# VRP : un protocole avec une tolérance de perte ajustable pour des hautes performances sur réseau longue distance

Alexandre Denis

**RenPar'2000**
**Besançon, 19 juin - 22 juin 2000**

# VRP : un protocole avec une tolérance de perte ajustable pour des hautes performances sur réseau longue distance

Alexandre DENIS

LIP, ENS Lyon,
46 allée d'Italie
69364 Lyon Cedex 07, France.
Internet : `Alexandre.Denis@ens-lyon.fr`.

**Résumé**

Les applications multimédia ont en général le choix pour leurs communications entre *TCP*, fiable mais souvent trop lent, et *UDP*, rapide mais sans contrôle sur la fiabilité. Robin KRAVETS a proposé une alternative basée sur une tolérance de perte ajustable. Nous proposons dans ce papier des améliorations du principe, la spécification d'un protocole appelé *VRP*, et une implémentation au sein de l'environnement de meta-computing Globus. Notre but est une augmentation du débit sur les longues distances qui ont typiquement un taux de perte élevé.

**Mots-clés :** protocole réseau, tolérance de perte, meta-computing, réseau longue distance, TCP.

## Motivations

L'émergence du meta-computing relance l'intérêt pour les communications longue distance sur Internet. Dans l'environnement de meta-computing Globus [4], les communications sont assurées par le module Nexus [3]. Pour les communications sur Internet, Nexus a essentiellement le choix entre *TCP* et *UDP*, tous les deux basés sur *IP*. Le protocole *TCP* est fiable mais fait payer sa fiabilité au prix d'une certaine lenteur. Le protocole *UDP* propose un service vraiment minimal, sans aucun contrôle sur la fiabilité. Les applications choisissent en général *TCP* à cause des trop faibles services offerts par *UDP*. Cependant il existe des applications qui préfèrent un compromis entre la fiabilité et la vitesse. Le protocole *RTP* [8] réalise un compromis de ce genre, mais est axé vers les transferts temps réel. Il se soucie des contraintes de temps plutôt que de la fiabilité.

L'objectif que nous visons est une amélioration des performances sur des connexions Internet grande distance qui ont typiquement un taux de perte élevé quand le réseau est chargé. La solution doit être adaptée au principe "best effort" qui règne encore en maître sur Internet. La notion de service garanti est encore peu présente dans le monde Internet et demande de plus de lourdes modifications des infrastructures existantes, routeurs en particulier. Nous nous plaçons dans le cadre de systèmes de meta-computing comme Globus qui mettent l'accent sur la portabilité et l'utilisation du matériel existant sans modification.

Dans ce papier, nous proposons un protocole appelé *VRP*, Variable Reliability Protocol, implémenté audessus d'*UDP*, donc portable sur toutes les machines d'Internet. Il est implémenté en espace utilisateur, il n'est donc pas nécessaire de modifier le système d'exploitation. Ce protocole est issu d'idées émises par Robin KRAVETS *et al.* dans [6]. Notre contribution est une amélioration des principes, une spécification complète du protocole et une implémentation au sein de l'environnement de meta-computing Globus. Nous aborderons le principe général de *VRP* dans une première partie. Nous examinerons ensuite les principaux algorithmes mis en œuvre. Nous exposerons et commenterons enfin les performances obtenues.

## 1. Principe général de *VRP*

### Tolérance de perte réglable

Une liaison fiable garantit l'arrivée des données, sans perte, sans duplication, et dans l'ordre. Le protocole *TCP* fournit ce service. Il parvient à ce résultat en mettant en place un mécanisme de fenêtre

glissante qui retransmet les paquets *IP* au cas où ils seraient perdus. Sur Internet, les pertes de paquets *IP* sont de deux types. Quand les routeurs n'ont pas le temps de traiter un paquet, ils le suppriment. Cela se traduit par des pertes isolées. Quand la machine qui reçoit est saturée, ou s'il y a une congestion dans un routeur, alors les pertes sont en rafales (*burst*). Dans ce cas, de nombreux paquets successifs sont perdus. Dans le cas d'un réseau très chargé qui perd beaucoup de paquets *IP*, le mécanisme de retransmission systématique de *TCP* est très pénalisant. Une application qui transmet par exemple des images préfère souvent améliorer le débit (le nombre d'images par secondes), même si ce but est atteint au détriment de la qualité. Elle est prête à accepter des pertes, qui se traduiront selon le cas par des lignes ou des images manquantes. L'application doit cependant garder un moyen de contrôle sur ces pertes pour qu'elles restent tolérables. Le protocole *VRP* met à la disposition de l'application les paramètres suivants :

- le taux de perte maximal tolérable, exprimé sous forme d'un pourcentage.
- la longueur maximale de perte consécutive, qui indique les plus grandes rafales autorisées. On contrôle ainsi la granularité des pertes. Si on reprend l'exemple d'une application qui transmet des images, ce paramètre permet de maîtriser le nombre maximal de lignes manquantes consécutives d'une image afin d'effectuer une interpolation pour retrouver des données manquantes.
- les données critiques, qui sont des parties de messages qui ne peuvent être perdues sous aucun prétexte. C'est le cas des en-têtes ou d'autres données de contrôle, que l'on trouve au milieu de données qui peuvent être perdues. Peu importe ce que sont les autres paramètres, les données marquées comme critiques n'ont pas le droit d'être perdues. Ce paramètre n'était pas présent dans [6] et a été ajouté pour que le protocole soit utilisable.

*VRP* garantit que les paramètres fournis par l'application sont respectés. On remarquera que ce sont des maxima, les pertes peuvent être en pratique bien moindres que ce qui est autorisé. Il est intéressant de noter que si l'on relâche tous les paramètres (toutes pertes autorisées), on retrouve les spécifications du protocole *UDP*. Si au contraire tout est marqué comme critique ou si l'on précise un taux de perte maximal de 0%, on retrouve *TCP*.

**Encapsulation**

A la différence de *TCP*, notre protocole n'est pas basé sur un flux d'octets mais sur l'envoi de messages — des unités logiques de niveau application — qui sont découpés en paquets par le protocole. Tout le contrôle de la fiabilité est fait au niveau de ces paquets. Cependant l'application n'ayant pas à savoir ce qui se passe au niveau du protocole — en particulier, elle n'a pas à connaître la taille des paquets —, elle fournit tous ses paramètres indépendamment de l'implémentation du protocole. Ainsi :

- le taux de perte maximal tolérable est donné en pourcentage, qui est traduit en nombre de paquets par fenêtre ;
- la taille maximale de perte consécutive est donnée en octets, *VRP* traduit en nombre de paquets consécutifs ;
- les données critiques sont données en octets, *VRP* se charge de traduire en paquets critiques.

Ces paramètres sont vérifiés sur des paquets complets. *VRP* est donc plus restrictif que les paramètres donnés par l'utilisateur. Si par exemple une zone de données est marquée comme critique au niveau application, alors tous les paquets qui en contiennent une partie sont critiques. On est ainsi assuré du respect des paramètres. De même, l'algorithme utilisé assure que contrôler le taux maximal de perte sur la largeur d'une fenêtre glissante implique que ce taux est respecté globalement sur le message complet.

**Feedback**

Lorsqu'il y a des pertes, l'application réceptrice est prévenue des zones de données perdues. Elle peut donc choisir de s'adapter en conséquence et tenter une récupération partielle, par exemple par interpolation s'il s'agit d'images.

Elle peut également choisir de ne pas prêter attention aux pertes, et faire comme si tout avait été reçu. En effet, elle reçoit des messages qui ont exactement la même taille que ce qui a été envoyé, quelles que soient les pertes. Les zones perdues sont simplement comblées avec des zéros. Le message est reçu virtuellement sans perte. On se ramène dans ce cas à une tolérance d'erreurs à la place d'une tolérance de perte. Les données qui arrivent sont à leur place dans le message, l'ordre est conservé, il n'y a pas

de duplication. L'ordre de réception des messages n'est pas conservé; les messages peuvent se doubler. On met en place des mécanismes qui contrôlent qu'aucun message n'est dupliqué. Aucun message n'est normalement perdu. Dans le cas où on n'arrive pas à joindre le récepteur, l'application émettrice est prévenue que le message n'est pas arrivé à destination. On fournit également les chiffres des pertes réelles à l'application émettrice pour qu'elle puisse éventuellement s'adapter aux conditions de trafic.

## 2. Algorithmes mis en œuvre

### 2.1. Fenêtre glissante et tolérance de perte

Le protocole *VRP* se base sur un mécanisme de fenêtre glissante à la manière de *TCP* [1]. L'émetteur envoie les paquets dans l'ordre et attend les accusés de réception (notés ACK dans la suite). Ces acquittements sont cumulatifs, c'est-à-dire qu'ils informent de la réception d'un paquet et de tous les paquets qui le précèdent. Pour chaque paquet transmis, l'émetteur déclenche un temporisateur. S'il arrive à expiration avant la réception de l'acquittement, on suppose que le paquet a été perdu. Dans ce cas, on ré-émet le paquet correspondant. Du point de vue de l'émetteur, la transmission est donc perçue comme fiable.

La tolérance de perte est gérée du côté du récepteur. Lorsqu'il détecte un paquet manquant, le récepteur examine l'historique des dernières pertes. Il contrôle si la perte est permise par les paramètres fournis par l'application. Si la perte est autorisée, il envoie un acquittement exactement comme si le paquet avait été reçu.

### 2.2. NACK, SACK et anticipation

Le mécanisme présenté ci-dessus est correct, mais peut être optimisé. Pour améliorer la vitesse de réaction aux pertes, on ajoute un système d'acquittements négatifs (notés NACK) émis dès que le récepteur détecte une perte. À leur réception, l'émetteur ré-émet immédiatement le paquet correspondant.

Au vu du nombre de paquets retransmis inutilement, il a également semblé nécessaire d'ajouter des acquittements individuels pour les paquets, en plus des acquittements cumulatifs, semblables aux SACK introduits dans *TCP* [7]. Ces acquittements sélectifs permettent d'acquitter tous les paquets reçus qui suivent un paquet perdu. L'utilisation conjointe des SACK et des NACK est donc tout-à-fait naturelle. Elle permet une récupération de perte plus rapide et évite les transmissions inutiles.

Notre but est d'avoir le moins de ruptures possible dans le flux. Si un paquet est perdu alors qu'un NACK est en cours (ie. un NACK a été envoyé, le paquet correspondant n'a pas encore été reçu), il est possible de ne pas rester bloqué. Il suffit d'anticiper ce qui va se passer. Comme le montre la figure 1 (en haut), une approche raisonnable est d'attendre que tout rentre dans l'ordre avant d'analyser la perte suivante. Cependant, on peut imaginer un scénario comme celui de la figure 1 (en bas) où l'on a simplement anticipé l'arrivée du paquet numéro 4. Puisque la perte du paquet numéro 4 n'a pas été autorisée et qu'un NACK a été envoyé, alors ce paquet arrivera nécessairement. Ce paquet étant virtuellement arrivé, le récepteur peut continuer à traiter la suite en connaissance de cause, et autoriser la perte du paquet

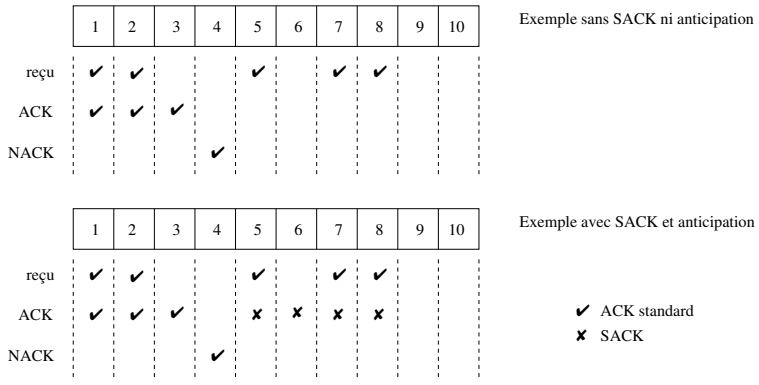


FIG. 1: Exemple d'état valide du récepteur sans SACK ni mécanisme d'anticipation (en haut), avec SACK et anticipation (en bas). La longueur de perte maximale consécutive est fixée à un paquet.

numéro 6. Il envoie donc des acquittements sélectifs SACK (notés ✘ sur la figure) pour les paquets 5 à 8 et évite leur retransmission qui serait inutile.

### 2.3. Transmission d'un message

Dans un environnement de meta-computing, la tolérance aux pannes est un aspect essentiel. C'est pourquoi nous proposons que *VRP* soit un protocole non-connecté, c'est-à-dire sans état. À la différence de *TCP*, si par exemple le récepteur coupe la liaison puis la rétablit, l'émetteur n'a pas besoin de réinitialiser la liaison. Bien sûr, pendant la transmission d'un message, une connexion existe. Il peut même y avoir plusieurs connexions simultanées si plusieurs messages sont transmis simultanément.

Le protocole proposé dans [6] est connecté, simplement pour disposer d'un ordre séquentiel sur les messages. Cette méthode permet de gérer correctement la fin des messages.

Nous adoptons une solution différente, plus robuste. Quand il reçoit un acquittement pour le dernier paquet d'un message, l'émetteur est sûr que le message complet a été reçu. Il détruit alors toutes les structures destinées à l'envoi du message. Le récepteur ne peut pas agir de manière symétrique. Si l'acquittement du dernier paquet se perd, l'émetteur ne serait jamais prévenu que le message a été reçu. Le récepteur attend donc que l'émetteur lui signale qu'il a bien reçu le dernier acquittement pour lui aussi détruire les structures correspondantes.

Cette notification de succès de transmission, les ordres d'initialisation de message, les spécifications des paramètres de tolérance de perte et les acquittements sont transmis dans des paquets de contrôle. Ces paquets de contrôle ont une structure modulaire et flexible à base de balises. Leur structure exacte est décrite dans [2]. Le récepteur acquitte les paquets de contrôle qu'il reçoit de l'émetteur, on est donc sûr qu'ils arrivent. Par contre dans l'autre sens, ces paquets ne génèrent pas d'acquittement (en particulier, on n'acquitte pas les ACK !) Pour gérer le cas du dernier message, et pour améliorer la tolérance aux pannes, le récepteur mesure les périodes d'inactivité, qui se traduisent par l'absence de réception d'ACK de numéro de séquence supérieur au dernier ACK reçu. En cas d'inactivité trop longue, il considère que la connexion a été rompue pour une raison quelconque — problème matériel, crash, congestion sévère du réseau, etc. — et il notifie l'application que le message n'a pas été transmis avec succès. Un mécanisme symétrique est mis en place du côté du récepteur.

### 2.4. Réglages

#### Temporisation

Le choix de la temporisation est déterminant pour diminuer le temps de réaction aux pertes. Étant donné que le temps qui s'écoule entre l'envoi d'un paquet et la réception de l'acquittement correspondant est très variable, il est nécessaire d'utiliser un algorithme adaptatif. Plutôt que de se lancer dans le développement d'un nouvel algorithme, on se base sur la méthode classique de Jacobson [5] utilisée dans *TCP*.

On approche le temps d'aller-retour moyen à l'aide d'un filtre passe-bas du type $RTT = \alpha \cdot RTT + (1 - \alpha) \cdot M$ où $RTT$ est l'approximation du temps d'aller-retour, $M$ est le temps d'aller-retour mesuré, et $\alpha$ est un paramètre de lissage. On approche l'écart moyen par $D = \alpha \cdot D + (1 - \alpha) \cdot |RTT - M|$. On choisit alors $timeout = RTT + 4 \times D$. Le facteur 4 peut sembler arbitraire : il a été déterminé expérimentalement pour que seulement 1% des expirations de temporisateur correspondent à des paquets qui ne sont pas perdus. Comme le montre la figure 2, le *timeout* choisi suit le temps d'aller-retour de très près. Il y a peu d'expirations pour rien, mais on détecte très tôt les paquets perdus.

#### Taille de fenêtre

La taille de la fenêtre représente le nombre maximum de paquets non-acquittés à chaque instant, et influe sur les performances du protocole. Il est essentiel d'avoir une fenêtre suffisamment large sur des réseaux de types WAN où la latence est typiquement élevée. L'émetteur risque en effet de passer son temps à attendre les acquittements. Si elle est trop grande, l'inconvénient majeur est le débordement des tampons du récepteur qui entraîne de nombreuses pertes en rafales. De plus, pour des raisons d'implémentation, ce nombre doit être une puissance exacte de 2.

Des mesures sur différentes liaisons, avec des machines très différentes (PC, Sun UltraSparc, SGI Origin 2000), et avec des conditions de trafic différentes ont montré que ce paramètre n'avait pas une influence très grande. La vitesse augmente progressivement avec un élargissement de la fenêtre ; à partir de 128

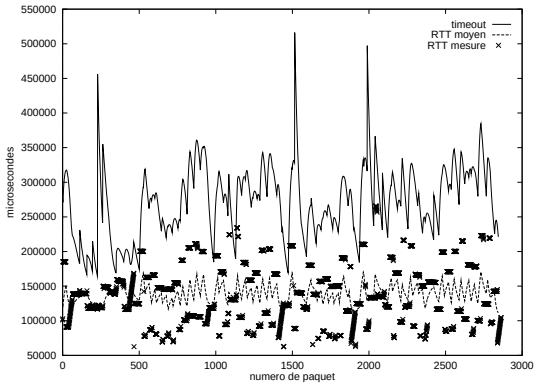


FIG. 2: Exemple d'évolution du timeout

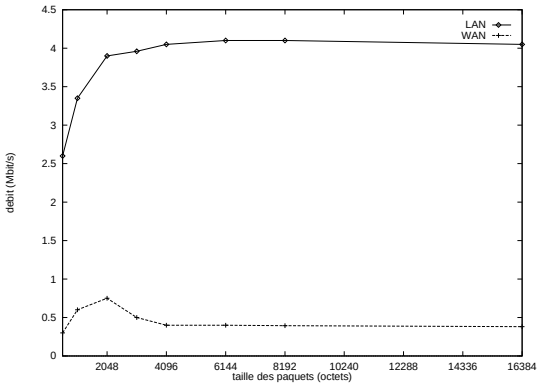


FIG. 3: Le débit en fonction de la taille des paquets

paquets par fenêtre, on note une augmentation du taux de perte. Nous choisissons une taille par défaut de 64, et le protocole laisse tout de même la possibilité à l'utilisateur d'imposer une autre valeur s'il le désire. Il n'a pas semblé nécessaire de mettre en place une taille variable avec un algorithme adaptatif.

### Taille des paquets

Trouver la bonne taille de paquet est difficile. Il faut se poser la bonne question : qu'est-ce que la *bonne* taille de paquet ? Est-ce celle qui obtient le débit le plus élevé ou celle qui minimise les pertes ? La figure 3 montre les résultats d'un benchmark réalisé sur LAN et sur WAN. Il est clair que sur le WAN, la bonne taille est 2 ko. Sur d'autres exemples, le meilleur résultat est obtenu pour 1 ko. Au-delà de ces valeurs, les pertes augmentent énormément. Pour rester compétitif sur LAN, nous choisissons une taille de 2 ko par défaut. L'utilisateur peut forcer sa propre valeur pour une utilisation spécifique.

Pour une description technique plus complète du protocole *VRP*, ainsi qu'un aperçu de l'implémentation qui en a été faite dans Globus, on se pourra se référer à [2].

## 3. Performances

Nous nous intéressons principalement aux performances sur les longues distances. Les mesures de performances sont difficiles à réaliser car les conditions sont extrêmement variables. Nous donnons des chiffres qui représentent des moyennes. La liaison Internet entre Chicago et Los Angeles nous sert de référence, en utilisant l'implémentation qui a été faite dans Globus. Nous transmettons des messages de 100 ko, en autorisant une perte globale de 25% et des rafales de perte de 4 ko. En pratique, la perte

mesurée est d'environ 10%. Nous atteignons un débit moyen de 1,2 Mbit/s avec *TCP* et environ 6 Mbit/s avec *VRP*, soit un rapport de 5. Même dans le pire des cas observés, *VRP* est encore 2 fois plus rapide que *TCP*. Des tests réalisés sur une liaison transatlantique mettent en évidence un rapport d'accélération de 3 de *VRP* par rapport à *TCP*. Sur un réseau local, comme prévu, *VRP* n'est pas plus rapide que *TCP*. Il est même légèrement plus lent, avec un débit inférieur de 5% à celui de *TCP*.

Nous expliquons la différence de performance par une stratégie complètement différente pour le contrôle de flux et le contrôle de congestion. Le contrôle de flux de *TCP* est précis et complexe. Il cherche à éviter à tout prix tout débordement de tampon. Le contrôle de flux de *VRP* est moins strict. Il exploite le fait qu'un paquet perdu de temps à autre est acceptable. Le contrôle de congestion de *TCP* est pessimiste. Il ralentit à la moindre alerte. Il n'y a pas de vrai contrôle de congestion dans *VRP*. En cas de congestion du réseau, le taux de perte augmente et l'application est prévenue. Libre à elle de s'adapter. Même sans contrôle de congestion, *VRP* n'aggrave pas les congestions : s'il est nécessaire d'effectuer des retransmissions de paquets, plus les retransmissions sont nombreuses, plus le temporisateur augmente. *VRP* ralentit donc de lui-même.

## 4. Conclusion

Nous avons dans ce papier proposé des améliorations pour le protocole *VRP* afin de le rendre utilisable en pratique. Nous avons de plus fourni des spécifications à un niveau d'abstraction assez haut, de manière à ce que les applications l'utilisent sans se soucier de l'implémentation. Nous avons décrit et justifié les mécanismes utilisés pour l'implémenter. *VRP* est maintenant plus tolérant aux pannes. Pour l'implémentation dans Globus, nous l'avons également rendu compatible avec les environnements multi-threads.

Les résultats sont encourageants et à la hauteur de nos espérances. *VRP* a été conçu pour la transmission de gros volumes de données sur de longues distances. Quand on l'utilise dans ces conditions, les performances sont excellentes. Les problèmes de congestion semblent être le seul point délicat. Les premiers tests n'indiquent pas de problème particulier. Nous n'avons cependant pas réalisé de test à très grande échelle, avec de nombreuses connexions simultanées. Il est possible de faire évoluer *VRP* vers un protocole adaptatif qui résoudrait ces difficultés : l'application spécifie un intervalle pour le taux de perte maximal tolérable, le protocole choisit lui-même le meilleur point de fonctionnement dans cet intervalle suivant les conditions de trafic.

Le protocole *VRP* que nous avons présenté est une alternative aux solutions basées sur la qualité de service. Il peut utiliser les infrastructures existantes, c'est là son principal atout. Son implémentation dans Globus autorise maintenant la transmission de données multimédia de manière satisfaisante dans le cadre du metacomputing.

## Bibliographie

1. David D. Clark. Window and acknowledgement strategy in TCP. Internet request for comment 813, july 1982.
2. Alexandre Denis. Variable Reliability Protocol : A protocol with a tunable loss tolerance for high performance over a WAN. Technical report, LIP, ENS Lyon, mars 2000. `ftp ://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2000/RR2000-11.ps.Z`.
3. 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.
4. The Globus project. `http://www.globus.org/`.
5. Van Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols*, pages 314–329, Stanford, CA, august 1988.
6. 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)*. Georgia Institute of Technology, april 1997.
7. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. Request for comment 2018, october 1996.
8. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP : A Transport Protocol for Real-Time Applications. Internet request for comment 1889, january 1996.