Variable Reliability Protocol: A protocol with a tunable loss tolerance for high performance over a WAN.

Alexandre Denis

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n^{o} 8512

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

SPI

Variable Reliability Protocol: A protocol with a tunable loss tolerance for high performance over a WAN

Alexandre Denis

March 2000

Research Report N^{o} 2000-11

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

Variable Reliability Protocol: A protocol with a tunable loss tolerance for high performance over a WAN

Alexandre Denis

March 2000

Abstract

Applications often have to chose between the slow *TCP* and the unreliable *UDP*. Robin Kravets from the GaTech has proposed an alternative: the Variable Reliability Protocol (*VRP*). The applications can specify what the allowable data losses are, and the protocol guarantees that the given loss tolerance parameters are respected. It should result in a faster throughput over WAN which have typically a higher loss rate.
In this paper we describe how *VRP* has been improved, implemented, and integrated into Nexus, the communication library of the meta-computing environment Globus.

Keywords: TCP, UDP, loss tolerance, network protocol, Globus, Nexus, WAN, Internet

Résumé

Les applications doivent jusqu'à présent choisir entre *TCP*, fiable mais lent, et *UDP*, rapide mais non-fiable. Robin Kravets du GaTech a proposé une alternative : le protocole à fiabilité variable (*Variable Reliability Protocol – VRP*). Les applications peuvent spécifier quelles sont les pertes tolérables dans les données, et le protocole garantit que les paramètres de tolérance de pertes seront respectés. Il en résulte un débit plus élevé sur les WAN qui ont typiquement un taux de pertes élevé.
Nous décrivons dans cet article comment nous l'avons amélioré, implémenté et intégré à Nexus, la couche de communication de l'environnement de meta-computing Globus.

Mots-clés: TCP, UDP, tolérance de pertes, protocole réseau, Globus, Nexus, WAN, Internet

Variable Reliability Protocol: A protocol with a tunable loss tolerance for high performance over a WAN

Alexandre Denis[*]

March 2, 2000

Abstract

Applications often have to chose between the slow *TCP* and the unreliable *UDP*. Robin Kravets from the GaTech has proposed an alternative: the Variable Reliability Protocol (*VRP*). The applications can specify what the allowable data losses are, and the protocol guarantees that the given loss tolerance parameters are respected. It should result in a faster throughput over WAN which have typically a higher loss rate.

In this paper we describe how *VRP* has been improved, implemented, and integrated into Nexus, the communication library of the meta-computing environment Globus.

Keywords: *TCP, UDP, loss tolerance, network protocol, Globus, Nexus, WAN, Internet.*

[*] LIP, ENS Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France. Email: `Alexandre.Denis@ens-lyon.fr`.

Contents

1 Introduction

This results have been worked out at the USC / ISI[1] from June through August 1999, under the supervision of Dr. Carl KESSELMAN[2].

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 multi-method communication, providing an application a single API to a wide range of communication protocols and characteristics.

Nexus is a portable library providing the multi-threaded 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.

Refer to `http://www.globus.org` for more information about the Globus project.

1.2 Variable reliability

1.2.1 Aim of variable reliability

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

The two types of reliability provided by *TCP* and *UDP* are not sufficient. Facing this restricted choice, most applications use reliability, even though they do not absolutely need it. It results in lower performance than what we could expect, partially due to many retransmissions. The messages do 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. The application could then adjust its requirements to the available network resources.

[1] University of Southern California / Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292-6695, USA, `http://www.isi.edu`.

[2] E-mail: `carl@isi.edu`.

1.2.2 Loss tolerance

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. Our goal protocol provides the application with the ability to define a level of allowable losses within the data transmitted. 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 (*e.g.*, to interpolate the missing data).

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

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

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 we proposed in order to increase performance and to adapt *VRP* to Nexus. Robin Kravets' implementation will be referred as "original" implementation, while the one we propose will be called "improved".

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.

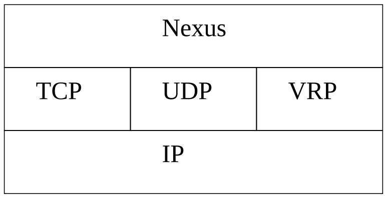


Figure 1: Conceptual position of *VRP* in the Internet model

However *VRP* is not implemented in the kernel of the operating system but in user space. As we are in user space, the only way to get *IP* service is to use *UDP*. Observe that *VRP* could be implemented in any protocol stack that follows the OSI model.

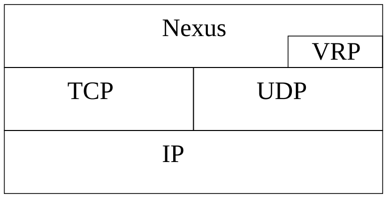


Figure 2: Actual position of *VRP* in the Internet model

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 splits this frames into *UDP* packets with fixed size. Why datagram oriented? As the application specifies data boundaries, the sliding window and loss tolerance parameters are reinitialized at the beginning of each frame, since there is often no logical link between the different frames. The original protocol used a connected mode.

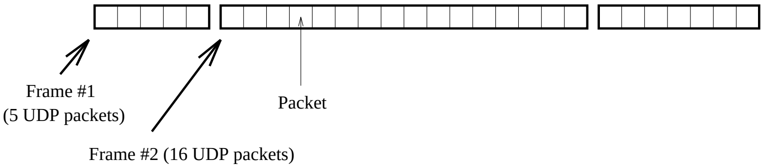


Figure 3: Data frames split into packets

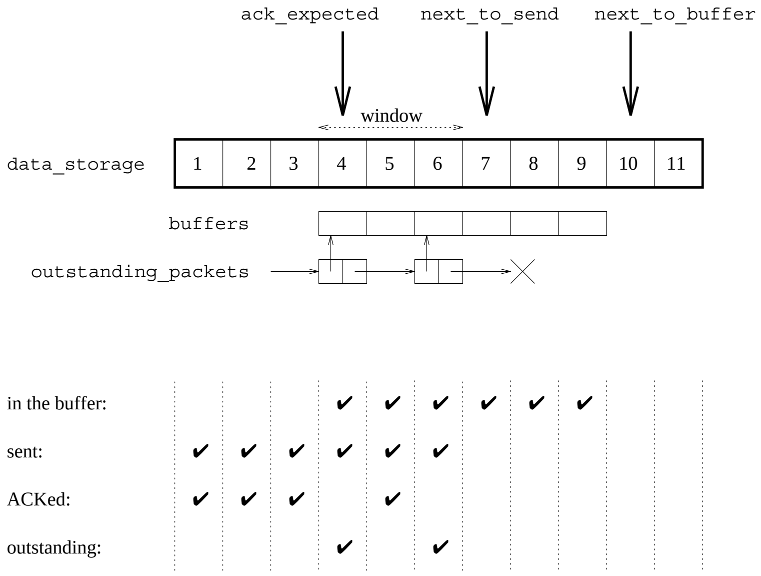
Sliding window. Packets are transmitted using a simple sliding window mechanism: the sender sends packets in order, and waits for acknowledgments. The window size represents the maximum number of unacknowledged packets. For each packet, a timer is started. When the timeout expires, the packet is sent 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 acknowledgment* (NACK, and the derived verb "to NACK") to let the sender send this packet again immediately. 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[4].

2.2 A preliminary implementation

2.2.1 Sender side

The preliminary implementation used a sliding window with a list of outstanding messages. Essentially, when data enters the window, it is copied into 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 4 shows a valid state for a sender.



	1	2	3	4	5	6	7	8	9	10	11
in the buffer:				✔	✔	✔	✔	✔	✔		
sent:	✔	✔	✔	✔	✔	✔					
ACKed:	✔	✔	✔		✔						
outstanding:				✔		✔					

Figure 4: Example of sender's state

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 been 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 was the well-known sliding window mechanism;
- an "history window" to keep track of the recent packet losses, in order to check whether the rules are broken.

hold_point is the lower border of the history window. It is the eldest message lost in the receive history, and must be at least wait_point minus the history window size.

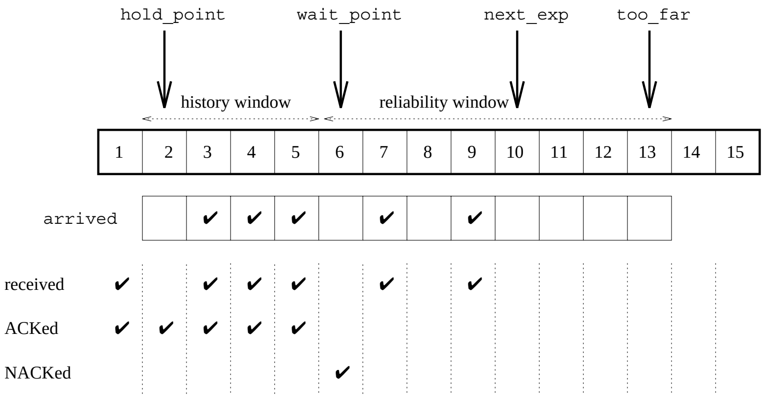


Figure 5: Example of receivers's state

wait_point is the lower border of the reliability window. It indicates the message we are currently 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 rejected.

arrived is the bitmap of messages arrived 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, then 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 rejected.

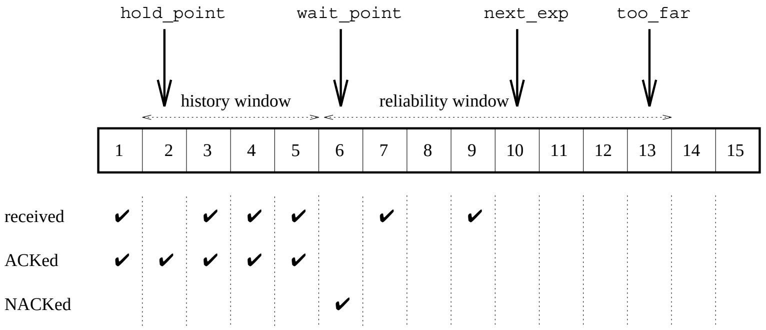


Figure 6: Without anticipation

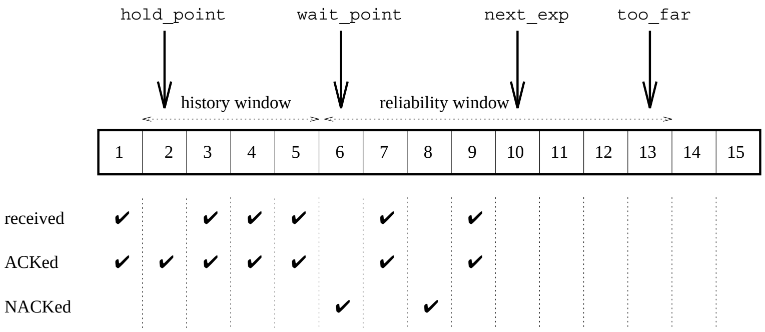


Figure 7: With anticipation

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 sends them 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 sending other NACKs. We can see on Figure 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 original one checked the 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 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 improved 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 large packet and window sizes, the basic method requires a lot of memory. With 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 decreases 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 (*i.e.*, 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 to avoid sending again a packet that has already been ACKed. It is necessary because there are now ACKs for individual packets together with cumulative ACKs.
- a timeout value. The packet is sent again if the timeout expires.

The variables `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 they are 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. It uses only cumulative ACKs. Typically, everything between the lost packet and the last packet received is sent again.

Incremental. We want an algorithm that does not 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 original 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 is not accurate since we do not 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`).

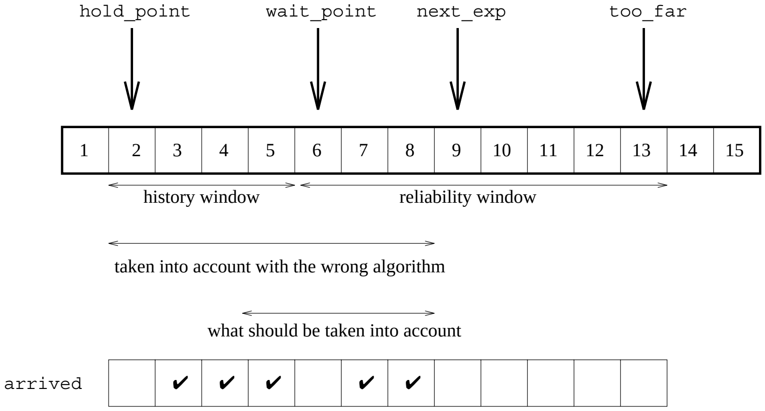


Figure 8: What should be taken into account for loss detection

2.3.4 New receiver algorithm

Instead of adapting the original protocol to use both incremental and anticipation approach, we write a new one from scratch. This algorithm uses a fixed history window (no more `hold_point` which makes a random-sized history window). `too_far` is deleted too: it may lead to an unpredictable behavior 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 received packet. See the section 2.4.5 for more information about the creation of a frame. Each cell contains 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, *i.e.*, 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 is not significant.

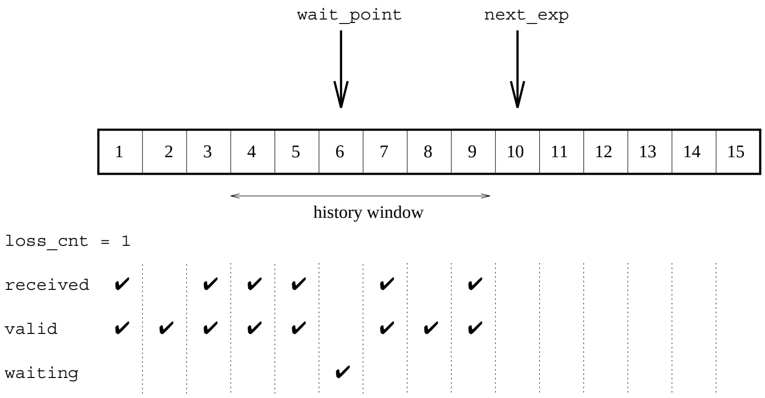


Figure 9: A valid state with the new algorithm

There are still:

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

next_exp is the next expected packet.

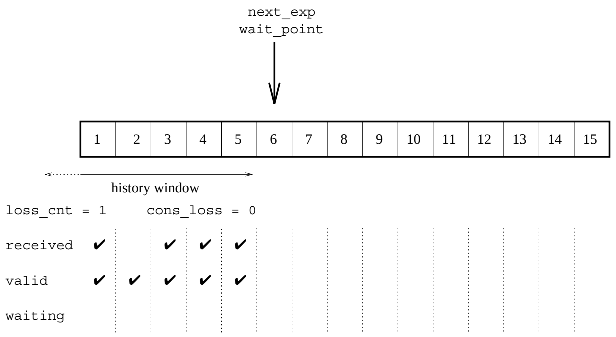
We add a loss_cnt variable that indicates the current number of packets lost among the window_size packets preceding next_exp. Observe that, window_size is the size of the *history window*. There is no more reliability window.

It is an anticipation algorithm because, as shown on Figure 9, loss_cnt counts only *actual* losses (*e.g.*, packets 2 or 8) and not packets tagged as waiting, because they are *virtually* not lost since they *have* to arrive!

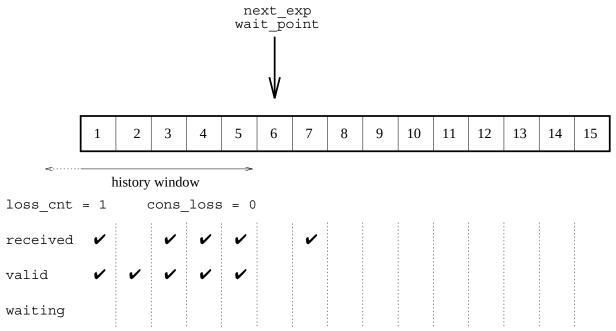
It is an incremental algorithm because we do not have to browse the whole history window to update loss_cnt. Each time we shift the window, we update the counter according to what enters and what leaves the window. Another loss counter, called cons_loss, stores how many consecutive losses are currently at the window beginning. It is non-zero only while we are updating the window (see examples below).

2.3.5 Examples

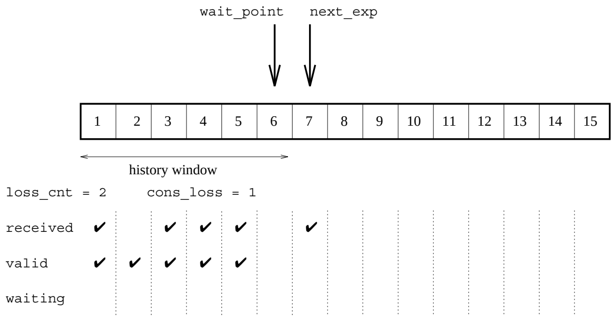
Example with maximum loss rate. The window size is 6, the maximum loss rate is one packet per window (16%).



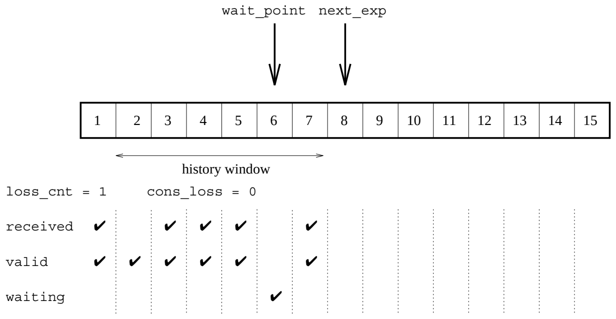
The loss of packet 2 was allowed. `loss_cnt` is 1. Packet 7 arrives, packet 6 is lost:



We update the history window:

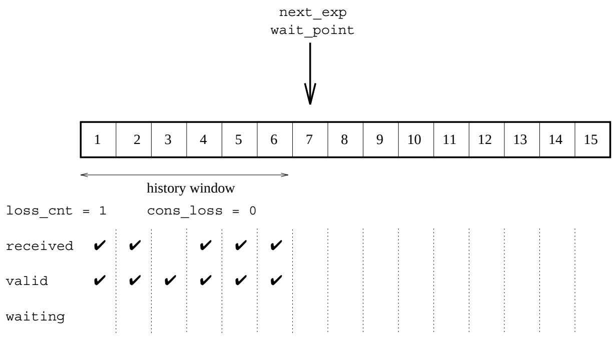


The counter `loss_cnt` is now 2, which is not allowed. We send a NACK and continue the history window update. `loss_cnt` is then only 1 because packet 6 is not lost: it has to arrive.

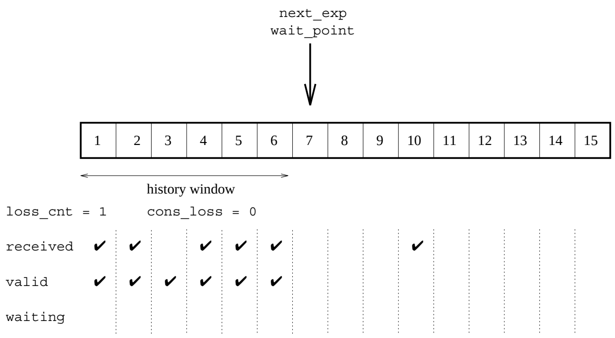


We are in a valid state, let's wait for another packet!

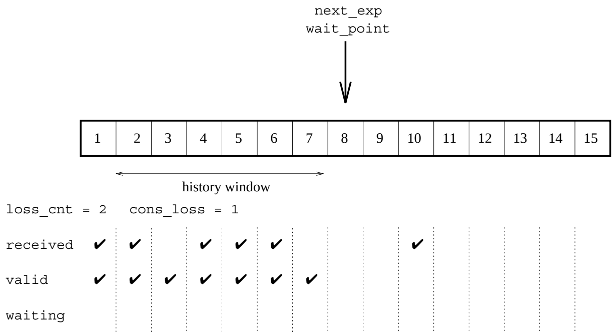
Example with maximum consecutive losses number. The window size is 6, the maximum loss rate is two packets per window (32%), the maximum consecutive losses number is set to 1.



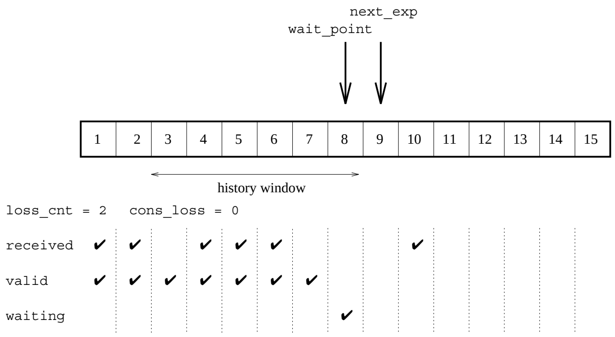
For some reason, 3 packets are lost in a row. We receive packet 10.



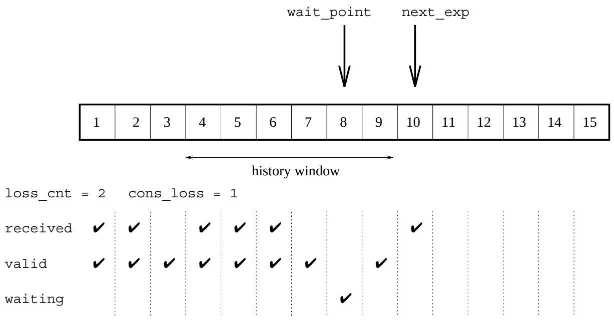
We start updating the history window:



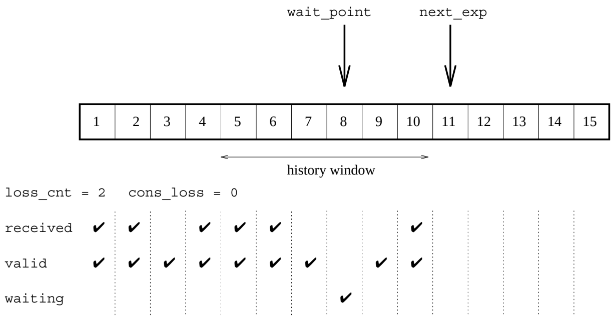
Everything is ok, the loss is allowed.



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 virtually not any longer lost) and we do not increment loss_cnt.



Packet 3 leaves the window and lost packet 9 enters the window, so loss_cnt does not 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 are simultaneously sent. It is even worse for the receiver: different clients from different hosts may send data at the same time. There is still another case: if a thread sends a large frame, it should not prevent another thread from sending another frame to the same receiver. With a usual FIFO connection, the large frame would lock the network socket causing the other thread to wait.

2.4.2 Connected vs. unconnected

The original implementation proposed a *connected* protocol, that has several 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 uncompatible (or at least, hard to make so) with multiple simultaneous 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 transmitted, 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 and incoming frames information blocks

Both sender and receiver data structures only contain global parameters about the connection (*e.g.*, destination address, port, information about the timeout, etc.). 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 does not cause a "traffic jam" on the socket, smaller frames overtake it. Notice however that a sender always sends frames to the same receiver.

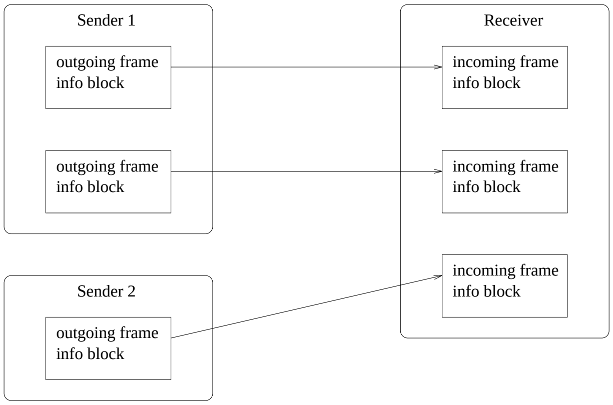

Sender 1
outgoing frame info block
outgoing frame info block
Sender 2
outgoing frame info block
Receiver
incoming frame info block
incoming frame info block
incoming frame info block

Figure 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 origins. 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 manages one connection. As soon as the frame is transmitted, the connection is closed (but both sender and receiver are still there!)

This method is quite flexible: 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 exists 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. On receiving a packet, we need to identify the frame it belongs to. The frame ID is:

- the sender's *IP* address and *UDP* port, that identify the sender,
- a frame number, that identifies 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 immediately up (but reinitialized) it is really unlikely that the new starting point is a current frame number (*i.e.*, frame not closed yet). If ever it happens, they do not understand each other and the connection is reset (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 original 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 is difficult in Nexus. We cannot detect the end of a frame thanks to the beginning of the next one. "The next frame" does not even make sense since the connection lives only for one frame. 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 (*i.e.*, 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. The receiver is then allowed to safely destroy 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 do not 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 is no deadlock, in any case, but it is up to the application to decide whether it tries again or if it gives up.

2.4.4 Critical data, loss tolerance parameters

The first loss tolerance parameters provide us with a powerful way of specifying the service we want, but it is not still enough. In many cases, there are parts where losses are allowable, but small other parts where nothing can be lost (*e.g.* 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 the other loss tolerance parameters are.

As the application does not 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 consecutive 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 transmitted as being itself "critical" and is never allowed to be lost. In case the packet with the frame header is duplicated, the frame creation is not duplicated. The frame header contains the frame number that is a unique identifier for each frame. We check a frame with the same number does not exist before creating a new one.

Thanks to these mechanisms, data inside a frame keeps its position relativelly to the frame· As a consequence, data inside a frame is never reordered nor duplicated. Frames are never duplicated because we check the frame number before creating a new one.

2.5 Tuning *VRP*

2.5.1 Adaptive timeout

The original 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 milliseconds, this has to be changed.

Given that developing such an adaptive timeout algorithm is difficult and would undoubtedly lead to an unstable dynamic system, we rather chose to use an existing one: the JACOBSON algorithm, originally 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 iterative 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 relationship 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 usually take $\alpha = 7/8$.

KARN algorithm. If a timeout expires, when the packet is eventually ACKed, do not update RTT and D since M is impossible to compute unless we specify which retransmission is being ACKed. KARN recommends to double the timeout each time a packet is lost. If we do not want to break the iterative formula, we have to multiply RTT and D rather than $timeout$.

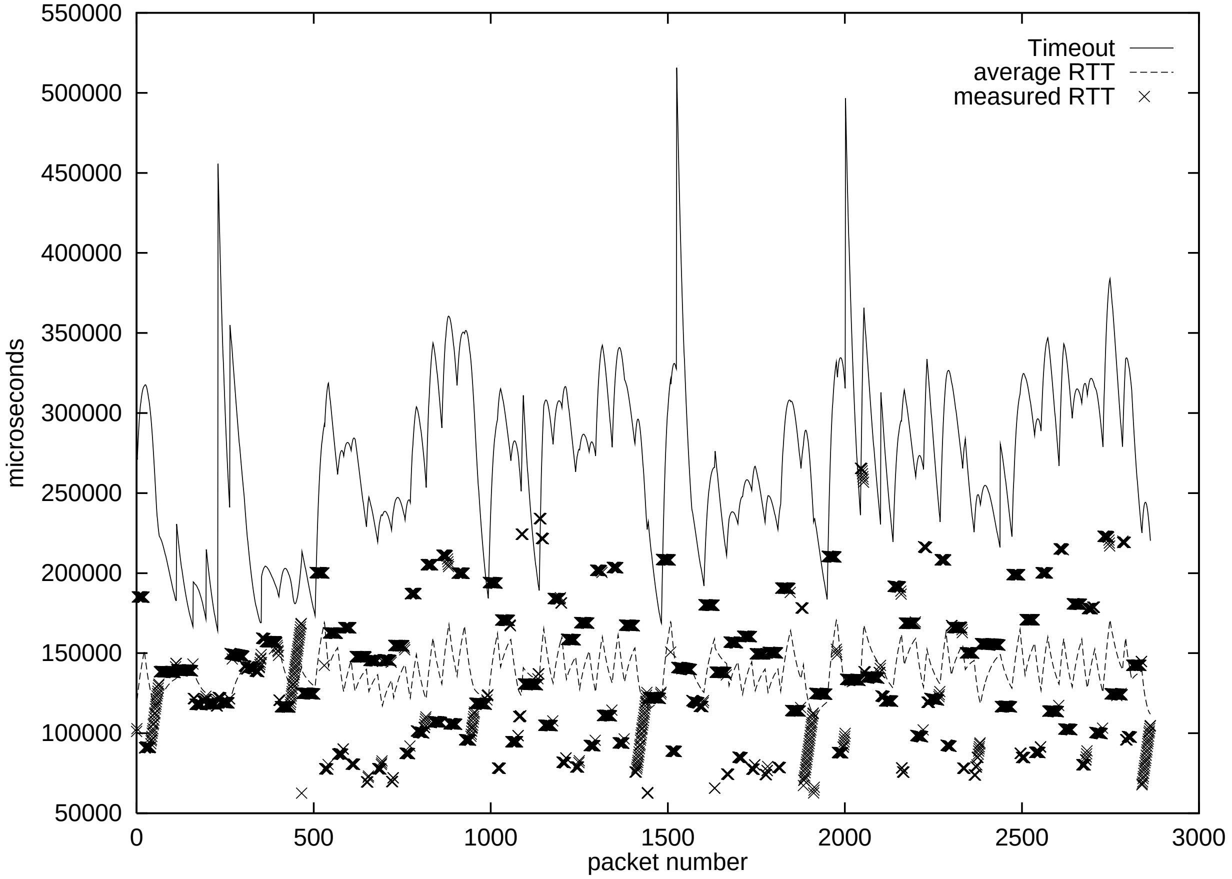


Figure 11: Example of timeout evolution

Initialization. When the sender is created, it sends a time-stamped "ping" to the receiver. When it gets the response, we are able to compute a first approximation of RTT. The ping itself has a timeout with a default value. As a sensible initial guess for $timeout$ is $2 \times RTT$, we chose $D_{init} = RTT_{init}/2$.

Figure 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 all packet duplications are due to this) and the delay does not diverge —it is almost flat on LANs with a steady RTT, and follows the evolution on less predictable connections such as WANs.

2.5.2 Packet size

Choosing the right packet size is essential to get high performances, but it sets 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?

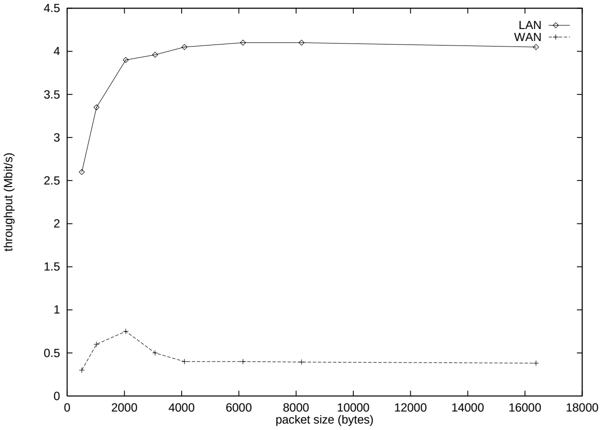


Figure 12: Throughput function of the packet size

Let's make some benchmarks: the window size is fixed, the timeout uses JACOBSON and KARN algorithms, we allow 0% loss. According to Figure 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, it sometimes is more efficient 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 as a reasonable tradeoff: not too slow on a LAN (rather than 1 kB) and a maximum performance on a WAN. This value though can be overridden by the application for specific uses.

2.5.3 Window size

Finding the right window size is not so difficult than other parameters. A large window is useful 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 packets per window, but rather a 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 overridden 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.

3 Implementation

3.1 Packet format

Every packet transmitted have the same 6-byte 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 then, the header is followed by the data and that's all. If it is a control 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 response to a ping. It is followed by the 8 bytes that followed the received `PING`.

`ACK` followed by a 4-byte packet number is a cumulative ACK for this packet and all the previous packets in this frame.

`SINGLE_ACK` followed by a 4-byte packet number is an individual ACK.

`NACK` followed by a 4-byte packet number is a negative acknowledgment for this packet number in the current frame.

`FRAME_CREATE` is the beginning of a new frame. It commands the receiver to create a new incoming frame information block if this frame number does not exist yet. This tag is followed by a 2-byte 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 (computed 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 interpolation rule). This tag is optional, too.

`FRAME_OK` followed by a 2-byte frame number indicates that the receiver has successfully created an incoming frame information block.

FRAME_REFUSED followed by a 2-byte 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-byte frame number asks the receiver to destroy this incoming frame information block.

MISSING_HDR followed by a 2-byte frame number indicates that the receiver has received data packets with an unknown frame number.

CRITICAL_PACKET followed by a 4-byte packet number warns the receiver that this packet is critical and is not allowed to be lost.

CRITICAL_INTERVAL followed by two 4-byte packet numbers warns the receiver that packets between these two numbers are critical.

PADDING followed by a 2-byte 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 sent in the wrong direction (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 do not make sense if they do not follow FRAME_CREATE. The tag itself is represented as a single byte, all multi-byte integers must be given big-endian for portability.

3.2 Asynchronous I/O

For performance issue and because there is no choice in Nexus, all input and output have to be asynchronous. It means that it is impossible to do a blocking read operation on the socket. The only way to read data is to provide Nexus with a handler that is triggered each time a select system call has checked that data are available.

3.2.1 Sender

The sender global structure. Figure 13 shows the global behavior of the sender. Such a structure is created for each socket.

Outgoing frame information block. Figure 14 shows the behavior 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.

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 compute 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.

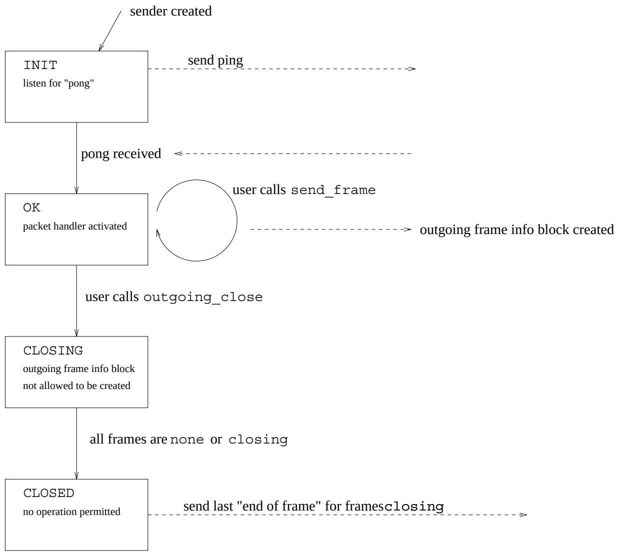

sender created
INIT
listen for "pong"
send ping
pong received
OK
packet handler activated
user calls send_frame
outgoing frame info block created
user calls outgoing_close
CLOSING
outgoing frame info block
not allowed to be created
all frames are none or closing
CLOSED
no operation permitted
send last "end of frame" for framesclosing

Figure 13: The sender's states

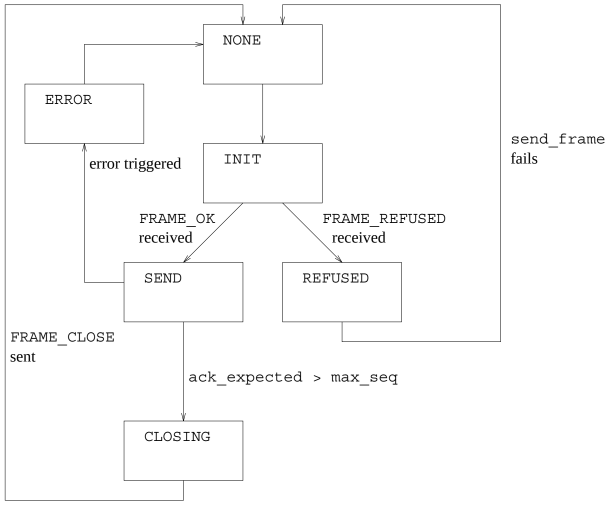


Figure 14: The states of the outgoing frame information block

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 retransmitted, 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 "retransmitted", else the frame state is set to ERROR.

All other tags are discarded.

3.2.2 Receiver

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.

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 forbidden.

closed: the block can be freed as soon as deliver is false.

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, computes 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 —notice frame headers are single packet.

FRAME_CLOSE: set the closed flag for this frame. It is not destroyed immediately 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, then discard it.
- if the packet was waiting, mark it as received, valid, and not waiting. Record the data into the storage, and do not update any loss variable, this packet was supposed to arrive.
- if the packet was not waiting:
  - if it was received, then 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 —do not 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 knows the receiver's state.

This implementation performs exactly what has been described in Section 2.3.4. See the examples on page 10.

3.3 *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)
```

This function 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 is sent with no such information, and the receiver uses its default values.

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

This function 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)
```

This function specifies that some data inside the buffer are critical and are not allowed to 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)
```

This 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)` physically opens the connection. It sends a ping and computes the initial timeout parameters.

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

This function 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)
```

This function creates a receiver and links it to a Nexus endpoint.

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

This function closes a receiver.

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

This is the function that is 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 will be triggered when a frame will be completed.

3.4 *VRP* in Nexus

3.4.1 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 into a frame —one RSR per frame, one frame per RSR. The RSR header is marked as "critical" and is not allowed to be lost. The remaining of the data has fixed loss tolerance parameters that cannot be changed from one frame to another.

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

3.4.2 Limitations

The receiving application gets the data through nexus_get_* primitives that do not provide other information than the data itself. If we want Nexus to give the application more information (*e.g.*, the actual losses), we have to rewrite a large part of its code. Nexus aims at being a general network protocol interface. Its generic API does not allow any application to use such protocol-specific information.

3.4.3 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.

4 Performance issues

4.1 Short distance

Over a short distance connection (*i.e.*, 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 sits on top of *IP* on a LAN. *VRP* is slower because it runs in user space and thus has to issue 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 suited for a LAN.

4.2 Long distance

4.2.1 Overall performance

The speed is simply amazing! On a long distance connection (*i.e.*, 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 twice 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 does not 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.

All benchmarks aim at measuring throughput. We send a huge amount of data (several megabytes) using *TCP* and *VRP*. We hope that traffic condition does not change too much while the test is running. Domain isi.edu is in L.A., California, anl.gov is in Chicago, Illinois, and

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.ens-lyon.fr`	0.21	0.56	2.66

Table 1: *VRP* vs. *TCP* on a WAN

`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 100 kB frames. 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 Tradeoff between reliability and throughput

We decrease reliability so that speed increases. True? Actually we don't. In the previous tests, *VRP* is much 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 where latency is high, anyway. For frames small enough so that throughput is not an issue, we approximately measure the same latency for *TCP* and *VRP*.

4.3 Congestion issues

It seems this protocol achieves so high a throughput by flooding the network, without any flow control nor congestion control. Since *VRP* is based on ACKs and NACKs, there is a flow control. Actually there is no dedicated congestion mechanism like the the congestion window of *TCP*, but our experiment shows that this is not that much a problem: when there are a lot of losses —in case of congestion—, the round-trip time increases, so that the retransmission timeout increases. As the consequence, the average throughput decreases.

VRP is more optimistic than *TCP* about the usable bandwidth, but it is unlikely to lead to congestion because some losses are allowed. Moreover, *VRP* may be extended in a future work to be a true adaptive protocol. If applications provide an interval of allowable loss, then the protocol adapt itself, and choose the right loss tolerance parameters. If the network load is low (low RTT), it chooses the lowest loss rate; if the network load is high, it chooses the highest allowable loss rate to avoid congestion.

5 Conclusion

What has been done. The new version of the *VRP* protocol has been successfully implemented and improved. It is now able to support multi-threaded applications and is rather well error tolerant (broken connection, program or machine crash, etc.).

When used in the context it was designed for, it is amazingly fast compared to the general purpose *TCP* protocol. It is not specifically designed for small messages though it behaves quite well. It is not perfectly suited for communication over a LAN where it is slightly slower than *TCP*. But large amounts of data over a WAN do not scare it and the throughput is 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 because the Nexus API is not flexible enough to allow applications to specify parameters on an RSR basis. *VRP* can show what it is really worth only when used directly by an application.

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 immediately, instead of keeping everything in memory.
- Dynamically allocate frame information blocks instead of having a fixed array. There should still remain a maximum number to avoid memory shortage in case frames cannot be delivered and are accumulating.

A few things might 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 goal of *VRP*.
- Design an algorithm to adapt dynamically the sender's window size and/or packet size, without the pessimistic approach that drastically slows down *TCP*.

We should 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 specify loss tolerance parameters on a frame basis and not only at the connection opening.

This is definitely the main point: *VRP* is able to provide the application with a lot of feedback information, and the application can give *VRP* specific loss tolerance parameters, specific window or packet size, or critical packets. What is very disappointing is that Nexus cannot use these features without a lot of changes. Nexus is not designed to provide applications with much feedback nor to allow applications to specify parameters for RSRs. 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/`.

References

[1] Hari Balakrishnan and Randy H. Katz. Explicit loss notification and wireless web performance. In *IEEE Globecom Internet Mini-Conference*. University of California at Berkeley, 1998.

[2] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.

[3] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[4] Robin Kravets, Ken Calvert, P. Krishnan, and Karsten Schwan. Adaptive variation of reliability. In *Proceedings of the Seventh IFIP Conference on High Performance Networking (HPN'97)*, Atlanta, Georgia, USA, april 1997. Georgia Institute of Technology.

[5] Jon Postel. Transmission control protocol, Darpa Internet program, protocol specification. RFC 793, september 1981.

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

[7] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, march 1996.

[8] Lixia Zhang. Why TCP timers don't work well. In *ACM SIGCOMM*, pages 397–405, August 1986.