same time, they will build tools on top of the Grid Advance Reservation API. For example, users may wish to reserve time on two computers, disk space on those two computers for storage of intermediate results, and network bandwidth between the two computers for communication. This API will be useful for these sorts of *co-reservations*.

1.3. Terms

Within the document several terms are used to precisely describe the semantics of the API calls. To clarify the use of these terms and to avoid misunderstandings, we briefly explain these terms:

Reservation: This is a promise from the system that an application will receive a certain level of service from a resource.

Creating a Reservation: This process involves asking for a specific resource behavior of given duration within a specified time interval.

Committing a Reservation: To facilitate the use of co-reservations, implementers might choose to use a two-phase commit protocol. Once, a reservation is created, it is on hold for a specific time. If the reservation is not committed within this time interval, it is cancelled.
Binding a Reservation: Some reservations cannot be instantiated without the provision of additional run-time information. The binding step is supposed to facilitate the creation of reservations without requiring a detailed knowledge about all required attributes. To actually use the reservation, however, the user must "bind" the reservation, that is, provide the missing attributes.

2. Reservations

Reservations have five important attributes:

Start Time: The earliest time that the reservation may begin. A reservation always has a start time, even if it is an immediate reservation, which begins as soon as the reservation is made. The start time is in seconds from 00:00:00 UTC, January 1, 1970. For example, to make an immediate reservation, the user can call the Unix time() function to discover the current time and use that as the start time.

Duration: The amount of time the reservation lasts, in seconds. All reservations must specify their duration, so that the underlying reservation managers can do appropriate admission control for reservations granted in advance.

Resource Type: The type of underlying resource, such as a network, a computer, or a disk.

Reservation Type: A particular kind of reservation.

Resource-Specific Parameters: Parameters that are unique to each type of resource, such as bandwidth for a network reservation and number of nodes for a computation reservation.

Optionally, a reservation may specify the attribute *End Time* in the same way that *Start Time* is specified. If the difference between *End Time* and *Start Time* exceeds the value of *Duration*, any given time interval of the correct duration starting at or after *Start Time* and not ending past *End Time* is accepted for the reservation. The exact start time is made by the reservation manager.

When a reservation is requested, one must specify these attributes, as shown below in the API description. If the reservation request is accepted, a *reservation handle* is returned from the system. This is an opaque string that uniquely identifies the reservation. All future operations require this handle. (The internal format of this string will be described in a different document. Generally, programmers and users do not need to be aware the format of this reservation handle. It may vary from reservation manager to reservation manager, and it does not need to be interpreted by anything other than the reservation manager for correct operation. The reservation handle should, however, encode how the reservation manager for that reservation can be contacted, because only the create reservation function actually allows the user to specify the reservation manager.

Once the application has received a reservation handle, it can perform several operations with that handle:

- *Modify Reservation:* One can request a modification to an existing reservation. For instance, one can increase the bandwidth that has already been requested. A modification that reduces its requirements normally succeeds, although certain factors may cause reduction modifications to fail, such as local policy that does not allow small reservations on some resources. In no case SHOULD the underlying implementation implement a modification such that if the modification fails, the original reservation is lost. For example, a simple implementation may implement modification by canceling a reservation and making a new reservation, but if the new reservation fails and it cannot be rolled back to the original reservation, this would be unacceptable.

- *Cancel Reservation:* One can inform the reservation manager that the reservation is no longer needed (i.e., canceled).

- *Bind Reservation:* When the application is ready to use a reservation, it may need to provide run-time information that was not available at the time the reservation was made. This is known as *binding* a reservation. For example, network reservations require port numbers to be specified, but they are not usually known at reservation time. Not all reservations require such run-time parameters.

- *Unbind Reservation:* A reservation can be unbound. It then will no longer be usable by the person using the reservation. It can be rebound, however, with new run-time parameters.

- *Commit Reservation:* When a reservation is created, it can be specified as a two-phase commit reservation. Such reservations time-out after a specified time-period, unless the reservation is committed.

- *Query Reservation Status:* One can discover the status of a reservation by polling it. The status includes whether the start of the reservation has begun and whether the reservation has been committed.

- *Query Reservation Attributes:* One can discover attributes associated with an existing reservation. These include begin and end time of the given reservation and whether it is a two-phase commit reservation. The attributes also include specific information required to actually use a reservation. Example attributes are a directory name where data was staged on, or a queue name which has to be used for submitting a job.

- *Register Callback:* One can provide a function that will be called when the status of a reservation changes or when the reservation manager wishes to provide extra information to the application. This information may include notification that the related reservation appears to be too small. Callbacks implement the monitor functionality described in S-RFC 3.

3. Using the API

The Advance Reservation API should be provided as a library that can be linked by programs written in C. Additional bindings (e.g., from a Java implementation) are not described here. We note that although this API is derived from the GARA API and has some references to Globus, an implementation of the API is not required to include any such references to Globus.

3.1. Initialization

Before one can use the API, an initialization of the module is required.

```
grid_reservation_activate();
```

Programs are allowed to call this activate function more than once. Invocations after the first have no effect. The advance reservation module does, however, keep track of how many times it has been activated; and the deactivation function (see Section 3.10) MUST be called that same number of times before it actually deactivates the Grid reservation module.

The idea of this behavior is to simplify application development if several portions of a program (particularly libraries) wish to independently use the Grid reservation API. Each portion of the program can independently invoke the Grid reservation API without worrying about when the module will be activated or deactivated.

3.2. Describing a Reservation Request

Until the Grid Information Group finishes its work in progress and defines a standard resource description language such as MDSML, reservation attributes or allocation properties will be described by using the Resource Specification Language (RSL) of the Globus Project. We note that since the API treats the reservation attribute specification as a string, we should not have to change the API once we describe a new specification method. Moreover, RSL offers the advantage that it can handle Boolean operations and comparison operations, although this use is not demonstrated here.

An RSL string is simply a list of attribute-value pairs that looks like the following.

```
&(attribute-1=value-1) (attribute-2=value-2) … (attribute-N=value-N)
```

An example RSL string for requesting a network reservation for 150Kbps between a source IP address of 140.221.48.146 and a destination address of 140.221.48.106 looks like the following.

```
&(resource-type=network)
 (start-time=953158862)
 (duration=3600)
 (endpoint-a=140.221.48.146)
 (endpoint-b=140.221.48.106)
 (bandwidth=150)
```

Note that this string was spaced out on several lines for readability, while RSL strings do not have newlines in them. More information about RSL is available from the Globus Project Web page: http://www.globus.org.

Following is a list of attributes that may be used to specify a reservation. The universal attributes are for all types of reservations, while the other attributes are for specific types of resources. Note that the compute resource attributes are mutually exclusive. This minimal set of supported attributes might be extended in future versions of the document.

Attribute	**Units**	**Default**	**Req?**	**Description**
Universal Attributes				
resource-type			Y	Allowable values: “network”, “compute”, “disk”, or “graphic-pipeline”.
reservation-type				Currently valid only for network reservations. If it is not specified, it is a foreground reservation. Otherwise it is one of “background” or “low-latency
start-time	secs		Y	Earliest acceptable time the reservation starts in seconds since 00:00:00 UTC, January 1, 1970. If one specifies “now”, then the reservation will begin immediately.
end-time	secs			Latest acceptable time the reservation might be active; UTC format as above.
duration	secs	100		Length of the reservation, in seconds.
two-phase	n/a	false		If “true”, then the reservation will time-out after some time, unless it is committed. See discussion in text.
two-phase-timeout	secs	implementation-dependent		The time for the timeout when the two-phase commit option is specified.
Compute Resource Attributes				
number-of-cpus	Int	1		Number of CPUs to be reserved.
percent-cpu	%	20		Percentage of the CPU’s time given to the reserved process.
Network Resource Attributes				
endpoint-a			Y	The machine at one end of the network flow. This may be specified as a dotted IP address, such as 140.221.48.162, or a machine name, such as dslnet2.mcs.anl.gov
endpoint-b			Y	The machine at the other end of the network flow. This may be specified as a dotted IP address, such as 140.221.48.162, or a machine name, such as dslnet2.mcs.anl.gov
bandwidth	Kbps	8		How fast a flow can transfer data.
directionality		bidirectional		unidirectional-ab: reservation for traffic from a to b. unidirectional-ba: reservation for traffic from b to a. bidirectional: reservation for traffic in both directions.

Disk Resource Attributes				
size	KB			The storage space needed for a single file or set of files.
bandwidth	Kbps	8		Speed for reading/writing file.

3.3. Passing Authorization Information

Before any of the following operations can be done, an important data structure must be initialized. The intention of the data structure is to pass mandatory information such as user credentials or authorization attributes. For example, some reservations may be granted only when specific authorization attributes are provided. The final implementation of the data structure will depend on the input from the Grid Security working group. For now, we assume that user credentials are accessed by denoting the file name of the proxy file that contains a valid pair of certificate and private key. If no name is supplied (NULL), it is assumed that the user identity can be extracted from the environment. Some toolkits, such as the Globus Toolkit, support this extraction of the user identity from the environment, and it simplifies many programming tasks. A superscheduler, on the other hand, is likely to wish to supply the identity, since it will be handling requests for many different users.

Enabling the specification of authorization and policy information in the API is important for implementing high-level functionalities such as a superscheduler. An example of policy information set by such an entity is the distinguished name of the owner of a reservation, a list of group memberships, and a list of user or groups allowed to use the reservation in subsequent calls. To ensure the integrity of the policy field, it must be signed by a trusted entity.

Data Structure:

```
      struct authorization_info_s
      {
         char                *user_proxy_file;
         int                  number_of_policies;
         policy_sequence_t   *policies;
      }
      typedef      struct authorization_info_s
                   authorization_info_t;


      struct policy_entry_s
      {
         char *attribute;
         char *value;
      }
      typedef struct policy_entry_s policy_entry_t;

      struct policy_sequence_s
      {
         policy_entry_t     policy_list[];
         char               dn_of_signing_entity
         unsigned char      *signature;
      }
      typedef struct policy_sequence_s policy_sequence_t;
```

3.4. Creating a Reservation

Before a reservation can be created, its needs must be specified as described above. To request a reservation, the application must call grid_reservation_create().

```
      int  error;
      char *request_rsl  = "&(resource-type=compute)
                             (number-of-nodes=14)";
      char *time_rsl     = "&(start-time=953158862) (duration=3600)"
      char *resource_manager_contact = "pitcairn.mcs.anl.gov:2119:/O=Grid/"
                                        "O=Globus/CN=pitcairn.mcs.anl.gov"
      char *reservation_handle;
      authorization_info_t  auth_info;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;
      error = grid_reservation_create( reservation_manager_contact,
                                      &auth_info,
                                       request_rsl, time_rsl,
                                      &reservation_handle);
```

The reservation manager contact is a string obtained from another location, such as the Grid Information Service. It specifies the URL of the related reservation manager.

Note that if the reservation had specified "(two-phase=true)" in the request_rsl, the reservation would have needed to be committed. See below.

3.5. Modifying a Reservation

Modifying a reservation is similar to creating a reservation, except that instead of providing a reservation manager contact, the application provides the handle to the reservation that was created earlier.

```
      int  error;
      char *request_rsl = "&(resource-type=compute)
                             (number-of-nodes=128)";
      char *time_rsl    = "(start-time=953158862) (end-time=953173262)
                            (duration=7200)"

      char *old_reservation_handle;
      char *reservation_handle;
      authorization_info_t  auth_info;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;
      error = grid_reservation_modify(&auth_info, old_reservation_handle,
                                      request_rsl, time_rsl,
                                     &reservation_handle);
```

3.6. Querying a Reservation

To find out the status of a reservation, one can issue the following query.

```
      int error;
      int status;
      char *reservation_handle;
      authorization_info_t  auth_info;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;


      error = grid_reservation_status(&auth_info, reservation_handle,
                                      &status);
```

If there is no error, the status will be one of the following.

```
      GRID_RESERVATION_STATUS_NOT_STARTED
      GRID_RESERVATION_STATUS_NOT_STARTED_BOUND
      GRID_RESERVATION_STATUS_READY_NOT_BOUND
      GRID_RESERVATION_STATUS_ACTIVE
      GRID_RESERVATION_STATUS_FINISHED
```

A reservation is bound if a previous call to grid_reservation_bind succeeded. A reservation is ready if the current time is later than the start time, and the duration has not yet elapsed. A reservation is active if it is both ready and bound. A reservation is finished if the current time is later than the start time plus the duration.

3.7. Reservation Attributes

Reservation attributes are a generalized version of the status call that allow more attributes to be queried. The following is an example of querying the time that a reservation begins.

```
      int                             error;
      char                           *reservation_handle;
      authorization_info_t            auth_info;
      grid_reservation_attribute_t  attribute;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;

      error = grid_reservation_attribute(&auth_info, reservation_handle,
                  GRID_RESERVATION_ATTRIBUTE_BEGIN_TIME, &attribute);
      printf("Start time is %s\n", ctime(attribute.value.time));
```

3.8. Committing a Reservation

When a reservation is a two-phase commit reservation (as specified in the reservation request, see Sections 3.2 and 3.4), it must be committed before the reservation times out. The time-out period can be specified when the reservation is made, but it defaults to an implementation-specified time that can be discovered through some external query mechanism.

```
int   error;
char  *reservation_handle;
authorization_info_t  auth_info;

auth_info.user_proxy_file    = NULL;
auth_info.number_of_policies = 0;
auth_info.policies           = NULL;


error = grid_reservation_commit(&auth_info, reservation_handle);
```

3.9. Binding a Reservation

When the application is ready to use a reservation, it may need to *bind* the reservation. Binding a reservation is a required step if the creation was done without providing all required attributes to instantiate it, and this is almost always the case. The following example binds a process-id to a given reservation.

```
int   error;
char  *bind_paramters = “&(process-id=5631)”;
char  *reservation_handle;
authorization_info_t  auth_info;

auth_info.user_proxy_file    = NULL;
auth_info.number_of_policies = 0;
auth_info.policies           = NULL;


error = grid_reservation_bind(&auth_info, reservation_handle,
            &bind_parameters);
```

Note that the run-time parameters are specified as an RSL string. This allows the integration of different reservation managers within one API. Bind parameters are resource dependent. For compute reservations, for instance, the only parameter to be specified might be the process-id, which specifies the process ID of the process that will be receiving the reservation. For some reservations, such as ones requesting a number of nodes on a machine, there may be no bind parameters, but the call still must be made. For network reservations, there are more parameters:

- which-endpoint: If the reservation is being bound from a machine involved in the reservation, this specifies which machine it is. The machine is either “a” or “b”, and it matches what was specified in the reservation request. If a different machine is binding the reservation on behalf of the processes involved, “a” is used.

- endpoint-a-port: This is the port used by endpoint-a, as specified in the reservation request. The implementation assumes that data is being sent from endpoint-a to endpoint-b; this will be the port used by the sender.
- endpoint-b-port: This is the port use by endpoint-b, as specified in the reservation request. The implementation assumes that data is being sent from endpoint-a to endpoint-b; this will be the port used by the receiver.

A reservation is not considered active until it is bound. Once a reservation has both begun and been bound, the reservation manager must do whatever setup is necessary in order to ensure that the reservation is granted. If the reservation was bound before it began, the reservation manager will automatically enable the reservation once it begins.

If the application is temporarily not using a reservation but will resume using it before the reservation has expired, the application can unbind the reservation.

```
int   error;
char  *reservation_handle;
authorization_info_t  auth_info;

auth_info.user_proxy_file    = NULL;
auth_info.number_of_policies = 0;
auth_info.policies           = NULL;


error = grid_reservation_unbind(&auth_info, reservation_handle);
```

Once an application unbind a reservation, it may bind the reservation again.

3.10. Using Callbacks

Callbacks are used to communicate monitoring functions to the user. If the application would like to be informed whenever the status of a reservation changes (see Section 3.6, *Querying a Reservation*), it can use a callback function. When the user registers a callback function, it will immediately be called once, to provide the current status, and will be called every time the status changes thereafters.

First, the application needs to create a callback function.

```
static void callback_handler(
    char                         *reservation_handle,
    grid_reservation_event_t      event,
    void                          *user_parameter)
{
  /* Place code here to examine the event */
  /* If it is a status event, event.event_type will be
     STATUS_EVENT, and the status will be in
     event.event. */
  if (event.event == STATUS_EVENT)
  {
    if (event.event_type == RESERVATION_STATUS_FINISHED)
    {
      /* React to reservation being finished */
```

```
          }
        }
        return;
      }
```

Then the application must register this function for each reservation that needs to be monitored.

```
      static void callback_handler(char *reservation_handle,
          grid_reservation_event_t event,
          void *user_parameter);

      int  error;
      char *reservation_handle;
      authorization_info_t  auth_info;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;


      error = grid_reservation_callback_register(&auth_info,
                   reservation_handle,
                   callback_handler, NULL);
```

Note that the last parameter passed to the registration function will be forwarded as the user_parameter to the callback function.

If the user no longer wants to have a function called when the status changes, it can be unregistered.

```
      int  error;
      char *reservation_handle;
      authorization_info_t  auth_info;

      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;


      error = grid_reservation_callback_remove(&auth_info,
                        reservation_handle, callback_handler);
```

Note that one can register multiple callback functions for a single reservation handle.

3.11. Canceling a Reservation

When the application has finished using a reservation, it should cancel the reservation by using the reservation handle that was obtained when the reservation was created.

```
      char *reservation_handle;
      authorization_info_t  auth_info;
      auth_info.user_proxy_file    = NULL;
      auth_info.number_of_policies = 0;
      auth_info.policies           = NULL;

      grid_reservation_cancel(&auth_info, reservation_handle);
```

When the application cancels a reservation, all of the callbacks that have been registered for that reservation will automatically be cancelled.

It is important to note that all reservation managers should clean up reservations automatically, once they are expired. However, the cancel call must not fail if the reservation manager has removed a reservation already.

3.12. Deactivating the Advance Reservation Module

When the application has finished using API, it should deactivate the API, to enable cleanup.

```
      grid_reservation_deactivate();
```

4. Grid Advance Reservation API Reference

We discuss in this section the constants, data structures, and functions that the API uses

4.1. Constants

The API uses various constants, including errors, callbacks and status constants, attribute types, and variable types.

Errors

Currently we define N errors, with the specific values as shown here. We reserve the right to define errors with the values 0-256. Implementations may use any other errors values they like, but until they become standardized, they must not use the values 0-256.

GRID_ERROR_NONE (0)
    No error has occurred.
GRID_ERROR_UNKNOWN (1)
    An error has occurred, but the reservation manager just doesn't know what it is.
GRID_ERROR_MODULE_NOT_ACTIVE (2)
    The user has tried to use the API without activating the module first.
GRID_ERROR_BAD_PARAMETER (3)
    A bad parameter, such as a NULL reservation handle, has been passed to an API function that actually expected a good parameter.
GRID_ERROR_ZERO_LENGTH_RESOURCE_SPECIFICATION (4)
    An resource description was provided, but it is empty. It may be that this is never returned.
GRID_ERROR_BAD_RESOURCE_DESCRIPTION (5)

There is an error, probably a syntax error, in the resource description string.

GRID_ERROR_BAD_RESERVATION_HANDLE (6)
The reservation handle that was provided isn’t really a reservation handle.

GRID_ERROR_BAD_USER_CREDENTIALS (7)
The credentials specified in the authorization_info_t structure are no good. Who are you anyway?

GRID_ERROR_NO_USER_CREDENTIALS (8)
The credentials specified in the authorization_info_t were NULL and they could not be extracted from the environment, leading the reservation manager to suspect you are a non-person.

GRID_ERROR_BAD_POLICY_DESCRIPTION (9)
The policy specification was not accepted.

GRID_ERROR_CONNECTION_FAILED (10)
The API was unable to connect to the reservation manager.

GRID_ERROR_AUTHORIZATION (11)
The API was unable to authenticate and authorize the user.

GRID_ERROR_VERSION_MISMATCH (12)
Protocol error with the reservation manager because of mismatch of version.

GRID_ERROR_INVALID_REQUEST (13)
The request cannot be handled by the reservation manager.

GRID_ERROR_UNKNOWN_RESERVATION_TYPE (14)
The reservation type in the reservation request must be one of “network”, “compute”, or “disk”, but it wasn’t.

GRID_ERROR_PROTOCOL_FAILED (15)
There was a problem communicating with the reservation manager.

GRID_ERROR_MISSING_RESERVATION_TYPE (16)
The reservation type in the RSL reservation request wasn’t provided.

GRID_ERROR_OUT_OF_MEMORY (17)
A request for memory failed.

GRID_ERROR_MISSING_ENDPOINT_A (18)
A network reservation request didn’t specify endpoint-a.

GRID_ERROR_MISSING_ENDPOINT_B (19)
A network reservation request didn’t specify endpoint-b.

GRID_ERROR_CANT_MAKE_RESERVATION (20)
The reservation can’t be made. Probably there are other reservations already at the same time, and there isn’t room for the new reservation.

GRID_ERROR_BAD_RESERVATION_OBJECT (21)
This error probably means that the user tried to make a network reservation for an endpoint that the reservation manager hasn’t been configured to allow reservations for.

GRID_ERROR_SERVICE_EXECUTABLE_NOT_FOUND (22)
The reservation manager is misconfigured.

GRID_ERROR_CANT_CONTACT_RESERVATION_MANAGER (23)
The reservation manager is unavailable. Check to make sure that it’s running or that the correct resource location was specified.

GRID_ERROR_UNKNOWN_GRAM_ERROR (24)
Some error in the underlying protocol has failed.

GRID_ERROR_MISSING_RESERVATION_TYPE (25)
The request indicates that a subtype has to be specified, but it wasn´t.

GRID_ERROR_ATTRIBUTE_UNAVAILABLE_FOR_RESERVATION_TYPE (26)
An attribute was requested for a type of reservation that can never have that attribute. For example, a queue name was requested for a network reservation.

GRID_ERROR_ATTRIBUTE_UNAVAILABLE_IN_IMPLEMENTATION (27)
An attribute that might be reasonable to request is not available, because the implementation doesn't support it.

GRID_ERROR_ATTRIBUTE_ONLY_AFTER_BIND (28)
The reservation has not yet been bound, but the attribute can only be provided after the reservation has been bound.

GRID_ERROR_ATTRIBUTE_ONLY_AFTER_COMMIT (29)
The reservation has not yet been committed, but the attribute can only be provided after the reservation has been committed.

GRID_ERROR_COMMIT_NOT_SUPPORTED (30)
The user called grid_reservation_commit, but this call is not supported for this API-implementation or this resource type.

GRID_ERROR_CALLBACKS_NOT_SUPPORTED (31)
The user tried to register a callback, but this is not supported by this API-implementation.

GRID_ERROR_INTERNAL_ERROR (256)
The request indicates that some other error occurred. (We need a good way to deal with these other errors in a regular way. Different errors can occur for different implementations, so how do we report them?)

Callback and Status Constants

The following events are reported to callbacks, with the specific values as shown here. We reserve the right to define constants with the values 0-256. Implementations may use any other constant values they like, but until they become standardized, they must not use the values 0-256.

GRID_RESERVATION_STATUS_EVENT (0)
The status of the reservation has changed. See the lists of status constants below.

GRID_RESERVATION_CHANGE_EVENT (1)
The reservation has been preempted, or the reservation quantity (like bandwidth) has changed. See the list changes below.

GRID_RESERVATION_MONITOR_EVENT (2)
The user is informed about specific monitor events such as that he is exceeding its reservation.

The following statuses can be reported to callbacks on a status event or in response to a user calling reservation_status.

GRID_RESERVATION_STATUS_NOT_STARTED (0)
The reservation has not yet begun (the current time is before the start time).

GRID_RESERVATION_STATUS_NOT_STARTED_BOUND (1)
Although the reservation has not yet begun, the reservation has been bound.

GRID_RESERVATION_STATUS_READY_NOT_BOUND (2)
The reservation has begun (the current time is after the start time) but can't yet be used because it has not been bound yet.

GRID_RESERVATION_STATUS_ACTIVE (3)
The reservation has begun and been bound.

GRID_RESERVATION_STATUS_FINISHED (4)
The reservation is over. That is, the current time is greater than the start time plus the duration of the reservation.

The following changes can be reported on a CHANGE_EVENT:

GRID_RESERVATION_RESERVATION_CHANGE_PREEMPTED (5)
: The reservation has been preempted because a more important reservation has occurred. Currently, this will not be reported, because preemption has not yet been implemented.

GRID_RESERVATION_CHANGE_QUANTITY (6)
: The quantity (like bandwidth) has been changed. This occurs for bulk transfer network reservations or for reservations that are adaptable in response to changing conditions.

Attribute Types

The following types of attributes can be requested in the grid_reservation_attribute function with the which_attribute parameter. Currently we define 12 attributes, with the specific values as shown here. We reserve the right to use attributes with the values 0-256. Implementations may use any other attribute values they like, but until they become standardized, they must not use the values 0-256.

GRID_RESERVATION_ATTRIBUTE_BEGIN_TIME (0)
: The time a reservation begins

GRID_RESERVATION_ATTRIBUTE_END_TIME (1)
: The time a reservation ends

GRID_RESERVATION_ATTRIBUTE_HAS_BEGUN (2)
: A Boolean indicating whether a reservation has begun and has not yet ended.

GRID_RESERVATION_ATTRIBUTE_IS_BOUND (3)
: A Boolean indicating whether a reservation has been bound and has not yet ended.

GRID_RESERVATION_ATTRIBUTE_IS_COMMITTED (4)
: A Boolean indicating whether a reservation has been committed and has not yet ended.

GRID_RESERVATION_ATTRIBUTE_NEEDS_BIND (5)
: A Boolean indicating whether a reservation needs to be bound.

GRID_RESERVATION_ATTRIBUTE_NEEDS_COMMIT (6)
: A Boolean indicating whether a reservation needs to be committed.

GRID_RESERVATION_ATTRIBUTE_AUTHORIZED (7)
: A Boolean indicating whether the identity/authorization given in the attribute request function call is sufficient to operate on the reservation. (In case different users want to work with a reservation.)

GRID_RESERVATION_ATTRIBUTE_QUEUE_NAME (8)
: A string indicating the queue a job should be submitted to, in order for the reservation to work.

GRID_RESERVATION_ATTRIBUTE_PATH_NAME (9)
: A string indicating where files should be written, in order for the disk reservation to work.

GRID_RESERVATION_ATTRIBUTE_MAX_NETWORK_DELAY_ESTIMATE (10)
: A floating-point number indicating the best guess as to the delay packets using a network reservation will experience

GRID_RESERVATION_ATTRIBUTE_LIKELIHOOD_OF_FULFILMENT (11)
: A number 0-100 indicating the system's best guess of the percent chance that a reservation will actually be given to the user. Preemptions, downtimes, and nasty system administrators may affect this percentage, and there is no guarantee that a system can accurately provide this estimate.

Variable Types

The following types are used to indicate what variable type is returned for an attribute.

GRID_RESERVATION_VARIABLE_INT
    The variable is of type “int” for an integer.
GRID_RESERVATION_VARIABLE_BOOLEAN
    The variable is of type “int” for an boolean (TRUE or FALSE).
GRID_RESERVATION_VARIABLE_FLOAT
    The variable is of type “float” for an floating point number.
GRID_RESERVATION_VARIABLE_STRING
    The variable is of type “char *” for an string.
GRID_RESERVATION_VARIABLE_TIME
    The variable is of type “time_t” for a specific time.

Miscellaneous Constants

GRID_RESERVATION_API_VERSION (1)
    The version of the library that must be compiled against or that one is running against. Currently this is defined to be 1.

4.2. Data Structures

This section describes the data structures used by the API.

The Event Data Structure

```
typedef struct
{
  int     event_type;
  int     event;
  double  quantity;
} grid_reservation_event_t;
```

This structure is provided to callback functions. The event type and event are constants from the list above. The quantity is provided when the event is a change event indicating that the quantity has changed.

The Authorization information Structure

```
typedef struct
{
  char               *user_proxy_file;
  int                 number_of_policies;
  policy_sequence_t  *policies;
} authorization_info_t;
```

This structure is provided to callback functions. The event type and event are constants from the list above. The quantity is provided when the event is a change event indicating that the quantity has changed.

The Attribute Structure

```
typedef int reservation_attribute_type_t;

typedef struct
{
  reservation_attribute_type_t  which_attribute;
  variable_type_t               variable_type;
  union
  {
    int     boolean; /* TRUE or FALSE */
    int     integer;
    float   number;
    char    *text;
    time_t  time;
  };
} grid_reservation_attribute_t;
```

This structure is returned to describe an attribute that is requested by grid_reservation_attribute().

Callback functions

```
typedef void (*grid_reservation_callback_t)(
    char                              *reservation_handle,
    grid_reservation_event_t          event,
    void                              *user_parameter);
```

This is the type of function that must be used for callback functions. It is the user's responsibility to implement such a function. When a user registers a callback function of this type, its reference is used to actually call this routine whenever a callback event occurs.

4.3. Functions

Note that all of the functions of the API return an integer. This integer is the error code, if any error occurred. See the list of errors in Section 4.1, *Constants*.

grid_reservation_activate

```
int grid_reservation_activate(void);
```

This function initializes the Grid Advance Reservation Module. This MUST be called before any other function in the module. Programs are allowed to call this activate function more than once; invocations after the first have no effect. The advance reservation module does, however, keep track of how many times it has been activated; and the deactivation function (see Section 3.10) MUST be called that same number of times before it actually deactivates the Grid reservation module.

grid_reservation_deactivate

```
int grid_reservation_deactivate(void);
```

This function informs the Grid Advance Reservation Module that it is no longer needed, so that it can perform any cleanup that it might need to do.

grid_reservation_create

```
int grid_reservation_create(
         const char                    *manager_contact,
         const authorization_info_t  *auth_info,
         const char                    *reservation_specification, /* RSL */
         const char                    *time_specification,        /* RSL */
         char                          **reservation_handle);
```

This function attempts to make a reservation.

In:

manager_contact: The contact string for access to the reservation manager for the resource the user wishes to make a reservation with.

auth_info: Data structure providing access to the user's credentials and additional policy information that might be used for approving authorization.

reservation_specification: An RSL string describing the attributes the user wishes to have for the reservation. See Section 3.2, *Describing a Reservation Request*.

time_specification: An RSL string describing the time interval the reservation should be in place. See Section 3.2, *Describing a Reservation Request*.

In/Out:

reservation_handle: If the reservation was successfully made, a pointer to the reservation handle will be provided in the reservation handle member of the advance_reservation_handle_s. The memory for this reservation handle is allocated by malloc(), and it is the user's responsibility to free the memory with free() when done. Note that the reservation_handle is also specified as input parameter. This is because some implementations might require that the user present a valid reservation handle for another resource such as a CPU reservation is prerequisite whenever a disk reservation is made.

grid_reservation_modify

```
int grid_reservation_modify(
         const authorization_info_t  *auth_info,
         const char                  *old_reservation_handle,
         const char                  *reservation_specification, /* RSL */
         const char                  *time_specification,       /* RSL */
         char                        **reservation_handle);
```

This function attempts to modify a new reservation. Note that if the reservation is changed, the user might receive a new reservation handle.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information which might be used for approving authorization.

old_reservation_handle: The handle of an existing reservation that the user wishes to modify. The memory for this reservation handle MUST be deallocated by the user.

reservation_specification: An RSL string describing the new attributes you wish to have for your reservation. See *Describing a Reservation Request* above.

time_specification: An RSL string describing the new time interval the reservation should be in place. See *Describing a Reservation Request* above.

Out:

reservation_handle: If the reservation was successfully modified, a pointer to a new reservation handle will be provided. The memory for the reservation handle is allocated by malloc(), and it is the user's responsibility to free the memory with free().

grid_reservation_commit

```
int grid_reservation_commit(
         const authorization_info_t  *auth_info,
         const char                  *reservation_handle);
```

This commits a two-phase commit reservation. It is not a replacement for binding run-time parameters.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to bind.

grid_reservation_bind

```
int grid_reservation_bind(
        const authorization_info_t  *auth_info,
        const char                  *reservation_handle,
        const char                  *bind_parameters);
```

This binds a reservation by providing run-time parameters.

In:

auth_info: Data structure providing access to the user’s credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to bind.

bind_parameters: An RSL string describing the new attributes the user wishes to have for the reservation. See Section 3.9, *Binding a Reservation* above.

grid_reservation_unbind

```
int grid_reservation_unbind(
        const authorization_info_t  *auth_info,
        const char                  *reservation_handle);
```

This undoes the “bind” for a reservation that has been bound. The reservation is still valid and can be used again by calling reservation_bind() again.

In:

auth_info: Data structure providing access to the user’s credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to bind.

grid_reservation_status

```
int grid_reservation_status(
        const authorization_info_t  *auth_info,
        const char                  *reservation_handle,
         int                         *status;
```

This function queries for a reservation’s status.

In:

auth_info: Data structure providing access to the user’s credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to query.

Out:

status: The status of the reservation. It is one of the constants described in Section 4.1, *Callback and Status Constants.*

grid_reservation_attribute

```
int grid_reservation_attribute(
         const authorization_info_t   *auth_info,
         const char                   *reservation_handle,
         reservation_attribute_type_t  which_attribute,
         grid_reservation_attribute_t *attribute);
```

This function returns an attribute for a reservation.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information that might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to query.

which_attribute: The desired attribute, described in Section 4.1, *Attribute Types*, above.

Out:

attribute: Information about the attribute. The data structure is described above in Section 4.2, *The Attribute Structure*. Note that the programmer can retrieve the correct attribute value by examining the variable_type field and checking the defined variable types above.

Common Errors:

```
GRID_ERROR_ATTRIBUTE_UNAVAILABLE_FOR_RESERVATION_TYPE
GRID_ERROR_ATTRIBUTE_UNAVAILABLE_IN_IMPLEMENTATION
GRID_ERROR_ATTRIBUTE_ONLY_AFTER_BIND
GRID_ERROR_ATTRIBUTE_ONLY_AFTER_COMMIT
```

grid_reservation_callback_register

```
int grid_reservation_callback_register(
         const authorization_info_t      *auth_info,
         const char                      *reservation_handle,
         reservation_callback_t          callback_function,
         void                            *user_parameter);
```

After this function successfully completes, the specified callback function will be called whenever the status of a reservation changes. It will also be immediately called once to provide the current status of the reservation. Note that multiple callbacks can be registered for a single reservation.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation for which the user wishes to receive callbacks.

callback_function: The function that will be called by GARA when the status of a reservation changes.

user_parameter: The value provided here will be passed to the callback function unmodified.

grid_reservation_callback_remove

```
int reservation_callback_remove(
         const authorization_info_t      *auth_info,
         const char                      *reservation_handle,
         reservation_callback_t          callback_function);
```

After this function successfully completes, the specified callback function will no longer be called when the status of the reservation changes.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation for which the user wishes to receive callbacks.

callback_function: The function that will be called by GARA when the status of a reservation changes.

user_parameter: The value provided here will be passed to the callback function unmodified.

grid_reservation_cancel

```
int grid_reservation_cancel(
         const authorization_info_t   *auth_info,
         const char                   *reservation_handle);
```

This cancels a reservation. When a reservation is cancelled, the reservation handle (and copies of it) may not be used anymore. For example, if the user tries to bind the cancelled reservation, it will fail.

In:

auth_info: Data structure providing access to the user's credentials and additional policy information which might be used for approving authorization.

reservation_handle: The handle for the reservation that the user wishes to cancel.

grid_reservation_version

```
int grid_reservation_version(void);
```

This returns the current version number for the reservation manager. The current version number is GRID_RESERVATION_API_VERSION, as defined above.

grid_reservation_client_debug

```
int grid_reservation_client_debug(int debug_on);
```

In:

debug_on: If true, debugging is turned on. If false, it is turned off. The exact effect of turning on debugging mode is implementation-dependent.

grid_client_error_string

```
const char *grid_client_error_string(
          int error_code);
```

For any error code returned by the reservation manager, this provides a printable string that corresponds to the error code.

In:

error_code: The error code for which the user wishes to obtain a string representation.

5. Security Considerations

Security issues are not discussed in this document. The reservation scenario described here assumes that security is handled at the point of job authorization/execution on a particular resource.

6. Author Information

Alain Roy
Department of Computer Sciences
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
(608) 265-5736
roy@cs.wisc.edu

Volker Sander
Central Institute for Applied Mathematic
Forschungszentrum Jülich GmbH
52425 Jülich,
Germany,
+49 2461 616586
v.sander@fz-juelich.de

7. Copyright Notice

8. Intellectual Property Notice