Alexandre DENIS
Ecole Normale Supérieure de Lyon
Magistère d'Informatique et Modélisation

Variable Reliability Protocol
in Globus - Nexus

June 7th, 1999 to August 15th, 1999
University of Southern California / Information Sciences Institute
Project leader: Dr. Carl KESSELMAN.

Abstract

Applications have to chose between the slow *TCP* and the unreliable *UDP*. There is now an alternative: the Variable Reliability Protocol. The applications can specify what are the allowable losses in the data, and the protocols guarantees that the given loss tolerance parameters will be respected. It should result in a faster throughput over WAN which have typically a higher loss rate.

VRP was proposed by Robin Kravets from the GaTech. In this paper we describe how it has been improved, implemented, and integrated to Nexus, the communication library of the meta-computing environment Globus.

Contents

Chapter 1

Introduction

1.1 Globus and Nexus

One of the primary goals of the Globus project is to develop the software tools and services necessary to build a computational grid infrastructure and to develop applications that can exploit the advanced capabilities of the grid.

Communication in grids is complicated by the heterogeneity of the environment and the large dynamic range in communication performance that components of an application see. Communication may cross high-performance parallel computer communication networks or the Internet. Complex applications may have a range of communication requirements and complex tradeoffs between communication characteristics, such as bandwidth, latency, jitter, and security.

To address these issues, a communication library, Nexus, has been designed specifically to operate in grid environment. Nexus is distinguished by its support for multimethod communication, providing an application a single API to a wide range of communication protocols and characteristics.

Nexus is a portable library providing the multithreaded communication facilities required to implement advanced languages, libraries, and applications in heterogeneous parallel and distributed computing environments. Its interface provides a global memory model via interprocessor references; asynchronous events; and is thread safe when used in conjunction with the Globus thread library. Its implementation supports multiple communication protocols and resource characterization mechanisms that allow automatic selection of optimal protocols.

`http://www.globus.org` for more informations about the Globus project.

1.2 Variable reliability

1.2.1 Aim of variable reliability

Applications have to chose between completely reliable message transfer (ie. *TCP*), with the price of unnecessary retransmission of messages and complex buffer management, and *UDP* completely unreliable, which may result in unacceptable losses.

The two types of reliability provided by *TCP* and *UDP* are insufficient. Given a lack of choice, most applications use reliability even when they do not absolutly need it. It results in lower performance than what we could expect, partially due to many retransmissions that indeed enable the messages to arrive, but sometimes too late.

Ideally the application should give the protocol some information about the allowable loss, and there should be a feedback to tell the application the actual losses. It could then adjust its requirements to the available network ressources.

1.2.2 Loss tolerance

Definition: Reliable data transfer can be defined as a service that guarantees to deliver data sent from a sender to a receiver without duplication, loss or messages delivered out of order. This protocol provides the application with the ability to define a level of allowable losses within the data transmitted.

Parameters: allowable loss is specified through two parameters:

- the *loss percentage* which indicates how much loss the application can tolerate. To ensure that the losses are spread within the data, we use a sliding window mechanism. This loss percentage becomes the maximum number of losses allowed in a window.
- the *maximum number of consecutive losses* which defines the maximum loss burst size. Several consecutive packets are often lost (case of buffers overflow), and the application tolerates only a maximum loss size (eg. to interpolate the missing data).

The application specifies both of these parameters and is guaranteed that every loss permited by the protocol will not break any of these rules. As these parameters are *maximum*, there may be much fewer losses in practical.

This work aims in both improving the Variable Reliability Protocol and implement it into Nexus.

Chapter 2

The Variable Reliability Protocol

The following is a description of the *Variable Reliability Protocol* (called *VRP* in this paper), proposed by Robin Kravets in her paper [4], and some improvements I proposed in order to increase performance and to adapt *VRP* to Nexus.

Robin Kravets' implementation will be referred as "first" or "basic" implementation, while the one I propose will be called "new".

2.1 General principle

2.1.1 *VRP* in the Internet protocol model

Theoretically, in the Internet protocols stack, *VRP* is a transport layer protocol, like *TCP* and *UDP*. It provides services that are actually an alternative to these protocols.

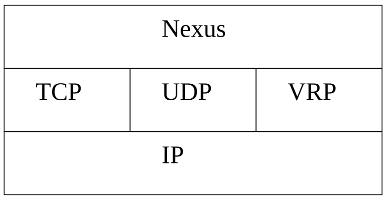


Figure 2.1: Theoretical place of *VRP* in the Internet model

However *VRP* is not implemented in the kernel of the operating system but in user space. Though it should be on top of *IP*, as we are in user space, the only way to get *IP* service is to use *UDP*.

VRP could be implemented in any protocol stack that follows the OSI model.

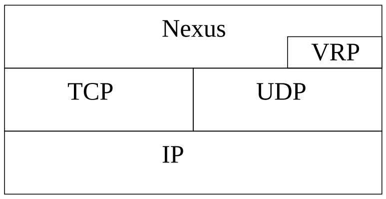


Figure 2.2: Actual place of *VRP* in the Internet model

2.1.2 Frames

VRP is a datagram-oriented protocol. The application sends *frames* of data which may have a random size, then the protocol cuts this frames into *UDP* packets of a fixed size. Why datagram oriented? As the application specifies data boundaries, the sliding window and loss tolerance parameters are reinitialized at the begining of each frame, since there is often no logical link between the different frames.

The first protocol used a connected mode.

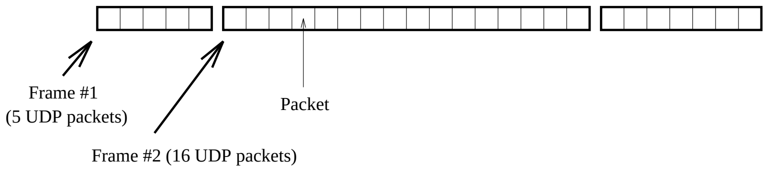


Figure 2.3: Data frames cut into packets

Sliding window: Packets are transmitted using a simple sliding window mechanism: the sender sends packets in order, and waits for acknowledgments. The window size represents the maximum number of unacknowledged packets. For each packet, a timer is started, when the timeout expires, the packet is sent again.

The receiver sends acknowledgments as soon as it receives the packets. It acknowledges a lost packet if it is an allowable loss. If a non-allowable loss is detected, it sends a *negative acknowledgement* (NACK, and the deduced verb "to NACK") for the sender to send this packet again immediatly.

Using this scheme, the sender believes that every packet arrives since the whole loss detection is performed in the receiving part.

The previous description was not specific enough for a straightforward implementation and to build something usable in Nexus. What I will describe here is a slightly modified version of the protocol found in Robin Kravets' paper.

2.2 Very first implementation

2.2.1 Sender side

The very first implementation used a sliding window with a list of outstanding messages. Basically, when data enters the window, it is copied in a buffer and the packet header is built. When the packet is sent, we put the buffer into the outstanding messages list. When the an acknowledgment is received, the corresponding packet is removed from the outstanding messages list. It is removed from the buffers when it leaves the window.

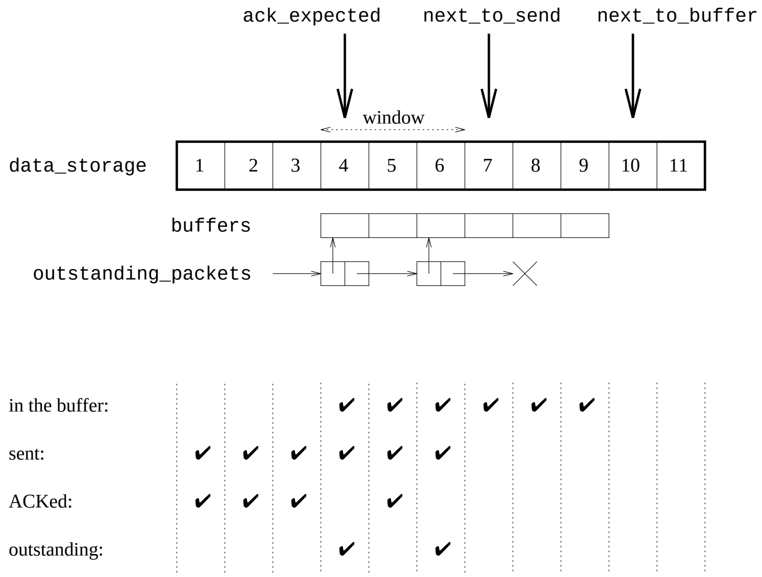


Figure 2.4: Example of sender's state

Figure 2.4 shows a valid state for a sender.

- `ack_expected` is the number of the first unACKed packet. When it reaches after the last packet number, then the transfer is complete.

- `next_to_send` is the next packet to be sent. It must already be in the buffers. `next_to_send` − `ack_expected` must be less than the window size. It represents the maximum number of simultaneous unACKed packets.

- `next_to_buffer` is the next packet to be prepared.

- `buffers` stores the packets ready to be sent. They are removed from this buffer when there number is below `ack_expected`.

- `outstanding_packets` is the list of the packets in the buffer that have been sent but not yet ACKed. Moreover, it contains the timeout date for each of these packets.

2.2.2 Receiver side

The receiver part proposed in the paper was a little confusing. There were basically two windows:

- a "reliability window", which aims to be a classic window
- an "history window" to remember the latest packet losses, in order to check if the rules are broken.

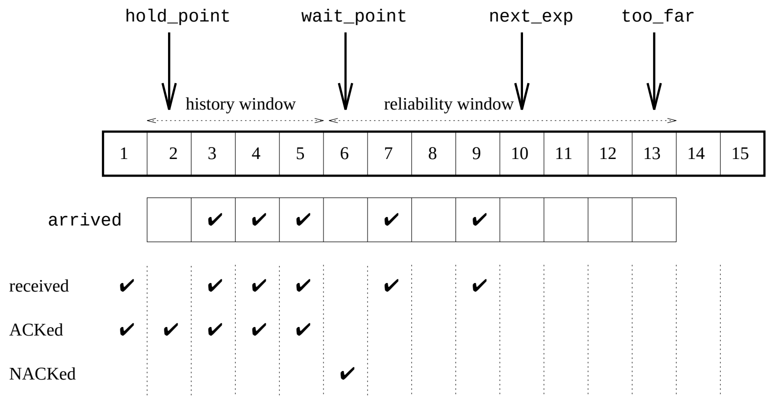


Figure 2.5: Example of receivers's state

The variable used are:

- `hold_point` is the lower edge of the history window. It indicates the oldest lost message in the receive history, and must be at least `wait_point` minus the history window size.
- `wait_point` is the lower edge of the reliability window. It indicates the message we are curently waiting for.
- `next_exp` is the next expected message. It is the sequence number following the highest-numbered received message. Notice that `wait_point` equals `next_exp` in case the receiver is not waiting for outstanding messages.
- `too_far` is `wait_point` plus the window size. Every message with a sequence number greater than this is refused.
- `arrived` is the bitmap of arrived messages in both windows.

2.3 Basic improvement

2.3.1 Analysis of loss detection

We assume that out of order packets are lost. It means that if we receive a packet between `next_exp` and `too_far`, packets between the one we have just received and `next_exp` are supposed to be lost. If we receive the packet with the sequence number `wait_point`, it is accepted. Every other packet is refused.

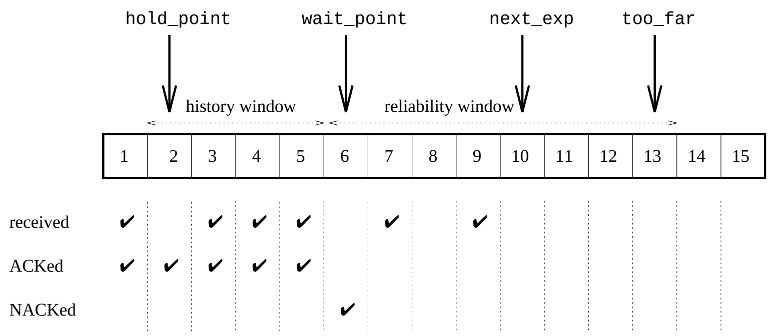


Figure 2.6: Without anticipation

Checking that the consecutive losses number is not greater than what can be allowed is easy and fast. The basic algorithm however sends too many NACKs or too late. No need to send a NACK for every packet that are in a too large hole, no need to wait for an outstanding packet before we send other NACKs. We can see on figure 2.6 that while we are waiting for packet 6 to arrive, the sender is likely to send again packets 6 through 9 because of timeouts expirements because we are stuck on packet 6, though only packets 6 and 8 are required.

The solution? Send exactly what is required, and as soon as possible! It is based on an algorithm that is able to *anticipate*. The first one checked the

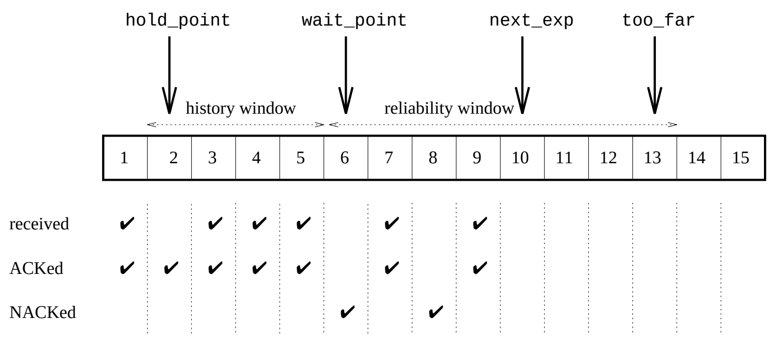


Figure 2.7: With anticipation

loss tolerance parameters (and send ACKs and NACKs) only for packets before `wait_point`, whereas the true active part is between `wait_point` and `next_exp`. It results in NACKs sent much too late, and unnecessary retransmissions for packets that were received but ACKed too late. Figure 2.7 shows what we want: packets 7 and 9 are individually ACKed, and packet 8 is NACKed while we are still waiting for packet 6 that was NACKed before.

In the new algorithm, the receiver must provide the sender with more information: we need cumulative ACKs, NACKs as soon as possible, and individual ACKs for packets beyond `wait_point`.

2.3.2 Sender optimization

With a big packet size and window size, the basic method requires a lot of memory. When there will be many frames handled at the same time, it would be an unacceptable waste of memory.

We decide to build the packets "on the fly": it drops the cost of buffers management and reduces the number of data copy. The packet header has to be rebuild if the packet has to be resent, but it is actually no heavy load.

The state of the window (ie. between `ack_expected` and `next_to_send`) is stored in an array, where a cell is the state of a packet. It contains:

- an ACK flag not to send again a packet that has been ACKed. It is necessary because we allow ACKs for a single packet instead of having only cumulative ACKs.
- a timeout value. The packet is sent again if the timeout expires.

`buffers` and `next_to_buffer` are deleted since there are no more buffers.

2.3.3 Incremental and anticipation algorithm

Anticipation: A first optimization consists in sending ACKs for individual packets as soon as it is received, and NACKs only when necessary, but as soon as possible. The previous algorithm sends NACKs only when we check the loss tolerance parameters when `wait_point` is incremented and uses only cumulative ACKS. Typically everything between the lost packet and the last packet received will be sent again.

Incremental: We want an algorithm that doesn't have to browse the whole history bitmap every time. It is a waste of time with large windows.

There are several methods to decide whether a loss violates the loss rate parameter or not in the first algorithm: we can browse the entire bitmap of the history window and reliability window. It is accurate but slow. Another method is to keep the number of losses inside the history window in a variable and test only when we increment `wait_point`. It is accurate too, and perfectly suited if we use cumulative ACKs only.

If we try to send ACKs for individual packets, the previous algorithm won't be accurate and will fail since we don't have to take the whole history window into account. It is correct only if we check at `wait_point` only. It is false if we check earlier (at `next_exp`).

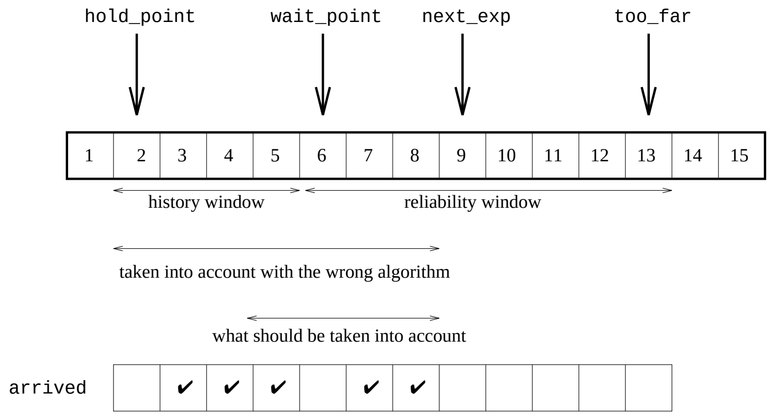


Figure 2.8: What should be taken into account for loss detection

Instead of adapting it to use both incremental and anticipation approach, we will write a new one from scratch.

2.3.4 New receiver algorithm

This algorithm uses a fixed history window (no more `hold_point` which makes a random-sized history window). `too_far` will be deleted too: it may lead to an unpredictable behaviour if the receiver's window size is not the same as sender's one. As the sender may have a variable size window, we would be lucky if sizes matched!

When a frame receiving information block is created, we allocate one cell per packet that will be received. See the section 2.4.5 for more information about the creation of a frame. These cells will contain the state of the packet with the following boolean fields:

- `received` is true if this packet has been received.
- `valid` is true if this packet is valid, ie. if it is either received or an allowable loss.
- `waiting` is true if we are waiting for this packet to arrive. A NACK has been sent.

As a typical packet size is 1024 bytes, the size of the state array will be negligible.

There are still:

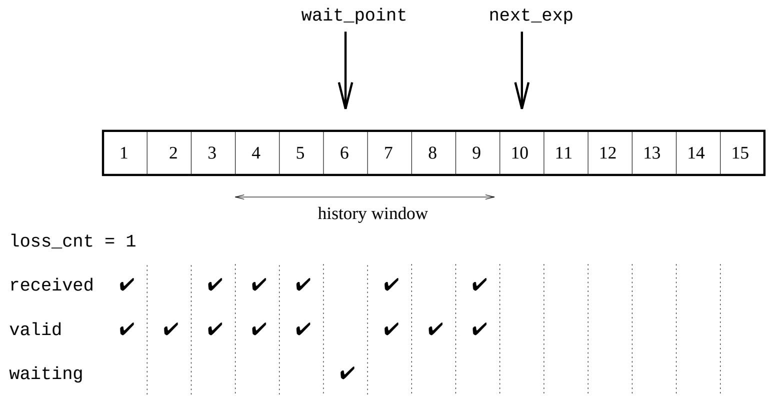


Figure 2.9: A valid state with the new algorithm

- `wait_point` which indicates the outstanding packet with the lowest number. When there are no outstanding packet, it is equal to `next_exp`. Each time an ACK or NACK is send, it contains an additional field with a cumulative ACK with `wait_point` value.

- `next_exp` is the next expected packet.

We add a `loss_cnt` variable that indicates the current number of packet lost among the `window_size` packets preceeding `next_exp`. Don't be confused here, `window_size` is the size of the *history window*. There is no more reliability window.

It is an anticipation algorithm because, as shown on figure 2.9, `loss_cnt` counts only *actual* losses (eg. packet 2 or 8) and not packets taged as `waiting`, because they are *virtualy* not lost since they *have* to arrive!

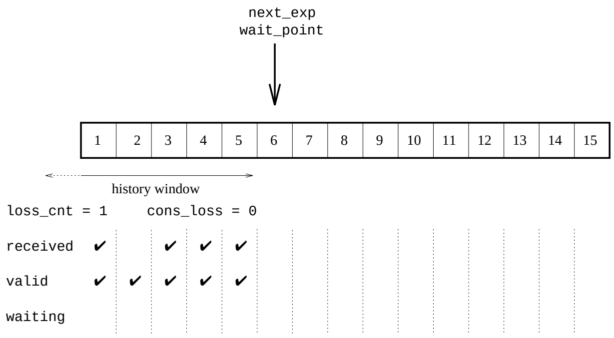
It is an incremental algorithm because we don't have to browse the whole history window to update `loss_cnt`. Each time we shift the window, we update the counter according to what enters and what leaves the window.

`cons_loss` is another loss counter. It indicates how many consecutive losses are currently at the window begining. It is non-zero only while we are updating the window (see examples below).

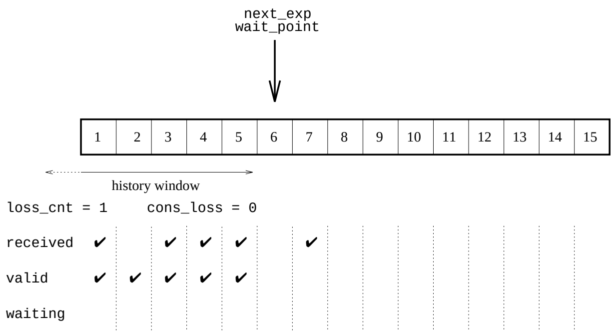
2.3.5 Examples

Example with maximum loss rate

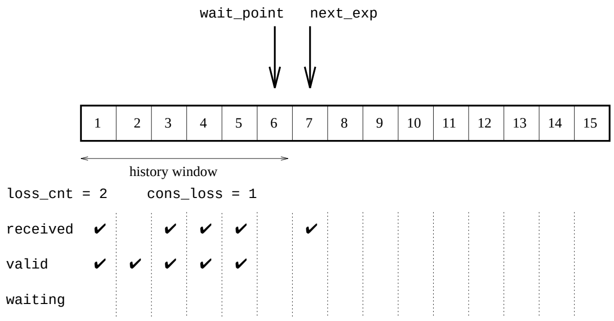
The window size is 6, the maximum loss rate is one packet per window (16%).



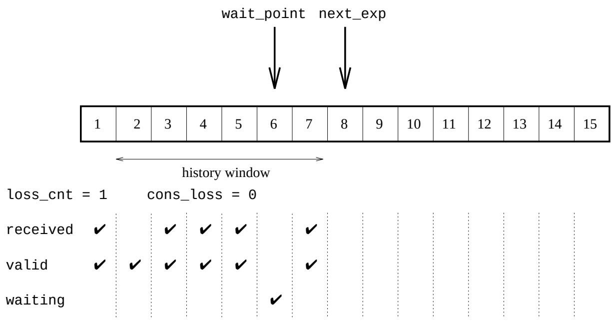
The loss of packet 2 was allowed. `loss_cnt` is 1. Packet 7 arrives, packet 6 is lost:



We update the history window:



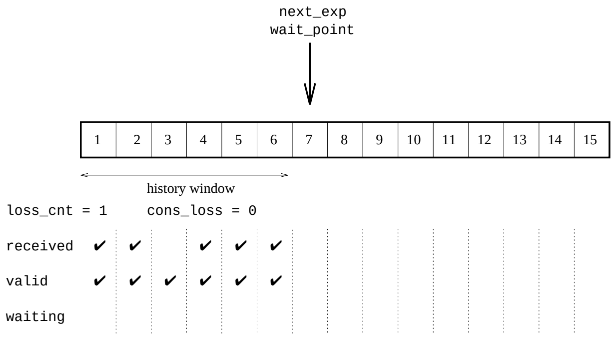
`loss_cnt` is now 2, which is not allowed. We send a NACK and continue the history window update. `loss_cnt` is then only 1 because packet 6 is not lost: it has to arrive.



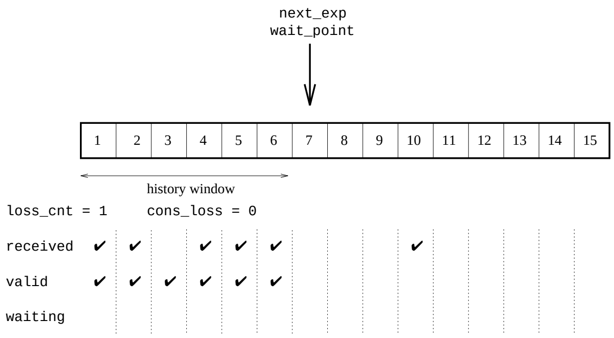
We are in a valid state, let's wait for another packet!

Example with maximum consecutive losses number

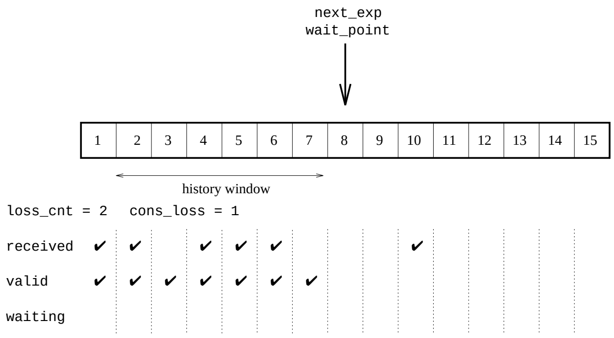
The window size is 6, the maximum loss rate is two packets per window (32%), the maximum consecutive losses number is set to 1.



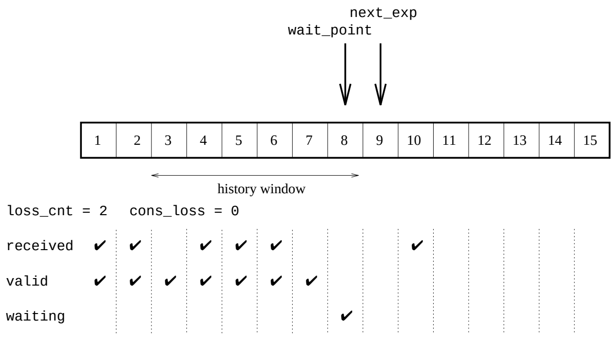
3 packets are lost. We receive packet 10.



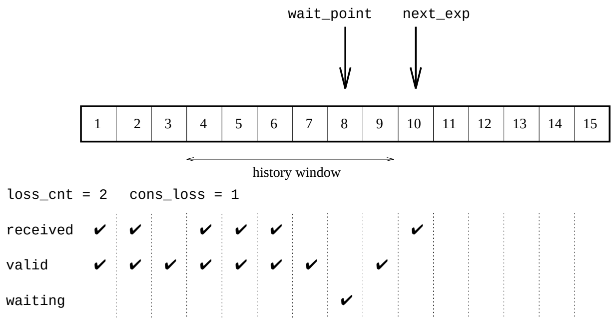
We begin to update the history window:



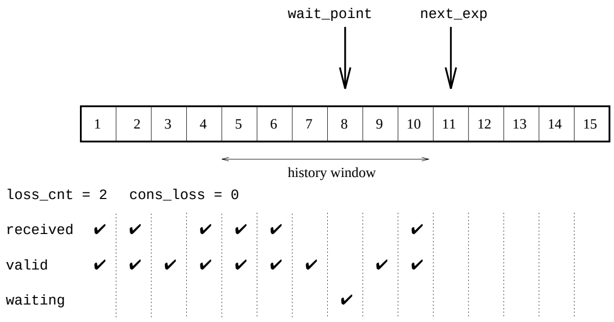
Everything is ok, the loss is allowed.



The loss of packet 8 is not allowed, since `cons_loss` would be 2 and the tolerance is only 1. The packet is marked as `waiting`, we send a NACK for packet 8, we reset `cons_loss` (since packet 8 is virtualy no more lost) and we don't increment `loss_cnt`.



Packet 3 leaves the window and lost packet 9 enters the window, so `loss_cnt` doesn't change. The loss of packet 9 is allowed, thanks to the anticipative algorithm.



The final state is valid, another packet can be received. `wait_point` will be updated when we will receive packet 8. It will be set to the next outstanding packet (or `next_exp` if none are outstanding) and a cumulative ACK will be sent for this new value.

2.4 Further on frames

2.4.1 Simultaneous frames – multi-threaded application

Until now, we have assumed only one frame was handled at a time. In a multi-threaded application, it is possible that more than one frame is sent simultaneously. It is even worse for the receiver: it can have different clients from different hosts that send data at the same time.

There is still another case: if a thread sends a large frame, it should not prevent another thread to send another frame to the same receiver. If the connection was a usual FIFO, the large frame would lock the network socket causing the other thread to wait.

2.4.2 Connected vs. unconnected

The first implementation proposed a *connected* protocol, that has a lot of disadvantages:

- it is much less fault tolerant than an unconnected protocol,
- it was only a trick to detect the end of frame,
- it is incompatible (or at least, hard to do) with the multiple simultaneaous frames.
- why should it be connected if it can be unconnected?

It seems impossible to be in an unconnected state while a frame is being transmited, but why should this connection be kept between frames? Why should every frame go through the same connection? (though they all use the same *UDP* socket).

2.4.3 Outgoing & incoming frames information blocks

Both sender and receiver data structures only contain global parameters about the connection (eg. destination address, port, info about the timeout...). When the sender has to send a frame, it creates an *outgoing frame information block* that contains all the info about the current frame. Several blocks can be created inside the same sender so that it can handle several frames at the same time – for example, a big frame won't cause a "traffic jam" on the socket, smaller frames will be permitted to overtake it. Notice however that a sender will always send frames to the same receiver.

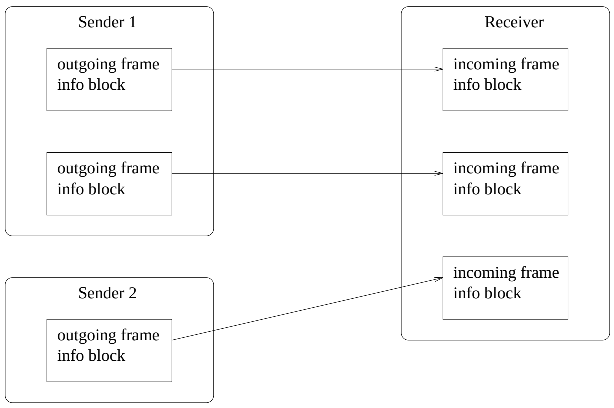


Figure 2.10: Outgoing & incoming frames information blocks

It is exactly the same on the receiver side. When it receives the order to create a new frame, it constructs an *incoming frame information block*. Many blocks can be created inside the receiver, and may have different origin. When a packet is received, the receiver decodes the frame ID that enables it to pass the packet to the right incoming frame handler.

Each frame information block represents one connection. As soon as the frame is transmited, the connection is closed (but both sender and receiver are still there!)

It is so a very flexible method: no permanent connection has to be kept, we have separate sliding windows for each outgoing frame, and the receiver can easily manage multiple connections from different senders at the same time.

Consequently, the connection will exist only while a frame is being transmitted, and the receiver has no state (it is always ready, can always accept frames whatever it is doing). Though it is unconnected, the sender keeps some information about the receiver (see the section 2.5.1 about adaptive timeout): it is a faked connected mode.

Frame ID: When we receive a packet, we need to identify from what frame it is. The frame ID will be:

- the sender's *IP* address and *UDP* port, that identify the sender,
- a frame number, that identify the frame information block among the others of the same sender

Frame number: The frame number is an arbitrary number that identifies the frame for a given *IP* address and port. They are assigned in order, starting with a random number.

In case a sender is down and immediatly up (but reinitialized) it is really unlikelly that the new starting point will be a current frame number (ie. frame not closed yet). If ever it happens, they won't understand each other and the connection will be reseted (see the "Frame timeout" paragraph below).

End of frame: The end of a frame is difficult to manage. Even if the receiver has received the whole frame, how can it be sure that the sender has received the ACKs? The first algorithm detected the end of a frame when the next one began. It used a timer in a connected mode when there was no current frame, the timer handler sent state messages.

The use of a timer will be difficult in Nexus, and we can't detect the end of a frame thanks to the begining of the next one since there is now no more order. We chose to keep the connection opened as long as it is useful. It means that even if the receiver and the sender believe that the frame transmission is finished, they keep outgoing and incoming frame information blocks.

When the sender detects that a frame transmission is finished (ie. last ACK received), it marks it with a tag. When a new outgoing frame information block is created, we delete every other block marked as finished and send an "end of frame" command in a reliable way (see "Frame header" below) to the receiver which is then allowed to destroy safely the corresponding incoming frame information block. It is sure the sender is no more waiting for ACKs that might have been lost.

Frame timeout: If ever things get wrong, every frame information block has a "life timeout": if the sender or the receiver don't receive new packets for a frame for a given time (typically a few seconds), or if these packets are not understood, then the frame is destroyed and the sending function returns an error.

It guarantees that there won't be any deadlock, in any case, but it's up to the application to decide wether it trys again or if it gives up.

2.4.4 Critical data, loss tolerance parameters

Critical data: The first loss tolerance parameters provide us with a powerfull way of specifying the service we want, but it isn't still enough. In many cases, there are parts where losses are allowable, but small other parts where nothing can be lost (eg. headers of the RSR in Nexus).

VRP provides us with another loss tolerance parameter called "critical data". It is an interval of data where nothing can be lost, whatever are the other loss tolerance parameters.

Data encapsulation: As the application doesn't have to know the packet size and window size, we propose to give all loss tolerance parameters in bytes and not in packets:

- the critical data is specified as an interval of bytes, *VRP* translates this into critical packets.
- the maximum conscutive loss size is given in bytes, *VRP* translates this into a maximum number of consecutive packets lost.
- the maximum loss rate is given as a percentage, *VRP* translates this into a maximum number of packets lost per window.

2.4.5 Frame header

To transmit all this additional information about the data, we propose a new principle: the "frame header". When the sender creates a new outgoing frame, it sends this header, a single packet that contains all the information about the new frame:

- the frame number,
- the data size and amount of data per packet,
- the loss parameters: maximum consecutive losses and maximum loss rate,
- the critical packets (packets required to arrive),
- a variable number of "end of frame" commands.

This frame header is transmited as being itself "critical" and will never be allowed to be lost. In case it is duplicated, the frame creation is not duplicated since the frame header contains the frame number.

2.5 Tuning *VRP*

2.5.1 Adaptive timeout

The first algorithm used a *fixed* timeout. Given that Globus uses machines between which the RTT (round-trip time) can be from 5 to more than 200 miliseconds, it has to be changed.

Given that developing such an adaptive timeout algorithm is difficult and will undoubtedly lead to an unstable dynamic system, we rather chose to use an existing one: the Jacobson algorithm, basically designed for *TCP*.

Jacobson algorithm: it is based on approximations of the round-trip time RTT and the average deviation D. Let M be the measured round-trip time. D is the average of $|RTT - M|$.

The recurrent formulas to approximate RTT and D are :

$$RTT = \alpha RTT + (1 - \alpha)M$$

$$D = \alpha D + (1 - \alpha)|RTT - M|$$

We then take

$$timeout = RTT + 4 \times D$$

It is more accurate than the old formula $timeout = \beta \times RTT$ since the average deviation has no link with RTT. The factor 4 is somewhat arbitrary, but we measure that less than 1% of the packets for which the timeouts expires are not lost. We take usually $\alpha = 7/8$.

Karn algorithm: if a timeout expires, when the packet is eventually ACKed, don't update RTT and D since M is impossible to calculate unless we specify which retransmission is being ACKed. Karn recommends to double the timeout each time a packet is lost. If we don't want to break the reccurrent serie, we had to multiply RTT and D rather than $timeout$.

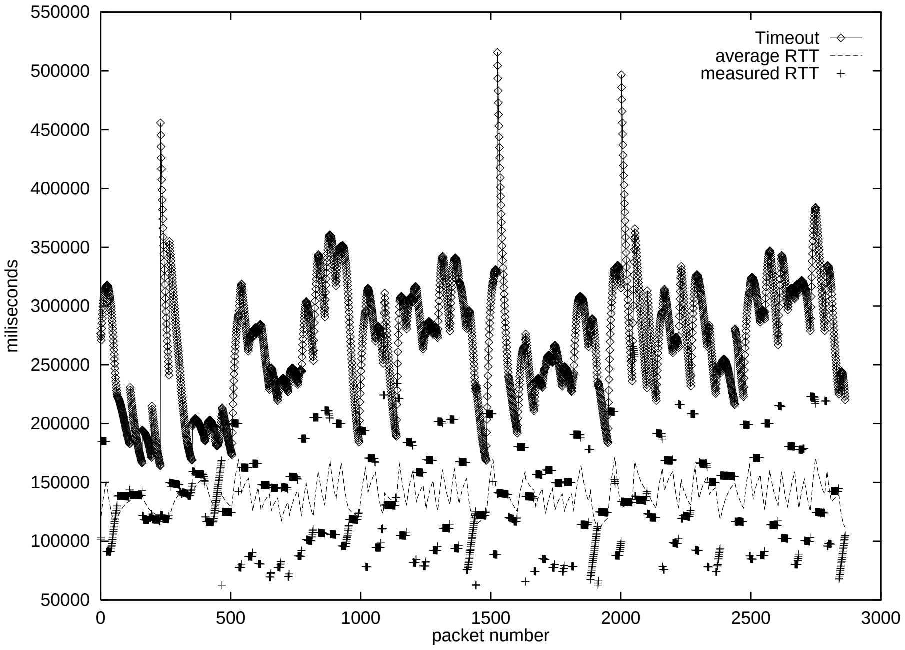


Figure 2.11: Example of timeout evolution

Initialization: when the sender is created, it sends a timestamped "ping" to the receiver. When we get the response, we are able to calculate a first approximation of RTT. The ping itself has a timeout with a default value.

As a rather good begining for *timeout* is $2 \times RTT$, we chose $D_{init} = RTT_{init}/2$.

Figure 2.11 shows the evolution of *timeout* and RTT over 3000 packets sent on a WAN. The adaptive method seems adequate. The timeout expires while the packet is not lost in less than 1% of the cases (assuming every packet duplications are due to this) and the delay doesn't diverge – it is almost flat on LANs with an even RTT, and follows the evolution on less predictable connections such as WANs.

2.5.2 Packet size

Chosing the right packet size is essential to get decent performances, but it asks the following question: what does the *right packet size* mean? Is it the one that allows the best throughput? Is it the one that causes the fewest losses?

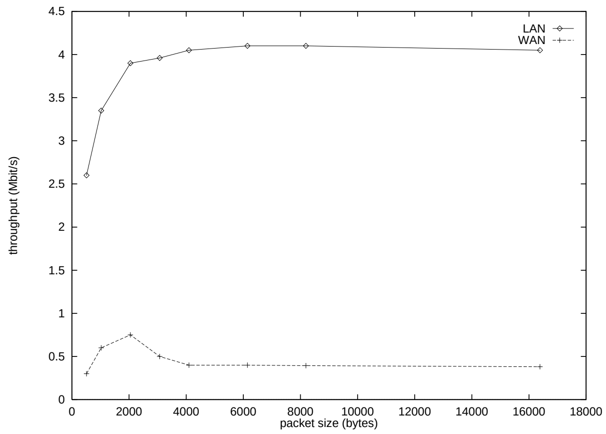


Figure 2.12: Throughtput function of the packet size

Let's make some benchmarks: the window size is fixed, the timeout uses Jacobson and Karn algorithms, we allow 0% loss.

According to figure 2.12, the throughput on a LAN increases when the packet size increases, while the throughput on a WAN reaches its maximum for 2 KB packets. That is only a typical value, there are some days where it is faster to use 1 KB packets! There are more losses on a WAN with 3 KB and beyond, but it is hard to give precise figures.

We chose 2 KB packets not to be too slow on a LAN (rather than 1 KB) and to have a maximum performance on a WAN. This value though can be overriden by the application for very specific uses.

2.5.3 Window size

Finding the right window is not so complicated than other parameters. A large window is usefull with high latency networks (long distance) but can waste some memory.

For implementation convenience, it has to be an exact power of 2. There are no more improvement in performance when we reach 128, but rather a sudden huge increase of the actual loss rate, as the receiver's buffers are full.

We chose a default size of 64, but it can be overriden by the application. This size is large enough for high-latency networks, but we never observed particular additional losses due to a too large window with 64.

Chapter 3

Implementation

3.1 Packet format

Every packet transmitted have the same 6 bytes header:

- 1 bit which indicates if the packet contains data or is a header
- 15 bits for the frame number
- 32 bits for the packet number inside the frame

If the packet contains data, the header is followed by the data and that's all. If it is a header packet, it has a variable length and is followed by a variable number of tags:

- `LAST` is the last tag of the header, and is followed by nothing.
- `PING` indicates that this packet is a ping and is followed by 8 bytes that contains the current time (seconds, microseconds).
- `PONG` indicates that this packet is a reponse to a ping. It is followed by the 8 bytes that followed the received `PING`.
- `ACK` followed by a 4 bytes packet number is a cumulative ACK for this packet and all the previous packets in this frame.
- `SINGLE_ACK` followed by a 4 bytes packet number is an individul ACK.
- `NACK` followed by a 4 bytes packet number is a negative acknowledgement for this packet number in the current frame.
- `FRAME_CREATE` is the begining of a new frame. It commands the receiver to create a new incoming frame information block if this frame number doesn't exist yet. This tag is followed by a 2 bytes frame number, 4 bytes for the data size and 2 bytes for the amount of data per packet.

- CONS_LOSS sets the maximum consecutive loss number in the new frame. It is followed by 1 byte which indicates the maximum number of consecutive packets allowed to be lost (calculated by the sender from the maximum burst loss size given by the application in bytes). It is not included in FRAME_CREATE since it is optional.
- LOSS_RATE is the maximum percentage of data that is allowed to be lost in this frame. It is followed by 1 byte where 0 means that 0% can be lost, 255 means that 100% percent can be lost (obvious linear iterpolation rule...). This tag is optional too.
- FRAME_OK followed by a 2 bytes frame number indicates that the receiver has successfully created an incoming frame information block.
- FRAME_REFUSED followed by a 2 bytes frame number indicates that the receiver has no more frame information block available. The sender will return an error to the application.
- FRAME_CLOSE followed by a 2 bytes frame number asks the receiver to destroy this incoming frame information block.
- MISSING_HDR followed by a 2 bytes frame numbr indicates that the receiver has received data packets with an unknown frame number.
- CRITICAL_PACKET followed by a 4 bytes packet number warns the receiver that this packet is critical and is not allowed to be lost.
- CRITICAL_INTERVAL followed by two 4 bytes packet numbers warns the receiver that packets between these two numbers are critical.
- PADDING followed by a 2 bytes size and padding data according to this size, useful to send a ping with the right packet size. If the ping packet is too small, the initial RTT is wrong.

Some of the tags are ignored if they are in the wrong way (for example a sender won't understand FRAME_CREATE).

A header packet can contain a variable number of these tags. There are some typical sequences: NACK is often followed by SINGLE_ACK; LOSS_RATE and CONS_LOSS don't make sense if they don't follow FRAME_CREATE.

The tag itself is represented as a single byte, all multi-bytes integers must be given big-endian for portability.

3.2 Asynchronous I/O

For performance issue and because there will be no choice in Nexus, all input and output have to be asynchronous.

It means that it is impossble to do a blocking read operation on the socket, the only way to read the data is to provide Nexus with a handler that will be triggered each time a `select` system call would have checked that data are available.

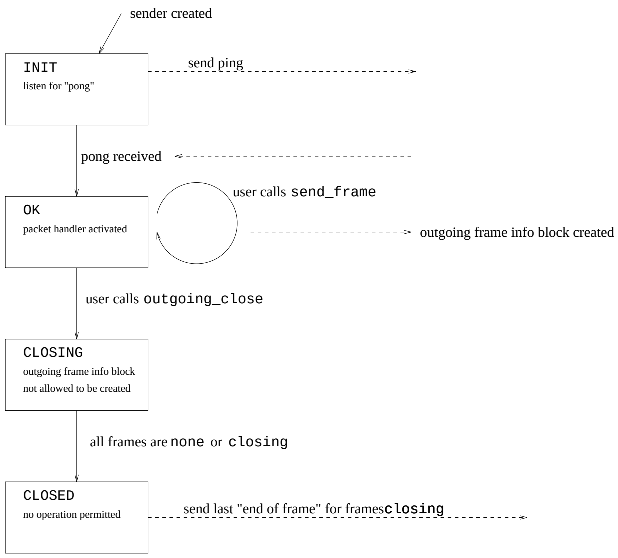


Figure 3.1: The sender's states

3.3 Sender

3.3.1 The sender global structure

Figure 3.1 shows the global behaviour of the sender. Such a structure is created for each socket.

3.3.2 Outgoing frame information block

Figure 3.2 shows the behaviour of a frame information block. One of these is created each time the user calls the function `send_frame` to send a frame. The function calls terminates before the structure is destroyed to handle properly the end of the frame.

3.3.3 Packet handler

The sender is supposed to receive only header packets and no data packets. It processes the tags in a loop, until it reaches `TAG_LAST`.

- `PONG`: if sender's state is `INIT`, then calculate *RTT* and *timeout*, set the sender state to `OK`.
- `FRAME_OK`: sets the frame state to `SEND`.
- `FRAME_REFUSED`: sets the frame state to `REFUSED`.

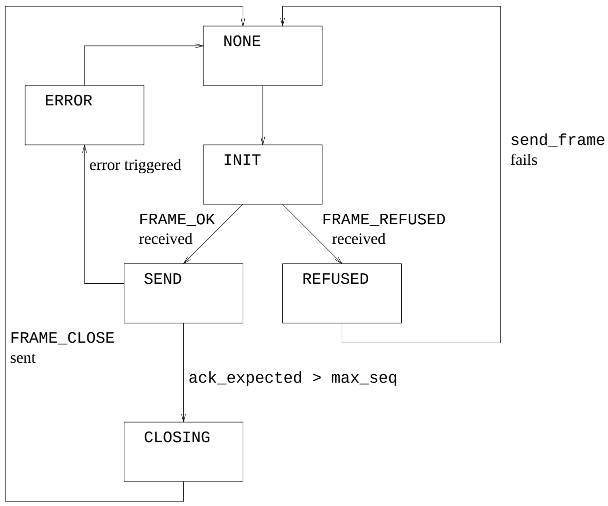


Figure 3.2: The states of the outgoing frame information block

- `FRAME_MISSING_HDR`: resend the frame header.
- `SINGLE_ACK`: if the sequence number is in the window, the packet is marked as ACKed; if it has not been retransmited, *RTT* is calculated and *timeout* is updated. If the packet has not been sent, the frame state is set to `ERROR`.
- `ACK`: same operation than `SINGLE_ACK` performed in a loop from `ack_expected` to the sequence number of the ACK.
- `NACK`: if the packet is in the window, it is sent again and marked as "retransmited", else the frame state is set to `ERROR`.

All other tags are discarded.

3.4 Receiver

3.4.1 The receiver global structure

As it is a connectionless protocol, there is no global receiver state. Once it is running, it is ready to accept frames.

Almost all of the information is in the incoming frame information blocks.

3.4.2 Incoming frame information block

Once again no state, but rather a set of flags:

- `in_use`: if this flag is not set, this incoming frame information block is empty.

- `receiving`: set if data is being received, otherwise the transfer is completed.
- `deliver`: if it is set, the Nexus RSR is being delivered. Every operation on the data storage are strictly forbiden.
- `closed`: the block can be freed as soon as `deliver` is false.

3.4.3 Packet handler

The receiver can receive both header packets and data packets.

Header processing

The tags are read in a loop with the appropriate response:

- `PING`: send `PONG`.
- `FRAME_CREATE`: check if the frame creation message is not duplicated. If it is, discard the whole header. If there is no space left for the frame information block, send `FRAME_REFUSED` else send `FRAME_OK`.

  Decode data size, packet size, caculates the number of packets and allocate a storage and initialize the array for packet state.
- `CONS_LOSS`: set the maximum number of consecutive packet lost allowed for the current frame.
- `LOSS_RATE`: set the maximum loss rate (number of packet lost per window) for the current frame.
- `CRITICAL_PACKET`: set the `required` flag for this packet in the packet state array.
- `CRITICAL_INTERVAL`: same as `CRITICAL_PACKET` but for an interval. It saves space in the header packet in case a large amount of data is critical – remember frame headers are single packet.
- `FRAME_CLOSE`: set the `closed` flag for this frame. It is not destroyed immediatly because the storage may be still in use and has to be destroyed by the protocol itself.

Data processing

The data processing in the receiver's side is the heart of *VRP*.

- if the packet is in the window (between `wait_point` and `next_exp`),then continue processing. If it is outside, discard it.

- if the packet was `waiting`, mark it as `received`, `valid`, and not `waiting`. Store the data in the storage, and don't update any loss variable, this packet was supposed to arrive.
- if the packet was not `waiting`:
  - if it was `received`, discard it, if it was `valid` and not `received`, the loss was allowed. Take it anyway and update loss counters.
  - if it was not `valid`, it is either the expected packet or one of the following ones. Increment `next_exp` one step at a time, until it reaches the sequence number of the packet we have just received. For each step, `next_exp` is the sequence number of a packet lost.
    - increment `cons_loss`
    - according to what leaves the window, increment or not `loss_cnt`
    - if `cons_loss` or `loss_cnt` are greater than what is allowed, or if the packet is marked as `required`, send NACK (actually pack a NACK into the current header – don't send a packet with only a NACK), decrement `loss_cnt` and reset `cons_loss`. Set the tag of the packet to `waiting`.
    - if the loss is allowed, mark it as `valid` (but not `received`).
  - mark the packet received as `received` and `valid`, and increment `next_exp`. Pack a single ACK into the header.
- update `wait_point` and pack a cumulative ACK with its value. If `wait_point` reaches `last_seq`, the frame is completed: call `deliver_rsr`.
- send the packed header (with various ACKs, NACKs, ...) even if the packet has been refused. This header always contains the latest cumulative ACK so that the sender will know the receiver's state.

This implementation performs exactly waht has been described in section 2.3.4. See the examples page 14.

3.5 *VRP* API

To initialize a transfer, there must be a sender (`vrp_outgoing_t`) ad a receiver (`vrp_incoming_t`). The data is packed into a *VRP* buffer (`vrp_buffer_t`) and the user specifies the loss tolerance parameters and triggers `send_frame`. It creates a `vrp_outgoing_frame_t` (outgoing frame information block) inside the sender and a `vrp_incoming_frame_t` inside the receiver. The "send" call is blocking, the "receive" operation is performed asynchronously (works like an interruption).

```
void vrp_buffer_construct(vrp_buffer_t*buffer,
                          char*data,
                          int size)
```

initializes a *VRP* buffer with the given `data` and `size`.

When the buffer is initialized, the loss tolerance parameters get no value. If they are not filled in, the frame will be sent with no such information, and the receiver will use its default values.

```
void vrp_buffer_destroy(vrp_buffer_t*buffer)
```

needs to be called to free the memory used by a `vrp_buffer_t` structure.

```
void vrp_buffer_add_critical(vrp_buffer_t*buffer,
                             char*data,
                             int size)
```

specifies that some data inside the buffer are critical and won't be lost. `data` is a pointer to such data (it must already be in the buffer), and `size` is the size of the data that is actually critical. As several parts of a buffer may be critical, this function can be called several times on the same buffer. If the buffer as not been initialized by `vrp_buffer_construct`, the result is not specified.

```
void vrp_buffer_set_loss_rate(vrp_buffer_t*buffer,
                              int max, int cons)
```

This function sets the maximum loss rate allowable for the frame currently in the buffer. `max` is the maximum percentage of data lost (0 means no loss, 255 means everything is allowed to be lost). `cons` is the length of the maximum consecutive loss (in bytes). -1 for either of these parameters means default value.

```
vrp_outgoing_t* vrp_outgoing_construct(char*hostname,
       int port,
       globus_nexus_proto_info_vrp_t*proto_info)
```

is the function that construct a sender's structure. `proto_info` is a structure that contains default loss tolerance parameters for this sender.

`void vrp_outgoing_open(vrp_outgoing_t*outgoing)` opens physically the connection. It send a ping and calculates the starting tiemouts parameters.

```
void vrp_outgoing_close(vrp_outgoing_t *outgoing)
```

destroys the sender's structure.

```
int vrp_outgoing_send_frame(vrp_outgoing_t *outgoing,
                            vrp_buffer_t *buffer)
```

This function sends the frame in the buffer `buffer` using the sender parameters `outgoing`.

```
vrp_incoming_t* vrp_incoming_construct(int*port,
                                       nexus_endpoint_t* endpoint)
```

creates a receiver and links it to a Nexus endpoint.

```
void vrp_incoming_close(vrp_incoming_t *incoming)
```

closes a receiver.

```
void nexusl_pr_vrp_incoming_deliver_rsr(vrp_incoming_t*incoming,
                                        vrp_incoming_frame_t*frame)
```

is the function that will be triggered each time a frame is received. This function is very specific to Nexus. In a standalone *VRP* package, the user will want to provide a pointer to a handler instead of a Nexus endpoint when he constructs the `vrp_incoming_t` structure, and this function wil be triggered when a frame will be completed.

3.6 *VRP* in Nexus

3.6.1 RSRs

All communications are asynchronous, the messages are delivered through *RSRs* (Remote Service Requests) that trigger a handler in the remote context.

We chose to pack an RSR in a frame – one RSR per frame, one frame per RSR. The RSR header is marked as "critical" and won't be allowed to be lost. The remaining of the data has fixed loss tolerance parameters that can't be changed from one frame to another.

As *VRP* always transmit frames, the losses will result in "holes" in the data, but the receiving application will be provided with a complete frame (with the right size). In the holes, what has been lost is actually replaced with junk data – null bytes.

3.6.2 Limitations

The receiving application gets the data through `nexus_get_*` primitives that won't provide other information than the data itself. If we want Nexus to give the application more information (eg. the actual losses), we have to rewrite a large part of its code.

Nexus aims to be a general network protocol interface and has so a generic API that won't allow any application to use such protocol-specific information.

3.6.3 Integration

There are different modules for each protocol. *VRP* is a new one called `pr_vrp.c` (actually based on `pr_udp.c`). It contains the whole *VRP* implementation and some functions that realize the interface between Nexus and *VRP*.

The program `test_vrp` is a simple test applications usable for benchmarks.

Chapter 4

Performance issues

4.1 Short distance

On a short distance connection (ie. LAN), *VRP* is almost as fast as *TCP*. It is no surprise that it is slower because it is not an easy job to beat *TCP* with a protocol which is still on top of *IP* on a LAN.

VRP is slower because it runs in user space and thus has to do a lot of system calls (send/receive, timeout checking). It is a good result since it is only 5% slower than *TCP*.

It confirms – once again, no surprise – that *VRP* has been designed for long distance connections with a high percentage of loss, but is not exactly suited for a LAN.

4.2 Long distance

4.2.1 Overall performance

The speed is simply amazing! On a long distance connection (ie. WAN), *VRP* is up to 10 times faster than *TCP* with no more than 10% residual loss.

It is very variable and depends on both sender and receiver machines. But even in the worst case measured, *VRP* is still more than 2 times faster (yes, the worst case is *only* 100% faster!).

Why is the throughput so high compared to *TCP*? Because as there is almost no flow control, *VRP* is pretty optimistic compared to *TCP* and then doesn't slow down to avoid losses since losses are allowable!

4.2.2 Some figures

Benchmarking a LAN is difficult but benchmarking a WAN is a nightmare. There can be sudden changes, and the performance may vary a lot from one day to another.

from:	to:	*TCP* (Mbit/s)	*VRP* (Mbit/s)	ratio
`flash.isi.edu`	`denali.mcs.anl.gov`	2.23	8.56	3.83
`flash.isi.edu`	`pitcairn.mcs.anl.gov`	1.05	7.98	7.60
`bolas.isi.edu`	`denali.mcs.anl.gov`	1.61	3.78	2.34
`bolas.isi.edu`	`pitcairn.mcs.anl.gov`	0.98	4.95	5.05
`denali.mcs.anl.gov`	`bolas.isi.edu`	0.93	9.56	10.27
`denali.mcs.anl.gov`	`huntsman.isi.edu`	0.88	4.28	4.86
`denali.mcs.anl.gov`	`vanuatu.isi.edu`	0.94	4.76	5.06
`pitcairn.mcs.anl.gov`	`bolas.isi.edu`	1.00	6.83	6.83
`pitcairn.mcs.anl.gov`	`vanuatu.isi.edu`	0.92	3.83	4.16
`dragon.ens-lyon.fr`	`vanuatu.isi.edu`	0.34	1.26	3.70
`bolas.isi.edu`	`dragon.esn-lyon.fr`	0.21	0.56	2.66

Table 4.1: *VRP* vs. *TCP* on a WAN

All benchmarks aim to measure throughput. We send a huge amount of data (several megabytes) using *TCP* and *VRP* and hope that while the test is running, traffic condition won't change too quickly.

Domain `isi.edu` is in L.A., California, `anl.gov` is in Chicago, Illinois, and `ens-lyon.fr` is in Lyon, France. There are 8 hops between `isi.edu` and `anl.gov`, more than 25 between `isi.edu` and `ens-lyon.fr`.
We used frames of 100 KB, reliability parameters were:

- maximum burst loss size: 4 KB
- maximum loss rate: 25%
- first kilobyte is critical

The measured loss rate is actually 10%.

4.2.3 Link between reliability and throughput

We decrease reliability so that speed increases. True? Actually we don't...

In the previous tests, *VRP* is more than 10% faster than *TCP*, with only 10% loss. If we set the loss rate to 0%, *TCP* is still far behind!

The performance is slightly unpredictable: it can be either as fast as with an allowable loss rate of 25% or up to 30% slower, depending on the actual loss rate of the network itself.

Even with 0% allowable loss, *VRP* is still up to 8 times faster than *TCP*, and still 50% faster in the worst case.

4.2.4 Latency

Everything has been optimized for throughput. What about latency? It is a critical issue with a cluster or a LAN, but not really with a WAN with which latency is high anyway.

For frames small enough so that throughput is not an issue, we measure approximatly the same latency for *TCP* and *VRP*.

Chapter 5

Conclusion

5.1 What is done

The *VRP* protocol proposed has been successfully implemented and improved. It is now able to act in multi-threaded applications and is rather well error tolerant (broken connection, program or machine crash...).

When used in the context it was designed for, it is amazingly fast compared to the all purpose *TCP* protocol. It is not explicitly designed for small messages though it will do the job quite well, and is not perfectly suited for communication over a LAN where it is slightly slower than *TCP*. But large amounts of data over a WAN don't scare it and the througput will be astonishing.

VRP is now a part of Nexus so that every Globus application can use it. However its power is a little curbed when in Nexus, *VRP* can show what it is really worth only when used directly by an application.

5.2 What remains to be done

There might be some improvement in the memory management:

- free everything but the storage when an incoming frame information block is closed but cannot be destroyed immediatly, instead of keeping everything in memory.

- allocate dynamically frame information blocks instead of having a fixed array. There should still remain a maximum number to avoid memory shortage in case frames can't be delivered and are accumulating.

A few things may improve network performance:

- send a single packet if header and data can fit into the maximum packet size allowed. It reduces latency for small frames, though small frames are not the primary aim of *VRP*;

- find an algorithm to adapt dynamically the sender's window size and/or packet size, without the pessimistic approach that drastically slows down *TCP*.

Improve the communication between *VRP* and the application in Nexus.

- tell the application what has been actually lost;
- provide the sending application with a true feedback for it to adapt its networks requirements;
- enable the application to give loss tolerance parameters for each frame and not only at the connection opening.

That is definitely the main point: *VRP* is able to provide the application with a lot of feedback information, and the application can tell *VRP* specific loss tolerance parameters, specific window or packet size, or critical packets, but what is very disappointing is that Nexus can't use these features without a lot of changes.

It will be possible in the standalone *VRP* package (without Nexus) if ever it will be released. Check `http://www.ens-lyon.fr/ denisa/work/VRP/`.

Appendix A

Data structures

A.1 *VRP* buffer

```
struct vrp_critical_cell_s
{
  int begin, end;
  struct vrp_critical_cell_s*next;
};

struct vrp_buffer_s
{
  int  size;
  char*data;
  struct vrp_critical_cell_s*critical;
  int max_loss_rate;
  int max_cons_loss;
};
typedef struct vrp_buffer_s vrp_buffer_t;
```

A.2 Sender

```
enum vrp_outgoing_state_e
{
  VRP_OUTGOING_INIT,
  VRP_OUTGOING_OK,
  VRP_OUTGOING_CLOSING,
  VRP_OUTGOING_CLOSED
};

/*
 * vrp_outgoing_t
 *
 * This is an overload of nexus_proto_t.  It adds the
```

```
 * vrp specific information to that structure.
 */
struct vrp_outgoing_s
{
  nexus_proto_type_t                type;
  nexus_proto_funcs_t *        funcs;
  int                                 version;
  unsigned long                direct_custom_min_size;
  unsigned long                direct_custom_max_size;
  unsigned long                direct_pointer_min_size;
  unsigned long                direct_pointer_max_size;
  nexus_bool_t                can_use_iovec;
  unsigned long                reserved_header_size;
  /* ---- end of common part */
  enum vrp_outgoing_state_e state;
  /* system */
  int fd;
  struct sockaddr_in local_addr;
  struct sockaddr_in outgoing_addr;
  struct hostent*host;
  char * hostname;
  int port;
  int callback_registered;

  /* dynamic timeout adjustment */
  int timeout;         /* current timeout, in usec */
  long long int RTT;   /* Round trip time, in usec */
  long long int average_deviation;

  struct vrp_outgoing_frame_s frames[VRP_OUTGOING_NUM_FRAMES];
  int last_frame_num;

  int max_cons_loss;
  int max_loss_rate;

  int n_read;
  char recv_buf[VRP_MAX_PACKET_SIZE];

  int fault_code;
  int reference_count;

#ifdef BUILD_RESOURCE
  nexus_resource_name_t        resource_name;
#endif /* BUILD_RESOURCE */
};
typedef struct vrp_outgoing_s vrp_outgoing_t;
```

A.3 Outgoing frame information block

```
enum vrp_outgoing_frame_state_e
{
  VRP_OUTGOING_FRAME_NONE,       /* no frame in this slot */
  VRP_OUTGOING_FRAME_INIT,       /* sending frame header */
  VRP_OUTGOING_FRAME_SEND,       /* sending data */
  VRP_OUTGOING_FRAME_REFUSED,    /* frame refused by incoming */
  VRP_OUTGOING_FRAME_ERROR,      /* transmission error */
  VRP_OUTGOING_FRAME_CLOSING     /* all data sent */
};

struct vrp_outgoing_frame_s
{
  int frame_num;
  enum vrp_outgoing_frame_state_e state;
  struct timeval deadline;

  /* sliding window */
  int ack_expected;
  int next_to_send;
  /* buffers */
  int next_to_buffer;
  struct vrp_outgoing_buffer_cell_s buffer[VRP_OUTGOING_NUM_BUFS];
  char send_buf[VRP_MAX_PACKET_SIZE];
  /* data */
  struct vrp_buffer_s*src;
  int max_seq;
  int data_per_packet;
  /* frame header */
  char frame_hdr[VRP_MAX_PACKET_SIZE];
  int frame_hdr_size; /* actual size */
  struct timeval frame_hdr_timeout;
};
typedef struct vrp_outgoing_frame_s vrp_outgoing_frame_t;
```

A.4 Receiver

```
/*
 * vrp_incoming_t
 *
 * One of these is allocated for each incoming file descriptor.
 */
struct vrp_incoming_s
{
  enum vrp_incoming_state_e state;

  /* current frames */
  struct vrp_incoming_frame_s frames[VRP_INCOMING_NUM_FRAMES];
```

```
  /* buffer for last received */
  char recv_buf[VRP_MAX_PACKET_SIZE];
  int n_read;

  /* id of last received */
  struct sockaddr from_addr;
  int from_len;

  /* system */
  int fd;
  struct sockaddr_in local_addr;
  int callback_registered;

  /* nexus specific stuff */
  nexus_bool_t dispatch_in_progress;
  struct globus_nexus_buffer_s* dispatch_q_head;
  struct globus_nexus_buffer_s* dispatch_q_tail;
  nexus_endpoint_t* endpoint;
};
typedef struct vrp_incoming_s vrp_incoming_t;
```

A.5 Incoming frame information block

```
struct vrp_incoming_frame_state_s
{
  unsigned int in_use:    1;
  unsigned int receiving: 1;
  unsigned int deliver:   1;
  unsigned int closed:    1;
};


struct vrp_incoming_buffer_cell_s
{
  unsigned int       received: 1;
  unsigned int       valid:    1;
  unsigned int       acked:    1;
  unsigned int       waiting:  1;
  unsigned int       required: 1;
};
typedef struct vrp_incoming_buffer_cell_s vrp_incoming_buffer_cell_t;

struct vrp_incoming_frame_s
{
  /* my id */
  struct sockaddr from_addr;
  int from_len;
  int frame_num;
```

```
  /* state */
  struct vrp_incoming_frame_state_s state;
  struct timeval deadline;
  /* storage */
  char *storage;
  int storage_size;
  /* sliding window */
  int next_exp;
  int waiting;
  struct vrp_incoming_buffer_cell_s*buffer;

  /* losses statistics */
  int loss_max;
  int cons_loss_max;
  int loss_cnt;
  int total_loss;

  /* data expected */
  int data_per_packet;
  int data_size;
  int last_seq;   /* last seq number of the frame */
};
typedef struct vrp_incoming_frame_s vrp_incoming_frame_t;
```

Appendix B

Source code

The entire source code is more than 3000 lines long. What is given here is only a sample of the most interesting parts.

B.1 In the sender

B.1.1 Packet processing

```
static void
nexusl_pr_vrp_outgoing_process_fragment(vrp_outgoing_t*outgoing)
{
  vrp_hdr_t hdr;
  vrp_outgoing_frame_t*frame;
  int i;
  int seq;
  char*last_byte, *c, *prev;
  struct timeval date;
  int sec, usec;
  struct vrp_outgoing_buffer_cell_s*cell;

  vrp_hdr_unpack(outgoing->recv_buf, &hdr);
  debugmsg(1,"vrp_outgoing_process_fragment() frame = %i seq = %i\n", hdr.frame, hdr.seq);
  c = (char*)outgoing->recv_buf + VRP_HDR_SIZE;
  last_byte = (char*)outgoing->recv_buf + outgoing->n_read - 1;
  if(hdr.frame != VRP_NO_FRAME)
    frame = nexusl_pr_vrp_outgoing_frame_lookup(outgoing, hdr.frame);
  else
    frame = NULL;
  if(!hdr.is_hdr)
    goto badheader;

  while(*c != VRP_TAG_LAST)
    {
      if(c > last_byte)
        {
          goto badheader;
        }
      prev = c;
```

```
if(frame != NULL)
  {
    switch(*c)
      {
      case VRP_TAG_ACK:
        if(c + 4 > last_byte)
          goto badheader;
        seq = vrp_unpack_int32(c+1);
        debugmsg(1,"frame %i seq %i ACK\n", frame->frame_num, seq);
        if((frame->ack_expected <= seq) &&
           (seq < frame->next_to_send))
          {
            cell = &frame->buffer[seq & VRP_OUTGOING_BUF_MASK];
            for(i = frame->ack_expected; i <= seq; i++)
              {
                frame->buffer[i & VRP_OUTGOING_BUF_MASK].acked = NEXUS_TRUE;
              }
            frame->ack_expected = seq + 1;
            nexusl_pr_vrp_outgoing_frame_alive(frame);
            if(!cell->expired)
              nexusl_pr_vrp_outgoing_adapt_timeout(outgoing, &cell->expire_date,
                                           cell->timeout_val);
          }
        else if(seq >= frame->next_to_send)
          {
            frame->state = VRP_OUTGOING_FRAME_ERROR;
          }
        c += 5;
        break;
      case VRP_TAG_NACK:
        if(c + 4 > last_byte)
          goto badheader;
        seq = vrp_unpack_int32(c+1);
        debugmsg(1,"frame %i seq %i NACK\n", frame->frame_num, seq);
        if((frame->ack_expected <= seq) &&
           (seq < frame->next_to_send))
          {
            nexusl_pr_vrp_outgoing_send_packet(outgoing, frame, hdr.seq);
          }
        else if(seq >= frame->next_to_send)
          {
            frame->state = VRP_OUTGOING_FRAME_ERROR;
          }
        c += 5;
        break;
      case VRP_TAG_SINGLE_ACK:
        if(c + 4 > last_byte)
          goto badheader;
        seq = vrp_unpack_int32(c+1);
        debugmsg(1,"frame %i seq %i single ACK\n", frame->frame_num, seq);
        if((frame->ack_expected <= seq) &&
           (seq < frame->next_to_send))
          {
            cell = &frame->buffer[seq & VRP_OUTGOING_BUF_MASK];
            cell->acked = NEXUS_TRUE;
            if(!cell->expired)
              nexusl_pr_vrp_outgoing_adapt_timeout(outgoing, &cell->expire_date,
```

```
                                      cell->timeout_val);
            nexusl_pr_vrp_outgoing_frame_alive(frame);
          }
        else if(seq >= frame->next_to_send)
          {
            frame->state = VRP_OUTGOING_FRAME_ERROR;
          }
        c += 5;
        break;
      default:
        break;
      }
    /* update ACK info */
    while((frame->ack_expected < frame->next_to_send) &&
          (frame->buffer[frame->ack_expected & VRP_OUTGOING_BUF_MASK].acked))
      {
        frame->ack_expected++;
        assert(frame->ack_expected <= frame->max_seq + 1);
      }
    if(frame->ack_expected > frame->max_seq)
      {
        frame->state = VRP_OUTGOING_FRAME_CLOSING;
        debugmsg(1,"vrp_outgoing_process_fragment(): frame %i closed\n",
                 frame->frame_num);
      }
  }
switch(*c)
  {
  case VRP_TAG_PONG:
    if(c + 8 > last_byte)
      goto badheader;
    if(outgoing->state == VRP_OUTGOING_INIT)
      {
        sec = vrp_unpack_int32(c+1);
        usec = vrp_unpack_int32(c+5);
        gettimeofday(&date, NULL);
        outgoing->RTT = (date.tv_usec - usec + 1000000 * (date.tv_sec - sec));
        outgoing->average_deviation = outgoing->RTT / 2;
        outgoing->timeout = outgoing->RTT + 4 * outgoing->average_deviation;
        if((outgoing->timeout > 0) && (outgoing->timeout < VRP_MAX_TIMEOUT))
          {
            outgoing->state = VRP_OUTGOING_OK;
            debugmsg(1,"vrp_outgoing_process_fragment(): timeout = %i usec\n",
                     outgoing->timeout);
          }
      }
    c += 9;
    break;
  case VRP_TAG_FRAME_SET:
    if(c + 2 > last_byte)
      goto badheader;
    seq = vrp_unpack_int16(c+1);
    frame = nexusl_pr_vrp_outgoing_frame_lookup(outgoing, seq);
    if(frame == NULL)
      goto badheader;
    c += 3;
      break;
```

```
        case VRP_TAG_FRAME_OK:
          if(c + 2 > last_byte)
            goto badheader;
          seq = vrp_unpack_int16(c+1);
          debugmsg(1,"vrp_outgoing_process_fragment(): frame accepted %i\n", seq);
          frame = nexusl_pr_vrp_outgoing_frame_lookup(outgoing, seq);
          if(frame != NULL)
            frame->state = VRP_OUTGOING_FRAME_SEND;
          else
            {
              goto badheader;
            }
          c += 3;
          break;
        case VRP_TAG_FRAME_REFUSED:
          if(c + 2 > last_byte)
            goto badheader;
          seq = vrp_unpack_int16(c+1);
          debugmsg(1,"vrp_outgoing_process_fragment(): frame refused %i\n", seq);
          frame = nexusl_pr_vrp_outgoing_frame_lookup(outgoing, seq);
          if(frame != NULL)
            frame->state = VRP_OUTGOING_FRAME_REFUSED;
          else
            {
              goto badheader;
            }
          c += 3;
          break;
        case VRP_TAG_FRAME_MISSING_HDR:
          if(c + 2 > last_byte)
            goto badheader;
          seq = vrp_unpack_int16(c+1);
          frame = nexusl_pr_vrp_outgoing_frame_lookup(outgoing, seq);
          debugmsg(1,"vrp_outgoing_process_fragment(): frame missing header %i\n", seq);
          if(frame != NULL)
            nexusl_pr_vrp_outgoing_frame_send_hdr(outgoing, frame);
          else
            {
              goto badheader;
            }
          c += 3;
          break;
        default:
          break;
        }
      if(prev == c)
        {
          goto badheader;
        }
    }
  return;
badheader:
  debugmsg(1,"vrp_outgoing_process_fragment(): bad header (frame %i)\n", hdr.frame);
  return;
}
```

B.1.2 Sliding window manager

```
static void
nexusl_pr_vrp_outgoing_send_routine(vrp_outgoing_t*outgoing,
                                    vrp_outgoing_frame_t*frame)
{
  struct vrp_outgoing_buffer_cell_s*cell;
  struct vrp_hdr_s hdr;
  int i;
  int first_timeout;

  debugmsg(0,"vrp_outgoing_send_routine() ack_expected = %i, next_to_send = %i, next_to_buffer = %i, max_seq = %i state
           frame->ack_expected, frame->next_to_send,
           frame->next_to_buffer, frame->max_seq, frame->state);
  /* clean buffers */
  while((frame->next_to_buffer < frame->ack_expected + VRP_OUTGOING_NUM_BUFS - 1) &&
        (frame->next_to_buffer <= frame->max_seq))
    {
      cell = &frame->buffer[frame->next_to_buffer & VRP_OUTGOING_BUF_MASK];
      cell->acked = NEXUS_FALSE;
      cell->sent = NEXUS_FALSE;
      cell->expired = NEXUS_FALSE;
      frame->next_to_buffer++;
    }

  /* check ACK timeout */
  first_timeout = NEXUS_TRUE;
  for(i = frame->ack_expected; i < frame->next_to_send; i++)
    {
      cell = &frame->buffer[i & VRP_OUTGOING_BUF_MASK];
      if(!cell->acked)
        if(vrp_check_timeout(&cell->expire_date))
          {
            debugmsg(1, "Timeout for packet %i/%i in frame %i (val %i)\n",
                     i, frame->max_seq, frame->frame_num, outgoing->timeout);
            if(first_timeout)
              nexusl_pr_vrp_outgoing_timeout_expired(outgoing);
            first_timeout = NEXUS_FALSE;
            nexusl_pr_vrp_outgoing_send_packet(outgoing, frame, i);
          }
    }

  /* send packets */
  while((frame->next_to_send < frame->next_to_buffer) &&
        (frame->next_to_send < frame->ack_expected + VRP_OUTGOING_WINDOW_SIZE))
    {
      nexusl_pr_vrp_outgoing_send_packet(outgoing, frame, frame->next_to_send);
      frame->next_to_send++;
    }
}
```

B.2 In the receiver

B.2.1 Header processing

```
static void
```

```
nexusl_pr_vrp_incoming_process_hdr(vrp_incoming_t *incoming,
                                   vrp_incoming_frame_t* frame)
{
  char *msg, *old_msg;
  int tmp_frame_num;
  int i, b, e;
  struct vrp_incoming_frame_s*lookup_frame;
  char*last_byte;
  char ack_buf[VRP_MAX_PACKET_SIZE];
  int ack_size;

  debugmsg(1,"vrp_incoming_process_hdr()\n");
  msg = (char*)incoming->recv_buf + VRP_HDR_SIZE;
  last_byte =  (char*)incoming->recv_buf + VRP_MAX_PACKET_SIZE - 1;

  while(*msg != VRP_TAG_LAST)
    {
      if(msg > last_byte)
        goto badheader;
      old_msg = msg;
      if(frame != NULL)
        {
          /* there is a current frame */
          switch(*msg)
            {
            case VRP_TAG_CONS_LOSS:
              if(msg + 1 > last_byte)
                goto badheader;
              frame->cons_loss_max = vrp_unpack_int8(msg + 1);
              msg += 2;
              debugmsg(1,"vrp_incoming_process_hdr(): consecutive losses %i\n",
                       frame->cons_loss_max);
              break;
            case VRP_TAG_LOSS_RATE:
              if(msg + 1 > last_byte)
                goto badheader;
              frame->loss_max = (VRP_INCOMING_WINDOW_SIZE * vrp_unpack_int8(msg + 1)) >> 8;
              debugmsg(1,"vrp_incoming_process_hdr(): losses %i\n", frame->loss_max);
              msg += 2;
              break;
            case VRP_TAG_CRITICAL_PACKET:
              if(msg + 4 > last_byte)
                goto badheader;
              b = vrp_unpack_int32(msg+1);
              if((b >= 0) && (b <= frame->last_seq))
                frame->buffer[b].required = NEXUS_TRUE;
              else
                goto badheader;
              msg += 5;
              debugmsg(1,"vrp_incoming_process_hdr(): critical packet %i\n", b);
              break;
            case VRP_TAG_CRITICAL_INTERVAL:
              if(msg + 8 > last_byte)
                goto badheader;
              b = vrp_unpack_int32(msg+1);
              e = vrp_unpack_int32(msg+5);
              if((b >= 0) && (e <= frame->last_seq) && (b <= e))
```

```
          {
            for(i = b; i<= e; i++)
              frame->buffer[i].required = NEXUS_TRUE;
          }
        else
          goto badheader;
        msg += 9;
        debugmsg(1,"vrp_incoming_process_hdr(): critical interval (%i - %i)\n",
                 b, e);
        break;
      default:
        break;
      }
  }
switch(*msg)
  {
  case VRP_TAG_LAST:
    break;
  case VRP_TAG_PING:
    if(msg + 8 > last_byte)
      goto badheader;
    b = vrp_unpack_int32(msg + 1);
    e = vrp_unpack_int32(msg + 5);
    vrp_hdr_pack_std(ack_buf, &ack_size);
    vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_PONG, b, e);
    vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_LAST);
    nexusl_pr_vrp_incoming_hdr_send(incoming, ack_buf, ack_size);
    msg += 9;
     break;
  case VRP_TAG_FRAME_CREATE:
    if(msg + 8 > last_byte)
      goto badheader;
    tmp_frame_num = vrp_unpack_int16(msg + 1);
    frame = nexusl_pr_vrp_incoming_frame_lookup(incoming,
      &incoming->from_addr, incoming->from_len, tmp_frame_num);
    if(frame != NULL)             /* duplicated frame header - discard */
      {
        debugmsg(2, "CREATE_FRAME: duplicated\n");
        goto badheader;
      }
    frame = nexusl_pr_vrp_incoming_frame_slot_lookup(incoming);
    if(frame != NULL)
      {
        nexusl_pr_vrp_incoming_frame_construct(incoming, frame, tmp_frame_num);
        debugmsg(0,"vrp_incoming_process_hdr(): CREATE_FRAME %i\n", tmp_frame_num);
        frame->data_size = vrp_unpack_int32(msg + 3);
        frame->data_per_packet = vrp_unpack_int16(msg + 7);
        frame->last_seq = (frame->data_size - 1) / frame->data_per_packet;
        frame->storage_size = frame->data_per_packet * (1 + frame->last_seq);
        /* storage allocated in multiple of packet size */
        debugmsg(0,"Allocating %i bytes\n", frame->storage_size);
        frame->storage = (char*) malloc(frame->storage_size);
        if(frame->storage == NULL)
          fatalerror("CREATE_FRAME: Out of memory\n");
        frame->buffer = (vrp_incoming_buffer_cell_t*)
          malloc(sizeof(vrp_incoming_buffer_cell_t) * (frame->last_seq + 2));
        if(frame->buffer == NULL)
```

```
              fatalerror("CREATE_FRAME: Out of memory\n");
            for(i = 0; i <= frame->last_seq; i++)
              {
                frame->buffer[i].received  = 0;
                frame->buffer[i].valid     = 0;
                frame->buffer[i].acked     = 0;
                frame->buffer[i].waiting   = 0;
                frame->buffer[i].required  = 0;
              }
            frame->state.in_use = 1;
            frame->state.receiving = 1;
            frame->state.deliver = 0;
            frame->state.closed = 0;
            debugmsg(1,"vrp_incoming_process_hdr():"
                     " CREATE_FRAME data size = %i, per packet = %i, last seq = %i, storage: %i\n",
                     frame->data_size, frame->data_per_packet,
                     frame->last_seq, frame->storage_size);
            vrp_hdr_pack_std(ack_buf, &ack_size);
            vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_FRAME_OK, frame->frame_num);
            vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_LAST);
            nexusl_pr_vrp_incoming_hdr_send(incoming, ack_buf, ack_size);
          }
        else
          {
            vrp_hdr_pack_std(ack_buf, &ack_size);
            vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_FRAME_REFUSED, tmp_frame_num);
            vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_LAST);
            nexusl_pr_vrp_incoming_hdr_send(incoming, ack_buf, ack_size);
            debugmsg(2,"vrp_incoming_process_hdr() CREATE_FRAME: no slot left\n");
          }
        msg += 9;
        break;
      case VRP_TAG_FRAME_CLOSE:
        if(msg + 2 > last_byte)
          goto badheader;
        tmp_frame_num = vrp_unpack_int16(msg+1);;
        debugmsg(1,"vrp_incoming_process_hdr(): close frame %i\n", tmp_frame_num);
        lookup_frame = nexusl_pr_vrp_incoming_frame_lookup(incoming,
           &incoming->from_addr, incoming->from_len, tmp_frame_num);
        lookup_frame->state.closed = 1;
        msg +=3;
        break;
      default:
        break;
      }
    if(old_msg == msg)
      goto badheader;
  }
  return;
badheader:
  debugmsg(2,"vrp_incoming_process_hdr(): bad frame header detected.\n");
}
```

B.2.2 Data processing

```
static void
nexusl_pr_vrp_incoming_process_data(vrp_incoming_t *incoming,
```

```
                                  vrp_incoming_frame_t*frame,
                                  vrp_hdr_t *hdr)
{
  vrp_incoming_buffer_cell_t state;
  int cons_loss;
  int seq;
  char ack_buf[VRP_MAX_PACKET_SIZE];
  int ack_size;

  debugmsg(1,"vrp_incoming_process_data() frame = %i, seq = %i / %i, size = %i\n",
          hdr->frame, hdr->seq, frame->last_seq, incoming->n_read );

  vrp_hdr_pack_std(ack_buf, &ack_size); /* prepare ACK/NACK/... header */
  vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_FRAME_SET, hdr->frame);
  vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_SINGLE_ACK, hdr->seq);

  if((frame->waiting <= hdr->seq) && (hdr->seq <= frame->last_seq))
    {
      /* in the window. Continue... */
      state = frame->buffer[hdr->seq];
      if(!state.waiting)
        { /* not outstanding */
          if(state.valid)
            { /* already validated? Uh?!? */
              if(state.received)
                {
                  /* message duplicated */
                }
              else
                {
                  /* loss was allowed - take it anyway */
                  frame->buffer[hdr->seq].received = NEXUS_TRUE;
                  frame->loss_cnt--;
                  frame->total_loss--;
                  assert(hdr->seq <= frame->last_seq);
                  assert(incoming->n_read - VRP_HDR_SIZE <= frame->data_per_packet);
                  memcpy(&frame->storage[frame->data_per_packet * hdr->seq],
                         incoming->recv_buf + VRP_HDR_SIZE,
                         incoming->n_read - VRP_HDR_SIZE);
                }
            }
          else
            {
              /* packet not yet validated, not outstanding: usual case */
              /* ** update losses counters */
              cons_loss=0;
              for(seq = frame->next_exp; seq < hdr->seq; seq++)
                {
                  cons_loss++;
                  frame->total_loss++;
                  if((seq < VRP_INCOMING_WINDOW_SIZE) ||
                     (frame->buffer[seq - VRP_INCOMING_WINDOW_SIZE].received))
                    frame->loss_cnt++;
                  if((cons_loss > frame->cons_loss_max) ||
                     (frame->loss_cnt > frame->loss_max) ||
                     (frame->buffer[seq].required))
                    {
```

```
                /* a rule is broken */
                debugmsg(1, "Broken rule for message %i\n", seq);
                vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_NACK, seq);
                cons_loss = 0;
                frame->buffer[seq].waiting = NEXUS_TRUE;
                frame->loss_cnt--;
              }
            else
              {
                /* loss is allowed */
                debugmsg(1,"Allowed loss for message %i\n", seq);
                vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_SINGLE_ACK, seq);
                frame->buffer[seq].valid = NEXUS_TRUE;
              }
          }
        if((hdr->seq >= VRP_INCOMING_WINDOW_SIZE) &&
           (!frame->buffer[hdr->seq - VRP_INCOMING_WINDOW_SIZE].received))
          frame->loss_cnt--;
        /* ** validate packet */
        frame->next_exp = hdr->seq + 1;
        frame->buffer[hdr->seq].received = NEXUS_TRUE;
        frame->buffer[hdr->seq].valid    = NEXUS_TRUE;
        assert(hdr->seq <= frame->last_seq);
        assert(incoming->n_read - VRP_HDR_SIZE <= frame->data_per_packet);
        memcpy(&frame->storage[frame->data_per_packet * hdr->seq],
               incoming->recv_buf + VRP_HDR_SIZE,
               incoming->n_read - VRP_HDR_SIZE);
      }
  }
else
  { /* outstanding packet */
    /* don't check the rules, it's ok... */
    frame->buffer[hdr->seq].received = NEXUS_TRUE;
    frame->buffer[hdr->seq].valid    = NEXUS_TRUE;
    frame->buffer[hdr->seq].waiting  = NEXUS_FALSE;
    assert(hdr->seq <= frame->last_seq);
    assert(incoming->n_read - VRP_HDR_SIZE <= frame->data_per_packet);
    memcpy(&frame->storage[frame->data_per_packet * hdr->seq],
           incoming->recv_buf + VRP_HDR_SIZE,
           incoming->n_read - VRP_HDR_SIZE);
    frame->total_loss--;
  }
/*  the packet is in the window (continued) */
/* ** update 'waiting', send ACKs */

while((frame->buffer[frame->waiting].valid) &&
      (frame->waiting < frame->next_exp))
  {
    frame->buffer[frame->waiting].acked = NEXUS_TRUE;
    frame->waiting++;
    debugmsg(1,"Now waiting %i\n", frame->waiting);
    assert(frame->waiting <= frame->last_seq + 1);
  }
/* if we get there, [waiting] not ACKed or waiting == next_exp */
/* ** check end of frame */
if((frame->waiting > frame->last_seq) && (frame->state.receiving))
  {
```

```
          frame->state.receiving = 0;
          nexusl_pr_vrp_incoming_deliver_rsr(incoming, frame);
        }
    }
  /* global ACK, close and send packet */
  if(frame->waiting > 0)
    {
      debugmsg(1,"Send ACK #%i \n", frame->waiting - 1);
      vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_ACK, frame->waiting - 1);
    }
  vrp_hdr_pack_tag(ack_buf, &ack_size, VRP_TAG_LAST);
  nexusl_pr_vrp_incoming_hdr_send(incoming, ack_buf, ack_size);

}
```

Bibliography

[1] Hari Balakrishnan and Randy H. Katz. Explicit loss notification and wireless web performance. Ieee globecom internet mini-conference, University of California at Berkeley, 1998.

[2] Robert Olson Ian Foster, Carl Kesselman and Steven Tuecke. *Nexus: an interoperability layer for parallel and distributed computer systems*, 1996.

[3] Jon Postel. Transmission control protocol, darpa internet program, protocol specification. Rfc 793, Information Sciences Institute, University of Southern California, Marina del Rey, California 90292, USA, september 1981.

[4] P. Krishnan Robin Kravets, Ken Calvert and Karsten Schwan. Adaptive variation of reliability. Technical report, Georgia Institute of Technology, Atlanta, Georgia, USA, 1996.

[5] W. Richards Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[6] Andrew Tanenbaum. *Réseaux*. Prentice Hall, 1997.

[7] L. Zhang. Why tcp timers don't work well. In *ACM SIGCOMM*, August 1986.