Travail de Bachelor – RTS Express Live
Rapport intermédiaire

Auteur : Sacha Bron
Superviseurs :
Olivier Liechti – Professeur à l'HEIG-VD
Sébastien Noir – Chef de projet à la RTS

Résumé

RTS Express Live est une application mobile permettant aux journalistes de diffuser en direct des images issues de leurs téléphones. Elle a été développée pour la RTS (Radio Télévision Suisse) dans le but d'ajouter cette fonctionnalité à la gamme RTS Express déjà existante.

Table des matières

Chapitre 1

Introduction

Dans le monde d'aujourd'hui, l'information instantanée prend de plus en plus d'ampleur à travers divers vecteurs. Les différents réseaux sociaux soutiennent ce mouvement en incitant son public à partager du texte, des clichés ou encore de la vidéo. En effet, de nombreuses applications et sites web voient le jour chaque année. Parmi eux, les plus connus sont Twitch.tv : un site web de diffusion vidéo en direct utilisé surtout par les amateurs de jeu vidéo, Periscope : une application mobile permettant la diffusion en direct depuis la caméra du smartphone vers d'autres smartphones à travers le monde, et la majorité des grands acteurs du web ont ajouté cette fonctionnalité à leurs sites web : YouTube, Facebook, Snapchat, etc.

Ce phénomène contraint les médias "classiques" à continuer à s'adapter aux nouvelles technologies de communication. La diffusion d'information en direct prend alors une place de plus en plus importante afin de d'atteindre ce nouveau public friand d'instantané. La RTS (Radio Télévision Suisse) fait partie de ces médias dont la popularité est en jeu. Il n'est donc pas étonnant qu'ils souhaitent aussi participer à l'expension de cette nouvelle manière de partager l'information. Bien évidemment, les émissions et retransmissions d'évenements en direct n'est pas chose nouvelle pour une chaîne de télévision mais la différence réside aussi dans la manière de procéder. En effet, en ayant une application de diffusion en direct sur leurs téléphones, les journalistes ont la libérté de créer du contenu à tout moment et sans préparation. Cela peut être très pratique pour émettre les images d'un festival de musique, d'un incendie qui vient de débuter ou encore d'interviewer une célébrité que l'on croiserait par hasard.

C'est pourquoi la RTS a proposé à l'HEIG-VD un travail de Bachelor ayant pour sujet la création d'une application pour iPhone permettant la diffusion en direct des images perçue par sa caméra, envoloppé dans une interface donnant la posibilité aux journalistes de s'authentifier, ainsi que d'ajouter des informations utiles concernant

la capture. Ce travail a alors été effectué en collaboration avec la RTS qui a, par exemple, fourni le cahier des charges.

Ce projet est constitué de trois phases :

- une phase de recherche comprenant les différents protocoles de diffusion en direct ainsi que leurs formats audio et vidéo, les applications déjà existantes, des bibliothèques qui peuvent être utile au développement d'un tel projet, des possibilités qu'offrent iOS, etc.
- une phase de création de prototypes, permettant de vérifier la faisabilité du projets et l'exploration approfondie des bibliothèques susmentionnées. Cette phase m'a aussi permis de me familiariser avec le développement sur iOS et tout ce que cela comprend (apprentissage du langage de programmation Swift, maitrise du logiciel Xcode, etc.).
- une phase de création de l'application finale, reprenant les concepts et algorithmes des prototypes. C'est cette application qui sera livrée à la RTS.

Ce rapport explique en détail le travail qui a été effectué durant ces différentes phases, les problèmes qui sont apparus au cours du développement et les solutions qui ont été employées afin d'y remédier.
Ainsi, la première partie explique en détails les buts et objectifs de ce travail selon les contraintes données par la RTS.
La seconde partie expose les différentes technologies existantes et explore les avantages et inconvéniants de chacune des méthodes de diffusion.
La troisième partie documente le travail effectué sur l'application finale ainsi que son fonctionnement.

Chapitre 2

État de l'art ("vite survolé", requiert plus de détail)

Cette section présente divers aspects factuels sur la diffusion de la vidéo en direct et du développement sur iOS.

2.1 Introduction sur les fichiers vidéo

TODO : - conteneur - encodeur - débit binaire - Keyframes

2.2 Introduction au diffusion vidéo en direct

La diffusion de vidéo en direct reste quelque chose de techniquement difficile à réaliser pour plusieurs raisons.

Tout d'abord, les fichiers vidéo comportent beaucoup de données et sont donc rapidement volumineux. Cela entraîne alors d'autres problèmes : le réseau de communication utilisé doit être capable de gérer un débit minimal afin que le temps de transmission des fichiers vidéos ne soit pas plus long que la durée de la vidéo envoyée.

Pour ce travail, nous nous concentrerons uniquement sur les systèmes de streaming vidéo de type client-serveur dans lesquels la latence n'est pas une priorité majeure et non les protocoles peer-to-peer et temps réel utilisés, par exemple, pour la vidéoconférence.

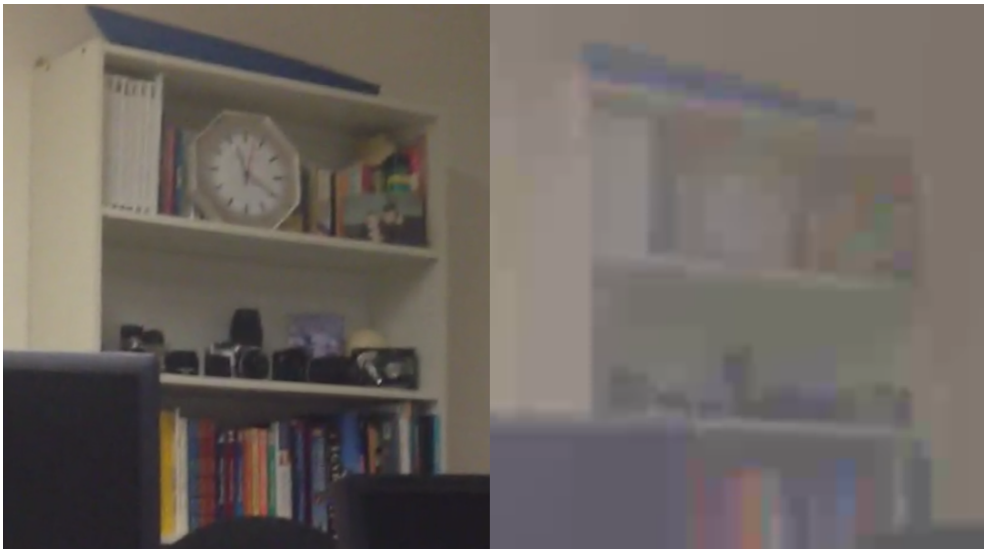


FIG. 2.1 : Exemple d'une image de vidéo peu compressée (débit binaire : 3'394 kb/s) par rapport à une image fortement compressée (débit binaire : 470 kb/s)

2.2.1 Utilisation d'HTTP

Les protocoles de streaming vidéo les plus répandus sont sans doute ceux basés sur HTTP.

L'avantage principal d'utiliser HTTP réside dans le fait que c'est un protocole de la couche applicative du modèle OSI. Ce niveau supplémantaire d'abstraction par rapport aux couches plus basses du modèle OSI signifie que le flux vidéo peut passer par des proxys HTTP et passer à travers les les pare-feux permissif à HTTP, contrairement à TCP ou UDP qui se situe sur des couches plus basses du modèle OSI, par exemple.

En outre, ses inconvéniants majeurs sont les contraintes qu'ils présentent quant à la manière de gérer les données. Par exemple, un protocole de diffusion de vidéo basé sur UDP pourraient simplement envoyer le flux de bytes de la vidéo à travers le réseau avec peu de traitements sur les données. Par contre, avec l'utilisation d'HTTP, les données doivent être contenues dans les requêtes ou les réponses, ou dans des fichiers (car HTTP requiert des en-têtes relativement lourd et envoyer seulement quelques bytes de cette manière serait inefficace). La vidéo doit alors être découpée en une mulitude de fichiers vidéo, aussi appelés segments, qui seront envoyés un par un sur le réseau. Cette contraite nous oppose à plusieurs choix, dont celui de la durée de ces segments de vidéo. En effet, si les segments sont trop longs,

la latence entre la capture de l'image et sa récéption va fortement augmenter car le smartphone devra attendre que le segment soit complet avant de l'envoyer, et son visionage ne pourra commencer qu'à ce moment-là.

2.2.2 RTMP

RTMP (Real-Time Messaging Protocol) est un protocole de communication permettant le streaming de vidéo. Il a été développé par Macromedia (aujourd'hui Adobe) et se base sur un client en Flash. Il est basé sur HTTP et utilise des vidéos au format FLV (Flash Video) et de l'audio en MP3 ou AAC.

Les principaux inconvéniants de ce protocole est qu'il est très lié aux clients Flash et que ces derniers sont aujourd'hui de plus en plus remplacés par les technologies HTML5. De plus, même si ce protocole est largement documenté, il est propriétaire.

RTMP est aujourd'hui utilisé par beaucoup de plateforme de streaming vidéo, comme Twitch.tv ou encore Periscope.

2.2.3 HLS

HLS (HTTP Live Streaming) est un système de streaming vidéo développé par Apple et basé sur HTTP. Il utilise un conteneur MPEG-2 TS et le codec H.264 pour ses segments vidéo et supporte le MP3 et le AAC pour le transport du son.

Une liste de lecture dynamique, aussi appelée manifeste, est nécessaire afin d'indiquer au client à quelle adresse il doit aller chercher les prochains segments vidéo. Ce manifeste est au format M3U8. Il doit aussi préciser la durée de chaque segment ainsi que le type de flux média qu'il représente. Il doit aussi indiquer si le flux est fini ou si d'autres segments sont susceptibles d'être ajoutés.

Les manifestes M3U8 peuvent en réalité représenter divers types d'utilisation des flux média. Dans notre cas, deux types de représentation nous intéressent : les "fenêtres glissantes" (*sliding window playlist*) et "événements" (*event playlist*). Toutes deux représentent une diffusion en direct. Cepandant, les listes de lecture à fenêtres glissantes permettent de faire des diffusions en live sur de très longues durées, car la liste de lecture ne présentent que des liens vers les derniers segments vidéos. Les listes de lecture de type événements, quant à elles, ont été créées plus particulièrement pour des événements à durée définie. Leur avantage principal est de permettre à l'utilisateur de lire le flux depuis le début.

Au final, la solution à fenêtres glissantes est intéressante car elle nous permet de choisir la taille de la fenêtre, et même de lui donner une taille infinie, permettant alors de simuler un flux de type "événement".

Afin de créer un flux vidéo au débit adaptatif, il est possible de préciser le débit binaire des segments vidéo. Le logiciel client pourra alors choisir les segments qui correspondent le mieux à sa bande passante disponible.

2.2.4 MPEG-DASH

MPEG-DASH (MPEG pour Moving Picture Experts Group et et DASH pour Dynamic Adaptive Streaming over HTTP) est un système de streaming vidéo, également basé sur HTTP. Contrairement à RTMP et à HLS, MPEG-DASH ne dépend d'aucun codec. De plus, il supporte plusieurs format de conteneur. Il est alors capable d'envoyer des vidéos en MPEG-2 TS (comme HLS) ainsi que du MPEG-4 et les formats similaires.

Ce protocole nécessite d'avoir des segments vidéo ne contenant que les données de la vidéo (et non les en-têtes). Il faut alors stocker les en-têtes dans un fichier séparé.

MPEG-DASH se repose aussi sur l'utilisation d'un manifeste contenant des liens vers les segments vidéos ainsi que des meta-données. Ces fichiers sont en XML.

Support des différentes technologies selon les navigateurs[1] :

Browser	DASH	HLS	Opus (Audio)
Firefox 32	Oui	Oui	v14+
Safari 6+		Oui	
Chrome 24+	Oui	Oui	
Opera 20+	Oui		
Internet Explorer 10+	v11	Oui	
Firefox Mobile	Oui	Oui	Oui
Safari iOS6+		Oui	
Chrome Mobile	Oui	Oui	
Opera Mobile	Oui	Oui	
Internet Explorer Mobile	v11	Oui	
Android	Oui		

2.3 Introduction au développement sur iOS

Le développement sur iOS se fait par le biais de XCode, l'IDE d'Apple constituant l'unique environnement de développement officiel pour le développement

[1] Source : Mozilla

d'application. C'est un IDE très complet offrant des outils de création d'interfaces, de gestion de projet, de diagnostique, ainsi qu'un simulateur d'iPhone permettant de tester la majorité des applications.

2.3.1 Swift

Swift est un langage de programmation conçu et maintenu par Apple. Sorti il y a environ 2 ans, il est en train de remplacer l'utilisation d'Objective-C pour la programmation sur iOS. C'est un langage moderne à la syntaxe épurée dont on retrouve les concepts dans certains autres langages modernes tel que Scala, C# ou encore Rust. Comme ces derniers, il est multi-paradigm, et permet donc de faire de la programmation orientée objet, de la programmation fonctionelle ou encore impérative.

2.3.2 Utilisation de bibliothèque Objective-C

Un des inconvéniants lors du changement de langage de programmation pour une plateforme est le portage de toutes les bibliothèques écrites dans le langage précédant. Pour le développement sur iOS, ce problème a été contourné de manière intelligente : les bibliothèques écrites en Objective-C peuvent être utilisée telle quelle par l'application, à condition de fournir un petit fichier appelé "Bridging Header". Ce fichier permet à Swift de savoir quel type de fonction, d'objet et de classe il peut utiliser. Cela permet donc d'appeler des bibliothèques Objective-C depuis du Swift et éviter de devoir porter de grandes quantités de code.

Un des problèmes de cette méthode est que nous devrions appeler les fonctions Objective-C en leur passant des types Objective-C depuis Swift. Il pourrait être alors problématique de convertir les variables typées en Swift en types Objective-C. Heureusement, Swift est capable de convertir implicitement les types les plus communs. Par exemple, une `String` pourra être implicitement convertie en `NSString` pour que le code en Objective-C puisse l'interpréter correctement.

2.3.3 Utilisation de bibliothèques externes

L'utilisation de bibliothèques externes est légèrement plus complexe. En effet, il est d'abord nécessaire de *cross-compilé* les bibliothèques pour supporter les processeurs des différentes versions de l'iPhone (arm64, armv7, armv7s). Il peut alors être pratique d'utiliser la commande `lipo` disponible sous Mac OS X afin de fusionner les multiples compilations de la bibliothèque en un seul fichier.

Une fois la bibliothèque compilée pour l'iPhone, nous pouvons l'inclure dans XCode et également inclure les en-têtes de la bibliothèque. Ces en-têtes peuvent alors être ajouté au Bridging Header pour être appelé par Swift.

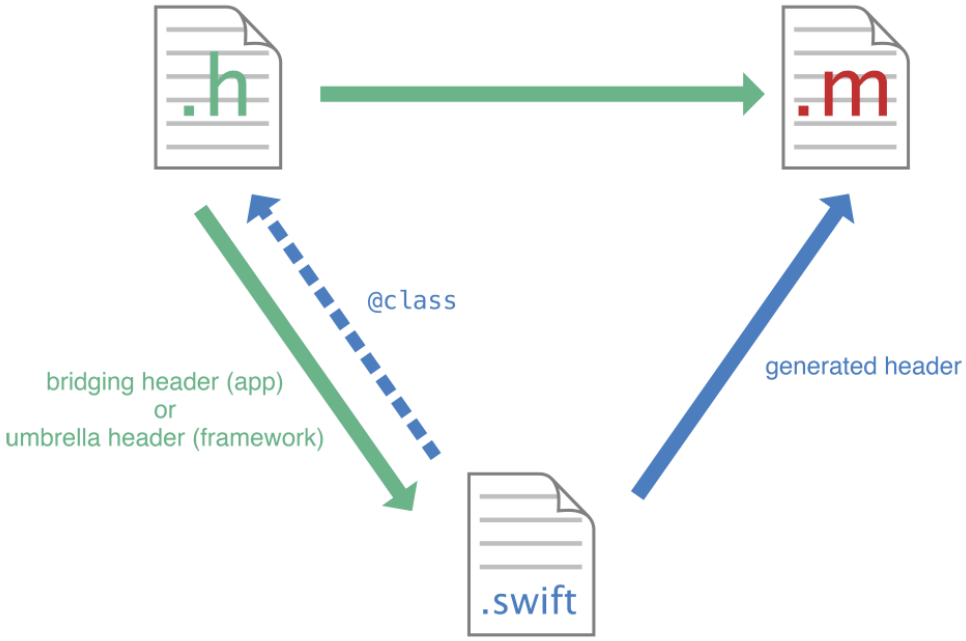


FIG. 2.2 : Schéma des liens possibles entre Objective-C (.m et .h) et Swift

Au niveau des types de variables, Swift est aussi capable de comprendre et convertir les variables issues de C, par exemple.

Chapitre 3

Approche du problème

La première partie du projet a consisté en la familiarisation avec les technologies susmentionnées, effectuer des recherches sur l'état de l'art ainsi que sur le futur de ces différents systèmes. En effet, ce genre d'application étant relativement moderne, il est utile d'essayer de prédire comment vont se développer les technologies de diffusion vidéo ainsi que leur support. Par exemple, l'abandon progressif des technologies Flash au profit d'HTML5 par les navigateurs constitue un paramètre non négligeable dans la création d'application de demain.

Une fois documenté, une phase de recherche de bibliothèques et de frameworks libres ou gratuits permettant de mettre en place un système de streaming depuis l'iPhone a débuter. Malheureusement, la grande majorité de ses systèmes sont payants, très souvent par mois, et incluent l'entier de l'écosystème (application(s), serveurs cloud, parfois même client(s)).

Il est intéressant de noter que les possiblité qu'offre AVFoundation, un framework d'audio-visuel fourni par Apple sont relativement faibles par rapport à l'utilisation particulière de la vidéo pour ce projet.

Cepandant, les bibliothèques fournies par FFmpeg pour les transformations sur la vidéo (débit binaire, encodage, conteneur, fusion) car c'est une bibliothèque très largement utilisée, aussi par les grandes entreprises, qu'elle permet de faire toute les opérations désirées. En outre, le projet actif depuis plus de 15 ans, ce qui est signe d'une grande stabilité et qu'il y a beaucoup de chance qu'elle continue à être supportée dans les années à venir.

Une fois renseigné sur les technologies existantes, une phase d'apprentissage de l'environnement de programmation sur iOS ainsi que du langage Swift a été nécessaire. Le développement de petites applications simples pour mieux comprendre comment utiliser les divers outils offert par Apple a été très utile. Il m'a aussi permis de comprendre certaines subtilité de Swift.

Par la suite, les premiers prototypes d'application permettant de tester les technologies de capture, conversion, et diffusion vidéo ont été créés.

3.1 Capture vidéo

La première partie à avoir été testée est la capture de la vidéo à l'aide de la caméra du téléphone ainsi que son affichage à l'écran.

La capture vidéo nécessite l'utilisation du framework AVFoundation. Il permet, entre autres, une gestion avancée de la caméra (résolution, balance des blancs, ISO, etc.).

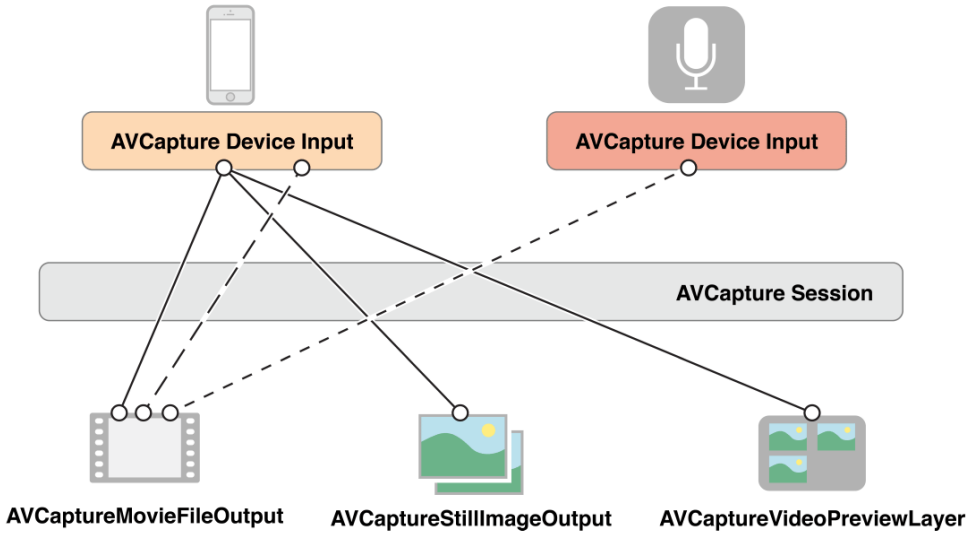


FIG. 3.1 : Fonctionnement de la caputre vidéo avec AVFoundation

Les fichiers enregistré peuvent l'être au format MPEG-4 (*.mp4*) ou Quicktime (*.mov*). FIXME : Étant donné que MPEG-DASH semble supporter le MPEG-4, j'ai choisi d'enregistrer dans ce format.

Afin d'utiliser la caméra du téléphone, il faut tout d'abord choisir le bon dispositif de capture (dans notre cas, la caméra principale). Cela peut se faire avec le code suivant :

```
let captureSession = AVCaptureSession()

// Parcourt de tous les dispositifs de capture du téléphone
```

```
for device in AVCaptureDevice.devices() {

    // On vérifie qu'il gère la vidéo et qu'il s'agit de
    // la caméra arrière
    if (device.hasMediaType(AVMediaTypeVideo) &&
            device.position == AVCaptureDevicePosition.Back) {
        captureDevice = device as? AVCaptureDevice

        // On peut commencer la capture de la vidéo
        captureSession.addInput(
                AVCaptureDeviceInput(device : captureDevice))

        // On crée un calque de prévisualisation de
        // la caméra auquel on lie l'image reçue
        previewLayer =
             AVCaptureVideoPreviewLayer(session : captureSession)
        parentLayer.addSublayer(previewLayer!)
        previewLayer?.frame = parentLayer.frame

        // On démarre l'enregistrement
        captureSession.startRunning()
    }
}
```

3.2 Conversion vidéo

Le prototype de conversion vidéo est celui qui a été le plus fastidieux à produire, mais sa création permet de mieux comprendre les principes de la conversion vidéo, les différents formats de fichiers, ainsi que les problèmes que l'on peut rencontrer.

3.3 Envoi de données

Comme mentionné précédement, l'envoi des segments vidéo se fait via HTTP. Le développement d'une petite application envoyant une requête de type POST sur un serveur et contenant un payload m'a permi de testé cette fonctionnalité.

Au niveau du code, executer une requête POST en HTTP est relativement simple.

Note : ce code a été simplifié afin de n'y garder que les aspects essentiels.

```
// URL et méthode
var request = NSMutableURLRequest(URL : "http ://example.com/endpoint")
request.HTTPMethod = "POST"

// En-têtes
let boundary = "string that express the end of the file" ;
let contentType = "multipart/form-data ; boundary=" + boundary
let mimeType = "video/mp4"

// Ajout des en-têtes à la requête
request.setValue(contentType, forHTTPHeaderField : "Content-Type")

// Création du payload
var payload = "--\(boundary)\r\n" +
     "Content-Disposition : form-data ; name=\"\("uploadFile")\" ; \
               filename=\"\(fileName)\"\r\n" +
        "Content-Type : \(mimeType)\r\n\r\n" +
        String(contentsOfFile : filePath) ! + "\r\n" +
        "--\(boundary)--\r\n"

// Ajout du payload à la requête
request.HTTPBody = payload.dataUsingEncoding(NSUTF8StringEncoding)

// Envoi de la requête asynchrone
NSURLConnection.sendAsynchronousRequest(request,
     queue : NSOperationQueue.mainQueue(), completionHandler : {})
```

3.4 Gestion des ressources

Les ressources étant limitées sur une plateforme mobile, il est important de les gérer méticuleusement. Il en va de soit pour la mémoire, la consommation énergétique (batterie) et donc des différentes puces du téléphone : processeur, module réseaux (wifi, EDGE, 3G, 4G), module vidéo (caméra, encodeurs/décodeurs), etc.

Dans le cas de cette application, elle est particulièrement gourmande en ressources, étant donné qu'elle doit :

- Utiliser la caméra : activité sur les puces dédiées à la vidéo
- Encoder la vidéo : activité sur les puces vidéo et sur le processeur
- Envoyer les segments : activité réseau (puces wifi/3G/EDGE/etc.)

3.5 Sécurité des données

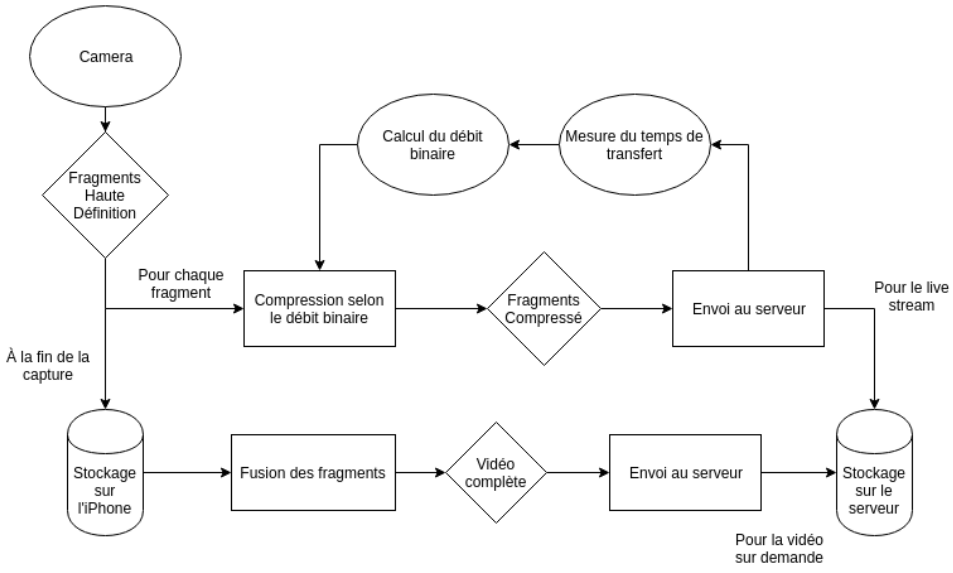
La securité des données sur le téléphone étant assurée par iOS, et les données stockées n'étant normalement pas extrêmement sensibles, il n'est pas nécessaire d'ajouter une couche de sécurité sur ces dernières.

En revanche, les segments vidéo transmis au serveur doivent être protégés. Ce niveau de sécurité doit être gérer par la partie de l'application s'occupant d'envoyer les segments sur le réseau. Heureusement, l'utilisation de HTTPS nous assure une connexion sécurisée par TLS/SSL. Il suffit alors d'utiliser un serveur muni d'un certificat TLS/SSL acceptant HTTPS et les données seront transmises de manière sûre.

Chapitre 4

Fonctionnement de l'application

Cette section présente le fonctionnement de l'application finale. Elle expose non seulement son utilisation du point de vue de l'utilisateur, mais révèle également son organisation interne.



(TODO fragment => segment)

4.1 Interface graphique

La contrainte principale liée à l'interface graphique est qu'elle doit rester simple et ergonomique, tout en permettant à l'utilisateur d'accéder aux fonctionnalités principales.

4.2 Capture et conversion de la vidéo

4.3 Envoi des segments vidéo

L'envoi des segments vidéo se fait via le protocole HTTP à l'aide de la méthode `POST`.

Le projet comprend un *protocol*[1] `Uploader` définissant les principales méthodes publiques permettant l'envoi de fichier.
Une classe `HTTPUploader`, héritant de `Uploader` implémente lesdites méthodes afin d'envoyer des fichier via HTTP.

Cette architecture permet de pouvoir facilement changer de type d'Uploader dans les révisions futures de l'application. On pourrait alors imaginer un `FTPUploader` ou un `UDPUploader`, par exemple, afin d'utiliser un autre protocole.

4.4 Adaptation du débit binaire

Comme nous l'avons mentionné dans la partie théorique, l'adaptation du débit binaire est primordiale afin de pouvoir continuer à diffuser même quand la connexion se dégrade. De plus, lorsque la connexion est de bonne qualité, la qualité de la vidéo doit être la meilleure possible pour garantir au spectateur une expérience agréable.

Concrètement, cela revient à mesurer le temps d'envoi d'un segment vidéo, de mesurer la durée de ce dernier, puis de comparer les deux afin d'obtenir un ratio durée_segment / durée_envoi > 1 (TODO Math). Comme nous avons connaissance du débit binaire du segment précédent, nous pouvons calculer le débit binaire de manière relativement simple, à l'aide de la formule suivante :

DébitBinaireSegment_n := DébitBinaireSegment_{n-1} · durée_segment / durée_envoi (TODO Math)

Enfin, comme le temps pour encoder le segment vidéo est volontairement omis et la connexion peut être relativement instable, nous choisissons empiriquement un ratio estimant la portion de temps que prendra l'envoi du fichier par rapport à sa taille. Par exemple, pour un segment de `8 secondes` et un temps d'envoi désiré de `4 secondes`, le ratio sera de `0.5`.

Ainsi, la formule finale calculant le débit binaire du prochain segment est la suivante :

DébitBinaireSegment_n := DébitBinaireSegment_{n-1} · ratio · durée_segment / durée_envoi (TODO Math)

[1] En Swift, un *protocol* est l'équivalent d'une *interface* Java.

Ce débit sera ensuite passé à l'encodeur afin qu'il compresse la vidéo au mieux selon le format choisi.

En pratique, l'application précise un débit binaire maximal et minimal afin d'éviter les cas extrèmes. En effet, en dessous d'un certain débit binaire, la vidéo devient inutilisable et il devient plus favorable d'attendre que l'utilisateur trouve une meilleure connexion. À l'inverse, une fois le débit binaire maximal atteint, on considère que la vidéo ne peut être de meilleure qualité visuellement, et qu'il est donc inutile d'ajouter de l'information.

4.5 Gestion des erreurs

4.5.1 Problèmes liés à la connexion

Chapitre 5

Validation

Comme il est difficile d'effectuer des tests unitaires sur la capture, la conversion et l'envoi de segments vidéo, la plupart des tests étaient des tests de confiance (*sanity tests*).

En premier lieu, des tests de diffusion simple ont été effectué. Le débit binaire était fixe, et le flux n'était pas lu en direct, mais une fois tous les fichiers envoyés. Cela permet de savoir si le processus de capture, conversion et envoi est fonctionnel. Ensuite, le débit binaire adaptatif a été implémenté, puis, la diffusion avec visionnage en direct a été testée.

Voici les résultats des différents tests de diffusion avec visionnage en direct :

Connexion	Durée	Résultat
Bonne	1 min	OK
Mauvaise	1 min	OK
Variable	3 min	OK
Interrompue	6 min	OK

Dans les deux derniers cas, le décalage entre la capture de la vidéo et sa diffusion était d'environ 25 secondes. Même si ce temps est assez court pour être considérer comme acceptable par l'utilisateur, il reste étonnamment grand. Or, après investigation, il s'avère que les fichiers vidéo sont bel et bien créés sur le serveur après une douzaine de secondes. Ce décalage est dû à la longueur du segment (~8 secondes) auquel s'ajoute son temps d'envoi (~4 secondes). Le fichier manifeste, quant à lui, est envoyé juste après le segment, mais sa faible taille rend son temps d'envoi négligeable (moins d'une seconde). Le reste du décalage, de l'ordre d'une douzaine de secondes, est alors dû au lecteur employé lors des tests afin de visionner le flux HLS. Ce lecteur est le lecteur HLS natif d'Apple qui est disponible sur iOS ainsi

que Mac OS X, et également sur Safari. On peut alors supposer que ce décalage est une contrainte techinque du client ou qu'il apporte un expérience plus agréable à l'utilisateur en cas de coupure.

Chapitre 6

Problèmes rencontrés

Durant le déroulement de ce projet, de nombreux problèmes ont été rencontré. Les principaux d'entre eux sont approfondis dans cette section.

6.1 Conversion vidéo à l'aide de FFmpeg

La partie qui m'a posé le plus de difficulté était sans doute d'appeler les différentes fonctions de FFmpeg depuis mon code Swift. En effet, je n'avais jamais essayer de lier des bibliothèques externes à XCode et à Swift. Lorsque j'ai rencontré des erreurs à la compilation lors de l'appel au linker, il m'était impossible de savoir d'où provenait l'erreur. Elle pouvait se trouver sur une multitude de niveaux :

- La cross-compilation
- La fusion des bibliothèques statiques
- Leur inclusion au projet XCode
- La configuration de mon projet XCode (tant au niveau des libs que des headers)
- L'utilisation du Bringing Header
- L'appel aux fonctions depuis Swift

De plus, certaines erreurs du linker provenaient du fait que FFmpeg a besoin des bibliothèques libssl, libcrypto et libiconv.

Finalement, après de nombreux essais et renseignements sur Internet, j'ai pu convertir des vidéos d'un format à l'autre, changeant non seulement le format du conteneur mais aussi le codec utilisé.

Ce prototype m'a pris de nombreuses heures à faire fonctionner parce que l'ensemble du système comporte de nombreuses couches et aussi à cause de mon manque d'expérience sur ces technologies. Cependant, la gestion de la vidéo est

une clé de voute de ce projet et ce prototype et les problèmes qu'il m'a causé m'aideront beaucoup dans la réalisation de l'application finale.

Chapitre 7

Conclusion (TODO)

Chapitre 8

Références

- Documentation sur AVFoundation : https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html#//apple_ref/doc/uid/TP40010188
- Documentation sur AVCapture : https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/04_MediaCapture.html#//apple_ref/doc/uid/TP40010188-CH5-SW2
- Documentation sur la diffusion en direct sur le web : https://developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/Live_streaming_web_audio_and_video
- Documentation sur la diffusion adaptative (changement de débit binaire) : https://developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/Setting_up_adaptive_streaming_media_sources
- Exemples de manifestes HLS : https://developer.apple.com/library/ios/technotes/tn2288/_index.html#//apple_ref/doc/uid/DTS40012238-CH1-TNTAG3
- Exemple de MPEG-DASH à l'aide de FFmpeg en ligne de commande : https://developer.mozilla.org/en-US/docs/Web/HTML/DASH_Adaptive_Streaming_for_HTML_5_Video
- Documentation sur les Bridging Headers : https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html
- Documentation de FFmpeg : http://www.ffmpeg.org/
- Tutoriel sur la capture vidéo sur iOS : http://jamesonquave.com/blog/taking-control-of-the-iphone-camera-in-ios-8-with-swift-part-1/

- StackOverflow pour une multitude de questions sur les diverses technologies ainsi que des exemples de code. :
  http://stackoverflow.com/
- Projet payant et propiétaire de diffusion vidéo en live :
  https://kickflip.io/
- Exemple d'envoi de fichier via HTTP :
  https://gist.github.com/janporu-san/e832cdee51974fc55660
- Module permettant de faire du HLS :
  https://github.com/hudl/iOS-FFmpeg-processor
- Script de compilation de FFmpeg pour iOS :
  https://github.com/bbcallen/ijkplayer/blob/fc70895c64cbbd20f32f1d81d2d48609ed13f597/ios/tools/do-compile-ffmpeg.sh
- Wrapper de FFmpeg pour convertir des vidéos :
  https://github.com/OpenWatch/FFmpegWrapper