INTERNET-DRAFT
Expiration: August 1999
File:draft-lindell-rsvp-scrapi-02.txt

Bob Lindell
ISI

# SCRAPI - A Simple “Bare Bones” API for RSVP

12 December 2000

## Status of this Memo

## 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.

To provide simplicity, event upcalls to the application do not exist in SCRAPI. Because of the design choice, it may 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.

There are three main functions included in this API, one which is used by the sender side, one to be used at the receiving end, and a function 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.

Error reporting is handled in two distinct forms. The first is an aggregated error model that is unique to SCRAPI. SCRAPI includes a tri-valued error model with error states of “red”, “yellow”, or “green”. This model gives feedback to an application on the status of reservations at an given time. The exact description of each error state will be described below. Second, it is possible to get an error number and corresponding error description after executing a SCRAPI sender, receiver, or close function. This is similar to the **errno** support in Unix. These error codes are of limited use, since they only provide feedback on an immediate failure of a request. This is most likely due to invalid arguments or some general system error. A timeout errno is returned if a timer expires on the sender or receiver calls. A timeout error does not abort the request, it is merely an indication that the timer has expired. Applications can abort a request following a timeout by closing the flow.

There is support for applications which have polling based event loops using a system call such as **select**. Analogous functions to the RAPI interface for supporting this functionality is provided in SCRAPI.

The remaining functions in the API are utility functions to ease the development of an interface which supports IPv4 and IPv6. These are documented in Appendix 1. These functions are not viewed as part of the SCRAPI API, but rather a collection of useful address manipulation functions which should be provided in system libraries. In the future, these functions will be removed in favor of system supplied functionality. The objective of these extensions was 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.

## 2. Functional Description

The basic abstraction for the SCRAPI API is a flow. A flow is defined in terms of three parameters: the *destination* address, a *protocol* number, and the *source* address. This triple is used for the sender, receiver, status, and close operations in SCRAPI.

Applications which use SCRAPI get a simplied service model. The average bandwidth specified by the sender is currently used for the integrated services controlled load [4] or guaranteed service [3] token bucket rate (r). The peak token bucket rate (p) is set to positive infinity and token bucket depth (b) is set to twice the average bandwidth. Minimum policed unit (m) and maximum packet size (M) 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 data rate (R) is set to the maximum of all senders and the slack term (S) is set to zero. If blocking is requested, the receiver blocks until the receipt of a CONFIRM event.

If any RSVP errors occur during a blocking sender or receiver API call, the call will unblock.

Internal to SCRAPI, there is support for both IPv6 Flow Labels and the Generalized Port Identifier (GPI) [5]. These are not visible at the API. It is assumed that applications would continue to use port numbers to specify flows and that SCRAPI internally would convert these port numbers to either a Flow Label or a GPI value using system supplied functionality.

A state diagram of the SCRAPI reservation API, for a given data flow, is shown in Figure 1. A sender call subscripted with zero designates a sender call with the bandwidth set to zero. Similarly, a receiver call subscripted with zero designates a receiver call with the reservation flag set to **FALSE**.

```
                                         +---------+
                         +----------| Closed  |----------+
               scrapi_sender    +---------+    scrapi_receiver
                         |                               |
scrapi_sender   |                               |   scrapi_receiver
            +----+   |                               |   +----+
            |    V   V                               V   V    |
            |   +---------+                    +---------+    |
            +---|         |                    |         |---+
                  |  Send   |                    |   Rcv    |
      +-------|         |<-----+      +----->|         |-------+
      |       +---------+      |      |      +---------+       |
      |            |           |      |            |           |
      |            |scrapi_receiver(0) |           |           |
      |            |           |      |            |           |
      |            |           |   scrapi_sender(0)|           |
      |            |           |      |            |           |
      |     scrapi_receiver    |      |     scrapi_sender      |
      |            |          +---------+          |           |
      |            +--------->|         |<---------+           |
      |                       | SendRcv |                      |
      |  scrapi_sender or +---|         |                      |
      |  scrapi_receiver  |   +---------+                      |
      |                   |    ^   |                           |
      |                   +----+   |                           |
      |                            |                           |
  scrapi_close or          scrapi_close           scrapi_close or
 scrapi_sender(0)                  |            scrapi_receiver(0)
      |                            V                           |
      |                       +---------+                      |
      +---------------------->| Closed  |<---------------------+
                                         +---------+
```

Figure 1: SCRAPI API State Diagram

## 3. The Simplified Error Model of SCRAPI

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.

A state diagram of the SCRAPI error model, for a given data flow, is shown in Figure 2. A close operation takes the model from any state back to the red state. This was intentionally omitted to increase the clarity of the diagram.

There are a number of difficulties in providing a simple, robust, aggregated error model based on RSVP signaling information. These issues are highlighted below as either "long term" or "short term" stability issues. If the model provides "long term" stability, it will eventually report the currect status. There should be no terminal states of the model that cause the reported status to remain fixed without the ability to transition to a new state if conditions change. If the model has "short term" stability, it attempts to damp rapid oscillations between states.

There are a few obstacles to "long term" stability is the current error reporting model for RSVP. Once an error message has been received, it is not always possible to determine when an error condition has cleared. If an application possessed global knowledge about refresh rates and link reliability assumptions along a path, the solution would be to wait until enough time has expired to assume that the lack of any subsequent error message is an indication that an error has cleared. Unfortunately, this information about refresh rates at any given point in the path is unknown to applications. The solution in SCRAPI is an attempt to match error messages with the receipt of other messages that could be used to indicate the clearing of a given error. For example, a CONFIRM message received subsequent to a RESV ERROR message might indicate that the error condition has cleared.

For unicast, path errors can be paired with the complementary RESV message to transition the error model between red and green. With multicast, an application could be receiving refreshes of a PATH ERROR message from one branch of the multicast tree and RESV messages from another branch. Should the error state of this model be red or green?

RSVP CONFIRM messages are not delivered reliably. SCRAPI should probably include a reliability model for confirmations so that the error model does not get stuck in a non-green state due to the loss of a CONFIRM message. This can be implemented by making repeated requests for a CONFIRM message from the API.

There are also "short term" stability issues that have not been adequately addressed in SCRAPI. In the multicast example above, an application could be receiving mixture of path errors and RESV messages for the same flow. Since the error messages will continue to refresh, this may cause oscillatory behavior of the error model. Similarly, RESV errors can be received soon after a confirmation due to the merging rules for RSVP. In both of these cases, it might be useful to add some delay to state transitions in the error model.

It is anticipated that improvements to the design and implementation of this error model will occur as we gain a better understanding of of the use of RSVP with applications.

```
                                       +---------+
                       +----------|   Red   |----------+
             scrapi_sender    +---------+   scrapi_receiver
                       |                              |
                       V                              V
                +---------+                   +---------+
    +-------| Yellow  |<-----+      +----->| Yellow  |-------+
    |       +---------+      |      |      +---------+       |
    |            |           |      |           |            |
    |            |scrapi_receiver(0) |           |            |
    |            |           |      |           |            |
    |            |           |  scrapi_sender(0)|            |
    |            |           |      |           |            |
    |      scrapi_receiver   |      |     scrapi_sender      |
    |            |          +---------+         |            |
    |            +--------->| Yellow  |<---------+           |
    |                       +---------+                      |
    |                            |                           |
RESV                        RESV or                    CONFIRM
    |                         CONFIRM                        |
    |                            |                           |
    |                            V                           |
    |                       +---------+                      |
    +----------------------->|  Green  |<---------------------+
                       +--------->|         |<---------+
                       |          +---------+          |
                       |            |      |           |
                       |         PATH    RESV          |
                       |        ERROR    ERROR         |
                   RESV         |       |       CONFIRM
                       |           V      V            |
                       |   +---------+  +---------+    |
                       +---|   Red   |  |   Red   |---+
                             +---------+  +---------+
```

Figure 2: SCRAPI Error Model State Diagram

## 4. A Comparison to RAPI

In this section, a brief comparison of SCRAPI and RAPI are provided in outline form.

- SCRAPI is entirely flow based, there is no concept of a session. Calls to RAPI session operations are internally hidden by the sender, receiver, and close operations.

- The SCRAPI sender command is simplified by not requiring sender templates or flow specifications. These are constructed internally based on a single bandwidth parameter provided. A RAPI based application has more control over specifying the contents of sender templates and flow specifications.

- The SCRAPI receiver command is simplified by not requiring filter or flow specifications. These are constructed internally based on the sender templates and flow specifications received in a PATH message. In addition, the receiver command may be called prior to the receipt of any PATH messages without causing an error. A RAPI based application has more control over specifying the contents of filter and flow specifications, including making a reservation for an amount different than the senders flow specification.

- A default SCRAPI callback function is defined to process RSVP messages. It supports automatic responses to PATH messages given prior information from a receiver call. It also maintains state for the simplified SCRAPI error model described in the previous section. A RAPI based application can define an arbitrary callback function which might implement a complex error model and contain functionality to perform QoS negotiation.

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.

## 5. Application Programming Interface Definition

The section describes the functions of the SCRAPI API. It includes the error reporting capabilities and the asynchronous event support.

### 5.1. Reservation API Description

| | |
|---|---|
| Function Name: | **scrapi_sender** |
| Syntax: | int scrapi_sender(<br>const struct sockaddr *destination,<br>int protocol,<br>const struct sockaddr *source,<br>double bw,<br>unsigned long msecs<br>); |
| Description: | SCRAPI call for a data sender. The *destination* address, *protocol* number, and the *source* address of the data flow are supplied as the first three arguments. The source address may be wild, but a non-wild port must be provided. The *bw* parameter is the average bandwidth of the flow in bytes/sec. 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* address, *protocol* number, and *source* address). A value of 0 for the *bw* parameter unregisters a sender for this data flow. |
| Return Values: | **TRUE** if successful, **FALSE** otherwise. Unsuccessful operations will set 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_service service,
        scrapi_style style,
        unsigned long msecs
);
```

Description: SCRAPI call for a data receiver. The *destination* address, *protocol* number, and the *source* address of the data flow are supplied as the first three arguments. The source address can be set to **NULL** to choose any source. In addition, the *source* address can be specified with a wild port number of 0 to match a source address regardless of port number. Wild port number requests take precedence over any source requests. The *reserve* parameter should be set to **TRUE** or **FALSE** to turn on and off a reservation for that data flow respectively. The *service* parameter specifies the service, currently either Controlled Load or Guaranteed. 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 including removing a reservation request.

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

Also See: scrapi_errno, scrapi_get_status

| | |
|---|---|
| Function Name: | **scrapi_close** |
| Syntax: | int scrapi_close(<br>const struct sockaddr *destination,<br>int protocol,<br>const struct sockaddr *source<br>); |
| Description: | SCRAPI call to close a flow for sending and receiving. If the *destination* address is set to **NULL**, all flows will be closed and the other parameter values will be ignored. If the *source* address is set to **NULL**, the close applies to all possible sources. |
| Return Values: | **TRUE** if successful, **FALSE** otherwise. Unsuccessful operations will set an appropriate error code. |
| Also See: | scrapi_errno, scrapi_get_status |

| | |
|---|---|
| Enum Name: | **scrapi_service**<br>1. scrapi_service_cl<br>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**<br>1. scrapi_style_shared<br>2. scrapi_style_distinct |
| Description: | Enumerated types for specificing the reservation style. Currently, shared (wildcard) and distinct styles are supported. |

## 5.2. Error Handling API Description

The section describes the error reporting functions of the SCRAPI API.

Function Name: **scrapi_get_status**

Syntax:

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

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. If the *source* address is set to **NULL**, a yellow status indicates that the flow is valid but no reservation operation(s) as a sender or receiver has completed successfully. Once a single reservation completion has been detected for either a sender or receiver, the flow has a green status. If the *source* address is not set to **NULL**, the status applies only to the receiver for the specified source. Thus, if an application is both a sender and receiver for a the given flow, any relevant sender status information is ignored.

Return Values: scrapi_status_red, scrapi_status_yellow, or scrapi_status_green.

Function Name: **scrapi_errno**

Syntax:

```
int scrapi_errno(
        const struct sockaddr *destination,
        int protocol,
        const struct sockaddr *source
);
```

Description: Set to the errno value of the last SCRAPI call for a given flow. Must be called with identical arguments given in a preceding SCRAPI function. Since failed SCRAPI functions may not have created a flow, or caused the closure of a flow, the errno may be transitory and should be observed immediately after the failure.

Also See: scrapi_perror, scrapi_errlist

| | |
|---|---|
| Function Name: | **scrapi_perror** |
| Syntax: | void scrapi_perror( const struct sockaddr *destination, int protocol, const struct sockaddr *source, const char *string ); |
| Description: | SCRAPI call to print an error message analogous to the perror() library call. |
| Also See: | scrapi_errno, scrapi_errlist |

```
void scrapi_perror(
        const struct sockaddr *destination,
        int protocol,
        const struct sockaddr *source,
        const char *string
);
```

| | |
|---|---|
| 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. |
| Also See: | scrapi_errno, scrapi_perror |

| | |
|---|---|
| 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**.

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.

## 5.3. Asynchronous Event Loop API Description

The section describes the asynchronous event functions of the SCRAPI API.

Function Name: **scrapi_poll_list**

Syntax:

```
void scrapi_poll_list(
        fdset *set
);
```

Description: SCRAPI call to get all API file descriptors to use in a subsequent **select** call.

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.

## 6. 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.

### 6.1. 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.

#### 6.1.1. 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>

#define        SAP(x)                ((struct sockaddr *) (x))

void
main(int argc, char *argv[])
{
        int fd,len;
        int timeout = 10000;   /* wait for at most 10 seconds */
        double bw = 1000;      /* Average bandwidth 1Kbytes/sec */
        struct SOCKADDR destination,source;
        scrapi_status status;

        char *hostname = "receive-hostname";
        unsigned short port = 1111;

        /* translate host name or address */
        if (!scrapi_sockaddr_parse(SAP(&destination),hostname,
                        htons(port))) {
                fprintf(stderr,"Could not parse host address");
                exit(1);
        }
        /* open, bind, and connect */
        /* fd = socket(...); */
        len = sizeof(source);
        if (getsockname(fd,SAP(&source),&len) == -1) {
                perror("getsockname");
                exit(1);
        }

        /* make an RSVP based reservation */
        if (!scrapi_sender(SAP(&destination),IPPROTO_TCP,SAP(&source),
                        bw,0,timeout))
                scrapi_perror(SAP(&destination),IPPROTO_TCP,SAP(&source),
                        "RSVP unable to reserve bandwidth");
        status = scrapi_get_status(SAP(&destination),IPPROTO_TCP,NULL);

        /* run test */
        if (status == scrapi_status_green) {
                status = scrapi_get_status(SAP(&destination),IPPROTO_TCP,
                        NULL);
                if (status != scrapi_status_green)
                        fprintf(stderr,
                                "RSVP reservation lost during test!");
        }
        scrapi_close(NULL,0,0);
}
```

### 6.1.2. 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>

#define        SAP(x)                ((struct sockaddr *) (x))

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,SAP(&destination),&len) == -1) {
                perror("getsockname");
                exit(1);
        }

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

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

## 6.2. 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>

#define        SAP(x)                ((struct sockaddr *) (x))

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

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

        /* translate host name or address */
        if (!scrapi_sockaddr_parse(SAP(&destination),hostname,
                        htons(port))) {
                fprintf(stderr,"Could not parse host address");
                exit(1);
        }
        len = sizeof(source);
        if (getsockname(fd,SAP(&source),&len) == -1) {
                perror("getsockname");
                exit(1);
        }

        /* make an RSVP based reservation */
        if (!scrapi_sender(SAP(&destination),IPPROTO_UDP,SAP(&source),
                        bw,0,0))
                scrapi_perror(SAP(&destination),IPPROTO_UDP,SAP(&source),
                        "RSVP unable to reserve bandwidth");
        if (!scrapi_receiver(SAP(&destination),IPPROTO_UDP,SAP(&source),
                        1,scrapi_service_cl,scrapi_style_distinct,0))
                scrapi_perror(SAP(&destination),IPPROTO_UDP,SAP(&source),
                        "RSVP unable to reserve bandwidth");
        scrapi_poll_list(&set);
        for (i = 0;i < FD_SETSIZE; i++) {
                if (!FD_ISSET(i,&set))
                        continue;
                Tk_CreateFileHandler((ClientData) i,TK_READABLE,
                        callback,(ClientData) i);
        }

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

## 7. 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.

## 8. Security Considerations

Security considerations are not discussed in this memo.

## 9. Acknowledgements

The author would like to thank Steve Berson for helping to define this API. I would also like to thank Bob Braden for his help in improving the definition of the API and corresponding documentation.

## 10. 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.

[5] Berger, L., O'Malley, T., *RSVP Extensions for IPSEC Data Flows*, RFC 2207, September 1997.

## 11. Author's Address

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

## 12. Appendix 1: Address Manipulation API Description

These are utility functions to ease the development of an interface which supports IPv4 and IPv6. These are not viewed as part of the SCRAPI API, but rather a collection of useful address manipulation functions which should be provided in system libraries. In the future, these functions will be removed in favor of system supplied functionality.

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(<br>    const struct sockaddr *address<br>); |
| Description: | Pretty print an address. |
| Return Values: | A valid string if printable, **NULL** otherwise. |

|  |  |
|---|---|
| Function Name: | **scrapi_sockaddr_get_addr** |
| Syntax: | int scrapi_sockaddr_get_addr(<br>    const struct sockaddr *address,<br>    char *addr<br>); |
| Description: | Store the address field of the sockaddr structure to a network order binary representation of an address which starts at the memory location specified. |
| Return Values: | **TRUE** if successful, **FALSE** otherwise. |

|  |  |
|---|---|
| Function Name: | **scrapi_sockaddr_set_addr** |
| Syntax: | int scrapi_sockaddr_set_addr(<br>    struct sockaddr *address,<br>    char *addr<br>); |
| Description: | Load the address field of the sockaddr structure from a network order binary representation of an address which starts at the memory location specified. |
| Return Values: | **TRUE** if successful, **FALSE** otherwise. |

|  |  |
|---|---|
| Function Name: | **scrapi_sockaddr_get_port** |
| Syntax: | unsigned short scrapi_sockaddr_get_port(<br>    const struct sockaddr *address<br>); |
| 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.

Function Name: **scrapi_sockaddr_length**

Syntax:
```
int scrapi_sockaddr_length(
        struct sockaddr *address
);
```
Description: Return the length of a struct sockaddr for the given address family.

Return Values: The length if successful, 0 otherwise.

## Table of Contents