Solstice Bandwidth Reservation Protocol 1.0 Programming Guide

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 805-0379-10
Revision A, March 1997

Contents

Tables

Preface

The *Solstice Bandwidth Reservation Protocol Programming Guide* describes the concepts necessary to use the RSVP API in sender or receiver applications. It includes a synopsis and description of each function together with procedural walkthroughs of initialization procedures.

For information regarding configuration and the logging facility, refer to the *Solstice Bandwidth Reservation Protocol Administration Guide*.

Who Should Use This Book

This book is for software engineers creating RSVP-enabled applications. They are likely to be experienced ‘C’ programmers.

How This Book Is Organized

Chapter 1, “Solstice Bandwidth Reservation Protocol API Introduction” briefly explains the API within the context of a sender or receiver application.

Chapter 2, “API Functions,” describes each API function using a synopsis, a description of its purpose, its arguments, and any return values or error codes.

Appendix A, “Solstice RAPI Objects,” details the terms and concepts used by RSVP to define notions of Quality of Service and traffic flow.

Appendix B, "Solstice RAPI Example Implementations," describes the procedures required to initialize parameters and fields in sender and receiver applications.

Related Books

The Use of RSVP with IETF Integrated Services. J Wroclawski October, 1996. Integrated Services Work Group Internet Draft.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

Table P-1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Abbreviations

The following abbreviations are used in this document.

Table P-3 Abbreviations

Abbreviation	**Meaning**
API	Application Program Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
bps	Bits Per Second
CCITT	International Telegraph and Telephone Consultative Committee
IA5	International Alphabet No. 5
ISO	International Organization for Standardization
OM	Object Management
OSI	Open Systems Interconnection
UCT	Universal Coordinated Time

Solstice Bandwidth Reservation Protocol API 1

In addition to providing the host parts of RSVP, Solstice Bandwidth Reservation Protocol offers an Application Programming Interface (API) which enables multimedia applications to take advantage of the RSVP protocol.

Calls to these functions are implemented into code in both sender and receiver applications. The RSVP API (RAPI) communicates with the `rsvpd` daemon over a UNIX socket. The `rsvpd` daemon translates the data between the function calls and the RSVP protocol.

A step by step description of the initialization process with example code is described in "RAPI Example Implementations" on page 37.

How the API Communicates with the Daemon

Applications use a series of function calls to establish and maintain an RSVP session. The following is a brief description of the main RAPI functions and their purpose within sender and receiver applications.

To establish an RSVP session, the application uses the `rapi_session()` function. If the application is to be used as a data sender, it will call `rapi_sender()`. If it is to act as a receiver, it will call the `rapi_reserve()` function to make a reservation for a Quality of Service.

The `rapi_getfd()` call returns the file descriptor of the socket opened by the API, between the application and the daemon. The file descriptor can then be used in a `select` call or passed to the notifier of another library.

When the file descriptor returns an event, the application calls `rapi_dispatch()`, which will execute the callback routines provided by the application to the API. if appropriate.

The session is closed using `rapi_release()`. If the application exits without properly closing the RAPI socket, the other side will be notified to perform a cleanup. Furthermore, if the user process terminates without explicitly closing the RAPI session, the `rsvpd` daemon will delete the corresponding reservation state from the routers.

Logging

For details of the logging facility and recommended levels of logging, see "RSVP Logging Facility" on page 17 of the *Solstice Bandwidth Management Administration Guide.*

Solstice Bandwidth Reservation Protocol API Functions 2

This chapter defines the API functions available in Solstice Bandwidth Reservation Protocol.

Refer to “RAPI Objects” on page 33 for an explanation of the terms and concepts used by RSVP to define notions of Quality of Service and traffic flow.

rapi_session()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

rapi_sid_t rapi_session(
          struct sockaddr_in        *Dest,
          int                       Protid,
          int                       flags,
          rapi_event_rtn_t          *Event_rtn,
          void                      *Event_arg,
          int                       *errnop,
          )
```

Description

rapi_session() is used by both sender and receiver applications to initialize the API prior to opening an RSVP session. It sends initialization information to the rsvpd daemon through a UNIX domain socket connection. The socket is opened the first time this function is called and will be used for further calls to rapi_session(). Applications using multiple streams of video on different multicast addresses will have to call this function once for each destination address.

When the function is called for the first time, it will try to open the pipe to the rsvpd daemon. If this fails, then the API will attempt to start the daemon, using two environment variables, RSVPDPATH and RSVPDCONF.

RSVPDPATH enables the user to tell the API where to find the daemon binary, in the event that the API needs to start the daemon. RSVPDCONF enables the user to tell the daemon where to find its configuration file, rsvpd.conf. The user will only need to set these variables when the daemon binary or rsvpd.conf are in non-standard locations.

If RSVPDPATH is set, this is used as the location of the daemon and the daemon is started from this directory. If RSVPDPATH is not set, the standard daemon locations /usr/lib and /opt/SUNWconn/rsvp/lib will be added to the PATH environment variable so that the daemon may be started. If the environment variable RSVPDCONF does not exist, then the API will set RSVPDCONF to *rsvpd_location*/../etc before starting the daemon (where *rsvpd_location* is the location of the rsvpd daemon). Otherwise RSVPDCONF is left unchanged. The daemon then uses RSVPDCONF to tell it the location of the rsvpd.conf configuration file.

Note – if the daemon is started manually and not by the API then the environment variable RSVPDCONF may not exist when the daemon runs. In this case, the daemon uses the standard default location /etc/opt/SUNWconn/rsvp.

Arguments

Dest

This parameter points to a structure of type sockaddr_in which contains the destination IP address (unicast or multicast) and the TCP/UDP port to which the data will be sent. This is mandatory for sender applications. Applications which act as receivers can set Dest=NULL.

Protid

This parameter is the IP protocol ID for the data session. If it is omitted (value 0), then protocol 17 is assumed (UDP).

flags

This parameter is reserved for future use.

Event_rtn

This parameter is an asynchronous callback routine to be notified by RSVP events. This parameter is optional.

Refer to rapi_dispatch() for more details about RSVP asynchronous events and the Event_rtn function.

Event_arg

This parameter points to the argument that will be passed to the Event_rtn function. This parameter is optional.

errnop

This parameter is the address of an integer where a RAPI error code will be stored.

Results

If rapi_session() succeeds, it returns a session identifier to be used in subsequent calls. If it fails, it returns NULL_SID and the location errnop will contain one of the values listed under Errors below.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the /include/rapi_err.h file.

RAPI_ERR_NORSVP, RAPI_ERR_INVAL, RAPI_ERR_MAXSESS,
RAPI_ERR_OVERFLOW, RAPI_ERR_OBJLEN, RAPI_ERR_OBJTYPE

rapi_release()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_release(
                                rapi_sid_t    Sid
                          )
```

Description

This function terminates the RSVP session by closing the connection to the `rsvpd` daemon.

Arguments

Sid

This is the session identifier returned by `rapi_session()`.

Results

If `rapi_release()` succeeds, it returns `RAPI_ERR_OK`. If it fails, it returns `RAPI_ERR_BADSID`.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the `/include/rapi_err.h` file.

`RAPI_ERR_BADSID`

rapi_sender()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_sender(
         rapi_sid_t            Sid,
         int                   flags,
         struct sockaddr_in    *Host,
         rapi_filter_t         *SenderTemplate,
         rapi_tspec_t          *SenderTspec,
         rapi_adspec_t         *SenderAdspec,
         rapi_policy_t         *SenderPolicy,
         int                   TTL
         )
```

Description

Once `rapi_session()` has returned successfully, the RSVP application must specify its intended function to the `rsvpd` daemon. If the application is to act as a sender, it must call `rapi_sender()` which will instruct the daemon to begin sending `PATH` messages for the session.

Arguments

Sid

This is the session identifier returned by `rapi_session()`.

flags

Reserved for future use.

Host

If this parameter is `NULL`, the application wishes to withdraw its registration as a sender. In this case, the other parameters are ignored and a `PATH_TEAR` message will be generated. If the IP source address is `INADDR_ANY`, and the system has a single network interface, the API will use the default IP address of the local host.

If the IP source address is INADDR_ANY and the system has more than one network interface, the API will select the address that is specified by the IP routing tables (this could also be the address associated with the route for the 224.0.0.0 network if the destination address is multicast).

The Bandwidth Reservation Protocol must be enabled on those interfaces used to send RSVP packets, otherwise an error will be produced. Use the default setting of all in the rsvpd.conf file to enable RSVP on all interfaces and avoid potential conflicts.

SenderTemplate

Reserved for future use.

SenderTspec

Describes the traffic that the receiver will have to reserve. This parameter is mandatory. For further information, see "Assigning a Value to Parameter 5 in rapi_sender() (rapi_tspec_t)" on page 41.

SenderAdspec

Describes the One Path With Advertisement (OPWA) parameters. Assigning a value to this parameter is NOT recommended. The daemon will automatically assign a default value.

SenderPolicy

Reserved for future use.

TTL

This parameter specifies the IP 'Time To Live' value with which multicast data will be sent.

Results

If rapi_session() succeeds, it returns a session identifier to be used in subsequent calls. If it fails, it returns NULL_SID and the location errnop will contain one of the values listed under Errors below.

Errors

RAPI_ERR_NORSVP, RAPI_ERR_INVAL, RAPI_ERR_MAXSESS, RAPI_ERR_OVERFLOW, RAPI_ERR_OBJLEN, RAPI_ERR_OBJTYPE

rapi_reserve()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_reserve(
          rapi_sid_t             Sid,
          int                    flags,
          struct sockaddr_in     *Rhost,
          rapi_styleid_t         StyleId,
          rapi_stylex_t          *Style_ext,
          rapi_policy_t          *Rcvr_policy,
          int                    FilterSpecNo,
          rapi_filter_t          *FilterSpec_list,
          int                    FlowSpecNo,
          rapi_flowspec_t        *FlowSpec_list
          )
```

Description

Solstice Bandwidth Reservation Protocol makes receivers responsible for their reservations and `rapi_reserve()` enables receivers to make, modify or delete a resource reservation for a session.

Some connection-oriented protocols such as ATM or Frame Relay Switched Virtual Circuits reserve network resources for the lifetime of the connection. Solstice Bandwidth Reservation Protocol, however, allows applications to change their reservation during the same session. `rapi_reserve()` is a complex call as it attempts to handle the most general reservation case for multiple senders on the same multicast session.

Arguments

Sid

This is the session identifier returned by `rapi_session()`.

flags

No flags are currently supported.

RHost

This parameter defines the interface address on which data will be received. If this value is omitted or the host address is `INADDR_ANY`, the application will choose the interface defined by the routing tables and the `rsvpd.conf` file.

If a host address is specified, the application should check that the interface is consistant with the routing table and the interfaces specified in the `rsvpd.conf` file. This parameter is optional.

StyleId

Specifies the style of the reservation.

rapi_stylex_t

Reserved for future use.

rapi_policy_t

Reserved for future use.

FilterSpecNo

Specifies the number of filterspecs corresponding to the different senders.

FilterSpec_list

Points to the filterspecs list.

FlowSpecNo

Specifies the number of flowspecs corresponding to the different senders.

FlowSpec_list

Points to the flowspecs list.

Styles

If *FlowSpecNo* is zero, the `rapi_reserve` call will remove any current reservation for the specified session. Otherwise, the parameters depend upon the following styles:

Wildcard Filter (WF)

Use *StyleId*=`WILDCARD`.The *FlowSpec_list* parameter may be empty (to delete a reservation) or point to a single flowspec. The *FilterSpec_list* parameter may be empty or point to a single filterspec containing appropriate wildcards.

Fixed Filter (FF)

Use *StyleId*=`FIXED`. *FilterSpec_No* must be equal to *FlowSpec_No.* Both lists may be empty (to delete the reservation) or of unrestricted length. Entries in *FlowSpec_list* and *FilterSpec_list* parameters will correspond in pairs.

Shared Explicit (SE)

Use *StyleId*=`SE`. The *FlowSpec_list* parameter may be empty (to delete the reservation) or point to a single flowspec. The *FilterSpec_list* parameter may point to a list of any length.

Results

If `rapi_reserve()` succeeds, it returns `RAPI_ERR_OK`. If it fails, it returns one of the values listed under Errors below.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the `/include/rapi_err.h` file.

```
RAPI_ERR_BADSID, RAPI_ERR_OVERFLOW, RAPI_ERR_NORSVP,
RAPI_ERR_SYSCALL, RAPI_ERR_N_FFS, RAPI_ERR_INVAL,
RAPI_ERR_BADSTYLE, RAPI_ERR_OBJLEN, RAPI_ERR_OBJTYPE
```

rapi_dispatch()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_dispatch()
```

Description

Applications use `rapi_dispatch()` to receive notification of RSVP events. It reads and interprets events from the daemon and executes the event callback routine passed by the application to the API library when `rapi_session()` is called.

For polling, it is sufficient to call `rapi_dispatch()` without getting the RSVP socket file descriptors from the `rapi_getfd()` function. This is because `rapi_dispatch()` calls `select()` on all the RSVP socket file descriptors. The `select()` function is called with an empty timeout object, and therefore acts as a poll. The alternative is to call `rapi_getfd()` to get an RSVP file descriptor, add it to a `poll()` or `select()` call, and call `rapi_dispatch()` when `select()` indicates an event on the RSVP file descriptor.

Applications may process RSVP events when `rapi_dispatch()` executes the callback routine `Event_rtn()` passed to `rapi_session()`. Two events are communicated to `Event_rtn()`:

- A set of parameters supplied by the library which detail the result of a previous `rapi_reserve()` function, the arrival of an RSVP `PATH`, `PATH_TEAR` or `RESV_TEAR` message.
- The `Event_arg` pointer supplied by `rapi_session()` to the library.

Arguments

The event callback routine passed when `rapi_session()` is called, has the following prototype:

```
  int Event_rtn
(
    rapi_sid_t           Sid,
    rapi_eventinfo_t     EventType,
    rapi_styleid_t       StyleId
    int                  ErrorCode,
    int                  ErrorValue,
    struct sockaddr_in,  ErrorNodeAddress,
    u_char               ErrorFlags,
    int                  FilterSpecNo,
    rapi_filter_t        *FilterSpec_list,
    int                  FlowSpecNo,
    rapi_flowspec_t      *FlowSpec_list,
    int                  AdSpecNo,
    rapi_adspec_t        *AdSpec_list,
    void                 *Event_arg
    )
```

Sid

This is the session identifier returned by `rapi_session()`.

EventType

This argument contains the RSVP event returned by the library. The library defines the following RSVP events:

`RAPI_PATH_EVENT`	received when the `rsvpd` daemon gets a `PATH` or `PATH_TEAR` message from the network.
`RAPI_PATH_ERROR`	received when the `rsvpd` daemon gets a `PATH_ERR` message as a response to a previous `PATH` message.
`RAPI_RESV_EVENT`	received when the `rsvpd` daemon gets a `RESV` or `RESV_TEAR` message from the network.
`RAPI_RESV_ERROR`	received when the `rsvpd` daemon gets a `RESV_ERR` message as a response to a previous `RESV` message.
`RAPI_RESV_CONFIRM`	received when the rsvpd daemon gets a `RESV_CONFIRMATION` message.

StyleId

Specifies the style of the reservation.

ErrorCode

This argument contains the error code returned by the library.

ErrorValue

This argument contains a value corresponding to *ErrorCode*.

ErrorNode

This argument contains the node address that generated the error.

ErrorFlags

This argument contains the error flags.

FilterSpec_list, FilterSpecNo

FilterSpec_list is a pointer to an area which contains filterspec or sender template objects. This allows a sender to read the Tspecs and Quality of Service requested by a receiver. (The event is `RAPI_RESV_EVENT`). The number of objects in this vector is specified in *FilterSpecNo*. If *FilterSpecNo* is zero, the *Filterspec_list* parameter will be `NULL`.

FlowSpec_list, *FlowSpecNo*

FlowsSpec_list is a pointer to an area which contains flowspec or Tspec objects. The number of objects in this vector is specified in *FlowSpecNo*. If *FlowSpecNo* is zero, the *Flowspec_list* parameter will be `NULL`.

AdSpec_list, *AdSpecNo*

AdSpec_list is a pointer to an area which contains Adspec objects. The number of objects in this vector is specified in *AdSpecNo*. If *AdSpecNo* is zero, the *Adspec_list* parameter will be `NULL`.

Event_arg

This is a copy of the *Event_arg* pointer supplied with `rapi_session()`.

Results

If rapi_dispatch succeeds, it returns RAPI_ERR_OK. If it fails, it returns one of the values listed under Errors below.

The return value from the event callback routine is currently unused by the daemon.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the /include/rapi_err.h file.

RAPI_ERR_BADSID, RAPI_ERR_NORSVP, RAPI_ERR_SYSCALL, RAPI_ERR_MEMFULL, RAPI_ERR_OVERFLOW, RAPI_ERR_OBJLEN, RAPI_ERR_INVAL, RAPI_ERR_OBJTYPE.

rapi_getfd()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_getfd(
          rapi_sid_t     Sid
          )
```

Description

Each time an RSVP-enabled application calls `rapi_session()` to initiate a new RSVP session, the API library opens an AF_UNIX domain socket and connects to the `rsvpd` daemon. All subsequent RSVP calls read or write on this socket. The `rapi_getfd()` function can be used to retrieve the socket file descriptor to put it in a `poll()` or `select()` call list.

Arguments

Sid

This is the session identifier returned by `rapi_session()`.

Results

If `rapi_getfd()` completes successfully, it will return the socket file descriptor. If it fails, it will return -1.

Errors

`-1`

rapi_lock()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

 void rapi_lock()
```

Description

This function is for multi-threaded use of the API. Any call to this function will lock all other API functions. For example, it can be used in the function called by `rapi_dispatch()`. It is not necessary to call `rapi_lock()` before an API function because the API function automatically calls `rapi_lock()` and `rapi_unlock()`.

Arguments

There are no arguments for `rapi_lock()`.

Results

There are no return values for `rapi_lock()`.

Errors

There are no error codes for `rapi_lock()`.

rapi_unlock()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

 void rapi_unlock()
```

Description

This function unlocks the access to the API functions.

Arguments

There are no arguments for `rapi_unlock()`.

Results

There are no return values for `rapi_unlock()`.

Errors

There are no error codes for `rapi_unlock()`.

rapi_version()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_version()
```

Description

This function returns the current version of the API library.

Arguments

There are no arguments for `rapi_version()`.

Results

Returns the current version of Solstice Bandwidth Reservation Protocol API.

Errors

There are no error codes for `rapi_version()`.

rapi_status()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

 rapi_status(
         rapi_sid_t              Sid,
         int                     flags
         )
```

Description

This function requests the daemon to notify the RSVP library of any PATH messages for a specified session which it may be storing. The application must call rapi_dispatch to receive these events. This function is not generally required as upcalls are automatically generated by the rsvpd daemon.

Arguments

Sid

This is the session identifier for which information will be retrieved. If *Sid* is set to 0, it will prompt an upcall for all the session identifiers that this process owns.

Flags

This should be set to RAPI_DIR_SEND. RAPI_DIR_RECV is not currently supported.

Results

If rapi_status() succeeds, it returns RAPI_ERR_OK. If it fails, it will return any other value.

Errors

Any value other than RAPI_ERR_OK.

rapi_fmt_flowspec()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

void rapi_fmt_flowspec(
                    rapi_flowspec_t    *Flowspec,
                    char               *buff,
                    int                len
                    )
```

Description

This flow specification describes a desired Quality of Service. The function formats a given RAPI flowspec template into a buffer of a specified address and size. The output is truncated if the size is too small.

Arguments

Flowspec

This parameter contains the flowspec structure to format.

buff

This parameter contains the result of the format.

len

This parameter contains the length of the supplied buffer.

Results

There are no return values for `rapi_fmt_flowspec()`.

Errors

There are no error codes for `rapi_fmt_flowspec()`.

rapi_fmt_filterspec()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

void rapi_fmt_filterspec(
                    rapi_filterspec_t  *Filterspec,
                    char               *buff,
                    int                len
)
```

Description

This filter specification describes the IP address of the sender and port number being used. This function formats a given RAPI filterspec template into a buffer of a specified address and size. The output is truncated if the size is too small.

Arguments

Filterspec

This parameter contains the filterspec structure to format.

buff

This parameter contains the result of the format.

len

This parameter contains the length of the supplied buffer.

Results

There are no return values for `rapi_fmt_filterspec()`.

Errors

There are no error codes for `rapi_fmt_filterspec()`.

rapi_fmt_tspec()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

void rapi_fmt_tspec(
                     rapi_tspec_t        *Tspec,
                     char                *buff,
                     int                 len
                     )
```

Description

This traffic specification describes a flow of data traffic. This function formats a given RAPI Tspec template into a buffer of a specified address and size. The output is truncated if the size is too small.

Arguments

Tspec

This parameter contains the Tspec structure to format.

buff

This parameter contains the result of the format.

len

This parameter contains the length of the supplied buffer.

Results

There are no return values for `rapi_fmt_tspec()`.

Errors

There are no error codes for `rapi_fmt_tspec()`.

rapi_fmt_adspec()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

void rapi_fmt_adspec(
                     rapi_adspec_t      *Adspec,
                     char               *buff,
                     int                len
                     )
```

Description

The Advertisement specification carries One Pass With Advertisement (OPWA) data in a PATH message. This call formats a given RAPI Adspec template into a buffer of a specified address and size. The output is truncated if the size is too small.

Arguments

Adspec

This parameter contains the Adspec structure to format.

buff

This parameter contains the result of the format.

len

Contains the length of the supplied buffer.

Results

There are no return values for `rapi_fmt_adspec()`.

Errors

There are no error codes for `rapi_fmt_adspec()`.

rapi_IS_tspec_init()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_IS_tspec_init(
                rapi_tspec_t *          new_sender_Tspec,
                u_char                  version,
                float32_t               rate,
                float32_t               depth,
                float32_t               peak,
                uint32_t                min_unit,
                uint32_t                max_unit
                )
```

Description

This function initializes an Integrated Services Tspec object before calling the `rapi_sender()` function to request `PATH` message generation. It can be used by the data sender application requesting the generation of RSVP `PATH` messages.

Arguments

new_sender_Tspec

The Tspec to be initialized by the function. This parameter is mandatory.

version

The Integrated Services version.

rate

The token bucket rate. This value is measured in bps of IP datagrams and can range from 1 byte to 40 terrabytes per second. For large bandwidths, only the first few digits are significant. This value is mandatory.

depth

The token bucket depth. This value is measured in bps of IP datagrams and can range from 1 byte to 250 gigabytes per second. For large bandwidths, only the first few digits are significant. By default, it is initialized to twice the rate.

peak

The peak rate. The maximum rate at which the source may inject bursts of traffic onto the network. This value is measured in bps of IP datagrams and can range from 1 byte to 40 terrabytes per second. It must be greater than or equal to the token bucket rate. By default, it is initialized to twice the rate.

min_unit

The minimum policed unit. This is an integer value measured in bytes. All IP datagrams less than this value will be considered by the token bucket as being of this value. By default, it is initialized to 0.

max_unit

The maximum packet size. This is an integer value measured in bytes. The flow will be rejected if the requested maximum packet size is larger than the MTU of the link. By default, it is initialized to 32000.

Results

If `rapi_IS_tspec_init()` succeeds, it will return the value `RAPI_ERR_OK`. If it fails, it will return an error code.

Errors

Error codes and their meanings are referenced in the ‘rapi_err’ section of the `/include/rapi_err.h` file.

`RAPI_ERR_TSPEC, RAPI_ERR_BADRATE, RAPI_ERR_NON_POSITIVE, RAPI_ERR_PEAK, RAPI_ERR_MAX_UNIT`

rapi_IS_fspec_G_init()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_IS_fspec_G_init(
               rapi_tspec_t *         old_Tspec,
               rapi_adspec_t *        old_Adspec,
               rapi_flowspec_t *      new_Flowspec,
               float32_t              g_rate,
               uint32_t               slack
               )
```

Description

This function converts a single Tspec object received from a `PATH` message into an Integrated Services Guaranteed Quality of Service flowspec object before calling the `rapi_reserve()` function to request `RESV` message generation. It can be used by the data receiver application requesting RSVP reservation.

Arguments

old_Tspec

The received Tspec. This parameter is mandatory.

old_Adspec

The received Adspec. This parameter is optional but strongly recommended.

new_Flowspec

The flowspec to be initialized from *old_Tspec*. This parameter is mandatory.

g_rate

This parameter describes the maximum end-to-end bandwidth provided along the data path. It must be greater than or equal to the token bucket rate.

slack

The slack term. This is the difference between the desired delay and the delay obtained by using a reservation level. This value must be non-negative. By default, it is initialized to 0.

Results

If `rapi_IS_fspec_G_init` succeeds, it will return `RAPI_ERR_OK`. If it fails, it will return an error code.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the `/include/rapi_err.h` file.

`RAPI_ERR_TSPEC, RAPI_ERR_FLOWSPEC, RAPI_ERR_GUAR_RATE, RAPI_ERR_NON_POSITIVE, RAPI_ERR_OBSOLETE`

rapi_IS_fspec_CL_init()

Synopsis

```
#include <rapi_lib.h>
#include <rapi_err.h>

int rapi_IS_fspec_CL_init(
                rapi_tspec_t *          old_Tspec,
                rapi_adspec_t *         old_Adspec,
                rapi_flowspec_t *       new_Flowspec
                )
```

Description

This function converts a single Tspec object received from a PATH message into an Integrated Services Controlled Load Quality of Service flowspec object before calling the rapi_reserve() function to request RESV message generation. It can be used by the data receiver application requesting RSVP reservation.

Arguments

old_Tspec

The received Tspec. This parameter is mandatory.

old_Adspec

The received Adpec. This parameter is optional.

new_Flowspec

The flowspec to be initialized from *old_Tspec*. This parameter is mandatory.

Results

If rapi_IS_fspec_CL_init succeeds, it will return the value RAPI_ERR_OK. If it fails, it will return an error code.

Errors

Error codes and their meanings are referenced in the 'rapi_err' section of the `/include/rapi_err.h` file.

`RAPI_ERR_TSPEC, RAPI_ERR_FLOWSPEC, RAPI_ERR_OBSOLETE`

RAPI Objects A

This section describes the terms and concepts used by the RSVP protocol to define notions of Quality of Service and traffic flow. These concepts are Tspecs (Traffic specifications), flowspecs (flow specifications), filterspecs (filter specifications) and Adspecs (advertisement specifications). The concept of the token bucket is also explained.

Integrated Services have defined the following service specifications for the reservation protocol.

Service Specifications used by Sender Applications

Tspec

Tspec is used by sender applications only, to describe the traffic flow they will generate. This data is later used by receiver applications when formulating their reservation demands.

Token Bucket Field Parameters

The following parameters are token bucket fields within a Tspec which describe the shape of the data flow.

Token bucket rate

Token bucket rate. This value is measured in bytes per second of IP datagrams and can range from 1 byte to 40 terrabytes per second.

This field must contain a positive value. For large bandwidths, only the first few digits are significant so using floating point representations to at least 0.1% is encouraged.

Token bucket depth

Token bucket depth. This value is measured in bps of IP datagrams and can range from 1 byte to 250 gigabytes per second.

This field must contain a positive value. For large bandwidths, only the first few digits are significant so using floating point representations to at least 0.1% is encouraged.

Peak rate

This parameter specifies the maximum rate at which the sender application may transmit bursts of traffic onto the network.

This value is measured in bps of IP datagrams and can range from 1 byte to 40 terrabytes per second. It must be greater than or equal to the token bucket rate.

Minimum policed unit

Minimum policed unit. This is an integer value measured in bytes. All IP datagrams less than this value will be considered by the token bucket as being of this value.

Maximum packet size

Maximum packet size. This is an integer value measured in bytes. The flow will be rejected if the requested maximum packet size is larger than the Maximum Trabsfer Unit (MTU) of the link.

None of the parameters contained within a Tspec are modified by any network element between the sender and receiver. If information is to be propagated to receivers, it is added to the Adspec template.

Adspec

Adspec data are contained in `PATH` messages only and carries one pass with advertisement information (OPWA). Adspecs are used to propagate information about the path to receivers.

For example, a router should complete the MTU (Minimum Transfer Unit) field as the packet travels through, enabling the receiver to trace the MTU value over the whole path. Alternatively, by specifying an Adspec at `rapi_sender()` time, a sender application can restrict the type of service it wishes to advertise (Controlled Load or Guaranteed service).

Service Specifications used by Receiver Applications

flowspec

This is a flow specification which describes a desired Quality of Service. The Quality of Services currently supported are Guaranteed and Controlled Load. For Controlled Load, only a Tspec is necessary. Guaranteed service requires a Tspec and also an Rspec which contains the parameters *slack term* and *guaranteed rate*.

Guaranteed service

Guaranteed service ensures that packets will arrive within the guaranteed delivery time and not be discarded in the event of queue overflows. The guarantee will only be valid if the traffic flow remains within its specified traffic parameters.

This service is intended for applications which require packets to arrive no later than a certain time after transmission by the sender application. For example, audio and video play-back applications may be intolerant of packets arriving after their play-back time. Guaranteed service is also necessary for applications with real-time requirements.

The *slack term* parameter contains the difference between the desired delay and the delay obtained by using a reservation level. This value must be positive.

The *guaranteed rate* describes the maximum end-to-end bandwidth provided along the data path. It must be greater than or equal to the token bucket rate becuase higher rates will reduce queuing delay. This value is expressed in bytes per second.

Controlled Load service

The Controlled Load service is designed to support a broad class of applications which are highly sensitive to overloaded conditions.

filterspec

The receiver application provides a filterspec to indicate to intermediate RSVP routers which senders should be covered by the reservation. This enables the receiver applications to apply the corresponding Quality of Service to data from the sender applications only.

Automatic Initialization

RAPI functions are available to automatically initialize Tspec and flowspec objects.

To initialize Tspec objects, use `rapi_IS_tspec_init()`. To initialize RAPI flowspec objects for a Controlled Load Quality of Service, use `rapi_IS_fspec_CL_init()`. To initialize RAPI flowspec objects for a Guaranteed Quality of Service use `rapi_IS_fspec_G_init()`.

Refer to "RAPI Example Implementations" on page 37 for a step by step description of service parameter initialization and the `C` structures involved.

RAPI Example Implementations — B

This appendix describes the procedures required to initialize parameters and fields in Solstice Bandwidth Reservation Protocol API. It consists of two parts, describing the implementation process in:

- a multicast sender application
- a multicast receiver application

The configuration of the parameters is presented in separate functional stages, starting with "Opening an RSVP session", "Sending RSVP packets", "Waiting for RSVP messages" and so on.

The reasons for assigning certain values to each parameter are explained and illustrated with code examples. Many of the procedures are relevant to both sender and receiver applications, highlighted by references between the two parts.

Some stages will not be relevant and can be omitted, depending upon the preferences of the user. For example, "Assigning a value to parameter 5 in `rapi_sender()` is for users who have chosen manual initialization of a parameter instead of using a function. These choices are indicated where appropriate.

Implementing Solstice Bandwidth Reservation Protocol in a Sender Application

Solstice Bandwidth Reservation Protocol can be implemented into a data sender application by adding some simple calls to RAPI functions. There are two important considerations when introducing these calls into the source code:

- Deciding where to place the calls
- Using the correct parameters to be passed to the RAPI functions

This section contains typical example configurations for establishing and maintaining an RSVP session for a sender application. It identifies the relevant functions and illustrates the correct places for them. Technical design options are emphasized where more than one solution is available.

Opening an RSVP Session

To start sending RSVP packets, you must first open an RSVP session using the `rapi_session()` API function. This must be the first RAPI function to be called from the sender application code.

An example call to the `rapi_session()` function:

```
Sid = rapi_session(&Multicast_address_destination,
 IPPROTO_UDP, 0, rsvp_callback, (void *) &RSVP_arg, &RSVP_error)
```

Parameter 1

This parameter is a structure of type `sockaddr_in`. It contains the multicast address to be used for the communication between sender and receiver applications. To receive data packets for a particular multicast session a host must have joined the corresponding IP multicast group. This can be done by calling the `setsockopt` function and setting the `IP_ADD_MEMBERSHIP` option. This must be done before calling the `rapi_session()` function.

Parameter 2

This parameter specifies the transport protocol of the dataflow being sent. This is not the protocol used to send the RSVP packets.

Parameter 4

This parameter specifies the address of the function that will be executed when a call is made to the `rapi_dispatch()` function. See “Retrieving and Processing RSVP Messages” on page 50.

Parameter 5

This parameter contains a pointer to the arguments passed to the function specified in *Parameter 4*. This is explained in more detail in “Waiting for RSVP Messages” on page 48 and “Retrieving and Processing RSVP Messages” on page 50.

Error Handling

`rapi_session()` is a blocking function and will not return until `rsvpd` daemon has established the requested RSVP session. The value returned will be a session identifier (*Sid*) value. If an error occurs whilst establishing the session, the return value will be 0 and *Parameter 7* will contain the error number. The error number can be referenced in the ‘rapi_errlist’ string array defined in the `rapi_err.h` include file.

For example:

```
if ( Sid == 0 )  /* An error has been produced */
printf("rapi_session returns an error: %s \n",
rapi_errlist[RSVP_error]);
```

Sending RSVP Packets

Once the RSVP session has been opened, the sender application must call `rapi_sender()` to transmit `PATH` messages. `PATH` messages are RSVP packets that describe the dataflow being transmitted to the receiver applications. The receivers use the `PATH` message information to request suitable bandwidth reservations.

`PATH` messages are generated by the `rsvpd` daemon at regular intervals, usually every 30 seconds. A single call to the `rapi_sender()` function requests the daemon to begin periodic `PATH` message generation.

Below are some recommendations for assigning values to `rapi_sender()` parameters.

An example call to the `rapi_sender()` function:

```
rapi_sender(Sid, 0, &Sender_host, NULL, &SenderTspec, NULL, NULL,
TTL)
```

Parameter 1

This parameter is the session identifier previously returned by the `rapi_session()` function.

Parameter 3

This parameter is a structure of type `sockaddr_in` and contains *the data sender address, not the multicast group address*. The multicast group address has already been passed to the `rsvpd` daemon by the `rapi_session()` function. Entering the multicast group address in *Parameter 3* of `rapi_sender()` is a common and serious error.

When multicast dataflow is to be sent through the default network interface, *Parameter 3* can be initialized by setting:

```
Sender_host.sin_addr.s_addr = INADDR_ANY;
```

This must be done before calling the `rapi_sender()` function.

Parameter 5

This parameter is used to initialize a sender Tspec template to be contained in the `PATH` message. It is recommended that you use the `rapi_IS_tspec_init()` function which automatically initializes Integrated Services Tspecs. If you decide to manually initialize the Tspecs, refer to the section “Assigning a Value to Parameter 5 in rapi_sender() (rapi_tspec_t)” on page 41.

Parameter 8

This is the 'Time to Live' parameter. It is related to the number of RSVP controlling objects the PATH message is going to pass through.

Table B-1 Recommended values for *TTL*

Network	Value
Broadcasting (same sub-net)	2
Multicast for a WAN	16
Multicast for a country	32
Multicast for a continent	64
Multicast for the whole world	255

Note – The value of 2 rather than 1 is recommended for broadcasting. This is because the rsvpd daemon decrements the *TTL* value before sending the PATH message.

Assigning a Value to Parameter 5 in rapi_sender() *(*rapi_tspec_t*)*

This section describes the initialization procedure for all the rapi_tspec_t() fields previous to calling the rapi_sender() function. It is not necessary to read this section if you are going to use rapi_IS_tspec_init() to initialize these fields automatically.

Parameter 5 in rapi_sender() is a structure of type rapi_tspec_t(). There are many possible implementations of its components (qos_flowspec_t, qos_flowspecx_t, qos_tspecx_t, IS_specbody_t and IS_tspbody_t), which makes manual initialization a complex task.

Data sender applications generate Tspecs to create PATH messages and data receiver applications generate flowspecs to create RESV messages. As illustrated in the definitions below, rapi_tspec_t and rapi_flowspec_t are of the same structural definition.

```
typedef struct {
int     len;/* actual length in bytes */
int     form;/* flowspec format (see enum above) */
    union {
```

```
    qos_flowspec_t qos;/* Legacy format */
    qos_flowspecx_t qosx;/* Extended legacy flowspec */
    qos_tspecx_t qosxt;/* Extended legacy Tspec */
    IS_specbody_t IS;/* Int-serv format flowspec */
    IS_tspbody_t ISt;/* Int-serv format Tspec */
    } specbody_u;
} rapi_flowspec_t;

typedef rapi_flowspec_t rapi_tspec_t; /* Same struct definition
*/
```

`qos_flowspec_t`

This is the data type used to fill in the legacy formatted flowspec and is not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`qos_tspecx_t` *and* `qos_flowspecx_t`

These are the data types used to fill in the extended legacy formatted Tspec and flowspec respectively. They are an extension to `qos_flowspec_t` but are not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`IS_tspbody_t` *and* `IS_specbody_t`

These are the data types used to fill in Integrated Services formatted Tspec and flowspec respectively. These are the standard formats chosen by the Integrated Services Workgroup (IETF) and are the most suitable choices for the `specbody_u` union. It is recommended that you use `IS_tspbody_t` when you want to send `PATH` messages in data sender applications and `IS_specbody_t` when you want to make `RESV` messages in data receiver applications.

You must use `IS_tspbody_t` to complete *Parameter 5*.

`IS_tspbody_t` offers a choice between two unions:

```
typedef struct {
IS_main_hdr_tst_mh;
    union {
    CL_Tspec_t   CL_stspec;     /* Controlled Load Tspec*/
```

```
    gen_Tspec_t  gen_stspec;     /* Generic Tspec */
    } tspec_u;
} IS_tspbody_t;
```

There are two types of Integrated Services Tspecs:

1. `GENERAL_INFO`. A generic service which allows the receiver applications to select the desired Quality of Service.

2. `CONTROLLED_LOAD_SERV`. A Controlled Load service which forces receiver applications to request a Controlled Load Quality of Service for their intended reservations.

The choice of the Tspec union component depends upon which type of Tspec is going to be used. Choose `gen_Tspec_t` when building a `GENERAL_INFO` Tspec and `CL_Tspec_t` for `CONTROLLED_LOAD_SERV`.

For the most flexible service, it is recommended that you use `gen_Tspec_t`. This allows the receiver to choose between Controlled Load and Guaranteed Service Quality of Service when requesting reservations. Using `CL_Tspec_t`, however, forces the receivers to request Controlled Load Quality of Service.

Below is an example of how a `rapi_tspec_t` can be initialized. For this example, `GENERAL_INFO` service is used.

```
rapi_tspec_t SenderTspec;
SenderTspec.len  = 2*sizeof(int) + sizeof(IS_main_hdr_t) +
sizeof(gen_Tspec_t);
SenderTspec.form = RAPI_FLOWSTYPE_Intserv;

SenderTspec.specbody_u.ISt.st_mh.ismh_version =
INTSERV_VERSION0;
SenderTspec.specbody_u.ISt.st_mh.ismh_unused  = 0;
SenderTspec.specbody_u.ISt.st_mh.ismh_len32b  =
sizeof(gen_Tspec_t)+3)/4;

gen_Tspec_t *Continue =
&(SenderTspec.specbody_u.ISt.tspec_u.gen_stspec);

Continue->gen_Tspec_serv_hdr.issh_service  = GENERAL_INFO;
Continue->gen_Tspec_serv_hdr.issh_len32b   =
(sizeof(IS_parm_hdr_t)+sizeof(TB_Tsp_parms_t)+3)/4;
Continue->gen_Tspec_serv_hdr.issh_break    = 0;
```

```
Continue->gen_Tspec_parm_hdr.isph_parm_num = IS_WKP_TB_TSPEC;
Continue->gen_Tspec_parm_hdr.isph_flags    = 0;
Continue->gen_Tspec_serv_hdr.isph_len32b   =
(sizeof(TB_Tsp_parms_t)+3)/4;

Continue->gen_Tspec_parms.TB_Tspec_r       = rate;
Continue->gen_Tspec_parms.TB_Tspec_b       = 2*rate;
Continue->gen_Tspec_parms.TB_Tspec_p       = 2*rate;
Continue->gen_Tspec_parms.TB_Tspec_m       = 0;
Continue->gen_Tspec_parms.TB_Tspec_M       = 32000;
```

len **field**

The len field contains the exact number of bytes of the flowspec or Tspec object. This value is format dependant and so determined by the selected format, for example legacy, extended legacy or Integrated Services.

`rapi_tspec_t` objects do not have padding bytes so the lengths may be calculated using the `sizeof` function.

form **field**

Three different format fields are supported: legacy, extended legacy and Integrated Services.

`qos_flowspec_t`

This is the data type used to fill in the legacy formatted flowspec and is not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`qos_tspecx_t` ***and*** `qos_flowspecx_t`

These are the data types used to fill in the extended legacy formatted Tspec and flowspec respectively. They are an extension to `qos_flowspec_t` but are not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`IS_tspbody_t` *and* `IS_specbody_t`

These are the data types used to fill in Integrated Services formatted Tspec and flowspec respectively. These are the standard formats chosen by the Integrated Services Workgroup and are the most suitable choices for `specbody_u` union. It is recommended that you use `IS_tspbody_t` when you want to send `PATH` messages in data sender applications and `IS_specbody_t` when you want to make `RESV` messages in data receiver applications.

ismh_len32b, *issh_len32b* and *isph_len32b* fields

These fields are format dependent. They indicate the number of remaining 32-bit words from the point they are placed to the end of the `rapi_tspec_t` object.

Table B-2 *len_32* fields in a Tspec template

<table>
<tr><td></td><td>32</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>24</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>16</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td></tr>
<tr><td>1</td><td colspan="4">0</td><td colspan="12">reserved</td><td colspan="16">ismh_len32b</td></tr>
<tr><td>2</td><td colspan="8">1</td><td>0</td><td colspan="7">reserved</td><td colspan="16">issh_len32b</td></tr>
<tr><td>3</td><td colspan="8">127</td><td colspan="8">0</td><td colspan="16">isph_len32b</td></tr>
<tr><td>4</td><td colspan="32">token bucket rate r (32 bit 1EEE floating point number)</td></tr>
<tr><td>5</td><td colspan="32">token bucket size b (32 bit 1EEE floating point number)</td></tr>
<tr><td>6</td><td colspan="32">peak data rate p (32 bit 1EEE floating point number)</td></tr>
<tr><td>7</td><td colspan="32">minimum policed unit m (32 bit integer)</td></tr>
<tr><td>8</td><td colspan="32">maximum packet size M (32 bit integer)</td></tr>
</table>

For instance, Table B-2 illustrates a Tspec template with the *len32b* fields:

- *ismh_len32b.* This should be assigned the value 7 because there are 7 remaining words, excluding the header.
- *issh_len32b.* This should be assigned the value 6 because there are 6 remaining words, excluding the header.
- *isph_len32b* This should be assigned the value 5 because there are 5 remaining words, excluding the header.

issh_service field

There are two services available for *issh_service*:

1. GENERAL_INFO. A generic service which allows the receiver applications to select the desired Quality of Service.
2. CONTROLLED_LOAD_SERV. A Controlled Load service which forces receiver applications to request a Controlled Load Quality of Service for their intended reservations.

Considering the evolution of the RSVP protocol, it is recommended that you use GENERAL_INFO.

Assigning Values to Token Bucket Parameters

TB_Tspec_r field

TB_Tspec_r contains the token bucket rate. This is the rate in bits per second of IP datagrams of the multicast dataflow.

TB_Tspec_b field

TB_Tspec_b contains the token bucket depth. This parameter limits the bursts of the traffic so that, for a period of T seconds, the flow will be restricted to (*TB_Tspec_b* + (T * *TB_Tspec_r*)).

It is recommended that you set this parameter to a value higher than the *TB_Tspec_r* but below *TB_Tspec_r* *2. For example, *TB_Tspec_b* = 1.4 * *TB_Tspec_r*.

TB_Tspec_m field

TB_Tspec_m is the minimum policed unit. All IP datagrams less than this value will be considered by the token bucket as being of this value. To benefit from the burst limiting features introduced by the *TB_Tspec_b* field, set *TB_Tspec_M* to 0.

TB_Tspec_M field

TB_Tspec_M is the maximum packet size that the sender application wishes to send. Similarly, when receiver applications request reservations, the flow will be rejected if the requested maximum packet size is larger than the MTU of the link. Considering this, the following should be noted when choosing the value for *TB_Tspec_M* field in PATH messages:

- You can set the value higher than the MTU because *TB_Tspec_M* in a `PATH` message should not be used to limit the size of the application packets. Packets should be limited by the *TB_Tspec_M* parameter included in a reservation message flowspec.
- Don't set the value too low. If the *TB_Tspec_M* parameter has a value less than the MTU of the global link, the full capacity of the link will not be utilized. This value should not be lower than 500.
- To benefit from the full capacity available from the global MTU, you can set *TB_Tspec_M* to a relatively high value, for example 32000.

TB_Tspec_p field

TB_Tspec_p is the Peak Rate (in bps of IP datagrams). This is the maximum rate at which the source may inject bursts of traffic onto the network. This value is measured in bps of IP datagrams and must be greater than or equal to the token bucket rate.

If the sender application is a regular stream sender on an efficient network, using a buffer of size *TB_Tspec_M* should be sufficient for the receiver application to process the received data. However, the minimum buffer size cannot be guaranteed to prevent data bursts. Using *TB_Tspec_p* can help to calculate the buffer length. For example, if *T* is the expected time period between the reception of two data packets, you can use this information to set the buffer size to `(TB_Tspec_M + (T * TB_Tspec_p))`.

Waiting for RSVP Messages

This procedure is the same as for receiver applications. For more information see “Waiting for RSVP Messages” on page 48.

Retrieving and Processing RSVP Messages

This procedure is the same as for receiver applications. For more information see “Retrieving and Processing RSVP Messages” on page 50.

Closing an RSVP Session

This procedure is the same as for receiver applications. For more information see “Closing an RSVP Session” on page 63.

Implementing Solstice Bandwidth Reservation Protocol in a Receiver Application

Solstice Bandwidth Reservation Protocol can be implemented into a data receiver application by adding some simple calls to RAPI functions. There are two important considerations when introducing these calls into the source code:

- Deciding where to place the calls
- Using the correct parameters to be passed to the RAPI functions

This section contains typical example configurations for establishing and maintaining an RSVP session for a receiver application. It identifies the relevant functions and illustrates the correct places for them. Technical design options are emphasized where more than one solution is available.

Opening an RSVP Session

This procedure is the same as for sender applications. For more information see "Opening an RSVP Session" on page 38.

Waiting for RSVP Messages

Although this section describes the process for a data receiver, it also applies to sender applications. It is not necessary for an application to wait for reservation confirmation before starting playback. Whereas receiver applications must wait for `PATH` messages before they can start sending `RESV` messages, sender applications can generate `PATH` messages anytime without having to wait for the arrival of an RSVP message. `PATH` messages are packets sent by the data senders which describe the dataflow being transmitted to the receiver applications. The receiver applications use the `PATH` message information to request suitable bandwidth reservations.

However, the application writer may decide that it is acceptable for the receiver to accept an incoming data stream without a reservation guarantee.

Sender applications are generally concerned about the *type* of a received RSVP message, for example the reception of a `PATH` error or a reservation event. However, receiver applications are also concerned with the *content* of the received RSVP message. This is because the contents of an RSVP reservation message are generated by receiver applications using the contents of a received `PATH` message.

Once the RSVP session has been opened, the data receiver application waits for the RSVP `PATH` messages.

When an application opens an RSVP session, a new file descriptor is allocated on the process file descriptor's table (See also "rapi_getfd()" on page 17). For the receiver application to identify and retrieve the content of a `PATH` message, it must wait for an event related to this file descriptor. This is achieved using the following code:

```
/* 1.- get the RSVP file descriptor using rapi_getfd function */
rapi_fd = rapi_getfd(Sid);

/* 2.- if rapi_getfd suceeded, include the RSVP file descriptor
in a file descriptor set using the FD_SET function */
if (rapi_fd == -1) {
   perror("rapi_getfd ERROR");
   closing_rsvp();
}
else {
   FD_SET(rapi_fd, &fdset);
}

/* 3.- make a call to the select function to wait for the RSVP file
descriptor to be ready for reading */
(void) select(rapi_fd + 1, &fdset,
         (fd_set*)NULL, (fd_set*)NULL, (struct timeval *)NULL);

/* 4.- if it was the RSVP file descriptor that unblocked the
select call, retrieve the content of the RSVP arriving message
calling the rapi_dispatch function */

if (FD_ISSET(rapi_fd, &fdset)){
   RSVP_error = rapi_dispatch();
   if (RSVP_error != 0) {
      perror("rapi_dispatch ERROR");
      closing_rsvp();
   }
}
```

As illustrated, the application waits for RSVP messages by calling the `select()` function. When an RSVP message is received, it can be retrieved by calling the `rapi_dispatch()` function. Retrieval and processing of the received RSVP message is described in the next section.

Retrieving and Processing RSVP Messages

The content of a received RSVP message is accessible to the application using the `rapi_dispatch()` function. A call to `rapi_dispatch()` executes the callback function that was specified when the RSVP session was opened.

When the callback function is executed, it passes all the template parameters that were read by the RSVP file descriptor to the main program. Considering this, you must design the callback function so that the receiver is able to generate an RSVP reservation message from the information supplied by the parameters of the callback function.

Below are two methods for constructing the callback function:

Method 1

This method is recommended for receiver applications. This is because reservation messages are built from the contents of a `PATH` message and most of the parameters passed by the callback function need to be accessed.

The example below shows a receiver application waiting for the arrival of a sender `PATH` message. This `PATH` message carries Tspec information from a host sender, hence the test for *filterspec_spec_no==1* in the sample code. To allow for the case where the flowspec, filterspec and Adspec lists contain more than one entry, more general code is required.

Once the `PATH` message has been received, it can generate the corresponding *fixed filter* style reservation message. It illustrates several possible reactions to the reception of an RSVP message.

`RAPI_PATH_TEAR_EVENT` is not currently implemented so this event is indicated by `RAPI_PATH_EVENT` with *filter_spec_no*==0.

The example is simplified and assumes the existence of the following functions: `closing_rsvp()`, `allocate_memory_for_a_CL_flowspec()`, `allocate_memory_for_a_G_flowspec()`. These peripheral functions should be written independently and are not included here.

```
int
rsvp_callback(
    rapi_sid_tsid,       /* Which sid generated event */
    rapi_eventinfo_teventtype, /* Event type */
    rapi_styleid_t styleID,  /* Style ID */
```

```
    int     errcode,  /* Error code */
    int     errvalue, /* Error value */
    struct sockaddr_inerrnode,  /* Error node address */
    u_char  errflags, /* Error flags */
    int     filter_spec_no,     /* # of filter specs in list */
    rapi_filter_t*filter_spec_list,  /* Filter spec/sender templ list */
    int     tspec_no,    /* # of flowspecs/Tspecs */
    rapi_tspec_t*tspec_list,        /* Flowspec/Tspec list */
    int     ad_spec_no,    /* # of adspecs */
    rapi_adspec_t   *ad_spec_list,    /* Adspec list */
    void    *args    /* Client supplied arg */ )
{
rapi_flowspec_t *    Flow = NULL;

switch(eventype){
/* if the RSVP received message was a PATH or a PATH TEAR... */
case RAPI_PATH_EVENT:
   Receiver_host.sin_addr.s_addr = INADDR_ANY;

   if (filter_spec_no > 1)
    printf("Something goes wrong: %d filterSpecs have been received!\n",
filter_spec_no);

   if (filter_spec_no == 1) {
     /* Suppose there's a global variable that indicates the QoS control
service to be used for reservation: Guaranteed when "Guaranteed == TRUE"
and Controlled Load when "Guaranteed == FALSE". Memory allocation and
flowspec initialization are going to be different depending on the value
of this variable */
      switch(Guaranteed){
      case FALSE:
     allocate_memory_for_a_CL_flowspec(Flow);
     ret = rapi_IS_fspec_CL_init(tspec_list,adspec_list, Flow);
     if (ret) {
        printf("%s\n", rapi_errlist[ret]);
        exit(1);
     }
     break;

      case TRUE:
         allocate_memory_for_a_G_flowspec(Flow);
     ret = rapi_IS_fspec_G_init(tspec_list,adspec_list,Flow, (float32_t)
0, (uint32_t) 0);
     if (ret) {
        printf("%s\n", rapi_errlist[ret]);
```

```
        exit(1);
     }
     break;
       }
       /* Now is time to make the reservation. See “sending reservation
messages” */
       RSVP_error = rapi_reserve(Sid, 0x20, &Receiver_host,
RAPI_RSTYLE_FIXED,
            NULL, NULL, 1, filter_spec_list, 1, Flow);

       if (RSVP_error == -1) {
     perror("rapi_reserve ERROR");
     closing_rsvp();
       }
   }

   /* The next piece substitutes RAPI_PATH_TEAR_EVENT, which is not
supported */
   if (!filter_spec_no) {
      RSVP_error = rapi_reserve(Sid, 0, &Receiver_host,
RAPI_RSTYLE_FIXED,
            NULL, NULL, 0, NULL, 0, NULL);

       if (RSVP_error == -1) {
     perror("rapi_reserve_tear ERROR");
     closing_rsvp();
       }
   }
   break;

/* if the RSVP received message was a RESERVATION ERROR... */
case RAPI_RESV_ERROR:
   printf("Error received\n");
   closing_rsvp();
   break;

/* if the RSVP received message was a RESERVATION CONFIRMATION... */
case RAPI_RESV_CONFIRM:
   printf("Reservation Confirmation Received!:\n");
   break;

default:
   break;
}
```

Method 2

This method is recommended for sender applications or applications which only require the *type* of RSVP event that has been produced, not the specific *content* of the RSVP message.

This method simplifies the structure of the callback function but increases the processing required at the main thread level. Information is passed from the callback function to the main program through the *args* parameter. Below is an example callback function:

```
int
rsvp_callback(
    rapi_sid_tsid,      /* Which sid generated event */
    rapi_eventinfo_teventype, /* Event type */
    rapi_styleid_t styleID,  /* Style ID */
    int     errcode,  /* Error code */
    int     errvalue, /* Error value */
    struct sockaddr_inerrnode,  /* Error node address */
    u_char  errflags, /* Error flags */
    int     filter_spec_no,     /* # of filter specs in list */
    rapi_filter_t*filter_spec_list,  /* Filter spec/sender templ
list */
    int     tspec_no,    /* # of flowspecs/Tspecs */
    rapi_tspec_t*tspec_list,        /* Flowspec/Tspec list */
    int     ad_spec_no,    /* # of adspecs */
    rapi_adspec_t   *ad_spec_list,    /* Adspec list */
    void    *args    /* Client supplied arg */ )
{
rapi_eventinfo_t * temp_pointer;

temp_pointer = (rapi_eventinfo_t *) args;

/* Passing the eventype argument of the callback function to the
main program through the "args" argument */

*temp_pointer = eventype;
}

The code for "opening an RSVP session" at the main program...:

/* Opening an RSVP session: a pointer to the callback return
argument must be specified */
```

```
Sid = rapi_session(&(Xmit.addrs.data), IPPROTO_UDP, 0,
rsvp_callback,
                   (void *) &RSVP_arg, &RSVP_error);

The code for "RSVP waiting" at the main program...:

rapi_fd = rapi_getfd(Sid);

if (rapi_fd == -1) {
   perror("rapi_getfd ERROR");
   closing_rsvp();
}

(void) select(rapi_fd + 1, &fdset,
         (fd_set*)NULL, (fd_set*)NULL, (struct timeval *)NULL);

if (FD_ISSET(rapi_fd, &fdset)){
   RSVP_error = rapi_dispatch();
   if (RSVP_error == 0) {
     /* rapi_dispatch suceeded: the callback function has passed
the callback return argument to the main program.The receiver
RSVP answers can now be processed with the callback return
argument */
      switch(RSVP_arg) {
      case RAPI_RESV_EVENT:
      .
      .
      .
      .case RAPI_PATH_ERROR:
      }
   }
   else {
      perror("rapi_dispatch ERROR");
      closing_rsvp();

   }
}
```

In the example above the callback return argument has been used to pass just one callback argument (the event type) to the main program. However, it is possible to pass several arguments by implementing a structure that would contain all the relevant parameters.

Sending Reservation Messages

This section describes some of the processing required to generate reservation messages. The following is a review of part of the code shown in the callback function example for Method 1, concerning the reservation message building process:

```
/* 1 */
case RAPI_PATH_EVENT:

/* 2 */
   if (filter_spec_no == 1) {
      switch(Guaranteed){
      case FALSE:
         allocate_memory_for_a_CL_flowspec(Flow);
         ret = rapi_IS_fspec_CL_init(tspec_list, adspec_list, Flow);
         if (ret) {
           printf("%s\n", rapi_errlist[ret]);
           exit(1);
         }
         break;

      case TRUE:
         allocate_memory_for_a_G_flowspec(Flow);
         ret = rapi_IS_fspec_G_init(tspec_list, adspec_list, Flow,
(float32_t) 0,(uint32_t) 0);

         if (ret) {
         printf("%s\n", rapi_errlist[ret]);
         exit(1);
         }
         break;
      }

/* 3 */
      /* Now to make the reservation */
      Receiver_host.sin_addr.s_addr = INADDR_ANY;
      RSVP_error = rapi_reserve(Sid, 0x20, &Receiver_host,
RAPI_RSTYLE_FIXED,NULL, NULL, 1, filter_spec_list, 1, Flow);
```

The above example code illustrates three independent stages:

1. The identification of an RSVP `PATH` message arrival.

2. The initialization of the flowspec parameters (for more information see "Assigning a Value to Parameter 10 in rapi_reserve() (rapi_flowspec_t)" on page 57).

3. The request for reservation message generation.

As with PATH messages, reservation messages are sent by the rsvpd daemon at regular intervals, depending on the load. This is usually every 30 seconds. A single call to the rapi_reserve() function requests the daemon to begin periodic RESV message generation.

Below are some recommendations for assigning values to rapi_reserve() parameters.

An example rapi_reserve() function:

```
rapi_reserve(Sid, flags, &Receiver_host, RAPI_RSTYLE_FIXED,
NULL, NULL, 1, filter_spec_list, 1, Flow);
```

Parameter 1

This parameter is a session identifier returned by the rapi_session() function. This indicates that a successful call to rapi_session() has been made.

Parameter 2

This is the flags parameter which should be set to 0X20 if the receiver requires a reservation confirmation message. If this is not required, this value should be set to 0.

Parameter 3

This parameter is a structure of type sockaddr_in and contains *the data receiver address, not the multicast destination address*. The multicast destination address has already been passed to the rsvpd daemon by the rapi_session() function. Entering the multicast destination address in *Parameter 3* of rapi_sender() is a common and serious error.

When multicast dataflow is to be received through the default network interface, *Parameter 3* can be initialized by setting:

```
Receiver_host.sin_addr.s_addr = INADDR_ANY;
```

This must be done before calling the `rapi_reserve()` function.

Parameter 8

This is a pointer to a structure of type `rapi_filter_t`. The value for this parameter should be the same as the `filter_spec_list` parameter that was received in the `PATH` message. This is available from one of the callback function parameters.

Parameter 10

This is a pointer to `rapi_flowspec_t`, and assigning it a value is a complex procedure. It is generated from the `flow_spec_list` parameter that was received in the `PATH` message and is available from one of the callback function parameters. However, they do not match exactly as `flow_spec_list` contains a list of sender Tspecs. These must be converted into receiver flowspecs, so some of the contents of the received structure will have to be updated when requesting a reservation.

It is strongly recommended that you use the `rapi_IS_fspec_G_init()` and `rapi_IS_fspec_CL_init()` functions, which enable the automatic initialization and update of Integrated Services flowspecs. It is not necessary to read the section below if you are going to use these functions.

Manual initialization of this parameter is fully covered in the next section.

Assigning a Value to Parameter 10 in `rapi_reserve()` *(*`rapi_flowspec_t`*)*

Parameter 10 in `rapi_reserve()` is a structure of type `rapi_flowspec_t`. There are many possible implementations of its union components (`qos_flowspec_t`, `qos_flowspecx_t`, `qos_tspecx_t`, `IS_specbody_t` and `IS_tspbody_t`) which makes initialization a complex task.

Data sender applications generate Tspecs to create `PATH` messages and data receiver applications generate flowspecs to create `RESV` messages. As illustrated in the definitions below, `rapi_tspec_t` and `rapi_flowspec_t` are of the same structural definition.

```
typedef struct {
int     len;/* actual length in bytes */
    int     form;/* flowspec format (see enum above) */
    union {
    qos_flowspec_tqos;/* Legacy format */
    qos_flowspecx_t qosx;/* Extended legacy flowspec */
    qos_tspecx_t qosxt;/* Extended legacy Tspec */
    IS_specbody_t IS;/* Int-serv format flowspec */
    IS_tspbody_t ISt;/* Int-serv format Tspec */
    } specbody_u;
} rapi_flowspec_t;

typedef rapi_flowspec_t rapi_tspec_t;/* Same struct definition
*/
```

`qos_flowspec_t`

This is the data type used to fill in the legacy formatted flowspec and is not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`qos_tspecx_t` *and* `qos_flowspecx_t`

These are the data types used to fill in the extended legacy formatted Tspec and flowspec respectively. They are an extension to `qos_flowspec_t` but are not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`IS_tspbody_t` *and* `IS_specbody_t`

These are the data types used to fill in Integrated Services formatted Tspec and flowspec respectively. These are the standard formats chosen by the Integrated Services Workgroup and are the most suitable choices for `specbody_u` union. It is recommended that you use `IS_tspbody_t` when you want to send `PATH` messages in data sender applications and `IS_specbody_t` when you want to make `RESV` messages in data receiver applications.

You must use `IS_tspbody_t` to complete *Parameter 10* in `rapi_reserve()`.

IS_specbody_t offers a choice between two unions:

```
typedef struct {

    IS_main_hdr_tspec_mh;
    union {
    CL_flowspec_t CL_spec;/* Controlled-Load service */
    Guar_flowspec_t G_spec;/* Guaranteed service*/
    } spec_u;
} IS_specbody_t;
```

It is recommended that you use Guar_flowspec_t when asking for a Guaranteed Quality of Service control service for your reservations, and CL_flowspec_t when asking for a Controlled Load Quality of Service control service. Below is an example procedure which initializes rapi_flowspec_t using the Guar_flowspec_t (Guaranteed Quality of Service) control service.

```
rapi_tspec_t    *old_Tspec;
rapi_flowspec_t *new_Flowspec;

Guar_flowspec_t *Continue_new;
gen_Tspec_t *Continue_old;

Continue_new = &(new_Flowspec-
>specbody_u.IS.spec_u.G_spec);
Continue_old = &(old_Tspec-
>specbody_u.ISt.tspec_u.gen_stspec);

new_Flowspec->len = sizeof(int)*2 + sizeof(IS_main_hdr_t)
+ sizeof(Guar_flowspec_t);
new_Flowspec->form = RAPI_FLOWSTYPE_Intserv;
new_Flowspec->specbody_u.IS.spec_mh.ismh_version =
old_Tspec->specbody_u.ISt.st_mh.ismh_version;
new_Flowspec->specbody_u.IS.spec_mh.ismh_unused  =
old_Tspec->specbody_u.ISt.st_mh.ismh_unused;
new_Flowspec->specbody_u.IS.spec_mh.ismh_len32b  =
(sizeof(Guar_flowspec_t)+3)/4;
Continue_new->Guar_serv_hdr.issh_service   =
GUARANTEED_SERV;
Continue_new->Guar_serv_hdr.issh_break=0;
Continue_new->Guar_serv_hdr.issh_len32b    =
(sizeof(Guar_flowspec_t)-sizeof(IS_serv_hdr_t)+3)/4;;
Continue_new->Guar_Tspec_hdr.isph_parm_num =
IS_WKP_TB_TSPEC;
```

```
Continue_new->Guar_Tspec_hdr.isph_flags    = 0;
Continue_new->Guar_Tspec_hdr.isph_len32b   =
0(sizeof(Guar_flowspec_t)-sizeof)IS_serv_hdr_t)-
sizeof(IS_parm_hdr_t)+3)/4;
Continue_new->Guar_Tspec_parms.TB_Tspec_r  =
Continue_old->gen_Tspec_parms.TB_Tspec_r;
Continue_new->Guar_Tspec_parms.TB_Tspec_b  =
Continue_old->gen_Tspec_parms.TB_Tspec_b;
Continue_new->Guar_Tspec_parms.TB_Tspec_p  =
Continue_old->gen_Tspec_parms.TB_Tspec_p;
Continue_new->Guar_Tspec_parms.TB_Tspec_m  =
Continue_old->gen_Tspec_parms.TB_Tspec_m;
Continue_new->Guar_Tspec_parms.TB_Tspec_M  = updated_MTU;
Continue_new->Guar_Rspec_hdr.isph_parm_num =
IS_GUAR_RSPEC;
Continue_new->Guar_Rspec_hdr.isph_flags    = 0;
Continue_new->Guar_Rspec.Guar_R            =
guaranteed_rate;
Continue_new->Guar_Rspec.Guar_S            = slack_term;
```

For the example below, `CL_flowspec_t` (Controlled Load) control service is used:

```
rapi_tspec_t    *old_Tspec;
rapi_flowspec_t *new_Flowspec;

CL_flowspec_t   *Continue_new;
gen_Tspec_t     *Continue_old;

Continue_new = &(new_Flowspec-
>specbody_u.IS.spec_u.CL_spec);
Continue_old = &(old_Tspec-
>specbody_u.ISt.tspec_u.gen_stspec);

new_Flowspec->len = sizeof(int)*2 + sizeof(IS_main_hdr_t)
+ sizeof(CL_Tspec_t);
new_Flowspec->form = RAPI_FLOWSTYPE_Intserv;
new_Flowspec->specbody_u.IS.spec_mh.ismh_version =
old_Tspec->specbody_u.ISt.st_mh.ismh_version;
new_Flowspec->specbody_u.IS.spec_mh.ismh_unused  =
old_Tspec->specbody_u.ISt.st_mh.ismh_unused;
new_Flowspec->specbody_u.IS.spec_mh.ismh_len32b  =
(sizeof(CL_flowspec_t)+3) /4;
```

```
Continue_new->CL_spec_serv_hdr.issh_service  =
CONTROLLED_LOAD_SERV;
Continue_new->CL_spec_serv_hdr.issh_break=0;
Continue_new->CL_spec_serv_hdr.issh_len32b   =
sizeof(IS_parm_hdr_t)+(sizeof(TB_Tsp_parms_t)+3) /4;
Continue_new->CL_spec_parm_hdr.isph_parm_num =
IS_WKP_TB_TSPEC;
Continue_new->CL_spec_parm_hdr.isph_flags    = 0;
Continue_new->CL_spec_parm_hdr.isph_len32b   =
(sizeof(TB_Tsp_parms_t)+3) /4
Continue_new->CL_spec_parms.TB_Tspec_r       =
Continue_old->gen_Tspec_parms.TB_Tspec_r;
Continue_new->CL_spec_parms.TB_Tspec_b       =
Continue_old->gen_Tspec_parms.TB_Tspec_b;
Continue_new->CL_spec_parms.TB_Tspec_p       =
Continue_old->gen_Tspec_parms.TB_Tspec_p;
Continue_new->CL_spec_parms.TB_Tspec_m       =
Continue_old->gen_Tspec_parms.TB_Tspec_m;
Continue_new->CL_spec_parms.TB_Tspec_M      = updated_MTU;
```

len **field**

The len field contains the exact number of bytes of the flowspec or Tspec object. This value is format dependant and so determined by the selected format, for example legacy, extended legacy or Integrated Services.

`rapi_flowspec_t` objects do not have padding bytes so the lengths may be calculated using the size of the function.

Specifying `rapi_flowspec_t` structures for data receiver applications is more complex. This is because some templates contain padding bytes so you must know the exact structure of the RSVP templates when transmitting them.

form **field**

Three different format fields are supported: legacy, extended legacy and Integrated Services.

`qos_flowspec_t`

This is the data type used to fill in the legacy formatted flowspec and is not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`qos_tspecx_t` *and* `qos_flowspecx_t`

These are the data types used to fill in the extended legacy formatted Tspec and flowspec respectively. They are an extension to `qos_flowspec_t` but are not recommended for a fully interoperable application. It is provided for backwards compatibility with early stage applications.

`IS_tspbody_t` *and* `IS_specbody_t`

These are the data types used to fill in Integrated Services formatted Tspec and flowspec respectively. These are the standard formats chosen by the Integrated Services Workgroup and are the most suitable choices for the `specbody_u` union. It is recommended that you use `IS_tspbody_t` when you want to send `PATH` messages in data sender applications and `IS_specbody_t` when you want to make `RESV` messages in data receiver applications.

ismh_len32b, *issh_len32b* and *isph_len32b* fields

These fields are format dependent. They indicate the number of remaining 32-bit words from the point they are placed to the end of the `rapi_tspec_t` object.

issh_service field

There are two Quality of Service choices available for *issh_service*:

1. `GUARANTEED_SERV`. The Quality of Service is specified by a Tspec (using the token bucket fields), and in addition, a *slack rate* and a *guaranteed rate*.

2. `CONTROLLED_LOAD_SERV`. The Quality of Service is specified by a Tspec (using the token bucket fields) only.

To verify that the desired service is available before completing this field, check that the received Adspec in the flows `PATH` message corresponds to the desired service. This is done by searching the Adspec for a fragment which corresponds to the required service.

If there is a mismatch, an error will be generated by the first RSVP-enabled router in the `PATH` to receive the reservation message.

Token bucket parameter fields

All the token bucket fields can be assigned values in the received Tspec except the TB_Tspec_M field. The TB_Tspec_M field must be updated as packet fragmentation is not supported for IS control services.

For example, a sender application proposes to send 32000 byte packets (TB_Tspec_M=32000 in the PATH message) but the PATH_MTU field contained in the Adspec of the received PATH message states that the minimum MTU of the link is 500 bytes. In this case, the receiver would have to fill the TB_Tspec_M in the flowspec with a value that could never exceed 500.

The TB_Tspec_M in a PATH message carries the request of the sender and is informational only. The TB_Tspec_M in a RESV message is used to control the packet length for a reservation. When the sender receives the TB_Tspec_M contained in the reservation message, it will be forced to limit the value to the size of the packets to be sent.

TB_Tspec_M in a flowspec should be assigned the minimum of:

- The TB_Tspec_M field in the Tspec
- The PATH_MTU in the Adspec

of the received PATH message.

rapi_IS_tspec_g_init() and rapi_IS_tspec_g_init() will automatically do this whenever the received Adspec is passed as a parameter to these functions.

Closing an RSVP Session

Closing an RSVP session is simple. A call to the rapi_release() function will end the session identified by the Session Identifier (*Sid*) passed as an argument to the function.

rapi_release() returns a value of 0 when it succeeds. A non-zero value will be returned if an error occurs whilst removing the session. This value will then contain an error code that can be referenced in the 'rapi_errlist' string array in the rapi_err.h include file.

Closing an RSVP session is illustrated in the example below:

```
RSVP_error = rapi_release(Sid);
if ( RSVP_error )  /* An error has been produced */
printf("rapi_session returns an error: %s \n",
rapi_errlist[RSVP_error]);
```

Index