# What are yRFCs?

**yRFCs are discussion documents on one or more issues related to the design, development or implementation of the Y-Comm architecture. Y-Comm is a new architecture being developed to support heterogeneous networking. yRFCs therefore represent the views of the authors of the document. They are non-binding and do not oblige anyone to agree with or to implement any concepts or details expressed therein. They can also be modified without notice. Finally, yRFCs are public documents and should not in whole or in part be the basis of a patent or copyright claim. Please contact the authors directly to discuss relevant issues.**

**Note: this is a draft document; please do not circulate!!!**

# yRFC3: The Simple Protocol Lite (SP-Lite) Specification

**Authors: Leroy Riley (lr347@live.mdx.ac.uk) and Glenford Mapp (g.mapp@mdx.ac.uk)**

**This document was released to the Y-Comm Website Team on 7th July 2014**

**Updated 4th November 2015**

**Updated 30th August 2016**

## 1.0 Introduction:

This yRFC discusses the specification of a new version of the Simple Protocol (SP) called SP-Lite. SP-Lite is a simpler version of SP, which was fully specified in yRFC2. SP was designed to optimize transport in the local area. This document defines SP-Lite and gives a definition of its interfaces. The motivation for doing this is based around the concept that it is necessary to separate the need for Local Area Networking (LAN), which may be defined by different local conditions such as heterogeneous wireless networking, high speed communications, and new environments such as the Cloud from Internetworking which is more based on Wide Area Networking (WAN). The strategy of tuning TCP to adapt to these conditions has met with mixed results. So the plan is to develop a simple protocol that can be used to optimize local interactions. The Simple Protocol, which we call SP, is being used to provide this functionality.

## 1.1 Background

Communications in Local-Area Network (LAN) and Wide-Area Network (WLAN) environments are beginning to take divergent paths. This has been motivated by several factors. The first is that local networks speeds are still increasing; 1 Gbps is common in the Local Area with 10 Gbps now becoming available. In addition, the rise of wireless also means that a lot of peripheral networks will be wireless networks. This indicates that the

transportation characteristics of end networks are will be dominated by characteristics of wireless communications, which are completely different from wired systems. Hence, transport protocols such as TCP, which were developed in the context of wired environments are not able to perform in an optimal way in wireless environments. Adapting TCP has had mixed results, because it is difficult to really tune the protocol for these diverse LAN conditions.

The authors therefore believe that the argument that one transport protocol should be used for both global and local environments has been severely weakened. This paper looks at the development of a transport protocol specially designed for the local area. The authors believe that TCP should be used as a WAN protocol while a local protocol is used for local communications.

## 2.0 Requirements for a LAN Transport

A transport protocol for LAN communications needs to have certain properties to optimize its performance that differs from WAN transport protocols such as TCP.

### Larger Window Sizes

In order to make use of high-speed LANs, LAN protocols should use a much larger window size compared with WAN protocols. Since the LAN is fast, a bigger window size can be used by default.

### Support for Message-Based Communications

TCP is based on stream-like communication. There are no message boundaries. However, most communication in LAN environments tends to be message or transactional based. So using a message-based approach is better for LAN protocols.

### Ease of Packet Processing: Keeping it simple.

There is a strong case to use this design to try to simplify protocol processing. Thus the idea would be to have a greater set of connection states as well as defined packet types. Thus the packet type is used as the key parameter to drive the main loop.

### Keep it flexible.

One of the key issues is that since there are a lot of diverse applications and it is necessary for the protocol to give different qualities of service to different applications, this means that various mechanisms such as check-summing and error correction need to be set independently to yield different qualities of service.

## 3.0 Protocol Specification

Figure 1 shows the Diagram of the Simple Protocol while Figure 2 shows the length of the individual fields.

## Structure of SP

<table>
<tr><td colspan="4">DEST_ID</td><td colspan="2">SRC_ID</td></tr>
<tr><td>PK_TYPE</td><td>PRI</td><td>CN</td><td>FLAGS</td><td colspan="2">CHKSUM</td></tr>
<tr><td colspan="4">TOTAL_LEN</td><td>PBLOCK</td><td>TBLOCK</td></tr>
<tr><td colspan="4">MESS_SEQ_NO</td><td colspan="2">MESS_ACC_NO</td></tr>
<tr><td colspan="2">SYNC_NO</td><td colspan="4">WINDOW_SIZE</td></tr>
</table>

Figure 1: The Structure of SP Header (Total Size is 20 bytes)

## Fields of SP

- DEST_ID (16): identifying connection at remote end
- SRC_ID (16): identifying connection at local end
- PK_TYPE (4): identifies the type of packet
- PRI (2): supports 4 levels pf priority
- CN(2): supports congestion notification
- CHKSUM (16): sixteen bit checksum
- TOTAL_LEN (16): the total packet length
- PBLOCK (8): identifies the current block
- TBLOCK (8): the total number of blocks
- MESS_SEQ_NO (16): last message sent
- MESS_ACK_NO (16): last message received
- SYNC_NO (10): the SYNC number
- WINDOW_SIZE (22): the window size

Figure 2: The Fields of SP

The individual fields are detailed below:

The DEST_ID is a connection identifier on the remote machine, while the SRC_ID identifies the connection on the local machine. So if the connection runs UDP/IP, the connection is independently identified by [DEST_ID, (IP_address, UDP_port (DEST_ID))] or [SRC_ID, (IPaddress, UDP_port(SRC_ID))]. Note that a value of zero is regarded as an

invalid connection identifier. PK_TYPE is the type of packet being sent or received. SP supports a number of them:

START: the first packet transmitted to set up a connection

REJ: this packet indicates that the connection request has been rejected.

DATA: this is a data packet

ACK: this is an Acknowledgement packet

NACK: this is a NACK packet and reports that data packets are missing

END: this is an end packet, which is used to close a connection.

FIN: this packet signals the end of the connections. A FIN packet must not be acknowledged.

ECHO: an ECHO packet is used to measure RTT. It is used to support applications that would like to monitor the latency of connections.

ECHO_1 and ECHO_2: These packets are sent back-to-back packets to measure the bandwidth and the Round Trip Times using the Packet-pair technique.

STATUS: This packet has been added to maintain flow control. A status packet records the state of the sender in terms of the last packet that has been transmitted. If a reply is requested, then the receiver must reply with a packet that shows the last packet properly received.

IDLE: an idle packet can be sent when there is no data being sent by the sender. So if the sender sends data and it is all acknowledged and it has no more data to send immediately then it should send an IDLE packet. The IDLE packet tells the other side that the sender has gotten all the acknowledgements for all the data it sent and there is no new data to be sent at the moment On receiving an idle packet with a requested reply, the receiver should only reply with an IDLE packet if it too has gotten all its data acknowledged and it has no more data waiting to be sent. If this is not the case then the receiver should reply with a STATUS packet with a requested reply.

CWIN: this packet is used to change the window size of a connection immediately to the window size of in the received CWIN packet. A CWIN packet is sent with a requested reply and a CWIN timer is started. A CWIN packet must be acknowledged by the remote end by the sending of a reply CWIN packet. CWIN packets are used to change the quality of service on a connection and to ensure that things such as handover can happen smoothly by setting CWIN to zero (closing the window) at the start of handover and opening the window again once handover has been completed.

Note, that if the station gets a CWIN packet and this results in a closed window; i.e., the sender has set its receive window to zero, then if the station that received the CWIN packet still has data to send on the connection, it must periodically send a STATUS packet with a requested reply. The receiver of STATUS packets must respond to STATUS packets even if it has closed the window

PRI – 2 bits are used; hence SP supports 4 levels of priority. SP guarantees that a higher priority packet will always be delivered before a lower-priority packet if both packets are sent at the same time.

CN – These bits are used to support congestion notification techniques such as ECN as detailed in RFC 3168.

FLAGS: this comprises a field containing 8 bits:

BIT (0): Window-Size is valid

BIT (1): ST_CKS: Checksum this packet

BIT (2): ST_RTR: Recover packet if checksum error or missing packets are detected

BIT (3): ST_RETRANS: This is an indication that the packet has been retransmitted

BIT (4): REMOTE_RESET: The connection has been reset by the other side

BIT (5): REPLY_REQUESTED: A reply has been requested for this packet

BIT (6): REPLY: This is a reply to a previous request

BIT (7): End-of-Message: Indicates that the last message was completely received.

CHKSUM: this is the 16-bit checksum. It is the same as used in TCP.

TOTAL_LEN: This is the total length of the packet INCLUDING the SP header.

PBLOCK: This is used to signify which part of the message is contained in this packet.

TBLOCK: The total number of blocks/packets in a message.

MESS_SEQ_NO: the last message sent. Note that only DATA and END packets can increase the MESS_SEQ_NO. The sending of other packet types does not increase the MESS_SEQ_NO of a connection.

MESS_ACK_NO: the last message received.

WINDOW_SIZE: This is 22 bits long and specifies the number of bytes that can be sent by the sending side before waiting for an acknowledgement from the receiver. Hence a maximum of 4 MBs can be sent before waiting for an acknowledgement.

SYNC_NO: This is a 10-bit random number that is generated at connection setup. This provides security against connection replay attacks. Every packet on a connection must have the correct SYNC number or a SECURITY violation error is returned. In addition, SYNC numbers are crossed when the connection starts which increases the security. So let us suppose that Client A begins a connection to Server B by sending a START packet with its initial SYNC random value of X. Then Server B sends a START reply with its initial SYNC random value of Y. When Client A sends data or any packet on the connection it must set the SYNC value to Y. This signals to the Server that Client A has successfully received the START reply packet. Similarly when Server B sends a packet on this connection it uses the SYNC value of X.

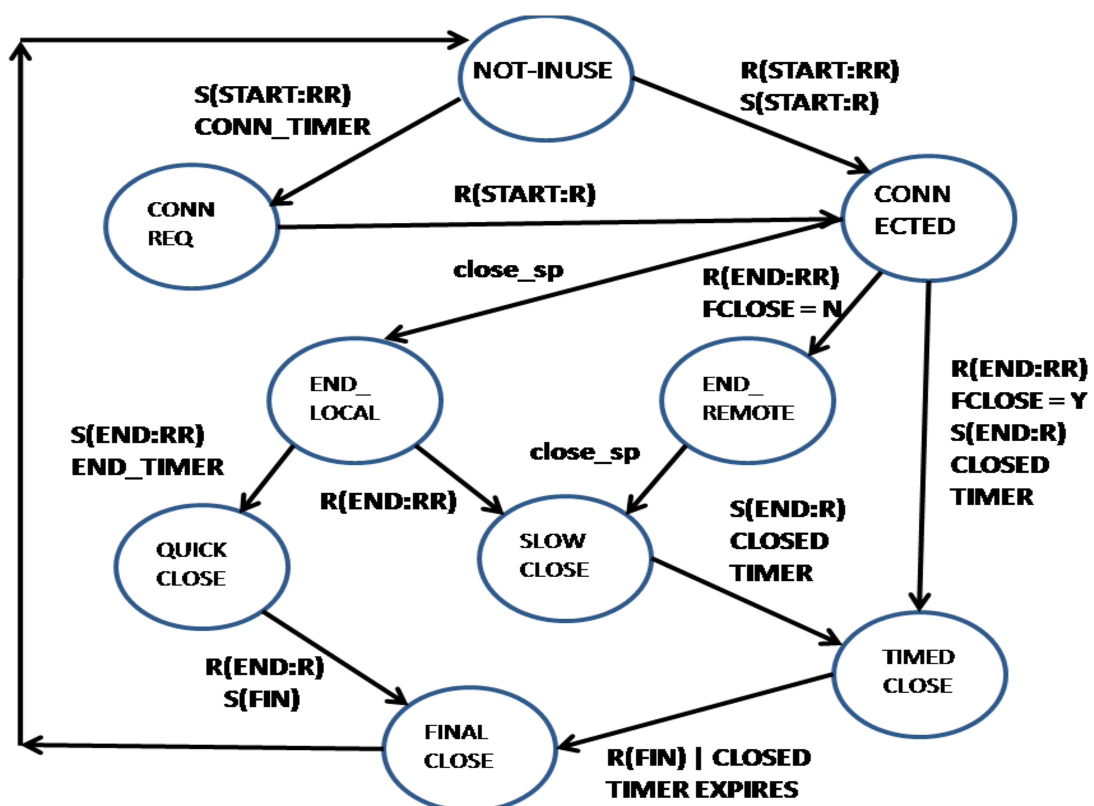


**Figure 3: Showing the Different Connections States**

**Connection States**

The SP-Lite supports the following connection states as shown in Figure 3.

NOT_INUSE = 0: this connection is not valid

CONN_REQUESTED = 1: a connection has been requested so a START packet with a Reply Requested (RR) bit set, has been transmitted but there has not been a reply.

CONNECTED = 2: the connection is in the connected state.

END_REQUEST_REMOTE: = 3: This state indicates that the application at the remote has sent all its data and now wishes to close the connection. Thus no new data packets will be sent from the remote end.

END_REQUESTED_LOCAL = 4: This state indicates that the application at the local end has sent all its data and now wishes to close the connection. This is done be calling the close_sp routine on the connection. This means that no new data packets can be sent on the connection from the local end. If all the data sent by the application has been acknowledged, the connection goes to the QUICK_CLOSE state.

QUICK_CLOSE = 5: This state is entered into when the client wishes to close the connection after sending all its data. So an explicit close function such as close_sp has been called. When the close_sp function is called, the client checks to see if all data has been sent and acknowledged and if so, an END packet with the REPLY_REQUESTED bit set is sent. The status of the connection goes to QUICK_CLOSE and the END_TIMER is started waiting to receive an END packet with the REPLY bit.

SLOW_CLOSE = 6: If a connection is in the END_LOCAL state and the endpoint gets an END packet indicating that the other side has sent all its data and that data has been acknowledged and it is ready to close the connection; if however, the local side still has data that is yet to be acknowledged then it goes to the SLOW_CLOSE state. SLOW_CLOSE can also be entered if the endpoint gets an END packet from the remote end and so goes to END_REMOTE, but it may have packets that are yet to be sent by the application. If the application now indicates that it wishes to close the connection and there are still packets to be acknowledged; then the system goes to the SLOW_CLOSE state. When all the outstanding data has been acknowledged there is no more local data to be sent or acknowledged, then the system sends an END packet with the REPLY bit set and goes to the TIMED_CLOSE state which is explained below.

TIMED_CLOSE = 7: if you are in SLOW_CLOSE waiting on the all the packets on the local end to be sent and acknowledged, when this occurs, then an END packet is sent with the REPLY bit set and the connection goes to the TIMED_CLOSE state. A CLOSED TIMER is started (usually set to 2 minutes). If this timer goes off, then the connection goes to the FINAL_CLOSE state.

FINAL_CLOSE = 8: This means that the connection is finally closed and all the resources associated with the connection can be de-allocated. The FINAL_COSE state may be entered into using two mechanisms: first is the expiration of the CLOSED TIMER when the connection is in the TIMED_CLOSE state. The other way is by sending or receiving a FIN packet. If the connection is in the QUICK_CLOSE state; meaning that it has sent an END packet with a REPLY_REQUESTED bit set; if it then receives an END packet with the REPLY bit set, indicating the other side

has also closed; it sends a FIN packet and goes to the FINAL_CLOSE state indicating at all its resources can be de-allocated. In addition, if you are in the TIMED_CLOSE: the system receives a FIN packet; it cancels the CLOSED TIMER and immediately moves to the FINAL_CLOSE state, where the resources of the connection are de-allocated.

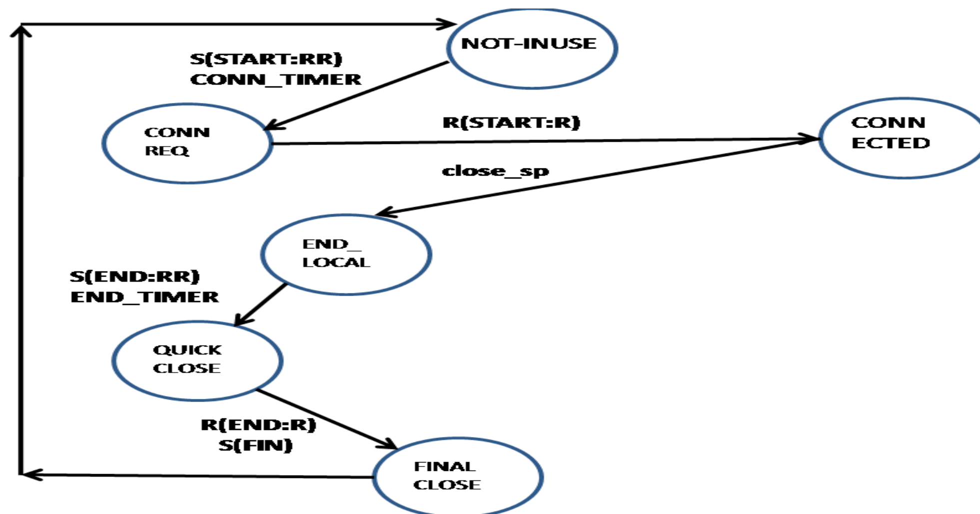


**Figure 4: State Diagram for SP Client**

**Client/Server Interaction in SP**

A typical SP connection as seen from the client side in a client/server environment is shown in Figure 4. The Client sends a START packet with the REPLY_REQUESTED bit set and goes to the CONN_REQ state. It then gets the START packet with the REPLY bit set indicating that the connection has been accepted and goes to the CONNECTED state. When finished with the server, the client calls close_sp, which puts it in the END_LOCAL state. When all packets have been sent and acknowledged, the client sends an END packet with the REPLY_REQUESTED bit set, and goes to the QUICK_CLOSE state. When it receives an END packet with the REPLY bit set and then sends a FIN packet, which closes the connection.

A connection at the server moves from the NOT_INUSE to the CONNECTED state when it receives a START packet with a REPLY_REQUESTED bit set and sends a START packet with the REPLY bit set indicating that it has accepted the connection. The server then services client requests. When the client is finished it will send an END_PACKET to the server with the REPLY_REQUESTED bit set. The server goes to the END_REMOTE state and can continue to send data until it is ready to close by calling the close_sp routine. It then goes to the SLOW_CLOSE state. If all the data it has sent has been acknowledged, then it sends an END packet with the REPLY bit set and goes to TIME_CLOSE and starts the CLOSED_TIMER. The connection goes to FINAL_CLOSE when the CLOSED_TIMER expires or it gets a FIN packet from the client.

### Supporting the Concept of FASTCLOSE for servers

In a client/server interaction, the server would normally close a connection immediately if a client wants to close the connection, whether or not it still has data to send to the client. This option is called FASTCLOSE and in SP, it can be set on each individual connection. If the FASTCLOSE option is set on a connection by a server and the current state of the connection is CONNECTED and the server now gets an END packet with the REPLY_REQUESTED bit set; what happens is that the server immediately sets the connection TIMED_CLOSE, sends an END packet with the REPLY bit set and starts the CLOSED timer. If the client is in QUICK_CLOSE and gets the END packet with the REPLY bit set, then it sends a FIN packet and goes to FINAL_STATE. Thus the FASTCLOSE option allows client/server connections to be terminated very quickly as shown in Figure 5.

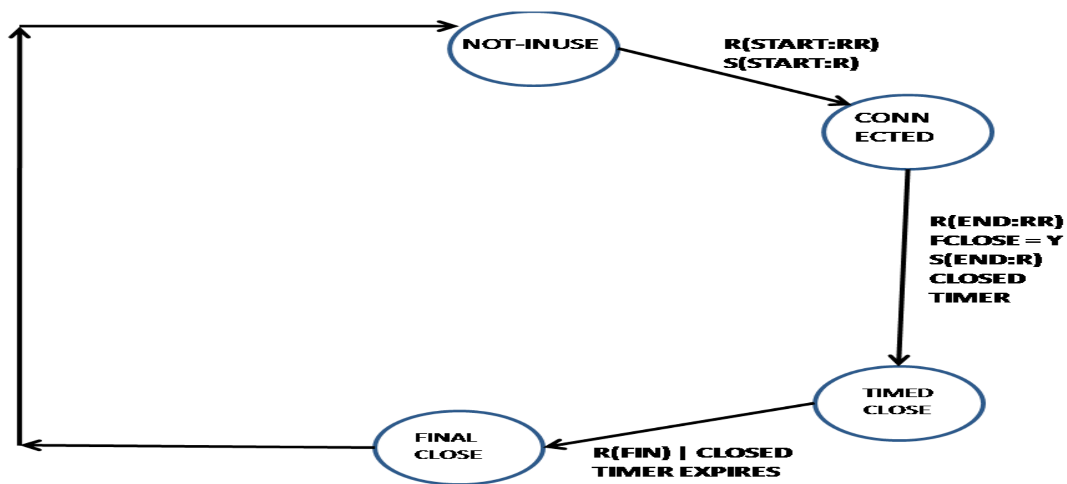


**Figure 5: State Diagram for Fast SP Server**

### TIMERS

There are a number of timers associated with every SP connection:

CONN_TIMER: This is activated when a connection request is sent. When the timer expires the CONN_TIMER packet is resent. This process is repeated 3 times after which the connection is dropped.

ACK_TIMER: This is activated when an acknowledgement is requested. When the timer expires an ACK packet is sent with ACK REPLY_REQUESTED. This process is repeated 3 times after which the outstanding data packets are retransmitted. Then the RETRANS_TIMER is started. When the RETRANS_TIMER expires the process is repeated 3 times then the connection is dropped.

NACK_TIMER: When SP detects that packets are missing, it sends a NACK packet, which indicates the gap of missing packets and then starts the NACK_TIMER. If the NACK timer expires and the correct data has not been received; then the

NACK packet is resent; this is repeated 3 times and then the connection is dropped.

END_TIMER:- This timer is set to ensure that the first END packet is acknowledged. An END packet is regarded as part of the data stream and has a distinct message sequence no.  When one side wants to close the connection, it sends an END_PACKET with a distinct message sequence number and starts the END_TIMER. If the END_TIMER expires then the END packet is resent. This process is repeated 3 times and then the connection is dropped.

CLOSED_TIMER: This timer is set when the connection reaches the TIMED_CLOSE state and thus the END_PACKET has been sent with the REPLY bit set; signalling the end of the connection. When the timer is going in this state, the connection engine must only response to retransmissions of the END packet by the other side (where it retransmits its END packet) or a FIN packet that forces it to move to the FINAL_CLOSE state.

ECHO_TIMER:- This is used to time the end-to-end network latency.  So when an ECHO_TIMER expires, a packet is sent with REPLY_REQUESTED. When the receiving stack gets this packet it simply replies to the packet. Echo packets are periodically sent to determine the latency of a connection.

CWIN_TIMER:- This timer is used to ensure that the other end is aware of a change of the Quality-of-Service in terms of an increased or decreased window size. Like TCP, receiving a Zero Window CWIN packet causes the receiver to stop sending data packets to the other side until another CWIN packet is received with a non-zero window.

STATUS_TIMER: This is activated when a STATUS packet is sent. When the timer expires a STATUS packet is re-sent with REPLY_REQUESTED. This process is repeated 3 times after which the connection is closed.

IDLE_TIMER: In order to make better use of resources, a server can release a connection due to inactivity. To operate this way the server can set a variable known as IDLE_CLOSE on a given connection. This means that when the server has done work on the connection, it sends an IDLE packet to the client with the REPLY_REQUESTED bit set. If the client responds with an IDLE packet with the REPLY bit set, the IDLE TIMER is started. It is stopped if a new data packet is received.  If no DATA packet is received and the IDLE_TIMER expires, then the connection is closed. Also note that when the connection is first set up and the IDLE_CLOSE variable is set, then the IDLE_TIMER is started immediately once the connection is set up. The connection is immediately released if no DATA packet is ever received and the IDLE_TIMER expiries.  This is used to prevent a DDoS attack, which is done by opening lots of inactive connections on the server, such as the SYN attack in TCP.

**A Common Interface**

It is important that SP has a common interface that is used by all applications. The prototype implementation of SP runs over UDP so the maximum size of a SP packet is (64K – 8 bytes (for the UDP header)).

Presently the following application interface is used.

int open_sp(unsigned int dest_IP_address, int conn_type, int max_mtu, int pri)

This call opens an SP connection to a machine with a given IP address. The conn_type is set as follows:

UNRELIABLE: There is no checksum and missing or corrupted packets are not retransmitted
FEC: There is a checksum but missing or corrupted packets are not retransmitted
RTR: There is no checksum but missing packets are retransmitted
RELIABLE: There is a checksum and missing or corrupted packets are retransmitted.

The max_mtu is the maximum message size that will be sent on this connection by the machine requesting the connection. The pri stands for the priority of the connection. This call returns the src_id for this connection that is the handle for this connection on that machine.

int close_sp(int src_id)

This closes a connection.

int recv_buffer(int src_id, struct sp_pckt_q *pcktq, int *status, int recv_flags)

This receives data on a connection. The src_id is the connection handle; the packtq parameter is actually a pointer to a queue of received messages. The status variable is used by the protocol to signal various things to the receiver such as ECN bits being set. The recv_flags can be used to signal specific conditions such as whether the receiver wants to block if there is no data.

int send_buffer(int src_id, struct sp_pckt_b *sbp, int pblock, int tblock, int *status, int send_flags)

This is the basic send call that sends an SP packet on a connection given by src_id. So it is up to the application to break down a message into a number of packets and then call this send_buffer function to send individual packets on that connection. This call assumes that the SP packet is passed in the argument has space for a SP header and data. It fills out the SP header and sends the packet. The send_flags parameter is used to do several things, including requesting an ACKNOWLEGDGEMENT of the data sent and to indicate whether the sender is willing to block if the window for the connection is full.

int change_window_size(int src_id, unsigned int new_window_size)

This call is used to change the window size of a connection. This is can be used to support changes in the Quality-of-Service (QoS) of the connection. By setting the window size to zero, it is possible to allow handovers to be managed.

int measure_echo(int src_id, struct timeval *echo_test, int *test_diff)

This call measures the Round Trip Time between two endpoints using the ECHO packets. The RTT is given in microseconds and is returned in the test_diff variable. The time the echo was done is placed in echo_test. This can be used to measure the RTT between SP running on one machine and SP running on another machine.

In order to provide more support for quality of service between devices, we use the packet pair technique to measure the bandwidth and latency (in this case the RTT) between two machines. We first define the structure used:

```
struct  qos_conn
{
  unsigned int     ip_address;  the address of the remote machine
  int buf_size;      what buffers are being exchanged
  int latency;       the Round Trip Time in microseconds
  int bandwidth;  using packet pair in Kilobytes per second
  int burst;          in bytes: the product of bandwidth by the Round Trip Time
}
```

int measure_echo_buf_bandwidth(int src_id, struct qos_conn *qcn)

This call measures the bandwidth, RTT and burst on an SP connection. Burst is the amount of bytes that would fill the pipe between the sender and the receiver and should be used to specify the maximum amount of unacknowledged bytes for a connection. So when a connection is started, this measure_echo_buf_bandwidth call is used to find the burst for the connection and then sets the MAX_CONGESTION_WINDOW to the burst value. This means that the number of unacknowledged bytes cannot exceed the burst value.

The burst value can also be used to set a steady flow on the connection. This is done by setting a variable called MAX_STATUS_THRESHOLD that will trigger the sending of a STATUS packet. This variable is set to half the burst value. So when the number of unacknowledged bytes is set to MAX_STATUS_THRESHOLD, a STATUS packet is sent with REPLY_REQUESTED and the STATUS_TIMER is started. STATUS packets are retransmitted after the STATUS_TIMER expires. The STATUS timer is turned off when the unacknowledged bytes moves below the MAX_STATUS_THRESHOLD.

The qos_conn structure can also be used by one machine to ask another machine to find out the Quality of Service between the second machine and a third machine that is specified by the ip_address of the qos_conn structure. This mechanism can be used to find out the QoS between a Cloud and a client, allowing a service to migrate to the Cloud to maintain the QoS between a Service and a Client.