RUSP
Reliable User Segment Protocol

Giacomo Marciani

University of Rome Tor Vergata
giacomo.marciani@gmail.com

Abstract. We present the RUSP reliable transport protocol, and its official implementation *librusp*. It is well known that TCP provides a reliable congestion controlled communication at a cost of overhead, whereas UDP provides efficiency at a cost of unreliability and congestion exposure. We would like to fill the TCP/UDP gap, developing an efficient reliable transport protocol for those applications that do not expose the connection end-points to the risk of congestion, thus not requiring neither congestion nor flow control. Still far from being able to aspire to a practice spread, RUSP, together with this paper, are meant to show how to design and realize a reliable transport protocol, alternative, though inspired, to existing transport protocols.

Introduction

In section 1, we introduce the RUSP transport protocol focusing on its reliable service-model and segment structure.

In section 2, we introduce *librusp*, the official implementation of the RUSP transport protocol, focusing on its architecture and API.

In section 3, we give some *librusp* sampling network applications, focusing on a simplified implementation of the FTP.

In section 4, we show the experimental results, proving pros/cons and performances of the protocol design and its implementative model.

In section 5, we state the future improvements of the RUSP protocol and *librsup*.

1 The RUSP Protocol

The Reliable User Segment Protocol (RUSP) is a reliable transport protocol that realizes a connection-oriented point-to-point full-duplex unbounded[1] communication.

Although clearly inspired by both the TCP, defined in [3], and the UDP, defined in [4], RUSP significantly differs from both of these protocols. Actually,

[1] RUSP does not impose any kind of transmission rate constraint, whereas the TCP's congestion control necessarily imposes it.

RUSP aims to fill the well-known gap between TCP and UDP, providing an efficient reliable service for those network applications that do not need neither congestion nor flow control.

In this section we will show the main building blocks that realize the RUSP reliable service, the RUSP segment structure and the pseudo-code of both the send and receive side of the RUSP connection.

In particular, we will focus on the RUSP finite state automaton, the connection establishment and shutdown, the SACK/CACKnoledgment discipline, the ISN computation, the sliding-window and the adaptive retransmission timeout.

1.1 The Reliable Service

A transport protocol is said to be reliable, or it realizes a reliable service, if and only if its service-model guarantees a correct in-order delivery of encapsulated data. To achieve this, the protocol must keep track of some useful communication-state information, thus implementing a connection-oriented service.

The Finite State Automaton When designing a connection-oriented protocol, the design of the connection's finite state automaton (FSA) is the first topic to take into account. So, let us now show the RUSP FSA, wich is clearly inspired by the TCP FSA, defined in [3].

The only difference with respect to the TCP's consists on the absence of the mutual transition between SYNSND and SYNRCV, and the presence of the transition from FINWT1 to TIMEWT, taken from [2].

The Connection Establishment The connection establishment allows the two connection end-points to synchronize their sequence number space. RUSP implements the connection establishment in the same way TCP does: thus realizing it by a *three-way handshake* procedure, during wich both the connection end-points exchange their respective *Initial Sequence Number (ISN)*. Figure 2 shows the usual RUSP three-way handshake.

ISN Computation The connection establishment imposes to both the end-points to respectively exchange their locally run-time computed ISNs, thus realizing the binding of the sequence spaces. Such a mechanism exposes the connection to an *ISN-guessing attack*. The pseudo-random ISN computation should then reduce the exposure to such an attack.

As stated in [9], we can prevent ISN-guessing attacks by giving each connection a *separate parametrized sequence space*. Let us identify each connection with the following tuple

$$Connection = (ip_{local}, port_{local}, ip_{peer}, port_{peer}) \tag{1}$$

and give a pseudo-random function $F(C, K)$ of the connection C, parametrized by some secret key K. Within each space, the ISN is incremented by $M(t)$ in a

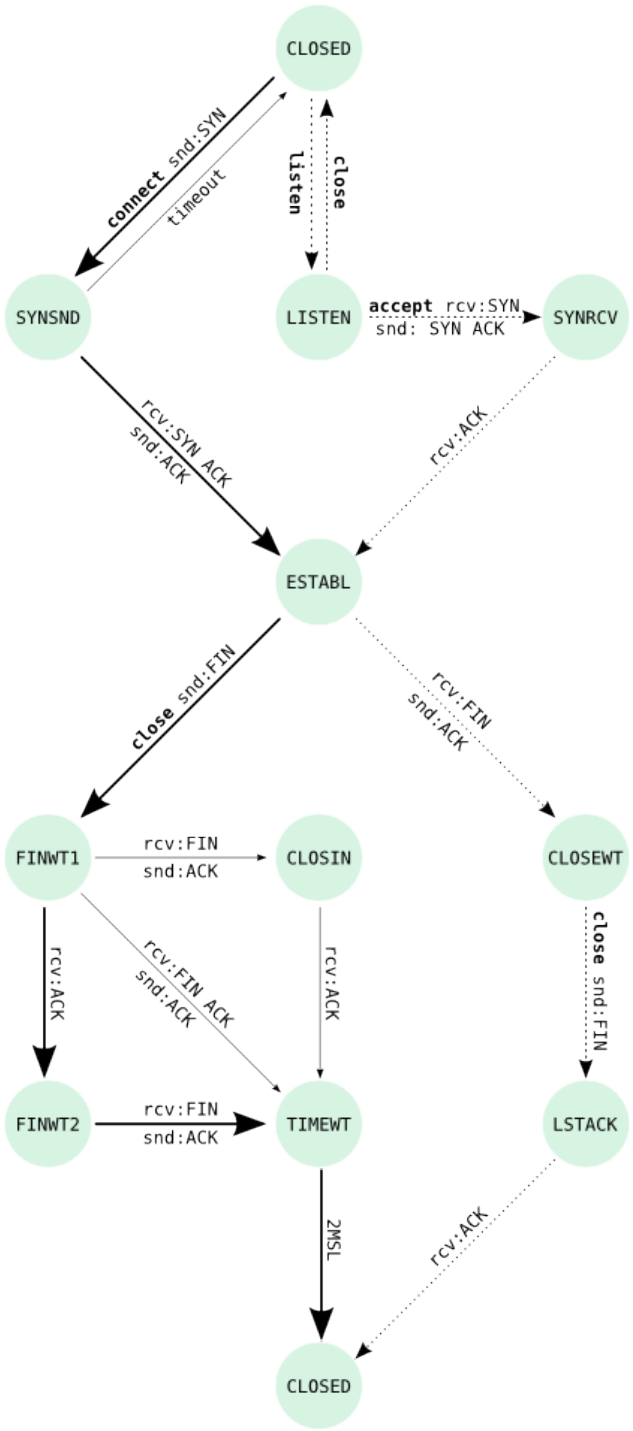

Fig. 1. RUSP Finite State Automaton

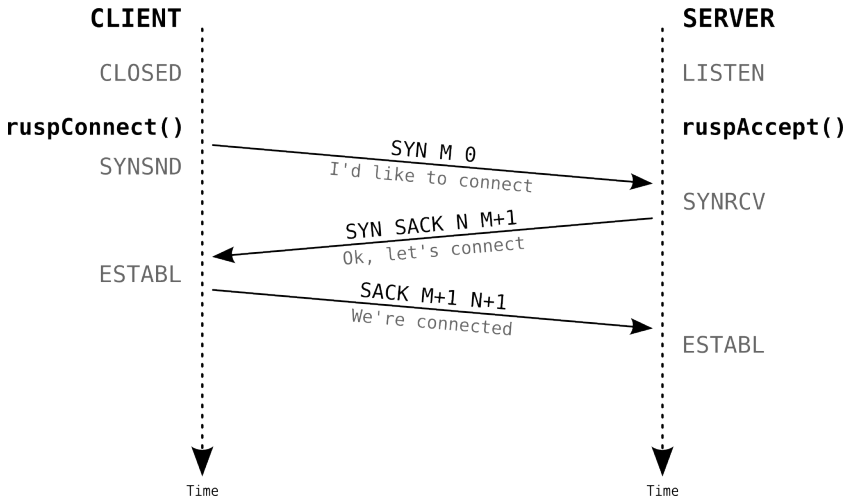

Fig. 2. Connection Establishment: usual case

time-based fashion, according to [3]. As suggested in [9], the MD5 cryptographic hashing, described in [10], would be a good choice for the implementation of the F function.

The RUSP algorithm for the ISN computation implements the following function

$$ISN = M(t) + MD5(C, K) \tag{2}$$

Interesting Scenarios Let us now consider an interesting scenario. Figure 3 shows how RUSP recovers from segment loss during the connection establishment. Note that in Figure 3 was omitted the FIN SACK's loss: the loss has been deriberately omitted due to the fact that the current version of the protocol does not provides any procedure to recover from this kind of loss. See section 4 for futher details about FIN SACK's loss recovery.

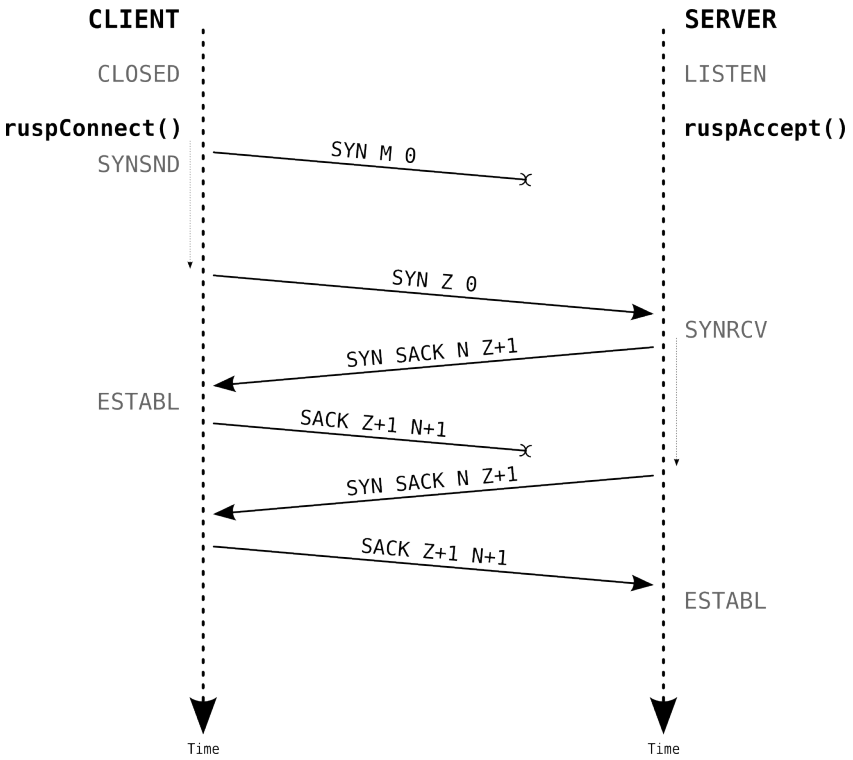

Fig. 3. Connection Establishment: segment loss recovery

The Connection Shutdown The connection shutdown allows the two connection end-points to gently close the connection. RUSP implements the connection shutdown in the same way TCP does: thus realizing it by a *four-way handshake* procedure.

Before going into details about the segments exchange involved in the procedure, we first need to distinguish two kind of connection shutdown: the *active close* and the *passive close*. The connection shutdown is initiated by an active close, and terminated by a passive close. Futhermore, RUSP requires an extra-state TIMEWT to reliably complete the connection shutdown on the active close side, as TCP does.

Notice that, as stated in [3] and shown in Figure 4, forwarding a connection close request, both the active and the passive one, does not sound as an imposition to shutdown, but a declaration to have no more data to send.

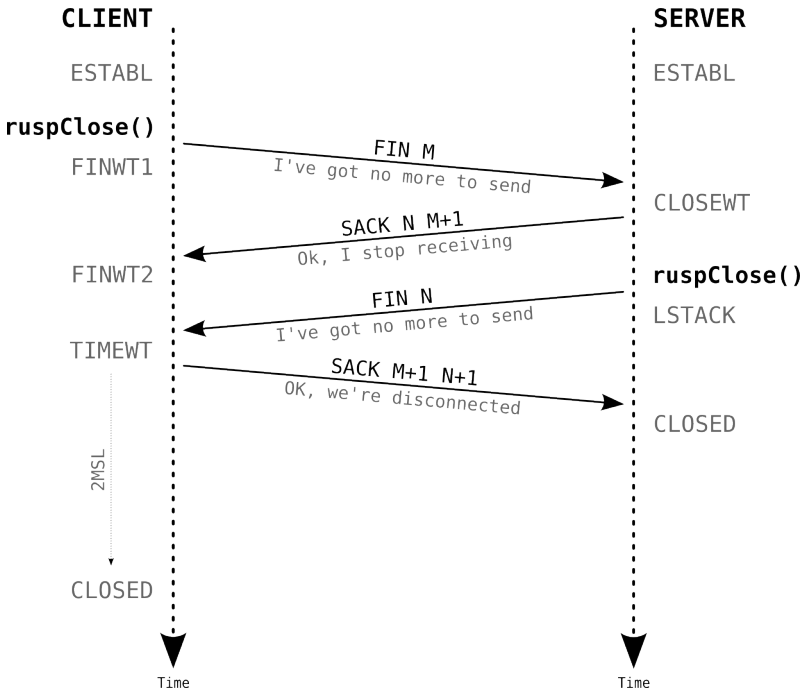

Fig. 4. Connection Shutdown: usual case

TIMEWT State Let us now give a short clarification about the TIMEWT state. The purpose of the TIMEWT state is to implement reliable connection termination and to allow expiration of old duplicate segments in the network. When opening a new connection, RUSP takes care to avoid new incarnations of connections in TIMEWT state. We notice that there is not an equivalent state for

the passive close side of the connection: in fact, it does not require it because it can't be exposed to duplicate segments storm. As stated in [3], the transition from TIMEWT state to the CLOSED state is taken in response of a timeout that is twice the *Minimum Segment Lifetime (MSL)*. Although BSD socket implementation assumes a 30 seconds MSL, we decided to mediate this value with the 120 seconds MSL suggested in [11]: thus considering a 75 seconds MSL and 150 seconds TIMEWT timeout.

Interesting Scenarios Let us now consider two interesting scenarios. Figure 5 shows how RUSP recovers from simultaneous active close requests, while Figure 6 shows how RUSP recovers from segment loss during the connection shutdown. In this latter scenario we may appreciate both the piggy-backed acknowledgement and the RUSP's FSA transition from state FINWT1 to state TIMEWT.

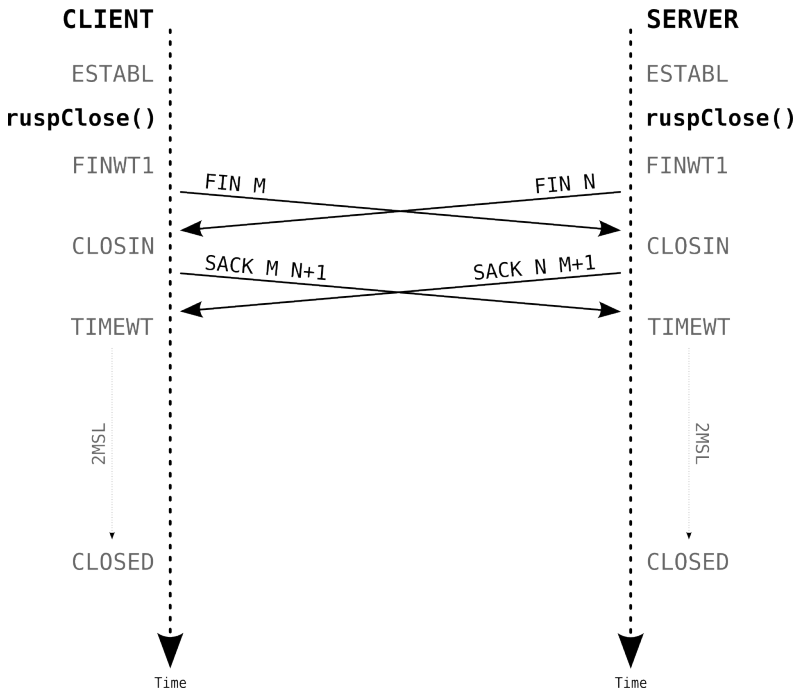

Fig. 5. Connection Shutdown: simultaneity

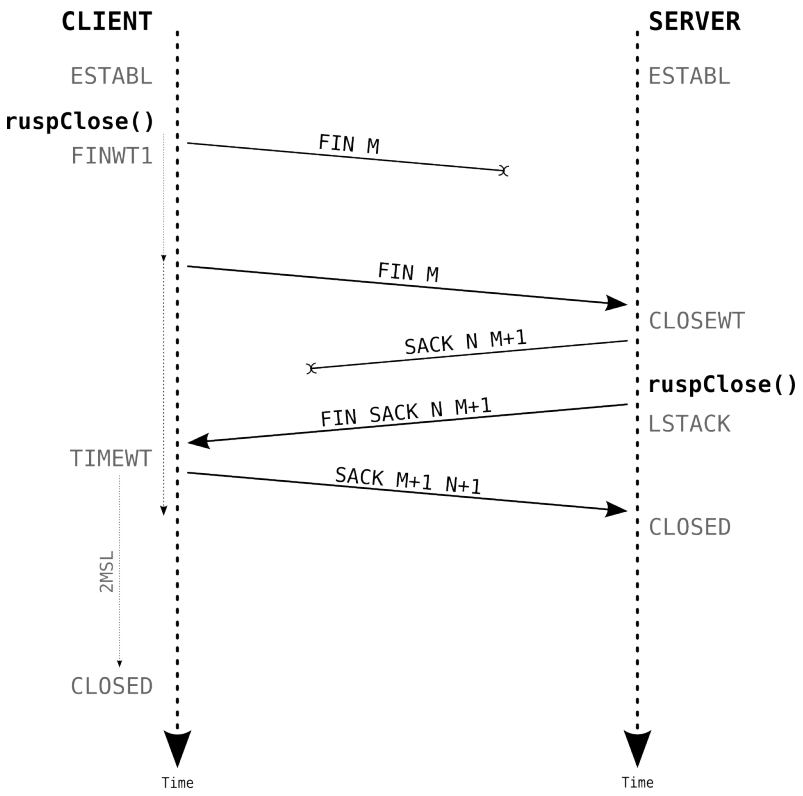

Fig. 6. Connection Shutdown: segment loss recovery

The Acknowledge Discipline RUSP employs an aknowledgement discipline that takes advantage of both the selective acknowledgement and cumulative acknowledgement.

As stated in [1], the selective acknowledgement, provided by Selective-Repeat, minimizes redundant retransmissions with respect to the cumulative acknowledgement provided by the Go-Back-N.

RUSP employs a *SACK* segment (selective acknowledgement) to acknowledge a first-time received segment, while employs a *CACK* segment (cumulative acknowledgement) to acknowledge a redundant retransmission. See the next paragraphs about the sliding-windows, to better understand the power of such a mixed acknowledgement discipline.

Figure 7 shows a usual data communication scneario, in wich RUSP employs SACKs to acknowledge first-time received segments.

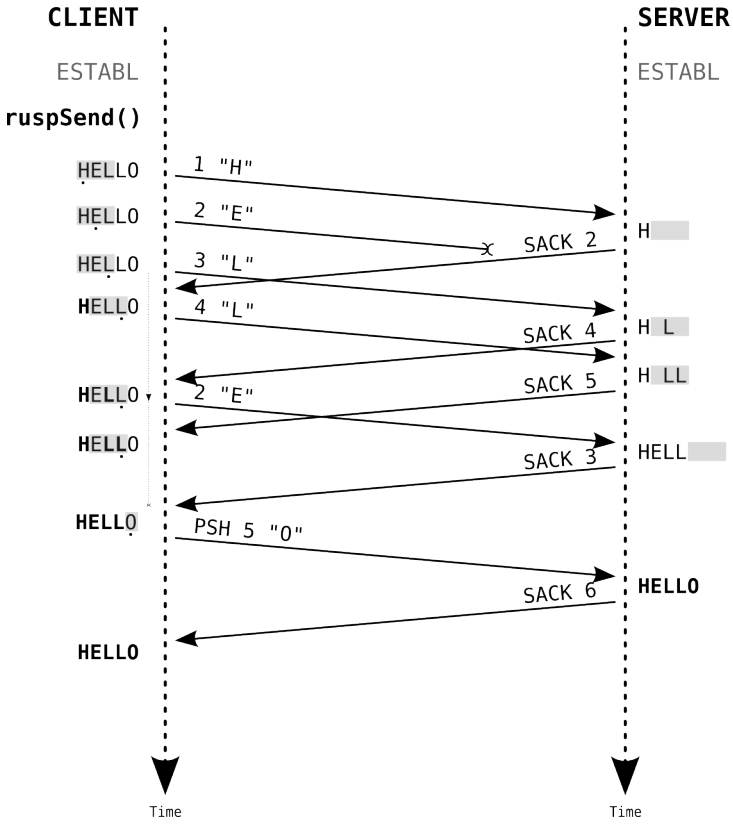

Fig. 7. Acknowledge Discipline: usual case

Interesting Scenarios Let us now consider an interesting scenario. Figure 8 shows a typical scenario in wich RUSP takes advantage of CACK to immediately restore windows allignment after a storm of SACKs loss.

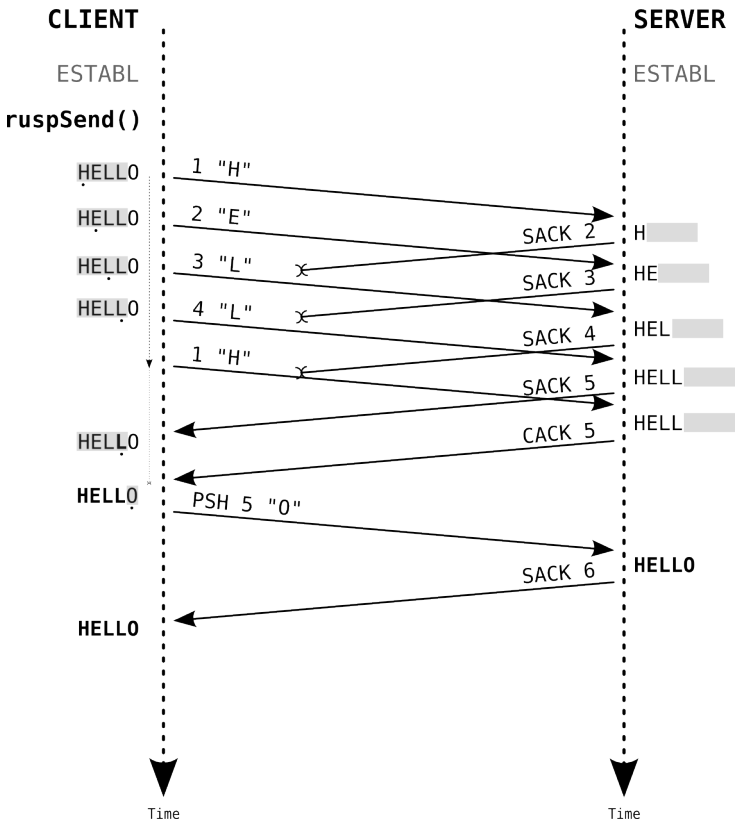

Fig. 8. Acknowledge Discipline: CACK advantages

The Sliding-Window RUSP is a *sliding-window protocol*, in the sense stated in [1]: each outcoming segment is processed with respect to the outbox window, *sndwnd*, while each incoming segment is processed with respect to the inbox window, *rcvwnd*. Let us convince you that this mechanism, togheter with the acknowledge discipline, realizes the reliable service.

Oubox Window The *sndwnd* partitions the sequence space of the outcoming segments. The sender shall send all those segments that fall within the sndwnd: the window size, *wnds*=*wnde*-*wndb*, thus imposes an upper bound on the number of unACKed segments in pipeline. The window base, *wndb*, is the sequence number of the oldest unACKed segment. The window next, *wndn*, is the sequence number of the next segment to send. The window end, *wnde*, is the sequence number of the first out-of-window segment, thus the first unsendable segment. Once *wndb* has been ACKed, the *sndwnd* shall slide to the next unACKed sequence number, thus allowing the sender to reliably proceeed its transmissions.

Figure 9 clearifies the above concepts.

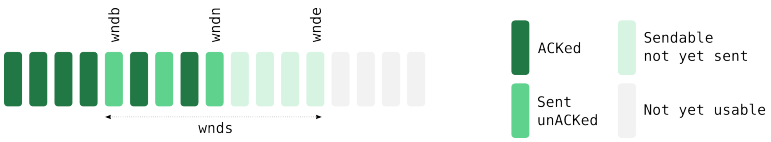

Fig. 9. Outbox Sequence Space: *sndwnd*

Inbox Window The *rcvwnd* partitions the sequence space of the incoming segments. The receiver shall receive all those segments that fall within the rcvwnd: the window size, *wnds*=*wnde*-*wndb*, thus imposes an upper bound on the number of bufferable segments. The window base, *wndb*, is the sequence number of the next expected in-order segment. The window end, *wnde*, is the sequence number of the first out-of-window segment, thus the first unbufferable segment. Once *wndb* has been received, the *rcvwnd* shall slide to the next not received sequence number, thus allowing the receiver to reliably proceeed its receptions.

Figure 10 clearifies the above concepts.

Fig. 10. Inbox Sequence Space: *rcvwnd*

Local Window and Peer Perception Let us now point out a primary key-point in the understanding of a sliding-window protocol. The local outbox[inbox] window does not only provides an effective mechanism to regulate local transmission[receptions], but also gives a local approximated reflection of the peer's inbox[outbox] window. Clearly, such an approximation is sensitive to segment loss: a single SACK loss is enough to misalign the local *sndwnd* with respect to the peer *rcvwnd*. Such a misalignment may cause serious communication lacks: indeed, the sender may waste time and resources congesting the network and peer with redundant retransmissions. The RUSP's CACK acknowledgement allows the sender to restore the window allignment as soon as the first redundant retransmission has been received by the peer.

Windows and Buffers The window size cannot be unlimited. A first upper bound is imposed by the modulo-2 arithmetic of the sequence space. The window size must be less than or equal to $\frac{2^{32}}{2} - 1$ to guarantee the total ordering of segments. A second upper bound, typically more severe than the previous one, is imposed by the size of the receving buffer. If no flow control is provided, the window size must be strictly less than the size of the receiving buffer, to avoid the possibility of a buffer overflows. As RUSP does not provide any flow control, it thus requires window size less than 65535 bytes, which is the receiving buffer size.

The Retransmission Timeout RUSP employs an adaptive time-based retransmission mechanism, thus it should pay particular attention to the timeout computation.

Clearly, the timeout should be larger than the connection's Round Trip Time (RTT), but how much larger? Indeed, a too short timeout would induce unnecessary retransmission, while a too large timeout would lead to a serious performance lack in case of network congestion.

Ideally, the timeout must be istantaneously adapted to the expected RTT. Such a predictive model must take into account the current network congestion and the variability trends of its fluctuations.

As stated in [8], such a scenario can be effectively approximated by a Exponential Weighted Moving Average (EWMA) statistical model, thus obtaining the following relations

$$extRTT = (1 - \alpha) \cdot extRTT + \alpha \cdot samRTT \tag{3}$$

$$devRTT = (1 - \beta) \cdot devRTT + \beta \cdot \mid samRTT - extRTT \mid \tag{4}$$

$$timeout = extRTT + \gamma \cdot devRTT \tag{5}$$

where $samRTT$ is the RTT measured for the acknowledgement of a non retransmitted segment, $extRTT$ is the weight average of the measured RTTs, $devRTT$ is the weight average of the deviations between the expected and measured RTTs, and *Timeout* is the current timeout value. Such an adaptive model let

the protocol to compute a new timeout value approximately once every SACKed segment.

As stated in [8], the reccomended values of α, β and γ are, respectively, $\alpha = 0.125$, $\beta = 0.25$ and $\gamma = 4$. Such values of α and β induce a greater weight on recent samples and deviations with respect to the old ones, while such a value of γ induces a timeout margin directly proportional to the weight of fluctuations. As reccomended in [8], the initial timeout value is set to 1 second. Figure 11 plots the bahaviour of *extRTT* *devRTT* ad *Timeout*, letting us appreciate the advantages of such a predictive model. Indeed, it highlights the initial exploration of the RTT variability space and its subsequent adaptations.

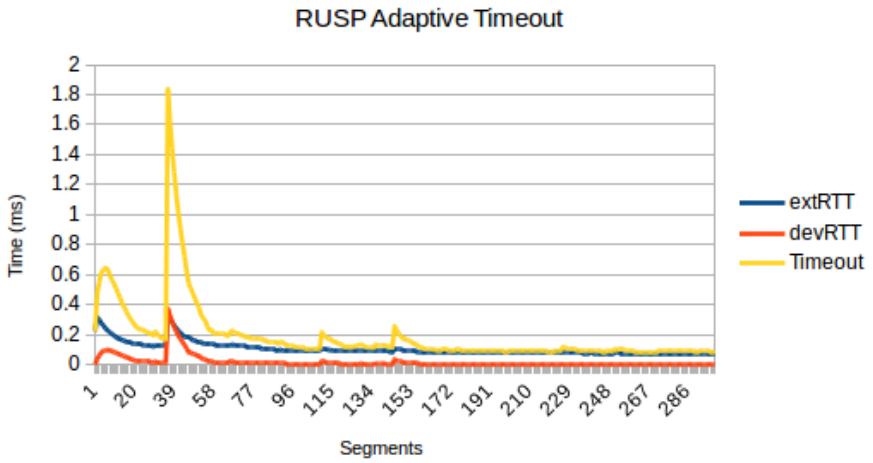

Fig. 11. Adaptive Timeout Behaviour

1.2 The RUSP Segment Structure

RUSP allows two processes to communicate through segments exchange, thus encapsulating upper-layer data and control data into a RUSP segment. Figure 12 shows the general RUSP segment structure.

- source/destination port (16 bits each): used for multiplexing/demultiplexing data from/to the application layer.
- checksum (8 bits): used for segment content error detection. It is computed on the sender side as the 1's complement of the sum of all the 16-bit words in the segment, with the wrapping around of any encountered overflow. As some UDP implementations do, RUSP does not provide any procedure to recover from a detected error, it simply discards every corrupted segments.

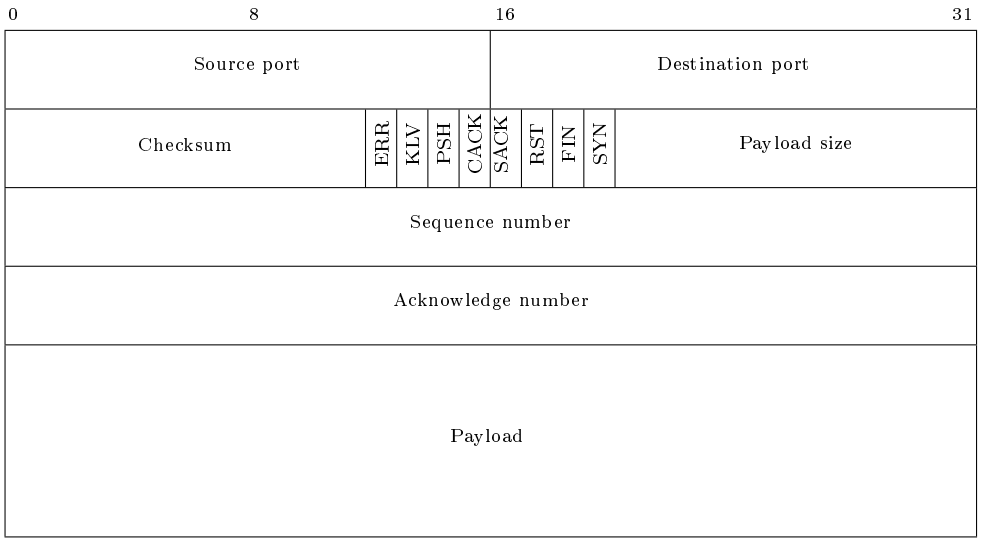

Fig. 12. RUSP Segment Structure

- control flags (8 bits): characterizes the segment's purpose.
  - SYN: used for connection establishment.
  - FIN: used for connection shutdown.
  - RST: used for connection establishment recovery.
  - SACK: used for selective acknowledgement.
  - CACK: used for cumulative acknowledgement.
  - PSH: used to push data to the upper layer.
  - KLV: used to keep-alive the peer.
  - ERR: used for local error notifications.
- length (16 bits): indicates the payload size.
- sequence/acknowledgement number (32 bits each): used for the implementation of the reliable data service.

Header Overhead RUSP minimizes the header overhead, with respect to TCP: indeed, it has only 16 bytes of overhead in every segment, whereas the TCP segment has a 40 bytes of overhead.

Note about the current implementation The current RUSP implementation provided by *librusp* builds the protocol on top of UDP, thus imposing, for now, some segment structure modifications. The source/destination port fields and the checksum field are included into the UDP header; moreover, the UDP header provides a 16-bits long length field and a 16-bits long checksum field instead of the above declared 12-bits. Futhermore, the current release of *librusp* does not make use of RST, KLV an ERR flags, See section 4 for futher details about future improvements.

1.3 The Sender/Receiver Pseudo-Code

Sender-Side The sender waits for data, loaded into the user buffer from the upper layer. As data has been detected, it creates a new segment with the maximum amount of data provided by the protocol, then waits for the necessary send-side sliding window's space. As the sliding window permits it (thus, when send-base has been acked), the sender buffers the new segment and sends it, then slides the send-next pointer. At the end, it pops the sent data from the user buffer. Algorithm 1 shows the pseudo-code of the RUSP sender.

Receiver-Side The receiver performs a timeout listen for incoming segments. If a segment is received, this segment is matched against the receive-side sliding window: an after-window segment is always ignored, a before-window segment is always CACKed, while an inside-window segment is always SACKed and processed. In case of an inside-window segment, its ACK field is always appropriately submitted to the send-side segment buffer (considered as SACK, if the SACK field is flagged, considered as CACK, if the CACK file is flagged). If the segment matches the receive-side window-base, then it is processed as an in-order segment, and the receive-side window is slided. Of corse, in case of a receive-side window-base match, the receive-side segment buffer is opportunely processed, looking for the next base. If the segment does not match the receive-side window-base, then it is buffered. If the timeout expires before receiving a segment, the sender induces the retransmission of the opportune unacknowledged segments. In the case of peer disconnection, the sender induces the connection shutdown. Algorithm 2, shows the pseudo-code of the RUSP receiver.

Algorithm 1 Sender Pseudo-code

```
while true:
 waitDataFromUserBuff;
 createSegment;
 waitSndWindowSpace;
 addToSndSegmentBuff;
 sendSegment;
 slideSndWindowNext;
 popDataFromUserBuff;
```

2 The *librusp* Library

The *librusp* is the official C static library that implements the RUSP transport protocol. As a free open source project, it can be consulted and downloaded from

http://github.com/giacomomarciani/rusp.

Algorithm 2 Receiver Pseudo-code

```
while true:
 receiveSegment;
 if not received:
  timeoutFunc;
  continue;
 if disconnected:
  closeConnection;
  break;
 if segmentInsideWindow:
  sendSAck;
  submitAck;
  if segmentIsRcvWindowBase:
   processBase;
   slideRcvWindowBase;
   while rcvBaseInSegmentBuff:
    processBase;
    slideRcvWindowBase;
  else:
   addSegmentToSegmentBuff;
 else if segmentBeforeRcvWindow:
   sendCAck;
 else:
   doNothing;
```

If you would like to build your own RUSP-based network application, you first need to install *librusp* on your UNIX system running the make utility, as usual, then link it by specifying the linkage directive *-lrusp*.

In this section we are going to show the library's architecture and the functions provided by the library's API.

2.1 The Architecture

The *librusp* arcitecture builds the RUSP protocol on top of UDP, realized by the BSD API on SOCK_DGRAM socket layer. The architecture consists of an API layer that allows the network application to interface to the Core layer, wich realizes the RUSP reliable service-model.

As RUSP is a connection-oriented protocol, the main architecture's entity is the Connection, wich is references and managed inside a Connections Pool. A Connection is a thread-safe structure that implements the RUSP FSA, employing data buffers, segments buffer and sliding-windows to realize the reliable service. The following Figure 13 shows the RUSP architecture model.

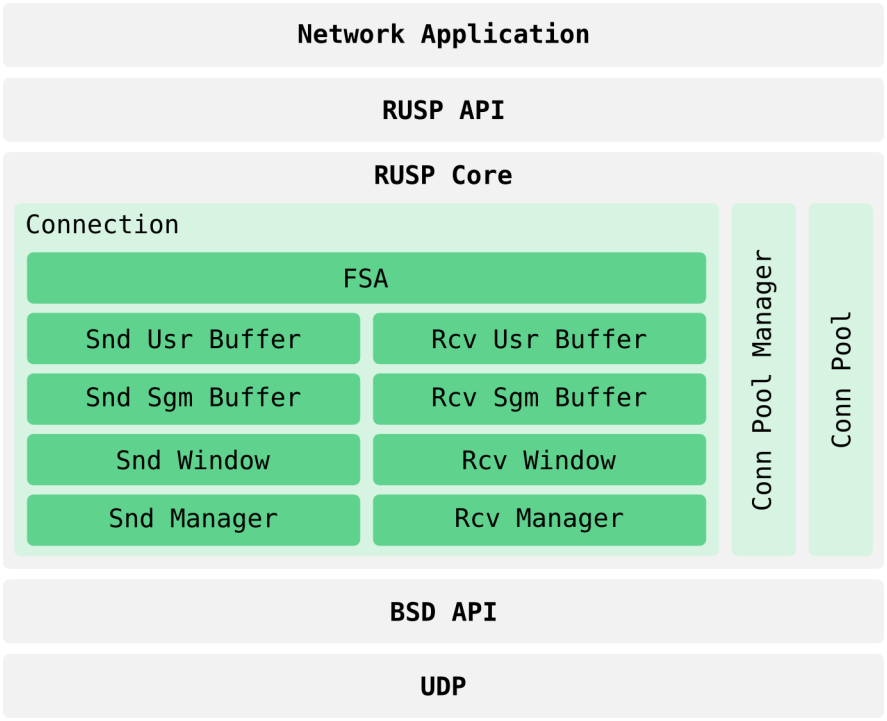

Fig. 13. RUSP Architecture

2.2 The API

The *librusp* API is mainly inspired by the BSD API's system calls used in SOCK_STREAM sockets context. From the application layer perspective, a *librusp* connection is nothing more than a positive integer, thus allowing an easy connection referenciation in every call. Futhermore, the *librusp* API exposes a little set of accessible developer utilities.

Let us now show the *librusp* API calls for the connection establishment *ruspListen()*, *ruspAccept()*, *ruspConnect()*, connection shutdown *ruspClose()* , connection I/O *ruspReceive()*, *ruspSend()* and connection end-points identification *ruspLocal()*, *ruspPeer()*. At the end of this section we will show the *librusp* developer utility calls *ruspSetAttr()*, *ruspGetAttr()*.

Listening for Incoming Connections The *listen()* function creates a new listening connection, binding it to the specified port number. The connection can be subsequentely used to accept incoming connection.

```
#include <rusp.h>

ConnectionId ruspListen(const int port);

                                    Returns a positive connection id, or -1 on error.
```

The function takes *port* as the local port number where the new listening connection will be bound to, and returns the connection id of the newly created listening connection.

The current version of *librusp* does not provide any kind of *backlog* management nor any kind of its optimization, but makes up for this lack by ensuring an expansion of the synchronization timeout and a failure-tolerance mechanism based on automatic resynchronization. See Section 4 for future improvements about listening backlog.

Accepting a Connection The *accept()* function lets a listening connection accept an incoming connection.

```
#include <rusp.h>

ConnectionId ruspAccept(ConnectionId connid);

                                    Returns a positive connection id, or -1 on error.
```

The *connid* is the id of a listening connection, through wich we wish to accept the next incoming connection. If there is no pending connection, the function blocks until the synchronization handshake has been successfully completed, then returns the id of the newly established connection. Once established, the new connection can be used to perform I/O operations, or can be closed. Any

attempt to accept incoming connections on a not listening connection will always fail.

Connecting to a peer The *connect()* function creates a new established connection with the peer listening on the specified address.

```
#include <rusp.h>

ConnectionId ruspConnect(const char *ip, const int port);

                                    Returns a positive connection id, or -1 on error.
```

The function takes *ip* and *port*, respectively, as the address and port number of the peer we wish to establish the connection to, and return the newly created established connection, or -1 in case of connection failure. In case of connection failure, nothing more than calling *ruspConnect()* once again is needed. Once established, the connection can be used to perform I/O operations, or can be closed.

Closing a Connection The *close()* function closes the specified connection.

```
#include <rusp.h>

void ruspClose(ConnectionId connid);
```

The *connid* is the id of the connection to close. The function blocks until the shutdown handshake has been successfully completed.

Once a connection has been closed, any futher attempt to accept or perform an I/O operation will always fail. Futhermore, any attempt to create an incarnation will fail during all the time necessary to its TIMEWT state.

Remember that, every open connection, either a listening or an established one, strictly needs to be closed to release all the associated system resources.

Connection I/O The *ruspReceive()* and *ruspSend()* functions allow the data I/O through an established connection.

```
#include <rusp.h>

ssize_t ruspReceive(ConnectionId connid, char *msg, size_t msgs);

                    Returns number of bytes received, 0 on EOF, or -1 on error.

ssize_t ruspSend(ConnectionId connid, char *msg, size_t msgs);

                                    Returns number of bytes sent, or -1 on error.
```

For both functions, *connid* is the id of the established connection we want to perform the I/O, *msg* is the message buffer, and *msgs* is the message buffer size, thus the maximum message size allowed for the current I/O.

For both function, the return value is the factual number of bytes involved in the current I/O, or -1 if an error occurred during the current I/O . Futhermore, the *ruspReceive()* function provides an easy way to detect the peer intention to stop sending data and to close the connection: indeed when the peer actively closes the connection, every local attempt to receive data through the connection will return EOF.

These functions let us perceive the advantage of a connection model built on top of a connected datagram socket. Indeed, we can use a simpler I/O model when trasmitting data on the connection: we don't need to specify neither the destination address nor its size, anymore. This I/O model is primarly useful in an application that needs to send multiple datagrams to a single peer, wich is the typical scenario in a client-server model implementation. Futhermore, as stated in [2], on some TCP/IP implementations, connecting a datagram socket to a peer yields an I/O sensitive performance improvement.

Addresses The *ruspLocal()* and *ruspPeer()* functions allow to respectively retrieve the address of the specified connection end-points.

```
#include <rusp.h>

int ruspLocal(const ConnectionId connid, struct sockaddr_in *addr);

                                                Returns 0 on success, -1 on error.

int ruspPeer(const ConnectionId connid, struct sockaddr_in *addr);

                                                Returns 0 on success, -1 on error.
```

For both functions, *connid* is the id of the connection we want to know the end-points addresses, and *addr* is the argument-value address requested.

Notice that the *ruspLocal()* function will always succeed when called on a not closed connection: indeed, every not closed connection has an underlying open datagram socket, that is, every not closed connection is associated to a local address. On the other hand, the *ruspPeer()* function will always fail when called on a not established connection: indeed, only an established connection has an underlying open and connected datagram socket, that is, only an established connection is associated to a specific peer address.

Developer utilities The *ruspSetAttr()* and *ruspGetAttr()* functions allow to respectively set and retrieve *librusp* attributes, such as segment drop rate and debug mode.

```
#include <rusp.h>

int ruspSetAttr(const int attr, const void *value);
int ruspGetAttr(const int attr, const void *value);

                                        Returns 0 on success, -1 on error.
```

For both functions, *attr* is the name of the attribute to set or retrieve and *value* is the argument-value of the attribute value to set or retrieve. Let us now show some get/setable *librusp* attributes.

RUSP_ATTR_DROPR allows to set the uniform probability of segment drop. As a reliable transport protocol, nothing is more important than valuate its reaction to segments loss and local segment drop. A non-zero drop rate provides an easy way to simulate these transmission lacks. Every RUSP session comes with a *drop* default value of 0.0, that is no segment drop imposition.

RUDP_ATTR_DEBUG allows to activate or deactivate the debug mode. As a case-study protocol, nothing is more important than exploring what is happening under the hood. The debug mode provides an easy way to visualize the real-time segment exchange. Every RUSP session comes with a deactivated debug mode, that is no segment exchange visualization.

3 Some *librusp* Network Applications

librusp comes with a little set of sampling client-server network applications, that can be executed from */bin* and whose source code can can be found in */samples*. Such network applications are meant to demonstrate the easy usage of the library, allowing to evaluate its performances in common real scenarios and giving some general guidelines for the design of simple network applications. Moreover, the library provides an utility for sample file generation (*samplegen*), thus making file transmission tests easy and immediate.

In the following paragraphs, we show these applications, describing their functionalities and giving some sampling output of their execution. Clearly, every application provides both a *-h* option for usage helper, and a very useful *-d* option for debug mode activation, thus showing what is happening under the hood.

3.1 Client-Server Model

We now show an implementative example of a multi-tasking client-server model, based on *librusp*. The considered scenario is the typical service provided by a concurrent echo server. The client and the server both employ the header file shown in Algorithm 3, wich defines the server's address and port number and the maximum size of exchangeable messages. The server and the client are respectively implemented by the algorithm shown in Algorithm 4 and Algorithm 5

Algorithm 3 common.h

```
#include <rusp.h>

#define ADDR "192.168.1.121" //The server address

#define PORT 55000           //The server port

#define MSGS 1024            //The maximum message size
```

Server-Side The server opens a connection (*lconn*) listening on port 55000, allowing the acceptance of incoming concurrent connections. As a new connection request arrives, the server accepts it by allocating the service process that will communicate with the client through the new connection (*aconn*). At this point, the listening process no longer needs the active connection, as well the service process will no longer needs the listening connection: so they close these redundant connections in their own process space.

From now on, the listening process will continue its acceptance cycle, whereas the service process will echo the received data, as long as the client does not require the active close of the connection.

When the client request the active close of the connection, the service process will be notified by an EOF on its reading attempt, and proceeds to the passive close of the connection, thus finishing its service cycle .

Client-Side The client actively opens a connection (*conn*) with the server, on the known ip and port. Once established, as long as the user types something on stdin, the input (snddata) will be sent to the server, then blocks until receiving the server response (*rcvdata*). When the client has no more data to send, it proceeds with the active close of the connection, waiting for the passive close from the server side. Once the passive close has been received, the connection will be definetedly closed.

3.2 RTT and Bitrate

The following applications allow to evaluate the network RTT and the transfer bitrate.

ECHO The *echo* application realizes the well-known Echo Protocol, defined in [12]. Once a connection is established any data received is sent back, until the client terminates the connection. This simple network application is very useful to evaluate the network RTT. Moreover, the activated debug mode let us appreciate the segments exchange, the connection state transitions and buffer management. Let us see an execution example: Figure 14 shows the client's execution output, while Figure 15 shows the server's.

Algorithm 4 server.c

```
#include <common.h>

int main(void) {
 ConnectionId lconn, aconn;
 char rcvdata[MSGS]
 ssize_t rcv;

 lconn = ruspListen(PORT);

 while(aconn = ruspAccept()) {
  switch (fork()) {
  case 0:
   ruspClose(lconn);
   while ((rcv = ruspReceive(aconn, rcvdata, MSGS)) > 0)
    ruspSend(aconn, rcvdata, rcv);
   ruspClose(aconn);
   break;
  dafault:
   ruspClose(aconn);
   break;
  }
 }

 exit(EXIT_SUCCESS);
}
```

Algorithm 5 client.c

```
#include <common.h>

int main(void) {
 ConnectionId conn;
 char snddata[MSGS], rcvdata[MSGS];
 ssize_t snd, rcv;

 conn = ruspConnect(ADDRESS, PORT);

 while ((snd = getUserInput("[SND]>", snddata, MSGS)) > 0) {
  ruspSend(conn, snddata, snd);
  if ((rcv = ruspReceive(conn, rcvdata, MSGS)) > 0)
   printf("[RCV] %.*s\n", (int)rcv, rcvdata);
  else
   break;
 }

 ruspClose(conn);

 exit(EXIT_SUCCESS);
}
```

./echoc 192.168.1.121 -p 55000 -d

```
[SGM ->] 13:25:10:187812 dst: 192.168.1.121:55000 ctrl:1 plds:0 seqn:0 ackn:0
STATE: CLOSED -> SYNSND
[<- SGM] 13:25:10:189657 src: 192.168.1.121:50132 ctrl:9 plds:0 seqn:10 ackn:1
STATE: SYNSND -> SYNRCV
[SGM ->] 13:25:10:190763 dst: 192.168.1.121:50132 ctrl:8 plds:0 seqn:1 ackn:11
STATE: SYNRCV -> ESTABL

WELCOME TO ECHO CLIENT
Running on 192.168.1.121:51654 Connected to 192.168.1.121:50132

[INPUT (empty to disconnect)]>Hello World!

[SGM ->] 13:25:14:962519 dst: 192.168.1.121:50132 ctrl:40 plds:12 seqn:1 ackn:11 Hello
World!
SND (NXT): base:1 nxt:13 end:4001 SNDUSRBUFF:12 SNDSGMBUFF:1
[<- SGM] 13:25:14:962986 src: 192.168.1.121:50132 ctrl:8 plds:0 seqn:11 ackn:13
SACKED: 1
SND (WND): base:13 nxt:13 end:4013 SNDUSRBUFF:0 SNDSGMBUFF:0
INSIDE RCVWND: base:11 end:4011 RCVUSRBUFF:0 RCVSGMBUFF:0
IS RCVWNDB: 11
[<- SGM] 13:25:14:963964 src: 192.168.1.121:50132 ctrl:40 plds:12 seqn:11 ackn:13 Hello
World!
INSIDE RCVWND: base:11 end:4011 RCVUSRBUFF:0 RCVSGMBUFF:0
[SGM ->] 13:25:14:964133 dst: 192.168.1.121:50132 ctrl:8 plds:0 seqn:13 ackn:23
IS RCVWNDB: 11
RCV (WND): base:23 end:4023 RCVUSRBUFF:12 RCVSGMBUFF:0

[RCV] Hello World!

[INPUT (empty to disconnect)]>

STATE: ESTABL -> FINWT1
[SGM ->] 13:25:16:210205 dst: 192.168.1.121:50132 ctrl:10 plds:0 seqn:13 ackn:23
SND (NXT): base:13 nxt:14 end:4013 SNDUSRBUFF:0 SNDSGMBUFF:1
[<- SGM] 13:25:16:210464 src: 192.168.1.121:50132 ctrl:8 plds:0 seqn:23 ackn:14
SACKED: 13
SND (WND): base:14 nxt:14 end:4014 SNDUSRBUFF:0 SNDSGMBUFF:0
INSIDE RCVWND: base:23 end:4023 RCVUSRBUFF:0 RCVSGMBUFF:0
IS RCVWNDB: 23
STATE: FINWT1 -> FINWT2
[<- SGM] 13:25:16:211262 src: 192.168.1.121:50132 ctrl:10 plds:0 seqn:23 ackn:14
INSIDE RCVWND: base:23 end:4023 RCVUSRBUFF:0 RCVSGMBUFF:0
[SGM ->] 13:25:16:211395 dst: 192.168.1.121:50132 ctrl:8 plds:0 seqn:14 ackn:24
IS RCVWNDB: 23
RCV (WND): base:24 end:4024 RCVUSRBUFF:0 RCVSGMBUFF:0
STATE: FINWT2 -> TIMEWT
```

Fig. 14. *echo* Client execution

./echos -p 55000 -d

STATE: CLOSED -> LISTEN

WELCOME TO ECHO SERVER
Running on 0.0.0.0:55000

[<- SGM] 13:25:10:186599 src: 192.168.1.121:51654 ctrl:1 plds:0 seqn:0 ackn:0
STATE: LISTEN -> SYNRCV
[SGM ->] 13:25:10:187250 dst: 192.168.1.121:51654 ctrl:9 plds:0 seqn:10 ackn:1
STATE: SYNRCV -> SYNSND
[<- SGM] 13:25:10:190185 src: 192.168.1.121:51654 ctrl:8 plds:0 seqn:1 ackn:11
STATE: SYNSND -> ESTABL
[<- SGM] 13:25:14:962672 src: 192.168.1.121:51654 ctrl:40 plds:12 seqn:1 ackn:11 Hello World!
INSIDE RCVWND: base:1 end:4001 RCVUSRBUFF:0 RCVSGMBUFF:0
[SGM ->] 13:25:14:962894 dst: 192.168.1.121:51654 ctrl:8 plds:0 seqn:11 ackn:13
IS RCVWNDB: 1
RCV (WND): base:13 end:4013 RCVUSRBUFF:12 RCVSGMBUFF:0
[SGM ->] 13:25:14:963813 dst: 192.168.1.121:51654 ctrl:40 plds:12 seqn:11 ackn:13 Hello World!
SND (NXT): base:11 nxt:23 end:4011 SNDUSRBUFF:12 SNDSGMBUFF:1
[<- SGM] 13:25:14:964208 src: 192.168.1.121:51654 ctrl:8 plds:0 seqn:13 ackn:23
SACKED: 11
SND (WND): base:23 nxt:23 end:4023 SNDUSRBUFF:0 SNDSGMBUFF:0
INSIDE RCVWND: base:13 end:4013 RCVUSRBUFF:0 RCVSGMBUFF:0
IS RCVWNDB: 13
[<- SGM] 13:25:16:210237 src: 192.168.1.121:51654 ctrl:10 plds:0 seqn:13 ackn:23
INSIDE RCVWND: base:13 end:4013 RCVUSRBUFF:0 RCVSGMBUFF:0
[SGM ->] 13:25:16:210398 dst: 192.168.1.121:51654 ctrl:8 plds:0 seqn:23 ackn:14
IS RCVWNDB: 13
STATE: ESTABL -> CLOSWT
RCV (WND): base:14 end:4014 RCVUSRBUFF:0 RCVSGMBUFF:0
STATE: CLOSWT -> LSTACK
[SGM ->] 13:25:16:211211 dst: 192.168.1.121:51654 ctrl:10 plds:0 seqn:23 ackn:14
SND (NXT): base:23 nxt:24 end:4023 SNDUSRBUFF:0 SNDSGMBUFF:1
[<- SGM] 13:25:16:211418 src: 192.168.1.121:51654 ctrl:8 plds:0 seqn:14 ackn:24
SACKED: 23
SND (WND): base:24 nxt:24 end:4024 SNDUSRBUFF:0 SNDSGMBUFF:0
INSIDE RCVWND: base:14 end:4014 RCVUSRBUFF:0 RCVSGMBUFF:0
IS RCVWNDB: 14
STATE: LSTACK -> CLOSED

Fig. 15. *echo* Server execution

UPLOAD The *up* application allows the client to upload a local file to server. The client-side aplication visualizes a short report about file transfer, thus allowing to evaluate the librusp performances in such a common scenario. This simple application is very useful to evaluate the file transmission bitrate. Let us see an execution example: Figure 16 shows the client's execution output.

./upc 192.168.1.121 -p 55000 100MB

WELCOME TO ECHO CLIENT
Running on 192.168.1.121:54037 Connected to 192.168.1.121:47800

Sending File (size: 104857600): 100MB
100% [====================] OK

Sent: 104857.600000KB Droprate: 0.000000% Time: 3.587552s Speed: 233825.403680Kbps

Fig. 16. *up* Client execution

3.3 LFTP

The *Light File Transfer Protocol (FTP)* is an application protocol that allow file transmission and remote repository management. The protocol is clarly inspired by the well-known File Transfer Protocol (FTP), defined in [13]. Actually, LFTP is a lighter and simplified version of FTP: it does not require any authentication procedure and provides a smaller set of commands, with respect to FTP.

The LFTP is an *out-of band-control* protocol, in the sense described in [1]. Indeed, a LFTP session is made of two separate connections: the persistent *control connection*, employed in transmission of command requests and responses, and the temporany *data connection*, employed for data transfer. The LFTP user interacts with the application through the *User Manager*, wich istructs the control connection through the *Control Manager*. Every data transfer trough data connection is done by the *Data Manager* in *transfer mode*, in the sense stated in [13]. The LFTP service architecture is clearly inspired by the FTP's. Figure 17 shows the LFTP service architecture.

LFTP allows two processes to communicate through message exchange, thus encapsulating application-layer data into a LFTP message. Figure 18 shows the LFTP message structure.

- Type: identifies the message typology, thus characterizing it as a request message (RQST), a success response message (SUCC) or a failure response message (BDRQST).
- Command: the command to execute. Figure 19 shows the set of command provided by LFTP.

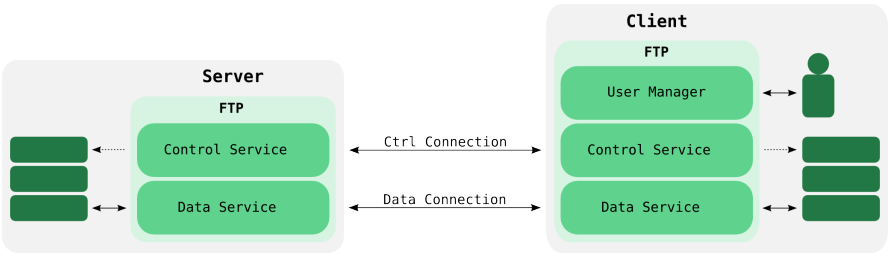

Fig. 17. FTP Service Architecture

Type	Command	Body

Fig. 18. LFTP Message Structure

Command	Description
GTCWD	retrieve the current working directory
CHDIR	change the current working directory
LSDIR	list the content
MKDIR	create a directory
RMDIR	remove a directory
CPDIR	copy a directory
MVDIR	move a directory
RETRF	download a file
STORF	upload a file
RMFIL	remove a file
CPFIL	copy a file
MVFIL	move a file.

Fig. 19. LFTP Commands

– Body: specifies the command arguments for request and response, if needed.

Let us now see an execution example: Figure 20 shows the client's execution output.

4 The Experimental Results

We now show the experimental results of using *librusp* in the typical scenario of a 100MB file transfer.

The testing environment consists of a server Ubuntu 14.10 Intel Core i5-3317U 1.70GHz 4GB and a client Ubuntu 14.04 Intel Atom N280 1.66GHz 1GB connected to the same WLAN through a router Technicolor TG784Nv3.

The purpose of the following tests is to verify the protocol robustness in response to segment loss, to optimize the sliding-window size and highlight any thread-balancing lack. Figure 21 and Figure 22 show the transmission speeds in function of segments loss and sliding-window size, and Figure 23 shows the average thread balancing detected by GNUProf.

Conclusions The recorded transmission rates prove the protocol robustness against segments loss and endorse our theoretical expectations about the sliding window. The protocol is in fact slightly elastic with respect to the segments loss rate, regardless of the window size. As expected, a window size close to the upper bound, imposed by the buffers, guarantees a significantly higher efficiency. However, the library turns out to be less powerful than the solution provided by the BSD's implementation of TCP, as well as suffering from a thread load imbalancing.

5 Future Improvements

We briefly present some planned improvements for future versions of the RUSP protocol and *librusp*. The main changes will concern the now unused control flags provided by the segment structure, the connection establishment procedure, the *librusp* architecture layering and the *librusp* thread load-balancing.

Control Flags RUSP will make use of the RST flag to recover from half-open connections, the KLV flag to keep-alive a long-silent peer, and the ERR flag to notify local error conditions.

Backlog RUSP will provide a dynamic backlog upper bound to accept large numbers of concurrent incoming connections. As stated in [2], the original BSD implementation provided a backlog upper bound set to 5, while modern Linux implementations set it to 128. Although these famous implementative decisions, we strongly believe that the backlog upper bound must be dynamically computed to minimize unserviceable incoming connections.

./lftpc 192.168.1.121 -p 55000 -r client-repo

```
WELCOME TO FTP CLIENT
Running on 192.168.1.121:38820 Connected to 192.168.1.121:37844

——-
MENU
——-
1 Get CWD
2 Change CWD
3 List Directory
4 New Directory
5 Remove Directory
6 Copy Directory
7 Move Directory
8 Download File
9 Upload File
10 Remove File
11 Copy File
12 Move File
13 Exit

[Your Action]>3
[Directory (empty to abort)]>.
[SUCCESS]>Listing server-repo
./
../
1MB1
1MB2
Folder/

[... Menu ...]
[Your Action]>9
[File (empty to abort)]>client-repo/1MB3
[SUCCESS]>Uploading client-repo/1MB3

[... Menu ...]
[Your Action]>8
[File (empty to abort)>1MB1
[SUCCESS]>Downloading server-repo/1MB1

[... Menu ...]
[Your Action]>13
Disconnected
```

Fig. 20. *lftp* Client execution

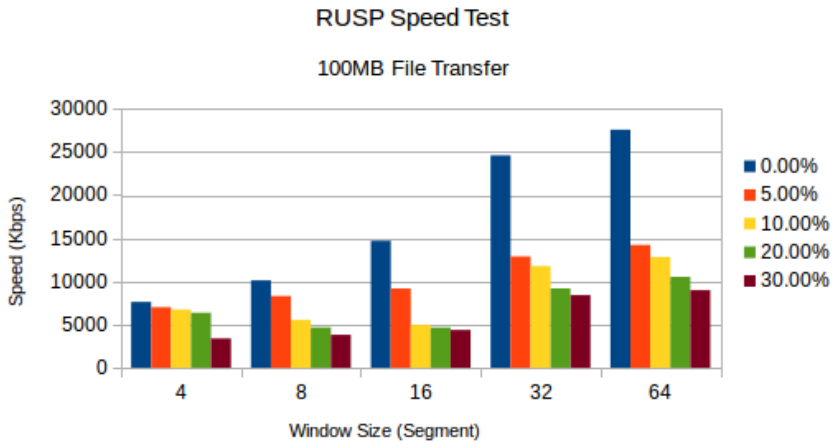

Fig. 21. RUSP Speed Test

Window Size	0.0%	5.0%	10.0%	20.0%	30.0%
4	7628	7024	6744	6372	3403
8	10126	8307	5539	4673	3819
16	14704	9187	4974	466623	4379
32	24604	12906	11785	9191	8426
64	27567	14219	12837	10534	9000

Fig. 22. RUSP Speed Test (tabular)

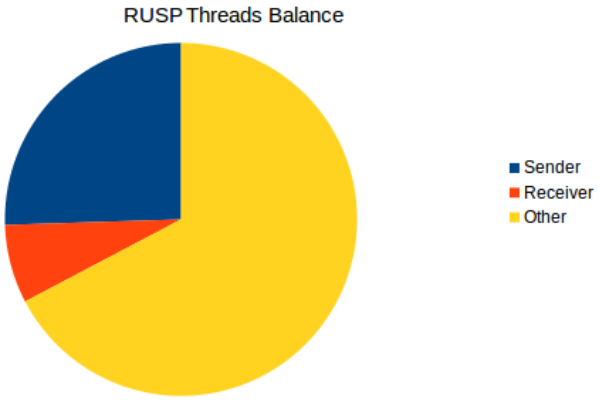

Fig. 23. RUSP and Threads-Balance

RUSP/IP librusp will no longer build its reliable service-layer on top of a BSD's SOCK_DGRAM unrealible layer, but directly on the IP's.

Thread Balancing The experimental results shown in Figure 23 highlight a severe thread load-imbalancing. Future versions of *librusp* will solve this kind of lack by increasing cooperation between send-side and receive-side threads.

References

1. J. F. Kurose, K. W. Ross, : *Computer Networking, a top-down approach*, 6th Edition, Pearson, October 2010.
2. M. Kerrisk: *The Linux Programming Interface*, 1st Edition, No Starch Press, October 2010.
3. J. Postel: *Transmission Control Protocol*, RFC 793, September 1981.
4. J. Postel: *User Datagram Protocol*, RFC 768, August 1980.
5. T. Bova, T. Krivoruchka: *Reliable UDP Protocol*, Internet Draft, Feburary 1999.
6. M. Mathis, J. Mahdavi, S. Floyd, A. Romanow: *TCP Selective Acknowledgment Options*, RFC 2018, October 1996.
7. M. Mathis, J. Mahdavi, S. Floyd, M. Podolsky: *An Extension to the Selective Acknowledgement (SACK) Option for TCP*, RFC 2883, July 2000.
8. V. Paxson, M. Allman, J. Chu, M. Sargent: *Computing TCP's Retransmission Timer*, RFC 6298, June 2011.
9. F. Gont, S. Bellovin: *Defending against Sequence Number Attacks*, RFC 6528, February 2012.
10. R. Rivest: *The MD5 Message-Digest Algorithm*, RFC 1321, April 1992.
11. R. Branden: *Requirements for Internet Hosts Communication Layers*, RFC 1122, October 1989.
12. J. Postel: *Echo Protocol*, RFC 862, May 1983.
13. J. Postel, J. Reynolds, *File Transfer Protocol (FTP)*, RFC 765, October 1985.