Irsa AHMAD
Tristan BRINGUIER
BUT3 R&T FAP ROM

SAE503

Mise en place et analyse d’un système de streaming multimédia

IUT de Villetaneuse – IUT de Créteil-Vitry – 2025-2026

Sommaire

Glossaire

AAC (Advanced Audio Coding) Norme de codage audio avec compression (codec) utilisée dans ce projet. Elle offre un meilleur rapport qualité/débit que le MP3 et est le standard requis pour accompagner le flux vidéo H.264 dans une transmission RTMP et HLS.

ABR (Adaptive Bitrate Streaming) Technique de diffusion qui consiste à encoder la vidéo en plusieurs qualités (débits) différentes. Le lecteur client choisit automatiquement la meilleure qualité possible en fonction de la bande passante disponible pour éviter les coupures (non implémenté ici, mais cité comme amélioration).

AMF (Action Message Format) Format de sérialisation binaire utilisé par le protocole RTMP pour échanger des commandes entre le client (OBS) et le serveur (OwnCast). Il permet d'envoyer des instructions comme connect, publish ou des métadonnées sur le flux.

Buffering (Mise en mémoire tampon) Action de précharger une certaine quantité de données vidéo dans la mémoire du lecteur client avant de commencer la lecture. Cela permet de compenser les irrégularités du réseau (gigue) et d'éviter les coupures d'image.

Burst Pic soudain de trafic réseau. Dans le contexte du HLS, le trafic n'est pas linéaire mais constitué de "bursts" périodiques correspondant au téléchargement rapide de chaque segment .ts par le client via TCP.

CBR (Constant Bitrate) Mode d'encodage où le débit de données est maintenu constant, quelle que soit la complexité de la scène visuelle. Utilisé dans ce projet (OBS) pour générer une charge réseau prévisible et faciliter l'analyse des goulots d'étranglement.

Chunk (RTMP) Fragment de message. Le protocole RTMP découpe les données (audio, vidéo, commandes) en petits morceaux (chunks) de taille fixe (ex: 600 ou 4096 octets) pour les multiplexer sur une seule connexion TCP.

Codec (Coder-Decoder) Dispositif matériel ou logiciel capable d'encoder (compresser) et de décoder (décompresser) un flux de données numériques.

- **H.264 (AVC) :** Codec vidéo utilisé pour sa grande compatibilité.
- **AAC :** Codec audio utilisé pour la synchronisation.

Conteneur (Docker) Environnement d'exécution léger qui contient une application (ici OwnCast) et toutes ses dépendances. Contrairement à une machine virtuelle, il partage le noyau du système hôte (Linux), ce qui le rend très économe en ressources.

Docker Compose Outil permettant de définir et de gérer des applications Docker multi-conteneurs via un fichier de configuration YAML. Il assure l'orchestration du déploiement d'OwnCast dans ce projet.

Gigue (Jitter) Variation de la latence (délai) entre les paquets arrivant à destination. Une gigue élevée perturbe la fluidité du streaming et nécessite un buffering plus important côté client.

GNS3 (Graphical Network Simulator-3) Logiciel d'émulation de réseaux permettant de combiner des équipements virtuels (routeurs Cisco, switchs) et réels (VMs Debian, Docker). Il est utilisé ici pour simuler la topologie réseau et introduire des perturbations (pertes, latence).

GOP (Group of Pictures) Structure d'un flux vidéo compressé, définissant l'ordre des images (Intra, Prédictive).

- **I-Frame (Keyframe) :** Image complète, point de référence indépendant.
- **Intervalle d'images clés :** Fixé à 2 secondes dans le projet pour permettre à OwnCast de découper les segments HLS proprement.

Handshake (Poignée de main) Processus initial d'établissement d'une connexion.

- **Handshake RTMP :** Échange de paquets (C0-C2 / S0-S2) pour valider la version du protocole et synchroniser le client et le serveur.
- **Handshake TCP :** Processus SYN, SYN-ACK, ACK pour établir la connexion fiable.

Headless Se dit d'un système informatique ou d'un logiciel fonctionnant sans moniteur, clavier ni souris ("sans tête"). Les serveurs Debian utilisés dans ce projet sont en mode headless pour économiser les ressources de simulation.

HLS (HTTP Live Streaming) Protocole de diffusion adaptatif basé sur HTTP développé par Apple. Il découpe le flux vidéo en petits fichiers séquenciels (.ts) référencés dans un fichier manifeste (.m3u8). Il utilise TCP (fiabilité) au lieu d'UDP.

Ingestion Première étape du streaming où le flux vidéo est capturé, encodé et envoyé vers le serveur de diffusion. Ici, l'ingestion se fait via le protocole RTMP entre OBS et OwnCast.

Latence Délai temporel entre la capture d'une image à la source et son affichage sur l'écran du spectateur.

- **Latence réseau :** Temps de transit d'un paquet (RTT).
- **Latence de diffusion :** Augmentée par le buffering et la segmentation HLS (souvent > 10s sans optimisation).

Manifeste (.m3u8) Fichier texte utilisé par le protocole HLS. Il agit comme une liste de lecture dynamique qui indique au lecteur vidéo l'URL des segments vidéo (.ts) disponibles à télécharger.

MPEG-TS (Transport Stream - .ts) Format conteneur utilisé pour encapsuler les données audio, vidéo et de synchronisation dans le protocole HLS. Chaque segment HLS est un petit fichier MPEG-TS.

NAL Unit (Network Abstraction Layer) Unité élémentaire de données dans le codec H.264. Elle contient soit des données vidéo compressées, soit des métadonnées (SPS, PPS) nécessaires au décodage.

NetEm (Network Emulation) Fonctionnalité du noyau Linux utilisée par GNS3 pour simuler des propriétés de réseau étendu (WAN) telles que le délai, la perte de paquets, la corruption ou la duplication.

OBS Studio (Open Broadcaster Software) Logiciel libre et open-source d'enregistrement vidéo et de streaming en direct. Il sert ici de client d'ingestion (source) capable d'encoder en H.264/AAC et de transmettre en RTMP.

OwnCast Serveur de streaming vidéo open-source et auto-hébergé. Il reçoit un flux RTMP et effectue le *transmuxing* vers HLS pour une diffusion web compatible avec les navigateurs modernes.

QoE (Quality of Experience) Mesure subjective de la satisfaction de l'utilisateur final (fluidité de la vidéo, netteté, synchronisation audio/vidéo).

QoS (Quality of Service) Mesure objective des performances du réseau (débit, latence, taux de perte de paquets). Une mauvaise QoS entraîne une mauvaise QoE.

RTMP (Real-Time Messaging Protocol) Protocole propriétaire (initialement Macromedia/Adobe) basé sur TCP, conçu pour la transmission de données audio/vidéo à faible latence. Utilisé ici exclusivement pour l'envoi du flux d'OBS vers le serveur.

Segments Fichiers vidéo de courte durée (quelques secondes) générés par le serveur HLS. Le lecteur télécharge ces fichiers les uns après les autres pour reconstituer le flux continu.

TCP Window Scaling Option du protocole TCP permettant d'augmenter la taille de la fenêtre de réception au-delà de 65 535 octets. Crucial pour maintenir un haut débit sur des réseaux à latence élevée (LFN).

Transmuxing Processus de changement de format conteneur (ex: de RTMP vers HLS) sans ré-encoder le flux audio et vidéo compressés. Cela demande beaucoup moins de CPU que le transcodage complet.

uBridge Outil utilisé par GNS3 pour connecter les interfaces réseau des machines virtuelles (VM) aux hubs et switchs virtuels, permettant l'analyse du trafic via Wireshark.

Wireshark Analyseur de paquets réseau libre. Il est utilisé dans ce projet pour capturer le trafic ("sniffer"), disséquer les protocoles (RTMP, TCP, HTTP) et diagnostiquer les problèmes de transmission.

Liste des figures

Fiche de Cahier des Charges

1. CONTEXTE ET OBJECTIFS DU PROJET

Dans un contexte où le trafic vidéo représente la majorité de la bande passante mondiale, l'objectif est de maîtriser la chaîne complète de transmission multimédia. Le projet consiste à concevoir, déployer et auditer une architecture de streaming en direct (Live Streaming) entièrement virtualisée. Au-delà de la mise en service, le but est d'éprouver la robustesse des protocoles (RTMP, HLS) face à des contraintes réseaux réalistes (latence, gigue, perte de paquets).

2. PÉRIMÈTRE FONCTIONNEL

Le système doit assurer les fonctions suivantes :

- **Acquisition et Encodage :** Capturer une source vidéo de référence, l'encoder en temps réel et l'envoyer vers un serveur distant.
- **Transmission Réseau :** Transiter via une infrastructure routée/commutée simulée, permettant l'injection de perturbations.
- **Diffusion (Serveur) :** Recevoir le flux source, le traiter (transmuxing) et le mettre à disposition via une interface Web.
- **Lecture (Client) :** Permettre la lecture fluide du flux sur un navigateur standard ou un lecteur multimédia.

3. SPÉCIFICATIONS TECHNIQUES

L'architecture doit respecter les choix technologiques et configurations suivants :

A. Infrastructure et Système

- **Virtualisation :** Utilisation exclusive de **GNS3** pour l'émulation réseau (Switch/Routeurs) et QEMU pour les machines hôtes.
- **Système d'Exploitation : Debian GNU/Linux 13 (Trixie)** configuré en mode *Headless* (sans interface graphique) pour optimiser les ressources.
- **Déploiement :** Conteneurisation du serveur de diffusion via **Docker** et orchestration par **Docker Compose**.

B. Chaîne de Traitement Média

- **Source (Encodeur) :**
  - Logiciel : **OBS Studio**.
  - Protocole de sortie : **RTMP** (Port 1935).
  - Encodage Vidéo : H.264 (x264), mode **CBR** (Constant Bitrate) à 2500 Kbps.
  - Encodage Audio : AAC à 160 Kbps.

- Structure GOP : Intervalle d'images clés (**Keyframe Interval**) fixé strictement à **2 secondes**.

- **Serveur de Diffusion :**
  - Solution : **OwnCast** (Open-source, Self-hosted).
  - Rôle : Ingestion RTMP et conversion en **HLS** (HTTP Live Streaming).
  - Sortie : Segments .ts et manifeste .m3u8 servis via HTTP (Port 8080).
- **Client :** Navigateur Firefox ou VLC Media Player.

4. SCÉNARIOS DE TEST ET QUALITÉ DE SERVICE (QoS)

Le projet doit inclure une phase expérimentale utilisant **Wireshark** et les outils de limitation de bande passante de GNS3 pour valider :

1. **Analyse Protocolaire :** Dissection du handshake RTMP et de la segmentation HLS/TCP.
2. **Résilience Réseau :** Simulation de dégradations pour observer l'impact sur la fluidité (QoE)

Chapitre 1 : Introduction

Le streaming multimédia est devenu aujourd'hui un élément incontournable de notre quotidien numérique. Qu'il s'agisse de regarder du contenu à la demande, de suivre des événements en direct ou d'utiliser la visioconférence, cette technologie représente désormais la majorité du trafic circulant sur les réseaux mondiaux. Cette explosion des usages impose des défis techniques importants aux administrateurs réseau, car la vidéo est un flux particulièrement gourmand en ressources et très sensible aux variations de la qualité de la connexion. Comprendre les mécanismes qui permettent de transporter ces données de manière fluide est donc essentiel pour n'importe quel technicien ou ingénieur en télécommunications.

Dans le cadre de cette SAÉ, notre objectif principal était de mettre en place et de comprendre la chaîne de diffusion multimédia. Nous avons cherché à reproduire un environnement de production complet, allant de la source de captation vidéo jusqu'au lecteur final utilisé par le spectateur. Au-delà de la simple mise en route des services, ce projet visait surtout à tester les limites du système. En utilisant des outils de simulation, nous avons pu confronter notre architecture à des contraintes réseaux réalistes, comme la perte de paquets ou la latence, afin d'observer précisément comment les protocoles réagissent et comment la qualité visuelle se dégrade pour l'utilisateur.

Pour mener à bien cette étude, nous avons d'abord mis en place une infrastructure virtualisée complexe s'appuyant sur GNS3 et la technologie des conteneurs Docker. Cette première étape technique nous a permis de construire un réseau sur mesure où chaque composant est isolé et configurable. Nous avons ensuite configuré les serveurs pour gérer l'encodage et la distribution des flux, ce qui nous a donné l'opportunité d'analyser en détail le rôle de protocoles spécifiques comme le RTMP pour l'envoi du flux et le HLS pour sa diffusion à large échelle.

Enfin, une partie importante de notre travail a consisté à réaliser des campagnes de tests approfondies. En manipulant les caractéristiques des liaisons réseau, nous avons pu identifier les seuils de rupture du service et comprendre l'importance de la configuration des serveurs pour maintenir une diffusion stable. Ce rapport présente donc l'ensemble de cette démarche, en commençant par la présentation de la maquette technique, suivie de l'analyse des protocoles de communication, pour finir sur les résultats de nos simulations de pannes et de ralentissements réseau.

Chapitre 2 : Présentation étendue du cahier des charges

2.1 Analyse du besoin et contexte applicatif

Le contexte de cette SAÉ s'inscrit dans une volonté de maîtriser les chaînes de transmission multimédia modernes. Aujourd'hui, la consommation de flux vidéo sur Internet représente la majorité du trafic mondial de données. Comprendre les mécanismes sous-jacents, depuis l'encodage à la source jusqu'au rendu final chez le client, est devenu une compétence incontournable pour un administrateur réseaux et télécoms.

L'objectif principal assigné à ce projet est double : d'une part, déployer une architecture fonctionnelle de streaming en direct, et d'autre part, éprouver cette architecture face à des conditions de réseau dégradées. Contrairement à une simple installation logicielle, notre démarche vise à ouvrir la "boîte noire" du streaming pour analyser l'impact concret de la latence, de la gigue et de la perte de paquets sur l'expérience utilisateur.

Le besoin fonctionnel central est la mise en place d'un système capable de capter un flux vidéo, de l'encoder en temps réel, de le transporter à travers un réseau simulé, et de le distribuer à des clients web ou logiciels. Ce système doit être suffisamment flexible pour nous permettre d'injecter des perturbations contrôlées afin d'analyser les protocoles de transport.

2.2 Définition des spécifications techniques

Pour répondre à ces besoins, nous avons établi un cahier des charges technique rigoureux, structuré autour de quatre axes majeurs :

- **L'environnement de virtualisation** : Afin de ne pas dépendre des aléas d'un réseau physique de production et pour garantir la reproductibilité des tests, l'intégralité de l'infrastructure doit être virtualisée. Nous avons spécifié l'utilisation de GNS3, qui permet l'émulation de matériel réseau tels que des switchs et la virtualisation de machines réelles avec QEMU. Cela nous permet de placer des sondes à n'importe quel point de la topologie.
- **Le système d'exploitation** : Le choix s'est porté sur la distribution linux Debian pour l'ensemble des nœuds serveurs et clients. Ce choix répond à une exigence de stabilité et de légèreté, essentielle lorsque plusieurs machines virtuelles tournent simultanément dans GNS3.
- La chaîne de traitement média
    - **Acquisition et Encodage** : Le système doit utiliser OBS Studio. Il est impératif d'utiliser un encodeur capable de gérer le protocole RTMP en sortie et d'offrir un contrôle précis sur le débit binaire et la structure de, éléments critiques pour nos tests de charge.
    - **Serveur de Diffusion** : Plutôt que d'utiliser une solution propriétaire complexe ou un simple relais VLC, nous avons spécifié l'usage d'OwnCast. Ce choix est stratégique : c'est une solution open-source, légère, auto-hébergée, qui gère nativement la transmutation du flux d'entrée RTMP vers un flux de sortie compatible web HLS.
- **Interopérabilité et Clients** : Le système doit permettre la lecture du flux sur des lecteurs standards. Nous avons retenu VLC Media Player pour sa capacité à lire des flux réseaux bruts et Firefox pour valider l'intégration web via le protocole HLS, simulant ainsi un usage "grand public" standard.

2.3 Argumentaire de la solution retenue

Justification de l'architecture réseau sous GNS3 : D'un point de vue théorique, l'analyse de protocole nécessite un environnement isolé. GNS3 nous permet de créer un "bac à sable" où nous pouvons insérer artificiellement des délais et des pertes via l'outil NetEm intégré aux noyaux Linux. Une solution basée sur un réseau physique simple avec un switch réel n'aurait pas permis de simuler de manière déterministe une perte de paquets de 5% ou une latence de 200ms, rendant l'analyse des performances impossible.

Justification du couple OBS / OwnCast : Le choix de scinder l'architecture en deux entités encodeur vs serveur est justifié par le fonctionnement standard des CDN.

OBS (Source) : Nous avons choisi OBS car il permet de fixer des paramètres d'encodage H.264 et AAC stricts. Pour nos analyses, il est crucial de pouvoir forcer un débit constant pour observer comment le réseau réagit à une charge stable, contrairement à un débit variable qui masquerait certains problèmes de congestion.

OwnCast (Serveur) : Le choix d'OwnCast, par rapport à une solution comme Wowza (souvent payante ou complexe) ou VLC (qui agit plus comme un simple diffuseur), se justifie par sa gestion moderne du protocole HLS. Théoriquement, le HLS découpe le flux en segments .ts listés dans un fichier manifeste .m3u8. Cela permet d'utiliser le protocole TCP avec HTTP, offrant une fiabilité différente du protocole UDP souvent utilisé en RTP pur. Utiliser OwnCast nous permet donc d'analyser le comportement de TCP face aux contraintes réseaux (retransmissions, effondrement de la fenêtre de congestion), ce qui est extrêmement pertinent pour le streaming web moderne.

Justification des Codecs : Conformément aux directives du projet, nous utilisons le codec vidéo H.264 et audio AAC. Le H.264 reste le standard industriel offrant le meilleur compromis entre taux de compression et compatibilité de décodage. Comprendre son comportement (notamment l'impact de la perte d'une image clé "I-frame" vs une image prédictive "P-frame") est au cœur de notre analyse qualitative.

2.4 Adéquation avec les exigences pédagogiques

La solution technique que nous avons déployée répond point par point aux critères d'évaluation définis dans le sujet de la SAE.

Premièrement, la mise en œuvre technique est respectée : nous avons configuré un serveur de streaming complet en utilisant exclusivement des outils open-source (OBS, OwnCast, Debian), démontrant notre capacité à intégrer des logiciels libres dans une chaîne complexe.

Deuxièmement, cette architecture est spécifiquement taillée pour l'analyse des contraintes réseau. En interposant des routeurs virtuels GNS3 entre la machine "OBS" et la machine "OwnCast", ou entre le "Serveur" et le "Client", nous avons créé des points de goulot d'étranglement contrôlables. Cela nous permet de répondre directement à l'objectif de "simuler des conditions réseau telles que la latence et les pertes de paquets" et de capturer ce trafic via Wireshark pour en faire l'analyse.

Enfin, le choix d'OwnCast intégrant une interface web et un chat permet de simuler une véritable plateforme de service, donnant une dimension professionnelle au projet et permettant d'évaluer la qualité de service non seulement via des métriques réseaux, mais aussi via le ressenti utilisateur final (fluidité de la vidéo, synchronisation audio/vidéo).

Chapitre 3 : Présentation des outils et du matériel

3.1 Système d'Exploitation : Debian GNU/Linux

Pour l'ensemble des nœuds serveurs et clients simulés au sein de notre infrastructure, nous avons opté pour la distribution Debian GNU/Linux, plus précisément la version 13 "Trixie" pour sa stabilité et empreinte mémoire. Dans un environnement virtualisé sous GNS3 où les ressources sont partagées, l'utilisation de Debian en mode "headless" est cruciale. Une installation minimale consomme moins de 500 Mo de RAM, permettant de multiplier les nœuds sans saturer l'hôte physique. La gestion des paquets et réseau a également été un point important pour le choix de ce système d'exploitation, ainsi que le standard serveur assurant une compatibilité parfaite avec les binaires de services comme OwnCast ou les bibliothèques FFmpeg.

3.2 Source et Encodage : OBS Studio

Le point d'entrée de notre chaîne de transmission est assuré par OBS Studio, qui fonctionne non seulement comme outil de capture mais surtout comme encodeur temps réel et client RTMP.

- **Encodage Vidéo (H.264/AVC) :** Nous privilégions l'implémentation logicielle x264 pour l'encodage vidéo. Pour garantir une analyse réseau pertinente, le contrôle du débit est configuré en CBR plutôt qu'en VBR ce qui force l'encodeur à bourrer le flux pour maintenir un débit stable et permet ainsi de faciliter la détection des goulots d'étranglement réseau.
- **Structure du GOP (Group of Pictures) :** Un paramètre critique configuré est l'intervalle d'images clés, fixé à 2 secondes. Cette régularité est essentielle pour la segmentation HLS en aval : le serveur de découpe ne peut segmenter le flux que sur une image intra.
- **Protocole d'ingestion :** OBS encapsule les paquets audio (AAC) et vidéo (NAL units H.264) dans un flux RTMP, transmis via TCP sur le port 1935 vers le serveur OwnCast.

3.3 Cœur de la Diffusion : OwnCast

Pour le serveur de streaming, nous avons privilégié OwnCast au détriment de solutions classiques comme VLC ou des serveurs lourds tels que Wowza. Cette application développée en Go est conçue pour être autonome ("self-hosted") et performante.

- **Ingestion RTMP et Transmuxing :** OwnCast reçoit le flux RTMP brut émis par OBS sur le port TCP 1935. Contrairement à un simple répéteur passif qui se contenterait de relayer les paquets sans modification, OwnCast applique un traitement actif de "transmuxing" en extrayant les pistes audio AAC et vidéo du conteneur RTMP pour les reconditionner en segments MPEG-TS compatibles HLS.
- **Distribution HLS (HTTP Live Streaming) :** Le rôle majeur d'OwnCast est la génération dynamiquement des segments vidéo courts au format .ts et la mise à jour en temps réel du fichier manifeste .m3u8. Cette segmentation temporelle, typique de protocole HTTP Live Streaming, assure une diffusion adaptative et résiliente, adaptée aux variations de bande passante.
- **Serveur Web Intégré :** Un serveur web léger est intégré à OwnCast pour servir ces fichiers statiques via HTTP. Cela permet de simuler un trafic web standard tel que le TCP plutôt qu'un flux UDP continu, nous permettant d'analyser le comportement du protocole TCP Window Scalings face aux pertes de paquets.

OwnCast nous apporte également un avantage sur VLC. VLC diffuse souvent en RTP/RTSP, peu-interopérable avec les navigateurs modernes sans plugin. Cependant, OwnCast fournit une expérience "client-less" native HTML5 via HLS, alignée sur les standards web actuels et conforme aux objectifs de modernisation des technologies multimédias.

3.4 La Conteneurisation : Docker

Pour le déploiement du serveur de diffusion OwnCast, nous avons fait le choix technologique de la conteneurisation via Docker, plutôt qu'une installation traditionnelle « en dur » sur le système Debian.

Docker est une plateforme permettant d'embarquer une application avec l'ensemble de ses dépendances (librairies, fichiers de configuration, binaires) dans un conteneur isolé.

- **Différence avec la virtualisation (VM) :** Contrairement à une machine virtuelle qui émule un matériel complet et nécessite son propre système d'exploitation invité, souvent lourd en ressources, un conteneur Docker partage le noyau de l'hôte Linux, qui est notre Debian VM sous GNS3.
- **Avantages techniques :** Le conteneur Docker présente plusieurs avantages déterminants dans ce contexte notamment sa légèreté puisque le conteneur ne pèse que quelques mégaoctets et démarre quasi instantanément ce qui optimise l'utilisation des ressources limitées de l'environnement GNS3. L'image Docker d'OwnCast apporte de la portabilité. Il fonctionnera exactement de la même manière sur notre machine de test GNS3 que sur un serveur de production, éliminant l'effet "ça marche chez moi mais pas sur le serveur". Enfin, l'isolation. Le service est isolé

du reste du système. Si le serveur de streaming plante ou est compromis, cela n'affecte pas l'OS hôte.

Dans le cadre de cette SAÉ, Docker nous permet de déployer une instance de serveur de streaming propre et standardisée en une seule ligne de commande, facilitant grandement la phase de configuration.

3.5 Émulation et Virtualisation Réseau : GNS3

L'infrastructure réseau est entièrement virtualisée via GNS3. Cet outil agit comme un orchestrateur connectant des machines virtuelles réelles (via QEMU/KVM) et des équipements réseaux émulés.

GNS3 nous permet de ne pas simuler le réseau de manière abstraite, mais de faire transiter de vrais paquets IP entre les VM Debian. Le switch et les routeurs virtuels gèrent le réseau de manière identique à du matériel physique.

C'est au niveau des liens inter-nœuds de GNS3 que nous configurons les paramètres de qualité de service. Nous utilisons les interfaces de configuration pour introduire artificiellement : de la latence (ex: 100ms) pour simuler un lien satellite ou transcontinental. Ainsi que la perte de paquets (Packet Loss) (ex: 1% à 5%) pour simuler un réseau Wi-Fi instable ou congestionné.

3.6 Analyse Métrologique : Wireshark

Enfin, l'analyse des résultats de notre implémentation repose sur l'analyseur de protocoles réseau standard, Wireshark qui offre les utilisations avancées dans notre topologie GNS3.

Utilisation avancée :

- **Sondes :** Wireshark est "branché" sur les liens virtuels de GNS3 via l'interface uBridge. Cela permet de capturer le trafic en mode promiscuité entre le diffuseur et le serveur OwnCast.
- **Dissecteurs de protocoles :** Nous utilisons les filtres de décodage spécifiques pour isoler :
    - Le flux RTMP entre OBS et OwnCast permettant d'analyser du handshake et de la taille des chunks.
    - Le flux HTTP/HLS entre OwnCast et le Client permettant d'analyser des requêtes GET pour les fichiers .ts et surveillance des retransmissions TCP, TCP Retransmission / Duplicate ACKs, qui signalent une dégradation de la qualité réseau.

Chapitre 4 : Présentation de notre système de transmission

4.1 Architecture Globale (Topologie GNS3)

L'architecture déployée dans le simulateur GNS3 modélise un réseau local commuté. Elle est constituée de trois nœuds principaux fonctionnant sous Debian 13, interconnectés via un switch virtuel Ethernet.

Voici la topologie GNS3 mis en place :

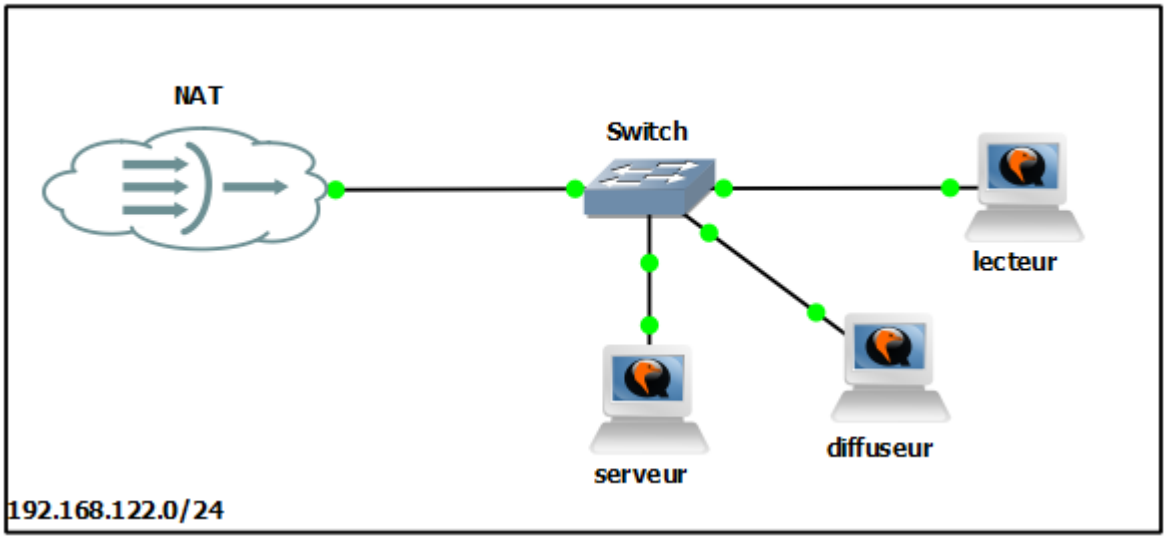


Figure 1 - Toplogie du réseau mis en place

```
gns3@serveur:~/OwnCast$ sudo cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
allow-hotplug ens3
iface ens3 inet dhcp
gns3@serveur:~/OwnCast$
```

Figure 2 - Configuration d'OwnCast

Chaque nœud joue un rôle important :

- **Serveur :** Cette machine héberge OBS Studio. Elle injecte le flux vidéo. Pour garantir une source de test standardisée et reproductible, nous utilisons le fichier vidéo de référence "Big Buck Bunny", configuré pour une sortie en 1080p à 30 images par seconde.

- **Diffuseur :** Cette machine héberge OwnCast. Elle agit comme un pivot : elle reçoit le flux continu et le transforme en fichiers segmentés pour le web.
- **Lecteur** : La vidéo peut être visionné sur cette machine via un lecteur vidéo tel que VLC ou un navigateur web comme Firefox/
- **Switch :** L'élément central de notre expérimentation est le lien Ethernet reliant le serveur, le diffuseur et le lecteur.

Dans GNS3, nous appliquons des filtres en cliquant sur ces liens afin d'introduire la latence, la gigue, la perte et coupure de paquets simulant ainsi la dégradation physique du média de transmission.

4.2 Protocole d'Ingestion : Analyse du Flux RTMP

La liaison entre le serveur et le diffuseur est assurée par le protocole RTMP. Ce choix est dicté par la nécessité d'une faible latence à l'envoi et d'une connexion persistante TCP.

Lors de l'analyse du trafic via Wireshark, nous observons que la communication s'établit en trois phases distinctes :

- **Handshake RTMP :** OBS initie la connexion avec les paquets C0 (version 3), C1 (timestamp 0x00000000 et bytes zéro) et C2. OwnCast répond par S0, S1 et S2, validant la compatibilité protocolaire avant toute transmission média.
- **Commandes AMF :** OBS transmet les commandes AMF encodées "connect" (paramètres : app="live", flashVer="Linux 6.12") suivie de "publish" (streamKey="abc123", type="live"), définissant nom et caractéristiques du flux.
- **Transmission des données :** RTMP découpe le flux en chunks de 600 octets : Basic Header + Message Header + payload. Ce multiplexage intercale AAC et H.264 sur la même session TCP, assurant une synchronisation audio-vidéo précise dès l'ingestion. Les paquets dominants sont PSH/ACK à MTU maximal (~1500 o), avec ~95% dédiés à la vidéo.

Ce mécanisme de multiplexage permet d'intercaler les données audio et vidéo sur la même connexion, assurant une synchronisation labiale précise avant même l'arrivée sur le serveur.

4.3 Protocole de Diffusion : Analyse du Flux HLS

En sortie du serveur OwnCast, le protocole change pour adopter le HLS. Ce protocole est basé sur HTTP, TCP dans notre configuration.

Le fichier manifeste '**.m3u8**' est un fichier texte utilisé, indexant les segments disponibles. Le client le télécharge périodiquement, toutes les quelques secondes, pour savoir quel est le prochain segment à lire. Les Segments **.ts**, est le flux vidéo 1080p découpé en fichiers conteneurs MPEG-TS d'une durée fixe.

Du côté client, nous n'observons pas un flux constant, mais des pics de téléchargement (Burst) correspondant aux requêtes GET /hls/segment_X.ts. Chaque segment contient une

portion de la vidéo commençant obligatoirement par une image clé. En conditions dégradées, retransmissions TCP et Duplicate ACKs apparaissent sur les bursts segment .ts.

4.4 Analyse des Codecs et Encapsulation

La réussite de la transmission vidéo repose sur le choix des codecs et leurs configurations.

Nous avons choisi le codec **H.264 MPEG-4 AVC** pour notre flux vidéo. Au niveau binaire, le flux est composé d'unités NAL. L'analyse des trames permet d'identifier les Sequence Parameter Sets et PPS Picture Parameter Sets qui contiennent les métadonnées de résolution, suivis des NAL units de type IDR et des P-frames soit les trames Predictive permettant de voir les différences.

En parallèle du H.264, l'audio nécessite une synchronisation parfaite. C'est pourquoi nous avons choisi le codec **Advanced Audio Coding** (AAC) pour notre flux audio. Ce codec encapsule l'audio avec des en-têtes ADTS pour permettre le décodage synchronisé. Le débit audio étant faible par rapport à la vidéo (ex: 128 kbps vs 4000 kbps), il est moins sensible à la congestion mais très sensible à la gigue.

Cette complémentarité H.264/AAC, encapsulée dans RTMP puis segmentée en HLS, garantit une synchronisation multimédia robuste tout en exposant les limites spécifiques de chaque piste face aux dégradations réseau observées dans nos expérimentations.

4.5 Justification de la séparation RTMP / HLS

Cette architecture hybride avec RTMP et HLS est justifiée par les contraintes de la production et de la diffusion, telles que la rapidité, la stabilité et le débit. Entre OBS et OwnCast, nous avons besoin de rapidité et de stabilité. C'est pourquoi nous utilisons RTMP sur cette connexion TCP persistante, Stateful, qui est optimal pour un débit important tel que 1080p sans overhead excessif.

Pour assurer compatibilité multi-navigateurs et résilience réseau, nous adoptons HLS. Si un segment .ts est corrompu lors du téléchargement à cause d'une mauvaise qualité de câble simulée dans GNS3, le client TCP redemandera les paquets manquants avant de passer le segment au lecteur, créant une mise en mémoire tampon (buffering) plutôt que des artefacts visuels immédiats.

Chapitre 5 : Présentation de la partie expérimentale et des résultats

5.1 Mise en place de l'environnement

La phase d'installation a consisté à configurer chaque maillon de la chaîne pour assurer une interopérabilité totale.

5.1.1 Configuration de l'encodeur OBS Studio

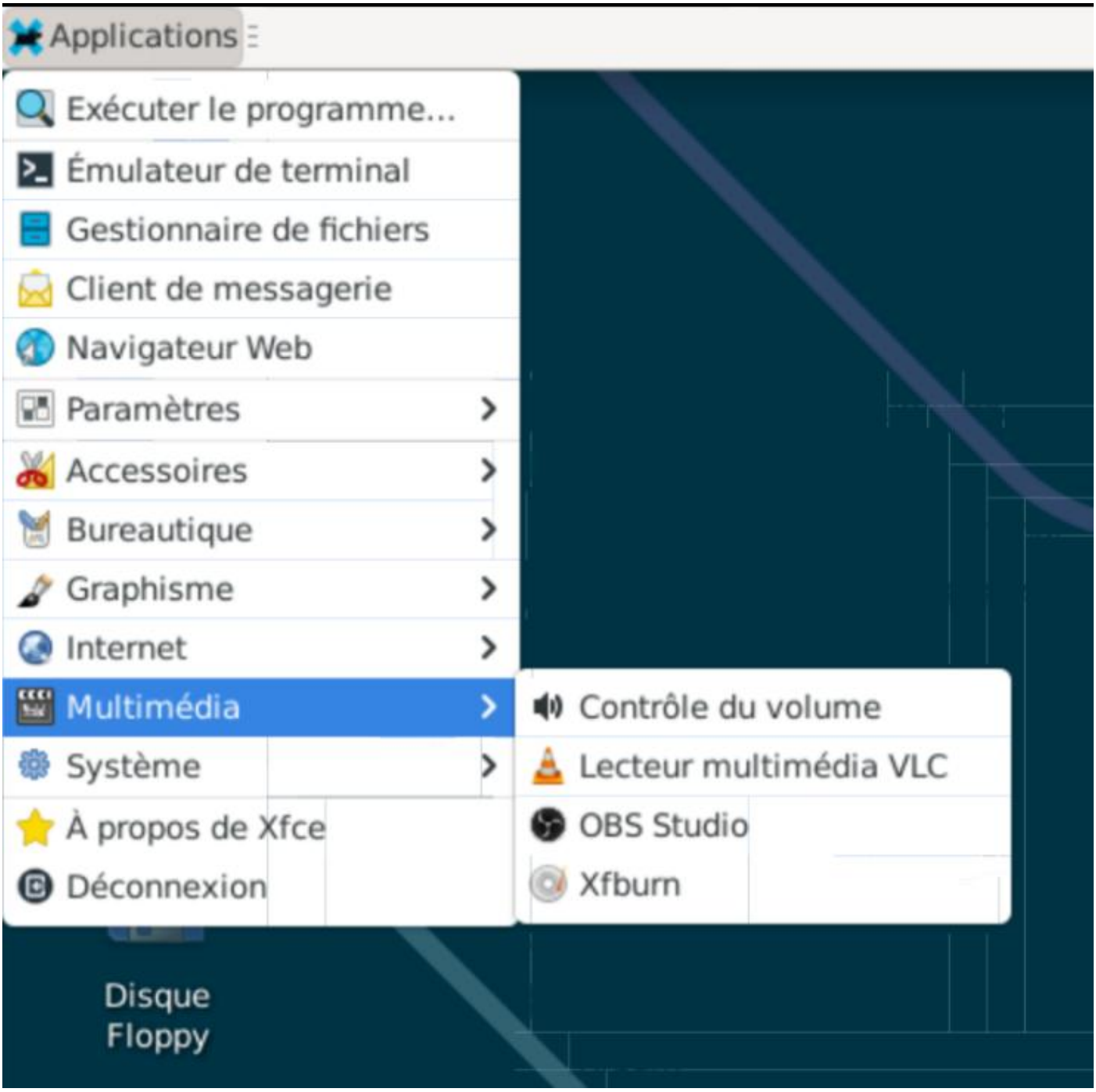


Figure 3 - OBS Studio installé sur la machine Debian

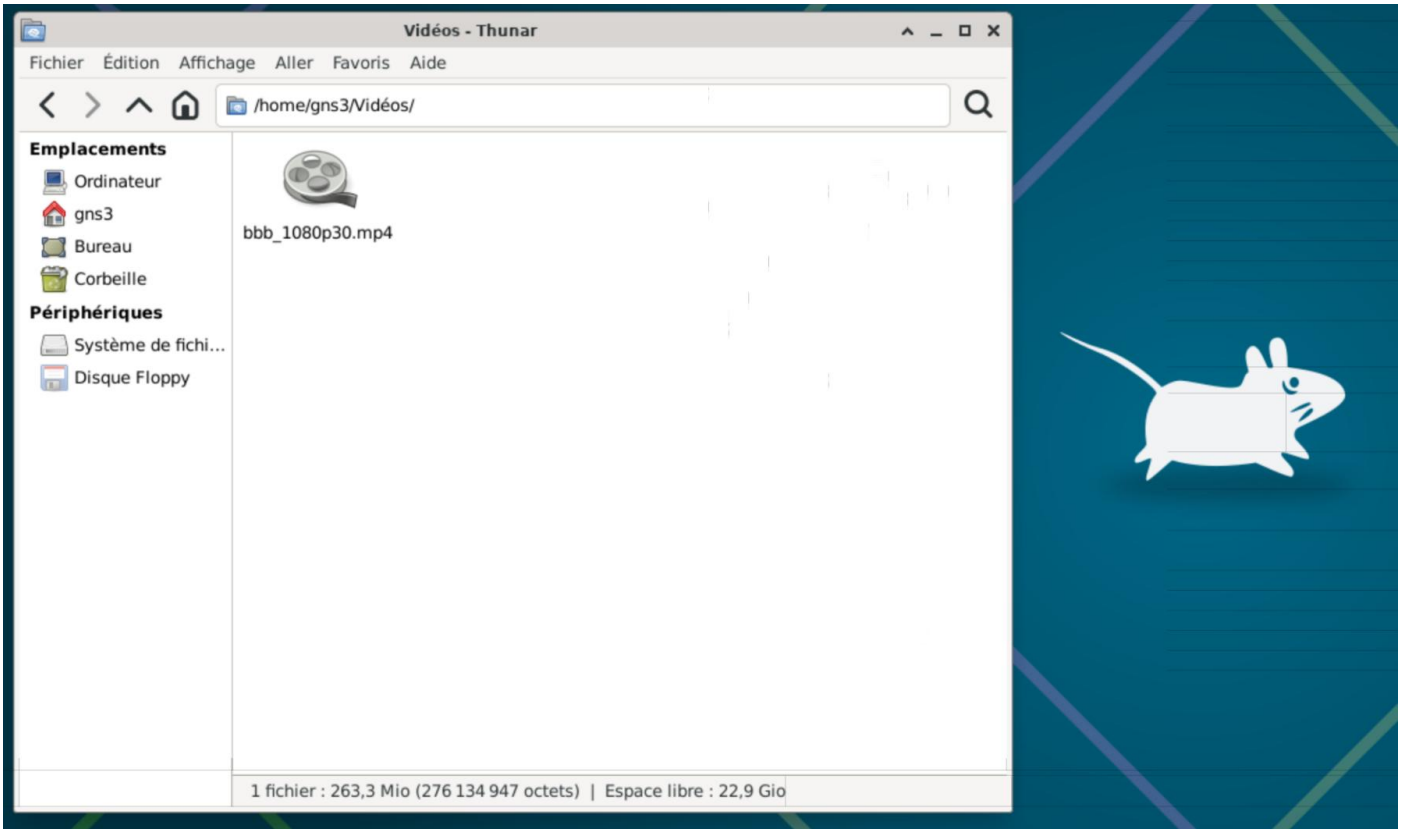


Figure 4 – Importation de la vidéo de Test Big Buck Bunny

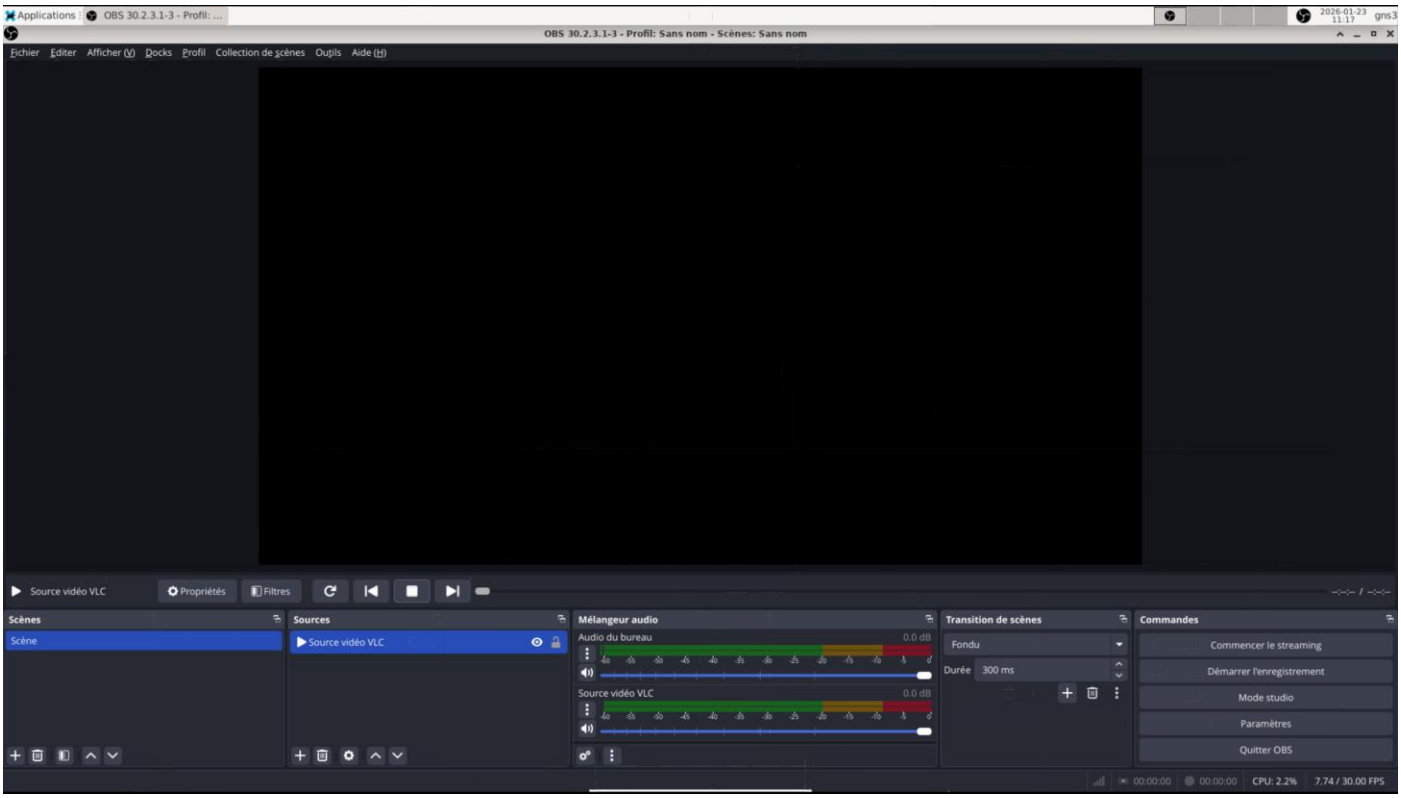


Figure 5 - L'interface OBS Studio

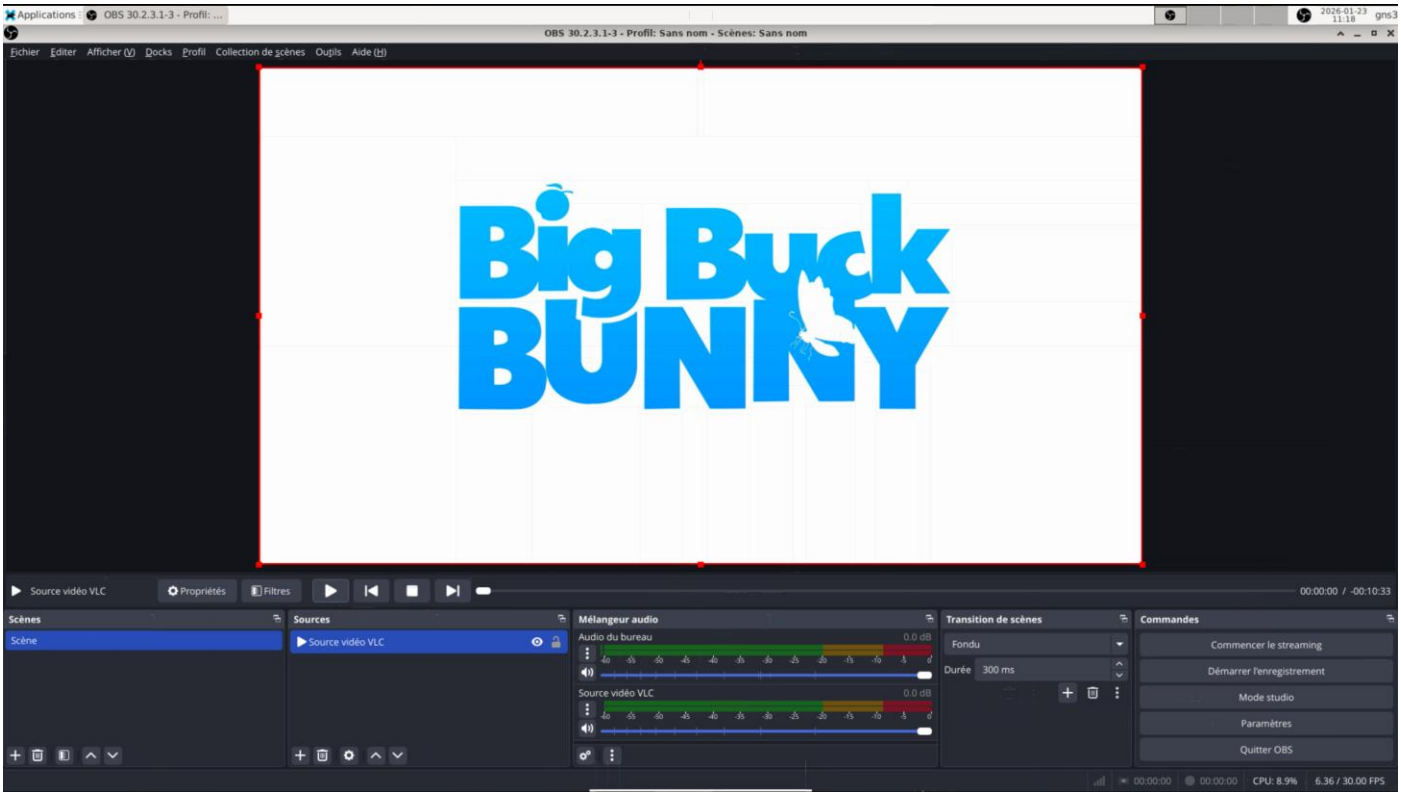


Figure 6 - Importation de la vidéo de test réussi

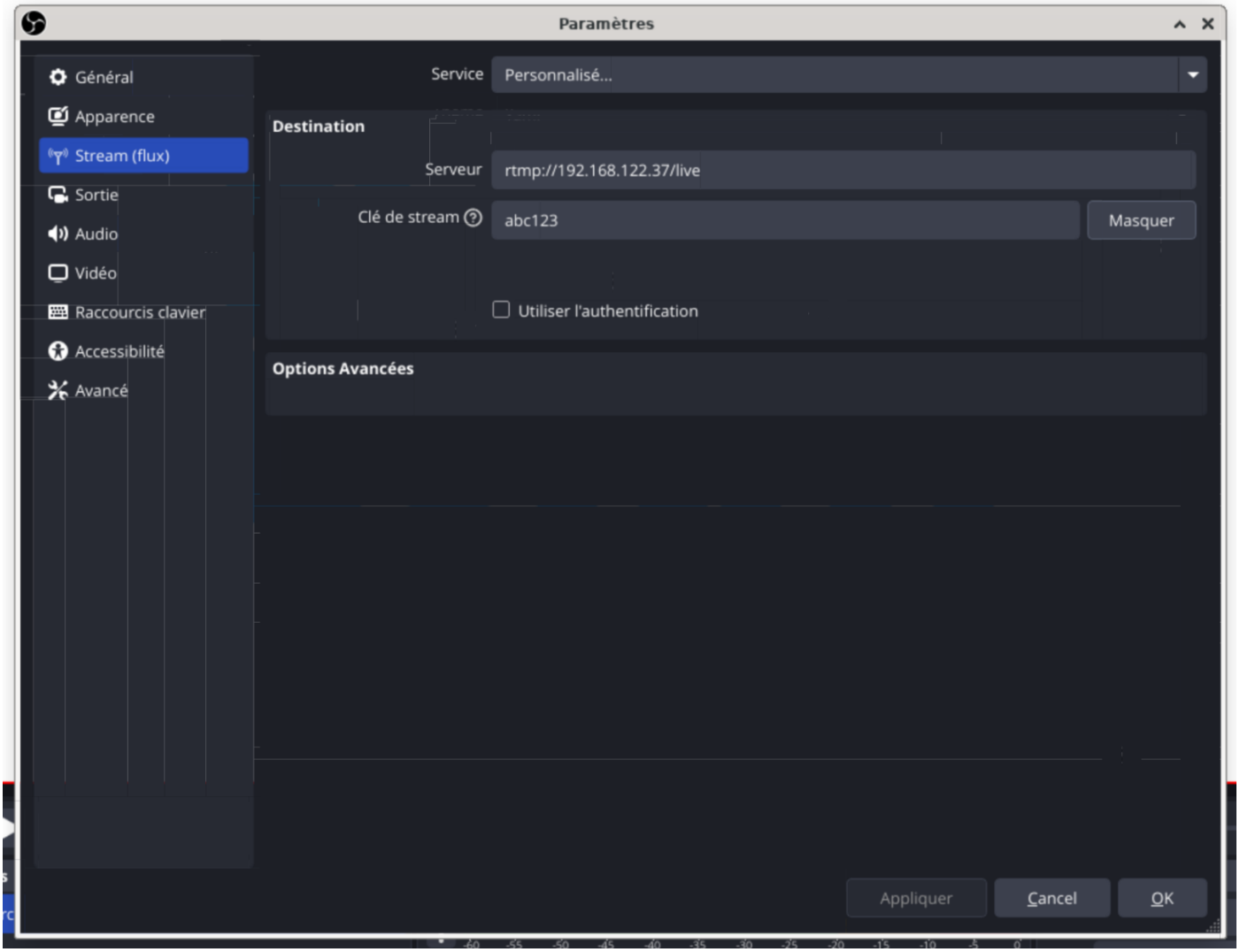


Figure 7 - Interface de réglage de flux

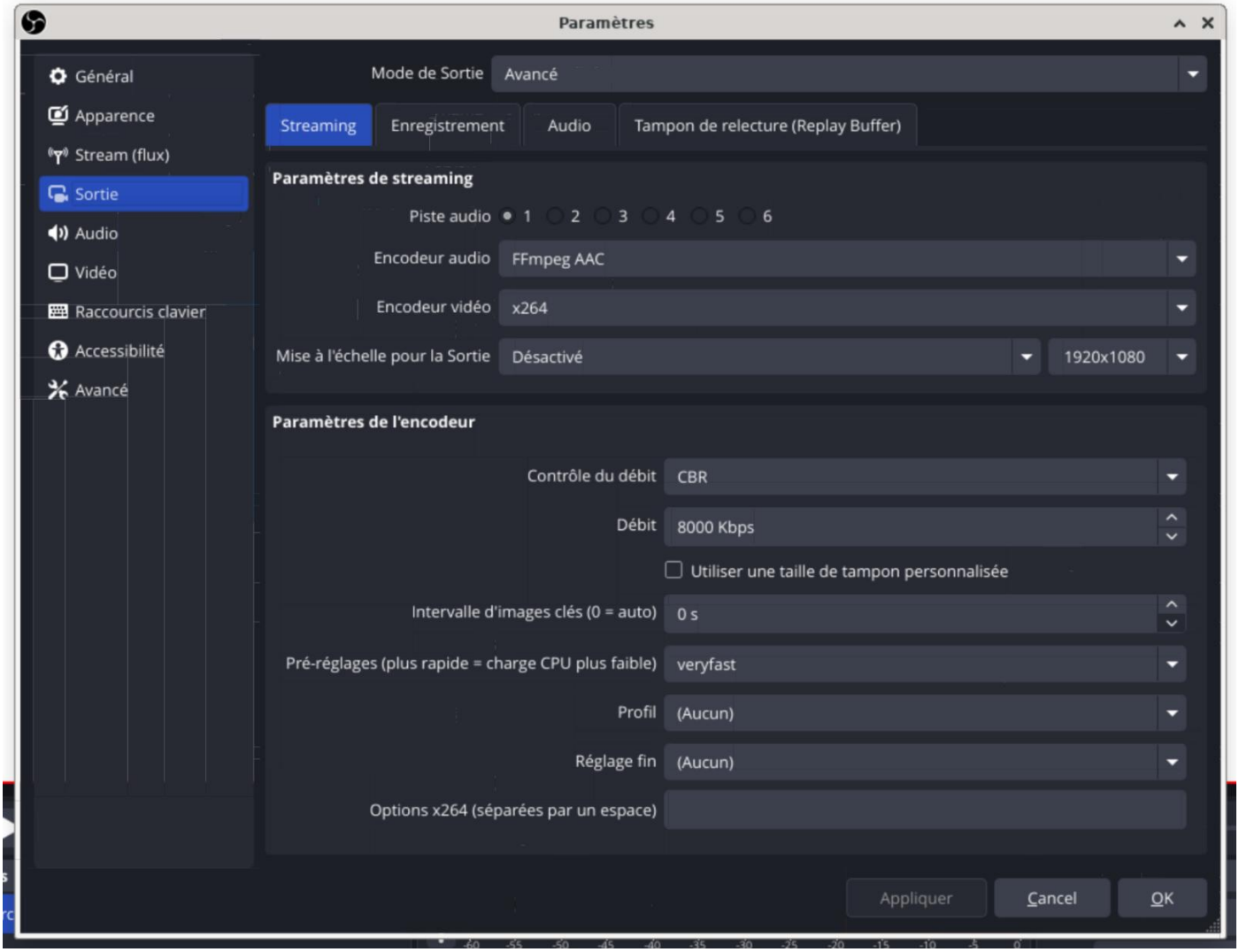


Figure 8 - Interface de réglage de sortie de stream

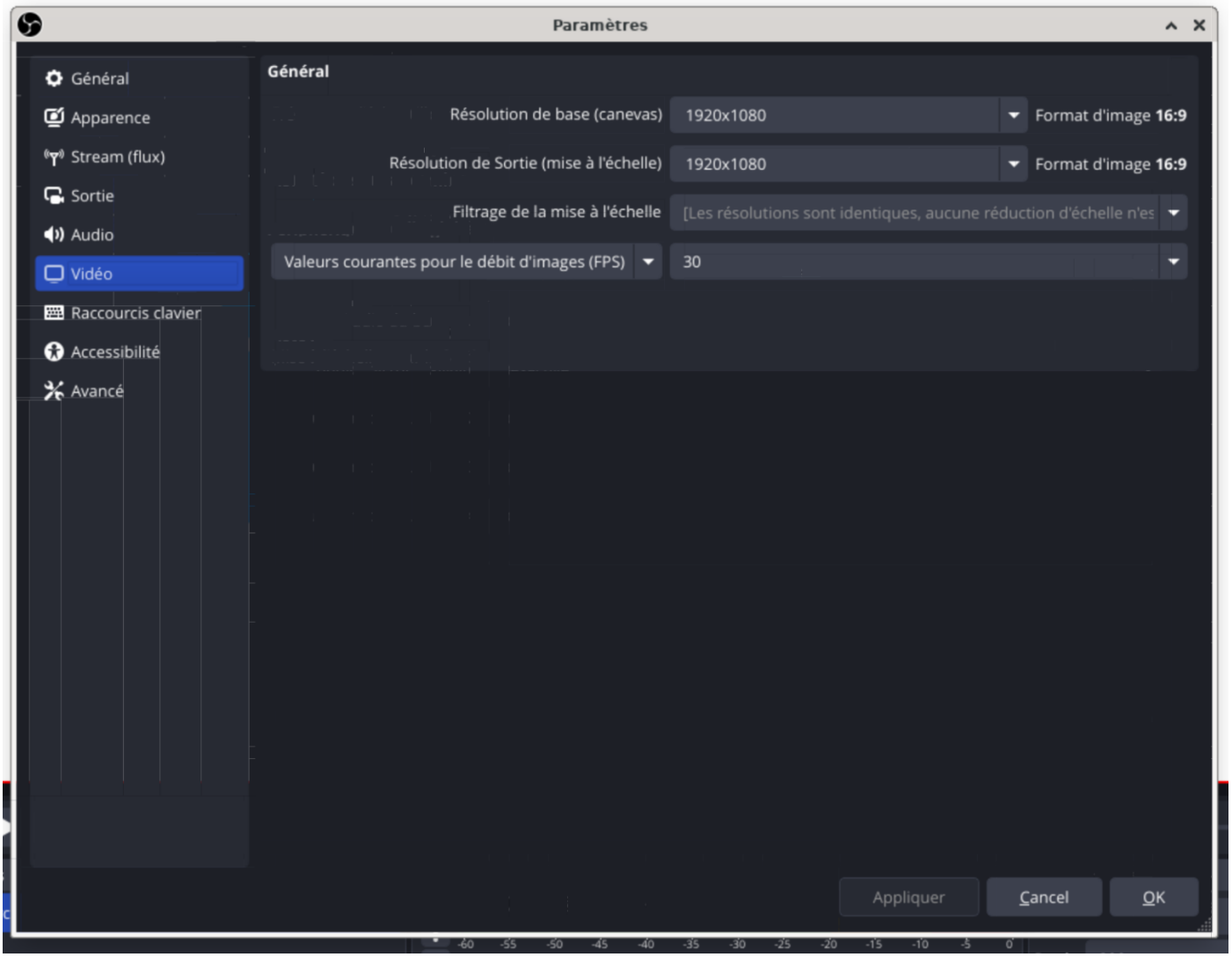


Figure 9 - Interface de réglage des paramètres vidéo

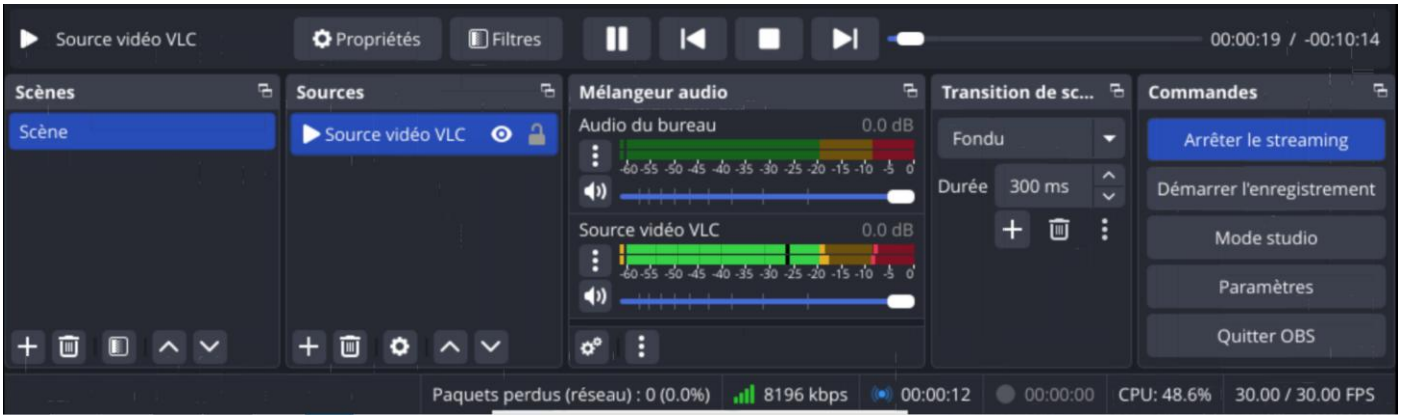


Figure 10 - Console du contrôle du son et le stream

Pour garantir la pertinence des tests, nous avons figé les paramètres d'encodage afin d'obtenir un flux déterministe. Contrairement à un usage ludique où les paramètres sont automatiques, nous avons forcé les valeurs suivantes :

- **Contrôle du débit avec CBR**. Cela nous permet de générer une charge réseau stable.
- **Débit cible** : 2500 Kbps (Vidéo) + 160 Kbps (Audio). Ce débit est choisi pour être réaliste (qualité 720p/1080p fluide) tout en restant inférieur à la capacité maximale des liens virtuels GNS3 par défaut.

- **Keyframe Interval** : Fixé strictement à 2 secondes.

L'utilisation de ces paramètres était cruciale puisque sans cette utilisation, le protocole HLS découperait la vidéo en segments. Cependant, chaque segment doit commencer par une "I-Frame". Si nous laissons ce paramètre en "auto" (souvent 10s), OwnCast ne pourrait pas créer de segments courts, augmentant drastiquement la latence de diffusion.

5.1.2 Déploiement orchestré via Docker Compose

```
gns3@serveur:~/OwnCast$ ls
compose.yaml  data
gns3@serveur:~/OwnCast$ cat compose.yaml
services:
  owncast:
    image: owncast/owncast:latest
    container_name: owncast
    restart: unless-stopped
    ports:
      - "80:8080"
      - "1935:1935"
    volumes:
      - ./data:/app/data
    environment:
      - TZ=Europe/Paris
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
gns3@serveur:~/OwnCast$ docker compose ps
NAME      IMAGE                  COMMAND          SERVICE   CREATED      STATUS             PORTS
owncast   owncast/owncast:latest   "/app/owncast"   owncast   8 days ago   Up About a minute   0.0.0.0:1935->1935/tcp, [::]:1935->1935/tcp, 0.0.0.0:80->8080/tcp, [::]:80->8080/tcp
gns3@serveur:~/OwnCast$
```

Figure 11 - Les commandes utilisées pour le déploiement du Docker

```
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Owncast v0.2.4-linux-64bit (7929580b98a6a84806b62f97c04306c17217ba38)"
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Initializing ActivityPub outbound worker pool with 10 workers for 0 followers"
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Web server is listening on port 8080."
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Configure this server by visiting /admin."
gns3@serveur:~/OwnCast$
```

Figure 12 - Docker Compose

Pour garantir la reproductibilité de l'installation et simplifier la maintenance du service sur le serveur Debian, nous avons opté pour Docker Compose. Contrairement à une commande docker run manuelle et volatile, Docker Compose nous permet de définir l'infrastructure du service dans un fichier de configuration déclaratif sous YAML.

Mise en place de l'environnement :

Nous avons d'abord installé le plugin Compose sur la machine Debian, puis créé l'arborescence nécessaire pour accueillir les données persistantes du serveur.

Configuration du service :

Nous avons créé un fichier nommé 'docker-compose.yml' à la racine du projet. Ce fichier agit comme la "recette" de notre serveur de streaming. Voici la configuration utilisée :

```yaml
version: '3'

services:

 owncast:

  image: owncast/owncast:latest

  container_name: owncast

  restart: unless-stopped

  ports:

   - "8080:8080" # Port Web HLS

   - "1935:1935" # Port Ingest RTMP

  volumes:

   - ./data:/app/data # Persistance des configurations et logs
```

Figure 13 - Script YAML utilisé pour la configuration du Docker Compose

Cette configuration définit plusieurs points critiques pour notre architecture réseau :

Nous avons retenu l'image officielle owncast/owncast:latest pour bénéficier des dernières optimisations HLS. Nous avons ouvert le port 1935/TCP pour l'ingestion RTMP depuis OBS via GNS3 et 8080/TCP pour l'interface web et segments .ts pour clients. Nous avons monté le volume ./data:/app/data pour garantir la persistance des configurations et logs lors des redémarrages conteneurs. La politique restart: unless-stopped pour assurer la résilience automatique après reboot Debian. Le déploiement s'effectue via docker compose up -d, validé par docker compose ps confirmant l'état "Up". L'argument -d (detached) lance le conteneur en arrière-plan.

Nous avons validé le bon fonctionnement du service via la commande **docker compose ps** qui nous confirme que l'état est "Up" et que les ports sont correctement liés à l'interface réseau de la machine virtuelle.

```
gns3@serveur:~/OwnCast$ ls
compose.yaml  data
gns3@serveur:~/OwnCast$ cat compose.yaml
services:
  owncast:
    image: owncast/owncast:latest
    container_name: owncast
    restart: unless-stopped
    ports:
      - "80:8080"
      - "1935:1935"
    volumes:
      - ./data:/app/data
    environment:
      - TZ=Europe/Paris
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
gns3@serveur:~/OwnCast$ docker compose ps
NAME      IMAGE                      COMMAND          SERVICE   CREATED      STATUS             PORTS
owncast   owncast/owncast:latest   "/app/owncast"   owncast   8 days ago   Up About a minute   0.0.0.0:1935->1935/tcp, [::]:1935->1935/tcp, 0.0.0.0:80->8080/tcp, [::]:80->8080/tcp
gns3@serveur:~/OwnCast$
```

Figure 14 - Bonne création du fichier Compose.yaml

5.1.3. Installation et Configuration d'OwnCast (Serveur Debian)

Sur le serveur Debian virtualisé, l'installation d'OwnCast a été réalisée en ligne de commande. Le fichier de configuration config.yaml a été édité pour activer l'écoute RTMP sur le port standard 1935, pour définir le dossier de sortie des segments HLS (/data/hls) et

pour configurer la rétention des segments, c'est-à-dire, le nombre de segments dans la playlist .m3u8, afin de gérer la taille du buffer côté serveur.

```
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Owncast v0.2.4-linux-64bit (7929580b98a6a84806b62f97c04306c17217ba38)"
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Initializing ActivityPub outbound worker pool with 10 workers for 0 followers"
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Web server is listening on port 8080."
owncast  | time="2026-01-23T10:11:25Z" level=info msg="Configure this server by visiting /admin."
gns3@serveur:~/OwnCast$
```

Figure 15 - OwnCast running

OwnCast a bien été installé et fonctionne sur le port 8080.

5.2 Fonctionnement

Une fois la configuration terminée, depuis le lecteur nous accédons à l'interface web du streaming : http://100.96.103.28/:

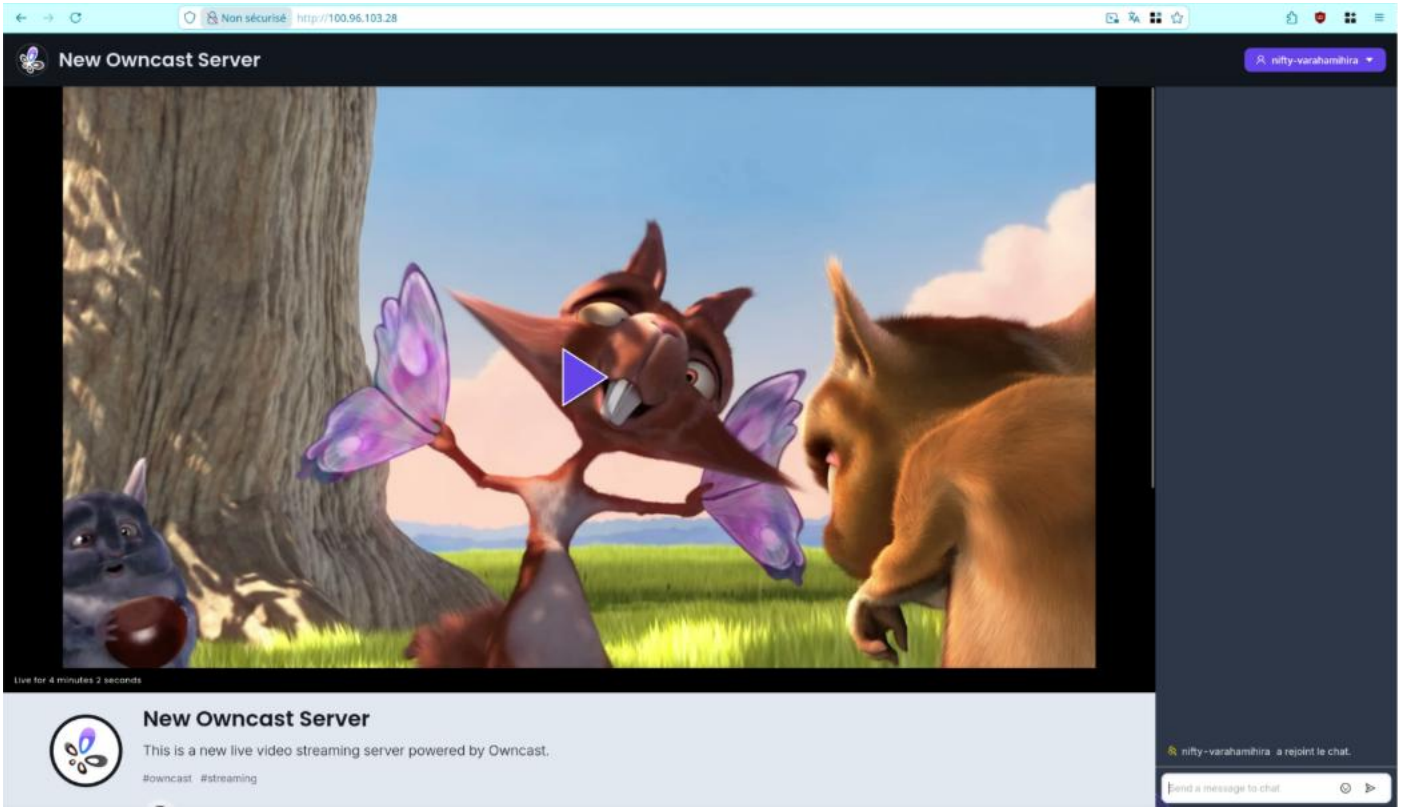


Figure 16 - Le stream sur le serveur Owncast

Le lien nous redirge vers l'interface web OwnCast complète avec le lecteur vidéo HLS HTML5, le chat intégré et les métadonnées en temps réel, tels que Viewer Count, Uptime et la resolution.

Le flux test "Big Buck Bunny" en 1080p s'affiche de manière fluide dans Firefox, confirmant l'intégralité de la chaîne technique : OBS Studio -> réseau GNS3 -> OwnCast -> diffusion web.

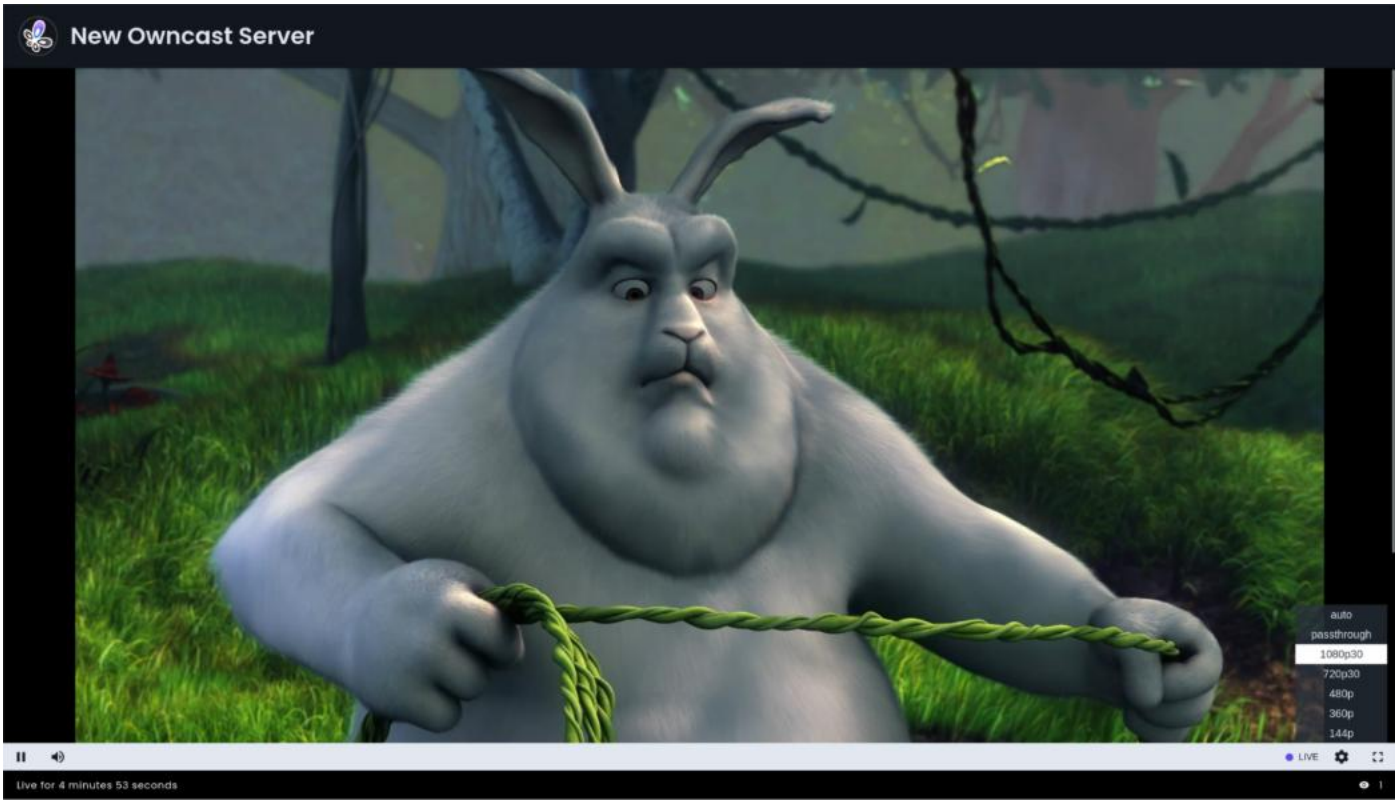


Figure 17 - Visualisation du stream

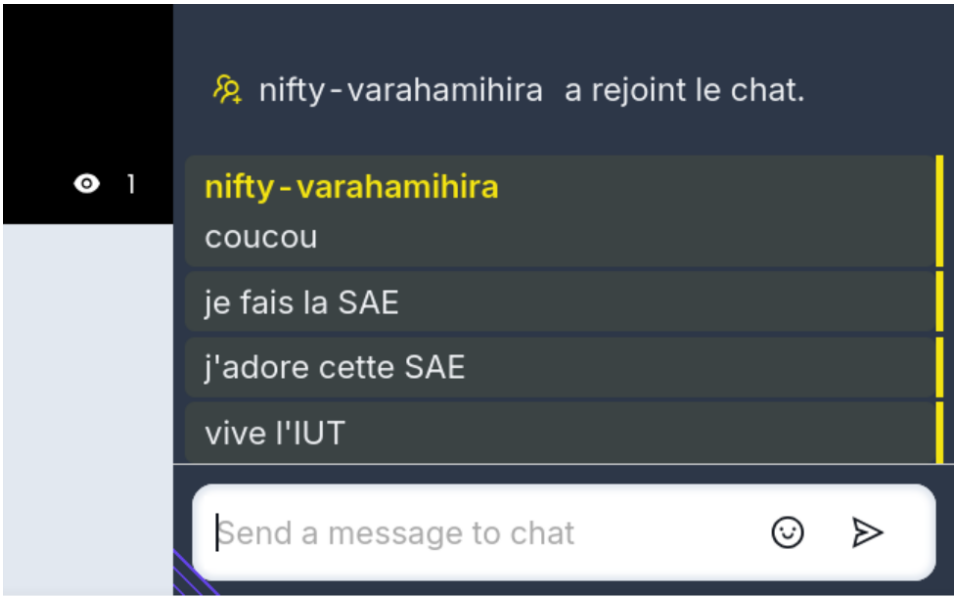


Figure 18 - Chat OwnCast

5.3 Analyse des résultats

L'expérimentation démontre une corrélation directe entre le protocole TCP et la fluidité visuelle. Le choix du HLS basé sur TCP privilégie l'intégrité de l'image pas d'artefacts au détriment de la latence et de la continuité.

Nous avons également accés au menu Admin de OwnCast:

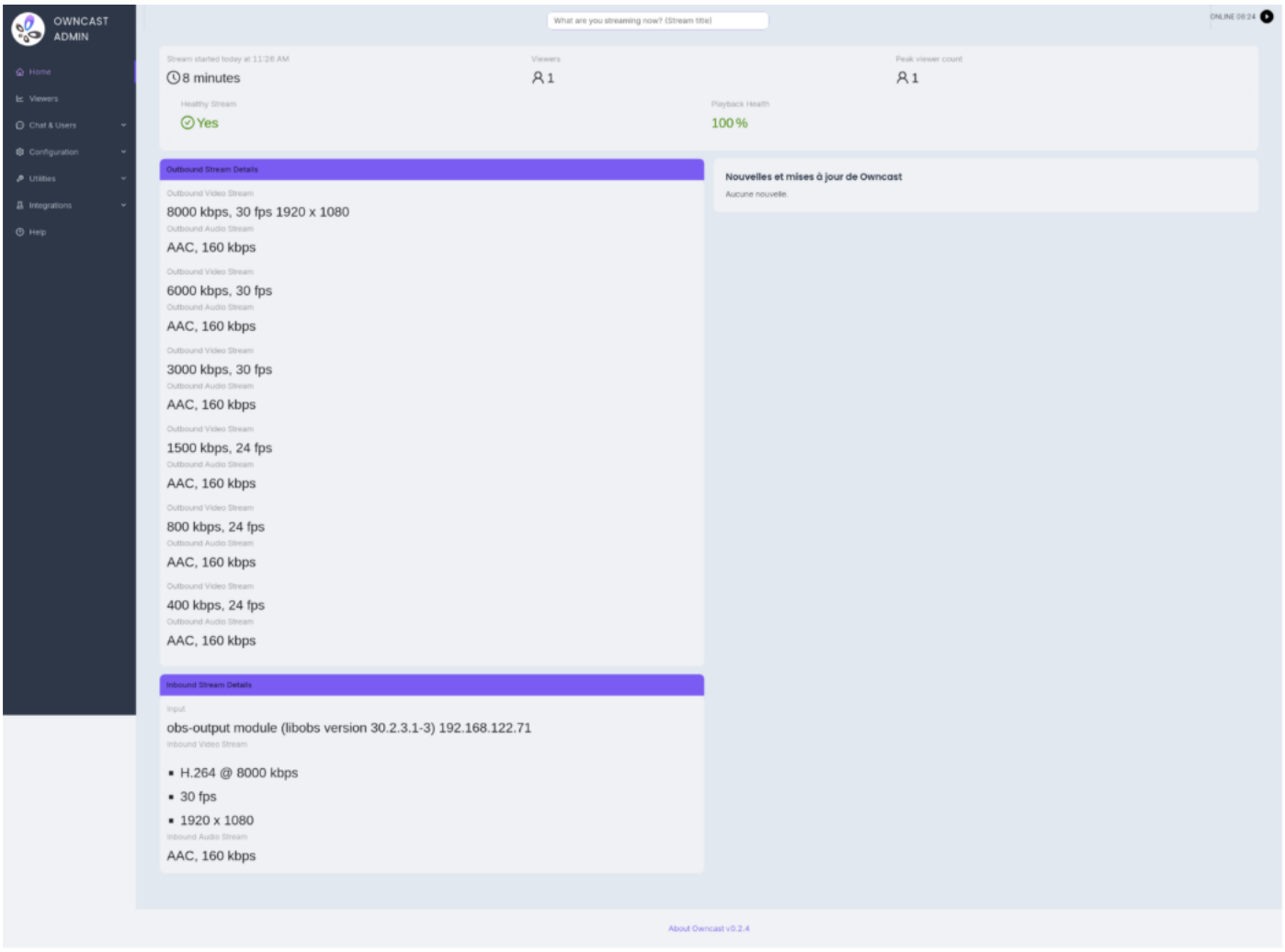


Figure 19 - Menu Admin OwnCast

Nous avons la possibilité d'avoir les informations sur le visionnage du stream :

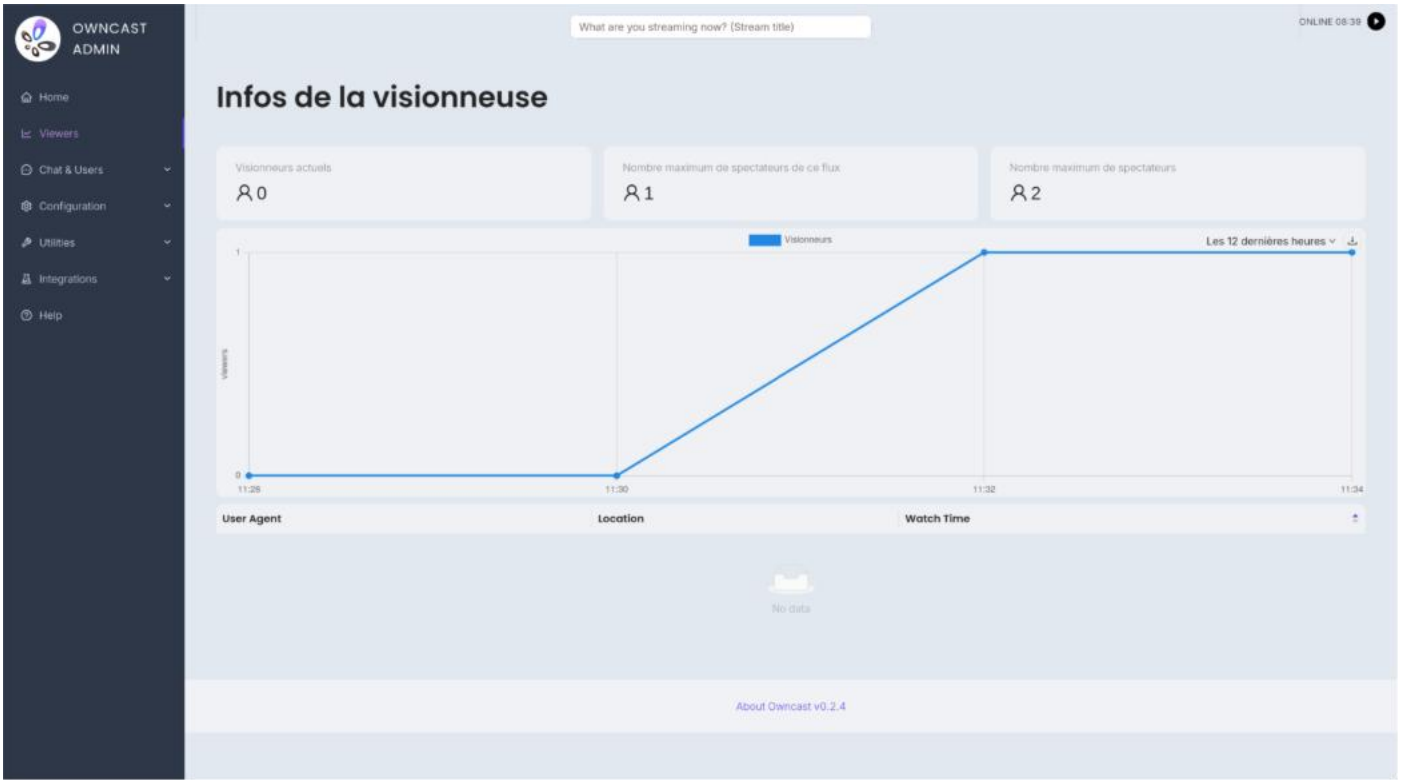


Figure 20 - Interface d'infos de la visionneuse

Nous avons un chat qui peut être surveillé :

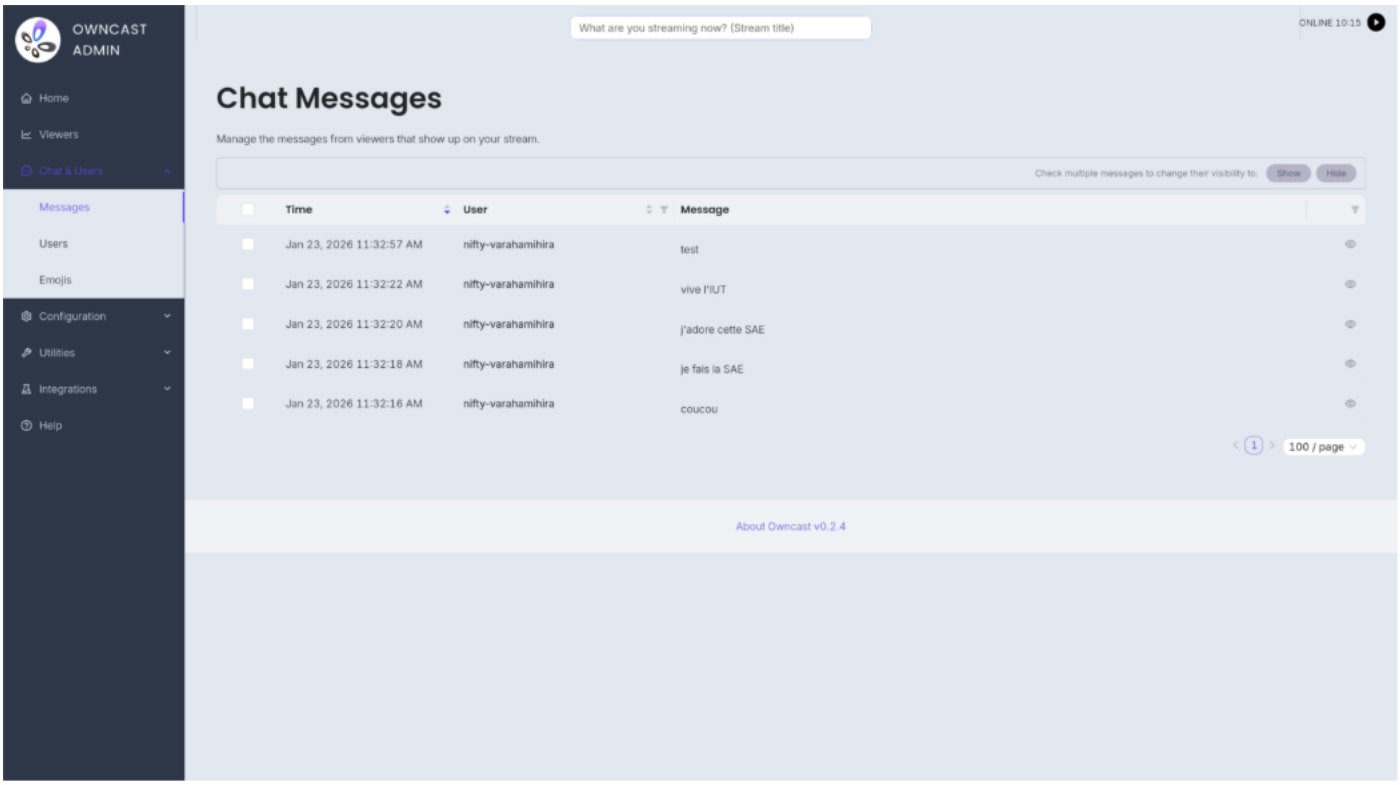


Figure 21 – L'interface admin du chat

L'utilisation des emojis est également possible :

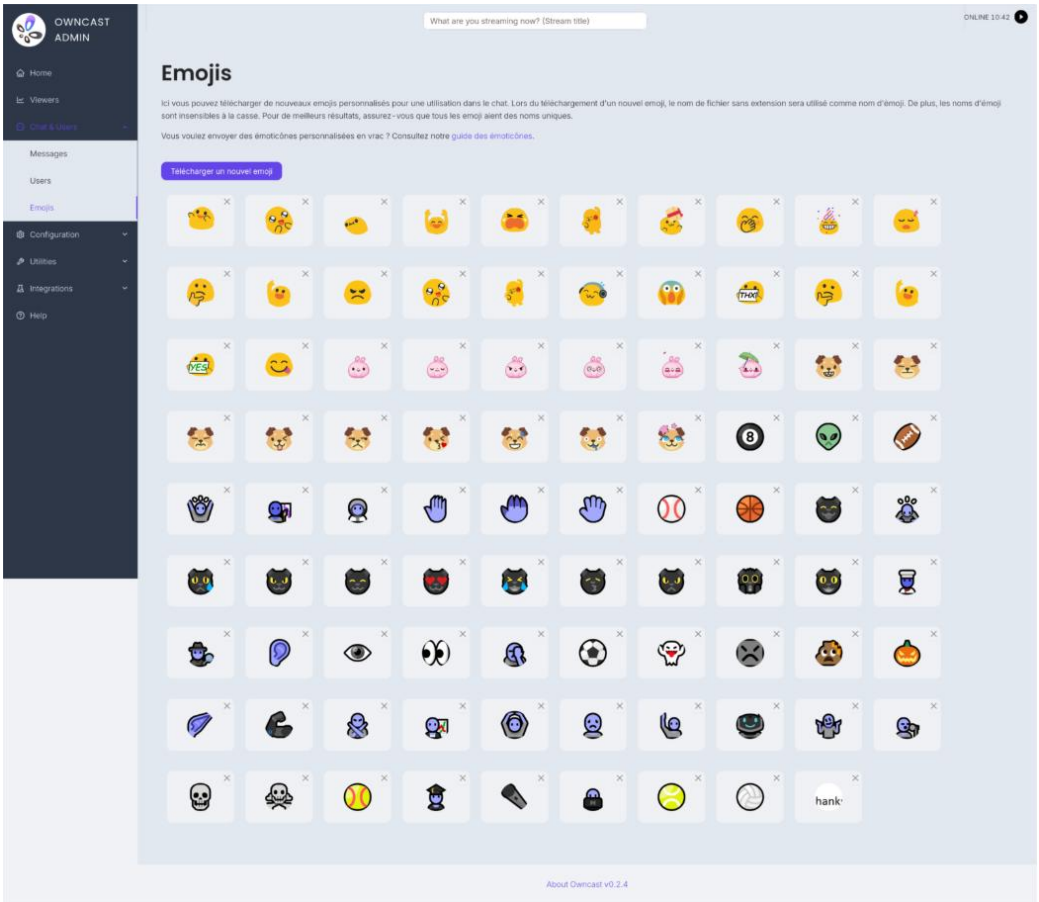


Figure 22 - L'interface admin des Emojis

Nous avons aussi la possibilité d'apporter de modifier le nom du serveur, d'ajouter des logo, et d'ajuster l'apparence du stream:

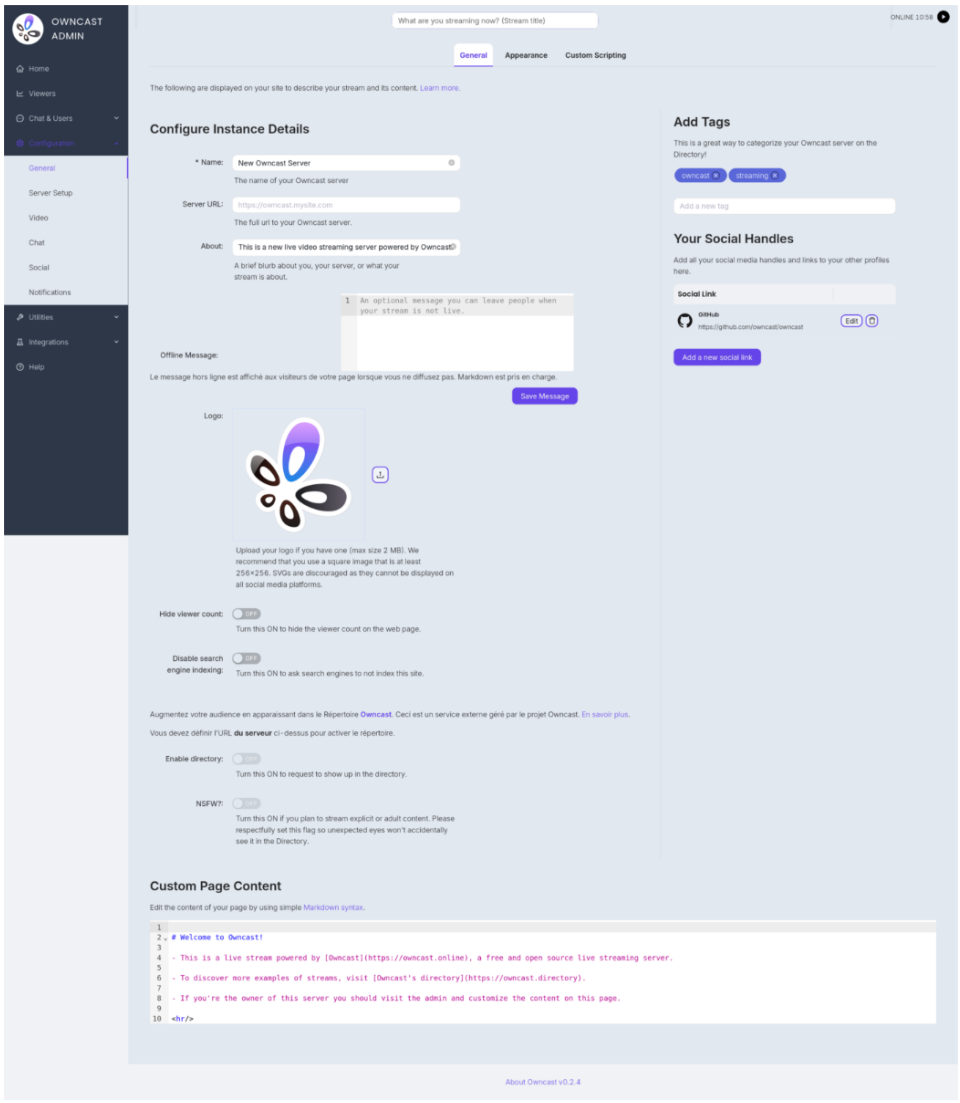


Figure 23 - L'interface Configure Instances Details

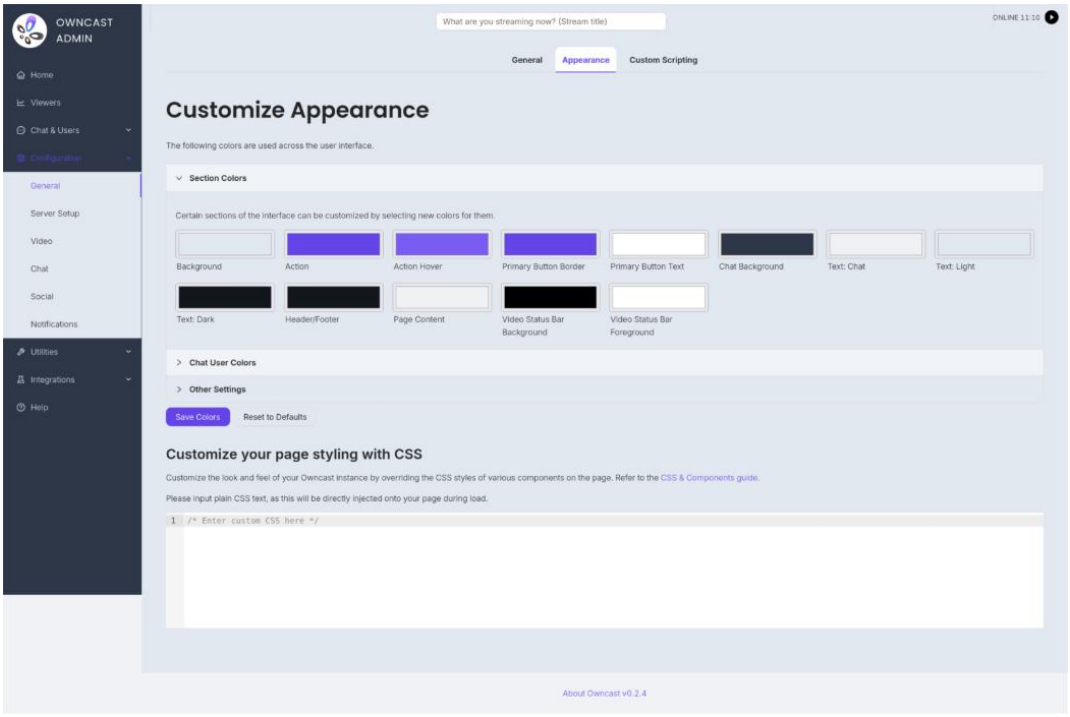


Figure 24 - L'interface Customize Appearance

Sur l'unterface admin, nous configurons serveur en indiquant le chemin vers FFMPEG, le port Owncast et RTMP, directement depuis le menu latéral :

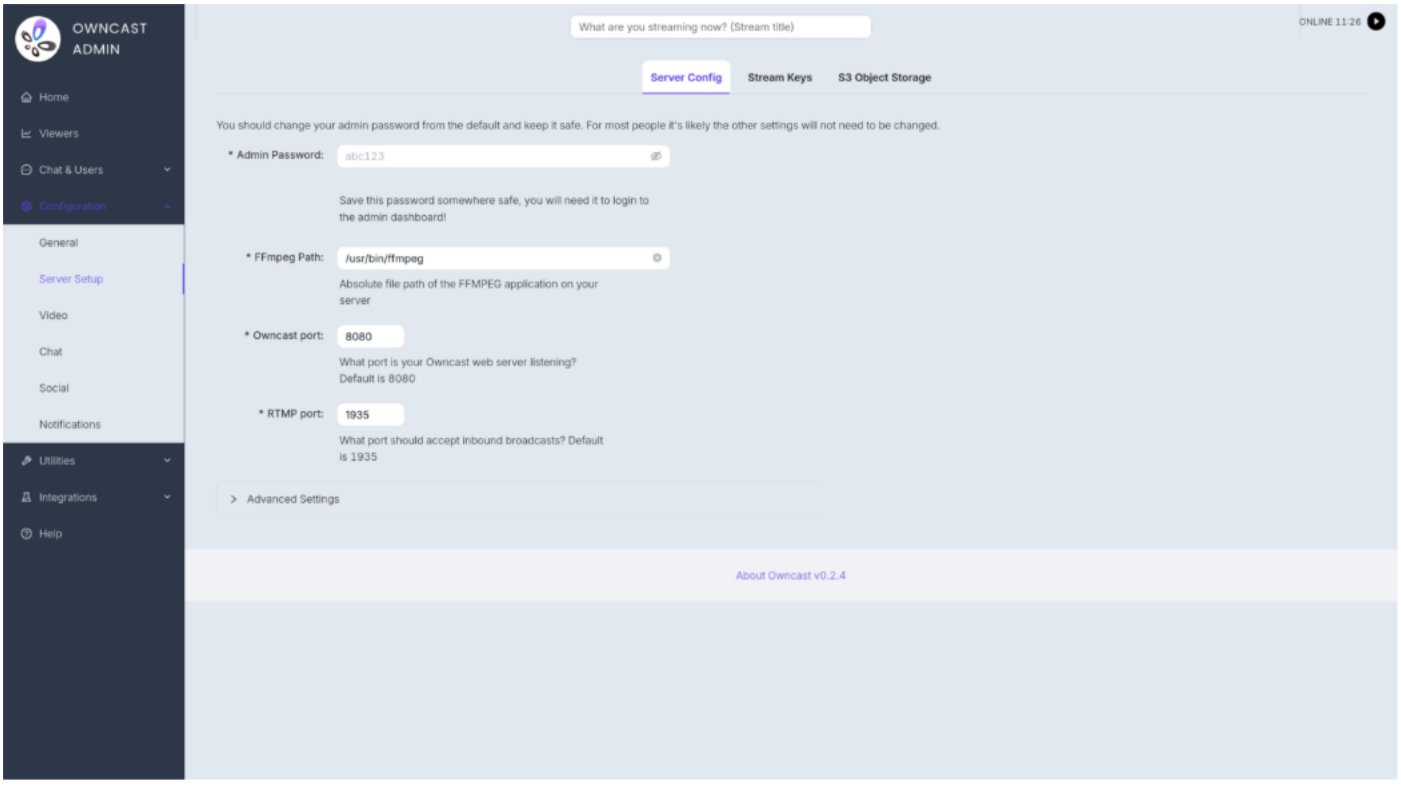


Figure 25 - L'interface de configuration du serveur

Il est nécessaire de configurer la clé du stream dans OwnCast avant la première diffusion afin de sécuriser l'ingestion RTMP et activer le player HLS une fois flux reçu :

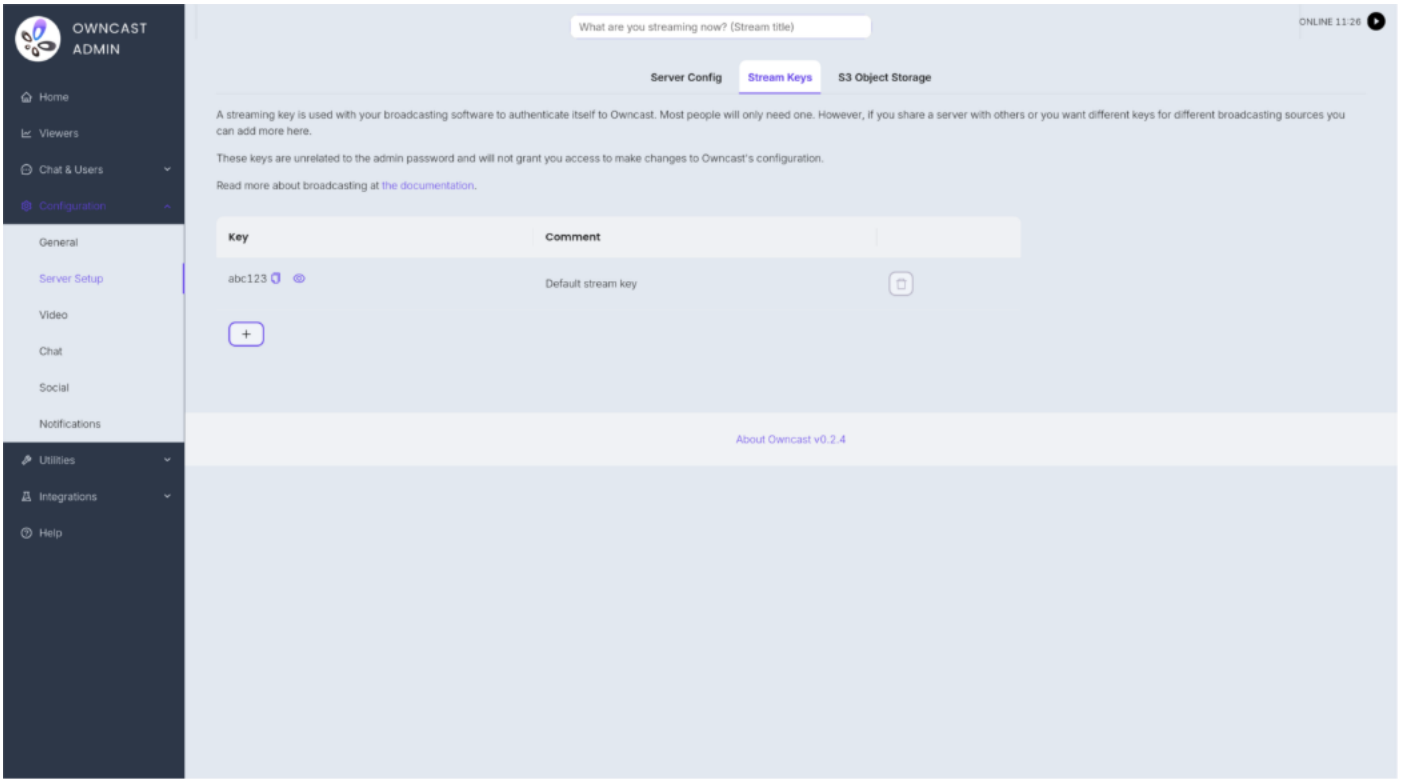


Figure 26 - L'interface Stream Keys

Il est également possible de définir la buffer latence directement depuis l'interface admin. Nous pouvons introduire une buffer latence minimale jusqu'à maximale

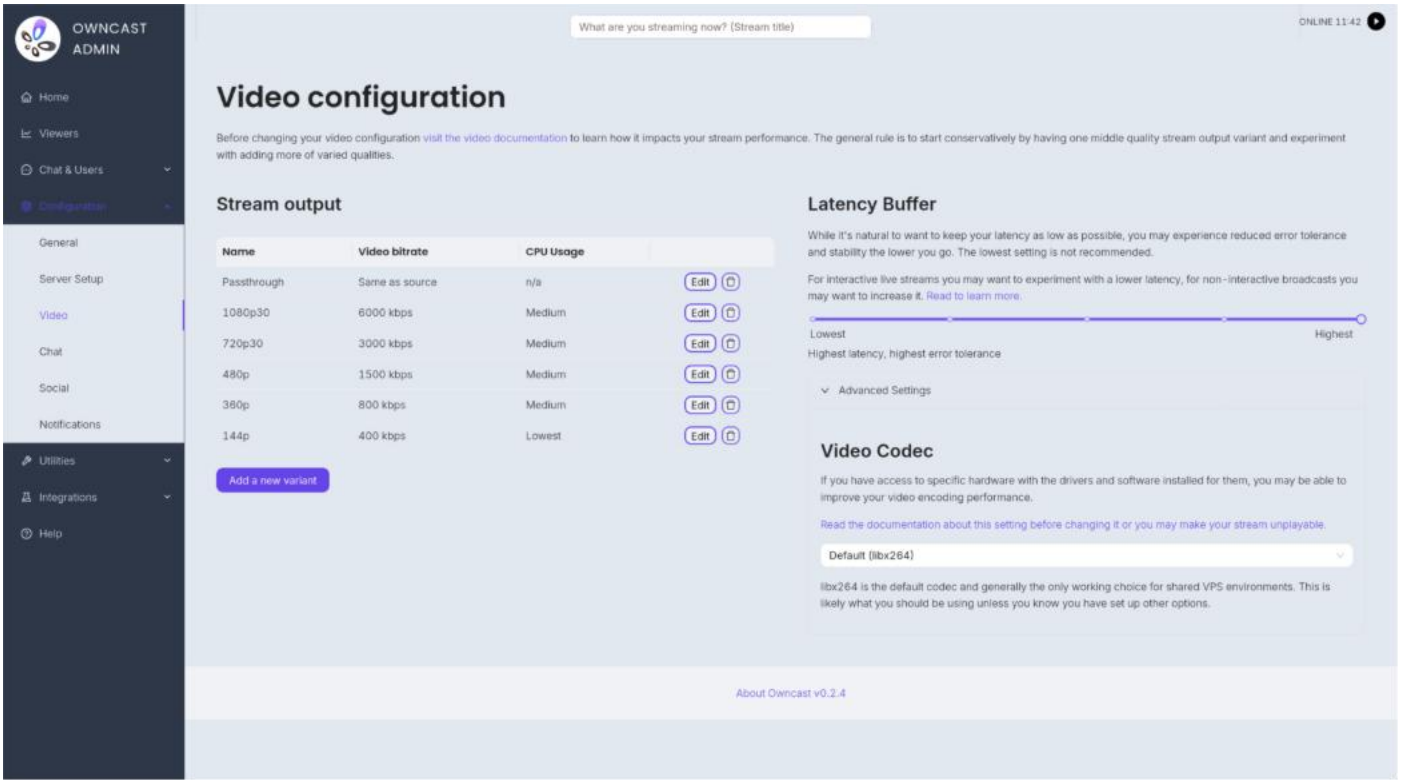


Figure 27 - L'interface Video Configuration

OwnCast nous permet également de surveiller l'activité du CPU et de choisir le débit binaire à envoyer, ainsi que la résolution de la vidéo :

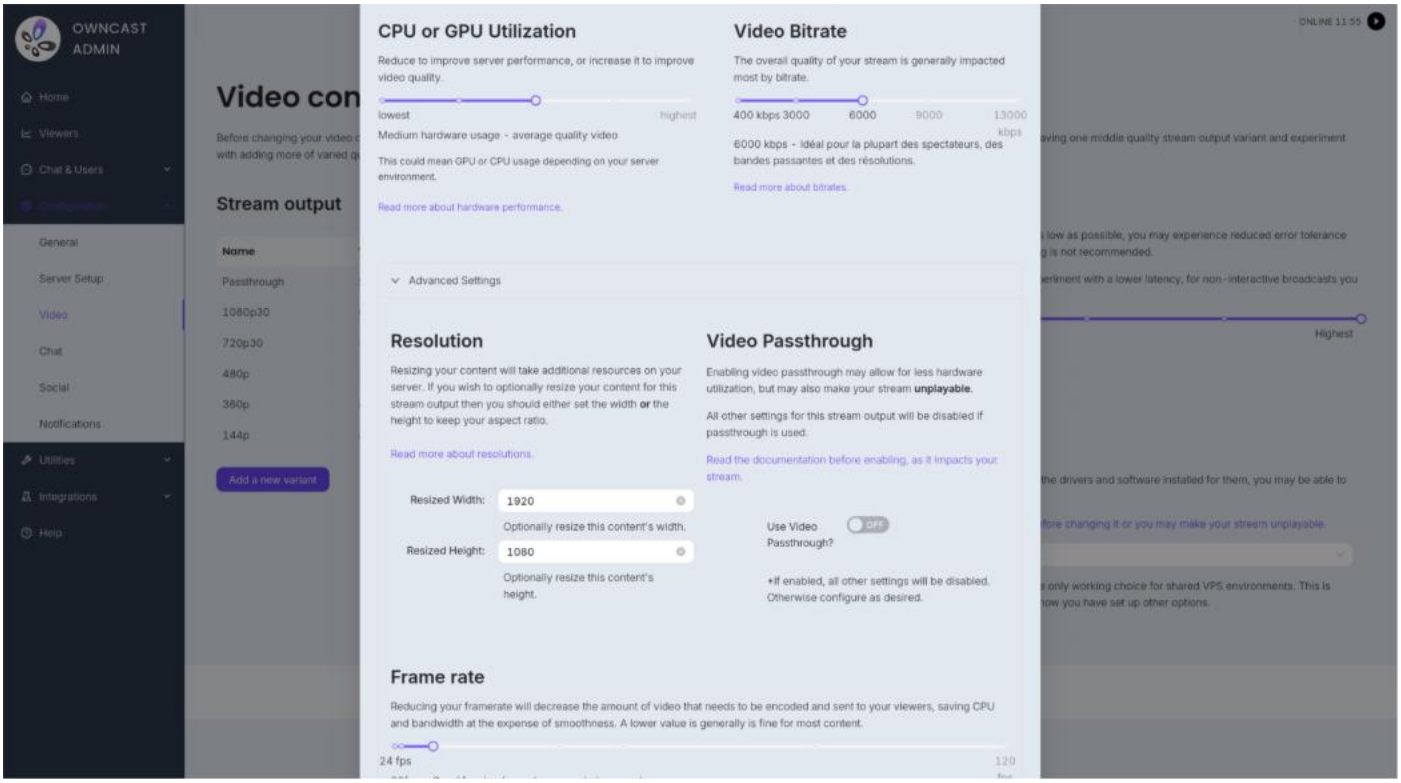


Figure 28 - L'interface de gestion de vidéo

Tout cela peut être suivi via le menu “Stream Health” d’OwnCast :

OWNCAST ADMIN

What are you streaming now? (Stream title)

ONLINE 12:27

- Home
- Viewers
- Chat & Users
- Configuration
- Utilities
  - Hardware
  - Stream Health
  - Logs
- Integrations
- Help

Stream Performance

This tool hopes to help you identify and troubleshoot problems you may be experiencing with your stream. It aims to aggregate experiences across your viewers, meaning one viewer with an exceptionally bad experience may throw off numbers for the whole, especially with a low number of viewers.

The data is only collected by those using the Owncast web interface and is unable to gain insight into external players people may be using such as VLC, MPV, QuickTime, etc.

0

Viewer Playback Speed
0 kbps

Recent Playback Errors
2

Video Segment Download

Once a video segment takes too long to download a viewer will experience buffering. If you see slow downloads you should offer a lower quality for your viewers, or find other ways, possibly an external storage provider, a CDN or a faster network, to improve your stream quality. Increasing your latency buffer can also help for some viewers.

In short, once the pink line consistently gets near the blue line, your stream is likely experiencing problems for viewers.

Player Network Speed

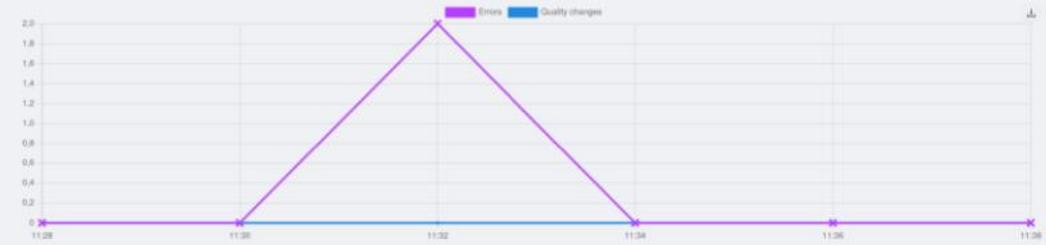
The playback bitrate of your viewers. Once somebody's bitrate drops below the lowest video variant bitrate they will experience buffering. If you see viewers with slow connections trying to play your video you should consider offering an additional, lower quality.

In short, once the pink line gets near the lowest blue line, your stream is likely experiencing problems for at least one of your viewers.

Errors and Quality Changes

Recent number of errors, including buffering, and quality changes from across all your viewers. Errors can occur for many reasons, including browser issues, plugins, wifi problems, and they don't all represent fatal issues or something you have control over.

A quality change is not necessarily a negative thing, but if it's excessive and coinciding with errors you should consider adding another quality variant.



Viewer Latency

An approximate number of seconds that your viewers are behind your live video. The largest cause of latency spikes is buffering. High latency itself is not a problem, and optimizing for low latency can result in buffering, resulting in even higher latency.

About Owncast v0.2.4

Figure 29 - L'interface Stream Performance

Ainsi que les journaux :

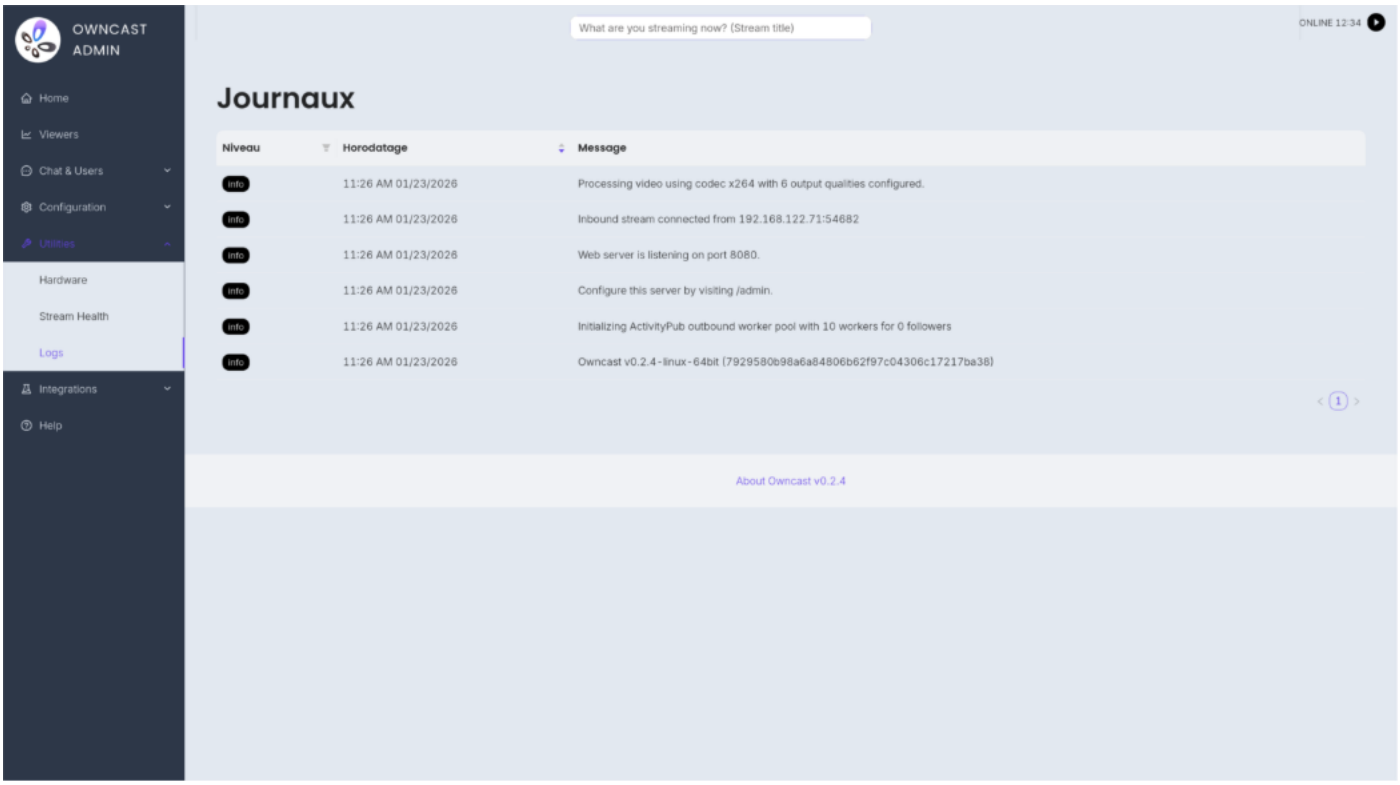


Figure 30 - L'interface des journaux

Une fois les paramètres définis, et le stream en fonction, nous pouvons consulter les métadonnées du stream nous permettant d'avoir des informations sur les méthodes de réception de vidéo (GET), le fichier en lecture ainsi que la taille et l'état :

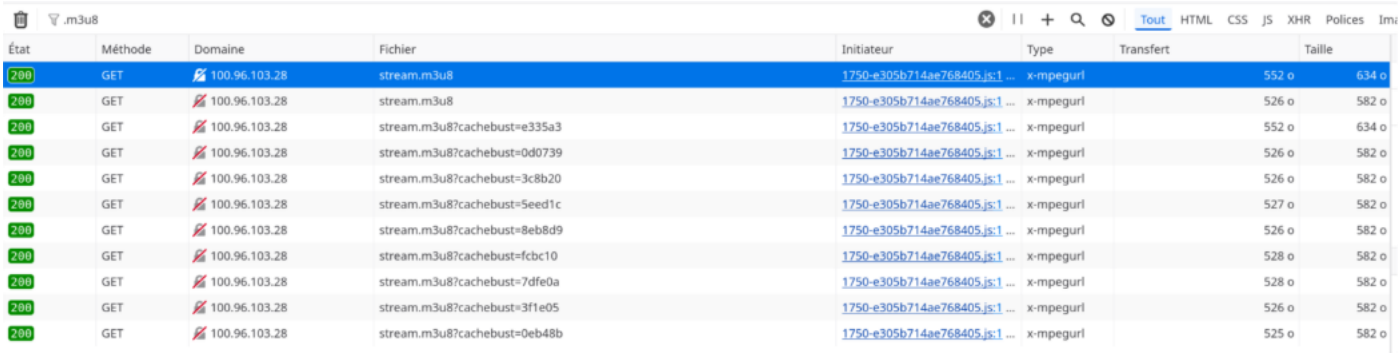


Figure 31 - Métadonnées du Stream

Nous pouvons également voir les métadonnées plus intéressantes telles que l'activité de Newcast Server, comme le montre la capture ci-dessous :

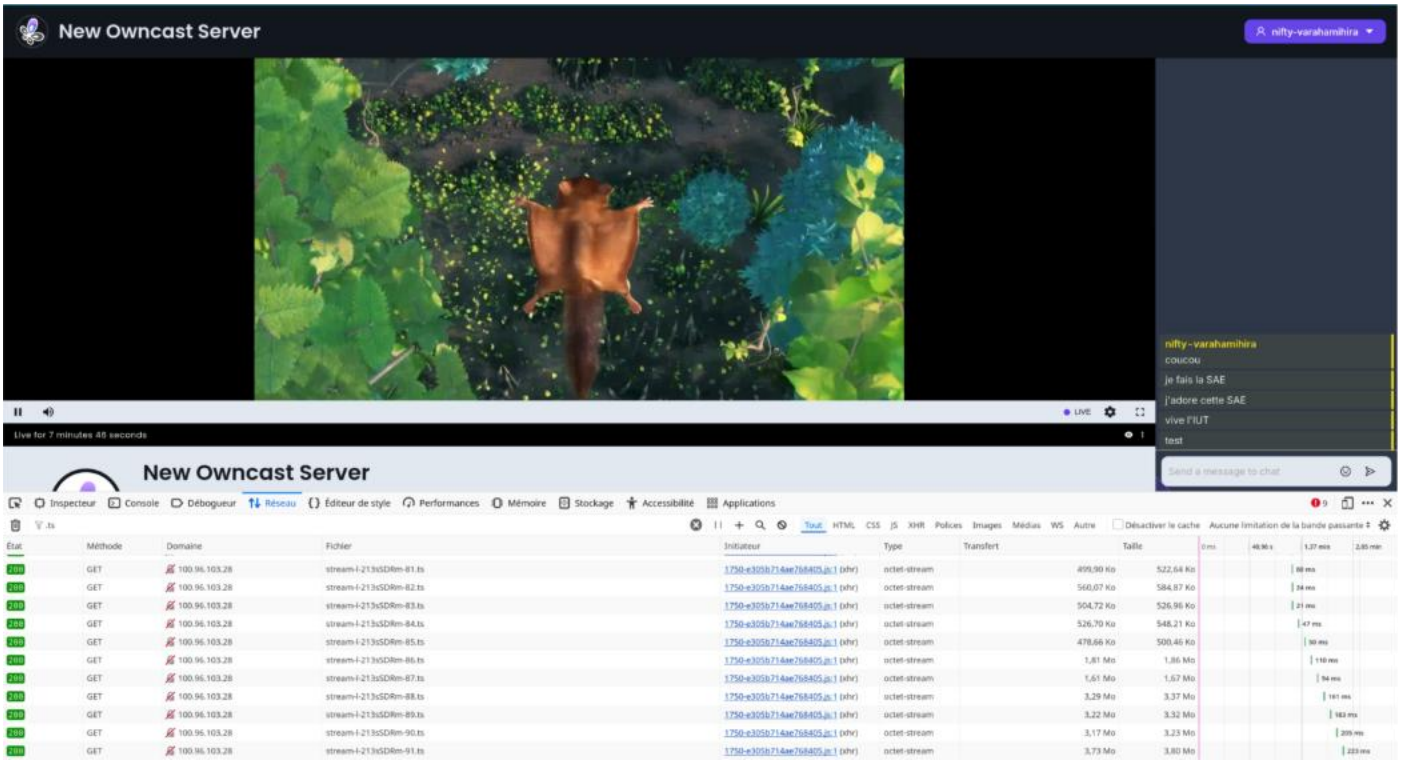


Figure 32 - L'interface complète du stream avec le chat, le stream et les métadonnées réseaux

La méthode de réception de stream est via “GET” de HTTP. La partie de droite montre la lecture réussie de la vidéo en temps réelle.

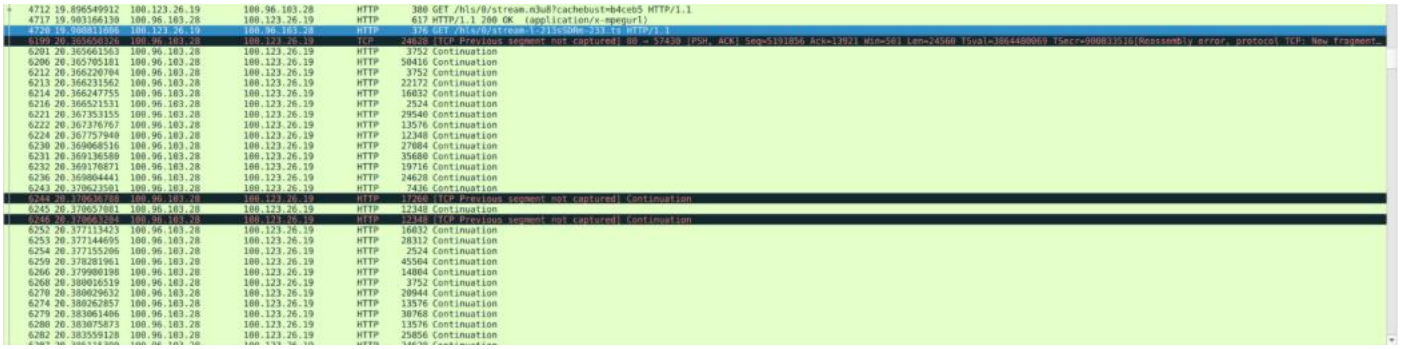


Figure 33 - Capture Wireshark

La capture ci-dessus réalisée durant la diffusion HLS, montre le filtre http révélant les bursts périodiques. Les multiples GET stream.m3u8 (~300 o) suivis de GET segment_X.ts (~1 Mo chacun), montrent la génération de pics débit (colonne "Length" 1400-1500 o, TCP PSH/ACK). Ce pattern est répétitif toutes les deux secondes confirmant la segmentation HLS et l’absence d'erreurs nominales.

Via l'interface GNS3, nous avons injecté des perturbations telle que la latence, la perte de paquets, la corruption en un clic sur le lien 'Diffuseur ↔ Switch', modifiant dynamiquement les caractéristiques du canal virtuel.

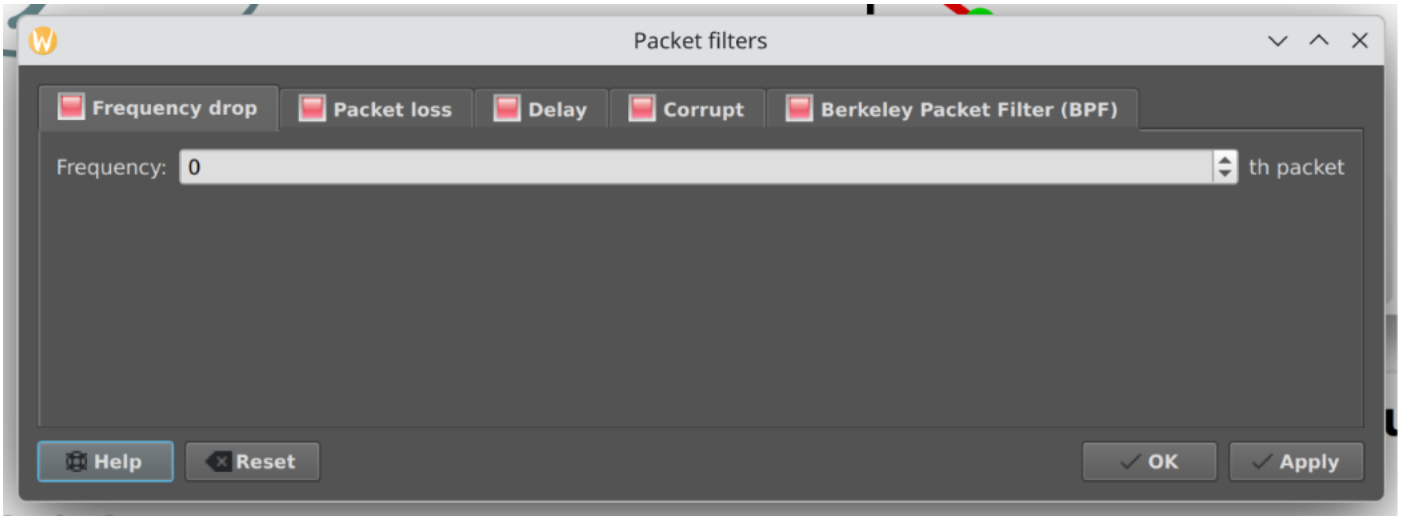


Figure 34 - Packet filters via GNS3

Dans notre cas, cela permet d'observer en temps réel l'impact direct des métriques QoS sur l'expérience utilisateur (QoE). Une latence excessive provoque l'arrêt du flux avec le message "Stream has ended" sur le client, les pertes de paquets entraînent des gels répétés avec icône de chargement, tandis que la corruption génère des artefacts macro-blocs H.264 ou le crash du lecteur. Ces observations concrètes valident la corrélation entre dégradations réseau mesurées et qualité de service perçue.

Chapitre 6 : Conclusion et Retour d'Expérience

La solution de streaming multimédia déployée au cours de cette SAE a pleinement atteint ses objectifs fonctionnels. Nous avons validé l'intégralité de la chaîne de diffusion, depuis l'encodage initial sur OBS jusqu'à la restitution finale en HLS via l'instance OwnCast. L'architecture virtualisée sous GNS3 a non seulement prouvé sa viabilité, mais a également servi de laboratoire d'expérimentation rigoureux. En simulant des dégradations réseau précises (latence, pertes et corruptions de paquets), nous avons pu corréler l'analyse de trafic brute avec l'expérence utilisateur, documentant ainsi des phénomènes critiques tels que le gel d'image (freezing), le buffering excessif et l'apparition d'artéfacts visuels.

Sur le plan pédagogique, ce projet a permis de consolider des compétences concrètes en administration réseau et télécommunications :

Système : Déploiement de serveurs Debian en mode headless et orchestration de services via Docker Compose.

Émulation Réseau : Configuration avancée sous GNS3 exploitant NetEm et uBridge pour la manipulation de flux.

Analyse de données : Dissection protocolaire via Wireshark, ciblant spécifiquement les échanges RTMP/AMF et la segmentation HLS sur TCP.

Les défis techniques, notamment les problématiques de liaisons sous GNS3 et la charge CPU induite par la virtualisation QEMU, ont été surmontés grâce à une optimisation fine des ressources. L'utilisation de distributions Debian minimales, le passage à des conteneurs Docker légers et un monitoring constant via docker stats ont été les clés de la stabilité.

Ce projet nous a permis de voir le rôle fondamental de la Qualité de Service (QoS) dans la diffusion multimédia. Contrairement aux protocoles de type UDP/RTP, particulièrement sensibles aux pertes, le couple HLS/TCP privilégie la résilience via des retransmissions, quitte à augmenter la latence globale. Ce projet démontre qu'une stratégie de QoS proactive, incluant la priorisation du flux RTMP et la mise en œuvre de l'ABR (Adaptive Bitrate), reste indispensable pour garantir un service professionnel stable. Cette expérience renforce ainsi notre maîtrise des stacks open-source au cœur des infrastructures télécoms modernes.

Bibliographie

Documentation Officielle et Manuels

- **Debian Project.** *Debian Administrator's Handbook*. Disponible sur : https://debian-handbook.info/
- **Debian Wiki.** *Network Configuration & System Administration*. Disponible sur : https://wiki.debian.org/fr/NetworkConfiguration
- **Docker Inc.** *Docker Documentation: Get Started with Docker*. Disponible sur : https://docs.docker.com/get-started/
- **GNS3 Technologies.** *GNS3 Official Documentation*. Disponible sur : https://docs.gns3.com/
- **OBS Project.** *Open Broadcaster Software Studio Help Guide*. Disponible sur : https://obsproject.com/help
- **OwnCast.** *OwnCast Documentation: Configuration and Deployment*. Disponible sur : https://owncast.online/docs/
- **Wireshark Foundation.** *Wireshark User's Guide*. Disponible sur : https://www.wireshark.org/docs/wsug_html_chunked/

Normes et Spécifications Techniques (RFC & Standards)

- **Adobe Systems Incorporated.** (2012). *RTMP Specification (Real-Time Messaging Protocol)*. Disponible sur : https://rtmp.veriskope.com/docs/spec/
- **Apple Inc.** *HTTP Live Streaming (HLS) Authoring Specification*. Disponible sur : https://developer.apple.com/documentation/http-live-streaming
- **Fielding, R., & Reschke, J.** (2014). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing (RFC 7230)*. Internet Engineering Task Force.
- **ITU-T.** (2019). *H.264 : Advanced video coding for generic audiovisual services*. International Telecommunication Union.
- **Pantos, R.** (2017). *HTTP Live Streaming (RFC 8216)*. Internet Engineering Task Force.

Articles Techniques et Ressources Pédagogiques

- **EBU (European Broadcasting Union).** (2003). *The emerging H.264/AVC standard*. EBU Technical Review.
- **VideoLAN Organization.** *VideoLAN Streaming Features*. Disponible sur : https://www.videolan.org/vlc/streaming.html
- **Vcodex.** *An Overview of H.264 Advanced Video Coding*. Disponible sur : https://www.vcodex.com/an-overview-of-h264-advanced-video-coding/

Implementation and Analysis of a Virtualized Multimedia Streaming Infrastructure

1. Introduction and Context

In the current digital landscape, multimedia streaming has established itself as the dominant form of traffic on the global internet. From on-demand entertainment platforms to real-time videoconferencing and live event broadcasting, video data transmission represents a critical challenge for network infrastructures. For network engineers and administrators, understanding the underlying mechanisms of these flows is no longer optional but a fundamental skill. Video traffic is resource-intensive and highly sensitive to network quality metrics such as latency, jitter, and packet loss.

The primary objective of this project (SAÉ) was to demystify the streaming delivery chain by building a complete, production-grade infrastructure from scratch. Rather than relying on pre-packaged commercial solutions, the aim was to deploy an open-source architecture capable of capturing, encoding, transporting, and distributing a live video stream.

Furthermore, the project moved beyond simple deployment to focus on **Network Metrology** and **Quality of Service (QoS)** analysis. By utilizing a virtualized environment, we aimed to subject our streaming architecture to controlled network degradation. The goal was to observe the precise correlation between network layer faults (Layer 3/4) and the user experience (Application Layer), thereby understanding the resilience and limitations of modern streaming protocols.

2. Technical Architecture and Tools

To ensure reproducibility and granular control over the network topology, the entire infrastructure was virtualized using **GNS3 (Graphical Network Simulator-3)**. This approach allowed for the emulation of physical network equipment and the integration of real virtual machines, creating a "sandbox" environment isolated from production networks.

2.1. System and Virtualization

The core computing nodes were powered by **Debian GNU/Linux 13 ("Trixie")**. This distribution was selected for its stability and low resource footprint. In the GNS3 environment, where hardware resources are shared, running the servers in "headless" mode (without a graphical user interface) was crucial to maximize performance and allow for multiple active nodes simultaneously.

For the application layer, we leveraged **Docker** containerization. specifically for the streaming server. Docker allowed us to deploy the application with all its dependencies in an isolated, lightweight container, ensuring that the service would run identically on the simulation machine as it would on a production server. Orchestration was managed via **Docker Compose**, providing a declarative configuration method that simplified deployment and maintenance.

2.2. The Media Processing Chain

The architecture was designed around a standard source-server-client topology:

- **Acquisition and Encoding (The Source):** We utilized **OBS Studio (Open Broadcaster Software)** as the ingestion client. To ensure scientific validity during testing, the encoder was configured with strict parameters:
    - **Codec:** H.264 (AVC) for video and AAC for audio.
    - **Rate Control: CBR (Constant Bitrate)** was enforced to generate a predictable network load, facilitating the identification of bottlenecks.
    - **Keyframe Interval:** Strictly set to **2 seconds**. This setting is critical for the downstream HLS segmentation process, ensuring that the server can cut the video stream cleanly at regular intervals.
- **Streaming Server (The Core):** The central node was **OwnCast**, an open-source, self-hosted streaming server written in Go. OwnCast was chosen over alternatives like VLC or Wowza because of its modern handling of web standards. It acts as a **transmuxer**: it receives the raw RTMP stream from OBS and converts it in real-time into HLS (HTTP Live Streaming) segments for web delivery.
- **Protocols:**
    - **Ingestion:** The **RTMP (Real-Time Messaging Protocol)** was used between OBS and OwnCast over TCP port 1935. RTMP is the industry standard for low-latency ingestion.
    - **Diffusion:** The **HLS protocol** was used for delivery to the client. This breaks the video into small .ts (MPEG-TS) files and indexes them in a .m3u8 manifest file, delivered via standard HTTP (TCP port 8080).

3. Methodology and Network Topologies

The experimental phase was conducted within GNS3. The topology consisted of the Debian server (hosting OBS), a virtual Ethernet switch, and the Debian diffusion server (hosting OwnCast/Docker), all interconnected via simulated Ethernet links.

A crucial aspect of the methodology was the use of **Wireshark** for deep packet inspection. By connecting Wireshark to the virtual links via uBridge, we were able to capture traffic in promiscuous mode. This allowed us to dissect the protocol headers and analyze the behavior of the TCP stack under stress.

To simulate real-world internet conditions, we utilized the network impairment features within GNS3 (based on the Linux kernel's **NetEm** module). We created specific test scenarios by injecting:

- **Latency:** Adding delay (e.g., 100ms - 200ms) to simulate satellite or transcontinental links.

- **Packet Loss:** Dropping a percentage of packets (1% to 5%) to simulate congested Wi-Fi or unreliable physical connections.
- **Jitter:** Varying the delay to test the buffering capabilities of the client.

4. Analysis of Results

The experiments yielded significant insights into the behavior of streaming protocols and their interaction with the TCP transport layer.

4.1. Protocol Analysis

Wireshark captures confirmed the distinct phases of the transmission. On the ingestion side (RTMP), we observed the specific handshake sequence (C0/C1/C2 and S0/S1/S2 packets) followed by the transmission of AMF (Action Message Format) commands such as connect and publish. The data stream was observed as a continuous flow of multiplexed audio and video chunks.

On the diffusion side (HLS), the traffic pattern was notably different. Instead of a smooth continuous flow, we observed "bursts" of TCP traffic occurring every few seconds. These bursts correspond to the client requesting the next .ts video segment. This confirms that HLS transforms a stream into a series of discrete file downloads.

4.2. Impact of Network Degradation

The stress tests highlighted the "reliable but slow" nature of using TCP for video streaming.

- **Latency Impact:** When latency was introduced, the initial start-up time (Time to First Byte) increased. However, once the buffer was filled, the stream remained relatively stable, provided the bandwidth was sufficient.
- **Packet Loss Impact:** This was the most critical factor. Because HLS relies on TCP, every lost packet must be retransmitted. In our Wireshark analysis, we observed a spike in **TCP Retransmissions** and **Duplicate ACKs** when packet loss was set to 5%.
    - *Observation:* Unlike UDP-based protocols (RTP) where a lost packet results in a visual artifact (glitch) but the stream continues, TCP halts the delivery of data to the application until the missing packet is successfully retransmitted.
    - *Consequence:* This mechanism preserves the integrity of the image (no visual corruption) but causes the player to empty its buffer, leading to the "buffering" wheel and playback freezes. This demonstrates a trade-off: **Image Quality vs. Latency/Continuity.**

5. Conclusion and Recommendations

This project successfully demonstrated the deployment of a functional, professional-grade streaming architecture using entirely open-source tools. The use of GNS3 and Docker allowed for a highly controlled environment where theoretical networking concepts could be visualized in practice.

The results confirm that while modern protocols like HLS provide excellent compatibility and firewall traversal (being HTTP-based), they are heavily reliant on a stable TCP connection. The analysis showed that even minor packet loss rates can drastically degrade the Quality of Experience (QoE) due to the mechanics of TCP congestion control and retransmission.

Future Improvements

Based on our findings, the resilience of this system could be improved through **Adaptive Bitrate Streaming (ABR)**. By configuring OwnCast to generate multiple quality tiers (e.g., 1080p, 720p, 480p), the client could dynamically switch to a lower bitrate stream when network congestion is detected. This would prevent buffering pauses by sacrificing resolution rather than continuity, which is generally preferred in live streaming scenarios.

In conclusion, this project bridged the gap between systems administration (Linux/Docker), network engineering (GNS3/Routing), and multimedia technology, providing a comprehensive understanding of the challenges involved in modern content delivery.