Groupe de travail Réseau
Request pour Comments : 5225
Catégorie : Sur la voie de la normalisation
Traduction Claude Brière de L'Isle

G. Pelletier, Ericsson
K. Sandlund, Ericsson
avril 2008

Compression d'en-tête robuste version 2 (ROHCv2) : profils pour RTP, UDP, IP, ESP et UDP léger

Statut du présent mémoire
Le présent document spécifie un protocole Internet sur la voie de la normalisation pour la communauté de l'Internet, et appelle à des discussions et suggestions pour son amélioration. Prière de se référer à l'édition en cours des "Normes officielles des protocoles de l'Internet" (STD 1) pour connaître l'état de la normalisation et le statut de ce protocole. La distribution du présent mémoire n'est soumise à aucune restriction.
(La présente traduction incorpore les errata 1421, 2703, 3185 et 3248)

Résumé
Le présent document spécifie les profils de compression d'en-tête robuste (ROHC, *Robust Header Compression*) qui compressent efficacement les en-têtes du protocole de transport en temps réel (RTP, *Real-Time Transport Protocol)* du protocole de datagramme d'utilisateur (UDP, *User Datagram Protocol*) du protocole Internet (IP, *Internet Protocol*), RTP/UDP-Léger/IP, UDP/IP, UDP-Léger/IP, IP et ESP/IP.

La présente spécification définit une seconde version des profils qui se trouvent dans les RFC 3095, 3843 et 4019 ; il se substitue à leur définition, mais ne les rend pas obsolètes.

Les profils ROHCv2 introduisent un certain nombre de simplifications aux règles et algorithmes qui gouvernent le comportement des points d'extrémité de compression. Il définissent aussi des mécanismes de robustesse qui peuvent être utilisés par une mise en œuvre de compresseur pour augmenter la probabilité de succès de la décompression quand des paquets peuvent être perdus et/ou réordonnés sur le canal ROHC. Finalement, les profils ROHCv2 définissent leur propre ensemble spécifique de formats d'en-tête, en utilisant la notation formelle ROHC.

Table des Matières

1. Introduction

Le groupe de travail ROHC a développé un cadre de compression d'en-tête sur lequel divers profils peuvent être définis pour différents ensembles de protocoles ou exigences de compression. Le cadre ROHC a été d'abord documenté dans la [RFC3095], avec des profils pour la compression des en-têtes RTP/UDP/IP (protocole de transport en temps réel, protocole de datagrammes d'utilisateur, protocole Internet), UDP/IP, IP et ESP/IP (Encapsulation de charge utile de sécurité). Des profils supplémentaires pour la compression des en-têtes IP [RFC3843] et UDP-léger (protocole léger de datagramme d'utilisateur) [RFC4019] ont été spécifiés ensuite pour compléter l'ensemble initial de profils ROHC.

Le présent document définit une version mise à jour pour chacun des profils mentionnés ci-dessus, et leur définition dépend du cadre ROHC qui se trouve dans la [RFC4995]. Il faut lire le cadre pour comprendre les définitions des profils, leurs règles, et leur rôle.

Précisément, le présent document définit les schémas de compression d'en-tête pour :

RTP/UDP/IP :	profil 0x0101
UDP/IP :	profil 0x0102
ESP/IP :	profil 0x0103
IP :	profil 0x0104
RTP/UDP-léger/IP :	profil 0x0107
UDP-léger/IP :	profil 0x0108

Chacun des profils ci-dessus peut compresser les types d'en-tête d'extension suivants :

- AH [RFC4302]
- GRE [RFC2784], [RFC2890]
- MINE [RFC2004]
- En-tête d'options de destination IPv6 [RFC2460]
- En-tête d'options bond par bond IPv6 [RFC2460]
- En-tête d'acheminement IPv6 [RFC2460]

2. Terminologie

Les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", et "FACULTATIF" en majuscules dans ce document sont à interpréter comme décrit dans le BCP 14, [RFC2119].

Le présent document est cohérent avec la terminologie du cadre ROHC [RFC4995] et la notation formelle pour ROHC [RFC4997]. De plus, le présent document utilise ou définit les termes suivants :

Numéro d'accusé de réception : il identifie quel paquet est acquitté dans l'élément de rétroaction RoHCv2 (voir le paragraphe 6.9). La valeur de ce champ correspond normalement au numéro de séquence maître (MSN, *Master*

Sequence Number) de l'en-tête qui a été décompressé en dernier avec succès, pour le contexte de compression (CID) auquel les informations de rétroaction s'appliquent.

Chaînage d'éléments : chaîne de champs de groupes d'éléments fondée sur des caractéristiques similaires. ROHCv2 définit des éléments de chaîne pour des champs statiques, dynamiques et irréguliers. Le chaînage est réalisé en ajoutant un élément à la chaîne pour chaque en-tête dans son ordre d'apparition dans le paquet non compressé. Le chaînage est utile pour construire les en-têtes compressés à partir d'un nombre arbitraire de tout en-tête de protocole pour lequel un profil ROHCv2 définit un format compressé.

Validation de champs de contrôle CRC-3 : la validation de champs de contrôle CRC-3 se réfère à la validation des champs de contrôle. Cette validation est effectuée par le décompresseur quand il reçoit un en-tête compressé (CO) qui contient un contrôle de redondance cyclique (CRC, *Cyclic Redundancy Check*) de trois bits calculé sur les champs de contrôle. Ce CRC de trois bits couvre les champs de contrôle portés dans l'en-tête CO ainsi que des champs de contrôle spécifiques dans le contexte. Dans la définition formelle des formats d'en-tête, ce CRC de trois bits est marqué "control_crc3" et utilise le codage de control_crc3_encoding (voir aussi le paragraphe 6.6.11).

Delta : delta se réfère à la différence de valeur absolue d'un champ entre deux paquets consécutifs traités par le même point d'extrémité de compression.

Profondeur de réarrangement : nombre de paquets dont un paquet est reçu en retard au sein de sa séquence du fait d'un réarrangement entre le compresseur et le décompresseur, c'est-à-dire, le réarrangement entre paquets associés au même contexte (CID). Voir la définition de paquet en retard en séquence ci-dessous.

Types d'en-têtes ROHCv2 : les profils ROHCv2 utilisent deux types d'en-tête différents : le type d'en-tête Initialisation et rafraîchissement (IR), et le type d'en-tête compressé (CO).

Paquet en avance en séquence : paquet qui atteint le décompresseur avant un ou plusieurs paquets qui ont été retardés sur le canal, où tous lesdits paquets appartiennent au même flux d'en-têtes compressés et sont associés au même contexte de compression (CID). Au moment de l'arrivée d'un paquet en avance en séquence, le ou les paquets retardés sur la liaison ne peuvent pas être différenciés des paquets perdus.

Paquet en retard en séquence : paquet en retard au sein de sa séquence si il atteint le décompresseur après qu'un ou plusieurs autres paquets appartenant au même CID ont été reçus, bien que le paquet en retard en séquence ait été envoyé du compresseur avant le ou les autres paquets. Comment le décompresseur détecte un paquet en retard en séquence sort du domaine d'application de la présente spécification, mais il peut par exemple utiliser le MSN à cette fin.

Pas d'horodatage (ts_stride) : c'est l'augmentation attendue de la valeur de l'horodatage entre deux paquets RTP avec des numéros de séquence consécutifs. Par exemple, pour un codage de supports avec un taux d'échantillonnage de 8 kHz produisant une trame toutes les 20 ms, l'horodatage RTP va normalement s'accroître de n * 160 (= 8000 * 0,02) pour un entier n.

Pas de temps (time_stride) : c'est l'intervalle de temps équivalent à un ts_stride, par exemple, 20 ms dans l'exemple de l'incrément d'horodatage RTP ci-dessus.

3. Acronymes

Cette Section fait la liste de la plupart des acronymes utilisés pour référence, en plus de ceux définis dans la [RFC4995].

AH *(Authentication Header)* : en-tête d'authentification.
CO : compressé.
CSRC *(Contributing Source)* : source contributive (l'en-tête RTP en contient une liste facultative).
ESP *(Encapsulating Security Payload)* : charge utile de sécurité encapsulante.
GRE *(Generic Routing Encapsulation)* : encapsulation générique d'acheminement.
FC *(Full Context)* : état de plein contexte (au décompresseur).
IP *(Internet Protocol)* : protocole Internet
IR *(Initialization and Refresh*) : initialisation et rafraîchissement
LSB *(Least Significant Bits)* : bits de moindre poids.
MINE *(Minimal Encapsulation in IP)* : encapsulation minimale dans IP.
MSB *(Most Significant Bits)* : bits de poids fort.
MSN *(Master Sequence Number)* : numéro de séquence maître.

NC *(No Context)* : état sans contexte (au décompresseur).
OA *(Optimistic Approach)* : approche optimiste.
RC *(Repair Context)* : état de contexte de réparation (au décompresseur).
ROHC *(Robust Header Compression)* : compression d'en-tête robuste (cadre de) (RFC 4995).
ROHCv2 : ensemble de profils de compression d'en-têtes défini dans le présent document.
RTP *(Real-time Transport Protocol)* : protocole de transport en temps réel.
SSRC (*Synchronization Source)* : source de synchronisation (champ dans l'en-tête RTP).
TC *(Traffic Class)* : classe de trafic (champ dans l'en-tête IPv6). Voir aussi TOS.
TOS *(Type Of Service)* : type de service (champ dans l'en-tête IPv4). Voir aussi TC.
TS *(TimeStamp)* : horodatage RTP .
TTL *(Time to Live)* : durée de vie (champ dans l'en-tête IPv4).
UDP *(User Datagram Protocol)* : protocole de datagrammes d'utilisateur.
UDP-Lite : protocole léger de datagrammes d'utilisateur.

4. Fondements (information)

Cette Section donne les informations de base sur les profils de compression définis dans le présent document. Les fondements de la compression générale d'en-tête et le cadre de ROHC se trouvent dans les sections 3 et 4 de la [RFC4995]. Les fondamentaux de la notation formelle pour ROHC sont définis dans la [RFC4997]. La [RFC4224] décrit les impacts de la livraison en désordre sur les profils fondés sur la [RFC3095].

4.1 Classification des champs d'en-tête

Le paragraphe 3.1 de la [RFC4995] explique que la compression d'en-tête est possible du fait qu'il y a beaucoup de redondances entre les valeurs des champs au sein des en-têtes d'un paquet, en particulier entre les en-têtes de paquets consécutifs.

L'appendice A fait la liste et classe en détails tous les champs d'en-tête relevant de ce document. L'appendice se conclut par des recommandations sur la façon dont les divers champs devraient être traités par les algorithmes de compression d'en-tête.

La principale conclusion est que la plupart des champs d'en-tête peuvent facilement être compressés car ils ne changent que rarement ou jamais. Un petit nombre de champs a cependant besoin de mécanismes plus sophistiqués.

Ces champs sont:
- Identification IPv4 (16 bits) - IP-ID
- Numéro de séquence ESP (32 bits) - ESP SN
- Somme de contrôle UDP (16 bits) - Somme de contrôle
- Somme de contrôle UDP-léger (16 bits) - Somme de contrôle
- Couverture de somme de contrôle UDP-léger (16 bits) - CCov
- Marqueur RTP (1 bit) - M-bit
- Numéro de séquence RTP (16 bits) - RTP SN
- Horodatage RTP (32 bits) - TS

En particulier, pour RTP, l'analyse de l'appendice A révèle que les valeurs du champ Horodatage RTP (TS) ont généralement une forte corrélation au numéro de séquence (SN) RTP, qui s'incrémente de un pour chaque paquet émis par une source RTP. Le bit M RTP est supposé avoir la même valeur la plupart du temps, mais il doit être communiqué explicitement à cette occasion.

Pour UDP, le champ Somme de contrôle ne peut pas être déduit ou recalculé à l'extrémité receveuse sans violer ses propriétés de bout en bout, et est donc envoyé tel quel quand il est activé (obligatoire avec IPv6). La même chose s'applique à la somme de contrôle UDP-léger (obligatoire avec IPv4 et IPv6) tandis que la couverture de somme de contrôle UDP-léger peut dans certains cas être compressible.

Pour IPv4, une corrélation similaire à celle de l'horodatage RTP avec le numéro de séquence RTP est souvent observée entre le champ Identifiant (IP-ID) et le numéro de séquence maître (MSN) utilisé pour la compression (par exemple, le numéro de séquence RTP quand on compresse des en-têtes RTP).

4.2 Améliorations de ROHCv2 sur les profils de la RFC 3095

Les profils ROHCv2 peuvent réaliser une compression efficace et robuste qui est au moins équivalente à celle des profils de la [RFC3095], quand elle est utilisée dans les mêmes conditions de fonctionnement. En particulier, la taille et la disposition binaire du plus petit en-tête compressé (c'est-à-dire, le format PT-0 U/O-0 dans la RFC 3095, et pt_0_crc3 dans ROHCv2) sont identiques.

Il y a un certain nombre de différences et d'améliorations entre les profils définis dans le présent document et leur version antérieure définie dans la RFC 3095. Cette section donne une vue d'ensemble des améliorations les plus significatives :

Tolérance au réarrangement : les profils définis dans la RFC 3095 exigent que le canal entre compresseur et décompresseur assure la livraison dans l'ordre entre les points d'extrémité de compression. Les profils ROHCv2 peuvent cependant traiter de façon robuste et efficace une quantité limitée de réarrangement après le point de compression au titre de l'algorithme de compression lui-même. De plus, cette amélioration de la prise en charge du réarrangement rend possible aux profils ROHCv2 de traiter plus efficacement le réarrangement avant la liaison.

Logique de fonctionnement : les profils dans la RFC 3095 définissent plusieurs modes de fonctionnement, ayant chacun une logique de mise à jour et des formats d'en-tête compressé différents. Les profils ROHCv2 opèrent en fonctionnement unidirectionnel jusqu'à ce que les premières rétroactions soient reçues pour un contexte (CID) qui est le moment où le fonctionnement bidirectionnel est utilisé ; les formats sont indépendants de la logique de fonctionnement utilisée.

En-tête d'extension IP : les profils dans la RFC 3095 compressent les en-têtes d'extension IP en utilisant la compression de liste. Les profils ROHCv2 traitent plutôt les en-têtes d'extension de la même manière que les autres en-têtes de protocole, c'est-à-dire, en utilisant le mécanisme de chaînage ; on suppose donc que les en-têtes d'extension ne sont pas ajoutés ou supprimés durant la vie d'un contexte (CID) autrement la compression doit être redémarrée pour ce flux.

Encapsulation IP : les profils dans la RFC 3095 peuvent compresser au plus deux niveaux d'en-têtes IP. Les profils ROHCv2 peuvent compresser un nombre arbitraire d'en-têtes IP.

Compression de liste : les profils ROHCv2 ne prennent pas en charge la compression de liste fondée sur la référence.

Robustesse et réparations : les profils ROHCv2 ne définissent pas de format pour le paquet IR-DYN ; chaque profil définit plutôt un en-tête compressé qui peut être utilisé pour effectuer une réparation de contexte plus robuste en utilisant une vérification de CRC de 7 bits. Cela implique aussi que seul l'en-tête IR peut changer l'association entre un CID et le profil qu'il utilise.

Rétroaction : les profils ROHCv2 obligent à un CRC dans le format de FEEDBACK-2, alors que c'est facultatif dans la RFC 3095. Un ensemble d'options de rétroactions différent de celui de la RFC 3095 est aussi utilisé dans ROHCv2.

4.3 Caractéristiques de fonctionnement des profils ROHCv2

La compression robuste d'en-têtes peut être utilisée sur différentes technologies de liaisons. Le paragraphe 4.4 de la [RFC4995] fait la liste des caractéristiques de fonctionnement du canal ROHC. Les profils ROHCv2 visent une large gamme d'applications, et ce paragraphe résume les caractéristiques de fonctionnement qui sont spécifiques de ces profils.

Longueur de paquet : les profils ROHCv2 supposent que la couche inférieure indique la longueur d'un paquet compressé. Les en-têtes compressés ROHCv2 ne contiennent pas d'information de longueur pour la charge utile.

Livraison déclassée entre points d'extrémité de compression : la définition des profils ROHCv2 ne pose pas d'exigences strictes sur la séquence de livraison entre les points d'extrémité de compression, c'est-à-dire, les paquets peuvent être reçus dans un ordre différent de celui de l'envoi par le compresseur et avoir quand même une bonne probabilité d'être bien décompressés.

Cependant, une fréquente livraison en désordre et/ou une profondeur de réarrangement significative vont avoir un impact négatif sur l'efficacité de compression. Plus précisément, si le compresseur peut fonctionner en utilisant une estimation appropriée des caractéristiques de réarrangement du chemin entre les points d'extrémité de compression, les plus grands en-têtes peuvent être envoyés plus souvent pour augmenter la robustesse contre les défaillances de décompression dues à une livraison en désordre. Autrement, l'efficacité de compression va être affaiblie par un accroissement de la fréquence des échecs de décompression et des tentatives de récupération.

5. Vue d'ensemble des profils ROHCv2 (information)

Cette section donne une vue d'ensemble des concepts importants et utiles pour les profils ROHCv2. Ces concepts peuvent être utilisés comme des lignes directrices pour les mises en œuvre mais elles ne font pas partie de la définition normative des profils, car ces concepts se rapportent à l'efficacité de compression du protocole sans impacter les caractéristiques d'interopérabilité d'une mise en œuvre.

5.1 Concepts de compresseur

La compression d'en-tête peut être caractérisée conceptuellement comme l'interaction d'un compresseur avec un automate à états de décompresseur, un par contexte. La responsabilité du compresseur est de convoyer les informations nécessaires pour réussir à décompresser un paquet, sur la base d'une certaine confiance concernant l'état du contexte du décompresseur.

Cette confiance est obtenue de la fréquence et du type des informations que le compresseur envoie quand il met à jour le contexte du décompresseur à partir de l'approche optimiste (paragraphe 5.1.1) et facultativement à partir des messages de rétroaction (voir le paragraphe 6.9) reçus du décompresseur.

5.1.1 Approche optimiste

Un compresseur utilise toujours l'approche optimiste quand il effectue des mises à jour de contexte. Le compresseur répète normalement le même type de mise à jour jusqu'à ce qu'il ait pleine confiance que le décompresseur a bien reçu les informations. Si le décompresseur reçoit bien les en-têtes contenant cette mise à jour, l'état va être disponible pour que le décompresseur traite de plus petits en-têtes compressés.

Si le champ X dans l'en-tête non compressé change de valeur, le compresseur utilise un type d'en-tête qui contient un codage du champ X jusqu'à ce qu'il ait confiance que le décompresseur a reçu au moins un paquet contenant la nouvelle valeur pour X. Le compresseur choisit normalement un format compressé avec le plus petit en-tête qui peut porter les changements nécessaires pour réaliser la confiance.

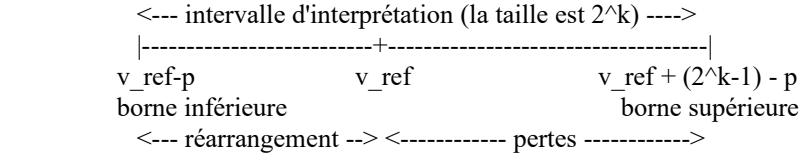
Le nombre de répétitions qui est nécessaire pour obtenir cette confiance est normalement en rapport avec les caractéristiques de perte de paquet et de livraison en désordre de la liaison où la compression d'en-tête est utilisée ; il n'est donc pas défini dans ce document. Il sort du domaine d'application de cette spécification et est laissé à la décision des mises en œuvre.

5.1.2 Compromis entre robustesse aux pertes et réarrangement

La capacité pour un algorithme de compression d'en-tête de traiter les paquets en retard en séquence est principalement limitée par deux facteurs : le décalage de l'intervalle d'interprétation de la fenêtre glissante utilisée pour les champs codés en "lsb" [RFC4997], et l'approche optimiste (voir le paragraphe 5.1.1) pour les champs qui changent rarement.

Champs codés en "lsb" : le décalage d'intervalle d'interprétation spécifie une limite supérieure à la profondeur maximum de réarrangement, par laquelle il est possible au décompresseur de récupérer la valeur originale d'un champ qui change de façon dynamique (c'est-à-dire, qui s'incrémente successivement) codé en utilisant un codage en lsb fondé sur la fenêtre. Sa valeur est normalement liée au nombre de bits compressés en lsb dans le format d'en-tête compressé, et donc croît avec le nombre de bits transmis. Cependant, le décalage et le codage en lsb ne fournissent de robustesse que pour le champ qu'ils compressent, et (implicitement) pour les autres champs changeant en séquence qui sont dérivés de ce champ.

C'est ce qui est montré dans la figure ci-dessous :



où p est le delta maximum négatif, correspondant à la profondeur maximum de réarrangement pour laquelle le codage de lsb peut récupérer la valeur originale du champ ;

où (2^k-1) - p est le delta maximum positif, correspondant au nombre maximum de pertes consécutives pour lequel le codage en lsb peut récupérer la valeur originale du champ ;

où v_ref est la valeur de référence, comme défini dans la méthode de codage en lsb au paragraphe 4.11.5 de la [RFC4997].

Il y a donc un compromis entre la robustesse au réarrangement et la robustesse aux pertes de paquets, par rapport au nombre de bits de MSN nécessaires et la distribution de l'intervalle d'interprétation entre deltas négatifs et positifs dans le MSN.

Champs qui changent rarement : l'approche optimiste (paragraphe 5.1.1) donne la limite supérieure pour la profondeur maximum de réarrangement pour les champs qui changent rarement. Il y a donc un compromis entre efficacité de compression et robustesse. Quand seules des informations sur les MSN ont besoin d'être convoyées au décompresseur, le compromis porte sur le nombre de bits de MSN compressés dans le format d'en-tête compressé. Autrement, le compromis porte sur la mise en œuvre de l'approche optimiste.

En particulier, les mises en œuvre de compresseur devraient ajuster leur stratégie d'approche optimiste pour atteindre les caractéristiques de perte de paquet et de réarrangement de la liaison sur laquelle la compression d'en-tête est appliquée. Par exemple, le nombre de répétitions pour chaque mise à jour d'un champ non codé en lsb peut être augmenté. Le compresseur peut s'assurer que chaque mise à jour est répétée jusqu'à ce qu'il ait une confiance raisonnable qu'au moins un paquet contenant le changement a atteint le décompresseur avant le premier paquet envoyé après cette séquence.

5.1.3 Interactions avec le contexte du décompresseur

Le compresseur commence normalement la compression avec l'hypothèse initiale que le décompresseur n'a pas d'informations utiles pour traiter le nouveau flux, et envoie des paquets d'initialisation et rafraîchissement (IR, *Initialization and Refresh*).

Initialement, quand il envoie le premier paquet IR pour un flux compressé, le compresseur ne s'attend pas à recevoir de retours pour ce flux, jusqu'à ce que de tels retours soient reçus. À ce moment, le compresseur peut alors supposer que le décompresseur va continuer d'envoyer des rétroactions afin de réparer son contexte quand nécessaire. Le premier est appelé "fonctionnement unidirectionnel", et le second "fonctionnement bidirectionnel".

Le compresseur peut alors ajuster le niveau de compression (c'est-à-dire, quel format d'en-tête il choisit) sur la base de sa confiance que le décompresseur a les informations nécessaires pour réussir à traiter les en-têtes compressés qu'il choisit.

En d'autres termes, les responsabilités du compresseur sont de s'assurer que le décompresseur fonctionne avec des informations d'état suffisantes pour réussir à décompresser le type des en-têtes compressés qu'il reçoit, et de permettre au décompresseur de réussir à récupérer ces informations d'état aussitôt que possible autrement. Le compresseur choisit donc le type d'en-tête compressé sur la base des facteurs suivants :

- le résultat de la méthode de codage appliquée à chaque champ ;
- l'approche optimiste, par rapport aux caractéristiques du canal ;
- le type de fonctionnement (unidirectionnel ou bidirectionnel) et si il est en fonctionnement bidirectionnel, les retours reçus du décompresseur (ACK, NACK, STATIC-NACK, et options).

Les méthodes de codage utilisent normalement la ou les valeurs précédentes à partir d'un historique des paquets dont il a précédemment compressé les en-têtes. L'approche optimiste est destinée à assurer qu'est reçu au moins un en-tête compressé contenant les informations pour mettre à jour l'état pour un champ. Finalement, les rétroactions indiquent quelles actions a pris le décompresseur par rapport à ses hypothèses concernant la validité de son contexte (paragraphe 5.2.2) ; elles indiquent quel type d'en-tête compressé le décompresseur peut ou ne peut pas décompresser.

Le décompresseur a les moyens de détecter les échecs de décompression pour tous les formats d'en-têtes compressés (CO) en utilisant la vérification de CRC. Selon la fréquence et/ou le type de défaillance, il peut envoyer un accusé de réception négatif (NACK) ou une demande explicite d'une mise à jour complète de contexte (STATIC-NACK). Cependant, le décompresseur n'a pas de moyen d'identifier la cause de la défaillance, et en particulier la décompression de quel champ est responsable de la défaillance. Le compresseur est donc toujours responsable de la détermination de la réponse la plus convenable à un accusé de réception négatif, en utilisant la confiance qu'il a dans l'état du contexte du décompresseur, quand il choisit le type d'en-tête compressé qu'il va utiliser lorsque il compresse un en-tête.

5.2 Concepts de décompresseur

Le décompresseur utilise normalement le dernier en-tête reçu et validé avec succès (paquets IR) ou vérifiés (paquets CO) comme référence pour la future décompression.

Le décompresseur est responsable de la vérification du résultat de chaque tentative de décompression, de mettre à jour son contexte quand elle est réussie, et finalement de demander des réparations de contexte en faisant un usage cohérent des rétroactions une fois qu'il a commencé d'utiliser les rétroactions.

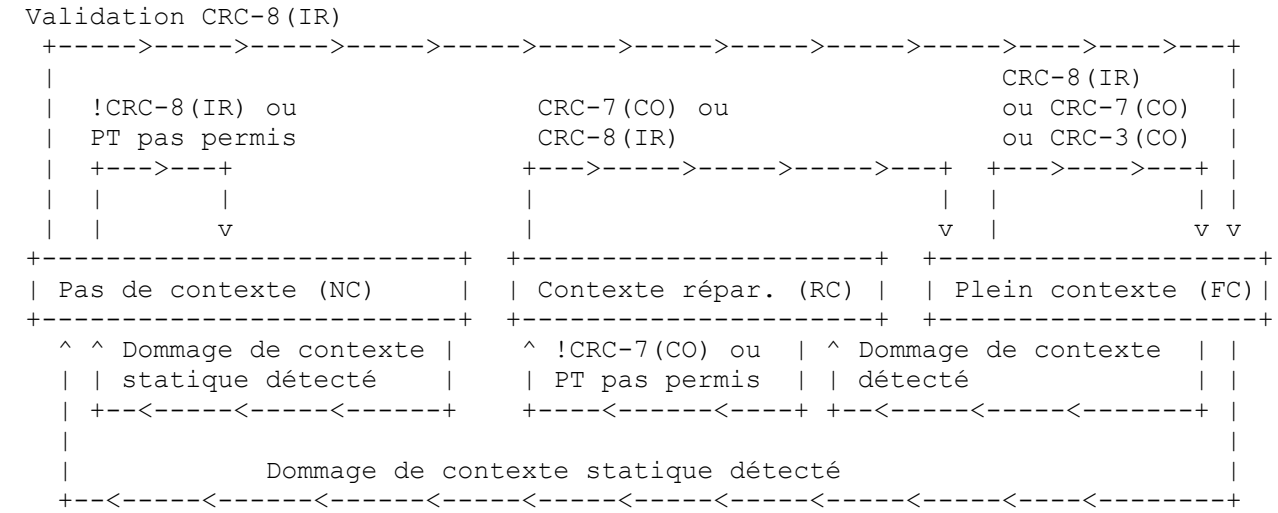
Précisément, le résultat de chaque tentative de décompression est vérifié en utilisant le CRC présent dans l'en-tête compressé ; le décompresseur met à jour les informations de contexte quand ce résultat est vérifié avec succès ; finalement, si le décompresseur utilise les rétroactions une fois pour un flux compressé, il va alors continuer de le faire tant que le contexte correspondant est associé au même profil.

5.2.1 Automate à états de décompresseur

Le fonctionnement du décompresseur peut être représenté comme un automate à états qui définiit trois état : No Context (NC) *(pas de contexte)* Repair Context (RC) *(contexte de réparation)*, et Full Context (FC) *(plein contexte)*.

Le décompresseur commence sans un contexte valide, l'état NC. À réception d'un paquet IR, le décompresseur valide l'intégrité de son en-tête en utilisant la validation CRC-8. Si l'en-tête IR est bien validé, le décompresseur met à jour le contexte et utilise cet en-tête comme en-tête de référence, et passe à l'état FC. Une fois que l'automate à état du décompresseur est entré dans l'état FC, il ne le quitte normalement plus ; seuls des échecs répétés de décompression vont forcer le décompresseur à repasser à un état inférieur. Quand un dommage de contexte est détecté, le décompresseur passe à l'état de contexte de réparation (RC) où il reste jusqu'à ce qu'il réussisse à vérifier une tentative de décompression pour un en-tête compressé avec un CRC de 7 bits ou jusqu'à ce qu'il réussisse à valider un en-tête IR. Quand un dommage de contexte statique est détecté, le décompresseur revient à l'état NC.

Voici l'automate à états pour le décompresseur. Les détails des transitions entre états et logique de décompression sont donnés dans les paragraphes qui suivent la figure.



où:

CRC-8(IR) : validation réussie de CRC-8 pour l'en-tête IR.

!CRC-8(IR) : échec de validation de CRC-8 pour l'en-tête IR.

CRC-7(CO) et/ou CRC-3(CO) : validation réussie de CRC pour la décompression d'un en-tête CO, sur la base du nombre de bits de CRC portés dans l'en-tête CO.

!CRC-7(CO) : échec de la vérification de CRC à la décompression d'un en-tête CO portant un CRC de 7 bits.

PT pas permis : le décompresseur a reçu un type de paquet pour lequel le contexte actuel du décompresseur ne fournit pas assez d'informations d'état valides pour décompresser le paquet.

Dommage de contexte statique détecté : voir la définition au paragraphe 5.2.2.

Dommage de contexte détecté : voir la définition au paragraphe 5.2.2.

5.2.1.1 État Pas de contexte (NC)

Initialement, quand il travaille dans l'état Pas de contexte (NC) le décompresseur n'a pas encore réussi à valider un en-tête IR.

Tentative de décompression : dans l'état NC, seuls les paquets qui portent des informations suffisantes sur les champs statiques (c'est-à-dire, les paquets IR) peuvent être décompressés.

Transition vers le haut : le décompresseur peut passer à l'état Plein contexte (FC) quand la validation de CRC d'un CRC de 8 bits dans un en-tête IR est réussie.

Logique de rétroaction : dans l'état NC, le décompresseur devrait envoyer un STATIC-NACK si un paquet d'un type autre que IR est reçu, ou si un en-tête IR a échoué à la validation de CRC-8, sous réserve de la limitation de débit de retours décrit au paragraphe 5.2.3.

5.2.1.2 État Contexte de réparation (RC)

Dans l'état Contexte de réparation (RC) le décompresseur a réussi à décompresser les paquets pour ce contexte, mais n'est pas sûr que le contexte entier soit valide.

Tentative de décompression : dans l'état RC, seuls les en-têtes couverts par un CRC de 8 bits (c'est-à-dire, IR) ou les en-têtes CO portant un CRC de 7 bits peuvent être décompressés.

Transition vers le haut : le décompresseur peut passer à l'état Plein contexte (FC) quand la vérification de CRC réussit pour un en-tête CO portant un CRC de 7 bits ou quand réussit la validation d'un CRC de 8 bits dans un en-tête IR.

Transition vers le bas : le décompresseur revient à l'état NC si il suppose un dommage au contexte statique.

Logique de rétroaction : dans l'état RC, le décompresseur devrait envoyer un STATIC-NACK quand la validation de CRC-8 d'un en-tête IR échoue, ou quand un en-tête CO portant un CRC de 7 bits échoue et qu'on suppose un dommage du contexte statique, sous réserve de la limitation de débit de rétroaction décrite au paragraphe 5.2.3. Si un autre type de paquet est reçu, le décompresseur devrait le traiter comme un échec de vérification de CRC pour déterminer si un NACK est à envoyer.

5.2.1.3 État Plein contexte (FC)

Dans l'état Plein contexte (FC) le décompresseur suppose que le contexte entier est valide.

Tentative de décompression : dans l'état FC, la décompression peut être tentée sans considération du type de paquet reçu.

Transition vers le bas : le décompresseur revient à l'état RC si il suppose un dommage du contexte. Si le décompresseur suppose un dommage du contexte statique, il passe directement à l'état NC.

Logique de rétroaction : dans l'état FC, le décompresseur devrait envoyer un NACK quand la validation de CRC-8 ou la vérification de CRC de tout type d'en-tête échoue et si un dommage du contexte est supposé, ou il devrait envoyer un STATIC-NACK si un dommage du contexte statique est supposé ; ceci est soumis à la limitation du taux de rétroaction décrit au paragraphe 5.2.3.

5.2.2 Gestion du contexte de décompresseur

Tous les formats d'en-tête portent un CRC et mettent à jour le contexte. Un paquet pour lequel le CRC réussit met à jour les valeurs de référence de tous les champs d'en-tête, explicitement (à partir des informations sur un champ porté dans l'en-tête compressé) ou implicitement (champs déduits d'autres champs).

Le décompresseur peut supposer qu'une partie ou le contexte entier est invalide, quand il échoue à la validation ou à la vérification d'un ou plusieurs en-têtes en utilisant le CRC. Parce que le décompresseur ne peut pas connaître la ou les raisons exactes de l'échec d'un CRC ou quel champ l'a causé, la validité du contexte ne se réfère donc pas à quelle partie spécifique du contexte est réputée valide ou non.

La validité du contexte se rapporte plutôt à la détection d'un problème avec le contexte. Le décompresseur suppose d'abord que le type d'information qui a le plus vraisemblablement causé la ou les défaillances est l'état qui change normalement pour chaque paquet, c'est-à-dire, un dommage de contexte de la partie dynamique du contexte. Si des échecs répétés de

décompression et des réparations non réussies se produisent, le décompresseur supposera alors que le contexte entier, y compris la partie statique, a besoin d'être réparé, c'est-à-dire, un dommage de contexte statique. L'échec de validation du CRC de 3 bits qui protège les champs de contrôle devrait être traité comme un échec de décompression quand le décompresseur certifie la validité de son contexte.

Détection de dommage du contexte : l'hypothèse de dommage du contexte signifie que le décompresseur ne va pas tenter la décompression d'un en-tête CO qui porte seulement un CRC de 3 bits, et va seulement tenter la décompression de ses en-têtes IR ou CO protégés par un CRC-7.

Détection de dommage du contexte statique : l'hypothèse de dommage du contexte statique signifie que le décompresseur s'abstient de tenter la décompression de tout type d'en-tête autre que l'en-tête IR.

Comment ces hypothèses sont faites, c'est-à-dire, comment le dommage du contexte est détecté, est à la discrétion des mises en œuvre. Cela peut se fonder sur le taux d'erreurs résiduelles, où un faible taux d'erreur fait que le décompresseur suppose un dommage plus souvent que sur une liaison où le taux est fort.

Le décompresseur met en œuvre ces hypothèses en choisissant le type d'en-tête compressé pour lequel il va tenter la décompression. En d'autres termes, la validité du contexte se réfère à la capacité d'un décompresseur de tenter (ou non) la décompression de types de paquets spécifiques.

Quand les profils ROHCv2 sont utilisés sur un canal qui ne peut pas garantir la livraison dans l'ordre, le décompresseur peut s'abstenir de mettre à jour son contexte avec le contenu d'un paquet en retard en séquence qui est décompressé avec succès. Ceci est pour éviter de mettre à jour le contexte avec une information qui est plus vieille que ce que le décompresseur a déjà dans son contexte.

5.2.3 Logique des rétroactions

Les profils ROHCv2 peuvent être utilisés dans des environnements avec ou sans capacités de rétroaction du décompresseur au compresseur. ROHCv2 suppose cependant que si un canal de rétroaction ROHC est disponible et si ce canal est utilisé au moins une fois par le décompresseur pour un contexte spécifique, ce canal va être utilisé durant l'opération de compression entière pour ce contexte (c'est-à-dire, un fonctionnement bidirectionnel).

Le cadre ROHC définit trois types de messages de rétroaction : les ACK *(accusé de réception)* les NACK *(accusé de réception négatif)* et les STATIC-NACK *(accusé de réception négatif statique)*. La sémantique de chaque message est définie au paragraphe 5.2.4.1 de la [RFC4995]. La rétroaction à envoyer est couplée avec la gestion du contexte du décompresseur, c'est-à-dire, avec la mise en œuvre des algorithmes de détection de dommage du contexte comme décrit au paragraphe 5.2.2.

Le décompresseur devrait envoyer un NACK quand il suppose un dommage du contexte, et il devrait envoyer un STATIC-NACK quand il suppose un dommage du contexte statique. Le décompresseur n'est pas strictement supposé envoyer des ACK en retour sur une décompression réussie, autrement que pour les besoins d'amélioration de l'efficacité de compression.

Quand les profils ROHCv2 sont utilisés sur un canal qui ne peut pas garantir la livraison dans l'ordre, le décompresseur peut s'abstenir d'envoyer des ACK en retour pour un paquet en retard en séquence qui est décompressé avec succès.

Le décompresseur devrait limiter son taux d'envoi des retours, pour les ACK et les STATIC-NACK/NACK, et devrait éviter d'envoyer des dupliqués inutiles du même type de message de rétroaction qui peuvent être associés au même événement.

6. Profils ROHCv2 (normatifs)

6.1 Paramètres, segmentation, et réarrangement de canaux

Le compresseur NE DOIT PAS utiliser la segmentation ROHC (voir le paragraphe 5.2.5 de la [RFC4995]) c'est-à-dire que l'unité maximum de réception reconstruite (MRRU, *Maximum Reconstructed Reception Unit*) DOIT être réglée à 0, si la configuration du canal ROHC contient au moins un profil ROHCv2 dans la liste des profils pris en charge (c'est-à-dire, le paramètre PROFILES) et si le canal ne peut pas garantir la livraison dans l'ordre des paquets entre les points d'extrémité de compression.

6.2 Fonctionnement du profil par contexte

Les profils ROHCv2 opèrent différemment, par contexte, selon la façon dont le décompresseur utilise le canal de retours, si il y en a un. Une fois que le décompresseur a utilisé le canal de retours pour un contexte, il établit le canal de retours pour ce CID.

Le compresseur commence toujours avec l'hypothèse que le décompresseur ne va pas envoyer de retours quand il initialise un nouveau contexte (voir aussi la définition d'un nouveau contexte au paragraphe 5.1.1. de la [RFC4995], c'est-à-dire, il n'y a pas de canal de retours établi pour le nouveau contexte. À ce point, en dépit de l'utilisation de l'approche optimiste, un échec de décompression est encore possible parce que le décompresseur peut ne pas avoir reçu des informations suffisantes pour décompresser correctement les paquets ; donc, jusqu'à ce que le décompresseur ait établi un canal de retours, le compresseur DEVRAIT envoyer périodiquement des paquets IR. La périodicité peut être fondée sur des temporisations, sur le nombre de paquets compressés envoyés pour le flux, ou toute autre stratégie que choisit la mise en œuvre.

La réception de retours positifs (ACK) ou négatifs (NACK ou STATIC-NACK) du décompresseur établit le canal de retours pour le contexte (CID) pour lequel le retour a été reçu. Une fois qu'il y a un canal de retours établi pour un contexte spécifique, le compresseur peut utiliser ce retour pour estimer l'état actuel du décompresseur. Cela aide à augmenter l'efficacité de compression en fournissant les informations nécessaires pour que le compresseur atteigne le niveau de confiance nécessaire. Quand le canal de retours est établi, il devient superflu pour le compresseur d'envoyer des rafraîchissements périodiques, et il peut plutôt s'appuyer entièrement sur l'approche optimiste et les retours provenant du décompresseur.

Le décompresseur PEUT envoyer des retours positifs (ACK) pour établir initialement le canal de retours pour un flux particulier. Les retours positifs (ACK) ou négatifs (NACK ou STATIC-NACK) établissent ce canal. Une fois qu'il a établi un canal de retours pour un CID, il est EXIGÉ du décompresseur qu'il continue d'envoyer des retours pour la durée de vie du contexte (c'est-à-dire, jusqu'à ce qu'il reçoive un paquet IR qui associe le CID à un profil différent) pour envoyer des demandes de récupération d'erreur et (facultativement) des accusés de réception de mises à jour de contexte significatives.

La compression sans un canal de retours établi va être moins efficace, à cause des rafraîchissements périodiques et de l'absence de retours pour déclencher la récupération d'erreur ; il va aussi y avoir une probabilité légèrement plus forte de propagation de perte comparée au cas où le décompresseur utilise des retours.

6.3 Champs de commande

ROHCv2 définit un certain nombre de champs de contrôle qui sont utilisés par le décompresseur dans son interprétation des formats d'en-tête reçus du compresseur. Les champs de contrôle mentionnés dans les paragraphes qui suivent sont définis en utilisant la notation formelle de la [RFC4997] au paragraphe 6.8.2.4 du présent document.

6.3.1 Numéro de séquence maître (MSN)

Le champ Numéro de séquence maître (MSN) est soit pris d'un champ qui existe déjà dans un des en-têtes du protocole que le profil compresse (par exemple, RTP SN) soit autrement il est créé au compresseur. Il y a un espace de MSN par contexte.

Le champ MSN a les deux fonctions suivantes :
- o Différencier entre les en-têtes de référence quand on reçoit des données de retour ;
- o Déduire la valeur des champs incrémentés (par exemple, Identifiant IPv4).

Il y a un champ MSN dans chaque en-tête ROHCv2, c'est-à-dire, le MSN est toujours présent dans chaque type d'en-tête envoyé par le compresseur. Le MSN est envoyé en entier dans les en-têtes IR, tandis qu'il peut être codé en lsb dans les formats d'en-tête CO. Le décompresseur inclut toujours les LSB du MSN dans le champ Numéro d'accusé de réception dans les retours (voir le paragraphe 6.9). Le compresseur peut utiliser plus tard ce champ pour déduire de quel paquet le décompresseur accuse réception.

Pour les profils pour lesquels le MSN est créé par le compresseur (c'est-à-dire, 0x0102, 0x0104, et 0x0108) on applique ce qui suit :
- o Le compresseur initialise seulement le MSN pour un contexte quand ce contexte est créé ou quand le profil associé à un context change ;
- o Quand le MSN est initialisé, il l'est à une valeur aléatoire ;
- o La valeur du MSN DEVRAIT être incrémentée de un pour chaque paquet que le compresseur envoie pour un CID spécifique

6.3.2 Ratio de réarrangement

Le champ de contrôle reorder_ratio spécifie la quantité de réarrangement qui est traitée par le codage de lsb du MSN. C'est utile quand la compression d'en-tête est effectuée sur des liaisons avec des caractéristiques de réarrangement variables. Le champ de contrôle reorder_ratio donne au compresseur le moyen d'ajuster les caractéristiques de robustesse de la méthode de codage de lsb par rapport au réarrangement et aux pertes consécutives, comme décrit au paragraphe 5.1.2.

6.3.3 Comportement IP-ID

Le champ IP-ID de l'en-tête IPv4 peut avoir différents schémas de changement : séquentiel dans l'ordre des octets du réseau, séquentiel avec échange d'octets, aléatoire ou constant (une valeur constante de zéro, bien que non conforme à la [RFC0791], a été observée en pratique). Il y a un champ de contrôle Comportement IP-ID par en-tête IP. Le champ de contrôle pour le comportement IP-ID de l'en-tête IP le plus interne détermine quel ensemble de formats d'en-tête est utilisé. Le champ de contrôle Comportement IP-ID est aussi utilisé pour déterminer le contenu de l'élément chaîne irrégulière, pour chaque en-tête IP.

Les profils ROHCv2 NE DOIVENT PAS allouer un comportement séquentiel (ordre des octets du réseau ou échange d'octets) à un IP-ID mais celui dans l'en-tête IP le plus interne quand ils compressent plus d'un niveau d'en-têtes IP. C'est parce que seul l'IP-ID de l'en-tête IP le plus interne va probablement avoir une corrélation suffisamment proche avec le MSN pour le compresser comme un champ changeant en séquence. Donc, un compresseur DOIT allouer aux en-têtes de tunnelage soit l'IP-ID zéro constant, soit le comportement d'IP-ID aléatoire.

6.3.4 Comportement de couverture UDP-léger

Le champ de contrôle coverage_behavior spécifie comment le champ Couverture de somme de contrôle de l'en-tête UDP-Léger est compressé avec RoHCv2. Il peut indiquer une des méthodes de codage suivantes : irrégulier, statique, ou déduit.

6.3.5 Cadence d'horodatage

Le champ de contrôle ts_stride est utilisé dans le codage d'horodatage RTP adapté (voir le paragraphe 6.6.8). Il définit l'augmentation attendue de l'horodatage RTP entre les numéros de séquence RTP consécutifs.

6.3.6 Cadence de temps

Le champ de contrôle time_stride est utilisé dans le codage de compression fondé sur le temporisateur (voir le paragraphe 6.6.9). Quand la compression fondée sur le temporisateur est utilisée, time_stride devrait être réglé à la différence attendue des temps d'arrivée entre des paquets RTP consécutifs.

6.3.7 CRC-3 pour champs de commande

Les profils ROHCv2 définissent un CRC-3 calculé sur un certain nombre de champs de contrôle. Ce CRC de 3 bits qui protège les champs de contrôle est présent dans le format d'en-tête pour les types d'en-tête co_common et co_repair.

Le décompresseur DOIT toujours valider l'intégrité des champs de contrôle couverts par ce CRC de 3 bits quand il traite un en-tête compressé co_common ou co_repair.

L'échec de validation des champs de contrôle en utilisant ce CRC devrait être considéré comme un échec de décompression par le décompresseur dans l'algorithme qui atteste de la validité du contexte. Cependant, si la tentative de décompression peut être vérifiée en utilisant le CRC-3 ou le CRC-7 calculé sur l'en-tête non compressé, le décompresseur PEUT quand même transmettre l'en-tête décompressé aux couches supérieures. C'est parce que les champs de contrôle protégés ne sont pas toujours utilisés pour décompresser l'en-tête (c'est-à-dire, co_common ou co_repair) qui met à jour leurs valeurs respectives.

Le polynôme du CRC et la couverture de ce CRC-3 sont définis au paragraphe 6.6.11.

6.4 Reconstruction et vérification

Validation de l'en-tête IR (CRC de 8 bits) : le décompresseur DOIT toujours valider l'intégrité de l'en-tête IR en utilisant le CRC de 8 bits porté dans l'en-tête IR. Quand l'en-tête est validé, le décompresseur met à jour le contexte avec les informations de l'en-tête IR. Autrement, si l'IR ne peut pas être validé, le contexte NE DOIT PAS être mis à jour et l'en-tête IR NE DOIT PAS être livré aux couches supérieures.

Vérification des en-têtes CO (CRC de 3 bits ou de 7 bits) : le décompresseur DOIT toujours vérifier la décompression d'un en-tête CO en utilisant le CRC porté dans l'en-tête compressé. Quand la décompression est vérifiée et réussie, le décompresseur met à jour le contexte avec les informations reçues dans l'en-tête CO ; autrement, si l'en-tête reconstruit échoue à la vérification de CRC, ces mises à jour NE DOIVENT PAS être effectuées.

Un paquet pour lequel la tentative de décompression ne peut pas être vérifiée en utilisant le CRC NE DOIT PAS être livrée aux couches supérieures.

Les mises en œuvre de décompresseur peuvent tenter des mesures correctives ou de réparation sur les en-têtes CO avant d'effectuer les actions ci-dessus, et le résultat de toute tentative de décompression DOIT être vérifié en utilisant le CRC.

6.5 Chaînes d'en-tête compressées

Certains types d'en-tête utilisent une ou plusieurs chaînes contenant des informations de sous en-tête. La fonction d'une chaîne est de grouper les champs sur la base de caractéristiques similaires, comme des champs statiques, dynamiques, ou irréguliers.

Le chaînage est fait en ajoutant à la chaîne un élément pour chaque en-tête dans son ordre d'apparition dans le paquet non compressé, en commençant par les champs les plus externes de l'en-tête.

Dans le texte ci-dessous, le terme <nom de protocole> est utilisé pour identifier la notation formelle des noms correspondant aux différents en-têtes de protocoles. La transposition entre eux est définie dans le tableau suivant :

Protocole	**RFC de définition**	**Nom de protocole**
IPv4	RFC 0791	ipv4
IPv6	RFC 2460	ipv6
UDP	RFC 0768	udp
RTP	RFC 3550	rtp
ESP	RFC 4303	esp
UDP-Léger	RFC 3828	udp_lite
AH	RFC 4302	ah
GRE	RFC 2784, RFC 2890	gre
MINE	RFC 2004	mine
Option destination Ipv6	RFC 2460	dest_opt
Option bond par bond IPv6	RFC 2460	hop_opt
En-tête acheminement IPv6	RFC 2460	rout_opt

Chaîne statique : la chaîne statique consiste en un élément pour chaque en-tête de la chaîne d'en-têtes de protocole qui est compressée, en commençant par l'en-tête IP le plus externe. Dans la description formelle des formats d'en-tête, cet élément de chaîne statique pour chaque type d'en-tête est marqué <protocol_name>_static. La chaîne statique est seulement utilisée dans le format d'en-tête IR.

Chaîne dynamique : la chaîne dynamique consiste en un élément pour chaque en-tête de la chaîne d'en-têtes de protocole qui est compressée, en commençant par l'en-tête IP le plus externe. Dans la description formelle des formats d'en-tête, l'élément de chaîne dynamique pour chaque type d'en-tête est marqué <protocol_name>_dynamic. La chaîne dynamique est seulement utilisée dans les formats d'en-tête IR et co_repair.

Chaîne irrégulière : la structure de la chaîne irrégulière est analogue à celle de la chaîne statique. Pour chaque en-tête compressé qui utilise le format général du paragraphe 6.8, la chaîne irrégulière est ajoutée à une localisation spécifique dans le format général des en-têtes compressés. Dans la description formelle des formats d'en-tête, l'élément de chaîne irrégulière pour chaque type d'en-tête est d'un format dont le nom est muni du suffixe "_irregular". La chaîne irrégulière est utilisée dans tous les en-têtes CO, sauf pour le format co_repair.

Le format de la chaîne irrégulière pour l'en-tête IP le plus interne diffère de celui utilisé pour les en-têtes IP externes, parce que l'en-tête IP le plus interne fait partie de l'en-tête compressé de base. Dans la définition des formats d'en-tête en utilisant la notation formelle, l'argument "is_innermost", qui est passé à la méthode de codage correspondante (ipv4 ou ipv6), détermine quels éléments de chaîne irrégulière utiliser. Le format de l'élément de chaîne irrégulière pour les en-têtes IP externes est aussi déterminé en utilisant un fanion pour TTL/Limite de bonds et TOS/TC. Ce fanion est défini dans le format de certains des en-têtes de base compressés.

Les profils ROHCv2 compressent les en-têtes d'extension comme les autres en-têtes, et donc les en-têtes d'extension ont une chaîne statique, une chaîne dynamique, et une chaîne irrégulière.

Les profils ROHCv2 définissent des chaînes pour tous les en-têtes qui peuvent être compressés, c'est-à-dire, RTP [RFC3550], UDP [RFC0768], ESP [RFC4303], UDP-Léger [RFC3828], IPv4 [RFC0791], IPv6 [RFC2460], AH [RFC4302], GRE [RFC2784], [RFC2890], MINE [RFC2004], en-tête Options destination IPv6 [RFC2460], en-tête Options bond par bond IPv6 [RFC2460], et en-tête Acheminement IPv6 [RFC2460].

6.6. Formats d'en-tête et méthodes de codage

Les formats d'en-tête sont définis en utilisant la notation formelle de ROHC. Certaines des méthodes de codage utilisées dans les formats d'en-tête sont définis dans la [RFC4997], tandis que les autres méthodes sont définies dans cette section.

6.6.1 baseheader_extension_headers

La méthode de codage baseheader_extension_headers saute tous les champs des en-têtes d'extension de l'en-tête IP le plus interne, sans en coder aucun. Les champs de ces en-têtes d'extension sont plutôt codés dans la chaîne irrégulière.

Ce codage est utilisé dans les en-têtes CO (voir le paragraphe 6.8.2). L'en-tête IP le plus interne est combiné avec le ou les autres en-têtes (c'est-à-dire, UDP, UDP-Léger, RTP) pour créer l'en-tête de base compressé. Dans ce cas, il peut y avoir un certain nombre d'en-têtes d'extension entre les en-têtes IP et les autres en-têtes.

L'en-tête de base définit une représentation des en-têtes d'extension, pour se conformer à la syntaxe de la notation formelle ; cette méthode de codage donne cette représentation.

6.6.2 baseheader_outer_headers

La méthode de codage baseheader_outer_headers saute tous les champs du ou des en-têtes d'extension qui n'appartiennent pas à l'en-tête IP le plus interne, sans en coder aucun. Les champs changeants dans les en-têtes externes sont plutôt traités dans la chaîne irrégulière.

Cette méthode de codage, comme pour la méthode de codage baseheader_extension_headers ci-dessus, est nécessaire pour garder la définition des formats d'en-tête syntaxiquement correcte. Elle décrit les en-têtes IP de tunnelage et leurs en-têtes d'extension respectifs (c'est-à-dire, tous les en-têtes situés avant l'en-tête IP le plus interne) ; pour les en-têtes CO (voir le paragraphe 6.8.2).

6.6.3 inferred_udp_length

Le décompresseur déduit la valeur du champ Longueur UDP comme étant la somme de la longueur d'en-tête UDP et de la longueur de charge utile UDP. Le compresseur doit donc s'assurer que le champ Longueur UDP est cohérent avec le ou les champs de longueur des sous en-têtes précédents, c'est-à-dire, il ne doit pas y avoir de bourrage après la charge utile UDP qui soit couvert par la longueur IP.

Cette méthode de codage est aussi utilisée pour le champ Couverture de somme de contrôle UDP-Léger quand il se comporte de la même manière que le champ Longueur UDP (c'est-à-dire, quand la somme de contrôle couvre toujours les charges utiles UDP-Léger entières).

6.6.4 inferred_ip_v4_header_checksum

Cette méthode de codage compresse le champ Somme de contrôle d'en-tête de l'en-tête IPv4. Cette somme de contrôle est définie dans la [RFC0791] comme suit :

Somme de contrôle d'en-tête : 16 bits. Une somme de contrôle seulement sur l'en-tête. Comme certains champs d'en-tête changent (par exemple, la durée de vie) cela est recalculé et vérifié chaque fois que l'en-tête Internet est traité.

L'algorithme de somme de contrôle est : le champ Somme de contrôle est le complément à un de 16 bits de la somme des compléments à un de tous les mots de 16 bits dans l'en-tête. Pour les besoins du calcul de la somme de contrôle, la valeur du champ Somme de contrôle est zéro.

Comme décrit ci-dessus, l'en-tête Somme de contrôle protège lcs bonds individuels contre le traitement d'un en-tête corrompu. Comme les données que protège cette somme de contrôle sont pour la plupart compressées et sont plutôt tirées de l'état mémorisé dans le contexte, cette somme de contrôle devient cumulative avec le CRC ROHC. Quand on utilise cette méthode de codage, la somme de contrôle est recalculée par le décompresseur.

La méthode de codage inferred_ip_v4_header_checksum compresse donc le champ Somme de contrôle d'en-tête de l'en-tête IPv4 jusqu'à une taille de zéro bit, c'est-à-dire, aucun bit n'est transmis dans les en-têtes compressés pour ce champ. En utilisant cette méthode de codage, le décompresseur déduit la valeur de ce champ en utilisant le calcul ci-dessus.

Le compresseur PEUT utiliser l'en-tête Somme de contrôle pour valider la correction de l'en-tête avant de le compresser, pour éviter de traiter un en-tête corrompu.

6.6.5 inferred_mine_header_checksum

Cette méthode de codage compresse la somme de contrôle d'en-tête d'encapsulation minimale. Cette somme de contrôle est définie dans la [RFC2004] comme suit :

Somme de contrôle d'en-tête : le complément à un de 16 bits de la somme des compléments à un de tous les mots de 16 bits dans l'en-tête de transmission minimal. Pour les besoins du calcul de la somme de contrôle, la valeur du champ Somme de contrôle est 0. L'en-tête IP et la charge utile IP (après l'en-tête de transmission minimal) ne sont pas inclus dans ce calcul de somme de contrôle.

La méthode de codage inferred_mine_header_checksum compresse la somme de contrôle d'en-tête d'encapsulation minimale jusqu'à une taille de zéro bit, c'est-à-dire, aucun bit n'est transmis dans les en-têtes compressés pour ce champ. En utilisant cette méthode de codage, le décompresseur déduit la valeur de ce champ en utilisant le calcul ci-dessus.

Les motivations pour déduire cette somme de contrôle sont similaires à celles expliquées au paragraphe 6.6.4.

Le compresseur PEUT utiliser la somme de contrôle d'en-tête d'encapsulation minimale pour valider la correction de l'en-tête avant de le compresser, pour éviter de traiter un en-tête corrompu.

6.6.6 inferred_ip_v4_length

Cette méthode de codage compresse le champ Longueur totale de l'en-tête IPv4. Le champ Longueur totale de l'en-tête IPv4 est défini dans la [RFC0791] comme suit :

Longueur totale : 16 bits. Longueur totale est la longueur du datagramme, mesurée en octets, incluant l'en-tête Internet et les données. Ce champ permet que la longueur d'un datagramme fasse jusqu'à 65 535 octets.

La méthode de codage inferred_ip_v4_length compresse le champ Longueur totale IPv4 jusqu'à une taille de zéro bit, c'est-à-dire, aucun bit n'est transmis dans les en-têtes compressés pour ce champ. En utilisant cette méthode de codage, le décompresseur déduit la valeur de ce champ en comptant les octets de la longueur du paquet entier après décompression.

6.6.7 inferred_ip_v6_length

Cette méthode de codage compresse le champ Longueur de charge utile dans l'en-tête IPv6. Ce champ de longueur est défini dans la [RFC2460] comme suit :

Longueur de charge utile : entier non signé de 16 bits. Longueur de la charge utile IPv6, c'est-à-dire, le reste du paquet qui suit cet en-tête IPv6, en octets. (Noter que tout en-tête d'extension présent est considéré faire partie de la charge utile, c'est-à-dire, y compris le compte de la longueur.)

La méthode de codage "inferred_ip_v6_length" compresse le champ Longueur de charge utile de l'en-tête IPv6 jusqu'à une taille de zéro bit, c'est-à-dire, aucun bit n'est transmis dans les en-têtes compressés pour ce champ. En utilisant cette méthode de codage, le décompresseur déduit la valeur de ce champ en comptant les octets de la longueur du paquet entier après décompression.

Les en-têtes IPv6 qui utilisent l'option Charge utile Jumbo de la [RFC2675] ne vont pas être compressibles avec cette méthode de codage car la valeur du champ Longueur de charge utile ne correspond pas à la longueur du paquet.

6.6.8 Compression d'horodatage RTP adaptée

Ce paragraphe donne des détails supplémentaires sur les codages utilisés pour adapter l'horodatage RTP, comme défini dans la notation formelle au paragraphe 6.8.2.4.

L'horodatage (TS) RTP augmente généralement d'un multiple du numéro de séquence RTP et est donc un candidat convenable pour l'adaptation du codage. Ce facteur d'adaptation est marqué ts_stride dans la définition du profil dans la notation formelle. Le compresseur règle le facteur d'adaptation sur la base du changement de TS par rapport au changement du numéro de séquence RTP.

La valeur par défaut du facteur d'adaptation ts_stride est 160, comme défini au paragraphe 6.8.2.4. Pour utiliser une valeur différente pour ts_stride, le compresseur met à jour explicitement la valeur de ts_stride au décompresseur en utilisant un des formats de l'en-tête qui peut porter cette information.

Quand le compresseur utilise un facteur d'adaptation différent de la valeur par défaut de ts_stride, il peut seulement utiliser le nouveau facteur d'adaptation quand il est assez assuré que le décompresseur a bien calculé le résidu (ts_offset) de la fonction d'adaptation pour l'horodatage. Le compresseur réalise cela en envoyant des valeurs d'horodatage non adaptées, pour permettre au décompresseur d'établir le résidu sur la base du ts_stride actuel. Le compresseur PEUT envoyer l'horodatage non adapté dans le même en-tête compressé qu'utilisé pour établir la valeur de ts_stride.

Une fois que le compresseur est suffisamment sûr que la valeur du facteur d'adaptation et la valeur du résidu ont été établies au décompresseur, le compresseur peut commencer à compresser les paquets en utilisant le nouveau facteur d'adaptation.

Quand le compresseur détecte que la valeur du résidu (ts_offset) a changé, il NE DOIT PAS choisir un format d'en-tête compressé qui utilise le codage d'horodatage adapté avant qu'il ait rétabli le résidu comme décrit ci-dessus.

Quand la valeur du champ Horodatage revient à zéro, la valeur du résidu de la fonction d'adaptation va vraisemblablement changer. Quand cela se produit, le compresseur rétablit la nouvelle valeur de résidu comme décrit ci-dessus.

Si le décompresseur reçoit un en-tête compressé contenant des bits d'horodatage adapté alors que le ts_stride est égal à zéro, il NE DOIT PAS livrer le paquet aux couches supérieures et il DEVRAIT traiter cela comme un échec de vérification de CRC.

Il appartient à la mise en œuvre de compresseur d'appliquer ou non l'adaptation au champ Horodatage RTP (c'est-à-dire, l'utilisation de l'adaptation est FACULTATIVE) et c'est indiqué par le champ de contrôle tsc_indicator. Quand l'adaptation est appliquée au champ Horodatage RTP, la valeur de ts_stride utilisée par le compresseur relève de la mise en œuvre. Une valeur de ts_stride qui est réglée à l'augmentation attendue dans l'horodatage RTP entre les augmentations d'unité consécutives du numéro de séquence RTP va donner le plus de gain pour le codage adapté. D'autres valeurs peuvent donner le même gain dans certaines situations, mais peuvent réduire le gain dans d'autres.

Quand lc codage d'horodatage adapté est utilisé pour des formats d'en-tête qui ne transmettent aucun bit d'horodatage codé en lsb du tout, le codage inferred_scaled_field du paragraphe 6.6.10 est utilisé pour coder l'horodatage.

6.6.9 timer_based_lsb

La méthode de codage de compression fondée sur le temporisateur, timer_based_lsb, compresse un champ dont le schéma de changement est approximativement une fonction linéaire de l'heure.

Ce codage utilise l'horloge locale pour obtenir une approximation de la valeur qu'il code. La valeur approximée est alors utilisée comme valeur de référence avec les num_lsbs_param bits de moindre poids reçus comme valeur codée, où num_lsbs_param représente un nombre de bits qui est suffisant pour représenter de façon univoque la valeur codée en présence de gigue entre les points d'extrémité de compression.

ts_scaled =:= timer_based_lsb(<time_stride_param>, <num_lsbs_param>, <offset_param>)

Les paramètres "num_lsbs_param" et "offset_param" sont les paramètres à utiliser pour le codage de lsb, c'est-à-dire, le nombre de bits de moindre poids et le décalage d'intervalle d'interprétation, respectivement. Le paramètre "time_stride_param" représente la valeur de contexte du champ de contrôle time_stride.

Cette méthode de codage utilise toujours une version adapté du champ qu'elle compresse.

La valeur du champ est décodée en calculant une approximation de la valeur adaptée, en utilisant :

tsc_ref_advanced = tsc_ref + (a_n - a_ref) / time_stride.

où :
- tsc_ref est une valeur de référence de la représentation adaptée du champ ;
- a_n est l'heure d'arrivée associée à la valeur à décoder ;
- a_ref est l'heure d'arrivée associée à l'en-tête de référence ;
- tsc_ref_advanced est une approximation de la valeur adaptée du champ.

Le codage de lsb est alors appliqué en utilisant les num_lsbs_param bits reçus dans l'en-tête compressé et le tsc_ref_advanced comme "ref_value" (conformément au paragraphe 4.11.5 de la [RFC4997]).

L'Appendice B.3 donne un exemple de la façon dont le compresseur peut calculer la gigue.

Le champ de contrôle time_stride contrôle si la méthode timer_based_lsb est ou non utilisée dans l'en-tête CO. Le décompresseur DEVRAIT envoyer l'option CLOCK_RESOLUTION avec une valeur de zéro, si :
o il reçoit une valeur non zéro de time_stride, et
o il n'a pas précédemment envoyé un retour CLOCK_RESOLUTION avec une valeur non zéro.

Ceci est pour permettre à la compression de récupérer du cas où un compresseur active par erreur une compression fondée sur le temporisateur.

La prise en charge et l'usage de la compression fondée sur le temporisateur sont FACULTATIFS pour le compresseur et le décompresseur ; le compresseur n'est pas obligé de régler le champ de contrôle time_stride à une valeur non zéro quand il a reçu une valeur non zéro pour l'option CLOCK_RESOLUTION.

6.6.10 inferred_scaled_field

La méthode de codage inferred_scaled_field code un champ défini comme changeant en relation avec le MSN, et pour lequel l'augmentation par rapport au MSN peut être adaptée par un facteur d'adaptation. Cette méthode de codage est utilisée dans les formats d'en-tête compressé qui ne contiennent aucun bit pour le champ adapté. Dans ce cas, le décompresseur déduit la valeur non adaptée du champ adapté provenant du champ MSN. La valeur non adaptée est calculée selon la formule suivante :

unscaled_value = delta_msn * stride + reference_unscaled_value

où "delta_msn" est la différence dans le MSN entre la valeur de référence du MSN dans le contexte et la valeur du MSN décompressé provenant de ce paquet, "reference_unscaled_value" est la valeur du champ qui va être adapté dans le contexte, et "stride" est la valeur d'adaptation pour ce champ.

Par exemple, quand cette méthode de codage est appliquée à l'horodatage RTP dans le profil RTP, le calcul ci-dessus devient :

horodatage = delta_msn * ts_stride + reference_horodatage

6.6.11 control_crc3_encoding

La méthode control_crc3_encoding donne un CRC calculé sur un certain nombre de champs de contrôle. La définition de cette méthode de codage est la même que pour la méthode de codage "crc" spécifiée au paragraphe 4.11.6 de la [RFC4997], avec la différence que les données couvertes par le CRC sont fournies par une liste de champs de contrôle enchaînés.

En d'autres termes, la définition de la méthode control_crc3_encoding est équivalente à la définition suivante :

```
control_crc_encoding(ctrl_data_value, ctrl_data_length)
{
 UNCOMPRESSED {
 }

 COMPRESSED {
  control_crc3 =:=
   crc(3, 0x06, 0x07, ctrl_data_value, ctrl_data_length) [ 3 ];
```

```
  }
}
```

où le paramètre "ctrl_data_value" relie aux valeurs enchaînées des champs de contrôle suivants, dans l'ordre ci-dessous :

- reorder_ratio, 2 bits bourrés avec 6 MSB de zéros
- ts_stride, 32 bits (seulement pour les profils 0x0101 et 0x0107)
- time_stride, 32 bits (seulement pour les profils 0x0101 et 0x0107)
- msn, 16 bits (non applicable pour les profils 0x0101, 0x0103, et 0x0107)
- coverage_behavior, 2 bits bourrés avec 6 MSB def zéros (seulement pour les profils 0x0107 et 0x0108)
- ip_id_behavior_outer, un octet pour chaque en-tête IPv4 sauf le plus interne dans la chaîne d'en-tête compressible commençant par l'en-tête le plus externe. Chaque octet consiste en 2 bits bourrés avec 6 MSB de zéros.
- ip_id_behavior_innermost, un octet si l'en-tête le plus interne est un en-tête IPv4. L'octet consiste en 2 bits bourrés avec 6 MSB de zéros.

Le "ctrl_data_length" relie à la somme de la longueur du ou des champs de contrôle qui sont applicables au profil.

Le décompresseur utilise le CRC de 3 bits résultant pour valider les champs de contrôle qui sont mis à jour par les formats d'en-tête co_common et co_repair ; ce CRC ne peut pas être utilisé pour vérifier le résultat d'une tentative de décompression.

Ce CRC protège la mise à jour des champs de contrôle, car les valeurs mises à jour ne sont pas toujours utilisées pour décompresser l'en-tête qui les porte et donc ne sont pas protégées par la vérification de CRC-7. Cela empêche des inconvénients qui pourraient se produire si la décompression d'un co_common ou co_repair réussit et que le décompresseur envoie un retour positif, alors que pour une raison quelconque les champs de contrôle sont incorrectement mis à jour.

6.6.12 inferred_sequential_ip_id

Cette méthode de codage est utilisée avec un comportement IP-ID séquentiel (séquentiel ou à échange d'octets séquentiel) et quand il n'y a pas de bit IP-ID codé dans l'en-tête compressé. Dans ce cas, le décalage IP-ID du MSN est constant, et le IP-ID augmente de la même quantité que le MSN (similaire à la méthode de codage inferred_scaled_field).

Le décompresseur calcule la valeur pour le IP-ID selon la formule suivante :

IP-ID = delta_msn + reference_IP_ID_value

où "delta_msn" est la différence entre la valeur de référence du MSN dans le contexte et la valeur non comprimée du MSN associé à l'en-tête compressé, et où "reference_IP_ID_value" est la valeur du IP-ID dans le contexte. Pour le comportement IP-ID échangé (c'est-à-dire, quand ip_id_behavior_innermost est réglé à IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED) les octets de "reference_IP_ID_value" et "IP-ID" sont échangés par rapport aux champs correspondants dans le contexte.

Si le comportement IP-ID est aléatoire ou zéro, cette méthode de codage ne met à jour aucun champ.

6.6.13 list_csrc(cc_value)

Cette méthode de codage compresse la liste des identifiants de CSRC RTP en utilisant la compression de liste. Ce codage établit un contenu pour les différents identifiants de CSRC (éléments) et une liste décrivant l'ordre dans lequel ils apparaissent.

Le compresseur passe un argument (cc_value) à cette méthode de codage : c'est la valeur du champ CC prise de l'en-tête RTP. Le décompresseur est obligé de lier la valeur de cet argument au nombre d'éléments de la liste, ce qui va permettre au décompresseur de reconstruire correctement le champ CC.

6.6.13.1 Compression de liste

Les identifiants de CSRC dans le paquet non compressé peuvent être représentés comme une liste ordonnée, dont l'ordre et la présence sont généralement constants entre les paquets. La structure générale de cette liste est la suivante :

Quand on effectue la compression de liste sur une liste de CSRC, chaque élément est la valeur non compressée d'un identifiant de CSRC.

Les principes de base de la compression fondée sur la liste sont les suivants :
Quand on initialise le contexte, la représentation complète de la liste des identifiants de CSRC est transmise.
Une fois que le contexte a été initialisé :
- quand la liste est inchangée, un en-tête compressé qui ne contient pas d'information sur la liste peut être utilisé ;
- quand la liste change, une liste compressée est envoyée dans l'en-tête compressé, incluant une représentation de sa structure et ordre. Les éléments antérieurement inconnus sont envoyés incompressés dans la liste, alors que les éléments connus précédemment sont seulement représentés par un indice pointant sur l'élément mémorisé dans le contexte.

6.6.13.2 Compression d'élément fondée sur un tableau

La compression d'élément fondée sur un tableau compresse les éléments individuels envoyés dans les listes compressées. Le compresseur alloue un identifiant unique, "Index", à chaque élément "Item" d'une liste.

Logique de compresseur : le compresseur maintient conceptuellement un tableau d'éléments contenant tous les éléments, indexés en utilisant "Index". La paire (Index, Item) est envoyée ensemble dans des listes compressées jusqu'à ce que le compresseur ait une confiance suffisante que le décompresseur a observé la transposition entre les éléments et leurs indices respectifs. La confiance est obtenue de la réception d'un accusé de réception du décompresseur, ou par l'envoi de paires (Index, Item) en utilisant l'approche optimiste. Une fois que la confiance est obtenue, l'indice seul est envoyé dans des listes compressées pour indiquer la présence de l'élément correspondant à cet indice.

Le compresseur PEUT réinitialiser le tableau des éléments à réception d'un accusé de réception négatif.

Le compresseur PEUT réallouer un indice existant à un nouvel élément en rétablissant la transposition avec la procédure décrite ci-dessus.

Logique de décompresseur : le décompresseur maintient conceptuellement un tableau des éléments qui contient toutes les paires de (Index, Item) reçues. Le tableau des éléments est mis à jour chaque fois qu'une paire (Index, Item) est reçue et que la décompression est réussie (vérification de CRC, ou validation de CRC-8). Le décompresseur restitue les éléments du tableau chaque fois qu'un Index est reçu sans être accompagné d'un élément.

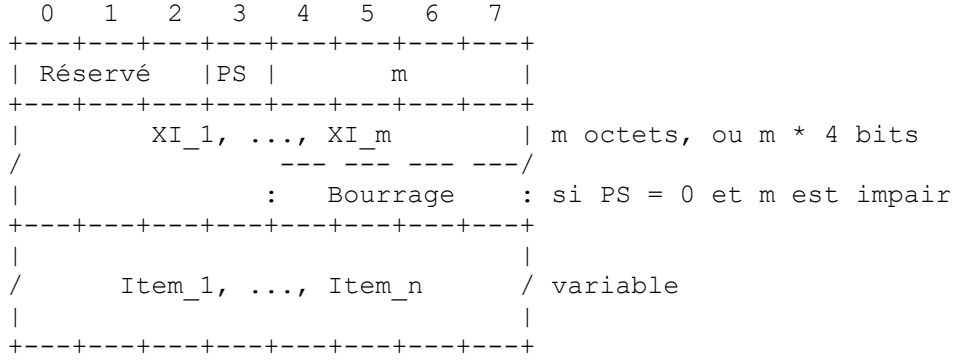
Si un indice est reçu sans être accompagné d'un élément et si le décompresseur n'a pas de contexte pour cet indice, le décompresseur NE DOIT PAS livrer le paquet aux couches supérieures.

6.6.13.3 Codage de listes compressées

Chaque élément présent dans une liste compressée est représenté par :
o un Index dans le tableau des élements, et un bit de présence qui indique si une représentation compressée de l'élément est présente dans la liste.
o un élément (si le bit de présence est établi).

Si le bit de présence n'est pas établi, l'élément doit être déjà connu du décompresseur.

Une liste compressée d'éléments utilise le codage suivant :



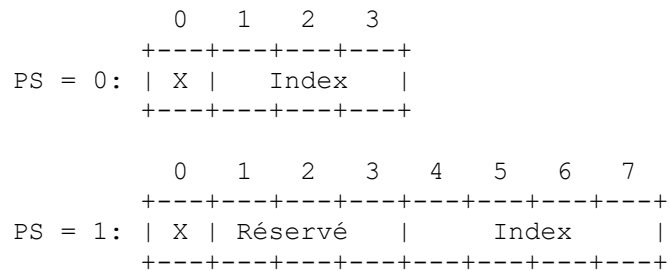
Réservé : DOIT être réglé à zéro ; autrement, le décompresseur DOIT éliminer le paquet.

PS : indique la taille des champs XI :
PS = 0 indique des champs de 4 bits ;
PS = 1 indique des champs XI de 8 bits.

M : nombre d'éléments XI dans la liste compressée. Aussi, la valeur de l'argument cc_value du codage list_csrc (voir le paragraphe 6.6.13).

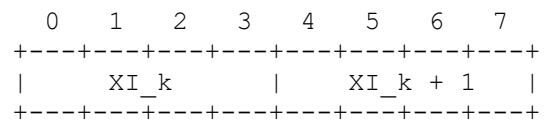
XI_1, ..., XI_m : m éléments XI. Chaque XI représente un élément de la liste des éléments de l'en-tête non compressé, dans le même ordre que celui dans lequel ils apparaissent dans l'en-tête non compressé.

Le format d'un élément XI est le suivant :



X : indique si l'élément est présent dans la liste :
X = 1 indique que l'élément correspondant à l'indice est envoyé dans la liste Item_1, ..., Item_n ;
X = 0 indique que l'élément correspondant à l'indice n'est pas envoyé.
Réservé : DOIT être réglé à zéro ; autrement, le décompresseur DOIT éliminer le paquet.
Index : indice dans le table d'éléments, voir le paragraphe 6.6.13.4.

Quand des éléments XI de quatre bits sont utilisés, les éléments XI sont placés en octets de la manière suivante :



Bourrage : un champ Bourrage de 4 bits est présent quand PS = 0 et que le nombre de XI est impair. Le champ Bourrage DOIT être réglé à zéro ; autrement, le décompresseur DOIT éliminer le paquet.

Item 1, ..., item n : chaque élément correspond à un XI avec X = 1 dans XI 1, ..., XI m. Chaque entrée dans la liste des éléments est la représentation non compressée d'un identifiant de CSRC.

6.6.13.4 Transpositions de tableaux d'éléments

Le tableau d'éléments pour la compression de liste est limité à 16 éléments différents, car l'en-tête RTP peut seulement porter 15 identifiants de CSRC simultanés. L'effet d'avoir plus de 16 éléments dans le tableau des éléments est seulement une légère surcharge du compresseur quand les éléments sont échangés en entrée/sortie du tableau des éléments.

6.6.13.5 Listes compressées dans les chaînes dynamiques

Une liste compressée qui fait partie de la chaîne dynamique doit avoir tous ses éléments présents, c'est-à-dire, tous les bits X dans la liste XI DOIVENT être établis. Tous les éléments précédemment établis dans le tableau d'éléments qui ne sont pas présents dans la liste décompressée provenant de ce paquet DOIVENT aussi être conservés dans le contexte du décompresseur.

6.7 Méthodes de codage avec paramètres externes comme arguments

Un certain nombre de méthodes de codage du paragraphe 6.8.2.4 ont un ou plusieurs arguments pour lesquels la déduction de la valeur du paramètre sort du domaine d'application de la spécification des formats d'en-tête de ROHC-FN [RFC4997].

Voici une liste des méthodes de codage avec des paramètres externes comme arguments, d'après le paragraphe 6.8.2.4 :
o udp(profile_value, reorder_ratio_value)
o udp_lite(profile_value, reorder_ratio_value, coverage_behavior_value)

- o esp(profile_value, reorder_ratio_value)
- o rtp(profile_value, ts_stride_value, time_stride_value, reorder_ratio_value)
- o ipv4(profile_value, is_innermost, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value))
- o ipv6(profile_value, is_innermost, outer_ip_flag, reorder_ratio_value))
- o iponly_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- o udp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- o udplite_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- o esp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- o rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
- o udplite_rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value, coverage_behavior_value)

On applique ce qui suit pour tous les paramètres mentionnés ci-après : au compresseur, la valeur du paramètre est réglée en accord avec les recommandations pour chaque paramètre. Au décompresseur, la valeur du paramètre est réglée à indéfini et va être liée par les méthodes de codage, sauf mention contraire.

Voici une liste des arguments externes avec leur définition :

- o profile_value : règle le nombre de 16 bits qui identifie le profil utilisé pour compresser ce paquet. Quand on traite la chaîne statique au décompresseur, ce paramètre est réglé à la valeur du champ de profil dans l'en-tête IR (voir le paragraphe 6.8.1).

- o reorder_ratio_value : valeur d'entier de deux bits, en utilisant une des constantes dont le nom commence par le préfixe REORDERING_ et comme défini au paragraphe 6.8.2.4.

- o ip_id_behavior_value : valeur d'entier de deux bits, en utilisant une des constantes dont le nom commence par le préfixe IP_ID_BEHAVIOR_ et comme défini au paragraphe 6.8.2.4.

- o coverage_behavior_value : valeur d'entier de deux bits, en utilisant une des constantes dont le nom commence par le préfixe UDP_LITE_COVERAGE_ et comme défini au paragraphe 6.8.2.4.

- o outer_ip_flag : ce paramètre est réglé à 1 si au moins un des champs TOS/TC ou TTL/Hop Limit dans les en-têtes IP externes a changé par rapport à sa valeur de référence dans le contexte ; autrement, il est réglé tout à 0. Ce fanion peut seulement être réglé à 1 pour le format d'en-tête "co_common" dans les différents profils.

- O is_innermost : ce fanion booléen est réglé à 1 lors du traitement du plus interne des en-têtes IP compressibles ; autrement, il est réglé à 0.

- o ts_stride_value : la valeur de ce paramètre devrait être réglée à l'augmentation attendue de l'horodatage RTP entre des numéros de séquence RTP consécutifs. La valeur choisie est spécifique de la mise en œuvre. Voir aussi au paragraphe 6.6.8.

- o time_stride_value : la valeur de ce paramètre devrait être réglée au temps inter arrivées attendu entre des paquets consécutifs pour le flux. La valeur choisie est spécifique de la mise en œuvre. Ce paramètre DOIT être réglé à zéro, sauf si le compresseur a reçu un message de rétroaction avec l'option CLOCK_RESOLUTION réglée à une valeur non zéro. Voir aussi au pragraphe 6.6.9.

6.8 Formats d'en-têtes

Les profils ROHCv2 utilisent deux types différents d'en-têtes : le type d'en-tête Initialisation et rafraîchissement (IR) et le type d'en-tête Compressé (CO).

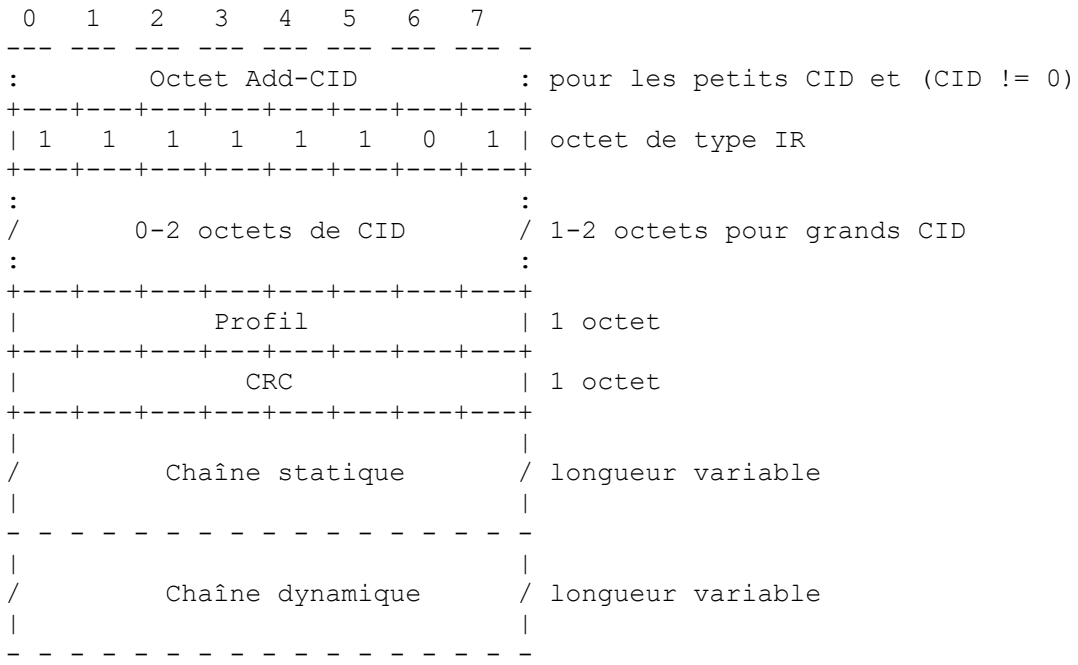
Le type d'en-tête CO définit un certain nombre de formats d'en-tête : il y a deux ensembles de formats d'en-tête de base, avec quelques formats supplémentaires qui sont communs aux deux ensembles.

6.8.1 Initialisation et rafraîchissement de format d'en-tête (IR)

Le format d'en-tête IR utilise la structure de l'en-tête IR ROHC définie au paragraphe 5.2.2.1 de la [RFC4995].

Type d'en-tête : IR. Ce format d'en-tête communique la partie statique et la partie dynamique du contexte.

L'en-tête IR ROHCv2 a le format suivant :



CRC : CRC de 8 bits sur l'en-tête IR entier, incluant tous les champs de CID et jusqu'à la fin de la chaîne dynamique, en utilisant le polynôme défini dans la [RFC4995]. Pour les besoins du calcul du CRC, le champ CRC est à zéro.

Chaîne statique : voir le paragraphe 6.5.

Chaîne dynamique : voir le paragraphe 6.5.

6.8.2 Formats d'en-tête compressés (CO)

6.8.2.1 Raison de la conception des formats compressés d'en-tête de base

Les formats d'en-tête compressé sont définis comme deux ensembles séparés pour chaque profil : un ensemble pour les en-têtes où l'en-tête IP le plus interne contient un IP-ID séquentiel (dans l'ordre des octets du réseau ou par échange d'octets) et un ensemble pour les en-têtes sans IP-ID séquentiel (aléatoire, zéro, ou non IP-ID). Il y a aussi un certain nombre de formats d'en-tête communs entre les deux ensembles. Dans la description qui suit, la convention de désignation utilisée pour les formats d'en-tête qui appartiennent à l'ensemble séquentiel est d'inclure "seq" dans le nom du format, tandis que similairement, "rnd" est utilisé pour ceux qui appartiennent à l'ensemble non séquentiel.

La conception des formats d'en-tête est dérivée de l'analyse de comportement du champ qui se trouve dans l'appendice A.

Tous les en-têtes de base compressés transmettent des bits MSN codés en lsb et un CRC.

Les formats d'en-tête suivants existent pour tous les profils définis dans le présent document, et sont communs aux deux ensembles de formats d'en-tête séquentiel et aléatoire :

co_common : ce format peut être utilisé pour mettre à jour le contexte quand le schéma de changement établi du champ dynamique change, pour tout champ dynamique. Cependant, tous les champs dynamiques ne sont pas mis à jour en portant leur valeur non compressée ; certains champs peuvent seulement être transmis en utilisant une représentation compressée. Ce format est particulièrement utile quand un champ qui change rarement a besoin d'être mis à jour. Ce format contient un ensemble de fanions pour indiquer quels champs sont présents dans l'en-tête, et sa taille peut varier en conséquence. Ce format est protégé par un CRC de 7 bits. Il peut metre à jour des champs de contrôle, et il porte donc aussi un CRC de 3 bits pour protéger ces champs. Ce format est similaire dans son objet au format 3 d'extension UOR-2 de la [RFC3095].

co_repair : ce format peut être utilisé pour mettre à jour le contexte de tous les champs dynamiques en portant leur valeur non compressée. Ceci est particulièrement utile quand un dommage du contexte est supposé (par exemple, de la

réception d'un NACK) et qu'une réparation de contexte est effectuée. Ce format est protégé par un CRC de 7 bits. Il porte aussi un CRC de 3 bits sur les champs de contrôle qu'il peut mettre à jour. Ce format est similaire dans son objet au format IR-DYN de la [RFC3095] quand on effectue des réparations de contexte.

pt_0_crc3 : ce format porte seulement le MSN ; il peut donc seulement mettre à jour le MSN et les champs qui sont dérivés du MSN, comme le IP-ID et l'horodatage RTP (pour les profils applicables). Il est protégé par un CRC de 3 bits. Ce format est équivalent au format d'en-tête UO-0 dans la [RFC3095].

pt_0_crc7 : ce format a les mêmes propriétés que pt_0_crc3, mais est protégé par un CRC de 7 bits et contient une plus grande quantité de bits MSN codés en lsb. Ce format est utile dans les environnements où une grande quantité de réarrangement ou un fort taux d'erreurs résiduelles peut se produire.

Les descriptions de format d'en-tête suivantes s'appliquent aux profils 0x0101 et 0x0107.

pt_1_rnd : ce format peut porter des changements au MSN et au bit Marqueur RTP, et il peut mettre à jour l'horodatage RTP en utilisant un codage d'horodatage adapté. Il est protégé par un CRC de 3 bits. Il est similaire dans son objet au format UO-1 dans la [RFC3095].

pt_1_seq_id : ce format peut porter des changements au MSN et au IP-ID. Il est protégé par un CRC de 3 bits. Il est similaire dans son objet au format UO-1-ID dans la [RFC3095].

pt_1_seq_ts : ce format peut porter des changements au MSN et au bit Marqueur RTP, et il peut mettre à jour l'horodatage RTP en utilisant un codage d'horodatage adapté. Il est protégé par un CRC de 3 bits. Il est similaire dans son objet au format UO-1-TS dans la [RFC3095].

pt_2_rnd : ce format peut porter des changements au MSN, au bit Marqueur RTP, et à l'horodatage RTP. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format UOR-2 dans la [RFC3095].

pt_2_seq_id : ce format peut porter des changements au MSN et au IP-ID. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format UO-2-ID dans la [RFC3095].

pt_2_seq_ts : ce format peut porter des changements au MSN, au bit Marqueur RTP et it peut mettre à jour l'horodatage RTP en utilisant un codage d'horodatage adapté. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format UO-2-TS de la [RFC3095].

pt_2_seq_both : ce format peut porter des changements à l'horodatage RTP et à l'IP-ID, en plus du MSN et du bit Marqueur. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format d'extension 1 UOR-2-ID dans la [RFC3095].

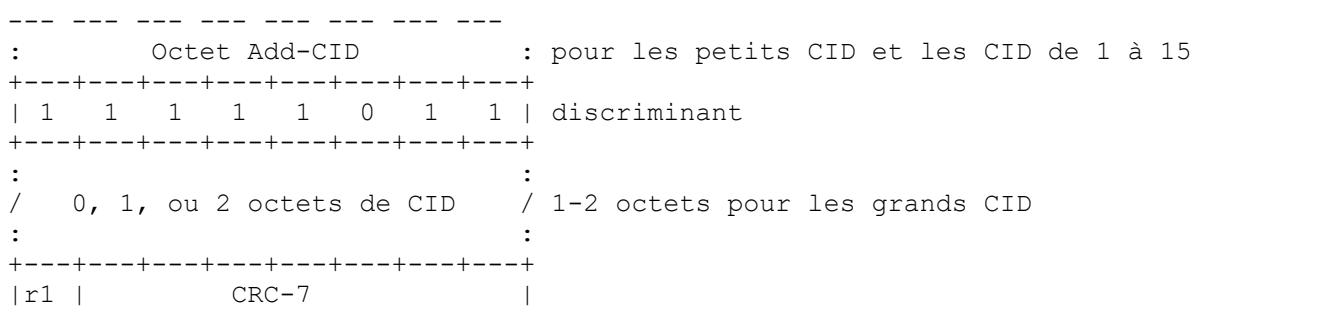
Les descriptions de format d'en-tête suivantes s'appliquent aux profils 0x0102, 0x0103, 0x0104, et 0x0108.

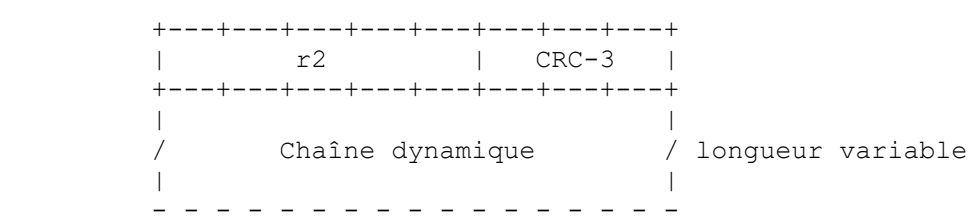
pt_1_seq_id : ce format peut porter des changements au MSN et à l'IP-ID. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format UO-1-ID dans la [RFC3095].

pt_2_seq_id : ce format peut porter des changements au MSN et à l'IP-ID. Il est protégé par un CRC de 7 bits. Il est similaire dans son objet au format UO-2-ID dans la [RFC3095].

6.8.2.2 Format d'en-tête co_repair

L'en-tête ROHCv2 co_repair a le format suivant :





r1 : DOIT être réglé à zéro ; autrement, le décompresseur DOIT éliminer le paquet.

CRC-7 : CRC de 7 bits sur l'en-tête non compressé entier, calculé en utilisant la méthode de codage crc7 (data_value, data_length) définie au paragraphe 6.8.2.4, où data_value correspond à la chaîne entière d'en-tête non compressé et où data_length correspond à la longueur de cette chaîne d'en-tête.

R2 : DOIT être réglé à zéro ; autrement, le décompresseur DOIT éliminer le paquet.

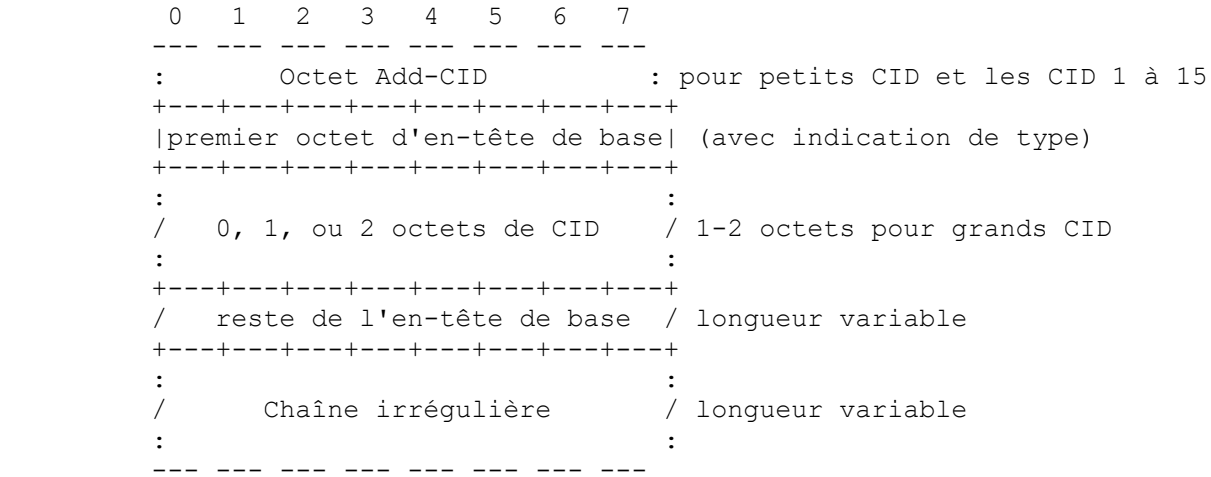
CRC-3 : codé en utilisant la méthode control_crc3_encoding définie au paragraphe 6.6.11.

Chaîne dynamique : voir le paragraphe 6.5.

6.8.2.3 Format d'en-tête général CO

Le format d'en-tête CO communique les irrégularités dans l'en-tête de paquet. Tous les formats CO portent un CRC et peuvent mettre à jour le contexte. Tous les formats d'en-tête CO utilisent le format général défini dans cette section, à l'exception du format co_repair, qui est défini au paragraphe 6.8.2.2.

Le format général pour un en-tête compressé est le suivant :



L'en-tête de base dans la figure ci-dessus est la représentation compressée de l'en-tête IP le plus interne et des autres en-têtes, si il en est, dans le paquet non compressé. Les formats d'en-tête de base sont définis au paragraphe 6.8.2.4. Dans la description formelle des formats d'en-tête, l'en-tête de base pour chaque profil est marqué <profile_name>_baseheader, où <profile_name> est défini dans le tableau suivant :

Numéro de profil	**profile_name**
0x0101	rtp
0x0102	udp
0x0103	esp
0x0104	ip
0x0107	udplite_rtp
0x0108	udplite

6.8.2.4 Formats d'en-tête dans ROHC-FN

Ce paragraphe définit l'ensemble complet des formats d'en-tête de base pour les profils ROHCv2. Les formats d'en-tête de base sont définis en utilisant la notation formelle de ROHC [RFC4997].

Note : les chaînes irrégulières, statiques, et dynamiques (voir le paragraphe 6.5) sont définies sur plusieurs méthodes de codage et sont incorporées dans les formats désignés correspondants au sein de ces méthodes de codage. En particulier, noter que les chaînes statiques et dynamiques vont ordinairement ensemble. Les champs non compressés sont définis à travers ces deux formats combinés, plutôt que dans l'un ou l'autre. Les éléments de chaîne irrégulière sont de même combinés avec un format d'en-tête de base.

Constantes
Constantes de comportement IP-ID
IP_ID_BEHAVIOR_SEQUENTIAL = 0 ;
IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED = 1 ;
IP_ID_BEHAVIOR_RANDOM = 2 ;
IP_ID_BEHAVIOR_ZERO = 3 ;

Constantes de comportement de couverture de somme de contrôle UDP-léger
UDP_LITE_COVERAGE_INFERRED = 0 ;
UDP_LITE_COVERAGE_STATIC = 1 ;
UDP_LITE_COVERAGE_IRREGULAR = 2 ;
La valeur 3 est réservée et ne peut pas être utilisée pour le comportement de couverture.

Décalage de réarrangement variable
REORDERING_NONE = 0 ;
REORDERING_QUARTER = 1 ;
REORDERING_HALF = 2 ;
REORDERING_THREEQUARTERS = 3;

Noms et versions de profil
PROFILE_RTP_0101 = 0x0101 ;
PROFILE_UDP_0102 = 0x0102 ;
PROFILE_ESP_0103 = 0x0103 ;
PROFILE_IP_0104 = 0x0104 ;
PROFILE_RTP_0107 = 0x0107 ; // avec UDP-Léger
PROFILE_UDPLITE_0108 = 0x0108 ; // sans RTP

Valeurs par défaut de codage d'horodatage RTP
TS_STRIDE_DEFAULT = 160 ;
TIME_STRIDE_DEFAULT = 0 ;

Champs de commandes globales

```
CONTROL {
  profile                    [ 16 ] ;
  msn                        [ 16 ] ;
  reorder_ratio              [  2 ] ;      // les champs ip_id sont pour l'en-tête IP le plus interne seulement
  ip_id_offset               [ 16 ] ;
  ip_id_behavior_innermost   [  2 ] ;
Les commandes suivantes ne sont utilisées que dans les profils fondés sur RTP :
  ts_stride                  [ 32 ] ;
  time_stride                [ 32 ] ;
  ts_scaled                  [ 32 ] ;
  ts_offset                  [ 32 ] ;      // profils fondés sur UDP-léger seulement
  coverage_behavior          [  2 ] ;
}
```

Méthodes de codage non spécifiées en syntaxe FN :

```
baseheader_extension_headers      "défini au paragraphe 6.6.1" ;
baseheader_outer_headers          "défini au paragraphe 6.6.2" ;
control_crc3_encoding             "défini au paragraphe 6.6.11" ;
inferred_ip_v4_header_checksum    "défini au paragraphe 6.6.4" ;
inferred_ip_v4_length             "défini au paragraphe 6.6.6" ;
inferred_ip_v6_length             "défini au paragraphe 6.6.7" ;
inferred_mine_header_checksum     "défini au paragraphe 6.6.5" ;
```

```
inferred_scaled_field                "défini au paragraphe 6.6.10" ;
inferred_sequential_ip_id            "défini au paragraphe 6.6.12" ;
inferred_udp_length                  "défini au paragraphe 6.6.3" ;
list_csrc(cc_value)                  "défini au paragraphe 6.6.13" ;
timer_based_lsb(time_stride, k, p)   "défini au paragraphe 6.6.9" ;
```

Méthodes de codage générales

```
static_or_irreg(flag, width)
{
 UNCOMPRESSED {
  field [ width ];
 }

 COMPRESSED irreg_enc {
  ENFORCE(flag == 1);
  field =:= irregular(width) [ width ];
 }

 COMPRESSED static_enc {

  ENFORCE(flag == 0);
  field =:= static [ 0 ];
 }
}

optional_32(flag)
{
 UNCOMPRESSED {
  item [ 0, 32 ];
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  item =:= irregular(32) [ 32 ];
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  item =:= compressed_value(0, 0) [ 0 ];
 }
}

// Envoie la valeur entière , ou garde la valeur précédente
sdvl_or_static(flag)
{
 UNCOMPRESSED {
  field [ 32 ];
 }

 COMPRESSED present_7bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^7);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '0' [ 1 ];
  field                 [ 7 ];
 }

 COMPRESSED present_14bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^14);
  ENFORCE(field.CVALUE == field.UVALUE);
```

```
  discriminator =:= '10'   [  2 ];
  field                    [ 14 ];
 }

 COMPRESSED present_21bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^21);

  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '110'  [  3 ];
  field                    [ 21 ];
 }

 COMPRESSED present_28bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^28);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '1110'  [  4 ];
  field                     [ 28 ];
 }

 COMPRESSED present_32bit {
  ENFORCE(flag == 1);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '11111111'  [  8 ];
  field                         [ 32 ];
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  field =:= static;
 }
}

// Envoie la valeur entière, ou revient à la valeur par défaut
sdvl_or_default(flag, default_value)
{
 UNCOMPRESSED {
  field [ 32 ];
 }

 COMPRESSED present_7bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^7);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '0' [ 1 ];
  field                 [ 7 ];
 }

 COMPRESSED present_14bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^14);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '10'   [  2 ];
  field                    [ 14 ];
 }

 COMPRESSED present_21bit {
  ENFORCE(flag == 1);
  ENFORCE(field.UVALUE < 2^21);
  ENFORCE(field.CVALUE == field.UVALUE);
  discriminator =:= '110'  [  3 ];
```

```
    field                [ 21 ];
   }

   COMPRESSED present_28bit {
    ENFORCE(flag == 1);
    ENFORCE(field.UVALUE < 2^28);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '1110'  [  4 ];
    field                     [ 28 ];
   }

   COMPRESSED present_32bit {
    ENFORCE(flag == 1);
    ENFORCE(field.CVALUE == field.UVALUE);
    discriminator =:= '11111111'  [  8 ];
    field                         [ 32 ];
   }

   COMPRESSED not_present {
    ENFORCE(flag == 0);
    field =:= uncompressed_value(32, default_value);
   }
  }

  lsb_7_or_31
  {
   UNCOMPRESSED {
    item [ 32 ];
   }

   COMPRESSED lsb_7 {
    discriminator =:= '0'                    [  1 ];
    item          =:= lsb(7, ((2^7) / 4) - 1)   [  7 ];
   }

   COMPRESSED lsb_31 {
    discriminator =:= '1'                    [  1 ];
    item    =:= lsb(31, ((2^31) / 4) - 1) [ 31 ];
   }
  }

  crc3(data_value, data_length)
  {

   UNCOMPRESSED {
   }
   COMPRESSED {
    crc_value =:= crc(3, 0x06, 0x07, data_value, data_length) [ 3 ];
   }
  }

  crc7(data_value, data_length)
  {
   UNCOMPRESSED {
   }

   COMPRESSED {
    crc_value =:= crc(7, 0x79, 0x7f, data_value, data_length) [ 7 ];
   }
  }
```

```
// La méthode de codage pour mettre à jour un champ adapté et ses champs de contrôle associés. Devrait être utilisé à la fois quand la valeur est adaptée ou non adaptée dans un format compressé. N'a pas de côté non compressé.

field_scaling(stride_value, scaled_value, unscaled_value, residue_value)
{
 UNCOMPRESSED {
// Rien
 }

 COMPRESSED no_scaling {
  ENFORCE(stride_value == 0);
  ENFORCE(residue_value == unscaled_value);
  ENFORCE(scaled_value == 0);
 }

 COMPRESSED scaling_used {
  ENFORCE(stride_value != 0);
  ENFORCE(residue_value == (unscaled_value % stride_value));
  ENFORCE(unscaled_value == scaled_value * stride_value + residue_value);
 }
}
```

Options d'en-tête de destination IPv6

```
ip_dest_opt
{
 UNCOMPRESSED {
  next_header [ 8 ];
  length      [ 8 ];
  value       [ length.UVALUE * 64 + 48 ];
 }

 DEFAULT {
  length      =:= static;
  next_header =:= static;
  value       =:= static;
 }

 COMPRESSED dest_opt_static {
  next_header =:= irregular(8) [ 8 ];
  length      =:= irregular(8) [ 8 ];
 }

 COMPRESSED dest_opt_dynamic {
  value =:= irregular(length.UVALUE * 64 + 48) [ length.UVALUE * 64 + 48 ];
 }

 COMPRESSED dest_opt_irregular {
 }
}
```

Options d'en-tête IPv6 bond par bond

```
ip_hop_opt
{
 UNCOMPRESSED {
  next_header [ 8 ];
  length      [ 8 ];
  value       [ length.UVALUE * 64 + 48 ];
 }

 DEFAULT {
```

```
  length         =:= static;
  next_header =:= static;
  value          =:= static;
 }

 COMPRESSED hop_opt_static {
  next_header =:= irregular(8) [ 8 ];
  length         =:= irregular(8) [ 8 ];
 }

 COMPRESSED hop_opt_dynamic {
  value =:= irregular(length.UVALUE*64+48) [ length.UVALUE * 64 + 48 ];
 }

 COMPRESSED hop_opt_irregular {
 }
}
```

En-tête d'acheminement IPv6

```
ip_rout_opt
{
 UNCOMPRESSED {
  next_header [ 8 ];
  length         [ 8 ];
  value          [ length.UVALUE * 64 + 48 ];
 }

 DEFAULT {
  length         =:= static;
  next_ header =:= static;
  value          =:= static;
 }

 COMPRESSED rout_opt_static {
  next_ header =:= irregular(8)               [ 8 ];
  length          =:= irregular(8)               [ 8 ];
  value           =:= irregular(length.UVALUE*64+48) [ length.UVALUE * 64 + 48 ];
 }

 COMPRESSED rout_opt_dynamic {
 }

 COMPRESSED rout_opt_irregular {
 }
}
```

En-tête GRE

```
optional_lsb_7_or_31(flag)
{

 UNCOMPRESSED {
  item [ 0, 32 ];
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  item =:= lsb_7_ou_31 [ 8, 32 ];
 }

 COMPRESSED not_present {
```

```
    ENFORCE(flag == 0);
    item =:= compressed_value(0, 0) [ 0 ];
   }
  }

  optional_checksum(flag_value)
  {
   UNCOMPRESSED {
    value       [ 0, 16 ];
    reserved1 [ 0, 16 ];
   }

   COMPRESSED cs_present {
    ENFORCE(flag_value == 1);
    value       =:= irregular(16)                    [ 16 ];
    reserved1 =:= uncompressed_value(16, 0) [  0 ];
   }

   COMPRESSED not_present {
    ENFORCE(flag_value == 0);
    value       =:= compressed_value(0, 0) [ 0 ];
    reserved1 =:= compressed_value(0, 0) [ 0 ];
   }
  }

  gre_proto
  {
   UNCOMPRESSED {
    protocol [ 16 ];
   }

   COMPRESSED ether_v4 {
    discriminator =:= '0'                                    [ 1 ];
    protocol      =:= uncompressed_value(16, 0x0800) [ 0 ];
   }

   COMPRESSED ether_v6 {
    discriminator =:= '1'                                   [ 1 ];
    protocol   =:= uncompressed_value(16, 0x86DD) [ 0 ];
   }
  }

  gre
  {
   UNCOMPRESSED {
    c_flag                                          [  1 ];
    r_flag =:= uncompressed_value(1, 0)     [  1 ];
    k_flag                                          [  1 ];
    s_flag                                          [  1 ];
    reserved0 =:= uncompressed_value(9, 0) [  9 ];
    version =:= uncompressed_value(3, 0)    [  3 ];
    protocol                                        [ 16 ];
    checksum_and_res                          [ 0, 32 ];
    key                                             [ 0, 32 ];
    sequence_number                           [ 0, 32 ];
   }

   DEFAULT {
    c_flag          =:= static;
    k_flag          =:= static;
    s_flag          =:= static;
    protocol        =:= static;
```

```
  key                    =:= static;
  sequence_number  =:= static;
 }

 COMPRESSED gre_static {
  ENFORCE((c_flag.UVALUE == 1 && checksum_and_res.ULENGTH == 32) || checksum_and_res.ULENGTH == 0);
  ENFORCE((s_flag.UVALUE == 1 && sequence_number.ULENGTH == 32) || sequence_number.ULENGTH == 0);
  protocol =:= gre_proto                          [ 1 ];
  c_flag     =:= irregular(1)                     [ 1 ];
  k_flag     =:= irregular(1)                     [ 1 ];
  s_flag     =:= irregular(1)                     [ 1 ];
  padding   =:= compressed_value(4, 0)            [ 4 ];
  key          =:= optional_32(k_flag.UVALUE) [ 0, 32 ];
 }

 COMPRESSED gre_dynamic {
  checksum_and_res =:= optional_checksum(c_flag.UVALUE)   [ 0, 16 ];
  sequence_number  =:= optional_32(s_flag.UVALUE)         [ 0, 32 ];
 }

 COMPRESSED gre_irregular {
  checksum_and_res =:= optional_checksum(c_flag.UVALUE)     [ 0, 16 ];
  sequence_number  =:= optional_lsb_7_or_31(s_flag.UVALUE)  [ 0, 8, 32 ];
 }

}
```

En-tête MINE

```
mine
{
 UNCOMPRESSED {
  next_ header [  8 ];
  s_bit       [  1 ];
  res_bits    [  7 ];
  checksum    [ 16 ];
  orig_dest   [ 32 ];
  orig_src    [ 0, 32 ];
 }

 DEFAULT {
  next_header =:= static;
  s_bit       =:= static;
  res_bits    =:= static;
  checksum    =:= inferred_mine_header_checksum;
  orig_dest   =:= static;
  orig_src    =:= static;
 }

 COMPRESSED mine_static {
  next_header =:= irregular(8)            [  8 ];
  s_bit       =:= irregular(1)            [  1 ];
// Les bits réservés sont inclus pour réaliser l'alignement sur l'octet
  res_bits    =:= irregular(7)            [  7 ];
  orig_dest   =:= irregular(32)           [ 32 ];
  orig_src  =:= optional_32(s_bit.UVALUE) [ 0, 32 ];
 }

 COMPRESSED mine_dynamic {
 }

 COMPRESSED mine_irregular {
```

```
  }
}
```

En-tête Authentification (AH)

```
ah
{
 UNCOMPRESSED {
  next_header                                  [  8 ];
  length                                       [  8 ];
  res_bits =:= uncompressed_value(16, 0) [ 16 ];
  spi                                          [ 32 ];
  sequence_number                              [ 32 ];
  icv                                          [ length.UVALUE*32-32 ];
 }

 DEFAULT {
  next_header         =:= static;
  length              =:= static;
  spi                 =:= static;
  sequence_number =:= static;
 }

 COMPRESSED ah_static {
  next_header =:= irregular(8)      [  8 ];
  length          =:= irregular(8)      [  8 ];
  spi             =:= irregular(32)    [ 32 ];
 }

 COMPRESSED ah_dynamic {
  sequence_number =:= irregular(32)            [ 32 ];
  icv =:= irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
 }

 COMPRESSED ah_irregular {
  sequence_number =:= lsb_7_ou_31   [ 8, 32 ];
  icv =:= irregular(length.UVALUE*32-32)  [ length.UVALUE*32-32 ];
 }
}
```

En-tête IPv6

```
fl_enc
{
 UNCOMPRESSED {
  flow_label [ 20 ];
 }

 COMPRESSED fl_zero {
  discriminator =:= '0'                                  [ 1 ];
  flow_label      =:= uncompressed_value(20, 0) [ 0 ];
  reserved         =:= '0000'                             [ 4 ];
 }

 COMPRESSED fl_non_zero {
  discriminator =:= '1'             [  1 ];
  flow_label  =:= irregular(20) [ 20 ];
 }
}

ipv6(profile_value, is_innermost, outer_ip_flag, reorder_ratio_value)
{
```

```
UNCOMPRESSED {
  version =:= uncompressed_value(4, 6) [   4 ];
  tos_tc                               [   8 ];
  flow_label                           [ 20 ];
  payload_length                       [ 16 ];
  next_header                          [   8 ];
  ttl_hopl                             [   8 ];
  src_addr                             [ 128 ];
  dst_addr                             [ 128 ];
}

CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(innermost_ip.UVALUE == is_innermost);
  innermost_ip [ 1 ];
}

DEFAULT {
  tos_tc         =:= static;
  flow_label     =:= static;
  payload_length =:= inferred_ip_v6_length;
  next_header    =:= static;
  ttl_hopl       =:= static;
  src_addr       =:= static;
  dst_addr       =:= static;
}

COMPRESSED ipv6_static {
  version_flag    =:= '1'                 [   1 ];
  innermost_ip    =:= irregular(1)        [   1 ];
  reserved        =:= '0'                 [   1 ];
  flow_label      =:= fl_enc              [ 5, 21 ];
  next_header     =:= irregular(8)        [   8 ];
  src_addr        =:= irregular(128)      [ 128 ];
  dst_addr        =:= irregular(128)      [ 128 ];
}

COMPRESSED ipv6_endpoint_dynamic {
  ENFORCE((is_innermost == 1) &&
          (profile_value == PROFILE_IP_0104));
  tos_tc          =:= irregular(8)                [  8 ];
  ttl_hopl        =:= irregular(8)                [  8 ];
  reserved        =:= compressed_value(6, 0)      [  6 ];
  reorder_ratio   =:= irregular(2)                [  2 ];
  msn             =:= irregular(16)               [ 16 ];
}

COMPRESSED ipv6_regular_dynamic {
  ENFORCE((is_innermost == 0) || (profile_value != PROFILE_IP_0104));
  tos_tc       =:= irregular(8) [ 8 ];
  ttl_hopl     =:= irregular(8) [ 8 ];
}

COMPRESSED ipv6_outer_irregular {
  ENFORCE(is_innermost == 0);
  tos_tc       =:= static_or_irreg(outer_ip_flag, 8) [ 0, 8 ];
  ttl_hopl     =:= static_or_irreg(outer_ip_flag, 8) [ 0, 8 ];
}

COMPRESSED ipv6_innermost_irregular {
  ENFORCE(is_innermost == 1);
```

```
  }

}
```

En-tête IPv4

```
ip_id_enc_dyn(behavior)
{
 UNCOMPRESSED {
  ip_id [ 16 ];
 }

 COMPRESSED ip_id_seq {
  ENFORCE((behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
  ip_id =:= irregular(16) [ 16 ];
 }

 COMPRESSED ip_id_random {
  ENFORCE(behavior == IP_ID_BEHAVIOR_RANDOM);
  ip_id =:= irregular(16) [ 16 ];
 }

 COMPRESSED ip_id_zero {
  ENFORCE(behavior == IP_ID_BEHAVIOR_ZERO);
  ip_id =:= uncompressed_value(16, 0) [ 0 ];
 }
}

ip_id_enc_irreg(behavior)
{
 UNCOMPRESSED {
  ip_id [ 16 ];
 }

 COMPRESSED ip_id_seq {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL);
 }

 COMPRESSED ip_id_seq_swapped {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED);
 }

 COMPRESSED ip_id_rand {
  ENFORCE(behavior == IP_ID_BEHAVIOR_RANDOM);
  ip_id =:= irregular(16) [ 16 ];
 }

 COMPRESSED ip_id_zero {
  ENFORCE(behavior == IP_ID_BEHAVIOR_ZERO);
  ip_id =:= uncompressed_value(16, 0) [ 0 ];
 }
}

ipv4(profile_value, is_innermost, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
{
 UNCOMPRESSED {
  version      =:= uncompressed_value(4, 4)  [  4 ];
  hdr_length   =:= uncompressed_value(4, 5)  [  4 ];
  tos_tc                                     [  8 ];
  length       =:= inferred_ip_v4_length     [ 16 ];
```

```
  ip_id                                              [ 16 ];
  rf              =:= uncompressed_value(1, 0)       [  1 ];
  df                                                 [  1 ];
  mf              =:= uncompressed_value(1, 0)       [  1 ];
  frag_offset =:= uncompressed_value(13, 0)          [ 13 ];
  ttl_hopl                                           [  8 ];
  protocol                                           [  8 ];
  checksum =:= inferred_ip_v4_header_checksum [ 16 ];
  src_addr                                           [ 32 ];
  dst_addr                                           [ 32 ];
}

CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(innermost_ip.UVALUE == is_innermost);
  ip_id_behavior_outer [ 2 ];
  innermost_ip              [ 1 ];
}

DEFAULT {
  tos_tc                    =:= static;
  df                        =:= static;
  ttl_hopl                  =:= static;
  protocol                  =:= static;
  src_addr                  =:= static;
  dst_addr                  =:= static;
  ip_id_behavior_outer =:= static;
}

COMPRESSED ipv4_static {
  version_flag    =:= '0'                 [  1 ];
  innermost_ip    =:= irregular(1)        [  1 ];
  reserved         =:= '000000'           [  6 ];
  protocol         =:= irregular(8)       [  8 ];
  src_addr         =:= irregular(32)      [ 32 ];
  dst_addr         =:= irregular(32)      [ 32 ];
}

COMPRESSED ipv4_endpoint_innermost_dynamic {
  ENFORCE((is_innermost == 1) && (profile_value == PROFILE_IP_0104));
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  reserved        =:= '000'                          [  3 ];
  reorder_ratio  =:= irregular(2)                    [  2 ];
  df               =:= irregular(1)                  [  1 ];
  ip_id_behavior_innermost =:= irregular(2)      [  2 ];
  tos_tc          =:= irregular(8)                   [  8 ];
  ttl_hopl         =:= irregular(8)                  [  8 ];
  ip_id            =:= ip_id_enc_dyn(ip_id_behavior_innermost.UVALUE) [ 0, 16 ];
  msn              =:= irregular(16)                  [ 16 ];
}

COMPRESSED ipv4_regular_innermost_dynamic {
  ENFORCE((is_innermost == 1) && (profile_value != PROFILE_IP_0104));
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  reserved        =:= '00000'                        [ 5 ];
  df               =:= irregular(1)                  [ 1 ];
  ip_id_behavior_innermost =:= irregular(2)      [ 2 ];
  tos_tc          =:= irregular(8)                   [ 8 ];
  ttl_hopl         =:= irregular(8)                  [ 8 ];
  ip_id            =:= ip_id_enc_dyn(ip_id_behavior_innermost.UVALUE) [ 0, 16 ];
}
```

```
 COMPRESSED ipv4_outer_dynamic {
  ENFORCE(est_innermost == 0);
  ENFORCE(ip_id_behavior_outer.UVALUE == ip_id_behavior_value);
  reserved                =:= '00000'                    [ 5 ];
  df                      =:= irregular(1)               [ 1 ];
  ip_id_behavior_outer =:= irregular(2)               [ 2 ];
  tos_tc                  =:= irregular(8)               [ 8 ];
  ttl_hopl                =:= irregular(8)               [ 8 ];
  ip_id                   =:= ip_id_enc_dyn(ip_id_behavior_outer.UVALUE)   [ 0, 16 ];
 }

 COMPRESSED ipv4_outer_irregular {
  ENFORCE(is_innermost == 0);
  ip_id    =:= ip_id_enc_irreg(ip_id_behavior_outer.UVALUE)   [ 0, 16 ];
  tos_tc   =:= static_or_irreg(outer_ip_flag, 8)                          [  0, 8 ];
  ttl_hopl =:= static_or_irreg(outer_ip_flag, 8)                          [  0, 8 ];
 }

 COMPRESSED ipv4_innermost_irregular {
  ENFORCE(is_innermost == 1);
  ip_id =:= ip_id_enc_irreg(ip_id_behavior_innermost.UVALUE)  [ 0, 16 ];
 }

}
```

En-tête UDP

```
udp(profile_value, reorder_ratio_value)
{
 UNCOMPRESSED {
  ENFORCE((profile_value == PROFILE_RTP_0101) ||
        (profile_value == PROFILE_UDP_0102));
  src_port                                  [ 16 ];
  dst_port                                  [ 16 ];
  udp_length =:= inferred_udp_length [ 16 ];
  checksum                                   [ 16 ];
 }

 CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  checksum_used [ 1 ];
 }

 DEFAULT {
  src_port          =:= static;
  dst_port          =:= static;
  checksum_used =:= static;
 }

 COMPRESSED udp_static {
  src_port   =:= irregular(16) [ 16 ];
  dst_port   =:= irregular(16) [ 16 ];
 }

 COMPRESSED udp_endpoint_dynamic {
  ENFORCE(profile_value == PROFILE_UDP_0102);
  ENFORCE(profile == PROFILE_UDP_0102);
  ENFORCE(checksum_used.UVALUE == (checksum.UVALUE != 0));
  checksum     =:= irregular(16)         [ 16 ];
  msn          =:= irregular(16)         [ 16 ];
```

```
  reserved       =:= compressed_value(6, 0) [  6 ];
  reorder_ratio =:= irregular(2)                [  2 ];
 }

 COMPRESSED udp_regular_dynamic {
  ENFORCE(profile_value == PROFILE_RTP_0101);
  ENFORCE(checksum_used.UVALUE == (checksum.UVALUE != 0));
  checksum =:= irregular(16) [ 16 ];
 }

 COMPRESSED udp_zero_checksum_irregular {
  ENFORCE(checksum_used.UVALUE == 0);
  checksum =:= uncompressed_value(16, 0)   [ 0 ];
 }

 COMPRESSED udp_with_checksum_irregular {
  ENFORCE(checksum_used.UVALUE == 1);
  checksum =:= irregular(16) [ 16 ];
 }

}
```

En-tête RTP

```
csrc_list_dynchain(presence, cc_value)
{
 UNCOMPRESSED {
  csrc_list;
 }

 COMPRESSED no_list {
  ENFORCE(cc_value == 0);
  ENFORCE(presence == 0);
  csrc_list =:= uncompressed_value(0, 0) [ 0 ];
 }

 COMPRESSED list_present {
  ENFORCE(presence == 1);
  csrc_list =:= list_csrc(cc_value) [ VARIABLE ];
 }
}

rtp(profile_value, ts_stride_value, time_stride_value, reorder_ratio_value)
{
 UNCOMPRESSED {
  ENFORCE((profile_value == PROFILE_RTP_0101) || (profile_value == PROFILE_RTP_0107));
  rtp_version =:= uncompressed_value(2, 2) [  2 ];
  pad_bit                                  [  1 ];
  extension                                [  1 ];
  cc                                       [  4 ];
  marker                                   [  1 ];
  payload_type                             [  7 ];
  sequence_number                          [ 16 ];
  timestamp                                [ 32 ];
  ssrc                                     [ 32 ];
  csrc_list                  [ cc.UVALUE * 32 ];
 }

 CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(time_stride_value == time_stride.UVALUE);
```

```
  ENFORCE(ts_stride_value == ts_stride.UVALUE);
  dummy_field =:= field_scaling(ts_stride.UVALUE,
    ts_scaled.UVALUE, timestamp.UVALUE, ts_offset.UVALUE) [ 0 ];
 }

 INITIAL {
  ts_stride      =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
  time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
 }

 DEFAULT {
  ENFORCE(msn.UVALUE == sequence_number.UVALUE);
  pad_bit              =:= static;
  extension            =:= static;
  cc                   =:= static;
  marker               =:= static;
  payload_type         =:= static;
  sequence_number =:= static;
  timestamp           =:= static;
  ssrc                 =:= static;
  csrc_list            =:= static;
  ts_stride            =:= static;
  time_stride          =:= static;
  ts_scaled            =:= static;
  ts_offset            =:= static;
 }

 COMPRESSED rtp_static {
  ssrc          =:= irregular(32)  [ 32 ];
 }

 COMPRESSED rtp_dynamic {
  reserved             =:= compressed_value(1, 0)  [  1 ];
  reorder_ratio        =:= irregular(2)                [  2 ];
  list_present         =:= irregular(1)               [  1 ];
  tss_indicator        =:= irregular(1)               [  1 ];
  tis_indicator        =:= irregular(1)                [  1 ];
  pad_bit              =:= irregular(1)                [  1 ];
  extension            =:= irregular(1)                [  1 ];
  marker               =:= irregular(1)                [  1 ];
  payload_type         =:= irregular(7)                [  7 ];
  sequence_number =:= irregular(16)                [ 16 ];
  timestamp            =:= irregular(32)               [ 32 ];
  ts_stride            =:= sdvl_or_default(tss_indicator.CVALUE, TS_STRIDE_DEFAULT)  [ VARIABLE ];
  time_stride     =:= sdvl_or_default(tis_indicator.CVALUE, TIME_STRIDE_DEFAULT)    [ VARIABLE ];
  csrc_list =:= csrc_list_dynchain(list_present.CVALUE, cc.UVALUE)                  [ VARIABLE ];
 }

 COMPRESSED rtp_irregular {
 }
}
```

En-tête UDP-Léger

```
checksum_coverage_dynchain(behavior)
{
 UNCOMPRESSED {
  checksum_coverage [ 16 ];
 }

 COMPRESSED inferred_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_INFERRED);
```

```
  checksum_coverage =:= inferred_udp_length [  0 ];
 }

 COMPRESSED static_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_STATIC);
  checksum_coverage =:= irregular(16)       [ 16 ];
 }

 COMPRESSED irregular_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_IRREGULAR);
  checksum_coverage =:= irregular(16)       [ 16 ];
 }
}

checksum_coverage_irregular(behavior)
{
 UNCOMPRESSED {
  checksum_coverage [ 16 ];
 }

 COMPRESSED inferred_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_INFERRED);
  checksum_coverage =:= inferred_udp_length [  0 ];
 }

 COMPRESSED static_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_STATIC);
  checksum_coverage =:= static              [  0 ];
 }

 COMPRESSED irregular_coverage {
  ENFORCE(behavior == UDP_LITE_COVERAGE_IRREGULAR);
  checksum_coverage =:= irregular(16)       [ 16 ];
 }
}

udp_lite(profile_value, reorder_ratio_value, coverage_behavior_value)
{
 UNCOMPRESSED {
  ENFORCE((profile_value == PROFILE_RTP_0107) || (profile_value == PROFILE_UDPLITE_0108));
  src_port                 [ 16 ];
  dst_port                 [ 16 ];
  checksum_coverage [ 16 ];
  checksum                 [ 16 ];
 }

 CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
 }

 DEFAULT {
  src_port                =:= static;
  dst_port                =:= static;
  coverage_behavior =:= static;
 }

 COMPRESSED udp_lite_static {
  src_port   =:= irregular(16) [ 16 ];
  dst_port   =:= irregular(16) [ 16 ];
 }
```

```
 COMPRESSED udp_lite_endpoint_dynamic {
  ENFORCE(profile_value == PROFILE_UDPLITE_0108);
  reserved              =:= compressed_value(4, 0)   [  4 ];
  coverage_behavior  =:= irregular(2)                [  2 ];
  reorder_ratio         =:= irregular(2)             [  2 ];
  checksum_coverage =:= checksum_coverage_dynchain(coverage_behavior.UVALUE)   [0, 16 ];
  checksum              =:= irregular(16)            [ 16 ];
  msn                   =:= irregular(16)            [ 16 ];
 }

 COMPRESSED udp_lite_regular_dynamic {
  ENFORCE(profile_value == PROFILE_RTP_0107);
  coverage_behavior =:= irregular(2)                 [  2 ];
  reserved =:= compressed_value(6, 0)                [  6 ];
  checksum_coverage =:= checksum_coverage_dynchain(coverage_behavior.UVALUE) [ 0, 16 ];
  checksum =:= irregular(16)                         [ 16 ];
 }

 COMPRESSED udp_lite_irregular {
  checksum_coverage =:= checksum_coverage_irregular(coverage_behavior.UVALUE) [ 0, 16 ];
  checksum              =:= irregular(16)            [ 16 ];
 }
}
```

En-tête ESP

```
esp(profile_value, reorder_ratio_value)
{
 UNCOMPRESSED {
  ENFORCE(profile_value == PROFILE_ESP_0103);
  ENFORCE(msn.UVALUE == sequence_number.UVALUE % 65536);
  spi                   [ 32 ];
  sequence_number [ 32 ];
 }

 CONTROL {
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
 }

 DEFAULT {
  spi                   =:= static;
  sequence_number =:= static;
 }

 COMPRESSED esp_static {
  spi =:= irregular(32)                  [ 32 ];
 }

 COMPRESSED esp_dynamic {
  sequence_number =:= irregular(32)              [ 32 ];
  reserved              =:= compressed_value(6, 0)   [  6 ];
  reorder_ratio         =:= irregular(2)             [  2 ];
 }

 COMPRESSED esp_irregular {
 }
}
```

Méthodes de codage utilisées dans les en-têtes CO des profils

```
// Décalage de réarrangement variable utilisé pour MSN
msn_lsb(k)
{
 UNCOMPRESSED {
  master [ VARIABLE ];
 }

 COMPRESSED none {
  ENFORCE(reorder_ratio.UVALUE == REORDERING_NONE);
  master =:= lsb(k, 1);
 }

 COMPRESSED quarter {
  ENFORCE(reorder_ratio.UVALUE == REORDERING_QUARTER);
  master =:= lsb(k, ((2^k) / 4) - 1);
 }

 COMPRESSED half {
  ENFORCE(reorder_ratio.UVALUE == REORDERING_HALF);
  master =:= lsb(k, ((2^k) / 2) - 1);
 }

 COMPRESSED threequarters {
  ENFORCE(reorder_ratio.UVALUE == REORDERING_THREEQUARTERS);
  master =:= lsb(k, (((2^k) * 3) / 4) - 1);
 }
}

ip_id_lsb(behavior, k)
{
 UNCOMPRESSED {
  ip_id [ 16 ];
 }

 CONTROL {
  ip_id_nbo    [ 16 ];
 }

 COMPRESSED nbo {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL);
  ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
  ip_id_offset =:= lsb(k, ((2^k) / 4) - 1) [ k ];
 }

 COMPRESSED non_nbo {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED);
  ENFORCE(ip_id_nbo.UVALUE == (ip_id.UVALUE / 256) + (ip_id.UVALUE % 256) * 256);
  ENFORCE(ip_id_nbo.ULENGTH == 16);
  ENFORCE(ip_id_offset.UVALUE == ip_id_nbo.UVALUE - msn.UVALUE);
  ip_id_offset =:= lsb(k, ((2^k) / 4) - 1) [ k ];
 }
}

ip_id_sequential_variable(behavior, indicator)
{
 UNCOMPRESSED {
  ip_id [ 16 ];
 }

 COMPRESSED short {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
```

```
  ENFORCE(indicator == 0);
  ip_id =:= ip_id_lsb(behavior, 8) [ 8 ];
 }

 COMPRESSED long {
  ENFORCE(behavior == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (behavior == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  ENFORCE(indicator == 1);
  ENFORCE(ip_id_offset.UVALUE == ip_id.UVALUE - msn.UVALUE);
  ip_id =:= irregular(16)  [ 16 ];
 }

 COMPRESSED not_present {
  ENFORCE(behavior == IP_ID_BEHAVIOR_RANDOM) || (behavior == IP_ID_BEHAVIOR_ZERO));
 }
}

dont_fragment(version)
{
 UNCOMPRESSED {
  df [ 0, 1 ];
 }

 COMPRESSED v4 {
  ENFORCE(version == 4);
  df =:= irregular(1) [ 1 ];
 }

 COMPRESSED v6 {
  ENFORCE(version == 6);
  unused =:= compressed_value(1, 0) [ 1 ];
 }
}

pt_irr_ou_static(flag)
{
 UNCOMPRESSED {
  payload_type [ 7 ];
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  payload_type =:= static [ 0 ];
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  reserved     =:= compressed_value(1, 0) [ 1 ];
  payload_type =:= irregular(7)           [ 7 ];
 }
}

csrc_list_presence(presence, cc_value)
{
 UNCOMPRESSED {
  csrc_list;
 }

 COMPRESSED no_list {
  ENFORCE(presence == 0);
  csrc_list =:= static [ 0 ];
 }
```

```
  COMPRESSED list_present {
    ENFORCE(presence == 1);
    csrc_list =:= list_csrc(cc_value) [ VARIABLE ];
  }
}

scaled_ts_lsb(time_stride_value, k)
{
  UNCOMPRESSED {
    timestamp [ 32 ];
  }

  COMPRESSED timerbased {
    ENFORCE(time_stride_value != 0);
    timestamp =:= timer_based_lsb(time_stride_value, k, ((2^k) / 2) - 1);
  }

  COMPRESSED regular {
    ENFORCE(time_stride_value == 0);
    timestamp =:= lsb(k, ((2^k) / 4) - 1);
  }
}

// Codage auto descriptif de longueur variable avec décalage de réarrangement
sdvl_sn_lsb(field_width)
{
  UNCOMPRESSED {
    field [ field_width ];
  }

  COMPRESSED lsb7 {
    discriminator =:= '0'   [ 1 ];
    field =:= msn_lsb(7)    [ 7 ];
  }

  COMPRESSED lsb14 {
    discriminator =:= '10'    [  2 ];
    field =:= msn_lsb(14)   [ 14 ];
  }

  COMPRESSED lsb21 {
    discriminator =:= '110'   [  3 ];
    field =:= msn_lsb(21)    [ 21 ];
  }

  COMPRESSED lsb28 {
    discriminator =:= '1110' [  4 ];
    field =:= msn_lsb(28)    [ 28 ];
  }

  COMPRESSED lsb32 {
    discriminator =:= '11111111'        [  8 ];
    field =:= irregular(field_width)    [ field_width ];
  }
}

//  Codage auto descriptif de longueur variable
sdvl_lsb(field_width)
{
  UNCOMPRESSED {
    field [ field_width ];
```

```
  }

  COMPRESSED lsb7 {
    discriminator =:= '0'                [ 1 ];
    field =:= lsb(7, ((2^7) / 4) - 1)    [ 7 ];
  }

  COMPRESSED lsb14 {
    discriminator =:= '10'               [  2 ];
    field =:= lsb(14, ((2^14) / 4) - 1) [ 14 ];
  }

  COMPRESSED lsb21 {
    discriminator =:= '110'              [  3 ];
    field =:= lsb(21, ((2^21) / 4) - 1) [ 21 ];
  }

  COMPRESSED lsb28 {
    discriminator =:= '1110'             [  4 ];
    field =:= lsb(28, ((2^28) / 4) - 1) [ 28 ];
  }

  COMPRESSED lsb32 {
    discriminator =:= '11111111'         [  8 ];
    field =:= irregular(field_width)     [ field_width ];
  }
}

sdvl_scaled_ts_lsb(time_stride)
{
  UNCOMPRESSED {
    field [ 32 ];
  }

  COMPRESSED lsb7 {
    discriminator =:= '0'                          [  1 ];
    field =:= scaled_ts_lsb(time_stride, 7)        [  7 ];
  }

  COMPRESSED lsb14 {
    discriminator =:= '10'                         [  2 ];
    field =:= scaled_ts_lsb(time_stride, 14)       [ 14 ];
  }

  COMPRESSED lsb21 {
    discriminator =:= '110'                        [  3 ];
    field =:= scaled_ts_lsb(time_stride, 21)       [ 21 ];
  }

  COMPRESSED lsb28 {
    discriminator =:= '1110'                       [  4 ];
    field =:= scaled_ts_lsb(time_stride, 28)       [ 28 ];
  }

  COMPRESSED lsb32 {
    discriminator =:= '11111111'                   [  8 ];
    field =:= irregular(32)                        [ 32 ];
  }
}

variable_scaled_timestamp(tss_flag, tsc_flag, ts_stride, time_stride)
{
```

```
 UNCOMPRESSED {
  scaled_value [ 32 ];
 }

 COMPRESSED present {
  ENFORCE((tss_flag == 0) && (tsc_flag == 1));
  ENFORCE(ts_stride != 0);
  scaled_value =:= sdvl_scaled_ts_lsb(time_stride) [ VARIABLE ];
 }

 COMPRESSED not_present {
  ENFORCE(((tss_flag == 1) && (tsc_flag == 0)) || ((tss_flag == 0) && (tsc_flag == 0)));
 }
}

variable_unscaled_timestamp(tss_flag, tsc_flag)
{
 UNCOMPRESSED {
  timestamp [ 32 ];
 }

 COMPRESSED present {
  ENFORCE(((tss_flag == 1) && (tsc_flag == 0)) || ((tss_flag == 0) && (tsc_flag == 0)));
  timestamp =:= sdvl_lsb(32);
 }

 COMPRESSED not_present {
  ENFORCE((tss_flag == 0) && (tsc_flag == 1));
 }
}

profile_1_7_flags1_enc(flag, ip_version)
{
 UNCOMPRESSED {
  ip_outer_indicator  [ 1 ];
  ttl_hopl_indicator  [ 1 ];
  tos_tc_indicator    [ 1 ];
  df                  [ 0, 1 ];
  ip_id_behavior      [ 2 ];
  reorder_ratio       [ 2 ];
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  ENFORCE(ip_outer_indicator.CVALUE == 0);
  ENFORCE(ttl_hopl_indicator.CVALUE == 0);
  ENFORCE(tos_tc_indicator.CVALUE == 0);
  df                  =:= static;
  ip_id_behavior      =:= static;
  reorder_ratio       =:= static;
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  ip_outer_indicator =:= irregular(1)                  [ 1 ];
  ttl_hopl_indicator =:= irregular(1)                  [ 1 ];
  tos_tc_indicator   =:= irregular(1)                  [ 1 ];
  df                 =:= dont_fragment(ip_version)     [ 1 ];
  ip_id_behavior     =:= irregular(2)                  [ 2 ];
  reorder_ratio      =:= irregular(2)                  [ 2 ];
 }
}
```

```
profile_1_flags2_enc(flag)
{
 UNCOMPRESSED {
  list_indicator            [ 1 ];
  pt_indicator              [ 1 ];
  time_stride_indicator [ 1 ];
  pad_bit                   [ 1 ];
  extension                 [ 1 ];
 }

 COMPRESSED not_present{
  ENFORCE(flag == 0);
  ENFORCE(list_indicator.UVALUE == 0);
  ENFORCE(pt_indicator.UVALUE == 0);
  ENFORCE(time_stride_indicator.UVALUE == 0);
  pad_bit     =:= static;
  extension   =:= static;
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  list_indicator =:= irregular(1)               [ 1 ];
  pt_indicator   =:= irregular(1)               [ 1 ];
  time_stride_indicator =:= irregular(1)        [ 1 ];
  pad_bit       =:= irregular(1)                [ 1 ];
  extension      =:= irregular(1)               [ 1 ];
  reserved       =:= compressed_value(3, 0)     [ 3 ];
 }
}

profile_2_3_4_flags_enc(flag, ip_version)
{
 UNCOMPRESSED {
  ip_outer_indicator [ 1 ];
  df                 [ 0, 1 ];
  ip_id_behavior     [ 2 ];
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  ENFORCE(ip_outer_indicator.CVALUE == 0);
  df                  =:= static;
  ip_id_behavior      =:= static;
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  ip_outer_indicator =:= irregular(1)                   [ 1 ];
  df                 =:= dont_fragment(ip_version)      [ 1 ];
  ip_id_behavior     =:= irregular(2)                   [ 2 ];
  reserved           =:= compressed_value(4, 0)         [ 4 ];
 }
}

profile_8_flags_enc(flag, ip_version)
{
 UNCOMPRESSED {
  ip_outer_indicator    [ 1 ];
  df                    [ 0, 1 ];
  ip_id_behavior        [ 2 ];
  coverage_behavior     [ 2 ];
```

```
 }

 COMPRESSED not_present {
  ENFORCE(flag == 0);
  ENFORCE(ip_outer_indicator.CVALUE == 0);
  df                  =:= static;
  ip_id_behavior      =:= static;
  coverage_behavior   =:= static;
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  reserved            =:= compressed_value(2, 0)      [ 2 ];
  ip_outer_indicator  =:= irregular(1)                [ 1 ];
  df                  =:= dont_fragment(ip_version)   [ 1 ];
  ip_id_behavior      =:= irregular(2)                [ 2 ];
  coverage_behavior   =:= irregular(2)                [ 2 ];
 }
}

profile_7_flags2_enc(flag)
{
 UNCOMPRESSED {
  list_indicator          [ 1 ];
  pt_indicator            [ 1 ];
  time_stride_indicator   [ 1 ];
  pad_bit                 [ 1 ];
  extension               [ 1 ];
  coverage_behavior       [ 2 ];
 }

 COMPRESSED not_present{
  ENFORCE(flag == 0);
  ENFORCE(list_indicator.CVALUE == 0);
  ENFORCE(pt_indicator.CVALUE == 0);
  ENFORCE(time_stride_indicator.CVALUE == 0);
  pad_bit             =:= static;
  extension           =:= static;
  coverage_behavior   =:= static;
 }

 COMPRESSED present {
  ENFORCE(flag == 1);
  reserved                =:= compressed_value(1, 0)   [ 1 ];
  list_indicator          =:= irregular(1)             [ 1 ];
  pt_indicator            =:= irregular(1)             [ 1 ];
  time_stride_indicator   =:= irregular(1)             [ 1 ];
  pad_bit                 =:= irregular(1)             [ 1 ];
  extension               =:= irregular(1)             [ 1 ];
  coverage_behavior       =:= irregular(2)             [ 2 ];
 }
}
```

Profil RTP

```
rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value,
reorder_ratio_value)
{
 UNCOMPRESSED v4 {
  ENFORCE(msn.UVALUE == sequence_number.UVALUE);
  outer_headers  =:= baseheader_outer_headers          [ VARIABLE ];
  ip_version     =:= uncompressed_value(4, 4)          [  4 ];
```

```
  header_length  =:= uncompressed_value(4, 5)          [  4 ];
  tos_tc                                               [  8 ];
  length             =:= inferred_ip_v4_length         [ 16 ];
  ip_id                                                [ 16 ];
  rf                 =:= uncompressed_value(1, 0)      [  1 ];
  df                                                   [  1 ];
  mf                 =:= uncompressed_value(1, 0)      [  1 ];
  frag_offset        =:= uncompressed_value(13, 0)     [ 13 ];
  ttl_hopl                                             [  8 ];
  next_ header                                         [  8 ];
  ip_checksum =:= inferred_ip_v4_header_checksum       [ 16 ];
  src_addr                                             [ 32 ];
  dest_addr                                            [ 32 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  src_port                                             [ 16 ];
  dst_port                                             [ 16 ];
  udp_length  =:= inferred_udp_length                  [ 16 ];
  udp_checksum                                         [ 16 ];
  rtp_version =:= uncompressed_value(2, 2)             [  2 ];
  pad_bit                                              [  1 ];
  extension                                            [  1 ];
  cc                                                   [  4 ];
  marker                                               [  1 ];
  payload_type                                         [  7 ];
  sequence_number                                      [ 16 ];
  horodatage                                           [ 32 ];
  ssrc                                                 [ 32 ];
  csrc_list                                            [ VARIABLE ];
}

UNCOMPRESSED v6 {
  ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
  ENFORCE(msn.UVALUE == sequence_number.UVALUE);
  outer_headers  =:= baseheader_outer_headers          [ VARIABLE ];
  ip_version     =:= uncompressed_value(4, 6)          [   4 ];
  tos_tc                                               [   8 ];
  flow_label                                           [  20 ];
  payload_length =:= inferred_ip_v6_length             [  16 ];
  next_header                                          [   8 ];
  ttl_hopl                                             [   8 ];
  src_add                                              [ 128 ];
  dest_addr                                            [ 128 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  src_port                                             [  16 ];
  dst_port                                             [  16 ];
  udp_length     =:= inferred_udp_length               [  16 ];
  udp_checksum                                         [  16 ];
  rtp_version    =:= uncompressed_value(2, 2)          [   2 ];
  pad_bit                                              [   1 ];
  extension                                            [   1 ];
  cc                                                   [   4 ];
  marker                                               [   1 ];
  payload_type                                         [   7 ];
  sequence_number                                      [  16 ];
  timestamp                                            [  32 ];
  ssrc                                                 [  32 ];
  csrc_list                                            [ VARIABLE ];
  df    =:= uncompressed_value(0,0)                    [   0 ];
  ip_id =:= uncompressed_value(0,0)                    [   0 ];
}

CONTROL {
```

```
  ENFORCE(profile_value == PROFILE_RTP_0101);
  ENFORCE(profile == profile_value);
  ENFORCE(time_stride.UVALUE == time_stride_value);
  ENFORCE(ts_stride.UVALUE == ts_stride_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  dummy_field =:= field_scaling(ts_stride.UVALUE,
    ts_scaled.UVALUE, timestamp.UVALUE, ts_offset.UVALUE) [ 0 ];
 }

 INITIAL {
  ts_stride      =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
  time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
 }

 DEFAULT {
  ENFORCE(outer_ip_flag == 0);
  tos_tc                          =:= static;
  dest_addr                       =:= static;
  ttl_hopl                        =:= static;
  src_addr                        =:= static;
  df                              =:= static;
  flow_label                      =:= static;
  next_header                     =:= static;
  src_port                        =:= static;
  dst_port                        =:= static;
  pad_bit                         =:= static;
  extension                       =:= static;
  cc                              =:= static;
// Quand le bit marqueur n'est pas présent dans les paquets, il est supposé être à 0.
  marker                          =:= uncompressed_value(1, 0);
  payload_type                     =:= static;
  sequence_number                 =:= static;
  timestamp                       =:= static;
  ssrc                            =:= static;
  csrc_list                       =:= static;
  ts_stride                       =:= static;
  time_stride                     =:= static;
  ts_scaled                       =:= static;
  ts_offset                       =:= static;
  reorder_ratio                   =:= static;
  ip_id_behavior_innermost        =:= static;
 }

// Remplacement pour UOR-2-ext3
 COMPRESSED co_common {
  ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
  discriminator      =:= '11111010'                              [ 8 ];
  marker             =:= irregular(1)                            [ 1 ];
  header_crc         =:= crc7(THIS.UVALUE, THIS.ULENGTH)  [ 7 ];
  flags1_indicator   =:= irregular(1)                            [ 1 ];
  flags2_indicator   =:= irregular(1)                            [ 1 ];
  tsc_indicator      =:= irregular(1)                            [ 1 ];
  tss_indicator      =:= irregular(1)                            [ 1 ];
  ip_id_indicator    =:= irregular(1)                            [ 1 ];
  control_crc3       =:= control_crc3_encoding                   [ 3 ];

 outer_ip_indicator : ttl_hopl_indicator : tos_tc_indicator : df : ip_id_behavior_innermost : reorder_ratio
    =:= profile_1_7_flags1_enc(flags1_indicator.CVALUE, ip_version.UVALUE)     [ 0, 8 ];
  list_indicator : pt_indicator : tis_indicator : pad_bit :
    extension =:= profile_1_flags2_enc(flags2_indicator.CVALUE)                [ 0, 8 ];
  tos_tc =:= static_ou_irreg(tos_tc_indicator.CVALUE, 8)                       [ 0, 8 ];
```

```
  ttl_hopl =:= static_ou_irreg(ttl_hopl_indicator.CVALUE, ttl_hopl.ULENGTH)    [ 0, 8 ];
  payload_type =:= pt_irr_ou_static(pt_indicator)                               [ 0, 8 ];
  sequence_number =:= sdvl_sn_lsb(sequence_number.ULENGTH)                      [ VARIABLE ];
  ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE, ip_id_indicator.CVALUE) [ 0, 8, 16 ];
  ts_scaled =:= variable_scaled_timestamp(tss_indicator.CVALUE,
    tsc_indicator.CVALUE, ts_stride.UVALUE, time_stride.UVALUE)                 [ VARIABLE ];
  timestamp =:= variable_unscaled_timestamp(tss_indicator.CVALUE, tsc_indicator.CVALUE)  [ VARIABLE ];
  ts_stride =:= sdvl_or_static(tss_indicator.CVALUE)                            [ VARIABLE ];
  time_stride =:= sdvl_or_static(tis_indicator.CVALUE)                          [ VARIABLE ];
  csrc_list =:= csrc_list_presence(list_indicator.CVALUE, cc.UVALUE)            [ VARIABLE ];
 }

// UO-0
 COMPRESSED pt_0_crc3 {
  discriminator =:= '0'                                          [ 1 ];
  msn           =:= msn_lsb(4)                                   [ 4 ];
  header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
  timestamp     =:= inferred_scaled_field                        [ 0 ];
  ip_id          =:= inferred_sequential_ip_id                   [ 0 ];
 }

/ Nouveau format, type 0 avec fort CRC et plus de bits de numéro de séquence
 COMPRESSED pt_0_crc7 {
  discriminator =:= '1000'                                       [ 4 ];
  msn           =:= msn_lsb(5)                                   [ 5 ];
  header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)              [ 7 ];
  timestamp     =:= inferred_scaled_field                        [ 0 ];
  ip_id         =:= inferred_sequential_ip_id                    [ 0 ];
 }

// Remplacement de UO-1
 COMPRESSED pt_1_rnd {
  ENFORCE(ts_stride.UVALUE != 0);
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
  discriminator =:= '101'                                        [ 3 ];
  marker        =:= irregular(1)                                 [ 1 ];
  msn           =:= msn_lsb(4)                                   [ 4 ];
  ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)         [ 5 ];
  header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
 }

// Remplacement de UO-1-ID
 COMPRESSED pt_1_seq_id {
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator  =:= '1001'                                      [ 4 ];
  ip_id          =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  msn            =:= msn_lsb(5)                                  [ 5 ];
  header_crc     =:= crc3(THIS.UVALUE, THIS.ULENGTH)             [ 3 ];
  timestamp      =:= inferred_scaled_field                       [ 0 ];
 }

// Remplacement de UO-1-TS
 COMPRESSED pt_1_seq_ts {
  ENFORCE(ts_stride.UVALUE != 0);
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator =:= '101'                                        [ 3 ];
  marker        =:= irregular(1)                                 [ 1 ];
  msn           =:= msn_lsb(4)                                   [ 4 ];
  ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)         [ 5 ];
```

```
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)  [ 3 ];
    ip_id         =:= inferred_sequential_ip_id        [ 0 ];
   }

  // Remplacement de UOR-2
   COMPRESSED pt_2_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM) ||
          (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '110'                                  [ 3 ];
    msn           =:= msn_lsb(7)                             [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 6)   [ 6 ];
    marker        =:= irregular(1)                           [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)        [ 7 ];
   }

  // Remplacement de UOR-2-ID
   COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
          (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11000'                                [ 5 ];
    msn           =:= msn_lsb(7)                             [ 7 ];
    ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)        [ 7 ];
    timestamp     =:= inferred_scaled_field                  [ 0 ];
   }

  // Remplacement de UOR-2-ID-ext1 (TS et IP-ID)
   COMPRESSED pt_2_seq_both {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
          (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '11001'                                [ 5 ];
    msn           =:= msn_lsb(7)                             [ 7 ];
    ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)        [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 7)   [ 7 ];
    marker        =:= irregular(1)                           [ 1 ];
   }

  // Remplacement de UOR-2-TS
   COMPRESSED pt_2_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
          (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '1101'                                 [ 4 ];
    msn           =:= msn_lsb(7)                             [ 7 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)   [ 5 ];
    marker        =:= irregular(1)                           [ 1 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)        [ 7 ];
    ip_id         =:= inferred_sequential_ip_id              [ 0 ];
   }
  }
```

Profil UDP

```
udp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
{
 UNCOMPRESSED v4 {
   outer_headers       =:= baseheader_outer_headers       [ VARIABLE ];
   ip_version          =:= uncompressed_value(4, 4)       [  4 ];
   header_length       =:= uncompressed_value(4, 5)       [  4 ];
```

```
  tos_tc                                                [  8 ];
  length              =:= inferred_ip_v4_length         [ 16 ];
  ip_id                                                 [ 16 ];
  rf                  =:= uncompressed_value(1, 0)      [  1 ];
  df                                                    [  1 ];
  mf                  =:= uncompressed_value(1, 0)      [  1 ];
  frag_offset         =:= uncompressed_value(13, 0)     [ 13 ];
  ttl_hopl                                              [  8 ];
  next_header                                           [  8 ];
  ip_checksum         =:= inferred_ip_v4_header_checksum[16 ];
  src_addr                                              [ 32 ];
  dest_addr                                             [ 32 ];
  extension_headers   =:= baseheader_extension_headers  [ VARIABLE ];
  src_port                                              [ 16 ];
  dst_port                                              [ 16 ];
  udp_length          =:= inferred_udp_length           [ 16 ];
  udp_checksum                                          [ 16 ];
}

UNCOMPRESSED v6 {
  ENFORCE(ip_id_behavior.UVALUE == IP_ID_BEHAVIOR_RANDOM);
  outer_headers       =:= baseheader_outer_headers      [ VARIABLE ];
  ip_version          =:= uncompressed_value(4, 6)      [  4 ];
  tos_tc                                                [  8 ];
  flow_label                                            [ 20 ];
  payload_length      =:= inferred_ip_v6_length         [ 16 ];
  next_header                                           [  8 ];
  ttl_hopl                                              [  8 ];
  src_addr                                              [ 128 ];
  dest_addr                                             [ 128 ];
  extension_headers   =:= baseheader_extension_headers [ VARIABLE ];
  src_port                                              [ 16 ];
  dst_port                                              [ 16 ];
  udp_length          =:= inferred_udp_length           [ 16 ];
  udp_checksum                                          [ 16 ];
  df                  =:= uncompressed_value(0,0)       [  0 ];
  ip_id               =:= uncompressed_value(0,0)       [  0 ];
}

CONTROL {
  ENFORCE(profile_value == PROFILE_UDP_0102);
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
}

DEFAULT {
  ENFORCE(outer_ip_flag == 0);
  tos_tc                      =:= static;
  dest_addr                   =:= static;
  ip_version                  =:= static;
  ttl_hopl                    =:= static;
  src_addr                    =:= static;
  df                          =:= static;
  flow_label                  =:= static;
  next_header                 =:= static;
  src_port                    =:= static;
  dst_port                    =:= static;
  reorder_ratio               =:= static;
  ip_id_behavior_innermost =:= static;
}
```

```
// Remplacement pour UOR-2-ext3
 COMPRESSED co_common {
  ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
  discriminator          =:= '11111010'                  [ 8 ];
  ip_id_indicator        =:= irregular(1)                [ 1 ];
  header_crc             =:= crc7(THIS.UVALUE, THIS.ULENGTH)    [ 7 ];
  flags_indicator        =:= irregular(1)                [ 1 ];
  ttl_hopl_indicator     =:= irregular(1)                [ 1 ];
  tos_tc_indicator       =:= irregular(1)                [ 1 ];
  reorder_ratio          =:= irregular(2)                [ 2 ];
  control_crc3           =:= control_crc3_encoding       [ 3 ];
  outer_ip_indicator : df : ip_id_behavior_innermost =:= profile_2_3_4_flags_enc(
   flags_indicator.CVALUE, ip_version.UVALUE)   [ 0, 8 ];
  tos_tc                 =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
  ttl_hopl               =:= static_or_irreg(ttl_hopl_indicator.CVALUE, ttl_hopl.ULENGTH)  [ 0, 8 ];
  msn                    =:= msn_lsb(8)                  [ 8 ];
  ip_id                  =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
    ip_id_indicator.CVALUE)                              [ 0, 8, 16 ];
 }

// UO-0
 COMPRESSED pt_0_crc3 {
  discriminator =:= '0'                                  [ 1 ];
  msn           =:= msn_lsb(4)                           [ 4 ];
  header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
  ip_id         =:= inferred_sequential_ip_id            [ 0 ];
 }

// Nouveau format, type 0 avec CRC fort et plus de bits SN
 COMPRESSED pt_0_crc7 {
  discriminator =:= '100'                                [ 3 ];
  msn           =:= msn_lsb(6)                           [ 6 ];
  header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
  ip_id         =:= inferred_sequential_ip_id            [ 0 ];
 }

// Remplacement de UO-1-ID (PT-1 seulement utilisé pour séquentiel)
 COMPRESSED pt_1_seq_id {
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator  =:= '101'                                 [ 3 ];
  header_crc     =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
  msn            =:= msn_lsb(6)                            [ 6 ];
  ip_id          =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
 }

// Remplacement de UOR-2-ID
 COMPRESSED pt_2_seq_id {
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator =:= '110'                                [ 3 ];
  ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
  header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)      [ 7 ];
  msn           =:= msn_lsb(8)                           [ 8 ];
 }
}
```

Profil ESP

```
esp_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
{
```

```
UNCOMPRESSED v4 {
 ENFORCE(msn.UVALUE == sequence_number.UVALUE % 65536);
 outer_headers       =:= baseheader_outer_headers      [ VARIABLE ];
 ip_version          =:= uncompressed_value(4, 4)      [  4 ];
 header_length       =:= uncompressed_value(4, 5)      [  4 ];
 tos_tc                                                [  8 ];
 length              =:= inferred_ip_v4_length         [ 16 ];
 ip_id                                                 [ 16 ];
 rf                  =:= uncompressed_value(1, 0)      [  1 ];
 df                                                    [  1 ];
 mf                  =:= uncompressed_value(1, 0)      [  1 ];
 frag_offset         =:= uncompressed_value(13, 0)     [ 13 ];
 ttl_hopl                                              [  8 ];
 next_header                                           [  8 ];
 ip_checksum         =:= inferred_ip_v4_header_checksum  [ 16 ];
 src_addr                                              [ 32 ];
 dest_addr                                             [ 32 ];
 extension_headers =:= baseheader_extension_headers [ VARIABLE ];
 spi                                                   [ 32 ];
 sequence_number                                       [ 32 ];
}

UNCOMPRESSED v6 {
 ENFORCE(msn.UVALUE == (sequence_number.UVALUE % 65536));
 ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
 outer_headers       =:= baseheader_outer_headers      [ VARIABLE ];
 ip_version          =:= uncompressed_value(4, 6)      [   4 ];
 tos_tc                                                [   8 ];
 flow_label                                            [  20 ];
 payload_length      =:= inferred_ip_v6_length         [  16 ];
 next_header                                           [   8 ];
 ttl_hopl                                              [   8 ];
 src_addr                                              [ 128 ];
 dest_addr                                             [ 128 ];
 extension_headers =:= baseheader_extension_headers [ VARIABLE ];
 spi                                                   [  32 ];
 sequence_number                                       [  32 ];
 df                  =:= uncompressed_value(0,0)       [   0 ];
 ip_id               =:= uncompressed_value(0,0)       [   0 ];
}

CONTROL {
 ENFORCE(profile_value == PROFILE_ESP_0103);
 ENFORCE(profile == profile_value);
 ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
 ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
}

DEFAULT {
 ENFORCE(outer_ip_flag == 0);
 tos_tc                      =:= static;
 dest_addr                   =:= static;
 ttl_hopl                    =:= static;
 src_addr                    =:= static;
 df                          =:= static;
 flow_label                  =:= static;
 next_header                 =:= static;
 spi                         =:= static;
 sequence_number             =:= static;
 reorder_ratio               =:= static;
 ip_id_behavior_innermost =:= static;
}
```

```
// Remplacement pour UOR-2-ext3
 COMPRESSED co_common {
  ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
  discriminator       =:= '11111010'                                  [ 8 ];
  ip_id_indicator     =:= irregular(1)                                [ 1 ];
  header_crc          =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
  flags_indicator     =:= irregular(1)                                [ 1 ];
  ttl_hopl_indicator =:= irregular(1)                                 [ 1 ];
  tos_tc_indicator    =:= irregular(1)                                [ 1 ];
  reorder_ratio       =:= irregular(2)                                [ 2 ];
  control_crc3        =:= control_crc3_encoding                       [ 3 ];

  outer_ip_indicator : df : ip_id_behavior_innermost =:= profile_2_3_4_flags_enc(
    flags_indicator.CVALUE, ip_version.UVALUE)          [ 0, 8 ];
  tos_tc =:= static_ou_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
  ttl_hopl =:= static_ou_irreg(ttl_hopl_indicator.CVALUE, ttl_hopl.ULENGTH)     [ 0, 8 ];
  sequence_number =:= sdvl_sn_lsb(sequence_number.ULENGTH)            [ VARIABLE ];
  ip_id =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
    ip_id_indicator.CVALUE)                    [ 0, 8, 16 ];
 }

// Numéro de séquence envoyé à la place du MSN à cause de la longueur du champ UO-0
 COMPRESSED pt_0_crc3 {
  discriminator         =:= '0'                                       [ 1 ];
  sequence_number   =:= msn_lsb(4)                                    [ 4 ];
  header_crc            =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
  ip_id                 =:= inferred_sequential_ip_id                 [ 0 ];
 }

// Nouveau format, type 0 avec CRC fort et plus de bits de SN
 COMPRESSED pt_0_crc7 {
  discriminator       =:= '100'                                       [ 3 ];
  sequence_number =:= msn_lsb(6)                                      [ 6 ];
  header_crc          =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
  ip_id               =:= inferred_sequential_ip_id                   [ 0 ];
 }

// Remplacement de UO-1-ID (PT-1 seulement utilisé pour séquentiel)
 COMPRESSED pt_1_seq_id {
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
       (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator       =:= '101'                                       [ 3 ];
  header_crc          =:= crc3(THIS.UVALUE, THIS.ULENGTH)             [ 3 ];
  sequence_number =:= msn_lsb(6)                                      [ 6 ];
  ip_id               =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
 }

// Remplacement de UOR-2-ID
 COMPRESSED pt_2_seq_id {
  ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
       (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
  discriminator       =:= '110'                                       [ 3 ];
  ip_id               =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
  header_crc          =:= crc7(THIS.UVALUE, THIS.ULENGTH)             [ 7 ];
  sequence_number  =:= msn_lsb(8)                                     [ 8 ];
 }
}
```

Profil IP seul

```
iponly_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value)
```

```
{
 UNCOMPRESSED v4 {
  outer_headers      =:= baseheader_outer_headers      [ VARIABLE ];
  ip_version         =:= uncompressed_value(4, 4)      [  4 ];
  header_length      =:= uncompressed_value(4, 5)      [  4 ];
  tos_tc                                               [  8 ];
  length             =:= inferred_ip_v4_length         [ 16 ];
  ip_id                                                [ 16 ];
  rf                 =:= uncompressed_value(1, 0)      [  1 ];
  df                                                   [  1 ];
  mf                 =:= uncompressed_value(1, 0)      [  1 ];
  frag_offset        =:= uncompressed_value(13, 0)     [ 13 ];
  ttl_hopl                                             [  8 ];
  next_header                                          [  8 ];
  ip_checksum        =:= inferred_ip_v4_header_checksum  [ 16 ];
  src_addr                                             [ 32 ];
  dest_addr                                            [ 32 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
 }

 UNCOMPRESSED v6 {
  ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
  outer_headers      =:= baseheader_outer_headers      [ VARIABLE ];
  ip_version         =:= uncompressed_value(4, 6)      [   4 ];
  tos_tc                                               [   8 ];
  flow_label                                           [  20 ];
  payload_length     =:= inferred_ip_v6_length         [  16 ];
  next_header                                          [   8 ];
  ttl_hopl                                             [   8 ];
  src_addr                                             [ 128 ];
  dest_addr                                            [ 128 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  df                 =:= uncompressed_value(0,0)       [   0 ];
  ip_id              =:= uncompressed_value(0,0)       [   0 ];
 }

 CONTROL {
  ENFORCE(profile_value == PROFILE_IP_0104);
  ENFORCE(profile == profile_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
 }

 DEFAULT {
  ENFORCE(outer_ip_flag == 0);
  tos_tc                        =:= static;
  dest_addr                     =:= static;
  ttl_hopl                      =:= static;
  src_addr                      =:= static;
  df                            =:= static;
  flow_label                    =:= static;
  next_header                   =:= static;
  reorder_ratio                 =:= static;
  ip_id_behavior_innermost =:= static;
 }

// Remplacement pour UOR-2-ext3
 COMPRESSED co_common {
  ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
  discriminator      =:= '11111010'                 [ 8 ];
  ip_id_indicator    =:= irregular(1)               [ 1 ];
  header_crc         =:= crc7(THIS.UVALUE, THIS.ULENGTH)      [ 7 ];
```

```
   flags_indicator      =:= irregular(1)                    [ 1 ];
   ttl_hopl_indicator   =:= irregular(1)                    [ 1 ];
   tos_tc_indicator     =:= irregular(1)                    [ 1 ];
   reorder_ratio        =:= irregular(2)                    [ 2 ];
   control_crc3         =:= control_crc3_encoding           [ 3 ];
   outer_ip_indicator : df : ip_id_behavior_innermost =:=
     profile_2_3_4_flags_enc(flags_indicator.CVALUE, ip_version.UVALUE)          [ 0, 8 ];
   tos_tc               =:= static_ou_irreg(tos_tc_indicator.CVALUE, 8)         [ 0, 8 ];
   ttl_hopl             =:= static_ou_irreg(ttl_hopl_indicator.CVALUE, ttl_hopl.ULENGTH)  [ 0, 8 ];
   msn                  =:= msn_lsb(8)                      [ 8 ];
   ip_id                =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
     ip_id_indicator.CVALUE)                               [ 0, 8, 16 ];
  }

// UO-0
  COMPRESSED pt_0_crc3 {
   discriminator =:= '0'                                   [ 1 ];
   msn           =:= msn_lsb(4)                            [ 4 ];
   header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
   ip_id          =:= inferred_sequential_ip_id            [ 0 ];
  }

// Nouveau format, type 0 avec CRC fort et plus de bits de SN
  COMPRESSED pt_0_crc7 {
   discriminator =:= '100'                                 [ 3 ];
   msn           =:= msn_lsb(6)                            [ 6 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
   ip_id         =:= inferred_sequential_ip_id             [ 0 ];
  }

// Remplacement de UO-1-ID (PT-1 seulement utilisé pour séquentiel)
  COMPRESSED pt_1_seq_id {
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
   discriminator =:= '101'                                 [ 3 ];
   header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)       [ 3 ];
   msn           =:= msn_lsb(6)                            [ 6 ];
   ip_id          =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  }

// Remplacement de UOR-2-ID
  COMPRESSED pt_2_seq_id {
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
   discriminator =:= '110'                                 [ 3 ];
   ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6) [ 6 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)       [ 7 ];
   msn           =:= msn_lsb(8)                            [ 8 ];
  }
}
```

Profil UDP-léger/RTP

```
udplite_rtp_baseheader(profile_value, ts_stride_value, time_stride_value, outer_ip_flag, ip_id_behavior_value,
reorder_ratio_value, coverage_behavior_value)
{
  UNCOMPRESSED v4 {
   ENFORCE(msn.UVALUE == sequence_number.UVALUE);
   outer_headers        =:= baseheader_outer_headers      [ VARIABLE ];
   ip_version           =:= uncompressed_value(4, 4)      [  4 ];
   header_length        =:= uncompressed_value(4, 5)      [  4 ];
   tos_tc                                                 [  8 ];
```

```
  length              =:= inferred_ip_v4_length           [ 16 ];
  ip_id                                                   [ 16 ];
  rf                  =:= uncompressed_value(1, 0)        [  1 ];
  df                                                      [  1 ];
  mf                  =:= uncompressed_value(1, 0)        [  1 ];
  frag_offset         =:= uncompressed_value(13, 0)       [ 13 ];
  ttl_hopl                                                [  8 ];
  next_header                                             [  8 ];
  ip_checksum         =:= inferred_ip_v4_header_checksum [ 16 ];
  src_addr                                                [ 32 ];
  dest_addr                                               [ 32 ];
  extension_headers   =:= baseheader_extension_headers    [ VARIABLE ];
  src_port                                                [ 16 ];
  dst_port                                                [ 16 ];
  checksum_coverage                                       [ 16 ];
  udp_checksum                                            [ 16 ];
  rtp_version         =:= uncompressed_value(2, 2)        [  2 ];
  pad_bit                                                 [  1 ];
  extension                                               [  1 ];
  cc                                                      [  4 ];
  marker                                                  [  1 ];
  payload_type                                            [  7 ];
  sequence_number                                         [ 16 ];
  timestamp                                               [ 32 ];
  ssrc                                                    [ 32 ];
  csrc_list                                               [ VARIABLE ];
}

UNCOMPRESSED v6 {
  ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
  outer_headers       =:= baseheader_outer_headers        [ VARIABLE ];
  ip_version          =:= uncompressed_value(4, 6)        [   4 ];
  tos_tc                                                  [   8 ];
  flow_label                                              [  20 ];
  payload_length      =:= inferred_ip_v6_length           [  16 ];
  next_header                                             [   8 ];
  ttl_hopl                                                [   8 ];
  src_addr                                                [ 128 ];
  dest_addr                                               [ 128 ];
  extension_headers   =:= baseheader_extension_headers    [ VARIABLE ];
  src_port                                                [  16 ];
  dst_port                                                [  16 ];
  checksum_coverage                                       [  16 ];
  udp_checksum                                            [  16 ];
  rtp_version         =:= uncompressed_value(2, 2)        [   2 ];
  pad_bit                                                 [   1 ];
  extension                                               [   1 ];
  cc                                                      [   4 ];
  marker                                                  [   1 ];
  payload_type                                            [   7 ];
  sequence_number                                         [  16 ];
  timestamp                                               [  32 ];
  ssrc                                                    [  32 ];
  csrc_list                                               [ VARIABLE ];
  df                  =:= uncompressed_value(0,0)         [   0 ];
  ip_id               =:= uncompressed_value(0,0)         [   0 ];
}

CONTROL {
  ENFORCE(profile_value == PROFILE_RTP_0107);
  ENFORCE(profile == profile_value);
  ENFORCE(time_stride.UVALUE == time_stride_value);
```

```
  ENFORCE(ts_stride.UVALUE == ts_stride_value);
  ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
  dummy_field =:= field_scaling(ts_stride.UVALUE,
    ts_scaled.UVALUE, horodatage.UVALUE, ts_offset.UVALUE) [ 0 ];
 }

 INITIAL {
  ts_stride      =:= uncompressed_value(32, TS_STRIDE_DEFAULT);
  time_stride   =:= uncompressed_value(32, TIME_STRIDE_DEFAULT);
 }

 DEFAULT {
  ENFORCE(outer_ip_flag == 0);
  tos_tc                    =:= static;
  dest_addr                 =:= static;
  ttl_hopl                  =:= static;
  src_addr                  =:= static;
  df                        =:= static;
  flow_label                =:= static;
  next_header               =:= static;
  src_port                  =:= static;
  dst_port                  =:= static;
  pad_bit                   =:= static;
  extension                 =:= static;
  cc                        =:= static;
// Quand le bit Marqueur n'est pas présent dans les paquets, il est supposé être à 0
  marker                    =:= uncompressed_value(1, 0);
  payload_type              =:= static;
  sequence_number           =:= static;
  timestamp                 =:= static;
  ssrc                      =:= static;
  csrc_list                 =:= static;
  ts_stride                 =:= static;
  time_stride               =:= static;
  ts_scaled                 =:= static;
  ts_offset                 =:= static;
  reorder_ratio             =:= static;
  ip_id_behavior_innermost =:= static;
 }

// Remplacement pour UOR-2-ext3
 COMPRESSED co_common {
  ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
  discriminator      =:= '11111010'                          [ 8 ];
  marker             =:= irregular(1)                        [ 1 ];
  header_crc         =:= crc7(THIS.UVALUE, THIS.ULENGTH)  [ 7 ];
  flags1_indicator   =:= irregular(1)                        [ 1 ];
  flags2_indicator   =:= irregular(1)                        [ 1 ];
  tsc_indicator      =:= irregular(1)                        [ 1 ];
  tss_indicator      =:= irregular(1)                        [ 1 ];
  ip_id_indicator    =:= irregular(1)                        [ 1 ];
  control_crc3       =:= control_crc3_encoding               [ 3 ];

  outer_ip_indicator : ttl_hopl_indicator : tos_tc_indicator : df : ip_id_behavior_innermost : reorder_ratio
    =:= profile_1_7_flags1_enc(flags1_indicator.CVALUE, ip_version.UVALUE)    [ 0, 8 ];
  list_indicator : pt_indicator : tis_indicator : pad_bit : extension : coverage_behavior =:=
    profile_7_flags2_enc(flags2_indicator.CVALUE)             [ 0, 8 ];
  tos_tc =:= static_ou_irreg(tos_tc_indicator.CVALUE, 8)      [ 0, 8 ];
  ttl_hopl =:= static_ou_irreg(ttl_hopl_indicator.CVALUE, 8)  [ 0, 8 ];
  payload_type =:= pt_irr_ou_static(pt_indicator.CVALUE)      [ 0, 8 ];
```

```
    sequence_number =:= sdvl_sn_lsb(sequence_number.ULENGTH)          [ VARIABLE ];
    ip_id        =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                                         [ 0, 8, 16 ];
    ts_scaled    =:= variable_scaled_timestamp(tss_indicator.CVALUE,
      tsc_indicator.CVALUE, ts_stride.UVALUE, time_stride.UVALUE)     [ VARIABLE ];
    timestamp  =:= variable_unscaled_timestamp(tss_indicator.CVALUE, tsc_indicator.CVALUE)  [ VARIABLE ];
    ts_stride    =:= sdvl_ou_static(tss_indicator.CVALUE)             [ VARIABLE ];
    time_stride =:= sdvl_ou_static(tis_indicator.CVALUE)              [ VARIABLE ];
    csrc_list    =:= csrc_list_presence(list_indicator.CVALUE, cc.UVALUE)  [ VARIABLE ];
  }

// UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                                          [ 1 ];
    msn           =:= msn_lsb(4)                                   [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
    timestamp     =:= inferred_scaled_field                        [ 0 ];
    ip_id         =:= inferred_sequential_ip_id                    [ 0 ];
  }

// Nouveau format, type 0 avec CRC fort et plus de bits de SN
  COMPRESSED pt_0_crc7 {
    discriminator =:= '1000'                                       [ 4 ];
    msn           =:= msn_lsb(5)                                   [ 5 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)              [ 7 ];
    timestamp     =:= inferred_scaled_field                        [ 0 ];
    ip_id         =:= inferred_sequential_ip_id                    [ 0 ];
  }

// Remplacement de UO-1
  COMPRESSED pt_1_rnd {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
    discriminator =:= '101'                                        [ 3 ];
    marker        =:= irregular(1)                                 [ 1 ];
    msn           =:= msn_lsb(4)                                   [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)         [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
  }

// Remplacement de UO-1-ID
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator  =:= '1001'                                      [ 4 ];
    ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
    msn           =:= msn_lsb(5)                                   [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
    timestamp     =:= inferred_scaled_field                        [ 0 ];
  }

// Remplacement de UO-1-TS
  COMPRESSED pt_1_seq_ts {
    ENFORCE(ts_stride.UVALUE != 0);
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                        [ 3 ];
    marker        =:= irregular(1)                                 [ 1 ];
    msn           =:= msn_lsb(4)                                   [ 4 ];
    ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)         [ 5 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)              [ 3 ];
```

```
   ip_id          =:= inferred_sequential_ip_id                    [ 0 ];
  }

 // Remplacement de UOR-2
  COMPRESSED pt_2_rnd {
   ENFORCE(ts_stride.UVALUE != 0);
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_ZERO));
   discriminator =:= '110'                                         [ 3 ];
   msn           =:= msn_lsb(7)                                    [ 7 ];
   ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 6)          [ 6 ];
   marker        =:= irregular(1)                                  [ 1 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)               [ 7 ];
  }

 // Remplacement de UOR-2-ID
  COMPRESSED pt_2_seq_id {
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
   discriminator =:= '11000'                                       [ 5 ];
   msn           =:= msn_lsb(7)                                    [ 7 ];
   ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)               [ 7 ];
   timestamp     =:= inferred_scaled_field                         [ 0 ];
  }

 // Remplacement de UOR-2-ID-ext1 (TS et IP-ID)
  COMPRESSED pt_2_seq_both {
   ENFORCE(ts_stride.UVALUE != 0);
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
   discriminator =:= '11001'                                       [ 5 ];
   msn           =:= msn_lsb(7)                                    [ 7 ];
   ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 5) [ 5 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)               [ 7 ];
   ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 7)          [ 7 ];
   marker        =:= irregular(1)                                  [ 1 ];
  }

 // Remplacement de UOR-2-TS
  COMPRESSED pt_2_seq_ts {
   ENFORCE(ts_stride.UVALUE != 0);
   ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
         (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
   discriminator =:= '1101'                                        [ 4 ];
   msn           =:= msn_lsb(7)                                    [ 7 ];
   ts_scaled     =:= scaled_ts_lsb(time_stride.UVALUE, 5)          [ 5 ];
   marker        =:= irregular(1)                                  [ 1 ];
   header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)               [ 7 ];
   ip_id         =:= inferred_sequential_ip_id                     [ 0 ];
  }
 }
```

Profil UDP-léger

```
udplite_baseheader(profile_value, outer_ip_flag, ip_id_behavior_value, reorder_ratio_value, coverage_behavior_value)
{
 UNCOMPRESSED v4 {
   outer_headers      =:= baseheader_outer_headers      [ VARIABLE ];
   ip_version         =:= uncompressed_value(4, 4)      [  4 ];
   header_length      =:= uncompressed_value(4, 5)      [  4 ];
   tos_tc                                               [  8 ];
```

```
  length              =:= inferred_ip_v4_length          [ 16 ];
  ip_id                                                  [ 16 ];
  rf                  =:= uncompressed_value(1, 0)       [  1 ];
  df                                                     [  1 ];
  mf                  =:= uncompressed_value(1, 0)       [  1 ];
  frag_offset         =:= uncompressed_value(13, 0)      [ 13 ];
  ttl_hopl                                               [  8 ];
  next_header                                            [  8 ];
  ip_checksum         =:= inferred_ip_v4_header_checksum  [ 16 ];
  src_addr                                               [ 32 ];
  dest_addr                                              [ 32 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  src_port                                               [ 16 ];
  dst_port                                               [ 16 ];
  checksum_coverage                                      [ 16 ];
  udp_checksum                                           [ 16 ];
 }

 UNCOMPRESSED v6 {
  ENFORCE(ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_RANDOM);
  outer_headers       =:= baseheader_outer_headers       [ VARIABLE ];
  ip_version          =:= uncompressed_value(4, 6)       [   4 ];
  tos_tc                                                 [   8 ];
  flow_label                                             [  20 ];
  payload_length      =:= inferred_ip_v6_length          [  16 ];
  next_header                                            [   8 ];
  ttl_hopl                                               [   8 ];
  src_addr                                               [ 128 ];
  dest_addr                                              [ 128 ];
  extension_headers =:= baseheader_extension_headers [ VARIABLE ];
  src_port                                               [  16 ];
  dst_port                                               [  16 ];
  checksum_coverage                                      [  16 ];
  udp_checksum                                           [  16 ];
  df                  =:= uncompressed_value(0,0)        [   0 ];
  ip_id               =:= uncompressed_value(0,0)        [   0 ];
 }

 CONTROL {
  ENFORCE(profile_value == PROFILE_UDPLITE_0108);
  ENFORCE(profile == profile_value);
  ENFORCE(coverage_behavior.UVALUE == coverage_behavior_value);
  ENFORCE(reorder_ratio.UVALUE == reorder_ratio_value);
  ENFORCE(ip_id_behavior_innermost.UVALUE == ip_id_behavior_value);
 }

 DEFAULT {
  ENFORCE(outer_ip_flag == 0);
  tos_tc                      =:= static;
  dest_addr                   =:= static;
  ttl_hopl                    =:= static;
  src_addr                    =:= static;
  df                          =:= static;
  flow_label                  =:= static;
  next_header                 =:= static;
  src_port                    =:= static;
  dst_port                    =:= static;
  reorder_ratio               =:= static;
  ip_id_behavior_innermost =:= static;
 }

// Remplacement pour UOR-2-ext3
```

```
  COMPRESSED co_common {
    ENFORCE(outer_ip_flag == outer_ip_indicator.CVALUE);
    discriminator          =:= '11111010'                          [ 8 ];
    ip_id_indicator        =:= irregular(1)                        [ 1 ];
    Y_crc                  =:= crc7(THIS.UVALUE, THIS.ULENGTH)  [ 7 ];
    flags_indicator        =:= irregular(1)                        [ 1 ];
    ttl_hopl_indicator     =:= irregular(1)                        [ 1 ];
    tos_tc_indicator       =:= irregular(1)                        [ 1 ];
    reorder_ratio          =:= irregular(2)                        [ 2 ];
    control_crc3           =:= control_crc3_encoding               [ 3 ];
    outer_ip_indicator : df : ip_id_behavior_innermost :
      coverage_behavior  =:= profile_8_flags_enc(flags_indicator.CVALUE,
      ip_version.UVALUE)                                           [ 0, 8 ];
    tos_tc                 =:= static_or_irreg(tos_tc_indicator.CVALUE, 8) [ 0, 8 ];
    ttl_hopl               =:= static_or_irreg(ttl_hopl_indicator.CVALUE,
      ttl_hopl.ULENGTH)                                            [ 0, 8 ];
    msn                    =:= msn_lsb(8)                          [ 8 ];
    ip_id                  =:= ip_id_sequential_variable(ip_id_behavior_innermost.UVALUE,
      ip_id_indicator.CVALUE)                                      [ 0, 8, 16 ];
  }

// UO-0
  COMPRESSED pt_0_crc3 {
    discriminator =:= '0'                                        [ 1 ];
    msn           =:= msn_lsb(4)                                 [ 4 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH) [ 3 ];
    ip_id          =:= inferred_sequential_ip_id                 [ 0 ];
  }

// Nouveau format, type 0 avec CRC fort et plus de bits de SN
  COMPRESSED pt_0_crc7 {
    discriminator =:= '100'                                      [ 3 ];
    msn           =:= msn_lsb(6)                                 [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH) [ 7 ];
    ip_id          =:= inferred_sequential_ip_id                 [ 0 ];
  }

// Remplacement de UO-1-ID (PT-1 seulement utilisé pour séquentiel)
  COMPRESSED pt_1_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '101'                                      [ 3 ];
    header_crc    =:= crc3(THIS.UVALUE, THIS.ULENGTH)            [ 3 ];
    msn           =:= msn_lsb(6)                                 [ 6 ];
    ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 4) [ 4 ];
  }

// Remplacement de UOR-2-ID
  COMPRESSED pt_2_seq_id {
    ENFORCE((ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL) ||
        (ip_id_behavior_innermost.UVALUE == IP_ID_BEHAVIOR_SEQUENTIAL_SWAPPED));
    discriminator =:= '110'                                      [ 3 ];
    ip_id         =:= ip_id_lsb(ip_id_behavior_innermost.UVALUE, 6)  [ 6 ];
    header_crc    =:= crc7(THIS.UVALUE, THIS.ULENGTH)            [ 7 ];
    msn           =:= msn_lsb(8)                                 [ 8 ];
  }
}
```

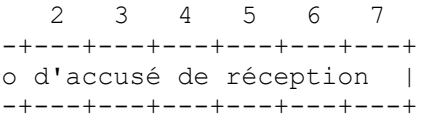
6.9 Formats et options de rétroaction

6.9.1 Formats de rétroaction

Ce paragraphe décrit le format des rétroactions pour les profils ROHCv2, en utilisant les formats décrits au paragraphe 5.2.3 de la [RFC4995].

Le champ Numéro d'accusé de réception des formats de rétroactions contient les bits de moindre poids du MSN (voir le paragraphe 6.3.1) qui correspondent à l'en-tête de référence dont il est accusé réception. Un en-tête de référence est un en-tête dont le CRC-8 a été validé avec succès ou dont le CRC est vérifié. Si il n'y a pas d'en-tête de référence disponible, la rétroaction DOIT porter une option ACKNUMBER-NOT-VALID *(numéro d'accusé de réception non valide)*.

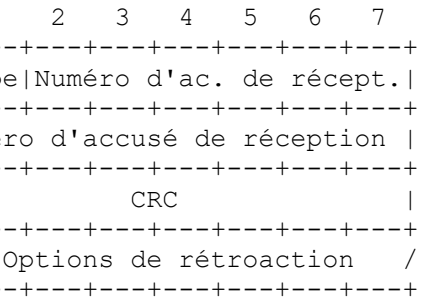
FEEDBACK-1



Numéro d'accusé de réception : les huit bits de moindre poids du MSN.

Un FEEDBACK-1 est un ACK. Afin d'envoyer un NACK ou un STATIC-NACK, FEEDBACK-2 doit être utilisé.

FEEDBACK-2



Acktype :
0 = ACK
1 = NACK
2 = STATIC-NACK
3 est réservé (NE DOIT PAS être utilisé pour l'analyse)

Numéro d'accusé de réception : les bits de moindre poids du MSN.

CRC : CRC de 8 bits calculé sur la charge utile de rétroaction entière incluant tous les champs de CID mais sans le type de rétroaction, le champ "Taille", et l'octet "Code", en utilisant le polynôme défini au paragraphe 5.3.1.1 de la [RFC4995]. Si le CID est donné avec un octet Add-CID, l'octet Add-CID précède immédiatement le format FEEDBACK-1 ou FEEDBACK-2. Pour les besoins du calcul du CRC, le champ CRC est zéro.

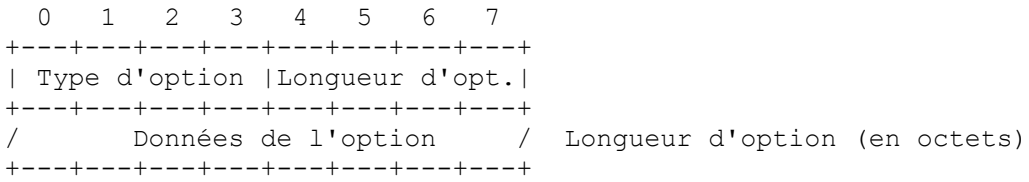
Options de rétroaction : nombre variable d'options de rétroaction, voir le paragraphe 6.9.2. Les options peuvent apparaître dans n'importe quel ordre.

Un FEEDBACK-2 de type NACK ou STATIC-NACK est toujours implicitement un accusé de réception pour un paquet décompressé avec succès, qui correspond à un paquet dont les LSB correspondent au numéro d'accusé de réception de l'élément de rétroaction, sauf si l'option ACKNUMBER-NOT-VALID (voir le paragraphe 6.9.2.2) apparaît dans l'élément de rétroaction.

Le format FEEDBACK-2 porte toujours un CRC et est donc plus robuste que le format FEEDBACK-1. Quand il reçoit un FEEDBACK-2, le compresseur DOIT vérifier les informations en calculant le CRC et en comparant le résultat au CRC porté dans le format de rétroaction. Si les deux ne sont pas identiques, l'élément de rétroaction DOIT être éliminé.

6.9.2 Options de rétroaction

Une option de rétroaction a une longueur variable et le format général suivant :



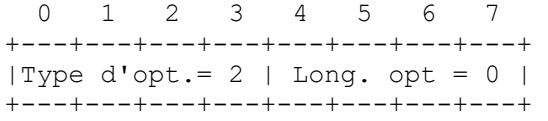
Type d'option : entier non signé qui représente le type de l'option de rétroaction. Les paragraphes 6.9.2.1 à 6.9.2.4 décrivent les options de rétroaction de ROHCv2.

Longueur d'option : entier non signé qui représente la longueur du champ Données de l'option, en octets.

Données de l'option : données spécifiques du type de rétroaction. Présent si la valeur du champ Longueur d'option est réglée à une valeur non zéro.

6.9.2.1 Option REJECT

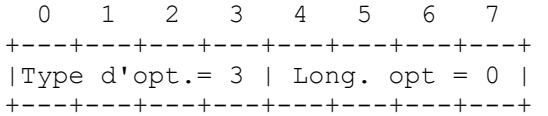
L'option REJECT informe le compresseur que le décompresseur n'a pas des ressources suffisantes pour traiter le flux.



Quand il reçoit une option REJECT, le compresseur DOIT arrêter de compresser le flux de paquets, et DEVRAIT s'abstenir de tenter d'augmenter le nombre de flux de paquets compressés pendant un certain temps. L'option REJECT NE DOIT PAS apparaître plus d'une fois dans le format FEEDBACK-2 ; autrement, le compresseur DOIT éliminer l'élément de rétroaction entier.

6.9.2.2 Option ACKNUMBER-NOT-VALID

L'option ACKNUMBER-NOT-VALID indique que le champ Numéro d'accusé de réception de la rétroaction n'est pas valide.

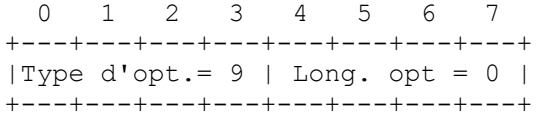


Un compresseur NE DOIT PAS utiliser le numéro d'accusé de réception de la rétroaction pour trouver l'en-tête envoyé correspondant quand cette option est présente. Quand cette option est utilisée, le champ Numéro d'accusé de réception du format FEEDBACK-2 est réglé à zéro. Par conséquent, un type de rétroaction de NACK ou STATIC-NACK envoyé avec l'option ACKNUMBER-NOT-VALID est équivalent à un STATIC-NACK par rapport au type de réparation de contexte demandé par le décompresseur.

L'option ACKNUMBER-NOT-VALID NE DOIT PAS apparaître plus d'une fois dans le format FEEDBACK-2 ; autrement, le compresseur DOIT éliminer l'élément de rétroaction entier.

6.9.2.3 Option CONTEXT_MEMORY

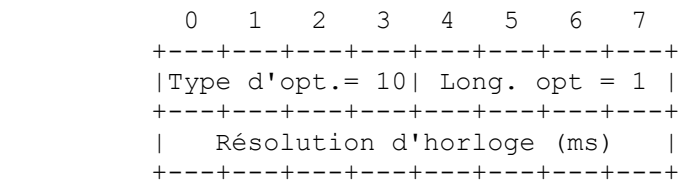
L'option CONTEXT_MEMORY informe le compresseur que le décompresseur n'a pas des ressources de mémoire suffisantes pour traiter le contexte de flux de paquets, car le flux est actuellement compressé.



Quand il reçoit une option CONTEXT_MEMORY, le compresseur DEVRAIT prendre des mesures pour compresser le flux de paquets d'une façon qui exige moins de ressources de mémoire du décompresseur ou arrêter de compresser le flux de paquets. L'option CONTEXT_MEMORY NE DOIT PAS apparaître plus d'une fois dans le format FEEDBACK-2 ; autrement, le compresseur DOIT éliminer l'élément de rétroaction entier.

6.9.2.4 Option CLOCK_RESOLUTION

L'option CLOCK_RESOLUTION informe le compresseur de la résolution d'horloge du décompresseur. Elle informe aussi si le décompresseur prend ou non en charge la compression de l'horodatage RTP (voir le paragraphe 6.6.9). L'option CLOCK_RESOLUTION est applicable par canal, c'est-à-dire, elle s'applique à tout contexte associé à un profil pour lequel l'option est pertinente entre une paire de compresseur et décompresseur.



Résolution d'horloge : entier non signé qui représente la résolution d'horloge du décompresseur exprimée in millisecondes.

La plus petite résolution d'horloge qui peut être indiquée est 1 milliseconde. La valeur zéro a une signification particulière : elle indique que le décompresseur ne peut pas faire la compression fondée sur le temporisateur de l'horodatage RTP. L'option CLOCK_RESOLUTION NE DOIT PAS apparaître plus d'une fois dans le format FEEDBACK-2 ; autrement, le compresseur DOIT éliminer l'élément de rétroaction entier.

6.9.2.5 Types d'option inconnus

Si un type d'option autre que ceux définis dans ce document est rencontré, le compresseur DOIT éliminer l'élément de rétroaction entier.

7. Considérations sur la sécurité

Des détériorations telles que des erreurs binaires sur les en-têtes compressés reçus, des paquets manquants, et des réarrangements entre paquets pourraient causer la reconstitution de paquets erronés par le décompresseur d'en-tête, c'est-à-dire, de paquets qui ne correspondent pas au paquet original, mais ont quand même des en-têtes IP, UDP (ou UDP-Léger), et RTP valides, et éventuellement aussi des sommes de contrôle UDP (ou UDP-Léger) valides.

Les profils de compression d'en-tête définis ici utilisent une somme de contrôle interne pour la vérification des en-têtes reconstruits. Cela réduit la probabilité qu'un décompresseur d'en-tête livre des paquets erronés aux couches supérieures sans que l'erreur soit remarquée. En particulier, la probabilité que des paquets erronés consécutifs ne soient pas détectés par la somme de contrôle interne est proche de zéro.

Cette petite, mais non nulle, probabilité reste inchangée quand la protection de l'intégrité est appliquée après la compression et vérifiée avant la décompression, dans le cas où un attaquant pourrait éliminer ou réarranger les paquets entre les points d'extrémité de compression.

Les détériorations mentionnées ci-dessus pourraient être causées par le mauvais fonctionnement ou la malveillance d'un compresseur de sous en-tête. Une telle corruption peut être détectée avec des mécanismes d'intégrité de bout en bout qui ne seront pas affectés par la compression. De plus, la somme de contrôle interne peut aussi être utile dans le cas de mauvais fonctionnement du compresseur.

Les attaques de déni de service sont possibles si un intrus peut introduire (par exemple) des paquets IR ou FEEDBACK bogués sur la liaison et par là causer la réduction de l'efficacité de compression. Cependant, un intrus qui a la capacité d'injecter des paquets arbitraires à la couche de liaison de cette manière soulève des problèmes de sécurité supplémentaires qui surpassent ceux relatifs à l'utilisation de la compression d'en-tête.

8. Considérations relatives à l'IANA

Les identifiants de profil ROHC suivants ont été alloués par l'IANA pour les profils définis dans ce document :

Identifiant	Profil
0x0101	ROHCv2 RTP
0x0102	ROHCv2 UDP
0x0103	ROHCv2 ESP
0x0104	ROHCv2 IP
0x0107	ROHCv2 RTP/UDP-Léger
0x0108	ROHCv2 UDP-Léger

9. Remerciements

Les auteurs tiennent à remercier Mark West, Robert Finking, Haipeng Jin, et Rohit Kapoor qui ont accepté d'être les relecteurs officiels du document , et aussi pour les discussions constructives durant son développement. Merci à Carl Knutsson pour pour ses larges contributions à cette spécification, ainsi qu'à Jani Juvan et Anders Edqvist pour leurs commentaires et retours utiles. Merci aussi à Elwyn Davies pour sa relecture au titre de l'équipe de révision générale de zone (Gen-ART) et à Stephen Kent pour sa relecture au nom de la direction de la sécurité de l'IETF, durant le dernier appel à l'IETF. Finalement, merci aux nombreuses personnes qui ont contribué aux précédentes spécifications ROHC et qui ont soutenu cet effort.

10. Références

10.1 Références normatives

[RFC0768] J. Postel, "Protocole de datagramme d'utilisateur (UDP)", (STD 6), 28 août 1980.

[RFC0791] J. Postel, éd., "Protocole Internet - Spécification du protocole du programme Internet", STD 5, septembre 1981.

[RFC2004] C. Perkins, "Encapsulation minimale au sein de IP", octobre 1996. *(P.S.)*

[RFC2119] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", BCP 14, mars 1997. *(MàJ par* RFC8174)

[RFC2460] S. Deering et R. Hinden, "Spécification du protocole Internet, version 6 (IPv6)", décembre 1998. *(MàJ par* 5095, 6564 *; D.S ; Remplacée par* RFC8200, STD 86*)*

[RFC2784] D. Farinacci, T. Li, S. Hanks, D. Meyer et P. Traina, "Encapsulation d'acheminement générique (GRE)", mars 2000.

[RFC2890] G. Dommety, "Extensions de clé et de numéro de séquence à GRE", septembre 2000. *(P.S.)*

[RFC3550] H. Schulzrinne, S. Casner, R. Frederick et V. Jacobson, "RTP : un protocole de transport pour les applications en temps réel", STD 64, juillet 2003. *(MàJ par* RFC7164, RFC7160, RFC8083, RFC8108, RFC8860)

[RFC3828] L-A. Larzon et autres, "Protocole léger de datagramme d'utilisateur (UDP-Léger)", juillet 2004. *(P.S, MàJ par* RFC6335)

[RFC4019] G. Pelletier, "Compression d'en-tête robuste (ROHC) : profils pour le protocole de datagramme d'utilisateur (UDP) léger", avril 2005. (*MàJ par* RFC4815) *(P.S.)*

[RFC4302] S. Kent, "En-tête d'authentification IP", décembre 2005. *(P.S.)*

[RFC4303] S. Kent, "Encapsulation de charge utile de sécurité dans IP (ESP)", décembre 2005. (*Remplace* RFC2406) *(P.S.)*

[RFC4995] L-E. Jonsson et autres, "Cadre de la compression d'en-tête robuste (ROHC)", juillet 2007. *(Remplacée par* RFC5795)

[RFC4997] R. Finking, G. Pelletier, "Notation formelle pour la compression d'en-tête robuste (ROHC-FN)", juillet 2007. *(P.S.)*

10.2 Références pour information

[RFC2675] D. Borman, S. Deering, R. Hinden, "Jumbogrammes IPv6", août 1999. *(P.S.)*

[RFC3095] C. Bormann et autres, "Compression d'en-tête robuste (ROHC) : cadre et quatre profils", juillet 2001. (*MàJ par* RFC3759, RFC4815) *(P.S.)*

[RFC3843] L-E. Jonsson, G. Pelletier, "Compression d'en-tête robuste (ROHC) : un profil de compression pour IP", juin 2004. (*MàJ par* RFC4815) *(P.S.)*

[RFC4224] G. Pelletier et autres, "Compression d'en-tête robuste (ROHC) : ROHC sur des canaux qui peuvent réordonner les paquets", janvier 2006. *(Information)*

Appendice A. Classification détaillée des champs d'en-tête

La compression d'en-tête est possible du fait que la plupart des champs d'en-tête ne varient pas de façon aléatoire d'un paquet à l'autre. Beaucoup des champs présentent un comportement statique ou changent d'une façon plus ou moins prévisible. Quand on conçoit un schéma de compression d'en-têtes, il est d'une importance fondamentale de comprendre le détail du comportement des champs.

Dans cet appendice, tous les champs des en-têtes compressibles par ces profils sont classés et analysés. L'analyse se fonde sur le comportement pour les types de trafic qui sont supposés être le plus fréquemment compressés (par exemple, le comportement de champ RTP est fondé sur le comportement de trafic de voix et/ou vidéo).

Les champs sont classés comme appartenant à une des classes suivantes :

INFERRED - Ces champs contiennent des valeurs qui peuvent être déduites d'autres valeurs, par exemple la taille de la trame portant le paquet, et donc n'ont pas à être incluses dans les paquets compressés.

STATIC - Ces champs sont supposés être constants durant toute la durée de vie du flux ; en général, il est suffisant de concevoir un format compressé de sorte que ces champs sont seulement mis à jour par les paquets IR.

STATIC-DEF - Ces champs sont supposés être constants durant toute la durée de vie du flux et leurs valeurs peuvent être utilisées pour définir un flux. Ils sont seulement envoyés dans les paquets IR.

STATIC-KNOWN - Ces champs sont supposés avoir des valeurs bien connues et donc n'ont pas besoin d'être communiqués du tout.

SEMISTATIC - Ces champs sont inchangés la plupart du temps. Cependant, occasionnellement, la valeur change mais va revenir à sa valeur d'origine. Pour ROHCv2, les valeurs de tels champs n'ont pas besoin d'être changées avec les plus petits formats de paquet compressé, mais il devrait être possible de les changer via des paquets compressés légèrement plus grands.

RARELY CHANGING (RACH) - Ce sont des champs qui changent leurs valeurs occasionnellement et ensuite gardent leur nouvelle valeur. Pour ROHCv2, les valeurs de tels champs n'ont pas besoin qu'il soit possible de les changer avec les plus petits formats de paquet compressé, mais il devrait être possible de les changer via des paquets compressés légèrement plus grands.

IRREGULAR - Ce sont les champs pour lesquels aucun schéma de changement utile ne peut être identifié et ils devraient être transmis non compressés dans tous les paquets compressés.

PATTERN - Ce sont des champs qui changent entre chaque paquet, mais selon un schéma prévisible.

A.1 Champs d'en-tête IPv4

Champ	Classe
Version	STATIC-KNOWN
Longueur d'en-tête	STATIC-KNOWN
Type de service	RACH
Longueur de paquet	INFERRED
Identification	
Séquentiel	PATTERN
Échange séquentiel	PATTERN
Aléatoire	IRREGULAR
Zéro	STATIC
Fanion Réservé	STATIC-KNOWN
Fanion Ne pas fragmenter	RACH
Fanion Fragments à venir	STATIC-KNOWN
Décalage de fragment	STATIC-KNOWN
Durée de vie	RACH
Protocole	STATIC-DEF
Somme de contrôle d'en-tête	INFERRED
Adresse de source	STATIC-DEF
Adresse de destination	STATIC-DEF

Version : le champ Version déclare quelle version IP est utilisée et est réglé à la valeur quatre.

Longueur d'en-tête : tant qu'aucune option n'est présente dans l'en-tête IP, la longueur d'en-tête est constante avec la valeur cinq. Si il y a des options, le champ pourrait être RACH ou STATIC-DEF, mais seulement les en-têtes sans option sont compressés par les profils ROHCv2. Le champ est donc classé STATIC-KNOWN.

Type de service : pour le type de flux compressés par les profils ROHCv2, les champs DSCP (Codet de service différencié) et ECN (Notification explicite d'encombrement) sont supposés changer relativement rarement.

Longueur de paquet : les informations sur la longueur de paquet sont supposées être fournies par la couche de liaison. Le champ est donc classé comme INFERRED.

Identification IPv4 : le champ Identification (IP-ID) est utilisé pour identifier quels fragments constituent un datagramme quand on réassemble des datagrammes fragmentés. La spécification IPv4 ne spécifie pas exactement comment des valeurs sont allouées à ce champ, mais seulement que chaque paquet devrait obtenir un IP-ID unique pour la paire source-destination et le protocole pour le temps où le datagramme (ou ses fragments) pourrait être en vie dans le réseau. Cela signifie que l'allocation des valeurs d'IP-ID peut être faite de diverses façons, mais les comportements attendus ont été séparés en quatre classes :

Séquentiel : dans ce comportement, le IP-ID est supposé être incrémenté de un pour la plupart des paquets, mais peut s'incrémenter d'une valeur supérieure à un, selon le comportement de la pile IPv4 transmetteuse.

Échange séquentiel : quand on utilise ce comportement, le IP-ID se comporte comme dans le comportement Séquentiel, mais les deux octets d'IP-ID sont échangés. Donc, le IP-ID peut être échangé avant la compression pour faire qu'il se comporte exactement comme le comportement Séquentiel.

Aléatoire : certaines piles IP allouent au IP-ID des valeurs en utilisant un générateur de nombres pseudo aléatoires. Il n'y a donc pas de corrélation entre les valeurs d'identifiant des datagrammes qui se suivent, et donc il n'y a pas de moyen de prédire la valeur de l'IP-ID pour le datagramme suivant. Pour les besoins de la compression d'en-tête, cela signifie que le champ IP-ID doit être envoyé non compressé avec chaque datagramme, résultant en deux octets d'en-tête supplémentaires.

Zéro : ce comportement, bien que de mise en œuvre non légale dans IPv4, est parfois vu dans des piles IPv4 existantes. Quand ce comportement est utilisé, tous les paquets IP ont la valeur de IP-ID réglée à zéro.

Fanions :le fanion Réservé doit être réglé à zéro et est donc classé comme STATIC-KNOWN. Le fanion Ne pas fragmenter (DF) change rarement et est donc classé comme RACH. Finalement, le fanion Fragments à venir (MF) est supposé être zéro parce que les fragments IP ne vont pas être compressés par ROHC et est donc classé comme STATIC-KNOWN.

Décalage de fragment : dans l'hypothèse où aucune fragmentation ne se produit, le décalage de fragment est toujours zéro et est donc classé comme STATIC-KNOWN.

Durée de vie : le champ Durée de vie est supposé être constant durant la vie d'un flux ou alterner entre un nombre limité de valeurs dues aux changements de chemin.

Protocole : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Somme de contrôle d'en-tête : la somme de contrôle d'en-tête protège les bonds individuels contre le traitement d'un en-tête corrompu. Quand presque toutes les informations d'en-tête IP sont compressées, il n'y a pas de raison d'avoir cette somme de contrôle supplémentaire ; elle peut plutôt être régénérée du côté du décompresseur. Le champ est donc classé comme INFERRED.

Adresses de source et de destination : ces champs font partie de la définition d'un flux et doivent donc être constants pour tous les paquets dans le flux.

A.2 Champs d'en-tête IPv6

Champ	Classe
Version	STATIC-KNOWN
Classe de trafic	RACH
Étiquette de flux	STATIC-DEF
Longueur de charge utile	INFERRED
Prochain en-tête	STATIC-DEF
Limite de bonds	RACH
Adresse de source	STATIC-DEF
Adresse de destination	STATIC-DEF

Version : le champ Version déclare quelle version IP est utilisée et est réglé à la valeur six.

Classe de trafic : pour le type de flux compressés par les profils ROHCv2, les champs DSCP et ECN sont supposés ne changer que relativement rarement.

Étiquette de flux : ce champ peut être utilisé pour identifier les paquets appartenant à un champ spécifique. Si il n'est pas utilisé, la valeur devrait être réglée à zéro. Autrement, tous les paquets appartenant au même flux doivent avoir la même valeur dans ce champ. Le champ est donc classé comme STATIC-DEF.

Longueur de charge utile : les informations sur la longueur du paquet (et, par conséquent, la longueur de la charge utile) sont supposées être fournies par la couche de liaison. Le champ est donc classé comme INFERRED.

Prochain en-tête : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Limite de bonds : le champ Limite de bonds est supposé être constant durant la vie d'un flux ou alterner entre un nombre limité de valeurs dues aux changements de chemin.

Adresses de source et destination : ces champs font partie de la définition d'un flux et doivent donc être constants pour tous les paquets du flux. Les champs sont donc classés comme STATIC-DEF.

A.3 Champs d'en-tête UDP

Champ	Classe
Accès de source	STATIC-DEF
Accès de destination	STATIC-DEF
Longueur	INFERRED
Somme de contrôle	
Désactivée	STATIC
Activée	IRREGULAR

Accès de source et destination : ces champs font partie de la définition d'un flux et doivent donc être constants pour tous les paquets du flux.

Longueur : les informations sur la longueur du paquet sont supposées être fournies par la couche de liaison. Le champ est donc classé comme INFERRED.

Somme de contrôle : la somme de contrôle peut être facultative. Si elle est désactivée, sa valeur est constamment zéro et peut être compressée. Si elle est activée, sa valeur dépend de la charge utile, qui pour les besoins de la compression est équivalente à un changement aléatoire à chaque paquet.

A.4 Champs d'en-tête UDP-léger

Champ	Classe
Accès de source	STATIC-DEF
Accès de destination	STATIC-DEF
Couverture de somme de contrôle	
Zéro	STATIC-DEF
Constante	INFERRED
Variable	IRREGULAR
Somme de contrôle	IRREGULAR

Accès de source et destination : ces champs font partie de la définition d'un flux et doivent donc être constants pour tous les paquets du flux.

Couverture de somme de contrôle : ce champ peut se comporter de différentes façons : il peut avoir une valeur de zéro, il peut être égal à la longueur du datagramme, ou il peut avoir toute valeur entre huit octets et la longueur du datagramme. Du point de vue de la compression, ce champ est supposé être entièrement prévisible (pour les cas où il suit le même comportement que le champ Longueur UDP, ou lorsque il prend une valeur constante) ou changer de façon aléatoire pour chaque paquet (rendant la valeur imprévisible du point de vue de la compression d'en-tête). Dans tous les cas, le comportement lui-même n'est pas supposé changer pour ce champ durant la vie d'un flux de paquets, ou changer relativement rarement.

Somme de contrôle : les informations utilisées pour le calcul de la somme de contrôle UDP-Léger sont gouvernées par la valeur de la couverture de somme de contrôle et incluent au minimum l'en-tête UDP-Léger. La somme de contrôle est un champ changeant qui doit toujours être envoyé tel qu'il est.

A.5 Champs d'en-tête RTP

Champ	Classe
Version	STATIC-KNOWN
Bourrage	RACH
Extension	RACH
Compteur CSRC	RACH
Marqueur	SEMISTATIC
Type de charge utile	RACH
Numéro de séquence	PATTERN
Horodatage	PATTERN
SSRC	STATIC-DEF
CSRC	RACH

Version : Ce champ est supposé avoir la valeur deux et le champ est donc classé comme STATIC-KNOWN.

Bourrage : l'utilisation de ce champ dépend de l'application, mais quand le bourrage de la charge utile est utilisé, il est probable qu'il sera présent dans la plupart ou tous les paquets. Le champ est classé comme RACH pour permettre les cas où la valeur de ce champ change.

Extension :si des extensions RTP sont utilisées par l'application, ces extensions sont souvent présentes dans tous les paquets, bien que l'utilisation de ces extensions soit peu fréquente. Pour permettre la compression efficace d'un flux en utilisant les extensions dans seulement quelques paquets, ce champ est classé comme RACH.

Compte de CSRC : ce champ indique le nombre d'éléments de CSRC présents dans la liste de CSRC. Ce nombre est supposé être le plus souvent constant paquet par paquet et quand il change, changer d'une petite quantité. Tant qu'on utilise pas de mixeur RTP, la valeur de ce champ va être zéro.

Marqueur : pour l'audio, le bit marqueur devrait n'être établi que dans le premier paquet d'une salve de parole, tandis que pour la vidéo, il devrait être établi dans le dernier paquet de chaque image. Cela signifie que dans les deux cas, le marqueur RTP est classé comme SEMISTATIC.

Type de charge utile : les applications pourraient s'adapter à l'encombrement en changeant le type de charge utile et/ou les tailles de trame, mais cela n'est pas supposé arriver fréquemment, de sorte que ce champ est classé comme RACH.

Numéro de séquence RTP : le numéro de séquence RTP va être incrémenté de un pour chaque paquet envoyé.

Horodatage :
Dans le cas audio : tant qu'il n'y a pas de pause dans le flux audio, l'horodatage RTP va être incrémenté d'une valeur constante, qui correspond au nombre d'échantillons de la trame de parole. Il va donc dans la plupart des cas suivre le numéro de séquence RTP. Quand il y a eu une prériode de silence et que commence une nouvelle salve de parole, l'horodatage va sauter en proportion de la longueur de la période de silence. Cependant, l'incrément va probablement être dans une gamme relativement limitée.
Dans le cas de la vidéo : entre deux paquets consécutifs, l'horodatage va être soit inchangé, soit augmenter d'un multiple d'une valeur fixe correspondant à la fréquence d'horloge de l'image. L'horodatage peut aussi diminuer d'un multiple de la valeur fixe pour certains schémas de codage. Le changement de valeur d'horodatage, exprimé comme un multiple de la fréquence d'horloge de l'image, est dans la plupart des cas dans une gamme limitée.

SSRC : ce champ fait partie de la définition d'un flux et doit donc être constant pour tous les paquets du flux. Le champ est donc classé comme STATIC-DEF.

Sources contributives (CSRC) : les participants à une session, qui sont identifiés par le champ CSRC, sont généralement supposés être inchangés d'un paquet à l'autre, mais vont changer de temps en temps par quelque ajout et/ou suppression.

A.6 Champs d'en-tête ESP

Champ	Classe
SPI	STATIC-DEF
numéro de séquence	PATTERN

SPI : ce champ est utilisé pour identifier un flux distinct entre deux homologues IPsec et il change rarement ; donc, il est classé comme STATIC-DEF.

Numéro de séquence ESP : le numéro de séquence ESP va être incrémenté de un pour chaque paquet envoyé.

A.7 Champs d'en-tête d'extension IPv6

Champ	Classe
Prochain en-tête	STATIC-DEF
Longueur d'en-tête externe	
Acheminement	STATIC-DEF
Bond par bond	STATIC
Destination	STATIC
Options	
Acheminement	STATIC-DEF
Bond par bond	RACH
Destination	RACH

Prochain en-tête : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Longueur d'en-tête externe : pour l'en-tête Acheminement, il est supposé que la longueur va rester constante pour la durée du flux, et qu'un changement de la longueur devrait être classé comme un nouveau flux par le compresseur ROHC. Pour les en-têtes d'option Bond par bond et Destination, la longueur est supposée rester statique, mais peut être mise à jour par un paquet IR.

Options : pour l'en-tête Acheminement, il est supposé que le contenu de l'option va rester constant pour la durée du flux, et qu'un changement des informations d'acheminement devrait être classé comme un nouveau flux par le compresseur

ROHC. Pour les en-têtes d'option Bond par bond et Destination, les options sont supposé rester statiques, mais peuvent être mises à jour par un paquet IR.

A.8 Champs d'en-tête GRE

Champ	Classe
Fanion C	STATIC
Fanion K	STATIC
Fanion S	STATIC
Fanion R	STATIC-KNOWN
Version réservé0,	STATIC-KNOWN
Protocole	STATIC-DEF
Somme de contrôle	IRREGULAR
Réservé	STATIC-KNOWN
Numéro de séquence	PATTERN
Clé	STATIC-DEF

Fanions : les quatre bits de fanion ne sont pas supposés changer pour la durée du flux, et le fanion R est supposé être toujours réglé à zéro.

Version, réservé0, : Ces deux champs sont supposés être réglés à zéro pour la durée de tout flux.

Protocole : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Somme de contrôle : quand le champ Somme de contrôle est présent, il est supposé se comporter de façon imprévisible.

Réservé : quand il est présent, ce champ est supposé être réglé à zéro.

Numéro de séquence : quand il est présent, le numéro de séquence augmente de un à chaque paquet.

Clé : quand il est présent, le champ Clé est utilisé pour définir le flux et ne change pas.

A.9 Champs d'en-tête MINE

Champ	Classe
Protocole	STATIC-DEF
Bit S	STATIC-DEF
Réservé	STATIC-KNOWN
Somme de contrôle	INFERRED
Adresse de source	STATIC-DEF
Adresse de destination	STATIC-DEF

Protocole : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Bit S : le bit S n'est pas supposé changer durant un flux.

Réservé : le champ Réserve est supposé être réglé à zéro.

Somme de cntrôle : l'en-tête Somme de contrôle protège les bonds d'acheminement individuels contre le traitement d'un en-tête corrompu. Comme tous les champs de cet en-tête sont compressés, il n'est pas besoin d'inclure cette somme de contrôle dans les paquets compressés et elle peut être régénérée du côté du décompresseur.

Adresses de source et destination : ces champs peuvent être utilisés pour définir le flux et ne sont pas supposés changer.

A.10 Champs d'en-tête AH

Champ	Classe
Prochain en-tête	STATIC-DEF
Longueur de charge utile	STATIC
Réservé	STATIC-KNOWN
SPI	STATIC-DEF

Numéro de séquence	PATTERN
ICV	IRREGULAR

Prochain en-tête : ce champ va avoir la même valeur dans tous les paquets d'un flux et est donc classé comme STATIC-DEF.

Longueur de charge utile : il est supposé que la longueur de l'en-tête est constante pour la durée du flux.

Réservé : la valeur de ce champ va être réglée à zéro.

SPI : ce champ est utilisé pour identifier un champ spécifique et ne change que quand le numéro de séquence revient à zéro, et est donc classé comme STATIC-DEF.

Numéro de séquence : le numéro de séquence va être incrémenté de un pour chaque paquet envoyé.

ICV : le ICV est supposé se comporter de façon imprévisible et est donc classé comme IRREGULAR.

Appendice B. Lignes directrices pour la mise en œuvre du compresseur

Cette section décrit quelques principes directeurs pour mettre en œuvre un compresseur ROHCv2 en mettant l'accent sur la façon de choisir efficacement les formats de paquet appropriés. Le texte de cet appendice devrait être considéré comme des lignes directrices ; il ne définit aucune exigence normative sur la façon dont les profils ROHCv2 sont mis en œuvre.

B.1 Gestion des références

Le compresseur tient généralement une fenêtre glissante d'en-têtes de référence, qui contiennent autant de références que nécessaire pour l'approche optimiste. Chaque référence contient une description des changements qui se produisent dans le flux entre deux en-têtes consécutifs dans le flux, et une nouvelle référence est insérée dans la fenêtre chaque fois qu'un paquet est compressé par ce contexte. Une référence peut par exemple être mise en œuvre comme une copie mémorisée de l'en-tête non compressé représenté. Quand le compresseur est sûr qu'une référence spécifique n'est plus utilisée par le décompresseur (par exemple en utilisant l'approche optimiste ou les rétroactions reçues) la référence est supprimée de la fenêtre glissante.

B.2 Codage de LSB fondé sur la fenêtre (W-LSB)

Le paragraphe 5.1.1 décrit comment l'approche optimiste impacte le choix du format de paquet pour le compresseur. Il appartient à la mise en œuvre de décider exactement comment le compresseur choisit un format de paquet, mais ce qui suit est un exemple de comment ce processus peut être effectué pour les champs codés en lsb par l'utilisation du codage de LSB fondé sur la fenêtre (W-LSB, *Window-based LSB*).

Avec le codage W-LSB, le compresseur utilise un certain nombre de références (une fenêtre) de son contexte. Quelles références utiliser est déterminé par son approche optimiste. Le compresseur extrait la valeur du champ à coder en W-LSB à partir de chaque référence dans la fenêtre, et trouve les valeurs maximum et minimum. Une fois qu'il a déterminé ces valeurs, le compresseur utilise l'hypothèse que le décompresseur a une valeur pour ce champ dans la gamme donnée par ces limites (incluses) comme sa référence. Le compresseur peut alors choisir un nombre de LSB à partir de la valeur à compresser, de sorte que les LSB peuvent être décompressés sans considération de si le décompresseur utilise la valeur minimum, la valeur maximum ou toute autre valeur dans la gamme des références possibles.

B.3 Codage W-LSB et compression fondée sur le temporisateur

Le paragraphe 6.6.9 définit le comportement du décompresseur pour la compression de l'horodatage RTP fondé sur le temporisateur. Ce paragraphe ne donne pas de directive sur la façon dont le compresseur devrait déterminer le nombre de bits de LSB qu'il devrait envoyer pour le champ Horodatage. Quand on utilise la compression fondée sur le temporisateur, ce nombre dépend de la somme de la gigue avant le compresseur et de la gigue entre le compresseur et le décompresseur.

La gigue avant le compresseur peut être estimée en utilisant le calcul suivant :

Max_Jitter_BC = max {|(T_n - T_j) - ((a_n - a_j) / time_stride)|, pour tous les en-têtes j dans la fenêtre glissante}

où (T_n - T_j) est la différence d'horodatage entre l'en-tête compressé actuel et un en-tête de référence et (a_n - a_j) est la différence d'heure d'arrivée entre ces deux mêmes en-têtes.

En plus de cela, le compresseur a besoin d'estimer une borne supérieure pour la gigue entre le compresseur et le décompresseur (Max_Jitter_CD). Cette information peut par exemple venir des couches inférieures.

Une mise en œuvre de compresseur peut déterminer si la différence de résolution d'horloge entre le compresseur et le décompresseur induit une erreur quand elle effectue une arithmétique d'entiers ; elle peut alors traiter cette erreur comme de la gigue supplémentaire.

Après avoir obtenu les estimations pour la gigue, le nombre de bits nécessaires à transmettre est obtenu en utilisant le calcul suivant :

plafond(log2(2 * (Max_Jitter_BC + Max_Jitter_CD + 2) + 1))

Ce nombre est alors utilisé pour choisir le format de paquet qui contient au moins autant de bits d'horodatage adapté.

Adresse des auteurs

Ghyslain Pelletier
Ericsson
Box 920
Lulea SE-971 28
Sweden
téléphone : +46 (0) 8 404 29 43
mél : ghyslain.pelletier@ericsson.com

Kristofer Sandlund
Ericsson
Box 920
Lulea SE-971 28
Sweden
téléphone : +46 (0) 8 404 41 58
mél : kristofer.sandlund@ericsson.com

Déclaration complète de droits de reproduction