INTERNET-DRAFT
Expiration: February 1999
File: draft-lindell-rsvp-scrapi-00.txt

Bob Lindell
ISI

SCRAPI - A Simple ‘‘Bare Bones’’ API for RSVP

Version 1

August 7, 1998

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as ‘‘work in progress.’’

To view the entire list of current Internet-Drafts, please check the ‘‘1id-abstracts.txt’’ listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), ftp.nordu.net (Northern Europe), ftp.nis.garr.it (Southern Europe), munnari.oz.au (Pacific Rim), ftp.ietf.org (US East Coast), or ftp.isi.edu (US West Coast).

Abstract

This document describes SCRAPI, a simple ‘‘bare bones’’ API for RSVP. The goal of this API is to produce an interface which simplifies the augmentation of applications with RSVP support.

1. Introduction

This document describes SCRAPI, a simple ‘‘Bare Bones’’ API for RSVP [1]. The goal of this API is produce an interface which simplifies the augmentation of applications with RSVP support. The main features of SCRAPI are:

- Allow the addition of RSVP support to applications by adding a few lines of code.
- Provide a portable interface which can be used with any vendor’s RAPI implementation.
- Avoid the introduction of RSVP specific data types and definitions.
- Support IPv4 and IPv6 in a transparent manner.

SCRAPI is layered on top of RAPI [2], an existing RSVP API, to provide portability across vendor implementations. Currently, SCRAPI has been tested only with the ISI implementation of RAPI.

There are three main routines included in this API, one which is used by the sender side, one to be used at the receiving end, and a routine to finalize the entire API at the end of execution. This simple API should be easy to insert into networking applications which require RSVP support.

The remaining routines in the API are utility functions to ease the development of an interface which supports IPv4 and IPv6. The objective is to provide enough functionality so that applications would not need to code explicitly for either IPv4 or IPv6, or use messy compilation conditionals to develop an interface to support both address families. As an example, a single data type for addresses is provided that is large enough to hold addresses from either address family. In addition, parsing an address string or performing a host name resolution for both address families is provided.

To provide simplicity, event upcalls to the application do not exist in SCRAPI. For this reason, it would be difficult to use SCRAPI for applications which negotiate QoS with the network. Applications requiring this type of functionality should use the standard RAPI interface.

2. Functional Description

Applications which use SCRAPI get a simplied service model. The average bandwidth specified by the sender is currently translated into the integrated services controlled load [4] or guaranteed service [3] with the given average bandwidth. The peak bandwidth and token bucket depth are set to twice the average bandwidth. Minimum policed unit and maximum packet size are set to 64 bytes and the largest MTU of all IP interfaces on the host.

Sender and receiver side API calls can block, if requested, until a reservation request has completed. The notion of completion can be difficult to define for multicast flows. We will define completion in the content of SCRAPI calls to refer to either partial or full completion of the reservation request.

A sender sources PATH messages using the Tspec described above. If blocking is requested with a non-zero timeout value, the sender blocks until the receipt of a single RESV event from any sender. If the specified flow is unicast, the call blocks until the reservation has completed. If the flow is multicast, then at least one receiver has a reservation in place when the call unblocks.

A receiver waits for a PATH event from the sender, and if requested, makes a reservation in response using the sender's Tspec and Adspec for guaranteed service. For guaranteed service, the slack term is set to zero. If blocking is requested, the receiver blocks until the receipt of a CONFIRM event. For a multicast flow, a CONFIRM event can be a unreliable indication that a reservation has been successful.

If any RSVP errors occur during a blocking sender or receiver API call, the call will unblock and return an error code.

A state diagram of the SCRAPI reservation API, for a given data flow, is shown in Figure 1. A data flow, or RSVP session, is defined as a unique destination address, port and protocol number.

SCRAPI provides a simple error status reporting on a per flow basis. Status can be in a ‘‘red’’, ‘‘yellow’’, or ‘‘green’’ state. The red state indicates that either the flow does not exist or is currently in an error state. Yellow state indicates that the reservation requests are pending, whereas green indicates that at least one request has completed.

In the next section, the SCRAPI application programming interface is defined. In subsequent sections, usage examples are offered as templates for programmers who are attempting to embed SCRAPI calls into existing applications.

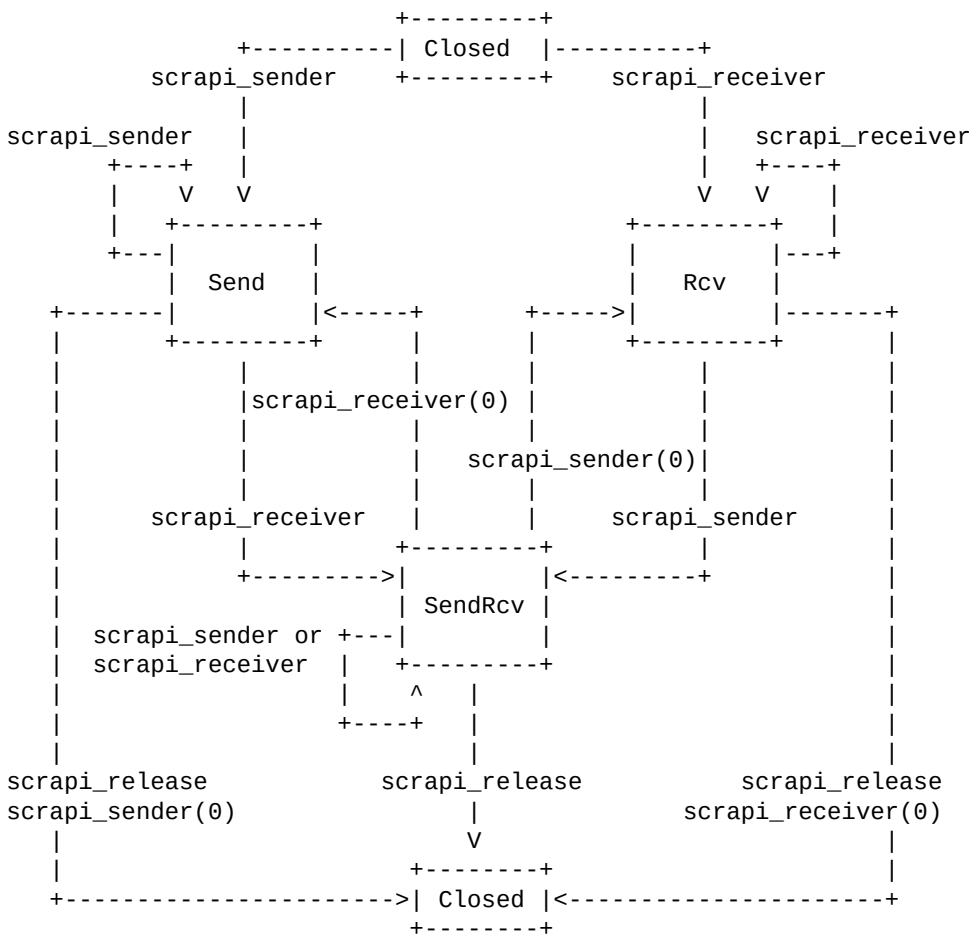

Figure 1: SCRAPI API State Diagram

3. Application Programming Interface Definition

3.1. Reservation API Description

Function Name: **scrapi_sender**

Syntax:
```
int scrapi_sender(
        const struct sockaddr *destination,
        int protocol,
        const struct sockaddr *source,
        scrapi_service service,
        double bw,
        int ttl,
        unsigned long msecs
);
```

Description: SCRAPI call for a data sender. The *destination* address and *protocol* number of the data flow are supplied as the first two arguments. The *source* address of the data flow can be specified, or set to **NULL** to choose the system default. The *service* parameter specifies the service, currently either Controlled Load or Guaranteed. The *bw* parameter is the average bandwidth of the flow in bytes/sec. The *ttl* of the packets can be specified or set to 0 to choose the system default. If *msecs* is greater than 0, the call to scrapi_sender blocks *msecs* milliseconds to receive a reservation event from at least one recipient. The call will also unblock prematurely if any errors are detected during this period. This function can be called repeatedly by an application to modify any parameters associated with this data flow (same destination and protocol values). A value of 0 for the *bw* parameter unregisters the data flow.

Return Values: **TRUE** if successful, **FALSE** otherwise. Unsuccessful operations set scrapi_errno to an appropriate error code.

Also See: scrapi_errno, scrapi_get_status

Function Name: **scrapi_receiver**

Syntax:
```
int scrapi_receiver(
        const struct sockaddr *destination,
        int protocol,
        const struct sockaddr *source,
        int reserve,
        scrapi_style style,
        unsigned long msecs
);
```

Description: SCRAPI call for a data receiver. The *destination* address and *protocol* number of the data flow are supplied as the first two arguments. The *source* address of the data flow can be specified or can be set to **NULL** to choose any source. In addition, the *source* address can be specified with a port number of 0 to match a source address regardless of port number. The *reserve* parameter should be set to **TRUE** or **FALSE** to turn on and off a reservation for that data flow respectively. The *style* parameter specifies whether the reservation is shared among multiple senders. If *msecs* is greater than 0, the call to scrapi_receiver blocks *msecs* milliseconds to receive a reservation confirmation event. The call will also unblock prematurely if any errors are detected during this period. This function can be called repeatedly by an application to modify any parameters associated with this data flow (same destination and protocol values).

Return Values: **TRUE** if successful, **FALSE** otherwise. Unsuccessful operations set scrapi_errno to an appropriate error code.

Also See: scrapi_errno, scrapi_get_status

Function Name: **scrapi_close**

Syntax: int scrapi_close(

const struct sockaddr *destination,
int protocol

);

Description: SCRAPI call to close either a sender or receiver flow. If the *destination* address is set to **NULL**, all flows will be closed and the *protocol* parameter value will be ignored.

Return Values: **TRUE** if successful, **FALSE** otherwise. Unsuccessful operations set scrapi_errno to an appropriate error code.

Enum Name: **scrapi_service**

1. scrapi_service_cl
2. scrapi_service_gs

Description: Enumerated types for specificing the desired service. Currently, Integrated Services Controlled Load and Guaranteed are supported.

Enum Name: **scrapi_style**

1. scrapi_style_shared
2. scrapi_style_distinct

Description: Enumerated types for specificing the reservation style. Currently, shared (wildcard) and distinct styles are supported.

Enum Name: **scrapi_status**

1. scrapi_status_red
2. scrapi_status_yellow
3. scrapi_status_green

Description: Enumerated types for status condition of a flow. The meaning of these values is described in the scrapi_status() function description.

3.2. Asynchronous Event Loop API Description

Function Name: **scrapi_getfd**

Syntax:
```
int scrapi_getfd(
        const struct sockaddr *destination,
        int protocol
);
```

Description: SCRAPI call to get the API file descriptor to use in a subsequent select() call.

Return Values: A file descriptor value if successful, -1 otherwise.

Function Name: **scrapi_dispatch**

Syntax:
```
int scrapi_dispatch();
```

Description: SCRAPI call to poll the API for new events.

Return Values: **TRUE** if successful, **FALSE** if RSVP support is no longer available.

3.3. Error Handling API Description

Function Name: **scrapi_get_status**

Syntax:
```
scrapi_status scrapi_get_status(
        const struct sockaddr *destination,
        int protocol
);
```

Description: SCRAPI call to get the status of a flow. If the status is red, either the flow was never defined or the flow is currently in an error state. A yellow status indicates that the flow is valid but no reservation operation(s) at the sender or receiver end has completed successfully. Once a single operation on either the sender or receiver end has completed successfully, the flow has a green status.

Return Values: scrapi_stat_red, scrapi_stat_yellow, or scrapi_stat_green.

Variable Name: **scrapi_errno**

Syntax: extern int scrapi_errno;

Description: Set to the return value of the last RAPI call made within the SCRAPI library.

Also See: scrapi_errlist

Function Name: **scrapi_perror**

Syntax:
```
void scrapi_perror(
        const char *string
);
```

Description: SCRAPI call to print an error message analogous to the perror() library call.

Function Name: **scrapi_errlist**

Syntax:
```
const char * scrapi_errlist(
        int errno
);
```

Description: SCRAPI call to get an error message string for a given errno.

Return Values: An error message string.

Function Name: **scrapi_stderr**

Syntax:
```
void scrapi_stderr(
        FILE *file
);
```

Description: Set the file pointer to be used by the SCRAPI library for standard error. If it is set to **NULL**, no messages are printed. The default value is **stderr**.

Function Name: **scrapi_debug**

Syntax:
```
void scrapi_debug(
        FILE *file
);
```

Description: Set the file pointer to be used by the SCRAPI library for debugging information. If it is set to **NULL**, no messages are printed. Debug messages include the logging of all asynchronous RSVP events. The default value is **NULL**.

3.4. Address Manipulation API Description

Macro Name: **SOCKADDR**

Description: A struct sockaddr data type that is large enough for both IPv4 and IPv6 addresses.

Function Name: **scrapi_sockaddr_multicast**

Syntax:
```
int scrapi_sockaddr_multicast(
        const struct sockaddr *address
);
```

Description: Determine if an address is multicast or unicast.

Return Values: **TRUE** if multicast, **FALSE** otherwise.

Function Name: **scrapi_sockaddr_parse**

Syntax:
```
int scrapi_sockaddr_parse(
        struct sockaddr *address
        const char *name,
        unsigned short port
);
```

Description: Perform a host name lookup or parse an address and initialize a struct sockaddr structure.

Return Values: **TRUE** if parsed, **FALSE** otherwise.

Function Name: **scrapi_sockaddr_print**

Syntax:
```
const char * scrapi_sockaddr_print(
        const struct sockaddr *address
);
```

Description: Pretty print an address.

Return Values: A valid string if printable, **NULL** otherwise.

Function Name: **scrapi_sockaddr_get_port**

Syntax:
```
unsigned short scrapi_sockaddr_get_port(
        const struct sockaddr *address
);
```

Description: Get the port number.

Return Values: The port number if successful, 0 otherwise.

Function Name: **scrapi_sockaddr_set_port**

Syntax:
```
int scrapi_sockaddr_set_port(
        struct sockaddr *address,
        unsigned short port
);
```

Description: Set the port number.

Return Values: **TRUE** if successful, **FALSE** otherwise.

Function Name: **scrapi_sockaddr_any**

Syntax:
```
int scrapi_sockaddr_any(
        struct sockaddr *address,
        int family
);
```

Description: Initialize a struct sockaddr to the wildcard address and port number for the given address family.

Return Values: **TRUE** if successful, **FALSE** otherwise.

4. Application Code Templates

This section provides examples, presented as code templates, to aid programmers in augmenting networking applications with SCRAPI calls. One example contains two simplex applications which attempt to wait for a reservation to be put in place before sending data on the network. The other example is a full duplex multimedia type application which sends and receives data without waiting for completion of the reservation.

4.1. Unicast Performance Measurement Application

The following example was derived from a network performance tool. It attempts to put in place a unicast reservation before measuring network performance. Waiting is accomplished using the timeout option in the sender and receiver calls.

4.1.1. Sender Application

This sender application opens a TCP connection to the ‘‘receive-hostname’’ on port 1111 and attempts to reserve 1000 bytes/sec of average bandwidth. After waiting at most 10 seconds for a reservation to be put in place, this application streams data to the receiver to measure network performance and then closes the connection.

```
#include <scrapi.h>

void
main(int argc, char *argv[])
{
        int timeout = 10000;        /* wait for at most 10 seconds */
        double bw = 1000;           /* Average bandwidth 1Kbytes/sec */
        struct SOCKADDR destination;
        char *hostname = "receive-hostname";
        unsigned short port = 1111;

        /* translate host name or address */
        if (!scrapi_sockaddr_parse((struct sockaddr *) &destination,
                        hostname,htons(port))) {
                fprintf(stderr,"Could not parse host address");
                exit(1);
        }

        /* make an RSVP based reservation */
        if (!scrapi_sender((struct sockaddr *) &destination,IPPROTO_TCP,
                        NULL,scrapi_service_cl,bw,0,timeout))
                scrapi_perror("RSVP unable to reserve bandwidth");

        /* run test */
        scrapi_close(NULL,0);
}
```

4.1.2. Receiver Application

This receiver application accepts a TCP connection and attempts to make a reservation. After waiting at most 10 seconds for a reservation to be put in place, this application consumes data from the sender to measure network performance and then closes the connection.

```
#include <scrapi.h>
#include <errno.h>

void
main(int argc, char *argv[])
{
        int fd,len,timeout = 10000;        /* wait for at most 10 seconds */
        struct SOCKADDR destination;

        /* open, bind, and listen for connection */
        /* fd = accept(...); */
        len = sizeof(destination);
        if (getsockname(fd,(struct sockaddr *) &destination,&len) == -1) {
                perror("getsockname");
                exit(1);
        }

        /* make an RSVP based reservation */
        if (!scrapi_receiver((struct sockaddr *) &destination,IPPROTO_TCP,
                        NULL,1,scrapi_style_distinct,timeout))
                scrapi_perror("RSVP unable to reserve bandwidth");

        /* run test */
        scrapi_close(NULL,0);
}
```

4.2. Multicast Multimedia Application

This example highlights the inclusion of the SCRAPI API into a Tcl/Tk event loop of a multimedia application. This is a full duplex application that is sending and receiving data on the same address. This application does not wait for completion status from the reservation calls.

```
#include <tcl.h>
#include <tk.h>
#include <scrapi.h>

void
callback(ClientData data, int mask)
{
        if (!scrapi_dispatch())
                Tk_DeleteFileHandler(data);
}

void
main(int argc, char *argv[])
{
        int fd;
        struct SOCKADDR destination;
        double bw = 1000;               /* Average bandwidth 1Kbytes/sec */
        char *hostname = "receive-hostname";
        unsigned short port = 1111;

        /* translate host name or address */
        if (!scrapi_sockaddr_parse((struct sockaddr *) &destination,
                        hostname,htons(port))) {
                fprintf(stderr,"Could not parse host address");
                exit(1);
        }

        /* make an RSVP based reservation */
        if (!scrapi_sender((struct sockaddr *) &destination,IPPROTO_UDP,
                        NULL,scrapi_service_cl,bw,0,0))
                scrapi_perror("RSVP unable to reserve bandwidth");
        if (!scrapi_receiver((struct sockaddr *) &destination,IPPROTO_UDP,
                        NULL,1,scrapi_style_distinct,0))
                scrapi_perror("RSVP unable to reserve bandwidth");
        fd = scrapi_getfd((struct sockaddr *) &destination,IPPROTO_UDP);
        Tk_CreateFileHandler((ClientData) fd,TK_READABLE,callback,
                (ClientData) fd);

        /* send data */
        scrapi_close(NULL,0);
}
```

5. Conclusion

The SCRAPI interface provides a simple method to add RSVP support to many network applications. It supports both IPv4 and IPv6 and attempts to simplify user developed code to support both address families. Coding examples are provided to give additional guidance on the usage of this API.

6. Security Considerations

Security considerations are not discussed in this memo.

7. Acknowledgements

The author would like to thank Steve Berson for helping to define this API.

8. References

[1] Braden, R., Ed., et. al., *Resource Reservation Protocol (RSVP) - Version 1 Functional Specification*, RFC 2205, September 1997.

[2] Braden, R., Hoffman, D., *RAPI -- An RSVP Application Programming Interface Version 5*, Work In Progress, November 1997.

[3] Shenker, S., Partridge, C., Guerin, R., *Specification of Guaranteed Quality of Service*, RFC 2212, September 1997.

[4] Wroclawski, J., *Specification of the Controlled Load Quality of Service*, RFC 2211, September 1997.

9. Author's Address

Bob Lindell
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Table of Contents

1. Introduction 2

2. Functional Description 3
3. Application Programming Interface Definition 4
3.1. Reservation API Description 4
3.2. Asynchronous Event Loop API Description 7
3.3. Error Handling API Description 8
3.4. Address Manipulation API Description 10
4. Application Code Templates 12
4.1. Unicast Performance Measurement Application 12
4.1.1. Sender Application 12
4.1.2. Receiver Application 13
4.2. Multicast Multimedia Application 14
5. Conclusion 15
6. Security Considerations 16
7. Acknowledgements 16
8. References 16
9. Author's Address 16