INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

orange™

Stage de fin d'étude d'ingénieur

Attaque, panne ou erreur - Conception d'heuristique de détection dans l'obscurité de l'Internet profondément chiffré

RONTEIX--JACQUET Flavien

Enseignant référent
BERTHOME Pascal

Tuteur Entreprise
HAMCHAOUI Isabelle

Année universitaire **2018/2019**

Remerciements

Je remercie bien évidemment mes encadrants, Isabelle Hamchaoui et Alexandre Ferrieux sans qui ce stage ne se serait pas si bien déroulé mais aussi toute l'équipe ITEQ qui m'a si chaleureusement accueillit. Antoine qui m'a donné de bons conseils, Olivier et Cyril avec qui j'ai partagé un bureau durant 6 mois et Jean-Marc, chef d'équipe qui fait un travail exceptionnel.
J'aimerais aussi remercier Camille, Madeleine et le club robotique qui m'ont soutenu durant ce travail, et Stéphane avec qui les discussions régulières ont grandement amélioré le pôle tactique.
Enfin, j'aimerais remercier l'INSA Centre Val de Loire et en particulier l'équipe pédagogique, sans qui cette thèse de master n'aurait jamais pu voir le jour. Trugarez !

RESUME

La version 3 du protocole HTTP utilisera le protocole QUIC et non plus TCP comme les versions précédentes. QUIC est en cours de standardisation au sein de l'IETF avec une première version attendue pour Automne 2019, mais est déjà massivement déployé sur Internet avec 15 % du trafic en moyenne. Ce nouveau protocole s'adapte à son époque avec notamment une prise en compte importante de la confidentialité et la sécurité des échanges. En chiffrant toute la communication et quasiment tout le paquet avec TLS1.3, QUIC rend opaque des informations précédemment visibles avec TCP/TLS1.1. Ces informations comme le numéro de séquence, étaient très utiles aux opérateurs de télécommunications comme Orange pour débugger les réseaux avec des sondes passives et des outils. Pour pallier cette perte de visibilité, des propositions ont été faites au sein de l'IETF pour ajouter 3 bits visibles d'informations utilisables à des fins de debuggabilité, les bits Spin, sQuare et Retransmit.
Pour prouver l'utilité de ses bits, qui ne sont pas encore dans la norme, il a été développé un testbed d'expérimentation utilisant des piles réseaux Quic open source instrumentées avec les bits, un client, un serveur et un nœud réseau « perturbateur ». Il a également été développé une interface Web de configuration et de visualisation des résultats ainsi qu'une sonde passive de démonstration.
Les résultats obtenus montrent que ces mécanismes fonctionnent et que la sonde avec l'interface d'analyse permet le debugging à partir d'une capture réseaux.
L'implémentation des extensions dans des piles open source et ces résultats seront des arguments pour pousser cette proposition au sein de l'IETF.

Mots-clés : *QUIC, Troubleshooting, IETF, Réseau, Open Source, Netem, Contrôle de congestion, Perturbations, Pertes*

ABSTRACT

Version 3 of the HTTP protocol will use QUIC protocol and no longer TCP as the previous versions. QUIC is being standardized by IETF with a first version expected for Autumn 2019; however, it is already massively deployed on the Internet with an average of 15 % of traffic. This new protocol adapts to its time within particular an important taking account of communications privacy and security. By encrypting entire communication and almost entire packets with TLS1.3, QUIC makes previously visible information in TCP/TLS1.1 completely dark. This information like the sequence number, was very useful for telecommunications companies as Orange to debug networks with passives probes and powerful tools. To fix this loss of visibility, proposals have been made within the IETF to add three clear bits for debuggability purposes, the Spin, sQuares and Retransmit bits.
To prove the usefulness of its bits, which are not yet in the drafts, an experimentation testbed was developed using Quic open source instrumented stacks with the bits, a client, a server and a network node called "perturbateur". A web-based configuration and results visualization has also been developed, alongside with a passive demonstration probe.
The results show that these mechanisms work and that the probe with the analysis interface allows us to debug from a network capture.
The extensions' implementations in open source stacks and these results will be arguments to push this proposal within QUIC standard.

Keywords: *QUIC, Troubleshooting, IETF, Network, Open Source, Netem, Congestion control, Losses*

Sommaire

Introduction

Les révélations d'Edward Snowden de 2013 ont suscité une vaste prise de conscience en Occident. L'informatique, et Internet sont apparus d'un coup non plus simplement comme des outils de la modernité, mais aussi et surtout comme un outil au service d'une surveillance généralisée à la main d'agences de renseignement des plus grands pays du monde. Ainsi La NSA, la CIA, avec l'aide de législation américaine, ont pu mettre la main sur les données de million d'américains, mais aussi d'étrangers, ou des dirigeants du monde entier, et ce, chaque jour.

Dès lors, une épidémie de "cache webcam" sur les ordinateurs se sont répandus auprès des utilisateurs pour protéger sa vie privée et le mot "Privacy" a commencé à être utilisé partout.
En premier lieu chez les fournisseurs de service sur Internet. En effet, comment ne pas penser aux giga-octets de données stockées, que nous utilisateurs offrons aux fournisseurs de service qui ont été et qui sont probablement utilisées par ces programmes Américains comme "Prism" ? Ces données, dont finalement nous n'avons que peu conscience de leur importance, dont nous croyons qu'elles sont bien gardées, bien personnelles et que personne ne peut utiliser. Après Snowden, les Google, Facebook, Microsoft ont compris que leurs utilisateurs voulaient avoir confiance en la sécurité de ces données, mais aussi en leur "privacy", ou être sûr que le gouvernement ne puisse pas récupérer toutes ces données à leur insu, et même, à l'insu de ces fournisseurs de service.

Internet raccorde des milliards d'utilisateurs de par le monde via l'interconnexion de centaines de réseaux. Ces réseaux, gérés par des entreprises de télécommunication, s'échangent et voient circuler des quantités gigantesques de données. Une aubaine pour les agences de renseignement qui peut donc assez facilement (avec la collaboration ou pas des gestionnaires du réseau) aspirer ces données et en déduire, voire beaucoup de chose. Adresses IP source, destination, protocole utilisé, contenu dès fois, métadonnées tout le temps.

Face à cette problématique de la confidentialité des communications, il existe depuis le début des années 2000 un protocole pour chiffrer les données de l'utilisateur, TLS, qui couplé avec HTTP sur TCP/IP représente 80 % du trafic d'Internet. Cependant, TCP avec tout son entête en clair qui comporte beaucoup de "métadonnées", source d'information non-négligeable quant à la connexion, ne permettait plus de garantir un niveau élevé de confidentialité. Dans l'optique de le remplacer (pour d'autres raisons, comme le fait que les technologies ont beaucoup évolué depuis les années 70), beaucoup de technique, de protocole a émergé, mais un est aujourd'hui un candidat sérieux au remplacement de la pile web classique à base de TCP, il s'agit du protocole **QUIC** proposé par Google à partir de 2013.

QUIC est un "protocole profondément chiffré" qui vise à remplacer TCP en lui apportant un haut niveau de performances et de confidentialité. QUIC est fonctionnellement très proche de TCP auquel il ajoute un niveau de chiffrement inédit, anonymat et non-traçabilité obligent. Bien sûr Quic propose bien autres améliorations, d'autant plus qu'il n'a pas été conçu dans l'optique de tout vouloir chiffrer, mais c'est cette propriété qui nous pose problème ici. Le chiffrement complet de Quic est un problème pour les opérateurs comme Orange. Car Orange est un responsable d'un réseau mondial, il a la nécessité de garantir une garantie de service optimale. Avec des protocoles comme TCP avec un entête visible et bien fourni (contenant notamment des informations sur les pertes), il est aisé d'extraire de l'information pour détecter et réparer les pannes. Orange le fait tous les jours avec des équipes dédiées à la télémétrie (le monitoring) et au débogage (le troubleshooting). Avec Quic, tout change puisque que nous n'avons plus accès à aucune information de ce type, compromettant la qualité de service du réseau.

D'où la problématique posée par le sujet de ce stage, ***"Attaque, panne ou erreur ? Conception d'heuristique de détection dans l'obscurité de l'internet profondément chiffré".***
Le travail se compose de deux parties, la première sur une étude bibliographique sur le troubleshooting, le debugging réseau, le contrôle de congestion et sur Quic avec son processus de normalisation au sein de l'IETF. La seconde partie présentera les expérimentations qui ont été menées avec les solutions et résultats obtenues.
Une brève présentation d'Orange Labs ainsi que des travaux annexes au sein du site pendant le stage sont aussi présentés en toute première partie et en fin de rapport.
Nous terminerons avec la conclusion sur le travail effectué.

I. Présentation d'Orange Labs

A. Historique

L'opérateur France-Télécom-Orange appelé Orange depuis Juillet 2013 est la première entreprise de télécommunication française avec plus de 152.000 salariés et un chiffre d'affaires en 2018 de 41.38 G€. Orange développe son activité dans le monde du mobile, des communications fixes, d'Internet, des réseaux, de la recherche et du développement. Les services sont proposés aux particuliers comme aux entreprises. Un des principaux acteurs de télécommunication mondial, Orange propose ses services à 226 millions d'utilisateurs à travers 35 pays différents et a pour Président Directeur Général M. Stéphane RICHARD.

B. La division Orange Labs Networks

Orange Labs est la division de recherche et développement du Groupe Orange. Avec plus de 5.000 chercheurs, scientifiques et ingénieurs regroupés dans 18 centres à travers le monde, la division a pour ambition de garantir l'efficience du socle technique du groupe. Orange Labs se divise en 4 entités :

1. IMT (Innovation, Marketing and Technologies)
2. OLS (Orange Labs System)
3. OLR (Orange Labs Research)
4. **OLN** (Orange Labs Networks)

Orange Labs Networks est l'entité qui organise la Recherche et développements dans les domaines du réseau et de l'infrastructure. Elle possède 26 centres de recherche dans 7 pays, dont 11 en France. Cette entité se divise elle-même en 13 divisions, une pour chaque mission de l'entité. 3 sont très spécifiques au réseau, Radio Networks and Microwaves (RNM), Convergent Networks Control (CNC), et WNI (Wireline Networks and Infrastructure).

C. Mon service : TGI/OLN/WNI/IPN/ITEQ

J'ai effectué ce stage de fin d'étude dans l'équipe (par ordre hiérarchique) :

1. TGI (Technology and Global Innovation)
2. OLN (Orange Labs Network)
3. WNI (Wireline Networks & Infrastructures)
4. IPN (IP Networks, Monitoring & Security)
5. **ITEQ** (Innovation Towards Enhanced QoS)

Cette équipe se situe sur le site d'Orange Labs à **Lannion** (22, Côte d'Armor), en Bretagne.

Orange Labs Lannion, autrefois connu comme le **CNET** (Centre National d'Etudes des Télécommunications) est un haut lieu de la recherche d'Orange, ex-France Telecom, ex-PTT. Il a été fondé en 1963 à l'initiative de Pierre Marzin dans un objectif de recherche et de décentralisation. C'est sur ce site que la première communication satellite à travers l'Atlantique fût établie, c'est aussi le site de conception du premier commutateur téléphonique électronique temporel PLATON, des premiers tests fonctionnels de la fibre optique et surtout de la conception de la fierté française, le Minitel.

Le site d'aujourd'hui représente 40 % des activités de recherche d'Orange Labs Network, d'Orange Labs Systems. Il est aussi le lieu des activités de vente avec la division Orange Business Services et des activités plus opérationnelles avec la DTSI (Direction Technique du système d'Information) et avec WIN (Wholesale & International Networks) / OINIS (Orange International Networks Infrastructures & Services).

Toutes ces activités regroupent plus de **1100** salariés et 100 étudiants (thésards, apprentis, stagiaires) sur près de 31 hectares dans un cadre naturel exceptionnel (au cœur du Trégor, et de la côte de granite rose). Autour du site un cadre très dynamique d'entreprises technologiques et de télécommunications avec notamment un centre de recherche de Nokia et d'Ericsson et de nombreuses entreprises expertes de la fibre optique. Le pôle de compétitivité reconnu dans le monde,

« Images et réseaux » regroupe 170 entreprises et 25 laboratoires et école de la région autour de ses thématiques du réseau, de la fibre et de l'informatique.

L'équipe **ITEQ** qui comprend une quinzaine de chercheurs et d'ingénieurs a pour objectif d'améliorer la qualité de service au sein des réseaux IP d'Orange. Elle effectue différents travaux de recherche et développement qui se regroupe en 3 thématiques :

1. Les **architectures réseaux innovantes** (SDN, NFV, Ipv6)
2. La **validation et intégration** (Automatisation)
3. Le **monitoring et troubleshooting** des réseaux IP de bout en bout

II. Le troubleshooting et QUIC

A. Sur le « le troubleshooting » et le « debugging » réseau chez Orange

Au sein de mon équipe IPN/ITEQ, l'objectif est d'améliorer la Qualité de Service des réseaux IP d'Orange. Cette tâche rassemble différents métiers dont celui de monitoring (surveillance) et de troubleshooting (dépannage) réseau.
La **qualité de service** (QoS) dans les réseaux se définit par différents paramètres :

- le temps d'établissement de la connexion
- la probabilité d'échec de l'établissement de connexion
- le **débit de la liaison**
- le **temps de transit** (RTT)
- le **taux d'erreur résiduel**
- la probabilité d'incident de transfert
- la confidentialité de la connexion
- la **résilience**
- Et bien d'autres !

Le **troubleshooting** réseau est un travail important pour assurer la qualité de service, car c'est ce travail qui vise à rétablir la normalité dans le réseau après une panne, une erreur ou même une attaque. Le monitoring est la méthode proactive qui permet de localiser et informer en cas de problème sur le réseau. Pour ces métiers, il est nécessaire d'avoir des outils, qui permettent de mener ce travail d'investigation, de compréhension pour ensuite proposer des solutions. Une partie de l'équipe travaille précisément sur le développement de ses outils et la compréhension des bugs.

Orange possède un réseau très grand et hétérogène (En Fig1 l'OTI [1]d'Orange), dans plusieurs pays avec différentes technologies, différents équipements, bref, le monitoring et le troubleshooting sont nécessaires pour maintenir un haut niveau de QoS[2].
Les outils se présentent sous 2 formes, **les sondes passives** et les **sondes actives**. Les premières s'appellent « **diagnet** » et servent à observer depuis des équipements réseaux (pour faire des captures notamment). Par recherche dichotomique (déplacement du point de mesure le long d'une connexion), elles permettent de détecter les nœuds à problèmes.
Les secondes sont les "**Lili**" (Linux for Local Investigation), des mini-pc sous Linux répartis un peu partout sur le réseau et qui permettent de la mesure active. Elles peuvent être connectées comme un client par réseau filaire ou 4G, mais aussi être connecté directement à un équipement réseau (mode plier) ou à un point de peering (mode peering). La mesure est active car ces sondes génèrent du trafic pour faire de la mesure et permettre aux sondes passives de trouver les nœuds à problèmes.

[1] Open Transit Internet
[2] Qualité de service [32]

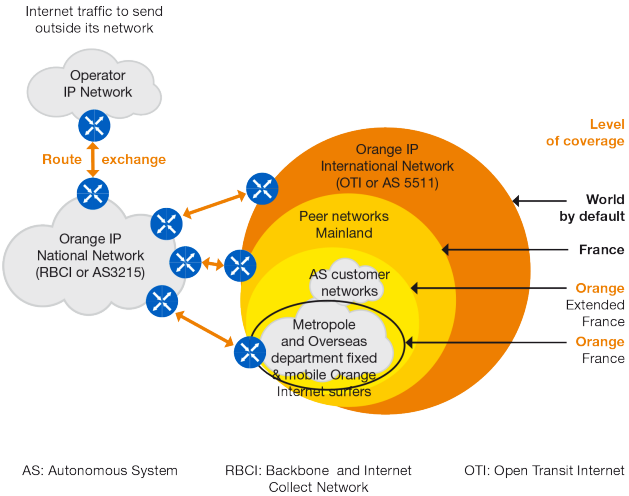


Fig1 : l'architecture de la solution Open Transit Internet d'Orange pour l'internet mondial d'Orange

Ces Lili servent aussi à remonter des données quant au réseau dans lequel elle se trouve (débit, latence, ...) vers lili centrale et son interface web pour assurer un monitoring en continu.
Sur les réseaux d'Orange et des opérateurs, la majorité des données des clients finaux qui circulent sont des données pour un service Web/Internet qui utilise la pile réseau (selon le modèle OSI[3]) :

- Couche 7 : **HTTP1.1/1.2** [4] pour le contenu
- Couche 5 : **TLS1.2/SSL** [5] pour la sécurité et la confidentialité du transport de la donnée
- Couche 4 : **TCP** [6] pour le transport avec un mécanisme de garantie de livraison
- Couche 3 : **IP** [7] pour le transport sur le réseau internet

Avec les entêtes TCP et IP qui nous sont visible, il est possible d'avoir une bonne idée de l'état du réseau. Certaines options de l'entête TCP (cf Fig2 extraite de la RFC793) sont particulièrement pratiques pour cette tâche.

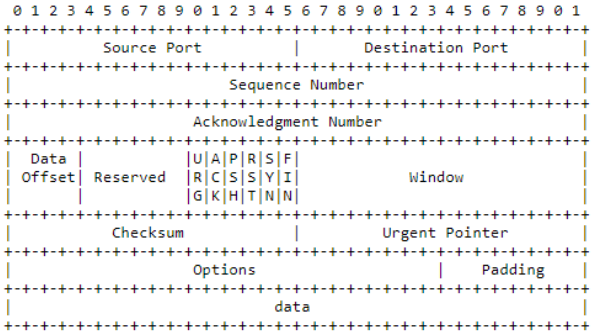


Fig2 : L'entête du protocole TCP

Le "**Sequence Number**" (numéro de séquence) nous donne la place du paquet dans le train de paquet, le "**Acknowledgment Number**" (numéro d'acquittement) permet de connaître le numéro du dernier paquet acquitté par le bout. La "**Window**" (fenêtre de congestion) permet de connaître le crédit de donnée que le bout est autorisé à recevoir et enfin les "flags" (drapeaux) ACK, SYN, FIN permettent de connaître le sujet du paquet et donc l'état de la connexion.
De ces informations, il est possible détecter les "trous" dans le flux de paquet et donc localiser la provenance des pertes, mais pas seulement, nous pouvons aussi comprendre le comportement des nœuds réseau (répartition des flux, débordement des caches), des piles à chaque bout... Bref acquérir beaucoup d'information pour comprendre, réparer et améliorer le réseau, et ce, avec peu d'information visible et des sondes placées intelligemment sur les routes.
Tout ça sans avoir accès au contenu des paquets ni compromettre l'identité des bouts.

[3] Open System Interconnection, norme de communication en réseau par l'ISO
[4] Hypertext Transfer Protocol
[5] Transport Layer Security, protocole de chiffrement
[6] Transmission Control Protocol
[7] Internet Protocol

Il est nécessaire de faire du troubleshooting car les réseaux ne sont pas parfaits. Ils peuvent comporter des erreurs, ils peuvent être dans un état de congestion....
Nous le verrons plus loin en détail dans une étude théorique puis pratique de la nature de ces perturbations.

Sur la gestion des capacités d'un réseau, le "congestion control"

TCP (Transmission Control Protocol), à la différence d'**UDP** (User Datagram Protocol) garantie la livraison des données (il est dit en mode "connecté"). Pour réaliser cette tâche, il possède toute une série de mécanisme qui permet de s'assurer que toutes les données (tous les octets au-dessus de lui dans les couches du modèle OSI) ont bien été reçues, tout en permettant un débit maximal (ce que souhaite l'utilisateur). Pour atteindre ce débit maximal possible tout en garantissant la livraison, le mécanisme de **contrôle de congestion [1]** est important. À l'issu de nos recherches, nous avons remarqué que 4 algorithmes sont majoritairement utilisés par la pile TCP [27]. Ils utilisent des mécanismes de détection différents et ont des bénéfices aussi différents.

Algorithme	Feedback	Bénéfices
Reno, NewReno	Pertes	Meilleur débit
Vegas	Délais	Moins de pertes
Cubic	Pertes	Meilleur débit
BBR	Délais	Moins de débordement de buffer

Tab1 : Les algorithmes de congestion

Ces algorithmes contrôlent la fenêtre de congestion, c'est-à-dire le crédit que la pile a pour émettre des données (ainsi que la fréquence d'émission des paquets, …). Pour simplifier, la fenêtre de congestion va indiquer la quantité de données qui est autorisée à être "inflight" (en vol) avant d'être acquittée.

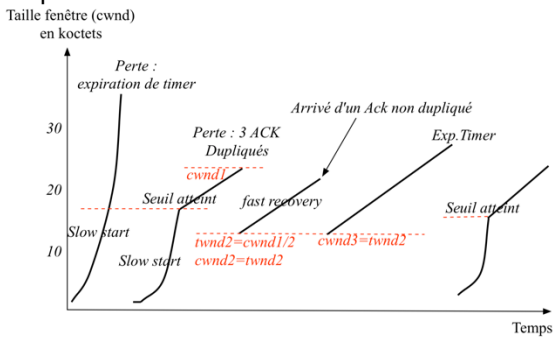


Fig3 : Fonctionnement de l'augmentation de la fenêtre de congestion pour l'algorithme NewReno

Dans un réseau bien dimensionné, la fenêtre va être grande alors qu'au contraire dans un réseau congestionné, cette fenêtre va être réduite pour ne pas prendre le "risque" de perdre. Ci-contre la fenêtre de congestion en fonction du temps et des évènements réseaux pour l'algorithme newReno. Nous y distinguons plusieurs comportements, un démarrage rapide avec beaucoup de données en vol, puis une phase de modération et une phase de « récupération » après avoir perdu des données.

En pratique, l'utilisation d'un algorithme plutôt qu'un autre par une pile amène à des comportements du réseau différents avec soit des délais observés plus élevés ou des buffers qui se remplissent sans pouvoir se vider. Il est nécessaire pour notre problématique d'avoir les notions préliminaires pour comprendre les comportements que nous allons observer sur le réseau au moment de l'analyse.

Les contrôles de congestion intègrent généralement les mécanismes de **Slow Start** ou **d'ECN**. Un mécanisme est intéressant à comprendre pour notre problème, le **pacing** (Fig4). Ce mécanisme fluidifie l'écoulement des paquets permet d'éviter les burst (rafale) de paquet, burst qui occasionne des congestions dû à des pics de charge.

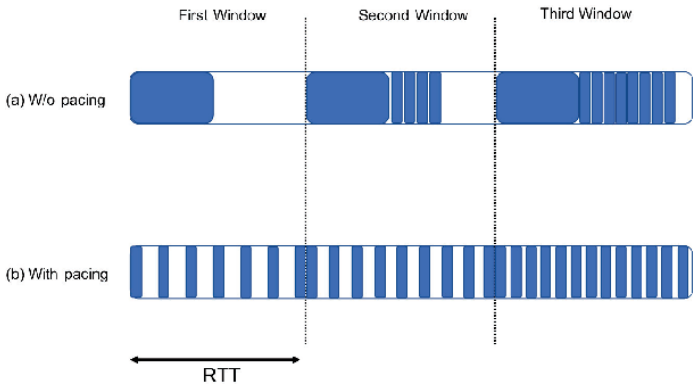


Fig4: Les effects du pacing [2]

Comme nous l'avons vu, avec TCP et les informations qu'il nous laisse voir à travers les captures sur le réseau, le troubleshooting est réalisable assez facilement et des outils toujours plus complexes permettent de comprendre au mieux les comportements des réseaux et donc améliorer la Qualité de service. Or, en 2013, Google propose un nouveau protocole, Quic, avec des propriétés assez différentes et dans le même temps semblable à TCP qui change la physionomie des observations du réseau.

B. Sur le protocole QUIC

1. Introduction à QUIC

Le protocole **Quic** (qui n'est plus un acronyme, au départ pour "Quick UDP Internet Connections", et se prononce comme le mot "quick" en anglais) a été proposé comme dit plus tôt par l'entreprise Google en 2013 [3]. Même s'il vise à remplacer TCP dans le transport de la donnée, il est en réalité basé sur le protocole UDP (couche 4). UDP assure le transport vers le bon port ainsi que le checksum (somme de contrôle) de la payload Quic. Quic peut être considéré comme un protocole hybride en comparaison avec la pile réseau classique. Présent en couche 4.5 via le contrôle de congestion et le mécanisme de retransmission (qu'UDP n'intègre pas), 5 via la cryptographie et 6 via HTTP-over-Quic et les Streams.

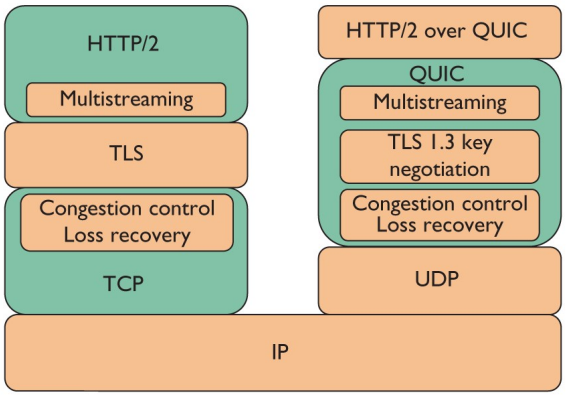


Fig5 : Les piles TCP vs Quic selon le modèle OSI (issu de 3g4g.co.uk)

Google a défini différents objectifs dans la perspective de pallier aux manquements de TCP. En effet TCP est un protocole vieilli (la première version date 1974) au regard des dernières innovations et technologies dans le domaine du réseau. Quic a également l'objectif de devenir la base de la prochaine version du protocole HTTP**, HTTP/3** (HTTP/2 étant basé sur le protocole SPDY, peine à se répandre à cause d'un problème de conception, le HoL[8]).

Réduire la latence des communications d'une part est important pour les nouveaux usages d'Internet comme avec l'internet des objets et d'autre part augmenter les débits pour les usages de streaming

[8] Head of Line blocking

vidéo et de cloud computing. Le tout avec un chiffrement important toujours dans l'optique de protéger les communications.

L'intérêt pour les serveurs utilisant Quic est d'offrir un meilleur service ainsi qu'une garantie au client d'avoir une communication chiffrée de bout en bout dès les premiers échanges. Le revers de la médaille, le coût en calcul d'une connexion Quic pour les serveurs à cause du chiffrement important.

Les rapports qu'entretiennent les différents acteurs avec Quic sont les suivants

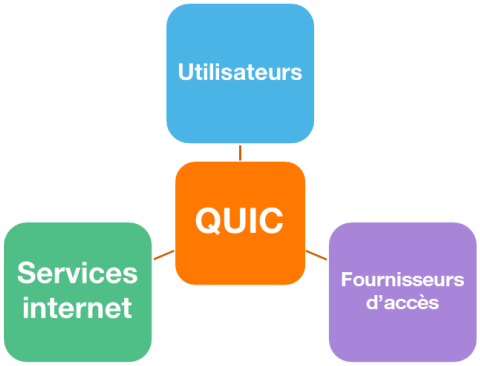


Fig6 : Un rapport différent à Quic

1- Les fournisseurs de services veulent un meilleur service avec des débits plus élevés, des latences plus faibles et des échanges plus sécurisés pour assurer la satisfaction du client

2- Les fournisseurs d'accès à Internet veulent pouvoir surveiller et diagnostiquer ce qu'il se passe sur leur réseau pour augmenter la qualité offerte aux clients tout en réduisant les coûts

3- Les utilisateurs veulent une meilleure qualité de service (qui se mesure en général en débit), une simplicité d'utilisation et de la confidentialité.

Ces volontés sont souvent antagonistes, en particulier entre fournisseurs de service et fournisseurs d'accès qui ne considèrent pas le réseau de la même façon.

2. Historique d'utilisation

Au départ déployé en expérimentation sur quelques serveurs de Google, Quic et en particulier **gQuic** s'est énormément répandue depuis 2017. En effet, c'est à partir de cette date que Quic a été activé par défaut dans le navigateur Google Chrome mobile et desktop (environ 2/3 de part de marché) et proposé par la majorité des services de Google, en particulier la recherche (Google Search), et le streaming vidéo (Youtube).

D'autres serveurs proposent l'utilisation de Quic comme ceux de Cloudflare et Akamai (CDN[9] de différents gros sites comme le site de mise à jour d'Apple ou même certain contenu Netflix). La télémétrie Orange nous indique qu'en moyenne **15** % des sites les plus consultés d'Internet répondent en Quic et que 22 % sont capable d'utiliser Quic si le client l'exige.

L'augmentation du trafic Quic est très importante avec en moyenne **15 %** sur un point de mesure MAWI fournie par **netray.io** [4] et en constante augmentation (MAWI pour Measurement and Analysis of Wide-area Internet est un groupe d'étude japonais qui cherche à faire de la mesure et de l'analyse du trafic sur l'Internet mondial).
Dans les réseaux d'Orange nous observons aussi une quinzaine de pour-cent de trafic Quic avec une forte disparité entre les pays, entre les supports (mobile, fixe, pro), au cours de la semaine et même au cours de la journée (probablement dû à une utilisation accrue des services de streaming vidéo en soirée).

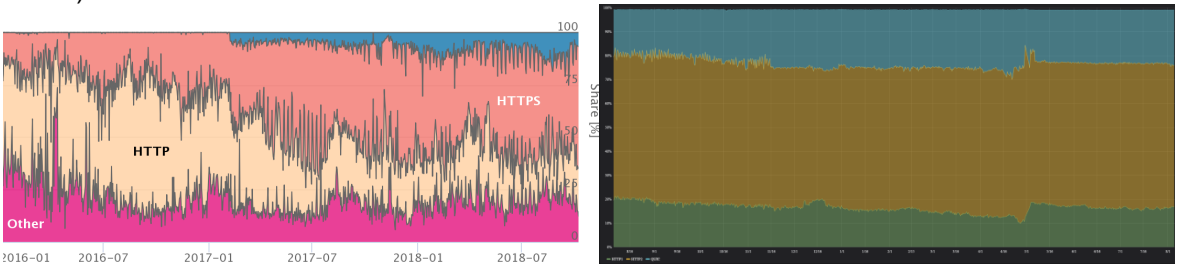


Fig7a : Utilisation sur le point de mesure MAWI (Quic en bleu)

[9] Content Delivery Network

Fig7b : Taux d'adoption de Quic par les serveurs selon les mesures d'Orange (Quic en bleu)

Il est fort à parier qu'avec la première version finale du protocole prévue pour cette année, la normalisation de HTTP/3 et l'utilisation par les GAFAM, la part de Quic dans le trafic de l'internet mondial va augmenter encore fortement dans les prochains mois et les prochaines années.

3. L'IETF QUIC Working Group

Le protocole Quic a été proposé, développé, et déployé par **Google dès 2013**. Il a été déployé très rapidement par la firme grâce à l'implémentation de la pile dans l'espace utilisateur (et non dans le noyau, dans le "kernel-space" comme pour TCP). Le projet Chromium a permis de toucher une large population d'utilisateur avec le navigateur Chrome et les nombreux services du géant californien.

Assez rapidement, dès 2016, l'IETF[10], organisme de normalisation de tout ce qui touche i
Internet et les réseaux s'est attelé à proposer une norme qui puisse être largement déployée sur tout internet. **L'IETF Quic Working Group** était créé. Depuis lors, ce groupe d'une centaine d'acteurs se propose de normaliser le protocole avec comme objectif de sortir Quic version 1 à l'automne 2019.

Le groupe rédige les **drafts** (brouillon) pour les futurs RFC[11], organise les groupes de discussion (le groupe de mail Quic est la principale source d'information), organise des conférences (meeting, comme celui du 20-26 Juillet 2019 à Montreal) et coordonne les implémentations. En référence, la liste des liens vers les groupes de travail IETF Quic [5,6,7,8].

Dans ce groupe il y a beaucoup de participants qui observent les directions prises par le groupe mais il y a surtout des entreprises et acteurs important d'internet très actif (même si j'ai retrouvé étonnamment peu d'opérateur dans les membres très actifs) comme présentés en Tab2 (une sélection purement subjective en fonction des contributeurs que nous avons souvent rencontrés dans les discussions)

Nom (pays)	Employeur	Rôle & notes
Martin Notthingham (Au)	Fastly	Directeur du groupe HTTP et Quic à l'IETF
Lars Eggert (Fi)	NetApp	Contributeur et dirigeant très actif à l'IETF
Martin Thomson (Au)	Mozilla	Principal rédacteur et mainteneur des draft et du repository git
Brian Trammell (Ch)	Google	Inventeur du spinbit et contributeur important
Robin Marx (Be)	Université Hasselt	Développeur de Quicker, inventeur des Qlog qui travaille beaucoup sur le troubleshooting de Quic
Christian Huitema (Fr)	Octopus (ex-Microsoft)	Contributeur influent à l'IETF, développeur de Picoquic
Kazuho Oku (Jp)	Fastly	Co-rédacteur de standards HTTP et Quic
Jan Rüth (De)	Université Aachen	Créteur du site netray.io

Tab2 : contributeurs principaux et suivie pendant ce stage au groupe de travail Quic IETF

Les documents liés à la définition de Quic sont disponibles sur le site du WG [8] et se compose ainsi :

- **Applicability of the Quic Transport Protocol** : Document informatif sur le déploiement du protocole Quic en tant que protocole d'application.
- **Hypertext Transfer Protocol v3** : Le standard proposé pour HTTP/3, HTTP-over-Quic qui a été annoncé en novembre 2018
- **Version-Indepandant properties of Quic** : Standard des invariants Quic entre les versions et les drafts
- **Manageability of the Quic Transport Protocol** : Document informatif sur la gestion du protocole, en particulier pour les opérateurs réseaux impliquant du Quic. Il a notamment été coécrit par B. Trammell qui est l'inventeur du spin bit.
- **QPACK : Header compression for HTTP/3** : Standard pour la compression d'entête HTTP/3

[10] Internet Engineering Task Force
[11] Request For Comment, document faisant force de norme [24]

- **Quic Loss Detection and congestion control** : Standard qui décrit comment générer les acquittements, estimer le RTT, détecter les pertes et faire du contrôle de congestion

- **Using TLS to secure Quic** : Le standard de description de l'utilisation de TLS1.3 pour le chiffrement de Quic

- et le plus important, Quic : **A UDP-based multiplexed and secure transport protocol** qui décrit le protocole Quic de manière générale, le format des entêtes, les frames, streams, ... Ce document est la référence principale lors d'une implémentation de Quic selon le standard IETF.

La réunion 105 de **l'IETF à Montréal du 20 au 26 juillet** a permis de montrer par les interventions d'Alexandre Ferrieux (Orange) et d'Igor Lubashev (Akamai) aux participants de l'importance du troubleshooting de Quic par des exemples puis de proposer l'extension des bits d'extension QR. Le travail de "lobbying" s'est effectué au travers du hackathon d'implémentation des piles Quic (avec des exemples utilisant les extensions), de la conférence HotRFC, maprg (Measurement and Analysis for Protocols), tsvwg (transport Area working group) et d'un "side-meeting" pour discuter avec les personnes intéressées.

De ces documents qui arrivent à maturation à l'approche de la première publication de la RFC, nous avons une vision proche de ce à quoi ressemblera le protocole Quic v1 final. Observons dans la prochaine partie les mécanismes principaux, et en particulier ceux qui vont nous intéresser pour notre problème.

4. Fonctionnement dans le détail

Il est noté que nous décrivons le fonctionnement de Quic tel que décrit dans les dernières **draft-22** IETF de juillet 2019. Il faut savoir que la version de Quic la plus utilisée aujourd'hui est la version de Google, gQuic tout simplement car il s'agit de la version déployée sur tous les services, systèmes et logiciels du fournisseur de contenu. Tant que la première version de Quic ne sera pas standardisée, la version IETF restera minoritaire. Toutefois, gQuic cherche à converger vers la version IETF au fur et à mesure que celle-ci s'approche d'une version définitive et les implémentations de Quic par Facebook, cloudflare ou encore indépendantes permettront au plus grand nombre de serveurs de passer à Quic. Par ces 2 faits, il nous paraît plus intéressant et pérenne de travailler sur la version IETF que la version de Google.

Etablissement d'une connexion

Quic dans sa recherche de la réduction des latences et des communications possèdent 2 modes de fonctionnement qui différent de TCP, le mode "1-RTT" et "0-RTT".
Pour rappel, la vie de d'une **connexion TCP** classique débute avec un "triple handshake" (Fig8 En bleu, le handshake, en vert, la communication cryptographique, en noir les requêtes et réponses utiles), donc 3 échange, SYN, SYN+ACK, ACK suivi de la communication en tant que tel avec des ACK réguliers et se termine avec le message FIN. Si nous utilisons TLS pour chiffrer en plus la communication, l'établissement de communication devient beaucoup plus long puisqu'il nécessite 3 RTT minimum. **RTT** pour Round-time trip est le temps nécessaire à un échange complet Client-Serveur-Client. Il est clair que pour réduire le temps d'établissement d'une connexion, il est nécessaire de réduire le nombre d'échanges nécessaire avant la transmission des données.

En **mode 1-RTT** (Fig9) qui est le cas classique de premier établissement de connexion entre un client et un serveur en Quic, un seul aller-retour est nécessaire avant l'émission des premières données utiles. Durant cette phase, la version de Quic est négociée, un ID de connexion est assigné et une sécurité cryptographique est mise en place. Ensuite, la connexion vue de l'extérieur est un simple échange de paquet UDP avec une payload Quic quasiment entièrement chiffrée et complètement authentifié (ce qui est plus fort qu'un checksum qui peut être recalculé, là sans les clefs, il est impossible de porter atteinte à l'intégrité du paquet)

Le mode le plus intéressant avec Quic est le **mode 0-RTT**, très attendu pour certains cas d'usage comme l'IoT, il s'agit du mode lorsque le Client et le Serveur ont déjà communiqué dans un passé pas si lointain. Dans ce cas, le client réutilise les configurations précédemment utilisées pour envoyer une requête au serveur déjà chiffrée et paramétrée. Si le serveur peut vérifier les certificats, les configurations, ... Il va régénérer des clefs de session et va répondre à la requête (sinon un REJ est

émis pour demander une nouvelle initialisation). Ce mode permet d'envoyer une requête au serveur très rapidement, réduisant la latence.

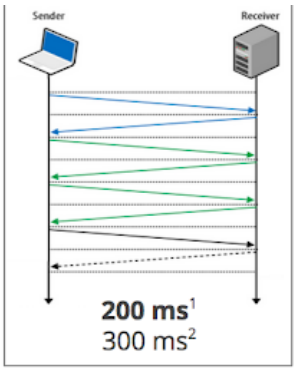


Fig8 : Connexion TCP-TLS (1 si pas de reprise de connexion)

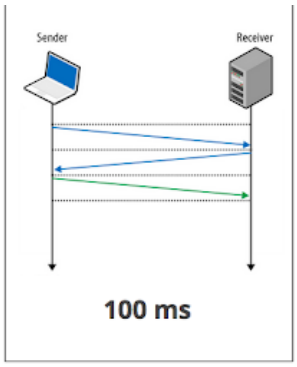


Fig9 : Etablissement d'une Connexion Quic en 1-RTT

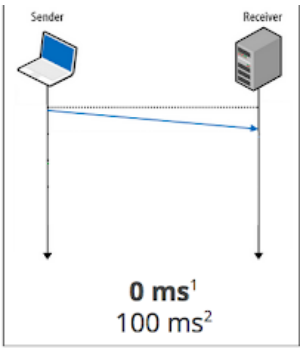


Fig10 : Reprise d'une Connexion Quic en 0-RTT (2 si changement de paramètres du serveur)

Dans tous les cas pour Quic, La connexion est toujours identifiée de manière unique par un 5-tuple (utile pour les répartiteurs de charge, les NAT) composé de

$$\{IP_{source}, IP_{dest}, Port_{source}, Port_{dest}, Connection\ ID\}$$

HTTP/3

En novembre 2018, l'IETF a annoncé que la prochaine itération du protocole du web HTTP/3 fonctionnera au-dessus de Quic. Cette nouvelle version de HTTP est très attendue car elle apportera des fonctionnalités attendues de longue date comme le **mutistream** ou un nouveau système de compression. La Fig11 permet de bien comprendre l'évolution de HTTP depuis la première itération (Issues de la présentation de C. Pearce [9]).

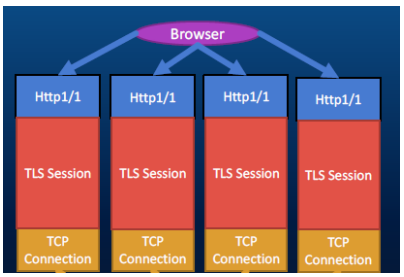


Fig11a : HTTP1/1 over TCP

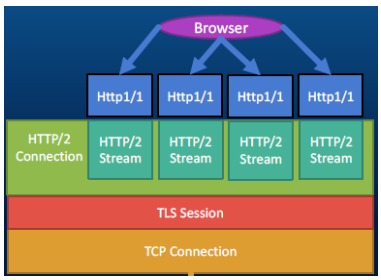


Fig11b : HTTP/2 over TCP

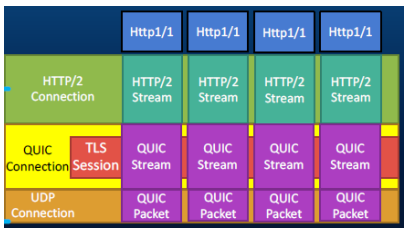


Fig11c : HTTP/3 over Quic

Quic fonctionne au-dessus du protocole **UDP** comme dit plus tôt. Pour rappel, l'entête UDP est extrêmement simple et ne permet que de transporter un datagramme (qui sera ici le protocole Quic et tout ce qu'il y a au-dessus).

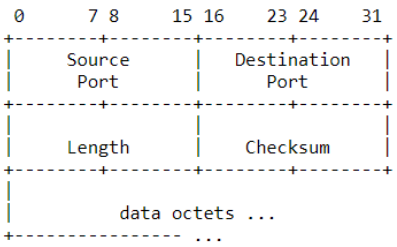


Fig12 : Protocole UDP définie par la RFC768

Le **format** du paquet Quic est présenté en Fig13

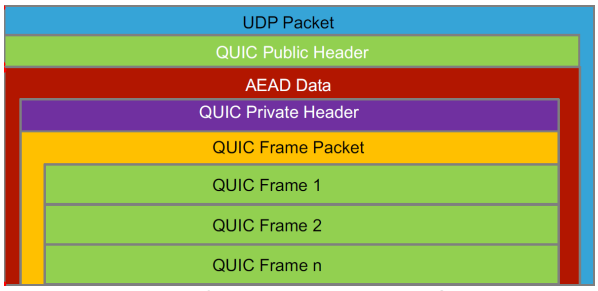


Fig13 : format d'un paquet Quic

Avec un entête public lisible mais authentifié et le reste chiffré et authentifié. (AEAD[12], propriété de TLS1.3)

L'entête public Quic est différent en fonction de l'état de la connexion.
Il existe d'abord l'entête "**Long Header**" qui peut être de 4 types :

- **Initial** : à l'initialisation de la connexion
- **0-RTT** : à la reprise de connexion
- **Handshake** : pour le handshake cryptographique
- **Retry** : nouvelle tentative de connexion

Ces 4 types nous montrent que les paquets Quic avec LH sont utilisés à 90 % en début de connexion.
Le point commun de tous ces types est le format de l'entête commençant par "0b11".

L'autre format est l'entête "**Short Header**" qui est utilisé au cours de la connexion et représente donc plus de 95 % des paquets d'une connexion Quic. C'est une entête courte par rapport à ce qu'il existe avec d'autres protocoles avec seulement 9 octets en moyenne et 25 au maximum (20 octets pour TCP).
La structure est la suivante (dans la dernière draft-22, en annexe 1, l'évolution du premier octet du paquet short header Quic) :

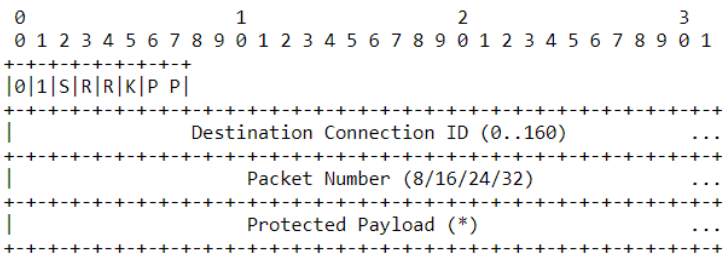


Fig14 : format de l'entête « Short Header » Quic

S : le Spinbit, présenté en C)1.
R : des bits réservés à l'expérimentation
K : la phase de clef pour la cryptographie
P : la longueur du Packet Number
Destination Connection ID : ce nombre est fondamental, il s'agit du seul invariant d'une connexion Quic, il permet au serveur (et au middleboxes du réseau) de suivre une connexion. Cet Id permet notamment d'être résilient à la migration de port, et même à la migration d'adresse IP. C'est une fonctionnalité très intéressante notamment pour les appareils mobiles car ceux-ci peuvent changer d'adresse IP dû au changement d'antenne d'attache, avant, la connexion se coupait. Maintenant, le serveur comprend que le client change d'IP sans couper la connexion. Cet id permet aussi de faire du load balancing dans les grands centres de cloud computing avec de la migration de connexion entre VM. L'id est négocié au début de la connexion, doit être unique (d'où le fait qu'il soit codé au minimum sur 64 bits).

[12] authenticated encryption with associated data

Le **packet number** n'est pas à confondre avec le sequence number TCP, puisque le PN ne peut qu'augmenter au cours de la connexion (une retransmission nécessite la fabrique d'un nouveau paquet avec un nouveau numéro de paquet) à la différence du sequence number TCP représente le numéro de segment dans le train de paquet et donc, peut être retransmis si considéré comme perdu.

Viennent ensuite les données chiffrées et authentifiées utilisant l'AEAD de TLS. Dans ces données qui ne sont donc pas visibles pour un observateur du réseau, se retrouve un entête sensiblement équivalent à celui de TCP ainsi que les "Frame".

Les **frames** finalement sont les éléments les plus fondamentaux de Quic et contiennent toutes les données à transmettre. Il en existe 19 types qui sont explicités en annexe 2. Les frames les plus présentes dans les échanges QUIC sont les ACK (les frames d'acquittement), MAX_DATA_STREAM (le crédit de données) et les STREAM qui contiennent les données utiles transmises.

Stream

Les Stream sont des éléments fondamentaux de Quic, car c'est à travers eux que les données sont transmises. Un Stream peut être représenté comme un tuyau temporaire avec un identifiant unique dans lequel sont transmis les données. Par ce mécanisme, chaque Stream peut transmettre des documents et ressources en parallèle par la même connexion.

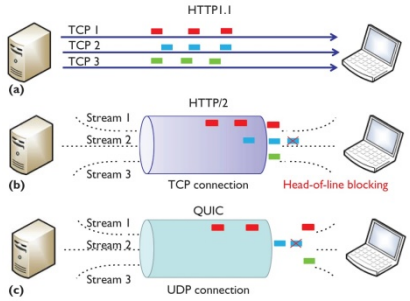


Fig15 : Stream Quic (c) comparé à (a) HTTP/TCP et (b)HTTPS/2/TCP

Les Acquittements

Une autre différence avec TCP est la méthode d'acquittement. Dans un protocole qui garantit la transmission des données, le mécanisme d'acquittement est fondamental pour s'assurer de la bonne réception de toutes les données. Dans TCP, un paquet acquittement est régulièrement envoyé à l'émetteur avec le dernier numéro de séquence reçu par le récepteur. Ce mécanisme permet à l'observateur d'obtenir ce genre de graphique en Fig16 (avec le SEQ), très pratique pour le troubleshooting, car il apporte beaucoup d'informations.

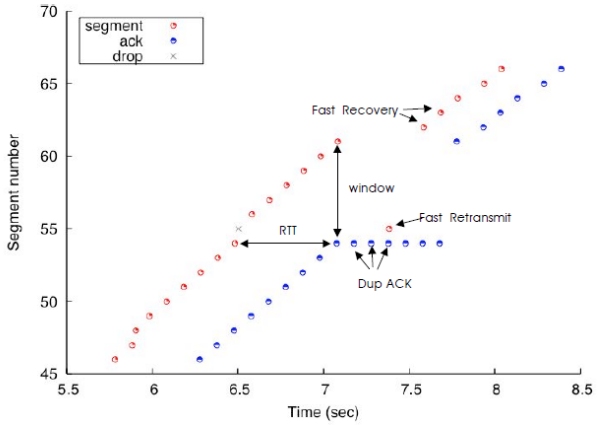


Fig16 : TCP seq/ack [1]

Le principal problème de cette méthode survient lorsqu'il y a des "trous" dans les acquittements. Dans ce cas, TCP renvoie tous les paquets depuis le paquet manquant, ce qui cause de nombreuses retransmissions inutiles (même si certains mécanismes de retransmissions permettent de pallier un peu au problème).

Quic a fait des choix différents quant aux acquittements. Le premier est de ne pas avoir des paquets uniquement dédiés aux acquittements puisque la frame ACK peut être empaqueté avec d'autres frames.
Un autre choix est celui du **SACK** (Selective Acknowledgment, RFC2018). Au lieu d'indiquer seulement le dernier numéro de séquence reçu sans perte, le SACK donne le nombre d'octets reçus sans perte puis des "range" de paquet reçu (voir Fig17). Ainsi, il suffit de renvoyer les morceaux manquant en même temps que le prochain envoie régulier (avec le mécanisme de fast retransmit). Le sack nécessite certes plus d'informations pour apporter la notification d'acquittement, mais il permet aussi d'être plus efficace dans la retransmission des données et donc optimiser l'utilisation de la bande passante.

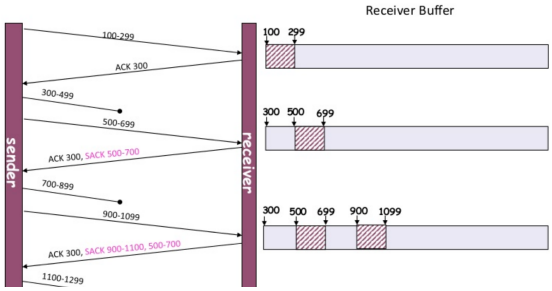


Fig17 : Mécanisme de SACK utilisé par Quic

<u>Les Pertes et le Contrôle de Congestion par Quic</u>

Définis par le standard IETF de Quic "Loss Detection and Congestion Control", la gestion des pertes, la méthode d'acquittement et le contrôle de congestion sont intéressants à connaître pour nous qui allons devoir débugger le réseau entre des piles qui se doivent de respecter ce standard.

La première différence notable par rapport à TCP est le **packet number** qui ne peux qu'augmenter (selon la philosophie que chaque paquet d'une connexion est unique et donc ne peut être renvoyé tel quel) contrairement au SEQ de TCP où un paquet du même SEQ peut être retransmis s'il a été considéré comme perdu.
Comme dit plutôt, Quic utilise les SACK et en particulier **les intervalles de SACK** ce qui permettent d'avoir une image très détaillée des données acquittées de celles qui ne le sont pas.
Un acquittement SACK devrait (Should) être envoyé à la réception de 2 paquets nécessitant un acquittement (ce qui n'est pas le cas de toutes les frames, comme les ACK, pas de ACK de ACK). L'émetteur de l'acquittement indique le délai qu'il a mis pour envoyer l'acquittement du dernier paquet, cette information est utile pour l'estimation du RTT par l'émetteur du paquet acquitté. L'estimation du RTT de la connexion se fait au travers du temps pour recevoir l'acquittement d'un paquet et est définit par 3 valeurs, le minimum, le RTT lissé et la variance (Jitter). Le calcul de ses valeurs est défini dans le RFC6298.

La détection des pertes se fait par 5 mécanismes (inspiré du Fast Retransmit RFC5681, Early Retransmit RFC5827, SACK loss recovery RFC6675 et RACK de TCP) :
- Le **paquet n'est pas acquitté**, en vol et un paquet suivant a été acquitté
- Le **packet-Threshold** est dépassé
- Le **Treshold de temps** est dépassé
- Le **temps de retransmission du paquet Crypto** est dépassé (ce paquet est critique car contient des informations nécessaires au chiffrement, donc renvoyé dès que possible)
- le mécanisme de **PTO** (Probe Timeout) considère le paquet comme perdu

(nous avons fait abstraction des mécanismes sous-jacent et des cas particuliers des frames à ne pas retransmettre, en mode datagramme)

La norme définit également les mécanismes de contrôle de congestion suivant :
- L'utilisation de l'algorithme **NewReno** (RFC6582) utilisant la taille en bytes plutôt qu'en paquet est conseillé même si l'utilisation d'autres algorithmes comme Cubic n'est pas exclu. Ces algorithmes définissent une fenêtre d'envoi qui limite le nombre de paquets en vol.

- l'ECN (RFC8311), Explicit Congestion Notification de IP doit être utilisé s'il est disponible sur le chemin de la connexion

- Les paquets Handshake, 0-RTT et 1-RTT perdu sont ignorés tant que les clefs cryptographiques ne sont pas mises en place

- Un mécanisme de reprise de connexion en cas de congestion persistante "Persistent Congestion"

- La norme recommande l'utilisation d'un "pacer" comme celui de Linux, Fair Queue packet scheduler (fq qdisc)

- Elle définit la machine à état présentée en Fig18 avec les différents comportements, ressemblant fortement à ce qu'il y a avec TCP.

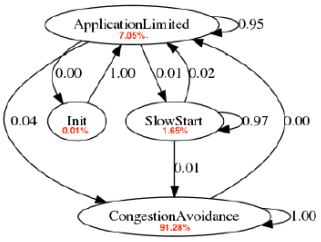


Fig18 : La Machine à état pour le contrôle de congestion Quic (avec les probabilités de transition approximative)

Avec une connaissance de ce que doivent utiliser les implémentations Quic en termes de contrôle de congestion, d'acquittement et de retransmission nous pourrons dans la suite analyser et comprendre le comportement d'une connexion observée.

Le chiffrement

À la différence de TCP, Quic est fortement chiffré et nativement. En plus d'un chiffrement des donnés utilisateurs, sont aussi chiffrés l'entête utile, les drapeaux de contrôle, comparé à TCP, Quic est clairement un protocole profondément chiffré et authentifié comme le montre la Fig19.

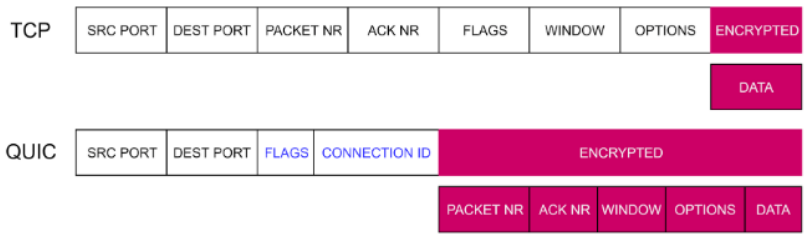


Fig19 : Chiffrement TCP vs QUIC/UDP (en bleu, authentifié, en rose, chiffré)

Depuis 2016, le protocole de chiffrement et d'authentification standard est la dernière version du standard TLS, **TLS1.3**, le même que pour TCP et beaucoup d'autres protocole à la différence de gQuic qui lui utilise la "Quic crypto". La Quic crypto avait l'avantage de remplacer avantageusement TLS1.2, jugé trop lent et trop coûteux en calcul. TLS1.3 corrige ces problèmes de TLS1.2.

Résumé

Pour conclure cette introduction à Quic, nous résumons **les principales caractéristiques que possèdent Quic** (inspiré d'un article de Robin Marx [10]) :

- Chiffrement de bout en bout de la couche application
- Chiffrement de bout en bout de la couche transport
- Prévention de l'ossification du protocole
- Mise en place d'une connexion sécurisée en 1-RTT
- Reprise de connexion sécurisée en 0-RTT
- Implémentation dans l'espace utilisateur
- Implémentation du contrôle de congestion en espace utilisateur
- Implémentation des contrôles de flux en espace utilisateur

- Stream de données multiple concurrent et indépendant
- Per-Stream Head-of-Line plutôt que Per-connexion (qui avait sonné la fin de SPDY, le protocole précédant Quic pour le remplacement de TCP)
- Migration de connexion grâce à l'identifiant de connexion
- Système de version qui permet la coexistence de version
- Modularité permettant d'ajouter des extensions
- Faible surcharge de bande passante (low bandwidth overhead) en diminuant le nombre de communication
- Réduction de la latence
- transmission de données ordonnée et garantie
- Prévention des attaques DDoS par design (coût élevé pour le client de l'initialisation d'une connexion)
- Prévention des attaques par token replay
- MTU discovery
- support de ECN (Explicit Congestion Notification)
- Fallback vers un autre protocole (TCP par exemple) en cas de non prise en charge par le serveur d'une version Quic
- Connexion à longue durée vie
- Stream à longue durée vie

C. La mesure passive sur Quic

Nous avons désormais un point de vue général sur le fonctionnement du protocole, mais rappelons-nous que l'opérateur comme Orange n'a pas accès aux contenus des paquets et donc ne voit que ce qui n'est pas chiffré. Avec Quic (utilisant UDP, reposant sur IP et non IPsec), les captures réseaux ne permettent que d'observer les entêtes Ethernet, IP, UDP et l'entête public Quic.
De ces informations, nous pouvons faire de la mesure passive de différents paramètres que nous allons maintenant expliciter.

Une capture passive du réseau est simplement la copie et l'enregistrement des paquets qui circulent sur une interface dans un fichier. Après traitement (c'est-à-dire filtrage d'une seule connexion grâce aux adresses IP et port source et destination) nous obtenons un morceau ou l'entièreté d'une connexion.

Dans un premier temps, nous pouvons repérer les paquets qui initialisent la connexion grâce à leurs entêtes caractéristiques de type Long Header. Ces paquets Initial et Handshake en particulier nous donnent la version de Quic utilisée (les versions sont normalisées auprès du groupe IETF de Quic) ainsi que les Identifiants de connexion.

Ensuite, les Short Header, paquets transportant les données, ne sont que très peu lisible avec seulement le premier octet et l'identifiant de connexion de visible. Or dans ce premier octet (ci-dessous pour rappel)

0	1	S	(R)	(R)	(K)	(PNLen)

Seulement 1 bit est vraiment utile pour les sondes ou les middleboxes et 2 sont utilisables pour expérimentations. Ce bit S a explicitement été introduit dans le draft transport de Quic en Mars 2019 pour répondre à un besoin de mesure passive et donc est particulièrement utile pour nous.

1. Le Spin bit S

Le **Spin bit** (0x20 du premier octet de l'entête quic) a été proposé par Brian Trammell et Mirja Kühlewind (chercheur à l'université ETH Zurich) à l'IETF en novembre 2016 [11,12] pour pallier au manque des "timestamp" présent dans TCP. Définitivement intégré dans la draft transport de Quic en avril 2019, le système proposé est assez simple et est utile à la mesure du RTT (un des paramètres qui permet de mesurer la qualité du réseau) par les sondes passives.

L'idée est que puisque la place pour intégrer de l'information dans chaque paquet est faible, nous devons coder l'information temporellement, le long du train de paquet. Cette information élémentaire

est le spin de ce bit (1 ou 0) par analogie au spin des particules élémentaires en physique. Au cours de la connexion, le spin change selon cette règle :

- Le *serveur* réfléchit le dernier spin bit qu'il a reçu sur le paquet qu'il s'apprête à envoyer.
- Le *client* envoie des paquets avec une polarité inverse au dernier paquet qu'il a reçu (en commençant par 0 en début de connexion)

La Fig20 (extraite de [12]) permet de se rendre compte de ce qu'il se passe au cours de la connexion. De par ce mécanisme, la sonde observe un signal carré avec des créneaux de longueur temporelle différent. De par ceci, nous pouvons en déduire les demi-RTT de chaque côté ainsi que le RTT down et Up Stream (Fig21) :

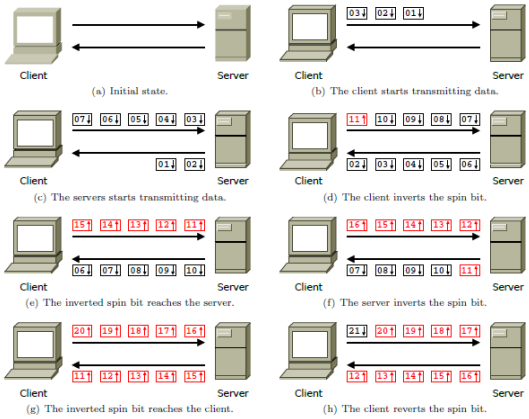


Fig20 : mécanisme du Spin bit

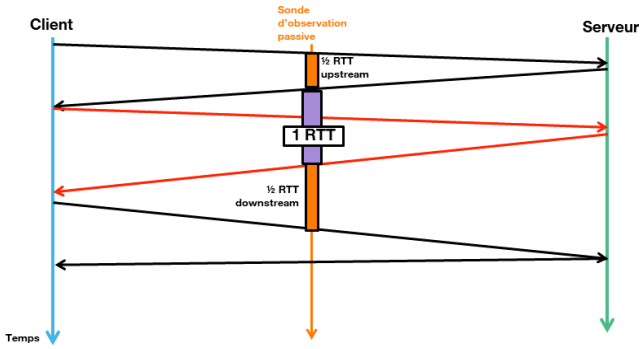


Fig21 : Mesure des demi-RTT et RTT total avec le Spin Bit

Proposé par Tramell et Kühlewind à l'IETF, le mécanisme a suscité de lourds débats sur le fait d'exposer des informations ainsi au réseau alors que la philosophie de Quic est bien de limiter l'exposition au réseau. Un des arguments contre a été la possibilité de faire de la géolocalisation de source par triangulation avec le RTT entre le client et différents serveurs connus (cf. [13] pour en comprendre le fonctionnement).

Geoff Huston disait du spinbit:

> *A cynical view would see the IETF as being incapable of holding the line that the end-to-end control state should be completely withheld from the network, and this spin bit is just one more step along an inexorable path of compromise that once more ends up gratuitously exposing user's actions to the network. There is probably a less cynical view as well, but I just can't think what it may be!*

Au final, le risque est minime et surtout, c'est une information accesible autrement par exemple, par les mécanismes du protocole IP.

De par ce mécanisme de mesure du RTT, nous avons un premier outil intéressant pour l'analyse des réseaux. En effet, le RTT et ses variations, en particulier lorsqu'il est mesuré régulièrement sur un même segment réseau permet de voir des congestions et des perturbations.

Pour autant, la seule mesure du RTT est insuffisante pour beaucoup de situations de panne, en particulier sur les pertes qui ne sont que peu visible par la seule connaissance du RTT. En particulier, il est difficile d'utiliser les méthodes précédemment utilisées par Orange pour le troubleshooting comme **la méthode de détection des pertes par dichotomie**. Ainsi il a fallu trouver une autre méthode ce qui s'exprime par la proposition d'Orange à l'IETF pour l'utilisation des 2 bits d'expérimentations.

2. Les extensions Q et R

Nous l'avons vu précédemment, le protocole dans les brouillons de l'IETF prévoit 2 bits réservés à l'expérimentation dans le premier octet de l'entête. Dans l'optique de la mesure passive pour la

debuggabilité des réseaux, Orange Labs au travers du travail d'A. Ferrieux et d'I. Hamchaoui, s'est efforcé de trouver une utilité à ses bits pour cette tâche.

Les informations qui nous intéressent pour avoir une information équivalente aux seq et et ack de TCP sont les suivantes :

- Le **nombre de pertes totales** sur la connexion
- Le nombre de **pertes à droite et à gauche** de la sonde (Serveur <-> Sonde et Sonde <-> Client) avec une granularité assez fine
- Le nombre de **pertes dans chaque sens**.

Le nombre de pertes rapportées au nombre de paquets de la connexion observée permet d'avoir un ratio de perte ainsi que le segment en faute dans ces pertes.
Pour chaque sens, nous avons besoin d'une information pour les pertes à l'amont et l'aval de la sonde. Le codage minimal de l'information est d'un bit. Donc nous avons besoin, pour l'information la plus minimaliste de 2 bits par sens de connexion (beaucoup plus pour un compteur comme avec le SEQ de TCP). Cela tombe bien, car 2 bits sont réservés à l'expérimentation.

Le sQuare Bit

Dans la proposition d'Orange, le premier bit (appelé **sQuare** ou **Q**) reprend l'idée du signal carré du spin bit, mais au lieu de coder une information temporelle, il code une information de cardinalité. Tous les N paquets émis, la polarité du bit Square est inversée (les paquets de type "Long Header" ne sont pas marqués mais participent tout de même à l'incrément interne du compteur de paquet émis). Ce mécanisme n'a besoin que d'un bout, l'autre n'ayant pas besoin de collaborer.
Sur le fil, le bit « bat » à une certaine fréquence 2N (**définis dans la norme**). Il ne reste plus qu'à la sonde de "compter" le nombre de bit Q avec la même polarité pour savoir si tous les paquets ont bien fait le trajet Emetteur-Sonde sans être perdu.
Par exemple, si N=64 et que la polarité s'inverse à 62 au niveau de la sonde, alors nous pouvons en déduire que 2 paquets de ce cycle ont été perdu (les paquets Long Header sont un problème car ne possédant pas l'information, il faut donc s'appuyer sur les paquets SH autour pour reconstituer le signal...)

Le Retransmit Bit

Il nous reste l'information des pertes après la sonde (entre la sonde et le récepteur). Il n'est pas possible pour la sonde de modifier le contenu de l'entête pour introduire un mécanisme similaire de comptage qui permettrait (par réflexion du bit reçu par le récepteur) de compter les pertes d'après, car les paquets, en plus d'un checksum (re calculable après modification), sont authentifié par de la signature cryptographique.
C'est donc la pile émettrice qui doit introduire une information sur ce qu'il s'est passé entre la sonde et le récepteur. Or la pile émettrice n'a pas conscience non plus de l'emplacement de la sonde. Il est donc impossible d'introduire une information directe sur les pertes sur ce segment.
Par contre, Quic à une garantie de livraison. Donc la pile émettrice possède l'information du nombre de pertes de bout en bout pour pouvoir retransmettre l'information. C'est cette information qui va nous permettre une mesure indirecte sur les pertes du segment Sonde <-> Reception, c'est cette information que l'émetteur va nous indiquer à l'aide du bit **R** pour **Retransmit**.
Dans la proposition d'Orange, le bit R pour Retransmit est à 0 par défaut et est mis à 1 pour chaque paquet que le pile pense perdu sur le chemin Emetteur<->Récepteur. S'il y a 5 paquets considérés comme perdus, les 5 prochains paquets émis auront le R à 1. Il suffit à la sonde de compter les paquets avec R à 1 pour connaître les pertes de bout en bout (avec un délai de 1 RTT upstream + ½ RTT downstream minimum, le temps que la pile émettrice sache qu'un paquet a été perdu, qu'elle marque un paquet sortant, et que ce paquet atteigne la sonde).
Par simple différence, connaissant les pertes de bout en bout et les pertes Emetteur<->Sonde nous en déduisons les pertes Sonde<->Récepteur. La preuve :

La transmission des paquets par le canal réseau peut être définit comme un système avec une fonction de transfert.

$$Y = h \cdot U, h \in [0,1]$$

Où Y les paquets arrivés et U les paquets à l'entrée. h = 0, tous les paquets ont été perdus, h=1 tous les paquets ont été transmis. On définit le taux de perte par $e = 1 - h$

Soit e, le taux de perte de bout en bout, $e = \frac{|p,p.R=1|}{|p|}$

Soit u, le taux de perte du côté émetteur (pour chaque bloc du signal), $u = 1 - \frac{|p,p.Q=0|+|p,p.Q=1|}{freq}$ où freq est la fréquence du battement du Q

De par la fonction de transfert on a d, le taux de pertes downstream,

$$h = (1-u) * (1-d)$$
$$(1-e) = (1-u) * (1-d)$$
$$\boldsymbol{d = \frac{e-u}{1-u} \approx e - u}$$

Ces premiers mécanismes ont l'avantage d'être simple à mettre en place, car ne nécessitent qu'un seul bout de la connexion pour fournir des informations sur un sens de cette connexion. Pour autant, il n'est peut-être pas complètement finalisé, car nous pouvons imaginer d'autre forme pour le signal (plus complexe que carré) et surtout d'une **période adaptée** pour être plus solide face à certaines autres perturbations comme le réordonnancement des paquets.

Ces extensions de Quic ont été proposé à l'IETF dans un **draft individuel du WG Quic, mais n'a pas été accepté** pour des raisons assez similaire à celle du spin bit, mais aussi car les participants à la conversation ne comprennent pas forcément l'intérêt d'avoir les taux de perte sachant que le RTT est connu. Cette méconnaissance, et le nombre de chantiers auxquels le groupe est déjà confronté ont fait un peu tomber dans l'indifférence ces extensions.

Pire, à la suite de discussion sur ce qui doit être en clair sur Quic en début d'année 2019, le standard s'est mis à préconiser également le chiffrement des 5 bits de poids faible du premier octet (avec les 2 bits réservés, la phase et la taille du numéro de paquet) car ils n'ont pas d'intérêt à être visible sur le réseau. Ce qui nous empêche donc de faire de la mesure passive sur les 2 bits réservés. (le protocole définit également qu'en production, si les 2 bits réservés sont différents de 0, alors le serveur doit stopper la connexion pour violation de protocole)
Pour ajouter encore de la difficulté à la mesure passive, il a été défini que le spin bit pouvait ne pas être présent dans toutes les connexions, en effet, il est décidé au début de la connexion de son utilisation ou non (au maximum 7/8 des connexions du serveur ont le spin bit d'activé) et donc un certain nombre de connexions ont une valeur du spin bit aléatoire à chaque paquet...

Sachant ça, nous pourrions se dire qu'il faudrait abandonner l'idée de debugger les pertes de réseaux pour le protocole Quic. Or, au vu des chiffres donnés plus tôt et de la tendance de l'adoption du protocole, il est plutôt intéressant de continuer à travailler sur ce sujet pour proposer une solution le jour où les opérateurs pousseront franchement en ce sens.
Ce stage ainsi que les expérimentations de la prochaine partie de ce rapport montrent pourtant que du travail doit être fait pour pousser ces extensions comme solutions à un problème qui peut être important pour les détenteurs de gros réseau, peut-être même pour la version 2 de Quic.

III. Les Expérimentations

Les expérimentations réalisées durant ce stage ont donc eu pour but de tester la validité des extensions ainsi que de monter une plateforme de démonstration pour le troubleshooting. Pour ce faire, nous avons besoin de 2 choses. Une **pile Quic** modifiable pour faire des tests de connexion avec des statistiques, et un **réseau** (si possible avec perturbations diverses) pour y mettre une **sonde** qui sera développée pour l'occasion.

A. Un emplacement pour les extensions

Du fait de la norme qui est devenue plus strict sur l'utilisation des bits réservés (comme le fait qu'ils soient chiffrés, qu'une exception PROTOCOL_VIOLATION soit levé en production s'ils sont différents de 0, ...) nous avons réfléchi à des alternatives pour implémenter les extensions ailleurs dans le paquet lié à une connexion Quic.

Le trailer UDP

La première idée a été de travailler sur le protocole de support de Quic, UDP. L'entête ne nous permet pas d'introduire le moindre bit car les 4 champs (ports, longueur, checksum) sont utilisés par les middleboxes et le routage. Néanmoins, il est possible d'ajouter les bits à la suite du paquet Quic au niveau du protocole UDP, dans « **l'UDP trailer** » (par analogie avec l'Ethernet Trailer).
Après avoir empaqueté le paquet Quic, un script ajoute les 2 bits (sur un octet) à la suite et recalcule le checksum UDP sans modifier la taille des données utiles.
L'avantage de cette méthode est qu'elle ne nécessite pas de modifier la norme, il suffit d'ajouter les bits au moment où les paquets UDP sont forgés.
Après des expérimentations à l'aide de la librairie python **scapy** (qui permet de forger des paquets, de les envoyer et les recevoir sans difficulté) nous avons remarqué que sur un réseau local, les paquets sont bien transmis sans erreur, sans modification, par contre, une fois sur le grand réseau internet, les paquets n'arrivent pas à destination ou bien dans certains cas, les données de bout n'y étaient plus (sans explication particulière sur l'origine de la "découpe"). Nous en avons conclu que certains équipements présents sur le réseau rejettent les paquets UDP qui possèdenr des données après les données utiles (définis par la longueur).
Nous ne pouvons donc pas mettre les bits à la suite de Quic dans les paquets UDP pour des expérimentations à large échelle.

Le TTL IPv4

Une seconde idée a été de travailler sur le protocole en dessous d'UDP, IP (plus particulièrement **IPv4** qui correspond à encore 75 % des connexions). Avec ce protocole, il y a beaucoup plus de place dans l'entête, et surtout des emplacements modifiables sans (trop) altérer le comportement. C'est le cas du champ "TTL" (Time To Leave) de 1 octet de long. Le TTL permet de limiter le temps de vie sur le réseau des paquets IP. L'émetteur du paquet fixe le TTL (en général 64 ou 0x0f) puis chaque nœud qui opère à la couche 3 sur le paquet, décrémente le paquet de 1. Si le TTL atteint 0, alors le paquet est tué. Du fait que très rarement ce comportement de mort d'un paquet est observé dans les réseaux et que le nombre de "hop" est souvent de 64, alors nous pouvons détourner le TTL et en particulier les bits de poids fort pour y incorporer les bits QR en particulier.
Ainsi, il ne suffira à la sonde qu'à appliquer un ET logique sur le 9eme octet de l'entête IP avec le masque 0x80 pour obtenir le Q et 0x40 pour le R. Nous avons utilisé cette technique pour une expérimentation avec l'entreprise Akamai sur des réseaux et serveurs réels, dont nous verrons un petit peu plus loin dans ce rapport les résultats obtenus.

Néanmoins, pour nos tests, qui ne seront pas avec des serveurs en production, **la solution de la modification des piles est la plus fiable et la plus proche de la réalité** que nous souhaitons pousser. La modification va donc consister à enlever la cryptographie sur le premier octet et ajouter les mécanismes de génération des bits.

La solution, finalement pour avoir un protocole debuggable et cohérent est bien de pousser à l'adoption de ces extensions dans la norme d'où se travaille de test, d'expérimentation et de preuve.

B. Sur les piles QUIC

Quic a été conçu pour être implémenté facilement en espace utilisateur à la différence de TCP qui l'est au niveau noyau. De plus, l'IETF pousse à la concurrence des piles avec un système de numérotation des implémentations qui permet d'avoir plusieurs piles qui cohabitent en même temps sur un serveur, ainsi au moment de la négociation de version, le client peut choisir la pile qu'il utilise. Il se trouve que la majorité des piles enregistrées auprès du Working Group sont aussi open-source pour permettre de vérifier leur implémentation du protocole et le respect qu'elles peuvent avoir de la dernière version de la norme.
Nous le répétons ici, nous ne traitons que des piles suivant la norme IETF (et donc pas gQuic) car le fonctionnement de gQuic est un petit peu différent même si les 2 tendent à se ressembler.

Comparatif

Sur la page du Working Group nous pouvons retrouver une liste d'implémentations (juillet-19, draft-22/gQuic046) présente en annexe 3.

Nous n'avions ni le temps ni intérêt à instrumentaliser toutes les piles. Nous n'en n'avions besoin que d'une voir de deux pour faire des comparaisons de performance. Il nous a fallu donc faire un choix.
Pour ce choix nous avons d'abord éliminé les piles dont nous n'avions pas accès aux codes, donc non-modifiable pour nos besoins.
Ensuite, nous avons éliminé les piles qui avaient du retard sur la norme.
L'article "Observing the Evolution of QUIC Implementations" [14] nous donne une méthode d'évaluation des respects de la norme par les piles. Nous l'avons appliqué aux piles, nous donnant une liste restreinte (Fig22a).
Enfin, nous avons utilisé les résultats des hackathon organisés par le WG IETF à chaque meeting pour le développement et le test des implémentations (ainsi que des outils). Ces résultats donnent une matrice d'interopérabilité des fonctionnalités Quic (Fig22b).

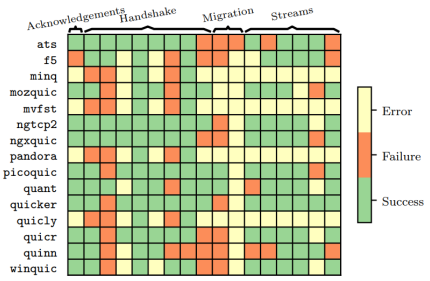


Fig22a : Matrice de fonctionnalités

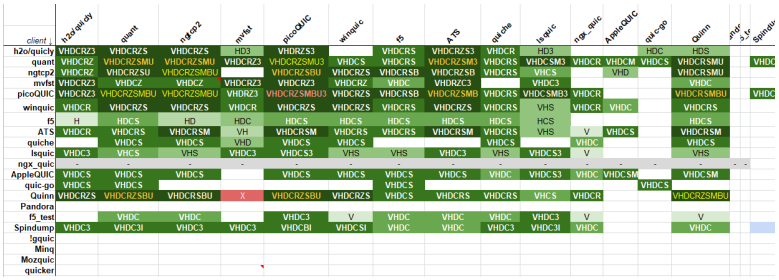


Fig22b : Matrice d'interopérabilité

Les fonctionnalités représentées sont :

- **V** : la réponse de la négociation de version est fonctionnelle
- **H** : Le "handshake" initial est complètement réussi
- **D** : Les flux de donnés sont échangés et acquittés
- **C** : La fermeture de connexion est complète et sans erreurs
- **R** : Resumption (Reprise)
- **Z** : l'accès au donnés en 0-RTT est fonctionnel
- **S** : "Stateless Retry", un handshake avec un retry de paquet est fonctionnel
- **M** : La migration est fonctionnelle
- **B** : Le rebinding de port est fonctionnel
- **U** : Un bout peut mettre à jour les clefs avec une réponse correcte de l'autre
- **3** : Une transaction en HTTP/3 réussie
- **|** : L'utilisation du spin bit est fonctionnel

Finalement, nous avons obtenu cette liste d'implémentation Quic à comprendre, compléter avec les extensions et instrumentaliser pour les tests.

- **Picoquic**
- **Mvfst** (ajouté en Juin 2019*)
- **Quicker**
- Quic-Go
- Quinn

*Note sur l'utilisation de mvfst

Mvfst a été rendu libre par Facebook en Mai 2019, il s'agit de l'implémentation de Quic (iQuic et gQuic) utilisée sur ses serveurs et ses applications mobiles principalement. Nous avons décidé de travailler dessus car d'une part, le développement est très avancé, mais surtout car Orange a été confronté à un problème avec l'application Facebook. Le 24 mai 2019, lors d'un évènement de retransmission en direct live de l'éruption du volcan de la Réunion via l'application Facebook, les équipes techniques ont remarqué qu'au bout de quelques secondes, le débit en 4G s'effondrait (divisé par 3, voir Fig23), ce qui diminuait la qualité du flux vidéo. Une équipe de Lannion a donc procédé à un troubleshooting du réseau de la réunion, mais là, problème, au lieu d'utiliser un protocole classique du streaming vidéo qu'ils savent débugger, l'application de streaming de Facebook utilise Quic.
Le contenu étant complètement chiffré, il était impossible détecter le problème (pertes ? latences ? où ?). Nous avons pris cet exemple comme un cas d'école de ce qui allait arriver dans les mois qui arrivent et avons donc pris l'initiative d'étudier et d'implémenter les extensions dans mvfst (implémentation utilisée dans l'application de facebook) et ainsi proposer les extensions au plus grand nombre, et en premier lieu, aux développeurs de mvfst, Facebook.

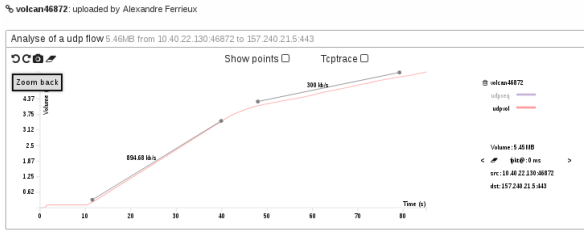


Fig23 : Troubleshooting Quic à la Réunion le 25-Mai 2019

Toutes ces piles proposent des "toy client/server" qui permettent de tester l'envoie et la réception de donnée. Picoquic permet également l'envoi de documents en HTTP/3.
Il nous a paru intéressant de regarder un peu plus en détail l'implémentation des piles choisies notamment :

- le fonctionnement des serveurs/clients
- les fonctionnalités implémentées
- les algorithmes de contrôle de congestion
- la méthode d'envoi des paquets
- le système de log
- la bibliothèque cryptographique utilisée...

Ce qui nous a confirmés dans notre idée que les extensions SQR sont faciles à ajouter.

Ajout des extensions

Même si l'étude de loin des piles est intéressante, il n'est pas nécessaire de comprendre en détail le fonctionnement interne des piles pour implémenter les bits (SQR). Nous avons développé une procédure à suivre de manière quasiment automatique pour les ajouter. Cette procédure est la suivante (tous les pseudocodes sont en annexe 4) :

1. Initialiser une machine virtuelle avec les dépendances et l'environnement liés à la pile
2. Cloner le répertoire de la pile, la compiler et la tester pour vérifier que la dernière version est fonctionnelle. Prendre connaissance des sorties de la compilation et du programme avant modification.
3. Si le **Spin bit** est présent allez à l'étape 4
   a. Ajouter à la structure/classe représentant une Connexion Quic les booléens **hasSQRBit** et **currentSpinBit**
   b. Ajouter les fonctions/méthodes **initSpinBit**(), **getSpinBit**(), **setSpinBit**()
   c. Appeler la fonction/méthode **initSpinBit**() à l'initialisation de la structure/classe connexion
   d. Appeler la fonction **getSpinBit**() au moment de la construction de l'entête des paquets LH et SH avant chiffrement
   e. Appeler la fonction **setSpinBit**() après déchiffrement de l'entête d'un paquet LH ou SH
4. Supprimer la cryptographie AEAD sur les 2 bits réservés du premier octet de l'entête SH Quic
5. Ajout du **sQuare bit**
   a. Ajouter à la structure/classe représentant une Connexion Quic le booléen **currentSquareBit**, l'entier **countSquare** et la constante **SQUARE_PERIOD** à 64
   b. Ajouter les fonctions/méthodes **getSquareBit**()
   c. Appeler la fonction/méthode **initSpinBit**() à l'initialisation de la structure/classe connexion pour initialiser le compteur et le bit
   d. Appeler la fonction **getSquareBit**() au moment de la construction de l'entête des paquets LH et SH avant chiffrement
6. Ajout du **Retransmit bit**
   a. Ajouter à la structure/classe représentant une Connexion Quic le booléen **currentRetransmitBit**, l'entier **countRetransmit**
   b. Ajouter les fonctions/méthodes **getRetransmitBit**()/**incrRetransmitBit**()
   c. Appeler la fonction/méthode **initSpinBit**() à l'initialisation de la structure/classe connexion pour initialiser le compteur et le bit
   d. Appeler la fonction **getRetransmitBit**() au moment de la construction de l'entête des paquets SH avant chiffrement

e. Appeler la fonction **incrRetransmitBit**() à l'endroit où la pile détecte un paquet perdu (mécanisme différent pour chaque implémentation)
7. (Optionnel) mettre à jour les tests unitaires avec les 3 bits
8. (Optionnel) Fournir un **.patch** à l'aide de la commande $\$ git\ format - patch\ 'commit - avant'\ 'commit - apres'\ >\ sqrextensions.patch$

La création de ces patchs pour **l'ajout des extensions dans les piles open source** est un argument indéniable pour pousser à l'adoption de ces extensions, en montrant que ce n'est pas un travail très compliqué, mais qu'il permet au final d'améliorer la QoS sur le réseau.

C. Première mouture de test

Avant même mon arrivée au sein d'Orange, des expérimentations avait été menée à l'aide d'une pile Quic sur des machines Lili présente à Lannion, à la Réunion et en Roumanie (maquette en Fig24) que j'ai perfectionné au début du travail comme prise en main.

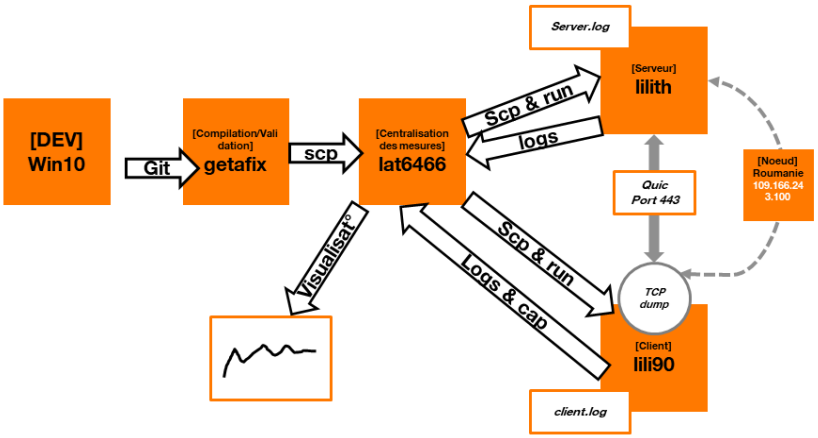


Fig24 : l'infrastructure de test Réunion-Roumanie-Aubervilliers

Le problème de cette solution est que nous ne pouvons introduire que des pertes aléatoires, et que finalement, nous ne contrôlons pas suffisamment tout l'environnement de test pour mener une étude. Les pertes pouvant être "naturel" sur le chemin Réunion <-> Aubervilliers, les RTT peuvent varier énormément d'une expérimentation à une autre sans compter les débits qui sont assez faible (maximum 5 Mbps). C'est ainsi que nous avons décidé de mettre en place un testbed, qui nous permet de valider certaines hypothèses et surtout de faire beaucoup d'expérimentations, très paramétrables, très rapidement.

D. Mininet

Avant de se décider à mettre en place toute une infrastructure d'expérimentation physique, nous avons étudié la solution de la simulation informatique de réseau. Pour ceci, nous avons testé l'outil Mininet[15].

Ce logiciel permet de programmer sur le réseau local un sous-réseau avec des nœuds et entités virtuels. Nous n'avons au final pas retenu cette solution, car pas assez stable, lourde en mémoire et surtout longue au redémarrage après chaque changement de configuration (comparé à un réseau physique monté et prêt à l'emploi). Nous nous sommes donc orientés vers une solution plus traditionnelle, celle "physique" avec une maquette et un réseau physique en laboratoire.

E. Testbed en laboratoire

Pour la conception d'un testbed d'expérimentation en laboratoire dédié à Quic et comprendre son comportement, nous avons d'abord réfléchi aux fonctionnalités nécessaires d'où la liste des **exigences** suivantes :

- Réseau complètement **contrôlé** (en limitant les sources de perturbation)
- **Générateur de perturbations** diverses contrôlable
- Mise en place **simple et modulaire**
- **Interface de contrôle** des expérimentations
- **Rapidité d'exécution** des expérimentations
- **Facilité de modifications** des paramètres de l'expérience
- **Maintenable** dans le temps
- **Debuggabilité** avec le maximum d'informations sur les états des piles, des erreurs, des bugs, ... pour réparer et améliorer plus rapidement

La maquette du réseau ainsi créée est le plus simple du monde comme la montre la figure ci-dessous avec une machine cliente, une machine serveur et une machine créant les perturbations qui intègre la sonde "middle", émulation des sondes qu'Orange possède sur ces réseaux.

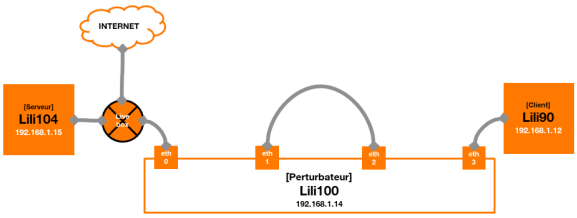


Fig25a : maquette du testbed

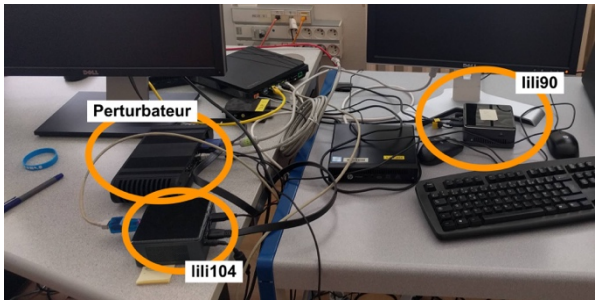


Fig25b : photo de la maquette

Ces 3 machines utilisent Linux comme système d'exploitation, plus particulièrement la distribution Debian modifiée, utilisée pour les Lili.

1. Architecture

L'architecture globale qui a été développée se décompose en 3 fonctions :

- L'**interface Web** de contrôle et de visualisation
- Le "**backend**" qui s'occupe de préparer, mener l'expérimentation, puis de récupérer et traiter les fichiers résultats. Ce backend se trouve sur la machine « **expérimentateur** » (lili104 dans la suite)
- Le "**testbed**" qui se compose lui aussi de 3 parties correspondant aux 3 machines
    - La machine **cliente** et **serveur** sur lesquelles se trouvent les clients et serveurs Quic
    - La machine **milieu** qui remplit plusieurs rôles
        - **Lien** entre les 2 bouts
        - **Applicateur** des paramètres
        - **Perturbateur** de réseau (sur les 4 segments)
        - **Exécuteur** de l'expérimentation
        - **Collecteur** des résultats

Cette architecture permet de découper facilement le travail en module. Modules qui deviennent relativement indépendant les uns des autres et donc apportent la flexibilité recherchée.
Il nous a fallu un certain temps de développement et de réflexion pour en arriver à l'architecture actuelle du code (présenté dans la Fig26 ci-dessous). Pour autant, le développement sous forme de module a permis de s'adapter aux contraintes et aux nouvelles fonctionnalités.

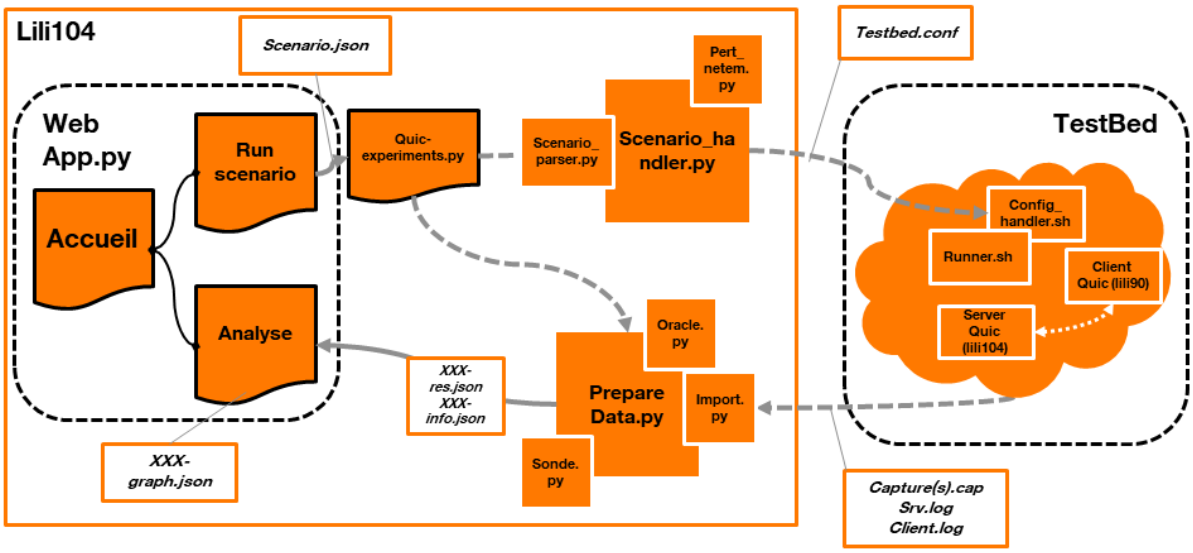


Fig26 : Infrastructure du code pour le testbed d'expérimentation

Le déroulement classique d'une expérimentation depuis l'interface web, est le suivant :

1. **Conception du scénario** avec l'interface web et enregistrement des paramètres dans un fichier au format json
2. **Lancement du scénario** toujours depuis l'interface
3. Lecture des paramètres du scénario avec le module **Scenario_parser**
4. **Vérification des paramètres** et conversion en fichier testbed.conf contenant les configurations pour le Testbed
5. **Exécution du runner** présent sur la machine "milieu" du testbed
6. Runner sur le testbed
7. Le module **prepare_data** avec d'abord l'importation et la décompression de l'archive de données
8. Si les captures pour les 2 bouts sont présentes dans les données
    a. **Calcul du referee** des pertes réelles
    b. Extraction des valeurs donnée par les 2 piles comme le seqquic
    c. Si le sslkeylogfile est présent, décodage les trames avec un dissecteur de paquet pour en comprendre le contenu
9. **Analyse du fichier de capture milieu** avec la **sonde** et générations des fichiers *res.json* et info.json
10. **Visualisation de l'analyse**, calcul des *graph.json* de représentation

Cette maquette est somme toute simple, les contributions intéressantes viennent de l'interface web (configuration et visualisation) et la conception du perturbateur.

2. Le perturbateur

La machine « milieu » ou « perturbateur » est un mini-ordinateur doté de **4 interfaces réseaux physiques**. Comme dit plus tôt, c'est sur cette machine que sont appliquées les perturbations, qui configure les paramètres de l'expérimentation, collecte les résultats et lie les 2 machines aux extrémités de la connexion Quic.

La particularité de posséder 4 interfaces nous permet donc de concevoir 2 segments de réseaux comportant 2 interfaces. Les segments sont en réalité des "bridge" logiciels entre 2 interfaces.
Les bridges ou "pont" sont des dispositifs logiciels opérant en couche L2 permettant la mise en lien de 2 interfaces, ainsi l'ordinateur joue le rôle d'un commutateur Ethernet.
Le montage d'un bridge sous linux (avec l'outil dédié au bridge, bridge-util) se fait avec

```
$ brctl addbr br0 # Creation
$ brctl addif br0 eth0; brctl addif br0 eth1 # Ajout des interfaces
```

$ ip link set dev br0 up # Demarrage du bridge

Chacun de ces segments est relié à une machine cliente ou serveur. Pour relier les 2 sous-segments et donc créer un lien indirect entre la machine cliente et serveur, nous utilisons un câble RJ45 physique entre les 2 interfaces restantes (physique pour ne pas avoir à le faire logiciellement et donc nous éliminer les risques de bug et de latence).

Les perturbations sont appliquées à l'aide du module **Netem** (Network Emulator) du package **tc** (Traffic Control) du noyau Linux. Nous expliciterons dans la prochaine partie le choix et le fonctionnement de Netem, mais il s'avère qu'il possède une fonctionnalité intéressante, c'est que les paquets sont traités en mode Egress.
Le mode **Egress** veut dire que les modifications sont appliquées sur le flux à la SORTIE de l'interface. Ainsi, puisque chaque segment possède 2 interfaces, nous pouvons appliquer netem sur chacune d'elles et donc contrôler les 2 sens de flux possible (extérieur<->perturbateur) indépendamment. Nous avons donc bien le contrôle sur nos 4 segments de réseaux montant/descendant, avant/après comme voulu. Il est à noter que la capture réseau avec l'outil **tcpdump** se fait APRES le Egress Netem, donc si la capture est réalisée sur eth2, c'est comme si elle avait été faite sur le câble reliant les 2 segments, ce qui est exactement l'emplacement des sondes sur le réseau.

La fig27 ci-dessous résume le fonctionnement du testbed et du perturbateur :

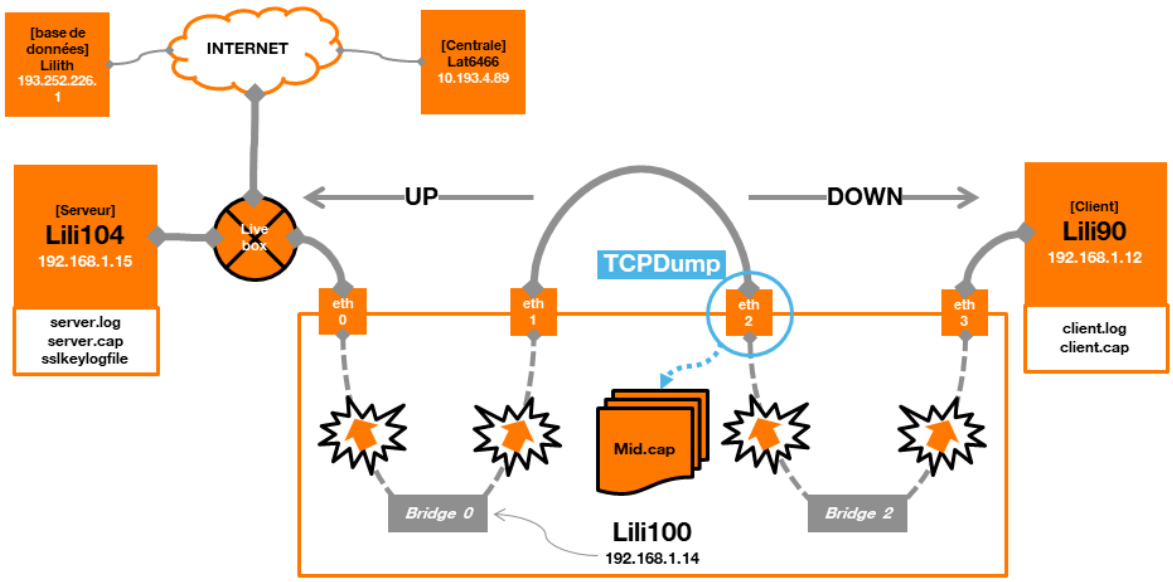


Fig27 : infrastructure réseau du testbed et du perturbateur

Le perturbateur n'a pas seulement le rôle de mauvais nœud réseau introducteur d'erreur. Il est celui qui **gère les expérimentations** et les **donnés résultats bruts** et fait aussi office de **sonde passive**. Pour ceci un script fait à peu près tout le travail, le **runner** dont les étapes sont les suivantes :

1. Nouvel id pour l'expérience à partir de l'id précédent dans le dossier de résultat
2. Chargement des paramètres du fichier testbed.conf
3. Application des règles netem
4. Lancement du serveur Quic
5. Lancement du client avec les paramètres et début du transfert des données
6. Fin de la capture réseau et suppression des règles netem
7. Récupération des résultats
8. Création d'une archive et compression des résultats
9. Renvoi de l'id d'expérience à l'expérimentateur

Il prend en entrée un fichier testbed.conf qui est mise à jour par l'expérimentateur. Ce fichier *testbed.conf* (un exemple en annexe5) comprend les paramètres d'expérimentation sur chaque ligne qui sont de 3 types

- Les paramètres de **debug**, numéro de version, date de création, ...

- Les paramètres liés aux machines Quic, pile à utiliser, période du sQuare, taille du fichier à transférer, ... C'est **l'environnement Quic** de l'expérimentation

- Les paramètres liés au réseau et aux perturbations, interface de capture, débit maximum, perturbations Netem pour chaque interface, ... C'est **l'environnement réseau** de l'expérimentation

La liste des **sorties** attendues est la suivante :

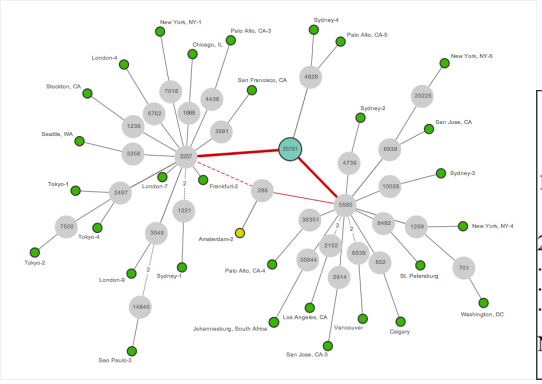
- Capture réseau de la sonde passive contenant uniquement les paquets chiffrés de la connexion
- Si la vue omnisciente est activée
    - Capture client
    - Capture serveur
    - Logs serveur
    - Logs client
    - SSLKeyLogFile pour le déchiffrement des trames

Entrons maintenant dans le détail des perturbations réseau avec la compréhension de l'outil Netem.

3. Modélisation des perturbations réseau

Les perturbations réseaux, théorie et réalité

Les réseaux informatiques sont modélisables par des graphes, structure de nœuds reliés par des arcs (Fig28a). Les nœuds concentrent des connexions pour les router vers un autre nœud et ainsi permettre aux paquets de parcourir le réseau d'un bout à un autre. Un exemple de nœud commun est le commutateur Ethernet (qui fonctionne en couche L2) qui permet de router les paquets (Fig28b).



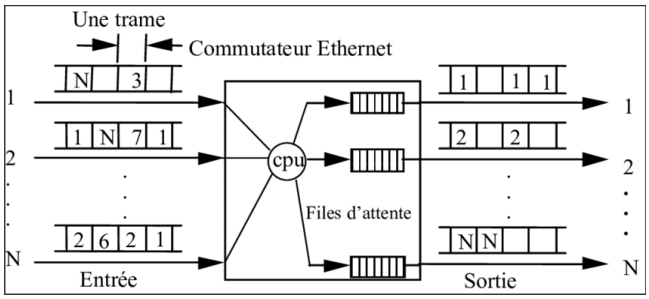


Fig28a : Réseau représenté sous forme de graphe
Fig28b : un nœud réseau représenté par un réseau de file d'attente

Ceux sont ces équipements réseaux et leur fonctionnement interne qui explique la majorité des perturbations. Il en existe 5 principales qui sont les suivantes :

- Les **délais**. Introduit en partie dû à la limite de la vitesse de propagation de l'information dans le médium (Avec une vitesse limite à c = 3.10^8 $m.s^{-1}$) mais aussi dû au temps de traitement par les nœuds réseau. Chaque nœud réseau traversé par le paquet va le retarder d'un delta t dû à la mise dans la queue et le traitement. C'est pour cette raison que les services nécessitant de faibles latences multiplient les CDN partout dans le monde pour toujours être au plus près du client et donc limiter le nombre de "hop". Il est à noter que les délais suivent en moyenne une loi normale, c'est-à-dire qu'ils vont être en moyenne d'un délai D mais qu'en réalité, ces délais vont s'étaler sur une plage D-σ_D et D+σ_D (σ la variance). Cet effet, appelé "**Jitter**" est défini comme une variance de la latence ou d'une "IP Packet Delay Variation" dans le RFC3393. Le Jitter est dû à des différences de configuration des routeurs, du multi-chemin... Il pose problème car il rend l'estimation du RTT plus difficile pour la pile et surtout mène à des réordonnancements dans le train de paquet.
- Les **duplications**. Perturbation très rare dans le réseau, elles ne sont pas produites par les équipements réseaux même plus par les piles émettrices qui auraient tendance à renvoyer trop tôt un paquet considéré comme perdu alors qu'il a juste pris du retard. Quic n'est donc pas concerné par la duplication de paquet, car jamais un paquet est renvoyé tel quel, la donnée étant rempaquetée.

- Les **corruptions** sont aussi des perturbations rares sur les réseaux modernes, car les erreurs de traitement sont devenues anecdotiques. Les corruptions sont plus souvent dues au medium utilisé, comme pour les réseaux non-filaires où les perturbations importantes sur le signal empêchent de reconstruire le paquet parfaitement et créer donc des paquets corrompus. Les corruptions se produisent donc seulement sur le dernier segment client. En général, les paquets corrompus sont jetés par les équipements du réseau, car soit ils ne sont pas reconnaissables, soit les checksums sont incorrects. Les corruptions deviennent des pertes pures lorsqu'elles surviennent ailleurs que sur le dernier segment.

- Des **réordonnancements** peuvent survenir lorsqu'un paquet passe devant un autre dans le flux. Cet effet survient lorsque le flux de paquet n'utilise pas le même chemin au sein du réseau (à cause du routage dynamique et de la répartition de charge) ou bien lorsque les paquets sont traités sur des processeurs différents (à cause la généralisation du multiprocesseur au sein des équipements). TCP, en particulier dû à son système d'acquittement, peut être sensible au réordonnancement car un paquet qui prend beaucoup de retard sur d'autres risque d'être considéré comme perdu et donc occasionné nombre de retransmissions. Nous verrons plus loin qu'il s'agit d'une contrainte importante lorsque nous concevrons une sonde Quic utilisant les extensions QR.

- Finalement, la plus importante cause de perturbation et surtout celles que nous cherchons à détecter lors des investigations, les **pertes simples** de paquet.

Les pertes dans les réseaux sont multifactorielles.
Les plus faciles à s'imaginer sont les pertes dues au **matériel ou au medium physique**, un câble Ethernet brisé, une perturbation dans l'onde radio, une simple carte électronique dans un équipement... Ces sources sont assez simples à trouver en utilisant la méthode dichotomique car les pertes se retrouvent toutes au niveau d'un seul équipement.
Un autre type de pertes, beaucoup plus communes quand nous parlons de réseau sont celles dues à la **congestion** des équipements.
La figure ci-dessous montre l'influence du goulot d'étranglement ou « bottleneck » d'un équipement réseau mal dimensionné sur le débit de donnée.

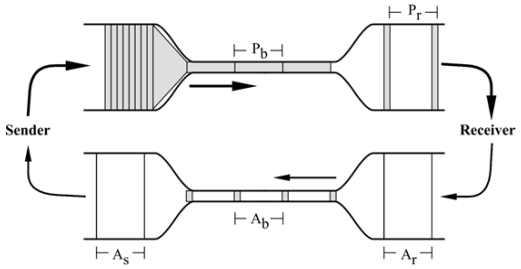


Fig29 : Goulot d'étranglement du réseau

Comme vu précédemment, un routeur et la plupart des nœuds réseaux sont comme des files d'attente qui se remplissent et se vident en fonction de l'entrée et de la vitesse de traitement. En phase de congestion, les files d'attente (donc le cache) se **remplissent jusqu'à débordement**. À ce moment, si rien n'est fait par les autres équipements ou les piles émettrices (avec des mécanismes comme ECN Explicit Congestion Notification ou RED, Random Early Detection), les files débordent et cela se traduit par des "drop" des nouveaux paquets par les schedulers (ordonnanceurs). Ce mécanisme est appelé "**tail drop**".
Les pertes peuvent être aussi causées par des effets purement protocolaires comme des erreurs de calcul du checksum, un TTL IP qui atteint 0... Elles peuvent aussi venir de mauvaises configurations d'un réseau (par exemple des règles de filtrage trop peu permissives) ou même d'attaques informatiques qui peuvent également occasionner ces pertes comme avec les "Packet drop attack" qui sont une sous-catégorie d'attaque par déni de service (DDoS).

Toutes les pertes ont un point commun, c'est que du point de vue de l'observateur du réseau, un certain nombre de paquet entre sur le réseau, mais n'en ressort pas. Le réseau peut donc être considéré comme une boîte noire avec un taux de transmission qui se doit d'être le plus proche possible de 1.0 pour assurer la qualité de service. Avec les débits en jeu sur certains cœurs réseaux, même une perte de 0.1 % des paquets peut correspondre à des gigas de données perdus qui devront être renvoyé, ce qui a un coût pour l'opérateur.
Des modèles ont été développés très tôt pour répondre à cette problématique de modélisation des pertes, dès le début des années 60 avant même l'apparition des réseaux téléphoniques commutés.

Il en existe 3 communément utilisés [16]:

- Modèle indépendant ou de **Bernoulli** basé sur la distribution du même nom où la probabilité p est celle de perdre un paquet. Il s'agit du modèle le plus simple où un paquet est considéré a une probabilité p d'être perdu avec une indépendance entre chaque paquet. L'indépendance des pertes est le gros point négatif de cette méthode, car elle ne reflète pas vraiment ce qu'il se passe dans la réalité. Les paquets sont rarement perdus individuellement de manière aléatoire mais plutôt perdus par "rafale", justement dû aux mécanismes de tail-drop, de burst et de congestion.
- Le **modèle de Gilbert-Elliott** (Fig30a) utilise un système à 2 états avec des probabilités de passage d'un état à un autre. Ces 2 états représentent les périodes de transmission sans perte et avec perte. De part ce modèle, un effet de corrélation fait que la probabilité de perdre un paquet après en avoir déjà perdu un est d'autant plus important et inversement, la probabilité de perdre un paquet après en avoir transmis plusieurs est plus faible. Pour autant, ce mécanisme ne rend pas compte des paquets perdus de manière isolé dans une phase sans perte ni la transmission d'un paquet dans une phase avec beaucoup de perte [17].
- Le dernier modèle (Fig30b), le plus difficile à estimer, mais le plus réaliste est appelé **"4-state Markov Model"** et comme son nom l'indique, il s'agit d'un modèle basé sur une chaîne de Markov à 4 états. L'état 1 est celui du paquet correctement transmis, l'état 2 d'un paquet transmis en période de burst, l'état 3, la perte d'un paquet en période de burst et l'état 4 la perte d'un paquet en période de transmission.

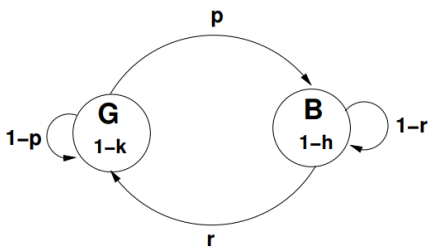


Fig30a : Modèle Gilbert-Elliott où G est l'état sans pertes, B l'état avec pertes et p,r,k,h les différentes probabilités de passage

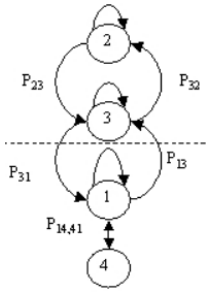


Fig30b : Modèle 4-State Markov

Les pertes sont aujourd'hui faibles (de l'ordre de 0.1 %) voire inexistantes au sein des réseaux moderne, bien dimensionnés, avec beaucoup de redondances et des connexions utilisateurs sortant rarement du pays pour atteindre un serveur possédant l'information recherchée. Pour autant, en cas de panne, les pertes deviennent importantes et peuvent atteindre les 5 %. C'est ce phénomène de pertes en rafale que nous avons émulé lors des expérimentations.

Nous savons maintenant les perturbations que nous souhaitons émuler sur notre testbed pour tester les mécanismes. Regardons maintenant les outils qui nous permettent de le faire.

Les outils

Dans l'article, "A comparative study of network links emulators" [19], les auteurs Nussbaum et Richard nous expliquent comment faire de l'émulation de réseau avec des logiciels et techniques différentes. Ils en ont remarqué 2 grands types, les émulateurs de réseaux virtuels comme mininet vu précédemment et les émulateurs de lien réseau (plus simple) ce que nous souhaitons. Parmi ces derniers, 3 sont de bonne qualité pour la recherche, utilisable sous linux et avec une communauté de chercheurs importante. Il s'agit de **Dummynet**, **NISTNet** et **TC/Netem**. Les principales caractéristiques sont présentées dans le tableau ci-dessous (extrait de l'article original) :

	Dummynet	NISTNet	TC/Netem
Availability	Included in FreeBSD	Available for Linux 2.4 and 2.6 (< 2.6.14), patch available for more recent versions	Included in Linux 2.6
Time resolution	system clock (up to 10 KHz)	Real time clock	system clock (up to 1 KHz) or high resolution timers
Interception point	Input and output	Input only	Output only
Latency	Yes, constant value	Yes, with optionally correlated jitter following uniform, normal, Pareto, or normal+Pareto distributions	Yes, with optionally correlated jitter following uniform, normal, Pareto, or normal+Pareto distributions
BW limitation	Yes, delay to add to packets is computed when they enter Dummynet	Yes, delay to add to packets is computed when they enter NISTNet	Yes, using the Token Bucket Filter from TC
Packet drop	Yes, but without correlation	Yes, optionally correlated	Yes, optionally correlated
Packet reordering	No	Yes, optionally correlated	Yes, optionally correlated
Packet duplication	No	Yes, optionally correlated	Yes, optionally correlated
Packet corruption	No	Yes, optionally correlated	Yes, optionally correlated

Fig31 : Comparatif des émulateurs de lien réseau

Précédemment nous avons vu que TC/Netem avait été choisi, pourtant en regardant ce tableau, son utilisation plutôt que NISTNet n'est pas clair. Les 2 raisons sont les suivantes :

- Le traitement des paquets en **Egress** plutôt qu'en Ingress
- L'intégration native dans le **noyau linux** avec toute la suite logiciel iproute2

En cherchant sur le site Github.com (qui rassemble la majorité des projets open-source), nous avons trouvé plusieurs implémentations de "shaper" de réseau se basant sur Netem, ce qui aurait pu simplifier le développement. En réalité, ces shaper ne sont souvent qu'une couche d'interface web pour netem avec moins d'option (les options les plus avancées n'étant pas accessibles). Nous avons essayé d'utiliser ATC (Augmented Traffic Control) de Facebook qui nous semblait complet, malheureusement, nous n'avons pas réussi à l'adapter facilement à notre architecture. Nous avons donc préféré concevoir notre propre surcouche modulaire, adaptée à notre infrastructure et qui permet d'utiliser le maximum d'options.

Dans notre démarche de validation, nous avons décidé de comprendre les fonctionnalités et surtout de comprendre le fonctionnement interne de Netem pour s'assurer de sa justesse dans nos tests et donc des expérimentations.

Traffic Control du noyau Linux

Le programme **tc** (Traffic Control, manuel en [19]) fait partie de l'ensemble iproute2 de Linux qui permet d'accéder à des configurations réseau.
Sous Linux les paquets IP sont traités selon les étapes suivantes :

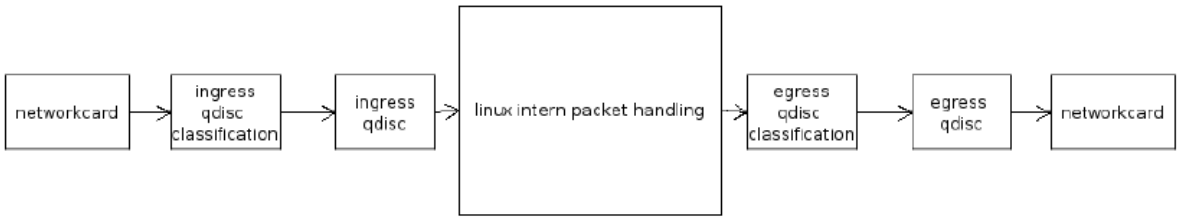


Fig32 : Gestion des paquets au sein du noyau linux

Le package TC possède 3 prérogatives :
- le **monitoring du réseau** pour le système
- la **classification** du trafic
- la **manipulation** du trafic (avec le module Netem notamment)

Un élément fondamental de tc est la **queuing discipline** (qdisc). Une qdisc est une file d'attente de paquet implémentant un algorithme pour décider quand sortir les paquets. Il en existe 2 types, classful/classless (avec classe ou sans classe) sur l'entrée et la sortie de l'interface réseau

(egress/ingress qdisc). Les classes sont une abstraction qui permet de classifier les paquets à l'aide de filtre.
À chaque interface est assignée une qdisc root (racine) de laquelle dérive des classes de différentes files d'attente (cette forme d'arbre est visible sur la Fig33a).

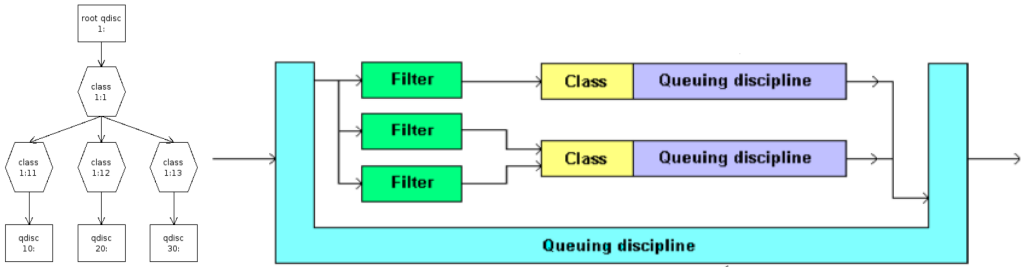


Fig33 : qdisc hierarchy

Les commandes pour paramétrer une qdisc (QDISC) root pour une interface DEV avec une classe et un filtre FILTER sont les suivantes :

```
$ tc qdisc add dev DEV handle 1: root QDISC [paramètres]
$ tc qdisc add dev DEV parent 1: classid 1:1 QDISC [paramètres]
$ tc filter add dev DEV parent 1: FILTER
```

Par défaut la "queueing discipline" est pfifo_fast, soit une file d'attente "FIFO" (Premier arrivé, premier servi) ce qui est le comportement par défaut pour interface réseau avec un débit qui n'est pas limité, l'interface est alors en mode "best effort". La liste des qdisc utilisables dans tc est présentée en Tab3.

qdisc	Signification	Type	Description
TBF	Token Bucket Filter	classless	Limite le débit des paquets, contrôle le débit de l'interface
SFQ	Stochastic Fairness Queueing	classless	Division du traffic dans différentes files d'attente puis envoie avec un ordonnanceur de type "round-robin"
PRIO	Prioritization	classful	Prioritise certaines classes de paquet sur d'autre
CBQ	Class Based Queueing	classful	Permet de faire du shaping mais assez complexe à utiliser
HTB	Hierarchy Token Bucket	classful	Dérivé de CBQ et plus simple à utiliser

Tab3 : les différentes Queueing discipline présentent dans tc

Le perturbateur utilise Netem comme annoncé plus tôt et doit différencier le trafic Quic qui doit être perturbé entre le client et le serveur de celui qu'il y a entre le perturbateur et l'expérimentateur (connexion par le protocole SSH (donc TCP) entre les 2 durant l'expérimentation). Ces 2 conditions sont réunies si nous utilisons la qdisc HTB.

HTB comme TBF fonctionne sur un principe de "Token" (jeton). Un jeton est donné au moment de l'ajout du paquet dans une queue et sert à prendre la décision de sortie. Ce système permet de contrôler le débit sur une connexion et d'introduire des effets de "burst".

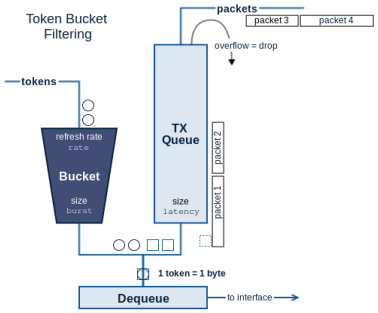


Fig34 : Le système de queue par jeton comme pour TBF ou HTB

HTB permet notamment de différencier le trafic par classe de trafic et appliquer des opérations sur chacune de ces classes. Ces opérations comme avec Netem.

Netem

Netem pour **Network Emulator** [28,29,31] est l'outil que nous avons utilisé pour l'expérimentation. Il a l'avantage d'être simple à utiliser et de proposer beaucoup d'options. En annexe 6 le manuel de netem nous donne toutes les options disponibles. Toutes les perturbations décrites dans le paragraphe « Les perturbations réseaux, théorie et réalité » sont bien présentes. Pour comprendre le fonctionnement interne de netem (notamment le fonctionnement du rejet de paquet), nous avons étudié la fonction dans le module noyau qui s'occupe dans l'envoi ou du drop des paquets annexe 7. Le fichier est dans */net/sched/sch_netem.c* (dans le noyau linux 4.9), a été écrit par Stefano Salsano et Fabio Ludovici de l'université de Rome pour le noyau linux >=2.6, et a été inspirée de l'outil NIST.

Le schéma ci-dessous résume le fonctionnement interne de netem à l'issu de la lecture de sch_netem.c

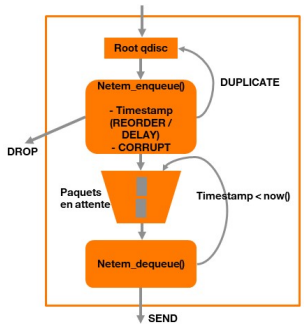


Fig35 : fonctionnement interne de netem

On y remarque les options de DUPLICATE, de DROP, de REORDER, de DELAY ou de Rate. Comme pour HTB, il s'agit d'une queue dont l'ordre des paquets à envoyer ainsi que le temps de départ va être dynamique en fonction des nouvelles entrées et des dequeue. Il est à noter que si le paquet est jeté, alors la fonction qdisc_qstats_drop() est appelé et va émettre un signal au processus émetteur du paquet de « non-autorisation d'envoi » (EPERM).

Netem est le cœur du perturbateur et sa pièce maîtresse, donc pour s'assurer de son bon fonctionnement, nous avons fait des tests à l'aide de "pings" et de capture réseau. Nous avons par exemple noté qu'il est préférable de créer les réordonnancements "soi-même", c'est-à-dire sans passer par l'option reorder, mais plutôt en utilisant un Jitter sur la latence. D'ailleurs, dans la réalité, le réordonnencement existe, car les paquets ne mettent pas le même temps à être transporter, donc à cause de la variance du délai de transport. Les délais d'une durée supérieure à 1 ms sont aussi très proches de ceux souhaités (écart de 0.1 ms maximum). Le montage avec les 2 bridges et les 4 interfaces fonctionnent correctement aussi avec un contrôle sur les 4 segments sans perturbations des autres.
Nous avons tout de même remarqué une baisse du débit maximale avec et sans tc et netem d'activer (même avec un débit au-dessus du maximum du lien) passant de 80mbps à 45mbps en burst et 200Mb (25mbps) sinon (ce qui correspond bien au maximum définit par tc htb).

Nous n'avons pas essayé toutes les options qu'offre notamment HTB avec les classes, les priorités, nous nous sommes contentés de filtrer les paquets (commandes ci-dessous) entre les flux sans queue particulière et avec le port source ou le port destination égale à 443 (le port utilisé par le serveur Quic).

```
$ tc filter add dev $eth parent 1: protocol ip prio 1 u32 flowid 1:1 match ip sport 443 0xffff
$ tc filter add dev $eth parent 1: protocol ip prio 1 u32 flowid 1:1 match ip dport 443 0xffff
```

Le perturbateur est défini, reste à le configurer et l'utiliser pour expérimenter. D'où l'implémentation de l'expérimentateur qui suit.

4. Implémentation

L'architecture du code a été présentée en Fig26. Elle a la particularité de la modularité, chaque module étant relativement assez indépendant les uns des autres.
Le langage utilisé principalement est le langage Python car il est polyvalent, lisible facilement et apporte une certaine flexibilité grâce au concept de module. Pour l'interface web, le HTML, CSS et Javascript ont été utilisé. Enfin, les scripts présents sur le perturbateur ont été écrits en Bash.
Les formats de fichier utilisé ont été standardisés et permettent d'avoir une interopérabilité dans le temps. Nous avons déjà décrit le fichier *testbed.conf*. Il nous reste 2 types de fichier, celui des scénarios et des résultats.

Le Fichier scénario

Les scénarios sont les fichiers qui décrivent l'expérimentation, que ce soit au niveau de **l'environnement d'expérimentation** (nom des machines, adresse IP, nom de l'expérimentation, paramètres de capture) que de **l'environnement réseau** (les perturbations) et les paramètres Quic (pile, valeur de N, …).

Un exemple de scénario est disponible en annexe8. Le format est clair et il est aisé de changer ou d'ajouter des paramètres. L'accès aux paramètres n'est pas suffisant, il faut aussi bien les configurer et sans difficulté.
Sans difficulté, se fait grâce à une interface web (capture d'écran partielle en Fig36) qui va en plus de présenter toutes les options configurables vérifier les paramètres entrés.

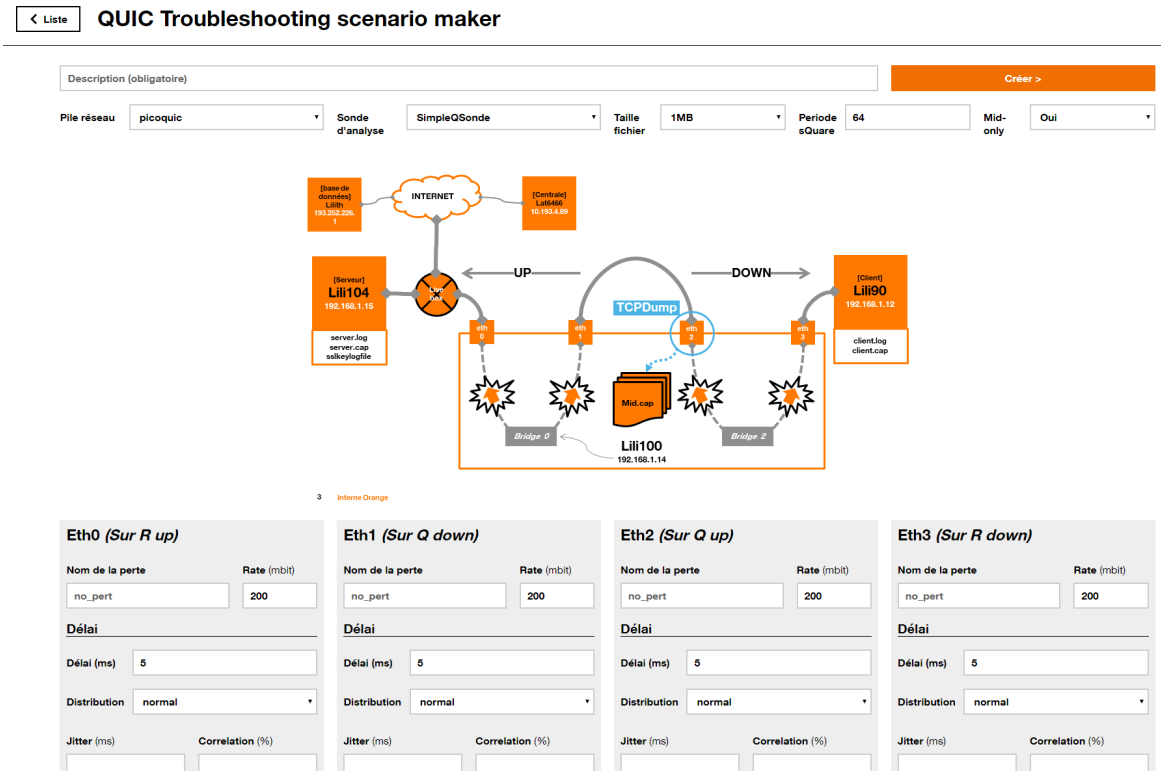


Fig36 : Interface de création d'un scénario

Bien les configurer pour notre cas, c'est comment les configurer pour se rapprocher de la réalité des réseaux et des cas extrêmes qu'il est possible de rencontrer dans ces réseaux et in fine valider le fonctionnement des bits SQR et la sonde.
Nous avons trouvé des "sets" de configuration ouverts sur Internet, comme ceux d'exemple utilisés dans ATC de Facebook [20]. Ces sets définissent en général le délai sur la connexion de bout en bout, la variance sur le délai, les pertes ainsi que le débit.
Quelques exemples de ces paramètres d'exemple sont présentés dans le tableau ci-dessous :

Type	Délai (en ms)	Jitter (en ms)	Pertes (%)	Débit (Mbps)
Wifi entreprise	5	4	0	15
Wifi aéroport	8	6	0.5	1
ISP congestionné	20	8	1	2
LTE	2.5	1	0.1	20

Tab4 : Exemple de configurations proposées par ATC

En réalité, nous pouvons obtenir ces chiffres dans les réseaux d'Orange grâce aux outils de monitoring de l'équipe. Ces chiffres sont intéressants, mais ne permettent de définir que la connexion de bout en bout, Or dans notre montage nous voulons pouvoir **régler les perturbations avant et après la sonde**. En fonction du segment avant et après, les paramètres changent grandement (ce qui justifie aussi le fait de pouvoir avoir une sonde déplaçable le long de la connexion pour faire de l'investigation dichotomique). Ainsi, les paramètres pour le segment Client<->Sonde seront complètement différents si la sonde se trouve au niveau du point de raccordement du client ou si elle est au cœur réseau. De plus, les connexions ne sont souvent pas symétriques du point de vue de l'utilisateur final avec souvent une grosse différence entre le download et l'upload (en termes de débit par exemple).

À partir de ces paramètres initiaux, des tests avec des scénarios intéressants ont été réalisés, mais le travail prend plus d'intérêt avec beaucoup de tests, avec **beaucoup de conditions initiales sur le réseau et des tests répétés pour valider les observations**.

C'est pour cela qu'en plus de l'interface web qui permet de faire simplement des scénarios, il est possible de **générer avec un CLI** (Command Line Interface) moult scénario avec un paramètre variant et les autres invariants et de lancer des tests en "**multiscénario**". Avec un fichier texte d'entrée avec la liste des scénarios à exécuter, l'expérimentateur va tester séquentiellement tous les scénarios. Ne reste plus qu'à en extraire les résultats et les analyser pour comprendre l'influence du paramètre. Ce programme permet par exemple d'observer l'influence de la période Q sur la détection des pertes pour un environnement réseau donné.

Les fichiers scénario sont l'apanage du module "**scenario_handler**" qui les traite et les exécute sur le testbed. Les résultats obtenus du testbed sont eux ceux du module "**prepare_data**" qui génère les fichiers de résultats.

Fichiers résultats

Les fichiers résultats sont obtenus à la suite d'une expérimentation à partir de données brutes du testbed et après traitement par le module **prepare_data**.
Si l'expérimentation s'est faite en mode "mid-only", c'est-à-dire que nous nous plaçons en tant qu'observateur du réseau, seul la capture du flux Quic sur le point de mesure est utilisée. Sinon les logs de la pile serveur, client et la clef SSL pour déchiffrer la connexion sont aussi présent.

Du fichier de capture au point de mesure, les bits SQR sont extraits ainsi que des informations de taille du paquet, de timestamp... ces informations "brutes" sont ensuite traitées par un module "**sonde**" facilement interchangeable qui en extrait les pertes, les RTTs, les bifs, ... Toutes ces informations sont ensuite compilées dans le fichier *XXX-res.json*. Ce fichier contient donc **des résultats utiles à l'analyse** avec au minimum les informations (dans les 2 sens) :

- Round-time-trip
- Pertes sQuare
- Pertes Retransmit
- données transmises (timestamp, taille des paquets depuis le début, le spin des 3 bits)

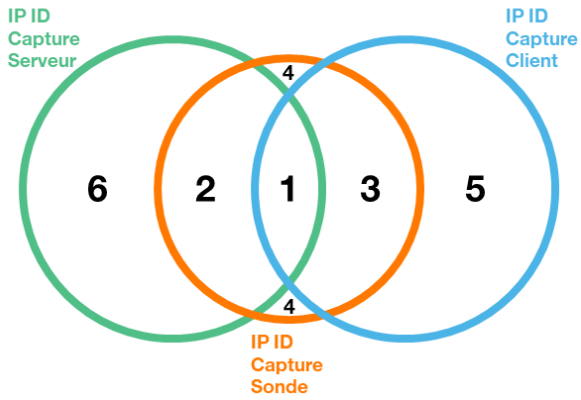
En plus du fichier *res.json*, un fichier *info.json* est généré contenant les informations utiles quant à l'expérimentation comme l'identifiant, le format du premier octet de Quic, la description de l'expérimentation, la date et la taille de la capture (en nombre de paquet). Ces informations étaient dans un premier temps dans le fichier *res.json*, mais ce fichier, pouvant être long à charger, entraînaient des lenteurs sur l'interface de présentation de la liste des résultats. En séparant ces informations de *res.json*, nous accélérons l'ouverture des fichiers et donc le temps de chargement de la page de présentation.

Si le point de vue de l'observateur omniscient est activé, beaucoup d'informations sont utilisées pour compléter les résultats.

Les **logs** des piles sont utilisés pour extraire le numéro de séquence interne, l'état du contrôleur de congestion... (un mot-clef en début de ligne est utilisé pour extraire uniquement l'information dont nous avons besoin et de manière standard entre les différentes piles).

Le fichier **SSLKEYLOGFILE**, généré par la librairie SSL/TLS permet de déchiffrer la capture (à condition d'avoir un dissecteur adapté aux modifications apportées). La capture centrale peut donc être déchiffrée avec le logiciel Wireshark, ce qui nous permet de comprendre encore une fois le fonctionnement interne avec les Frames, notamment d'acquittement qui nous renseigne sur les Sack.

La fonctionnalité la plus importante est d'avoir un "**referee**" de pertes pour comparer la vérité des pertes avec ce que la sonde calcule. Pour ceci, nous utilisons un "**Oracle**" qui prend en entrée les captures faites à chaque bout et celle au centre. L'Oracle classe les paquets entre les 2 flux (montant et descendant) puis liste tous les IP ID de ces paquets. L'IP IDentification est un champ de l'entête IP qui doit être unique pour une connexion donnée (il sert notamment au moment de la fragmentation des paquets IP trop gros en paquet plus petit pour pouvoir les réassembler à l'autre bout). De ce fait, tous les IP ID sont différents pour une expérimentation (donc 1 IP ID = 1 paquet transporté). Ainsi, une fois fais la liste de tous les IP ID dans les 3 captures dans des ensembles (au sens mathématiques) ont obtient le diagramme de Venn ci-dessous.



Les zones de l'ensemble :

1. Les paquets IP sont dans les 3 captures, donc pas de pertes
2. Paquets perdus sur Mid ==> Client (Serveur – Client & Sonde)
3. Paquets perdus sur Mid ==> Serveur (Client – Serveur & Sonde)
4. Impossible (la sonde ne produit pas de paquet)
5. Paquets perdus sur Client ==> Mid (Client – Serveur – Sonde)
6. Paquets perdus sur Serveur ==> Mid (Serveur – Client – Sonde)

Fig37 : Diagramme de Venn des IP ID dans les captures

Ainsi, l'oracle nous permet d'avoir une idée des paquets perdus et donc comparer avec les résultats obtenus par la sonde (ce qui est nécessaire pour la validation du logiciel avant la mise en production).

En annexe 9 est disponible le "README" du projet permettant d'expliquer le fonctionnement des différents modules et notamment de l'installation et de l'utilisation en mode CLI.

L'interface web

L'interface CLI est intéressante pour faire les premières expérimentations, mais ce qui est plus intéressant, c'est d'avoir une interface qui permet :

- de **configurer** facilement les expérimentations (comme vu précédemment avec les scénarios)
- d'avoir **la liste des scénarios et résultats** d'expérimentation
- de **lancer** les scénarios
- **d'ajouter un scénario** depuis un fichier
- **d'ajouter une capture** pour l'analyser
- Surtout, avoir accès aux données sous un format graphique qui permet de mener une **analyse** pertinente !

Pour toutes ces raisons, et aussi pour avoir accès aux systèmes facilement en interne, nous avons développé un site web (liste des pages en Tab5 et des captures d'écran en annexe 10). L'application est accessible en HTTPS uniquement depuis le réseau interne de l'entreprise (à travers un proxy). L'accès se fait via le site *https://lat6466:8080*, le flux est ensuite redirigé vers l'expérimentateur sur le port 8080 puis en local vers le port 5000 sur lequel attend l'application web via un « reverse tunnel » mis en place par la commande
$ ssh lili104 -L 0.0.0.0:8080:localhost:5000 -N -f.

Chemin	Titre	Description
/	Accueil	Liste des scénarios disponibles à gauche, liste des résultats pour analyse à droite
/create	Quic scenario maker	Création d'un scénario de d'expérimentation
/upload_scenario	Scenario uploader	Importation d'un fichier scenario
/upload	Capture uploader	Importation et analyse d'une capture (pour le troubleshooting)
/infos	Information	Page d'information sur Quic, les extensions et le testbed
/id/XXX	Analyse	Page d'Analyse des résultats de l'expérimentation XXX

Tab5 : Liste des pages web de l'outil d'expérimentation

Pour garder une certaine cohérence de développement avec les modules présentés précédemment, l'application web est aussi modulaire, indépendante et développée en Python. La librairie utilisée pour l'application web est **Flask**. Elle offre une simplicité de mise en place et est assez populaire dans la communauté Python.

La **page d'analyse** est véritablement la pièce centrale de cet outil, car elle permet de faire l'investigation des résultats de la capture. On y trouve certaines informations, mais surtout des graphiques montrant les informations du fichier résultat sous une forme commode. Cette interface est facilement scriptable car il suffit dans le code l'app d'ajouter un placeholder pour le graphique dans la page html et une fonction qui calcule le graphique pour ajouter une nouvelle représentation dérivée des résultats. Les graphiques sont obtenus à l'aide la librairie javascript « **Plotly** » qui permet d'obtenir des représentations dynamiques sur une page Web à partir de donnée au format json. Pour éviter de devoir régénérer tous les graphiques à chaque ouverture de la page d'analyse, une mise en cache est faite en sauvegardant le fichier pour plotly dans un fichier "graph.json". Un mécanisme de « recompute » permet de mettre à jour ce fichier et donc les graphiques.
En annexe 10 une capture de la page au 1-août est présentée.
Les représentations graphiques les plus importantes sont (dans haut en bas) :

- Le « **serpent** », cumulatif des tailles de paquet Quic dans chaque sens de la connexion
- L'**historique des bits SQR** au cours de la connexion
- Les **pertes calculées** au cours de la connexion
- Les **demi-RTT**
- La quantité de données en vol, **Bytes-in-flight** (ce qui nous donne une idée de la taille de la fenêtre de congestion si pas via les logs des contrôles de congestion)

De plus, la page nous permet de télécharger les différents fichiers liés à l'expérimentation s'ils sont présents (capture, résultat, graphique, scenario) et des informations sur le nombre de paquets, le format de l'entête, le débit, … Pour un traitement externe.

Nous détaillerons les graphiques que nous générons, leur intérêt en plus des conclusions des expérimentations dans la partie résultat.

5. Solution de sonde pour le traitement des résultats

Le fichier de résultats, celui dont les informations à afficher sont extraites, générées par le module PrepareData, a la particularité de dépendre des fichiers sources (quelles captures avons-nous accès, avons-nous accès au contenu ?) mais aussi de la sonde utilisée. La **sonde est l'élément fondamental de l'analyse** puisque c'est elle qui va, en fonction des bits SQR dans le flux de paquet d'entré déterminer les pertes sur ce flux. Avec 2 sondes (une dans chaque sens qui peut être différente en fonction des informations que nous souhaitons extraire) il est possible de déterminer les pertes sur les 4 segments.
Dans le code, les sondes sont des classes dérivées de la classe de base "qSonde" qui implémente 3 méthodes d'entrées et 3 méthodes de sortie ainsi que des membres internes.

Méthodes d'entrée :

- *add_spin*(timestamp, spinbit): Ajout du spinBit
- *add_square*(timestamp, squarebit): Ajout du squareBit
- *add_retransmit*(timestamp, retransmitbit): Ajout du retransmitBit

Méthodes de sorties:

- *get_rtts*(): Retourne une liste des rtts mesurés à l'aide du spin bit
- *get_retrans*(): Retourne une liste des pertes de bout en bout
- *get_slosses*(): Retourne les pertes Square, dernière méthode appelée à la fin de l'analyse.

Constantes:

- demi-Periode du bit square

Les membres internes:

- *spins*: liste des spins ajoutés à la sonde
- *spinned*: le train de paquet a changé de polarité au dernier paquet ajouté
- *squares*: liste des bits square ajoutés à la sonde
- *squared*: le dernier paquet est dans une sonde de front
- *s_losses*: liste des pertes de type square
- *i_square*: incrément interne de la sonde sur le square
- *retrans*: liste des bits retransmit ajoutés à la sonde
- *rtts*: les demi-rtt mesurés par la sonde

Avec cette abstraction de ce qu'est une sonde pour les 3 bits d'observations Quic, il nous est facile de développer une instance de sonde à partir de la base offerte pour mesurer au mieux les pertes et la latence, les 2 valeurs fondamentales pour le troubleshooting.

Sonde simple globale

Une façon simple globale de compter les pertes totales et le RTT moyen sur toute la capture est de :

- compter le nombre de changements de polarité du spin et divisé par le temps de la capture pour le RTT
- compter le nombre de Q jusqu'au dernier changement de polarité et faire le module SQUARE_PERIOD, s'il n'y a pas eu de perte, alors le modulo vaut 0, sinon c'est le nombre de perte en amont
- compter le nombre de R jusqu'au dernier changement de polarité du Q et soustraire au nombre de pertes amont pour déterminer le nombre de pertes avales

Malheureusement, cette méthode par moyenne sur l'ensemble de la capture est approximative et surtout, sur une capture de plusieurs méga octets, nous perdons l'information temporelle sur le moment des pertes et leurs conséquences sur le débit sans compter la variation du RTT.

Sonde simple fine (SimpleQSonde)

Dans un premier temps, nous avons développé la sonde la plus simple possible qui prend comme postulat que le train de paquet est toujours dans le bon ordre (ce qui arrive finalement peu souvent dans le réseau dû à la variance sur la latence). Les algorithmes sont les suivants :

Fonctions	Algorithmes
add_spin()	Si le spin actuel ≠ dernier spin alors ajouter le temps depuis le dernier spin dans la liste de rtt
add_square()	Si le square actuel ≠ dernier square alors ajouter SQUARE_PERIOD - i_square à la liste des pertes slosses (et i_square = 0) sinon incrémenter i_square
add_retransmit()	Si le retransmit = 0 alors ajouter l'incrément du nombre de perte de retrans

Cette méthode fonctionne plutôt bien dans le cas où il n'y a pas de réordonnancement dans le train de paquet (comme nous le verrons dans les résultats), permettant d'avoir une bonne approximation des pertes et des délais.
Pour autant, comme dit plus tôt, les réordonnancements, en particulier dans les phases de burst sont nombreux et donc cette première version nous donne des pertes sQuare totalement faussé voir des rtts faussés si des réordonnancement au moment des fronts...

Sonde de reconstruction du train de paquet (qfpost)

La seconde idée de sonde a donc été de **remettre dans l'ordre** autant que faire ce peu le train de paquet. De fait que nous n'avons pas accès au numéro de séquence (ce qui nous simplifierait la vie comme dit en introduction), nous devons reconstruire les créneaux, "recoller" les morceaux pour ensuite utiliser le même principe que la sonde simple. Pour ce qui est du RTT, un simple lissage sur plusieurs valeurs permet de supprimer les "pics" dû au réordonancement au moment des fronts.
Les algorithmes plus avancés sont les suivants :

Fonctions	Algorithmes
Init()	Paramétrer le N_limit
add_spin()	Si le spin actuel ≠ dernier spin alors ajouter le temps depuis le dernier spin dans

	la liste de rtt
add_square()	Si le i_square + 1 de la polarité > N : Si la polarité s'inverse ajouter le i_square à l'accumulateur sinon ordonner le signal pour « coller » le paquet au front précédent (au maximum eloigné de N_limit) Incrémenter i_square
add_retransmit()	Si le retransmit = 0 alors ajouter l'incrément du nombre de perte de retrans
get_rtts()	Appliquer une moyenne mobile de N sur le RTT
get_slosses()	Calculer les pertes pour toutes les périodes

Pour autant, le réordonnancement ne créé pas seulement de la confusion dans le signal Q, il en créer aussi dans le signal R car certaines piles, en fonction du contrôle de congestion et de la gestion des acquittements, peuvent considérer un paquet comme perdu alors qu'il n'en est rien et donc émettre un R qui se trouve être faux car le paquet n'a pas été perdu. D'où l'idée de corréler les 3 bits pour corriger les erreurs de la pile.

Sonde par corrélation triple (SondeCorre)

Au lieu de faire un traitement indépendant pour les 3 grandeurs fournies par les 3 bits, nous pouvons les corréler et ainsi être plus précis.
En effet, les pertes Q ne sont mesurables que par paquet de taille SQUARE_PERIOD (ce qui limite la précision de notre outil de mesure), les R ont au minimum 1.5 demi-RTT de retard sur les Q (1 demi-RTT pour le chemin client-serveur + [0~0.5] demi-RTT pour le chemin mid-client + 0.5 demi-RTT serveur-mid + le temps pour chaque pile pour envoyer un paquet), donc 1 RTT de manière simple. De même, le S nous donne une variation du RTT (qui peut indiquer une congestion en cas d'augmentation). Ces 3 bits se complètent et c'est pour cela que nous avons conçu une troisième sonde tentant d'utiliser ces propriétés. Malheureusement, cette sonde n'est pas encore au point au moment où nous écrivons ce rapport et donc l'algorithme n'est pas encore définitif.

Méthodes exploratoires (Traitement du signal, Machine learning)

Le développement étant assez libre pour répondre à des problématiques toujours plus complexes et particulières, nous avons utilisé une méthode exploratoire pour utiliser des outils mathématiques permettant de trouver de l'information sur la nature des pertes.

Une première idée se base sur l'utilisation du **traitement du signal** pour détecter des motifs.
Les pertes peuvent être très corrélées entre elles (comme vu dans la partie "perte") et peuvent être cyclique. Un effet d'entrée en résonance au niveau d'un équipement peut survenir lorsque les utilisateurs de ce nœud mettent en place les mêmes stratégies de contrôle de congestion. Le remplissage de la file se comporte alors comme un oscillateur, entraînant une rafale de perte à chaque dépassement de la pile. Ce genre de comportement d'autocorrélation peut être détecté par changement de référentiel, du temporel au spectrale par transformation de Fourrier.
Pour ceci, nous transformons le signal carré Q en signal sinusoïdale avec déphasage lorsque la polarité change. En appliquant la transformée de Fourrier sur ce signal sinusoïdal nous obtenons une signature spectrale du signal Q sur la connexion que nous pouvons comparer avec le signal pur. Un signal sinusoïdal pur donnant un dirac à la fréquence f0, il est facile de nettoyer le signal obtenu pour trouver les perturbations dans ce signal.

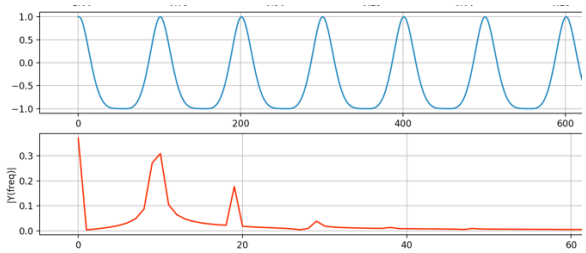


Fig38a : Signal Q sans réordonnancements après traitement

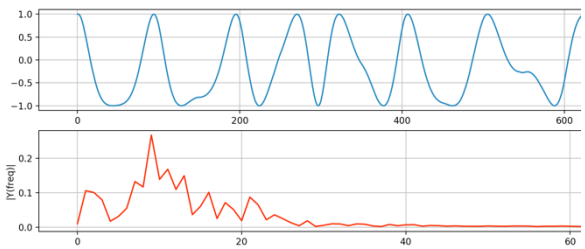


Fig38b : Signal Q bruité après traitement

Pour autant, nous n'avons pas réussi à développer une sonde vraiment intéressante avec de réels avantages qui soit utilisable en production.

Une seconde idée se base sur l'utilisation du « machine learning » pour apprendre à détecter les pertes en fonction des 3 signaux SQR et possiblement d'autres features comme le RTT et le cumul de donnée. Le machine learning, en s'entrainant sur des milliers d'exemple peut comprendre des corrélations que nous humain ne verrions pas. Le problème de cette méthode est qu'il est nécessaire de constituer des datasets de centaines d'exemples classifiés, ce qui est une tâche très longue à réaliser. C'est pour cette raison que nous n'avons pas développé cette solution, pour autant, il s'agit d'une bonne idée de développement, surtout si les bits SQR deviennent très utilisés.

F. Outils de troubleshooting

Jusqu'à maintenant nous avons surtout parlé des outils qui permettent de faire des expérimentations pour valider la solution des SQR bits pour le debugging réseau.
En réalité, l'outil et l'interface se veulent plus utile encore en permettant de faire du **troubleshooting réseau**. En effet, avec l'interface "importation de capture", il suffit de définir la sonde à utiliser, la demi-période du square (par défaut 64, comme définit dans la norme), les IP des bouts pour procéder à l'analyse d'une capture réseau.
Ceci permet de faire une première analyse et par dichotomie trouver l'endroit de la panne. C'est gagné, avec quelques points de capture, quelques analyses avec l'outil, le trafic Quic perd de son obscurité en permettant le troubleshooting.

IV. Résultats et analyses

A. Observations générales

Le temps de l'expérimentation

(Moyenne sur 9 expérimentations avec un transfert de 1Mbit)

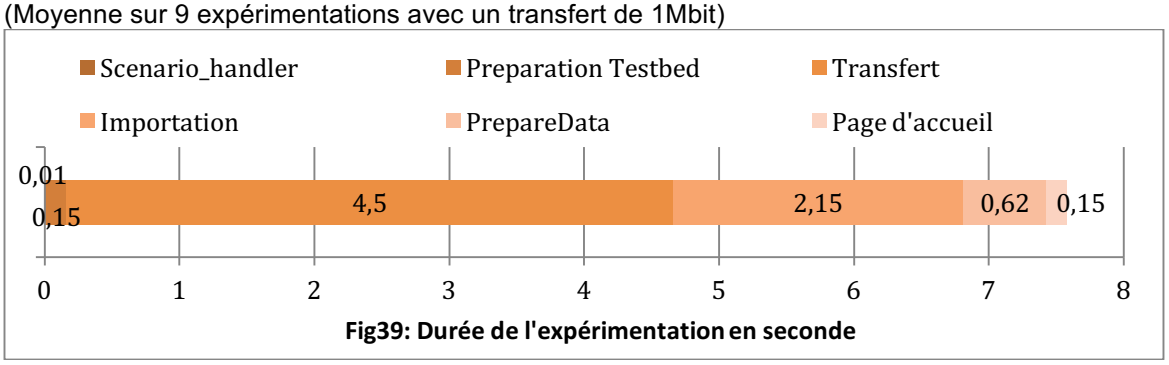


Fig39: Durée de l'expérimentation en seconde

Le temps de l'expérimentation est assez court, en moyenne **7.58 s**, ce qui permet de réaliser au moins 6 expérimentations par minute et donc potentiellement plusieurs centaines d'expérimentations par Heures.
Finalement, étant donné que les expérimentations sont courtes à réaliser, la plus grosse partie du travail est la conception du scénario et l'analyse des résultats via les représentations graphiques obtenues.

Les représentations graphiques obtenues

Dans la suite de l'étude, nous utiliserons les représentations graphiques présentées ci-dessous. Ces représentations sont obtenues directement des donnés résultats de **prepareDate** (*res.json*) ou en sont dérivés.

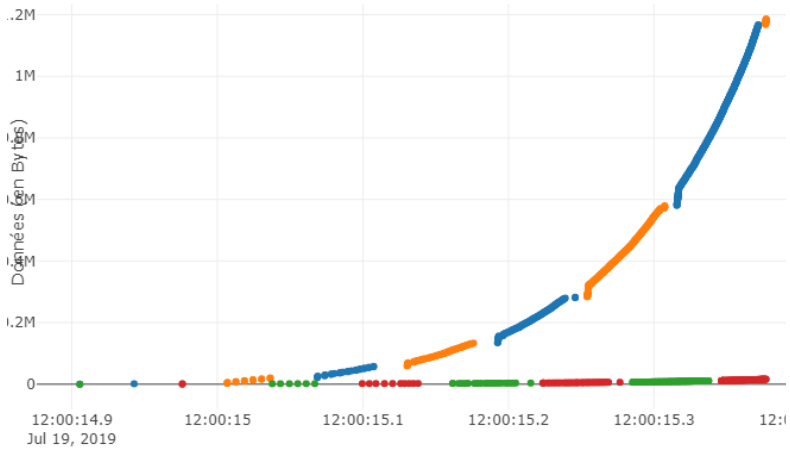


Fig40 : Graphique « serpent »

Le premier graphique dit « **serpent** » représente le cumul de la taille des paquets Quic au cours de la communication. Il est un peu équivalent au graphique des « seq » TCP, mais n'apporte pas la même information puisque le cumul C_{tot} ne peut qu'augmenter (et donc, ne permet pas de voir les données retransmises).

Il y a 2 courbes avec 2 couleurs. La première Orange et Bleu représente le serpent pour le cumul dans le sens Serveur-> Client (download) avec les paquets marqués bleu pour un spin à 0 et orange pour un spin à 1. Nous avons choisi de représenter le spin, car il est un bon indicateur visuel de la quantité de données en vol (bytes-in-flight) et du temps d'aller-retour (RTT). En rouge et vert, le cumul pour le sens serveur-> client. Étant donné la nature de la communication émulée (le téléchargement d'un fichier de N octets), la majorité de la communication se fait dans le sens Download. Dans le sens upload, uniquement la requête initiale puis les frames d'acquittement et de mise à jour (97 % des données sont dans le sens descendant pour un transfert de 100 ko, jusqu'à **99.5 %** pour un transfert de 10Mo).

Cette représentation du cumul nous permet aussi de voir les burst et les examiner précisément avec un zoomn car les représentations sont dynamiques grâces à la bibliothèque Javascript **Plotly**.

Ce graphique nous permet aussi de connaître le débit de la connexion par différentiation finie ($Debit = \frac{C_{n+1}-C_n}{t_{n+1}-t_n}$). Ainsi, sur l'exemple, le débit augmente linéairement au cours de la connexion puisque la courbe du cumul ressemble à une courbe de la fonction quadratique (un outil nous permet sur les représentations dynamiques de cliquer entre 2 points pour connaître le débit moyen entre ces 2 points).

Fig41 : Graphiques « bits SQR »

La seconde représentation est celle des bits SQR **observés** dans la capture (avant traitement par la sonde) en fonction du temps. De haut en bas le Spin, le sQuare, et le Retransmit. Par cette représentation, le comportement différent et complémentaire des 3 bits sont visibles. Le premier bat au rythme du rtt, le second tous les N paquets et le dernier montre des Dirac pour chaque paquet perdu.

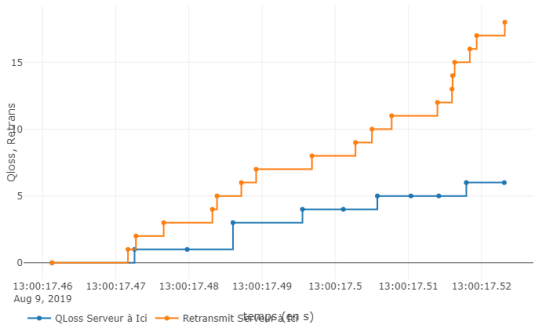


Fig42a : Graphique des pertes calculées par les sondes à l'aide des bits Q et R

La troisième représentation montre l'évolution des **pertes calculées par la sonde au cours du temps**. La courbe Orange représentant le nombre de pertes de bout-en-bout (retransmission) et la bleue les pertes en amont (sQuare) en fonction du temps.
La courbe de pertes de bout-en-bout majore globalement la courbe des pertes avale, tout simplement car nous ne pouvons pas perdre des paquets en amont et les retrouver en aval de la sonde. Pour autant, cette affirmation n'est pas vraie tout le temps, car comme dit plus tôt, les pertes Q sont détectées de manière directe par la sonde (comptage direct après le segment) alors que les pertes de bout en bout sont connues avec un retard. Ainsi, momentanément à un front du signal Q, des pertes amont détectées par Q ne sont pas encore parvenus à la sonde via R.
Une autre observation, les 2 courbes n'ont pas la même **granularité** de détection. Ainsi, la courbe des Retransmissions permet d'avoir chaque perte de bout en bout une fois détectée par la pile émettrice alors que la seconde ne permet que d'avoir les pertes globalement durant une demi-période et toutes les demi-périodes. Ce qui pose la question de la taille de la période. Pour autant, comme discuté précédemment avec les sondes, il est tout à fait possible de raffiner le Q avec le R pour réduire cette granularité dans la représentation. Déjà, visuellement, il est possible de comprendre les pertes comme illustrer sur la figure ci-dessous :

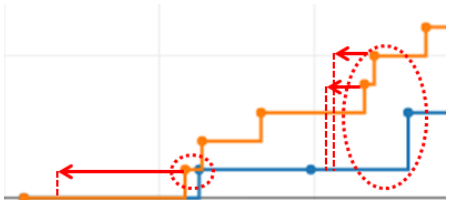


Fig42b : zoom sur les courbes de pertes

Avec les demi-RTT extrait du spin, nous pouvons replacer approximativement le moment des pertes (les flèches représentant les délais et donc le moment des pertes réelles sur la courbe des Q). Cette technique ne fonctionne pas à tous les coups, par exemple s'il y a une perte aval et 2 pertes amont dans une demi-période, il est impossible de les différencier…

Une quatrième représentation graphique très utile est représentée ci-dessous, celle des pertes de bout-en-bout en fonction des pertes Q en aval.

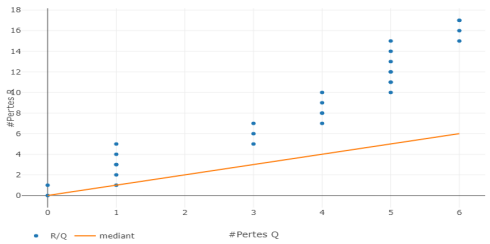


Fig43 : Graphique des pertes R au regard des pertes Q

Ce graphique est directement issu des données de la dernière représentation des pertes. Il met en regard les pertes R et les pertes Q. Chaque point est forcément seul sur ces coordonnées, car les pertes ne peuvent qu'augmenter. Donc chaque nouveau point est forcément à droite de Nq ou au-dessus de 1 du précédent. Ainsi, nous pouvons retracer la vie de la connexion en se déplaçant du point 0,0 de bas en haut puis de gauche à droite. Nous pouvons aussi y repérer des zones remarquables comme décrites sur le schéma ci-après (Fig44).

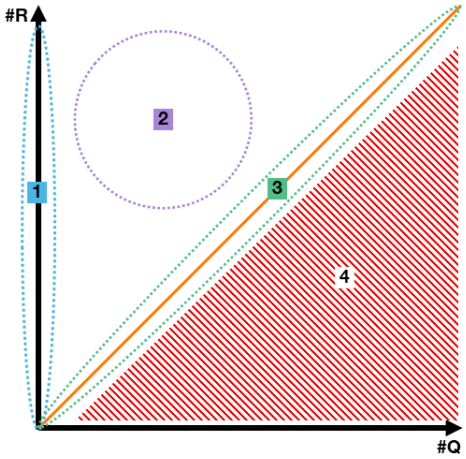


Fig44 : Schéma d'analyse de la courbe R/Q

1- Zone des pertes en aval pure
2- Zone de pertes en aval et amont
3- Zone de pertes en amont pure
4- Zone impossible, si des points sont présent dans cette zone, cela signifie que certaines pertes Q n'ont pas été comptabilisé

Si la ligne de Q=R forme un angle de $\frac{\pi}{4}$ avec R=0 et que Q=0 forme un angle de $\frac{\pi}{2}$, alors la ligne formant un angle de $\frac{3\pi}{8}$ représente le cas particulier d'un nombre égale de perte en amont et en aval de la sonde. Si les points sont en dessous de cette ligne, alors les pertes sont plutôt amont, sinon, elles sont plutôt en aval.

Ce graphique est le plus important pour analyser les pertes, car il nous donne l'information du taux de perte (en divisant par le nombre de paquets de la connexion) ainsi que la nature de ces pertes au cours de la connexion étudiée.

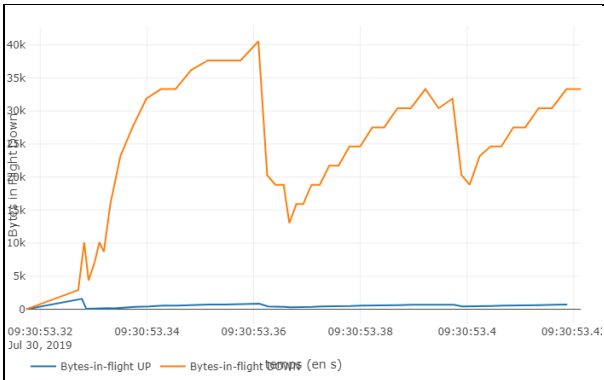


Fig45a : Graphique des Bytes-in-flight

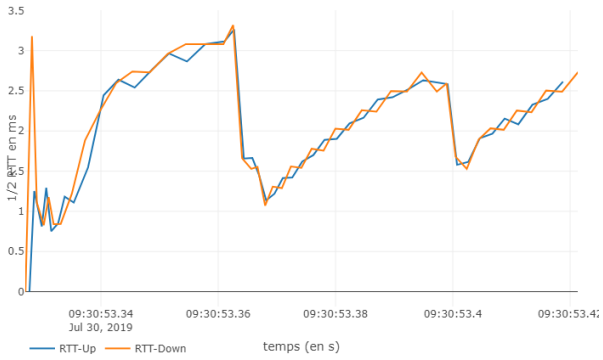


Fig45b : Graphique du RTT

Ces 2 graphiques en Fig45 donnent pour chaque sens la quantité de données en vol et les demi-RTT. Le RTT est le second outil utile pour le troubleshooting (avec le taux de perte) et est calculé avec le Spin bit. Le bytes-in-flight est intéressant, car il nous donne une idée de la fenêtre de congestion, fenêtre de congestion qui est un bon indicateur du type d'algorithme de congestion à l'œuvre une fois mis en regard avec les pertes détectées précédemment.

L'overhead Quic

L'overhead correspond au rapport entre la quantité de données totale transmises sur la quantité de données utiles transmises. Sur nos graphiques serpents, l'overhead Quic correspond au cumul des données Quic transmises sur la quantité qui a été réellement transmise. Nous connaissons aussi la quantité de données transmises. Nous obtenons un overhead en moyenne, sans retransmission de **4.71 %** pour une communication de 100 ko. Étant donné qu'au cours de la connexion, les paquets Long Header sont plus rare, le rapport quantité de donnée utile transmises devient plus important jusqu'à atteindre 2.9 % pour les plus grosses transmissions. En ajoutant les 24 octets de l'entête UDP et IP, l'overhead de la pile Quic (à 2.9 % en moyenne) est plus élevé que celui de TCP qui est de 2.8 % en moyenne. Cependant, cet overhead « par paquet une fois la connexion établie » ne prend pas en compte le coût de l'établissement. En effet, en 0-RTT ou 1-RTT, le coût en donnée de la mise en place d'une connexion Quic est beaucoup moins élevé que le coût de la même opération en TCP/TLS. Ce qui confirme Quic comme plus avantageux pour les connexions courtes.
Nous n'avons pas eu l'occasion de le vérifier par nous-mêmes, mais il s'avère que grâce aux divers mécanismes de contrôle de congestion, de multistream, … Quic obtient toujours une **meilleure utilisation de la bande passante** que TCP [21], ce qui contrebalance l'overhead légèrement plus élevé.

Fichier scenario crédible pour la simulation réseau

Un scénario crédible est un scénario qui reproduit ce qui peut être observé sur les réseaux. Ce que nous recherchons à reproduire est certes les cas où tout se passe bien pour avoir un point de référence, mais surtout les différents cas où cela se passe mal.
Pour déterminer ces scénarios, nous nous sommes basés comme vu précédemment sur les chiffres de pertes, de délais, de débits de papier de recherches, d'exemples, des chiffres d'Orange… Puis nous exécutons le scénario et analysons le comportement pour le comparer à la réalité.

Dans la **suite des expérimentations**, sauf indications contraire, la pile utilisée est picoquic avec le contrôle de congestion par défaut de Quic (newReno), N=64 (demi-période du signal Q) et un transfert de 1 Mo.

B. Influences de l'environnement réseau

Les taux de pertes, le modèle de pertes

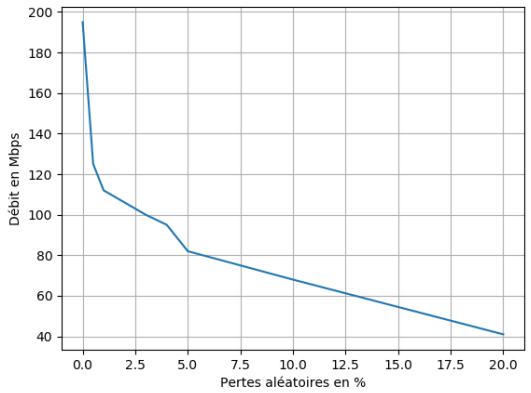


Fig46a : Graphique influence des pertes sur le débit de la communication

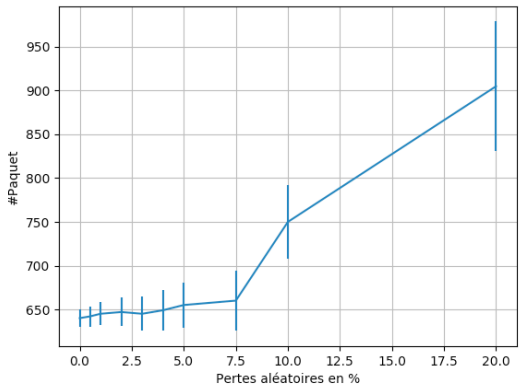


Fig46b : Graphique influence des pertes sur le nombre de paquet transmis

Nous avons d'abord étudié l'influence des pertes aléatoires sur les différents paramètres de la communication. Pour cela, le scénario est le transfert de 1 Mo, sans délais, avec un débit maximal et les pertes aléatoires comme paramètre.

Première Observation, le débit qui **décroît assez rapidement avec les premières pertes** puis plus lentement après 5 %. Même à 10 % de perte (ce qui est extrêmement élevé), le débit n'est que divisé par 2.5 par rapport au cas sans pertes. Lorsque nous regardons le BIF, la fenêtre de congestion atteint 25 ko avant que ne survienne la première perte puis n'est plus que de 8 ko (soit environ 5 paquets) ce qui induit une fréquence pour le spin bit très élevée (puisque le nombre de paquets en vol est très faible, beaucoup d'aller-retour sont fait pendant la communication). C'est le fait que peu de paquets sont en vol (pour éviter les pertes) que le débit est tant réduit. Si nous introduisons alors des délais, le débit s'effondre violemment…

Seconde observation, le **nombre de paquets nécessaire à la transmission**. Nous observons qu'au de-là de 10 %, le nombre de paquets nécessaire à la transmission de 1 Mo devient très élevé. Pour autant, dans une situation de perte faible, le nombre de paquets n'évolue pas beaucoup. De 635 en moyenne à 650. Tout d'abord, à 1 % de perte, nous pouvons s'attendre à perdre 7 paquets (7*1,400 octets pour une payload Quic en moyenne = 10 ko soit environ 1 % de 1 Mo). C'est assez peu, peu significatif même en répétant l'expérience sur plusieurs expérimentations. Pour autant, à 5% nous devrions voir une différence significative par rapport à 0 %, or ce n'est pas le cas.
Cela s'explique par le fait que la taille moyenne des paquets augmente, que les paquets ne sont pas renvoyé tels quels une fois perdu, les données sont rempaquetées à côté d'autres données et que finalement le contrôle de congestion essaie de diminuer les pertes en diminuant les données en vol, ce qui diminue le nombre de données à renvoyer.

Sauf qu'en général, les pertes ne sont pas aléatoires au cours de la communication, mais corrélées avec des « burst » de pertes. Pour ça, nous appliquons sur l'interface eth1 (donc pour observer des pertes de download en amont) des pertes suivant le **modèle Gilbert-Elliott**, plus simple à configurer que le modèle de Markov. En effet, nous avons trouvé beaucoup de probabilité crédible pour Gilbert-Elliott et peu pour Markov dans la littérature.

Avec les valeurs p=0.02, r=0.4, 1-h = 1 et 1–k = 0, nous obtenons un comportement ressemblant à des pertes par rafale. Le modèle GE peut être vu comme une chaîne de Markov avec une matrice de transition$P = \begin{pmatrix} 0.98 & 0.02 \\ 0.60 & 0.40 \end{pmatrix}$. En résolvant le système $q(I_2 - P) = 0$ où q est le vecteur de probabilité, I la matrice identité 2x2 et P la matrice de transition, nous obtenons la probabilité de temps passé dans un état. Le calcul nous donne qG = 0.968 et qB = 0.032. La probabilité totale de perdre un paquet est donnée par l'équation$Loss = (1 - h) * qB + (1 - k) * qG$. La probabilité de pertes attendues est donc de 3.2 %.

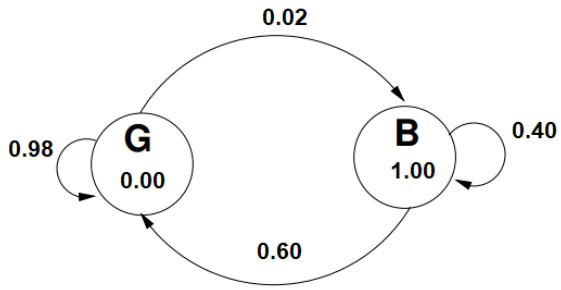


Fig47a : Le modèle utilisé

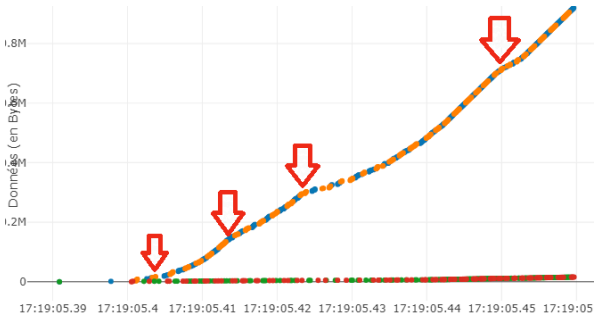


Fig47b : Le serpent donne un débit de 108Mbps

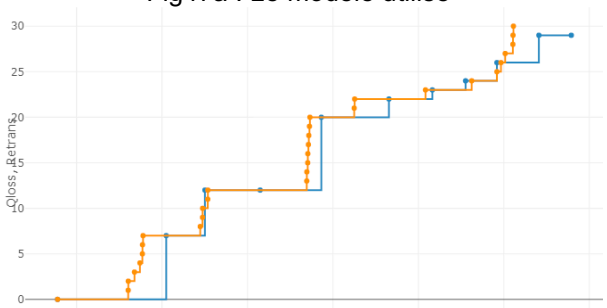


Fig47c : Le cumul des pertes R et Q

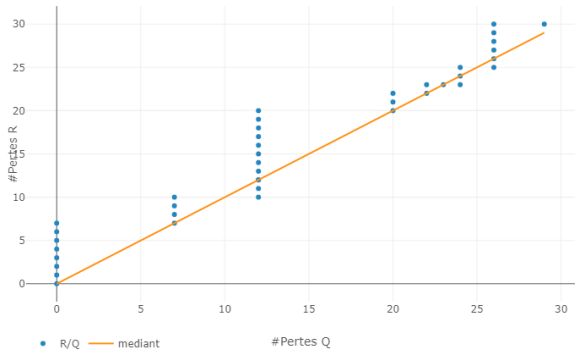


Fig47d : Le R/Q nous indique que les pertes ne sont qu'en amont de la sonde

Nous obtenons une perte Q = 29 soit 4.1 %, proche des 3.2 % attendu, ce qui confirme le fonctionnement du modèle.
Sur le cumul des pertes, nous observons bien que toutes les pertes (excepté une) sont en amont comme voulu et surtout se font en rafales espacées de quelques paquets. La pile semble peu perturbée par ces pertes avec un débit de **108 Mbps**. Sur le graphique serpent, nous voyons clairement les pertes puis la reprise (nous pouvons aussi le voir sur le bif).

Influence des délais

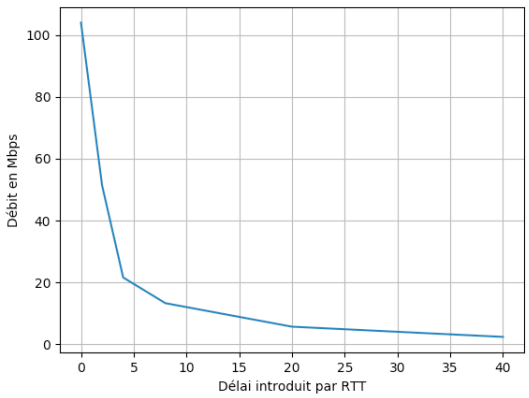


Fig48 : Graphique du débit d'une connexion à 2 % de perte en fonction du délai introduit par RTT

Les délais sont un facteur important de diminution du débit, en effet, ils augmentent le RTT (ce qui n'a pas d'influence lorsqu'il n'y a pas de pertes), ce qui augmente le délai de retransmission et donc la capacité de retransmission en cas de pertes.
Sur le graphique en Fig48, nous avons mené des expérimentations sur l'influence des délais introduit sur une connexion avec 2 % de pertes. À 0ms introduit, nous observons un rtt de l'ordre de 1 ms, dû notamment au perturbateur et nous retrouvons bien le 100 Mbps d'une connexion à 2 % de pertes sans délais. Le débit moyen de la connexion décroît de manière exponentielle, ce qui nous confirme l'influence première des délais sur une connexion déjà perturbé par des pertes, alors même que nous avons vu précédemment, qu'autour de 2 % de pertes, le débit reste stable et élevé.

Les Réordonnancements, modélisation et observations

Nous avons utilisé le Jitter pour modéliser les réordonnancements, car le reoder en pourcentage est moins réaliste. En effet, le « reorder » netem va retarder un certain pourcentage de paquet de N ms. Or, en réalité, les réordonnancements dans le réseau sont dû à des différences de trajet qui ralentissent ou accélèrent les paquets autour de la moyenne du demi-RTT. Donc le jitter, suivant une loi normale permet de mieux rendre compte de ce phénomène.

Avec un Jitter faible (=0.2ms) au regard du délai (=5ms) pour chacune des interfaces nous obtenons la page de résultat suivante.

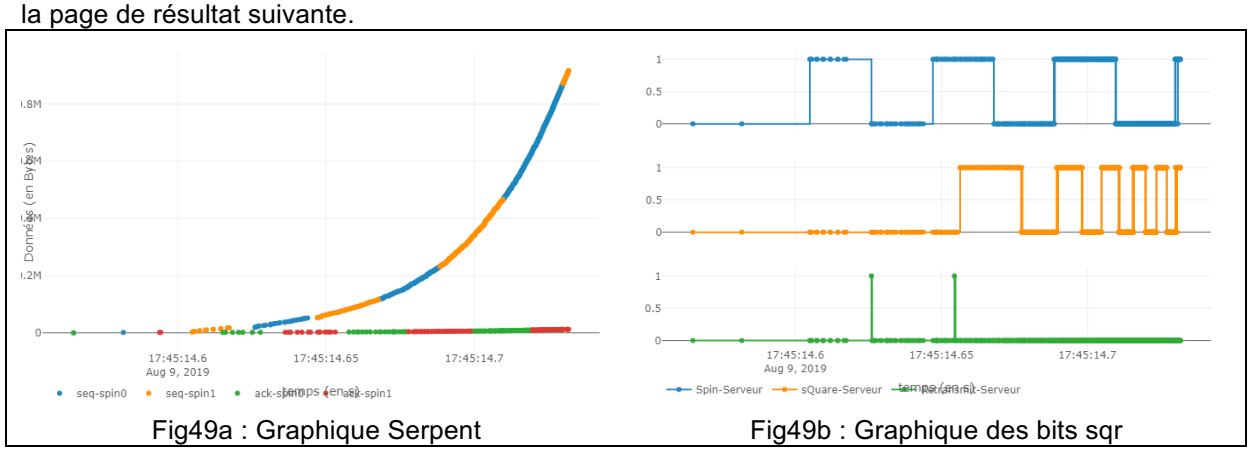


Fig49a : Graphique Serpent

Fig49b : Graphique des bits sqr

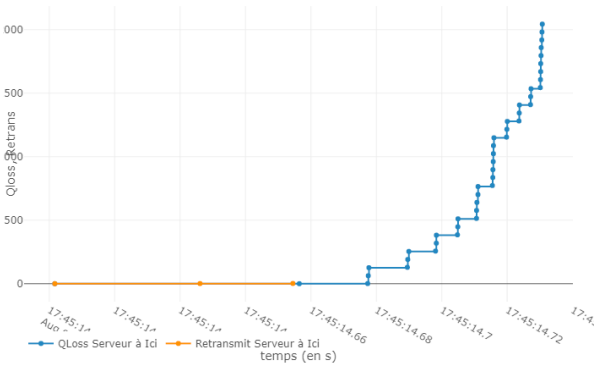


Fig49c : Graphique des pertes

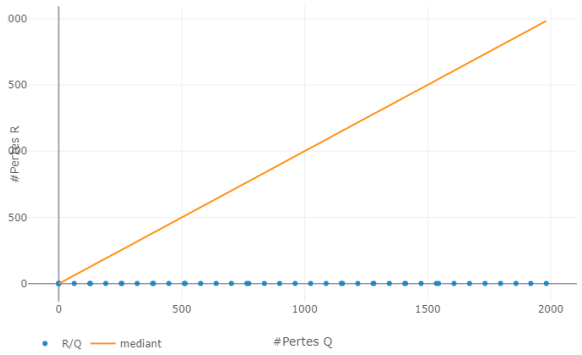


Fig49d : Graphiques des #R par rapport à #Q

Tout d'abord, avec un jitter faible, la probabilité qu'un paquet passe devant un autre est faible, notamment au début de la connexion où le contrôle de congestion ne permet pas l'envoie de trop de paquet par RTT. La probabilité devient plus élevée en phase de burst et en fin de connexion, une fois le débit maximum atteint. D'ailleurs, sur le serpent, nous n'observons plus les effets de « rattrapages » avec des pics de débit, ces paquets étant très rapprochés, ils se confondent avec les paquets autour.
L'analyse des pertes a été réalisée avec la sonde simple fine. Cette sonde détecte beaucoup de pertes en amont (pertes Q, 2044 pour une connexion de 729 paquets) mais seulement 2 de bout en bout (pertes présumées par la pile, même s'il doit s'agir de fausse détection, seulement dû à une variation importante de RTT). Si nous zoomons sur un changement de polarité en fin de connexion, nous observons un effet de bruit sur les fronts du S et du Q.

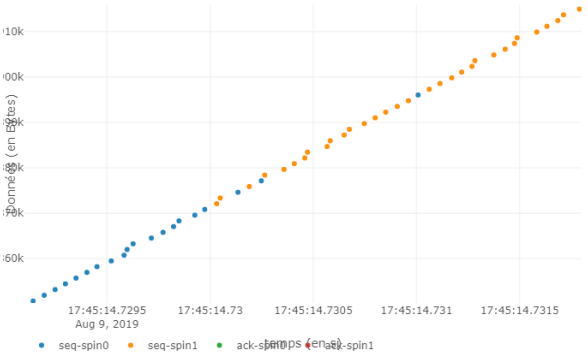


Fig49e : Zoom sur le serpent avec le jitter

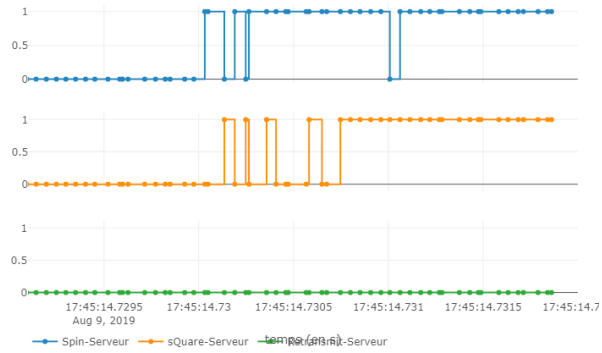


Fig49f : Zoom sur les bits SQR avec Jitter

La sonde simple, au vu de son algorithme des pertes Q, va tout simplement considérer qu'il manque 63 paquets de la même polarité à chaque changement (d'où les presque 500 paquets considérés comme perdus).
En utilisant la sonde qfpost, le signal S est reconstitué et ne trouve plus de pertes. Le problème est le même avec le spin qui, à chaque changement d'état donne des valeurs de RTT et de BIF erronées.

En faisant varier le Jitter (le délai de 20 ms total par RTT constant), nous observons l'influence sur les pertes détectées par la pile et le débit.

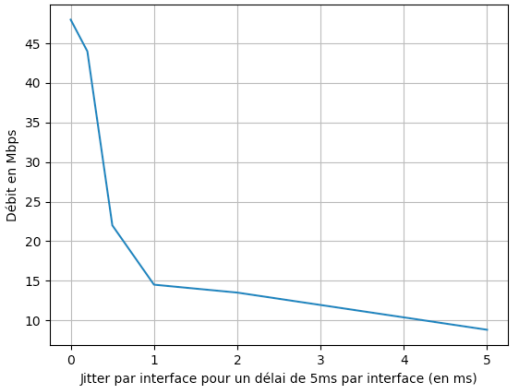


Fig50a : Débit en fonction du Jitter

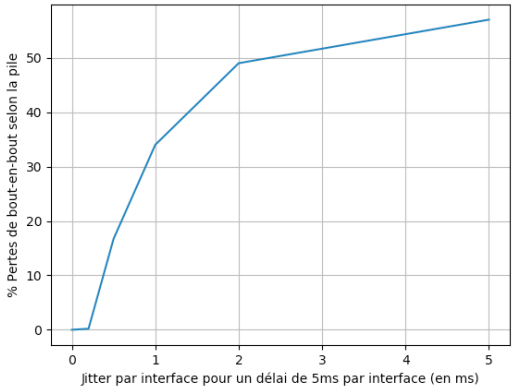


Fig50b : Pertes estimées en fonction du Jitter

Au-delà d'un Jitter de 0.4 ms pour un délai de 5 ms sur chaque segment, la pile commence à avoir du mal à déterminer les pertes réelles des réordonnancement. D'ailleurs, au-delà de 0.5 ms, la sonde qfpost ne fonctionne plus correctement et commence à avoir du mal à « recoller » les morceaux… On peut donc considérer qu'au-delà d'une **variance supérieure à 10 % du délai**, ni les sondes ni les piles ne réagissent correctement pour détecter les vraies pertes de paquet.

Les autres types de perturbations, corruption, duplication

Les **corruptions** sont considérées comme des pertes par les bouts, donc déclenchent des Retransmit. Par contre, à part, si le paquet est trop modifié, et donc pas reconnu comme un paquet Quic, la sonde va compter le paquet. Ainsi, nous ne pouvons pas déterminer si la corruption a eu lieu avant ou après la sonde. De tout de manière, en général, les paquets corrompus sont jetés dans le vrai réseau une fois détectés.

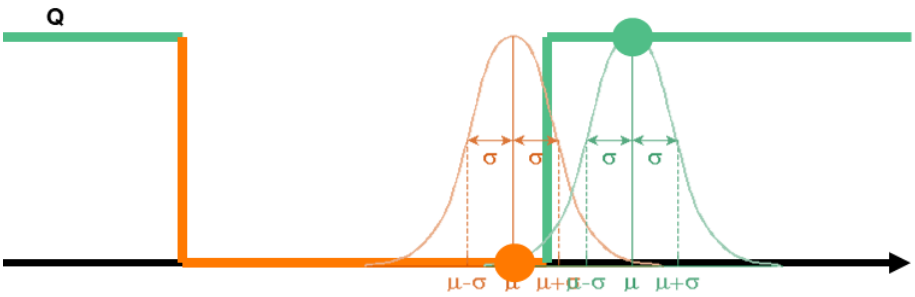
Les **duplications** sont en générales bien gérées par les piles et ne déclenchent pas de retransmissions. Par contre, si elles surviennent en amont de la sonde, le nombre de paquets avec la même polarité va être supérieur à N (sauf si des pertes sont plus importantes) ce qui nous donne des pertes négatives en amont. Pareil, nous ne pouvons pas faire grand-chose pour détecter un paquet dupliquer à part de conserver une trace de tous les paquets et vérifier que 2 traces ne sont pas identiques. Puisque les duplications sont des phénomènes assez rares, nous ne résoudrons pas ce problème.

Choix de la valeur de Q

Une des grandes problématiques à résoudre durant ce stage est la valeur N de changement de polarité du bit Q (et donc la fréquence f=2N du signal carré). Dans la norme actuelle, N=64.
Le testbed permet de modifier la valeur de N pour mener des expérimentations sur l'influence du N sur la mesure des pertes.
Avant d'expérimenter, il convient d'énumérer les facteurs influant sur le choix de N.

- La **longueur de la communication** : Une communication courte comporte peu de paquets. Donc un N trop élevé et une période correspondra à l'entièreté de la connexion, perdant l'information fine des pertes amont.
- Le **réordonnancement** : Dans un réseau avec un fort effet de Jitter, un N trop petit induit une impossibilité de reconstruire le signal, puisque les fronts sont très rapprochés les uns des autres
- En cas de **pertes fortement corrélées**, un N trop petit, pourrait nous empêcher d'avoir une bonne granularité pour observer les burst de pertes, les rendant plus indiscernables les uns des autres.
- l'**écartement entre les paquets** : Lors d'une période de burst, les paquets sont émis de manière très rapprochée. Il est possible de perdre tout ou partie des paquets. Un N trop petit, et il peut être carrément perdu tous les paquets d'une polarité. Inversement, un N trop grand, et la granularité ne permettra pas de déterminer que les pertes ont eu lieu pendant le burst et non pendant la période de calme suivant.

Pour une communication à D octets/s, (environ T/1500 paquets par seconde), l'écartement moyen entre 2 paquets est de 1500/D seconde. La durée t en s d'une polarité du bit Q est donc de $t = \frac{1500*N}{D}$. Avec N = 64, D=10 Mo/s, t = 9.6 ms. On considère qu'un signal est trop bruité si 50 % paquets de la polarité sont dans la zone des bords floutés (25 % au début et 25 % à la fin).



Le schéma Fig51 ci-contre permet de se rendre compte de l'influence d'un jitter suivant une loi normale (jitter est une variance, représentée ici par σ) sur la position probable du paquet dans le train.

Fig 51 : Effet du Jitter, probabilité de position d'un paquet dans le signal Q

Le paquet a 68 % de chance de se retrouver dans la zone comprise entre μ-σ et μ+σ. Ainsi, si 2 paquets se trouvent dans la zone [μ-σ ; μ+σ] de l'un de l'autre, ils ont une forte probabilité de s'intervertir. Plus σ est grand, plus la probabilité de permuter les positions est importantes (cette probabilité ce calcul de la même façon que pour la superposition de fonction d'onde). Un calcul statistiques poussé serait nécessaire pour trouver la valeur exacte du Jitter pour laquelle 50 % du signal est bruité. En première approximation, nous considérons que pour l'intervalle à 68 %, 2σ= ¼ D, le signal est bruité. Avec notre exemple, le signal est bruité pour un Jitter = ¼ *3.2 = 0.4 ms. Nous retombons sur les ordres de grandeur observée précédemment pour ce débit (la retransmission étant un signe que même la pile considère le train de paquet est trop confus).

Pour ce qui est de la taille des communications

Selon le rapport d'http archive [22], la taille moyenne d'une page web est de 1.9 MB (beaucoup plus lorsqu'il y a de la vidéo ou du son). Donc nous pouvons nous baser sur cette valeur pour la taille d'une connexion moyenne. Pour N=64, cela correspond à environ $\#P = \frac{Taille_{communication}}{Charge_{utile}/_{paquet}} * f = \frac{1,900,000}{1,400} * \frac{1}{128} \approx 11$ où **21** créneau avec la même polarité.

Pour ce qui est de la granularité

En faisant varier la demi-période N pour le modèle GE vu précédemment nous obtenons des graphiques R/Q montrant l'influence de la granularité de la mesure.

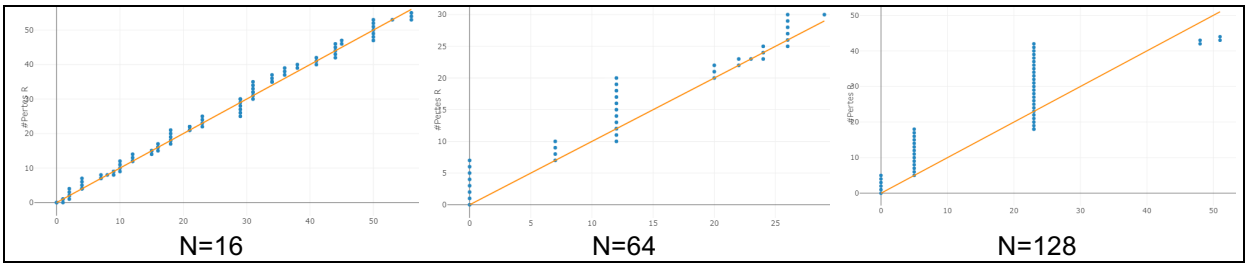


Fig52 : R/Q pour différentes valeurs de N

Avec un N petit, il est aisé de remarquer que toutes les pertes sont en amont alors qu'avec un N plus élevé, la déviation vers les R ascendants rend plus difficile cette analyse. De par cet exemple, nous serions tentés de dire que N=16 est très intéressant, mais c'est sans compter le Jitter.

Pour ce qui est du Jitter

Pour expérimenter l'influence du Jitter nous avons repris l'expérimentation du Jitter précédente avec σ=1ms, nous avons fait varier la valeur de N et observé le signal brut AVANT traitement. Nous prenons σ=1ms (par interface et pour un délai de 5ms) car nous considérons qu'il s'agit d'une borne

haute à partir de laquelle de tout de façon, le train de paquet est considéré comme brouillé par la pile Quic.

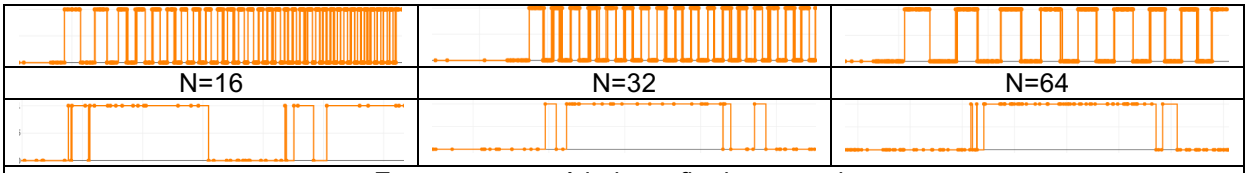


Fig53 : Signal Q avant traitement pour un Jitter de 1ms et différentes valeurs de N

Sont représentés (en Fig53) les signaux Q observés pour les différentes valeurs de N. Il est clair que le signal pour N=16 est très difficilement lisible quand celui de N=64 semble presque parfait. Une fois zoomé sur une période en fin de connexion (le moment où le débit est maximal, donc les créneaux les plus rapprochés), le signal N=16 semble vraiment bruité et pas facile à reconstruire (est-ce que le 1 seul va avec le créneau de 1 d'avant ou le suivant). Les créneaux de N=32 et N=64 ont certes les bords bruités, mais il est assez aisé de reconstruire le train de paquet (ce que nous confirme la sonde qfpost en reconstruisant presque entièrement le signal).
La mesure de la largeur approximative d'un créneau nous donne 9.7 ms pour N=16, 16.5 ms pour N=32 et 36.7 pour N=64 (bien sûr, puisque le débit n'est pas le même, $T_{32} \neq 2T_{16}$). Pour autant, nous pouvons en conclure que la limite basse du temps que doit durer un créneau est de **10 ms.**

Par la formule $N_{opt} = \frac{10^{-2}D}{l}$ (où D est le débit en Octets/s et l la taille moyenne des paquets ≈ 1500), nous obtenons pour un débit de 10Mo/s, N=66. En réalité, avec un Jitter considéré comme « non-perturbatif » par la pile, nous avons calculé de la même manière qu'un créneau peut ne durer que **5 ms** pour être utilisable par une sonde ce qui nous un **N=33**.

Jitter et pertes

Nous avons ensuite mené une expérimentation avec 0 ou 2 % de pertes avec 3 valeurs de N et pour différentes valeurs de Jitter (0, 0.1, 0.5 ms pour 5 ms) avec un traitement par une sonde forte aux réordonnancements. Les données des 180 expérimentations sont compilées dans le graphique en Fig54.

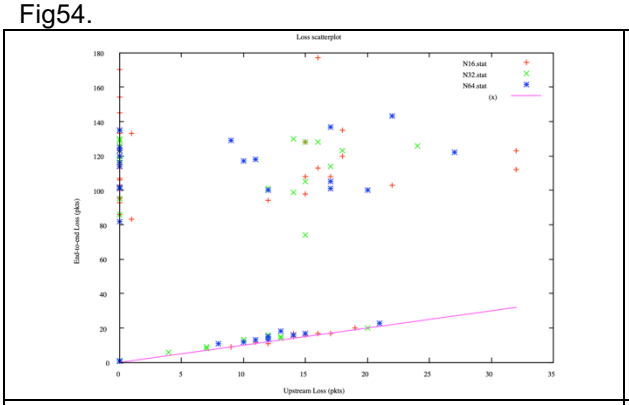


Fig54a : R/Q pour différentes valeurs de N pour différentes expérimentations

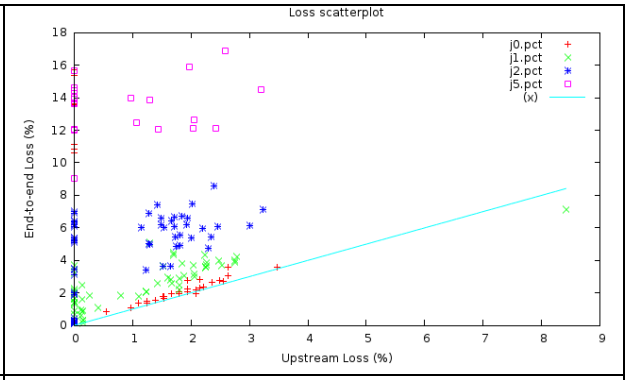


Fig54b : R/Q en fonction du Jitter (0.0, 0.1, 0.2, 0.5) avec 2% de pertes

L'analyse détaillée montre que le mécanisme QR fonctionne bien et qu'une valeur de 32 est bonne.

Conclusion

Il nous semble que 64 est un bon N pour des connexions à très haut débit, mais pour un débit moyen, dans un compromis de granularité de la mesure/justesse de la mesure, N=**32 semble plus adapté.**

Pour autant, il n'est pas impossible d'imaginer un N choisis une fois pour toute au début de la connexion (puisque le Q n'a pas besoin de la collaboration de l'autre bout) parmi des valeurs préétablies dans la norme, par exemple, [16, 32, 64] (avec une valeur par défaut). Ceci permettrait au bout d'adapter ce signal en fonction du type de communication qu'il s'attend à avoir. Par exemple, un serveur de streaming vidéo pourra choisir un N à 64 quand un serveur pour des connexions courtes

type IoT de choisir un N à 16. La sonde devra bien sûr s'adapter dynamiquement (et dans les 2 sens) mais au bout de 2 périodes complète, il est facile de différencier N=16, N=32 et N= 64.

C. Expérimentation avec Akamai

En parallèle de ce travail, une expérimentation à taille réelle a été réalisée par Orange Labs en collaboration avec Akamai, un des plus gros CDN du monde. Cette expérimentation a eu pour objectif d'implémenter les bits du côté serveur dans le TTL du protocole IP pour obtenir des résultats à présenter durant le meeting 105 de l'IETF. Sur un réseau réel, les bits QR ont montré leur utilité et ces résultats ont intéressé de nombreux acteurs durant les meetings.

V. Travaux annexes

A. Salon de la recherche

Le 16 juin 2019, j'ai eu l'occasion de présenter les extensions QR, le protocole Quic et mon travail au cours de l'évènement "les Salons de la Recherche de Lannion". Le site de Lannion étant un des plus gros en termes de recherche du groupe, il est organisé chaque année un salon qui permet aux différentes équipes de présenter les dernières recherches dans tous les domaines. 5G, IoT, Virtualisation et donc nous avec Quic. En plus de présenter et vulgariser mon travail, il m'a permis de découvrir la recherche à Orange Labs Lannion et rencontrer d'autres chercheurs.

B. Présentation de l'e-estonia

Grâce à l'INSA, j'ai pu étudier un semestre en 2018 à l'université technologique de Tallinn en Estonie. Durant ce semestre, j'ai pu voir et comprendre par moi-même le modèle de la e-société estonienne dont j'avais tant lu et entendu parler pendant des années.
J'ai eu l'occasion à 2 reprises de pouvoir parler de cette société numérique et de faire découvrir ce pays passionnant aux salariés d'Orange au travers de 2 présentations en amphithéâtre. Cette expérience fut très intéressante pour moi, car elle m'a obligé à vulgariser et expliquer devant un parterre d'une centaine de personnes un sujet qui me tenait à cœur.

Conclusions

Sur les bits SQR

Pour l'instant le groupe IETF Quic semble peu enclin à vouloir utiliser ces bits même si beaucoup de travail de démonstration, de lobbying a été fait, le groupe étant concentré sur la clôture des problèmes restant (encore 132 issues sur Github) avant de publier une version 1 d'ici la fin de cette année. Pour autant il n'est pas du tout impossible comme démontré avec Akamai d'utiliser cette solution en mode « debug » en cas de besoin même si cela inclut un travail de développement et un manque d'universalité. De plus, au vu de la conférence de Montréal et de l'intérêt de certains, les extensions sont toujours d'actualité pour une possible implémentation dans une version 2 du standard Quic.

Sur le chiffrement des protocoles

La mode est au tout chiffrement, au plus grand bénéfice de la confidentialité des données qui circulent sur Internet pour l'utilisateur. Il n'est donc pas trop spéculatif de penser que de plus en plus de protocoles seront conçus « encrypted-by-design ». C'est d'autant plus vrai que le matériel dédié au chiffrement est de plus en plus commun, de moins en moins cher, de plus en plus petit même dans l'IoT. Les extensions proposées pour Quic et expérimentée durant ce stage ne sont absolument pas spécifiques à ce protocole et pourraient très bien être utilisées pour tous les protocoles de l'internet profondément chiffré dans l'optique de toujours proposer la meilleure qualité de réseau possible aux utilisateurs.

Sur le testbed d'expérimentation

Bien sûr, ce travail est loin d'être parfait et mérite donc d'être poursuivie et amélioré.

D'abord, sur la partie des expérimentations, comme nous allons le voir dans la partie "Résultats et Analyses", l'émulation n'est pas parfaite car beaucoup de paramètres sont ultras simplifiés, presque caricaturaux et sont difficile à paramétrer pour émuler la réalité.
Ensuite, il y a le problème de la sous-exploitation des données récupéré des bouts. En effet, actuellement, nous ne récupérons finalement que les pertes réelles avec l'Oracle et les numéros de séquence interne et les numéros de stream fournie par les piles. En réalité, puisque nous pouvons déchiffrer et décoder les trames, il serait possible d'extraire et afficher les Sack ainsi que les Offset et le datasegment. Malheureusement, l'outil développé en parallèle n'a pas encore été intégré à l'analyse finale.
Les trames décodées permettraient aussi d'utiliser les Qlogs, une solution proposée par l'université d'Hasselt [23] et les intégrer à l'interface d'analyse. Cet outil est aujourd'hui de plus en plus utilisé dans la communauté Quic IETF et devrait trouver sa place dans notre outil pour appuyer la proposition des bits QR.
Enfin, il serait intéressant de s'appuyer sur des outils de machine learning pour aider à l'analyse des résultats et opérer un troubleshooting « automatique ».

Sur le travail réalisé

Au cours de ce stage, j'ai mis à jour la base de connaissance sur Quic, développé un testbed d'expérimentation, un site web pour la configuration et l'analyse de capture, l'analyse des données obtenues et confirmé le mécanisme des bits QR en apportant des arguments. Ce travail peut donc être considéré comme un travail de recherche avec une problématique, un état de l'art, des hypothèses, un système d'expérimentation, des analyses, une collaboration avec d'autres acteurs et enfin une conclusion.

Personnelle

Ce stage de fin d'études chez Orange Labs m'a permis de mettre en application de nombreuses compétences acquise au sein de l'INSA centre val de Loire dans la filière STI comme le développement informatique, le réseau, les mathématiques de l'informatique, tout en me permettant de mener un travail de recherche. Ce stage m'a aussi permis de découvrir un domaine, celui des télécommunications et des réseaux opérationnels. Ce stage m'a également permis de conforter mon choix de poursuivre avec une thèse de doctorat dans ce domaine. La thèse me permettant d'être à terme chercheur en informatique.

Tables des illustrations

Références

[1] Pascal Anelli, *Contrôle de congestion*, IREMIA
[2] Rezaei et al, *ICON: Incast Congestion Control using Packet Pacing in Datacenter Networks*, University of Illinois
[3] https://www.chromium.org/quic, site de presentation de Google Quic
[4] https://quic.netray.io/stats.html, site de presentation des résultat de netray
[5] https://groups.google.com/a/chromium.org/forum/#!forum/proto-quic, groupe de discussion gQuic
[6] https://mailarchive.ietf.org/arch/browse/quic/#, groupe de discussion de l'IETF WG Quic
[7] https://github.com/quicwg/base-drafts/wiki/, le github du groupe IETF WG Quic
[8] https://datatracker.ietf.org/wg/quic/documents/, la liste des draft de l'IETF WG Quic
[9] Catherine Pearce, *HTTP/2 & Quic Teaching good protocols to do bad things*, Blackat USA 2016
[10] https://calendar.perfplanet.com/2018/quic-and-http-3-too-big-to-fail/, "quic and http3 too big to fail", Robin Marx
[11] https://trammell.ch/post/2018-03-29-and-yet-it-spins/, "And yet, it spins", Brian Trammell
[12] Piet de Vaere, *Adding Passive Measurability to QUIC*, ETH Zurich, 2018
[13] B. Trammell et M. Kühlewind, *Revisiting the Privacy Implications of Two-Way Internet Latency Data*, Springer, 2018
[14] Piraux et al., *Observing the Evolution of QUIC Implementations*, UC Louvain, 2018
[15] http://mininet.org/, site official de l'outil mininet
[16] http://www.voiptroubleshooter.com/indepth/burstloss.html, site de VoIP troubleshooter
[17] Hasslinger et Hohlfeld, *The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet*, Université de Darmstadt, 2008
[18] Nussbaum et Richard, *A Comparative Study of Network Link Emulators,* CNS'09 San-Diego, 2009
[19] Keller, *Manual TC Packet Filtering and Netem*, ETH Zurich, 2006
[20] https://github.com/facebookarchive/augmented-traffic-control/tree/master/utils/profiles, Exemples de configuration issu de Augmented Traffic Control de Facebook
[21] https://blog.apnic.net/2018/01/29/measuring-quic-vs-tcp-mobile-desktop/, Quic vs TCP, Arash Molavi Kakhki, 2018
[22] https://httparchive.org/reports/state-of-the-web, State of the Web, HTTP Archive, 2019
[23] https://quic.edm.uhasselt.be/, Towards Quic Debuggability, Hasselt University, 2018
[24] https://www.rfc-editor.org/search/rfc_search.php, outils de recherche de RFC, IETF
[25] Maxime Piraux, *A test suite for Quic*, Université catholique de Louvain, 2018
[26] Sy et al., *A Quic look at Web tracking,* Proceedings on Privacy Enhancing Technologies, 2019
[27] *A comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas*
[28] Daniel Moss, *Modeling DSL with NetEm*
[29] Salsano et al., *Definition of a general and intuitive loss model for packet networks and its implementation in the Netem module in the Linux kernel,* University of Roma, 2010
[30] Jero et al., *How sercure and Quick is QUIC ?* , Purdue University, 2016
[31] Jurgelionis et al., *An empirical Study of NetEm Network emulation functionalities,* Norwegian University of Science and Technology, 2011
[32] *The Quality of Internet Service: AT&T's global IP network performance measurements*, AT&T, 2013

Annexes

1. L'évolution du premier octet du paquet short header Quic

0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
gQuic							
Reserved	Multipath	Packet number length		CID Length		Reset	Version
Draft-00							
Flags							
Draft-02							
0	C	K	Type				
Draft-10							
0	C	K	1	0	Type		
Draft-11							
0	K	1	1	0	R	Type	
Draft-12							
0	K	1	1	0	Reserved		
Draft-17							
0	1	S	Reserved		K	Packet number length	
Draft-17-SQR							
0	1	S	Q	R	K	Packet number length	

2. Frame type

(Extrait du draft Quic transport)

Type Value	Frame Type Name	Definition
0x00	PADDING	Section 19.1
0x01	PING	Section 19.2
0x02 - 0x03	ACK	Section 19.3
0x04	RESET_STREAM	Section 19.4
0x05	STOP_SENDING	Section 19.5
0x06	CRYPTO	Section 19.6
0x07	NEW_TOKEN	Section 19.7
0x08 - 0x0f	STREAM	Section 19.8
0x10	MAX_DATA	Section 19.9
0x11	MAX_STREAM_DATA	Section 19.10
0x12 - 0x13	MAX_STREAMS	Section 19.11
0x14	DATA_BLOCKED	Section 19.12
0x15	STREAM_DATA_BLOCKED	Section 19.13
0x16 - 0x17	STREAMS_BLOCKED	Section 19.14
0x18	NEW_CONNECTION_ID	Section 19.15
0x19	RETIRE_CONNECTION_ID	Section 19.16
0x1a	PATH_CHALLENGE	Section 19.17
0x1b	PATH_RESPONSE	Section 19.18
0x1c - 0x1d	CONNECTION_CLOSE	Section 19.19

3. Liste des implémentations

Nom	Auteur	Dernière Version	Langage de programmation	ID	License
Aioquic	Jeremy Lainé, Spacinov	Draft-22	Python	0xff000016	BSD-3
AppleQUIC	Apple	Draft-20	Objective-C	0xff000014	Closed Source

F5	F5	Draft-21	C	0xff000016	Closed source
kwik	Free Software Foundation	Draft-20	Java	0xff000013	GNU GPL
Lsquic	LiteSpeed	draft-20 /gQuic046	C	0xff000014	Closed source
minq	Eric Rescorla Firefox	Draft-11	Go	0xff00000b	MIT
Mozquic	Mozilla	Draft-06	C++	0xf123f0c[0-f]	Mozilla Public License 2.0
Mvfst	Facebook	Draft-22	C++	0xfaceb00[0-f	Depuis Mai, MIT
Ngtcp2	Tatsuhiro Tsujikawa	Draft-22	C	0xff000016	MIT
Pandora	Université d'Aalto (Fi)	Draft-20	C	0xff000014	Closed source
Picoquic	C. Huitema, Octopus	Draft-22	C	0x5043513[0-2]	MIT
Quant	Netapp	Draft-22	C11	0x454747[00-ff]	BSD-2
Quic-go	Lucas Clémente	draft-22 /gQuic046	Go	0x51474f[0-f]	MIT
quiche	Cloudflare	Draft-20	Rust	0xff000014	BSD-2
Quicker	Université d'Hasselt (Be)	Draft-20	TypeScript	0xff000014	ISC
Quicly	Kazuo Oku	Draft-18	C	0x91c170[0-f]	MIT
Quinn	Ochtman & Saunders	Draft-20	Rust	0xff000014	MIT
Sora_quic		Draft-19	Erlang	0xff000013	Closed source
Winquic	Microsoft	Draft-22	C	0xabcd000[0-f]	Closed source
Chromium /cronet	Google	gQuic046	C++	0x51303[0-4]3[0-9]	Chromium

4. Fonctions liées aux bits

```
function initSpinBit(connexion):
   if random(0,7) == 0:  //Une connexion sur 8 n'a pas les bits d'activés
      connexion.hasSQRBit = False
   else:
      connexion.hasSQRBit = True
      connexion.currentSpinBit = 0
      connexion.currentSquareBit = 0
      connexion.currentRetransmitBit = 0
      connexion.countSquare = 0
      connexion.countRetransmit = 0

function getSpinBit(connexion) -> byte:
   if not connexion.hasSQRBit:
      connexion.currentSpinBit = random(0,1)
   return connexion.currentSpinBit << 5

function setSpinBit(connexion, received_spinbit):
   if connexion.client:
      connexion.currentSpinBit = not received_spinbit
   else:
      connexion.currentSpinBit = received_spinbit

function getSquareBit(connexion) -> byte:
   if connexion.hasSQRBit:
      if connexion.countSquare == SQUARE_PERIOD:
         connexion.countSquare = 0
```

```
            connexion.currentSquareBit = not connexion.currentSquareBit
        else:
            connexion.countSquare = connexion.countSquare + 1
    else:
        connexion.currentSquareBit = random(0,1)
    return connexion.currentSquareBit << 4

function getRetransmitBit(connexion) -> byte:
    if connexion.hasSQRBit:
        connexion.countRetransmit = connexion.countRetransmit - 1
    else:
        connexion.currentRetransmitBit = random(0,1)
    return connexion.currentRetransmitBit << 3

function incrRetransmitBit():
    connexion.countRetransmit = connexion.countRetransmit + 1
```

5. testbed.conf

```
Li
Jitter_everywhere_1_32
2019-08-04 08:55:05.921760
picoquic
m
1
c32
eth2
200mbit
4
eth3
delay 5ms 1ms distribution normal
eth2
delay 5ms 1ms distribution normal
eth1
delay 5ms 1ms distribution normal
eth0
delay 5ms 1ms distribution normal
```

6. man netem

```
NAME
      NetEm - Network Emulator

SYNOPSIS
      tc qdisc ... dev DEVICE ] add netem OPTIONS

      OPTIONS := [ LIMIT ] [ DELAY ] [ LOSS ] [ CORRUPT ] [ DUPLICATION ] [ REORDERING ][
RATE ]

      LIMIT := limit packets

      DELAY := delay TIME [ JITTER [ CORRELATION ]]]
            [ distribution { uniform | normal | pareto |  paretonormal } ]

      LOSS := loss { random PERCENT [ CORRELATION ]  |
                  state p13 [ p31 [ p32 [ p23 [ p14]]]] |
                  gemodel p [ r [ 1-h [ 1-k ]]] }  [ ecn ]

      CORRUPT := corrupt PERCENT [ CORRELATION ]]
```

DUPLICATION := duplicate PERCENT [CORRELATION]]

REORDERING := reorder PERCENT [CORRELATION] [gap DISTANCE]

RATE := rate RATE [PACKETOVERHEAD [CELLSIZE [CELLOVERHEAD]]]]

7. netem kernel function enqueue

```
static int netem_enqueue(struct sk_buff *skb, struct Qdisc *sch,
             struct sk_buff **to_free)
{
    struct netem_sched_data *q = qdisc_priv(sch);
    /* We don't fill cb now as skb_unshare() may invalidate it */
    struct netem_skb_cb *cb;
    struct sk_buff *skb2;
    struct sk_buff *segs = NULL;
    unsigned int len = 0, last_len, prev_len = qdisc_pkt_len(skb);
    int nb = 0;
    int count = 1;
    int rc = NET_XMIT_SUCCESS;
    int rc_drop = NET_XMIT_DROP;

    /* Do not fool qdisc_drop_all() */
    skb->prev = NULL;

    /* Random duplication */
    if (q->duplicate && q->duplicate >= get_crandom(&q->dup_cor))
        ++count;

    /* Drop packet? */
    if (loss_event(q)) {
        if (q->ecn && INET_ECN_set_ce(skb))
            qdisc_qstats_drop(sch); /* mark packet */
        else
            --count;
    }
    if (count == 0) {
        qdisc_qstats_drop(sch);
        __qdisc_drop(skb, to_free);
        return NET_XMIT_SUCCESS | __NET_XMIT_BYPASS;
    }

    /* If a delay is expected, orphan the skb. (orphaning usually takes
     * place at TX completion time, so _before_ the link transit delay)
     */
    if (q->latency || q->jitter || q->rate)
        skb_orphan_partial(skb);

    /*
     * If we need to duplicate packet, then re-insert at top of the
     * qdisc tree, since parent queuer expects that only one
     * skb will be queued.
     */
    if (count > 1 && (skb2 = skb_clone(skb, GFP_ATOMIC)) != NULL) {
        struct Qdisc *rootq = qdisc_root(sch);
        u32 dupsave = q->duplicate; /* prevent duplicating a dup... */

        q->duplicate = 0;
        rootq->enqueue(skb2, rootq, to_free);
        q->duplicate = dupsave;
        rc_drop = NET_XMIT_SUCCESS;
    }

    /*
     * Randomized packet corruption.
     * Make copy if needed since we are modifying
     * If packet is going to be hardware checksummed, then
     * do it now in software before we mangle it.
     */
    if (q->corrupt && q->corrupt >= get_crandom(&q->corrupt_cor)) {
        if (skb_is_gso(skb)) {
            segs = netem_segment(skb, sch, to_free);
            if (!segs)
                return rc_drop;
        } else {
            segs = skb;
        }

        skb = segs;
        segs = segs->next;

        skb = skb_unshare(skb, GFP_ATOMIC);
        if (unlikely(!skb)) {
            qdisc_qstats_drop(sch);
            goto finish_segs;
        }
        if (skb->ip_summed == CHECKSUM_PARTIAL &&
            skb_checksum_help(skb)) {
            qdisc_drop(skb, sch, to_free);
            goto finish_segs;
        }
```

```
        skb->data[prandom_u32() % skb_headlen(skb)] ^=
            1<<(prandom_u32() % 8);
    }

    if (unlikely(sch->q.qlen >= sch->limit)) {
        qdisc_drop_all(skb, sch, to_free);
        return rc_drop;
    }

    qdisc_qstats_backlog_inc(sch, skb);

    cb = netem_skb_cb(skb);
    if (q->gap == 0 ||      /* not doing reordering */
        q->counter < q->gap - 1 ||  /* inside last reordering gap */
        q->reorder < get_crandom(&q->reorder_cor)) {
        u64 now;
        s64 delay;

        delay = tabledist(q->latency, q->jitter,
                  &q->delay_cor, q->delay_dist);

        now = ktime_get_ns();

        if (q->rate) {
            struct netem_skb_cb *last = NULL;

            if (sch->q.tail)
                last = netem_skb_cb(sch->q.tail);
            if (q->t_root.rb_node) {
                struct sk_buff *t_skb;
                struct netem_skb_cb *t_last;

                t_skb = skb_rb_last(&q->t_root);
                t_last = netem_skb_cb(t_skb);
                if (!last ||
                    t_last->time_to_send > last->time_to_send)
                    last = t_last;
            }
            if (q->t_tail) {
                struct netem_skb_cb *t_last =
                    netem_skb_cb(q->t_tail);

                if (!last ||
                    t_last->time_to_send > last->time_to_send)
                    last = t_last;
            }

            if (last) {
                /*
                 * Last packet in queue is reference point (now),
                 * calculate this time bonus and subtract
                 * from delay.
                 */
                delay -= last->time_to_send - now;
                delay = max_t(s64, 0, delay);
                now = last->time_to_send;
            }

            delay += packet_time_ns(qdisc_pkt_len(skb), q);
        }

        cb->time_to_send = now + delay;
        ++q->counter;
        tfifo_enqueue(skb, sch);
    } else {
        /*
         * Do re-ordering by putting one out of N packets at the front
         * of the queue.
         */
        cb->time_to_send = ktime_get_ns();
        q->counter = 0;

        __qdisc_enqueue_head(skb, &sch->q);
        sch->qstats.requeues++;
    }

finish_segs:
    if (segs) {
        while (segs) {
            skb2 = segs->next;
            skb_mark_not_on_list(segs);
            qdisc_skb_cb(segs)->pkt_len = segs->len;
            last_len = segs->len;
            rc = qdisc_enqueue(segs, sch, to_free);
            if (rc != NET_XMIT_SUCCESS) {
                if (net_xmit_drop_count(rc))
                    qdisc_qstats_drop(sch);
            } else {
                nb++;
                len += last_len;
            }
            segs = skb2;
        }
        sch->q.qlen += nb;
        if (nb > 1)
            qdisc_tree_reduce_backlog(sch, 1 - nb, prev_len - len);
    }
    return NET_XMIT_SUCCESS;
}
```

8. scenario.json

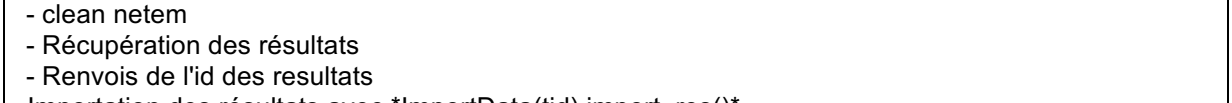

```
{
    "name": "Jitter_everywhere_1_32",
    "names": {
        "server": "lili104",
        "mid": "lili100",
        "client": "lili90"
    },
    "fsize": "m",
    "qshape": "c32",
    "stack": "picoquic",
    "sonde": "SimpleQSonde",
    "rate": "200",
    "pert": {
        "eth0": {
            "name_config": "default",
            "delay": 5,
            "distribution": "normal",
            "delay_jitter": 1
        },
        "eth1": {
            "name_config": "default",
            "delay": 5,
            "distribution": "normal",
            "delay_jitter": 1
        },
        "eth2": {
            "name_config": "default",
            "delay": 5,
            "distribution": "normal",
            "delay_jitter": 1
        },
        "eth3": {
            "name_config": "default",
            "delay": 5,
            "distribution": "normal",
            "delay_jitter": 1
        }
    },
    "mido": true
}
```


9. Readme du gitlab Quic-Experiments

```
# Quic-experiments

***Par Ronteix--Jacquet Flavien pour Orange Labs(c) 2019***

## Installation

python >= 3.6

`make install`

## Fonctionnement

### Infrastructure du code

![Infrastructure du code](static/infra-code.png)

Tout le controlleur doit être sur une machine ayant accès en ssh au testbed via la machine **lili100**

L'execution de l'expérimentation se fait ainsi:

- *init* des variables globales
- création de l'objet *Scenario* avec le fichier scenario en paramètre
- préparation du scenario *Scenario.prepare()*
- envois du testbed.conf sur le testbed avec *Scenario.run()*
- *runner.sh* sur le testbed
  - calcul du nouvel id
  - chargement du testbed.conf avec *config_handler.sh*
  - application de netem
  - capture de l'interface
  - run test
```

    - run server
    - run client pour transférer une certaine quantité de donné
    - stop server
  - clean netem
  - Récupération des résultats
  - Renvois de l'id des resultats
- Importation des résultats avec *ImportData(tid).import_res()*
- création de l'objet *processData()* pour traiter les résultats
- traitement des données pour obtenir le fichier résultat *processData.run()*
- vérification que tous les fichiers résutats sont présents avec *check_results()*

Fonctionnement du testbed d'expérimentation



Les formats de fichier

Scenario

Les scenarios sont les paramètres d'une expérimentation. Ils sont dans le dossier `/testbed` et sont au format json.
Les paramètres sont:

- *name* (required): nom du scenario, souvent le même nom que le fichier
- *names*: nom des machines du testbed et leur rôle (lili104,...)
  - server/mid/client
- *fsize*: la taille des données à tranférer
- *mido*: mid-only, est-ce que l'on recupere aussi les données sur les endpoints ?
- *qshape*: la forme et la période pour le bit square (=c64 pour carré de période 128)
- *rate*: Le débit maximale de la connexion client-serveur (=100mbit)
- *stack*: la pile quic à utiliser (=picoquic)*
- *pert*: Perturbations netem appliquées aux interfaces
  - *eth0*/*eth1*/*eth2*/*eth3* voir plus bas sur les [perturbations netem](#Perturbations-Netem)

Testbed.conf

Le fichier **testbed.conf** contient les paramètre pour l'expérimentation dont les règles de perturbations netem.

Exemple :
```
Li
All_perts
2019-07-16 12:50:29.474652
picoquic
m
1
c64
eth2
500mbit
4
eth3
delay 5ms loss random 4% 0%
eth2

eth1
delay 20ms loss random 1% 25%
eth0

```

L'historique de version:

- *H*: Version initiale
- *He*: Ajout du sQuare Shape
- *Li*: Ajout du rate

Results et informations

Les fichiers résultats **XXX-res.json** et **XXX-info.json** sont produits à l'issu de l'analyse des données, ils se trouvent dans le dossier `/results` et débute par le numéro d'id de l'expérimentation (tout comme les fichiers de capture et de log qui leur sont associés)

XXX-info.json:

- *id*: l'id de l'expérimentation
- *layout*: le format du premier octet short header Quic (peut évoluer dans le temps et en fonction des implémentations)
- *desc*: la description de l'expérimentation (qui sert aussi de nom dans les affichages)
- *date*: La date à laquelle a été faite l'expérimentation (date du premier paquet capturé sur le midpoint)
- *size*: Taille en paquet de la capture
  - up/down

Il est nécessaire de garder ce fichier le plus petit possible car il est lu à chaque fois que l'on charge la liste des expérimentations, donc si chaque json fait plusieurs kilo, la lecture en sera d'autant plus ralentie.

XXX-res.json:

Contient tous les résultats extrait des captures/logs du testbed et généré avec le run() du processData :

- *transmitted* (u/d): Liste de paquet
  - (date, données cumulées, spinbit, squarebit, retransmitbit)
- *rtt* (u/d): Liste des demi roundtime-trip
  - (date, rtt en ms)
- *bif* (u/d): Liste des Bytes-in-flight
  - (date, \#bytes)
- *square* (u/d): Liste des pertes comptées par la sonde à l'aide du bit Q
  - (date, \#Pertes)
- *retransmit* (u/d): Liste des pertes comptées par la sonde à l'aide du bit R
  - (date, \#Pertes)

Graphs

Les **XXX-graph.json** sont générés à la suite d'un recompute sur l'interface d'analyse, c'est une forme de cache au format json des graphs plotly affichés. Ne pas modifier à la main.

Penser à refaire un recompute à chaque modifications des graphs.

Utilisation

Configuration dans **config.py**, les paramètres importants:

- *DEBUG*: N'execute pas certaines commande en mode Debug
- *GID*: Le dernier ID pour les resultats, modifiés au cours de la vie d'un programme
- *SQUARE_PERIOD*: Période pour le bit Square
- *LINUX*: True si Linux et donc si l'on peut executer des commandes Bash
- *OLD_FORMAT*: Utilisation des résultats d'ancien format (va les convertir)

- *VERSION*: Version ecrite dans le testbed.conf pour assurer un suivi des compatibilité (Actuellement He)
- *TEST_ID*: L'id reservé au test (=333)
- *UP_TAG*/*DOWN_TAG*: 'up'/'down'
- *BASE_FOLDER*: Path vers le dossier de l'expérimentation
- *CONFIG_FILE*: Path vers le fichier de configuration pour config.py, pas encore entièrement fonctionnel
- *RESULTS_FOLDER*: Path vers le dossier des résultats
- *TESTBED_FOLDER*: Path vers les scenarios et config de testbed
- *LOG_FILE*: Fichier de logs
- *STACKS*: Liste des stacks prise en charge
- *FSIZES*: Taille de fichier, utilisé sur le testbed
- *CCS*: Liste des algorithmes de controle de congestion prise en charge
- *MIDO*: Mid only, Recupération des resultats uniquement sur la sonde centrale
- *PORT_SERV*: Port utilisé par le serveur Quic
- *IP_MIDDLE*, *IP_CLIENT*, *IP_SERV*: Les IP sur le test
- *NAME_MIDDLE*, *NAME_CLIENT*, *NAME_SERV*: Les noms des machines du testbed
- *HAS_MIDDLE*: Si le midpoint fait du forward et que son adresse IP apparaît
- *SONDE*: Sonde utilisée (=SimpleQSonde)
- *LISTE_SONDE*: Liste des sondes utilisables
- *LOG_TAG*: Mot clef utilisé dans les logs des piles pour générer les chiffres pour le seqquic

Utilisation de l'**application web** : `./app.py`

quic_experiments.py

Exemple d'experimentation avec un scenario : `./quic_experiments -i testbed/scenario_vierge.json`

Exemple de l'utilisation d'une sonde : `./quic_experiments -s qpost2 -i testbed/Jitter_to_reorder.json`

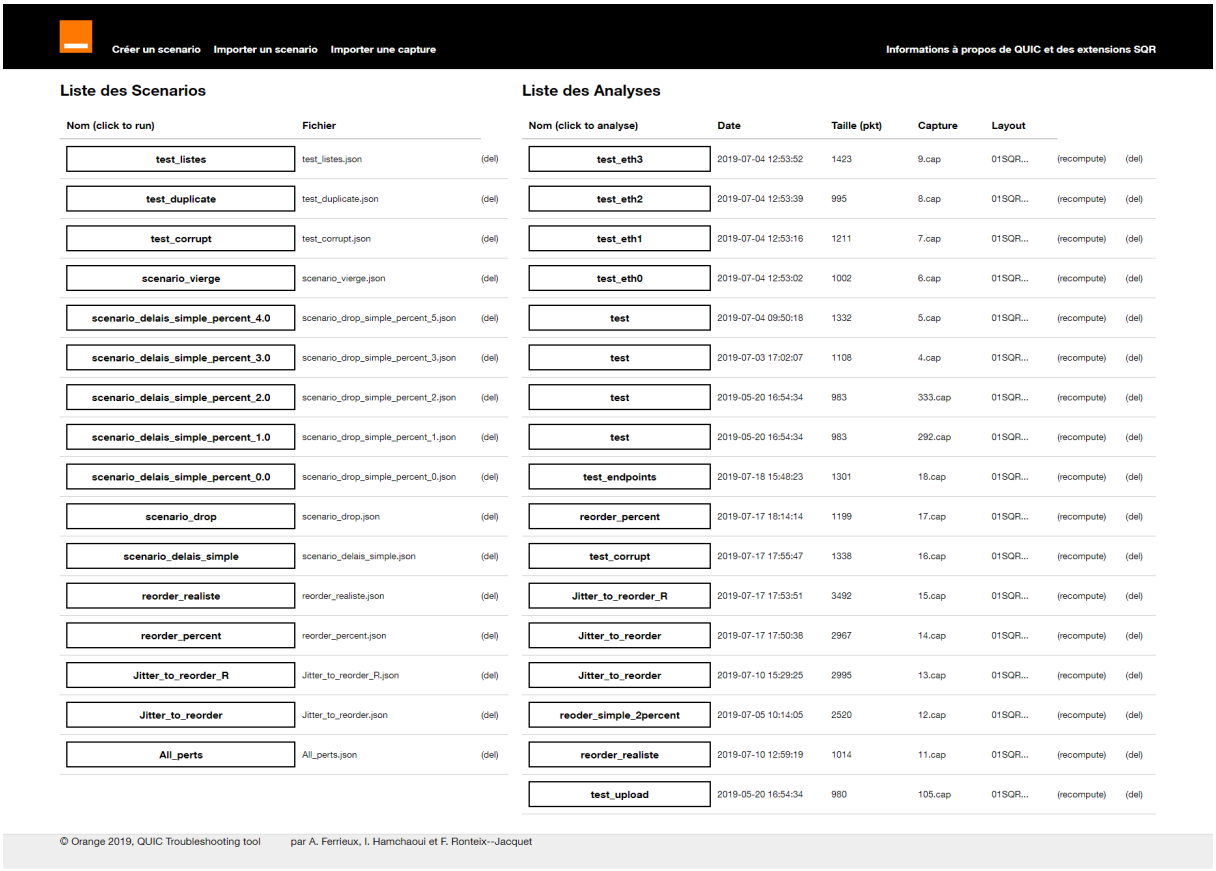
Execution des tests : `./quic_experiments -t {prepare_data, scenario, import_data} -i testbed/scenario_vierge.json`

Extensions

Certaines partie du testbed ont été pensé explicitement pour être extensible. c'est le cas de la config du testbed, les sondes, les stacks utilisées ou encore la forme du signal Q.

10. Captures du site web

/Home

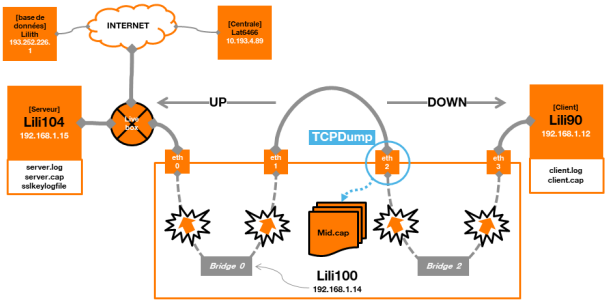

Créer un scenario
Importer un scenario
Importer une capture
Informations à propos de QUIC et des extensions SQR
Liste des Scenarios
Nom (click to run)
Fichier
test_listes
test_listes.json
(del)
test_duplicate
test_duplicate.json
(del)
test_corrupt
test_corrupt.json
(del)
scenario_vierge
scenario_vierge.json
(del)
scenario_delais_simple_percent_4.0
scenario_drop_simple_percent_5.json
(del)
scenario_delais_simple_percent_3.0
scenario_drop_simple_percent_3.json
(del)
scenario_delais_simple_percent_2.0
scenario_drop_simple_percent_2.json
(del)
scenario_delais_simple_percent_1.0
scenario_drop_simple_percent_1.json
(del)
scenario_delais_simple_percent_0.0
scenario_drop_simple_percent_0.json
(del)
scenario_drop
scenario_drop.json
(del)
scenario_delais_simple
scenario_delais_simple.json
(del)
reorder_realiste
reorder_realiste.json
(del)
reorder_percent
reorder_percent.json
(del)
Jitter_to_reorder_R
Jitter_to_reorder_R.json
(del)
Jitter_to_reorder
Jitter_to_reorder.json
(del)
All_perts
All_perts.json
(del)
Liste des Analyses
Nom (click to analyse)
Date
Taille (pkt)
Capture
Layout
test_eth3
2019-07-04 12:53:52
1423
9.cap
01SQR...
(recompute)
(del)
test_eth2
2019-07-04 12:53:39
995
8.cap
01SQR...
(recompute)
(del)
test_eth1
2019-07-04 12:53:16
1211
7.cap
01SQR...
(recompute)
(del)
test_eth0
2019-07-04 12:53:02
1002
6.cap
01SQR...
(recompute)
(del)
test
2019-07-04 09:50:18
1332
5.cap
01SQR...
(recompute)
(del)
test
2019-07-03 17:02:07
1108
4.cap
01SQR...
(recompute)
(del)
test
2019-05-20 16:54:34
983
333.cap
01SQR...
(recompute)
(del)
test
2019-05-20 16:54:34
983
292.cap
01SQR...
(recompute)
(del)
test_endpoints
2019-07-18 15:48:23
1301
18.cap
01SQR...
(recompute)
(del)
reorder_percent
2019-07-17 18:14:14
1199
17.cap
01SQR...
(recompute)
(del)
test_corrupt
2019-07-17 17:55:47
1338
16.cap
01SQR...
(recompute)
(del)
Jitter_to_reorder_R
2019-07-17 17:53:51
3492
15.cap
01SQR...
(recompute)
(del)
Jitter_to_reorder
2019-07-17 17:50:38
2967
14.cap
01SQR...
(recompute)
(del)
Jitter_to_reorder
2019-07-10 15:29:25
2995
13.cap
01SQR...
(recompute)
(del)
reoder_simple_2percent
2019-07-05 10:14:05
2520
12.cap
01SQR...
(recompute)
(del)
reorder_realiste
2019-07-10 12:59:19
1014
11.cap
01SQR...
(recompute)
(del)
test_upload
2019-05-20 16:54:34
980
105.cap
01SQR...
(recompute)
(del)
© Orange 2019, QUIC Troubleshooting tool
par A. Ferrieux, I. Hamchaoui et F. Ronteix--Jacquet

/Create

Liste **QUIC Troubleshooting scenario maker**

Description (obligatoire) — Créer >

Pile réseau: picoquic | Sonde d'analyse: SimpleQSonde | Taille fichier: 1MB | Periode sQuare: 64 | Mid-only: Oui



3 Interne Orange

Eth0 *(Sur R up)*

Nom de la perte: no_pert | Rate (mbit): 200

Délai: Délai (ms): 5 | Distribution: normal | Jitter (ms) | Correlation (%)

Pertes: Type: none | Type: 0 (% de pertes) 0(% de correlation

Reordonnancement: Taux (%) | Delais (ms)

Corruption: Taux (%) | Delais (%)

Duplication: Taux (%) | Delais (%)

Eth1 *(Sur Q down)*

Nom de la perte: no_pert | Rate (mbit): 200

Délai: Délai (ms): 5 | Distribution: normal | Jitter (ms) | Correlation (%)

Pertes: Type: none | Type: 0 (% de pertes) 0(% de correlation

Reordonnancement: Taux (%) | Delais (ms)

Corruption: Taux (%) | Delais (%)

Duplication: Taux (%) | Delais (%)

Eth2 *(Sur Q up)*

Nom de la perte: no_pert | Rate (mbit): 200

Délai: Délai (ms): 5 | Distribution: normal | Jitter (ms) | Correlation (%)

Pertes: Type: none | Type: 0 (% de pertes) 0(% de correlation

Reordonnancement: Taux (%) | Delais (ms)

Corruption: Taux (%) | Delais (%)

Duplication: Taux (%) | Delais (%)

Eth3 *(Sur R down)*

Nom de la perte: no_pert | Rate (mbit): 200

Délai: Délai (ms): 5 | Distribution: normal | Jitter (ms) | Correlation (%)

Pertes: Type: none | Type: 0 (% de pertes) 0(% de correlation

Reordonnancement: Taux (%) | Delais (ms)

Corruption: Taux (%) | Delais (%)

Duplication: Taux (%) | Delais (%)

/Upload_scenario

Liste **QUIC Troubleshooting testbed scenario uploader**

Scenario (format json) * — Choisir un fichier | Aucun fichier choisi — Téléverser >

/Upload_capture

Liste **QUIC Troubleshooting capture uploader**

Nom * — Téléverser >

Fichier de capture * Choisir un fichier | Aucun fichier choisi

layout: 01SQR... | Sonde: SimpleQSonde | IP client: 192.168.1.12 | IP serveur: 192.168.1.15

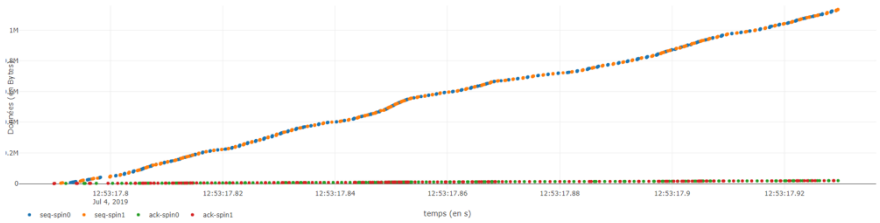
/Analyse

Liste **QUIC Troubleshooting testbed > test_eth1 (7.cap | 7-res.json | 7-graph.json)**

2019-07-04 12:53:16.784789 | 1206 paquets (up:300 / down:906) | 0.99 Mbps | 01SQR... | (recompute)

Colored snake and SeqQUIC

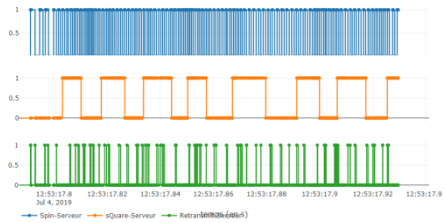
Cliquez sur 2 points pour mesurer



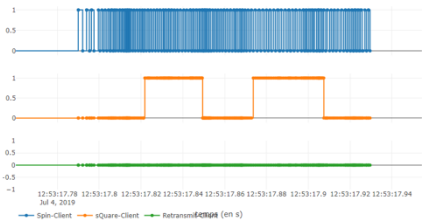
Down stream

Up stream

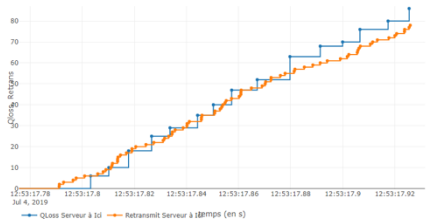
SQR Down



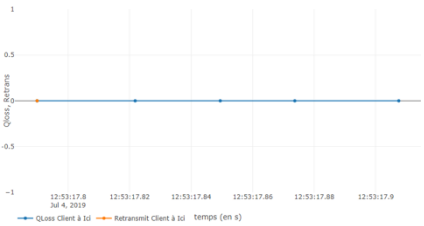
SQR Up



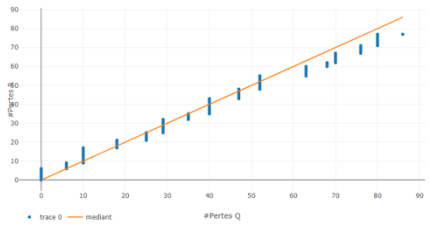
Pertes Down



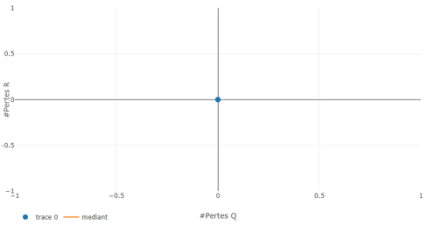
Pertes Up



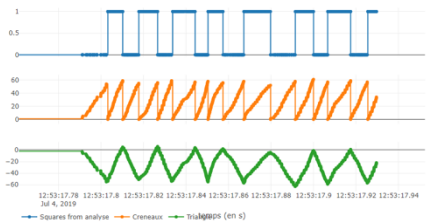
R/Q Down



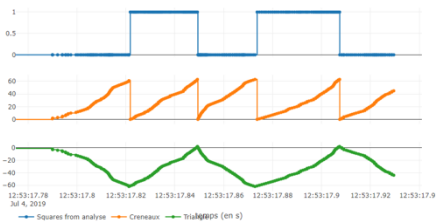
R/Q Up



Etude du Square Down

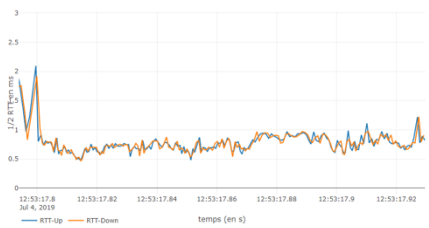


Etude du Square Up



Autres données

RTT



Bytes in Flight