Internet et Multimedia Streaming
avec les exercices sur Tegra Jetson X1

Les laboratoires M1 (Internet et Multimédia) sont basées sur les cartes **Tegra Jetson X1**. Cette plateforme embarquée avec les processeurs **ARM** et **GPU** Nvidia intégrée est une de plus puissantes architectures pour le développement des systèmes multimédia et du streaming sur Internet.
Dans les premiers laboratoires ,**Lab 1** et **Lab 2** (**Internet**), nous allons reprendre les sujets de base concernant la communication UDP et TCP avec le modèle Client-Serveur.
Avant de commencer le travail vérifiez les interfaces réseau (voir si elle sont déjà correctement configurées) sur la carte Jetson X.

Contenu

Carte Jetson TX1 1
Connexion de l'interface Ethernet/WiFi 1

Lab 1 2
Programmation «socket» en C, protocole UDP 2
1.1 Les fonctionnalités de base de l'API « socket » 2
1.2 Caractéristiques des sockets 3
1.2.1 Création et attachement d'une socket 3
1.2.2 Communication UDP par envoi de message 4
1.2.2.1 Code de l'émetteur - " client " 4
1.2.2.2 Code du récepteur - " serveur " 5
A faire : 5

Lab 2 6
Programmation avec API socket en C , protocole TCP 6
2.1 socket – TCP 6
2.1.1 TCP sender (client) 6
2.2.2 TCP receiver (server) 7
A faire : 8

Lab 3 9
Gstreamer-1.0 , fonctions de base 9
3.1 Compression, décompression vidéo 9
3.1.1 Fonctionnement de la compression et de la décompression vidéo 9
3.1.2 Sources 10
3.1.3 Multiplexeurs et Démultiplexeurs 10
3.1.4 Encodeurs et décodeurs 10
3.1.5 Sinks 11
Gstreamer 1.0 - installation 11
Sur Ubuntu 11
Sur Windows 11
3.3 Lecture et présentation des contenus audio/vidéo par commandes (pipelines) GStreamer 12
3.3.1 Quelques exemples de pipelines pour commencer 12
3.3.2 Un moyen simple de décoder les flux vidéo et audio avec GStreamer 13
3.3.3 Lire et encoder un fichier audio/vidéo dans un autre format 13
3.4 Lecture des flux vidéo/audio de la webcam 15
3.4.1 Lecture des flux vidéo et leur affichage 16
3.4.2 Lecture des flux vidéo et leur enregistrement 16
3.4.3 Capture audio de la webcam avec microphone 16
3.4.4 Lecture des flux vidéo à partir de la camera intégrée 17

Lab 4 – projet final....18
Gstreamer-1.0 , streaming....18
4.1 Protocoles utilisés....18
Configuration....18
4.1.1 UDP....18
Emission :....18
Réception :....18
4.1.2 TCP....19
Emission :....19
Réception :....19
4.1.3 RTP/UDP....19
4.1.3.1 Emission et réception d’un flux vidéo (h264)....19
Emission :....19
Réception avec décodage hardware avec omx (accéléré) :....19
4.1.3.2 Emission et réception d’un flux audio (SPEEX)....20
4.1.4 RTCP/RTP/UDP....20
4.1.5 Vidéo et audio en RTCP/RTP/UDP avec rtpbin....21
Emission avec h264:....21
A faire....22
Emission avec jpeg 22
A faire :....22
4.2 Streaming simultané de plusieurs vidéos....23

Internet et Multimedia Streaming
avec les exercices sur Tegra Jetson X1

Les laboratoires M1 (Internet et Multimédia) sont basées sur les cartes **Tegra Jetson X1**. Cette plateforme embarquée avec les processeurs **ARM** et **GPU** Nvidia intégrée est une de plus puissantes architectures pour le développement des systèmes multimédia et du streaming sur Internet.
Dans les premiers laboratoires ,**Lab 1** et **Lab 2** (**Internet**), nous allons reprendre les sujets étudiés en quatrième année ETN :
Avant de commencer le travail vérifiez les interfaces réseau (voir si elle sont déjà correctement configurées) sur la carte Jetson X.

Lab 1 et Lab2 : Programmation « socket » en langage C et développement des applications client-serveur en mode datagramme (UDP et en mode connecté (TCP).

Dans la deuxième partie du module (Multimédia et Streaming) nous allons utiliser le framework de **Gstreamer**, la plus importante plateforme open source pour le développement du multimédia et du streaming.
Vous allez expérimenter avec :
Lab 3. Les entrées/sorties multimédia et la capture et la compression des flux vidéo/audio, le « playout » et le transcodage des formats multimédia.
Lab 4. Le streaming vidéo, audio, vidéo/audio simple avec **UDP**, **TCP**, et **RTP et RTCP**
Lab 5. Le dernier laboratoire permettra de choisir une application à développer (par exemple streaming vidéo avec l'incrustation d'un logo) et de préparer le compte rendu de l'ensemble des laboratoires

Comme source d'information et des codes nous allons utiliser une préparation installée sur la carte et la documentation Gstreamer sur Tegra X1. Les codes de départ sont fournis dans un répertoire installé sur la carte.

Carte Jetson TX1

La carte utilisée dans ces Labs est la Jetson TX1 de NVIDIA qui contient le processeur Tegra X1. Le GPU est un processeur graphique NVIDIA Maxwell à 256 coeurs de 1 teraflops et le CPU est un processeur ARM Cortex A57 64 bits à 4 coeurs (1,9 GHz) il est secondé par un autre processeur ARM Cortex A53 à 4 coeurs (1,3 GHz). Il y a donc huit coeurs mais ils ne sont jamais activés en même temps. Pour des tâches lourdes les coeurs du Cortex A57 s'activent tandis que pour de petites tâches c'est ceux du Cortex A53 qui s'activent.
La carte peut traiter un flux vidéo en définition 4K à 60 images par secondes d'où son utilisation pour un tel projet. La carte embarque également une caméra qui supporte 1300 mégapixels par seconde.

Pour la configuration des cartes, Ubuntu était déjà pré-installé sur la carte, cependant il est utile de mettre à jour le système avec la fonction "**update**".
La carte Tegra X1 peut être connectée au réseau de différentes manières.

Connexion de l'interface Ethernet

L'adresse IP de l'interface Ethernet (sur le câble Ethernet du PC **école**) doit être configurée statiquement dans le fichier avec les droits de **`root`**: (mot de passe **`jetson`**)
`/etc/network/interfaces`

```
iface eth0 inet static
address  172.19.65.60
network  255.255.248.0
gateway  172.19.64.3
nameserver 172.19.0.4
```

Les adresses IP disponibles pour ce laboratoire sont entre :
`172.19.65.50` **et** **`172.19.65.62`**

La commande :
```
$ ip a
```
permet de visualiser les paramètres des interfaces réseau.
Attention : Elle nécessite également l'utilisation d'un proxy : **`193.52.104.20:3128`** .

Pour lancer le **browser** :
```
chromium-browser  --proxy-server= 193.52.104.20:3128
```

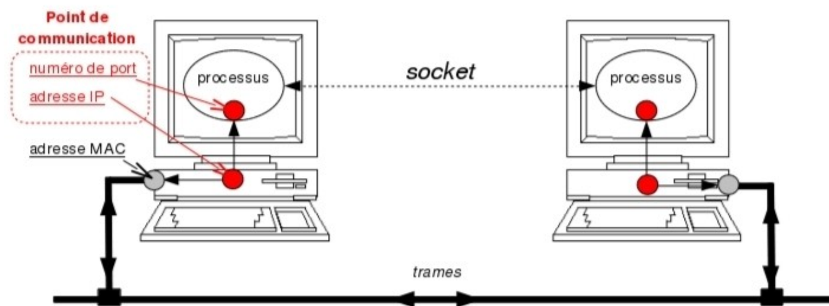
Lab 1

Programmation «socket» en C, protocole UDP

Dans ce laboratoire nous allons étudier les fonctionnalités de l'interface de programmation réseaux« socket » puis nous allons développé une application de transfert de fichiers en mode non-connecté avec le protocole **UDP** avec les differents modes d'adressage IP :

- **`uni-cast,`**
- **`broad-cast et`**
- **`multi-cast`**

1.1 Les fonctionnalités de base de l'API « socket »



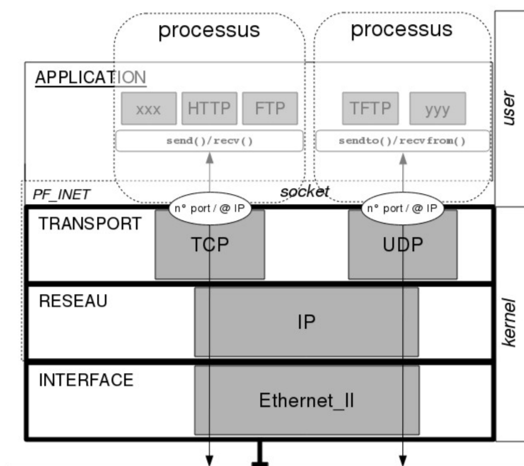
Une socket est communément représentée comme un point d'entrée initial au niveau **transport** du modèle à couches TCP/IP.

La couche **transport** est responsable du transport des messages complets de bout en bout au travers du réseau. En programmation, si on utilise comme point d'entrée initial le niveau **transport**, il faudra alors choisir un des deux protocoles de cette couche :

- **User Datagram Protocol – UDP** est un protocole souvent décrit comme étant non-fiable, en mode **non-connecté**
- **Transmission Control Protocol - TCP** est un protocole de transport fiable, en **mode connecté**.

Numéro de ports est un numéro qui sert à identifier un **processus** (**l'application**) en cours de communication par l'intermédiaire de son protocole de couche application (associé au **service** utilisé, exemple : **80** pour **HTTP**).
L'attribution des ports est faite par le système d'exploitation, sur demande d'une application. Ici, il faut distinguer les deux situations suivantes :

1. cas d'un **processus client**: le numéro de port utilisé par le client sera envoyé au processus serveur. Dans ce cas, le processus client peut demander à ce que le système d'exploitation lui attribue n'importe quel port, à condition qu'il ne soit pas déjà attribué.
2. cas d'un **processus serveur**: le numéro de port utilisé par le serveur doit être connu du processus client. Dans ce cas, le processus serveur doit demander un numéro de port précis au système d'exploitation qui vérifiera seulement si ce numéro n'est pas déjà attribué



1.2 Caractéristiques des sockets

Les sockets compatibles représentent une interface uniforme entre le processus utilisateur(user) et les piles de protocoles réseau dans le noyau (***kernel***) du Linux. Pour dialoguer, chaque processus devra préalablement **créer une socket** de communication en indiquant le domaine de communication (dans notre cas Internet IPv4) et le type de protocole à employer (**TCP**/**UDP**..).
Pour le protocole `PF_INET`, on aura le choix entre :`SOCK_STREAM` (qui correspond à un mode connecté donc **TCP** par défaut), et `SOCK_DGRAM` (qui correspond à un modenon connecté donc **UDP**).

Un `SOCK_RAW` permet un accès direct aux protocoles de la **couche réseau** et la **couche physique/lien** comme **IP**, **ICMP**, et **Ethernet/WiFi**).

1.2.1 Création et attachement d'une `socket`

La création d'une socket se fait par l'appel système `socket(2)` dont la déclaration se trouve dans `<sys/socket.h>`. Cet appel permet de créer une structure en mémoire contenant tous les renseignements associés à la socket (**buffers**, **adresse**, etc.) ; il renvoie un descripteur de fichier permettant d'identifier la socket créée (`-1` en cas d'erreur).

```
int socket (
  int domain,   /* AF_INET pour l'internet */
  int type,     /* SOCK_DGRAM pour une communication UDP,
                   SOCK_STREAM pour une communication TCP */
  int protocole /* 0 pour le protocole par defaut du type */
);
```

Une fois la socket créée, il est possible de lui **attacher une adresse** qui sera généralement l'adresse locale ; sans adresse une socket ne pourra pas être contactée (il s'agit simplement d'une structure qui ne peut pas être vue de l'extérieur).
L'attachement permet de préciser l'**adresse ainsi que le port de la socket**.
On attache une adresse à une socket à l'aide de la fonction **bind(2)** qui renvoie **0** en cas de succès et **-1** sinon.

```
int bind (
  int descr,             /* descripteur de la socket */
  struct sockaddr *addr, /* adresse a attacher       */
  int addr_size          /* taille de l'adresse      */
);
```

L'exemple ci-dessous définit une fonction permettant de créer une socket et de l'attacher sur le port spécifié de l'hôte local.

```
/* ****************************************************************
 * type : type de la socket a creer = SOCK_DGRAM ou SOCK_STREAM
 * port : numéro de port désiré pour l'attachement en local
 *****************************************************************/
int creer_socket (int type, int port)
{
  int desc;
  int longueur=sizeof(struct sockaddr_in);
  struct sockaddr_in adresse;
  /* Creation de la socket */
  if ((desc=socket(AF_INET,type,0)) == -1)
  {
    perror("Creation de socket impossible"); return -1;
  }
  /* Preparation de l'adresse d'attachement = adresse IP Internet */
  adresse.sin_family=AF_INET;
  adresse.sin_addr.s_addr=htonl(INADDR_ANY);
  // Attention : sur NetKit vous avez 2 adresses
  // si port=0, numéro aléatoire (disponible) entre 0 et 1024 */
  adresse.sin_port=htons(port);
  if (bind(desc,(struct sockaddr*)&adresse,longueur) == -1)
  {
    perror("Attachement de la socket impossible"); close(desc); return -1;
  }
  return desc;
}
```

1.2.2 Communication UDP par envoi de message

Afin d'établir une communication UDP entre deux machines, il faut d'une part créer un serveur sur la machine réceptrice et d'autre part créer un client sur la machine émettrice. Ensuite la communication peut se faire à l'aide des fonctions `sendto()` et `recvfrom()`. Chaque utilisation de `sendto()` génère un **paquet UDP** qui doit être lu en **une seule fois** par la fonction `recvfrom()`.

1.2.2.1 Code de l'émetteur - " client "

```
//  udpsend.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUFLEN 512       // Max length of buffer
#define PORT 8888        // The port on which to send or listen for data
#define REMOTE_ADDR  "192.168.1.72"

void die(char *s)      // exit with message
{
  perror(s); exit(1);
}

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s, i, slen = sizeof(si_other) , recv_len;
  char buf[BUFLEN];
  // create a UDP socket
  if ((s=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP))==-1){die("socket");}
  // zero out the structure me (sender)
  // this structure and bind() are optional !
  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  //bind socket to port
  if(bind(s,(struct sockaddr*)&si_me,sizeof(si_me) )==-1){die("bind");}
  // zero out the structure other (receiver) - this structure is mandatory
  memset((char *) &si_other, 0, sizeof(si_other));
  si_other.sin_family = AF_INET;
  si_other.sin_port = htons(PORT);    // to work remotely
//si_other.sin_port = htons(PORT+1);  // to work locally
//si_other.sin_addr.s_addr = htonl(INADDR_ANY); // to work locally
  si_other.sin_addr.s_addr = inet_addr(REMOTE_ADDR);   // to work remotely
  //keep sending data
  while(1)
    {
    printf("write th0,25 cme message\n");scanf("%s",buf);
    //try to send data, this is a non blocking call
    if((sendto(s,buf,strlen(buf)+1,0,(struct sockaddr *)&si_other,slen)) == -1)
      { die("sendto()"); }
    }
  close(s);
  return 0;
}
```

1.2.2.2 Code du récepteur - " serveur "

```
//  udprecv.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUFLEN 512      //Max length of buffer
#define PORT 8888       //The port on which to listen for incoming data

void die(char *s)
{
        perror(s);
        exit(1);
}

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s, i, slen = sizeof(si_other),rlen;
  char buf[BUFLEN];
  //create a UDP socket
  if ((s=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP))==-1) {die("socket");}
  // zero out the structure
  memset((char *) &si_me, 0, sizeof(si_me));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  //bind socket to port
  if( bind(s,(struct sockaddr*)&si_me,sizeof(si_me))==-1) {die("bind");}
  //keep listening for data
  while(1)
    {
    printf("Waiting for data...");
    //try to receive some data, this is a blocking call
    if((rlen=recvfrom(s,buf,BUFLEN,0,(struct sockaddr *)&si_other,&slen))==-1)
      {die("recvfrom()"); }
    //print details of the client/peer and the data received
    printf("IP, port %s:%d\n",inet_ntoa(si_other.sin_addr),
                              ntohs(si_other.sin_port));
    printf("Received data: %s\n", buf);
    }
  close(s);
  return 0;
}
```

A faire :

1. Compléter le code ci-dessus pour obtenir une **communication bidirectionnelle** , un " **chat** "
2. Compléter le code ci-dessus pour obtenir une application de transfert de fichiers – **remote copy**.

Dans ce sujet il faut utiliser les arguments du programme `main(argc,*argv[])` pour indiquer le nom du fichier à transmettre – source (lecture) et destination (création).

Rappel :

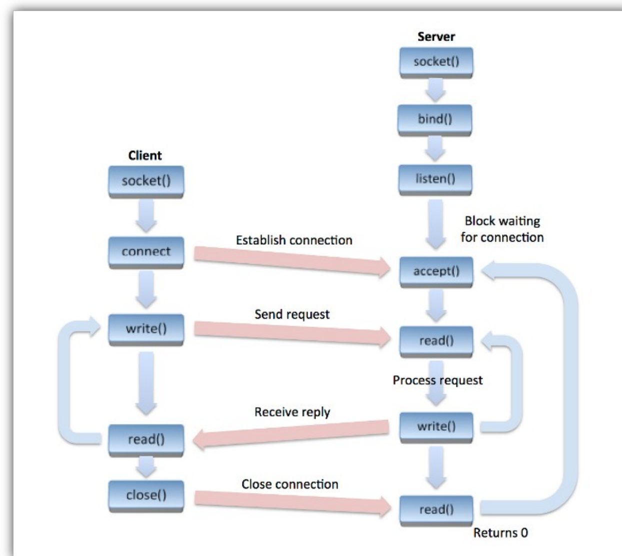
Les fonctions de l'ouverture, de la création, de la lecture, et l l'écriture d'un fichier sont les suivantes :

```
int fr,fw ; char buf[512]; int nb;
fr=open(argv[1],0) ;    // ouverture en lecture
fw=creat(argv[2],0777) ;  // création avec tous les droits
nb=read(fr,buf,512) ;      // lecture dans le buf de max 512 octets
write(fw,buf,nb) ;          // écriture à partir de buf de nb octets
```

Lab 2
Programmation avec API socket en C , protocole TCP

2.1 socket – TCP

Afin d'établir une communication TCP entre deux machines, il faut d'une part créer un serveur sur la machine réceptrice, d'autre part créer un client sur la machine émettrice. Il faut ensuite réaliser une connexion entre les deux machines, qui sera gérée **côté serveur** par les fonctions `listen()` et `accept()`, et **côté client** par la fonction `connect()`. La communication peut alors se faire à l'aide des fonctions `write()` et `read()`.



2.1.1 TCP sender (client)

```
// tcpsender.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 512       // Max length of buffer
#define PORT 8888        // the port on which to send or listen for data
#define REMOTE_ADDR  "192.168.1.72"

void die(char *s)       // exit with message
{
        perror(s);exit(1);
}

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s, i, slen = sizeof(si_other) , recv_len;
  char buf[BUFLEN];
  s = socket(AF_INET, SOCK_STREAM, 0);
  memset((char *) &si_other, 0, sizeof(si_other));
  si_other.sin_family = AF_INET;
  si_other.sin_port = htons(PORT);     // to work remotely
//si_other.sin_port = htons(PORT+1);   // to work locally
//si_other.sin_addr.s_addr = htonl(INADDR_ANY); // to work locally
```

```
  si_other.sin_addr.s_addr = inet_addr(REMOTE_ADDR);   // to work remotely
  if(connect(s,(struct sockaddr *)&si_other,sizeof(si_other))<0)
                                                           die("connect") ;
  else
    {
    while(1)
      {
      memset(buf,0x00,BUFLEN);
      printf("write the message, replace spaces by _, end session with
             . message\n");scanf("%s",buf);
      if(write(s,buf,strlen(buf)+1)<0) die("write") ;
      sleep(3);
      if(buf[0]=='.') break;
      }
    }
  close(s);
}
```

2.2.2 TCP receiver (server)

```
// tcprecv.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 512      // Max length of buffer
#define PORT 8888       // The port on which to send or listen for data

void die(char *s)      // exit with message
{
        perror(s); exit(1);
}

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s,ns, nb, slen = sizeof(si_other);
  char buf[BUFLEN];
  s=socket(AF_INET, SOCK_STREAM, 0);
  memset((char *) &si_other, 0, sizeof(si_other));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  // Bind is mandatory for the server
  if( bind(s,(struct sockaddr *)&si_me , sizeof(si_me)) < 0) die("bind");
  listen(s,5);   // log length
  while(1)
    {
    //Accept and incoming connection
    puts("Waiting for incoming connections...");
    ns=accept(s, (struct sockaddr *)&si_other,(socklen_t*)&slen);
    if (ns<0) { puts("accept failed"); continue; }
    else  puts("Connection accepted");
    while(1)
      {
      memset(buf,0x00,BUFLEN);
      nb=read(ns,buf,BUFLEN);
      printf("Got message %d bytes: %s\n",nb,buf); if(nb==0) break;
      if(buf[0]=='.') { printf("session ended\n");break;}
      }
    close(ns);  // close the working socket
    }
  close(s);
}
```

A faire :

1. **Compléter** les codes ci-dessus pour **transmettre un fichier** – voir **Lab1**
2. **Ecrire** une version «**parallèle**» du serveur (`tcprecvfork.c`) avec une fonction `fork()` qui génère un **processus fils** pour s'occuper du «client».

Rappel :
La fonction **fork()** permet de créer un nouveau processus , qui peut s'exécuter sur une autre unité centrale (CPU) pendant que le processus fils attend une nouvelle demande de connexion sur le socket principal (`s`).

```
// rappel sur fork()
  ns=accept(s, (struct sockaddr *)&si_other,(socklen_t*)&slen);
  if (ns<0) { puts("accept failed"); continue; }
  else
    {
    puts("Connection accepted");
    if(fork()==0)  // I am child
      {
      while(1)
        {
        memset(buf,0x00,BUFLEN);
        nb=read(ns,buf,BUFLEN);
        printf("Got message %d bytes: %s\n",nb,buf); if(nb==0) break;
        if(buf[0]=='.') { printf("session ended\n");break;}
        }
        close(ns);
      }
   else close(ns);
```

Lab 3

Gstreamer-1.0 , fonctions de base

GStreamer est une librairie logicielle gratuite et open-source écrite en C. En se basant sur un système de **pipelines**, GStreamer permet de manipuler du son et des images.
Grâce à GStreamer il est possible de lire, coder, décoder, transcoder, filmer, émettre et recevoir des fichiers audios ou vidéos.

Il existe deux moyens d'utiliser GStreamer, on peut soit utiliser directement des commandes GStreamer dans un terminal soit faire appel à un fichier C qui lui même va faire appel à GStreamer.
GStreamer fonctionne principalement en utilisant le CPU, cependant il existe quelques fonctions qui permettent d'utiliser le GPU. Celles utilisées dans les Labs sont des fonctions d'encodage et de décodage avec la dénomination omx.

Pour le fonctionnement basique de Gstreamer, deux parties se dégagent, la partie compression, décompression vidéo et la partie émission réception de données.

Attention : **Le texte suivant est à lire avant le passage aux expérimentations !**

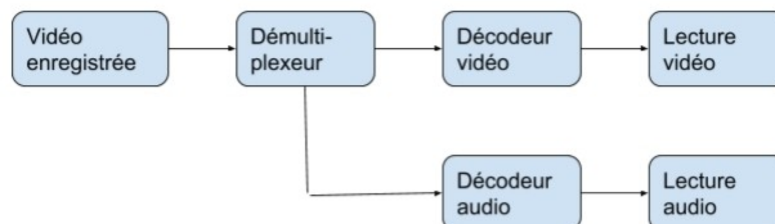
3.1 Compression, décompression vidéo

3.1.1 Fonctionnement de la compression et de la décompression vidéo

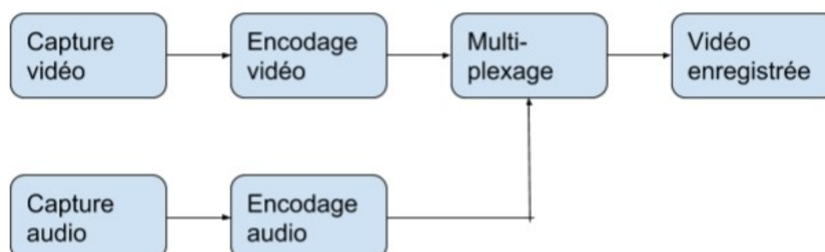
Avant de pouvoir utiliser les commandes Gstreamer, il faut d'abord se pencher sur les étapes de la compression et décompression vidéo qu'il faudra détailler lors de l'utilisation de Gstreamer.
Deux cas sont mis en avant car utilisés lors du projet technique.

Le **premier cas** est l'utilisation d'une **vidéo enregistrée en mémoire** dans un format comprenant la vidéo et l'audio. Pour récupérer les images en brut, il faut commencer par **démultiplexer** la vidéo enregistrée pour séparer les images de l'audio en utilisant le décodeur adéquate.

Ensuite, il faut **décoder** les images avec le décodeur adapté. On obtient alors les images au format brut, il est maintenant possible d'apporter des modification aux images de la vidéo ou simplement de les afficher. Pour le son, le processus est similaire, après le démultiplexage, il faut décoder le son qui est alors au format brut, et qui peut être lu.



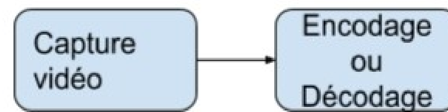
Pour obtenir une **vidéo enregistrée**, il faut réaliser le cheminement inverse, après avoir récupéré la vidéo et le son en format brut, il faut les **encoder** dans un format ou ils seront compatibles pour un multiplexage, puis on applique le **multiplexage** et on obtient une **vidéo enregistrée**.



Le deuxième cas est l'utilisation de la vidéo uniquement en sortie d'une **webcam**. La webcam peut fournir plusieurs formats en vidéo, comme le format **brut, mjpeg,** ou le **H264**. Lorsque l'on veut modifier le flux vidéo de la webcam et que le format de la vidéo est brut alors il est possible de travailler directement sur le flux.

En revanche, si le format est **H264**, alors il faut d'abord **décoder le flux** avant de pouvoir travailler dessus.

À l'inverse, si on veut récupérer la vidéo sans la modifier et que le format est en H264 il n'y a rien à modifier, mais si le format est brut alors il faut encoder le flux vidéo.



3.1.2 Sources

Afin d'avoir un flux vidéo ou audio il est nécessaire d'avoir une source, cela peut venir d'une caméra, un microphone ou encore un fichier enregistré sur un disque ou SSD

Liste des sources utilisées :

- **`v4l2src`** permet d'utiliser les sources externes venant d'un dispositif branché à un périphérique tel qu'une webcam. Il est souvent utile de préciser quel **device** doit être utiliser dans le cas où il y en aurait plusieurs.
- **`udpsrc`** permet d'utiliser les données reçues par le protocole **udp** (et par extension, **rtp** et **rtcp**) comme source.
- **`filesrc`** permet d'utiliser un fichier enregistré dans la mémoire interne de la carte comme source.
- **`alsasrc`** permet d'utiliser une source externe en périphérique comme pour **`v4l2src`** mais pour le son.

3.1.3 Multiplexeurs et Démultiplexeurs

Le multiplexage est une étape importante qui va permettre de rassembler plusieurs flux d'informations dans un signal unique avant d'émettre. Le but est de faciliter le transport des données, il faut cependant séparer les données après réception pour pouvoir ensuite les utiliser.
Typiquement cela peut être utilisé pour une vidéo où l'on va multiplexer le son et les images pour la transmission.

Il existe de nombreux multiplexeurs pouvant être utilisés. Cependant, le projet utilisera principalement des vidéos au format **H264**, donc un multiplexeur comme **`mp4mux`** est suffisant.

Les vidéos enregistrées sont au format **mp4**, c'est pourquoi le démultiplexeur à utiliser est **`qtdemux`**.

3.1.4 Encodeurs et décodeurs

Pour les encodeurs, et décodeurs, seuls ceux utilisant la norme H264 sont utilisés. De plus, les encodeurs et décodeurs utilisant le VPU de la carte NVIDIA sont priorisés du fait de leur plus grande vitesse et performance. Les encodeurs et décodeurs tels que **`x264enc`** ou **`avdec_h264`** peuvent être utilisés mais ils fonctionnent moins vite.

L'encodeur vidéo accéléré utilisé est **`omxh264enc`** qui permet d'encoder en format **H264** en utilisant le VPU.

Le décodeur vidéo accéléré est **`omxh264dec`** qui permet de décoder un fichier en format H264 en utilisant le VPU.

L'encodeur possède des **options** permettant de choisir la manière dont le flux est encodé, comme le nombre d'images **I**, P, et B à utiliser, l'intervalle entre les images **I**, la qualité ou encore définir un **bitrate** et choisir s'il est constant ou variable.

L'encodage et le décodage en H264 est très souvent accompagné de la fonction **`h264parse`**. Cette fonction analyse le flux H264 et permet de convertir le flux H264 d'un format H264 à un autre ou d'insérer périodiquement des **SPS**/**PPS** (Sequence Parameter Set/Picture Parameter Set) via la propriété **`config-interval`**, les deux entités contiennent des informations qu'un décodeur H264 a besoin pour décoder les données vidéo, par exemple la résolution et la fréquence d'images de la vidéo.

Cette fonction est utile lorsqu'il y a plusieurs sources ou à la transmission d'une vidéo.

L'élément **decodebin** est particulier car il permet de trouver **automatiquement** le meilleur **démultiplexeur et/ou décodeur** à appliquer sur les flux vidéos.

3.1.5 Sinks

Les **sinks** permettent de transmettre les données par un **protocole de communication**, d'un fichier ou en sortie d'un périphérique. C'est donc grâce aux **sinks** qu'on va pouvoir lancer un lecteur pour lire une vidéo ou un son.

Liste des **sinks** utilisés :

- **autovideosink** détecte et utilise automatiquement le lecteur adapté pour la vidéo.
- **alsasink** permet de lire une piste audio.
- **glimagesink** utilise **openGL** intégré à Gstreamer pour faire de l'affichage.
- **udpsink** permet d'envoyer les données sur le réseau avec le protocole UDP.
- **ximagesink** permet de faire un affichage vidéo.
- **xvimagesink** fait de l'affichage vidéo avec **XVideo** qui permet d'accélérer la mise à l'échelle des images en acceptant toutes les trames quelques soient leurs tailles avant de les mettre à l'échelle directement.

Gstreamer 1.0 - installation

Sur Ubuntu

Exécuter la commande suivante :

```
apt-get install libgstreamer1.0-0 gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-libav gstreamer1.0-doc gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-pulseaudio
```

Sur Windows

Allez sur le lien suivant et téléchargez 2 fichiers d'installation et installez les :

https://gstreamer.freedesktop.org/data/pkg/windows/1.18.1/mingw/

gstreamer-1.0-mingw-x86-1.18.1.msi

gstreamer-1.0-devel-mingw-x86-1.18.1.msi

Puis utilisez un terminal pour travailler en ligne de commande.

L'ensemble de commandes et des composants sera installé dans :

```
C:\gstreamer\1.0\mingw_x86\bin
```

Vous pouvez également ajouter un **PATH** dans votre environnement.

3.3 Lecture et présentation des contenus audio/vidéo par commandes (pipelines) GStreamer

Prenons un exemple assez général: une vidéo avec du son. Nous supposerons que vous souhaitez lire un fichier **`.mp4`** contenant un film complet. L'idée est d'obtenir deux flux vidéo et audio.
C'est ce que nous appelons le démultiplexage. Le fichier conteneur génère une source binaire que nous devons couper en deux flux séparés.
Ces **flux** ont un format d'encodage (**`mp3`**, **`vorbis`**, **`h264`**, **`theora`**, ...) qui doit être décodé pour être utilisé.
Enfin, vous devez envoyer le flux décodé dans les sorties appropriées: la vidéo à l'écran, et le son vers les haut-parleurs .

Prenez d'abord un fichier **`mp4`** et essayez de lire et de lire uniquement la partie **audio**. Commençez par la commande **`gst-typefind-1.0`** qui nous montre le type de codec utilisé pour construire le fichier multimédia.

Notez que le fichier **`tom.mp4`** doit se trouver dans le même répertoire à partir duquel vous lancez la commande **`gst-launch-1.0`** , sinon vous devez fournir le *chemin d'accès*, par exemple: **`../../samples/tom.mp4`**.

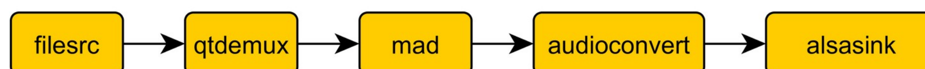
```
gst-typefind-1.0 tom.mp4
tom.mp4 - video/quicktime, variant=(string)iso
```

La commande fournit les informations sur les **plugins** utilisés pour multiplexer-démultiplexer et coder un fichier **`mp4`**:

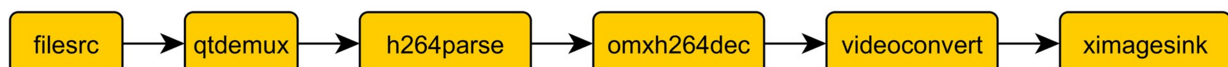
```
$ gst-inspect-1.0 | grep mp4

libav:  avmux_mp4: libav MP4 (MPEG-4 Part 14) muxer (not recommended, use mp4mux
instead)
typefindfunctions: video/quicktime: mov, mp4
isomp4:  qtdemux: QuickTime demuxer
isomp4:  rtpxqtdepay: RTP packet depayloader
isomp4:  qtmux: QuickTime Muxer
isomp4:  mp4mux: MP4 Muxer
isomp4:  ismlmux: ISML Muxer
isomp4:  3gppmux: 3GPP Muxer
isomp4:  mj2mux: MJ2 Muxer
isomp4:  qtmoovrecover: QT Moov Recover
rtp:  rtpmp4vpay: RTP MPEG4 Video payloader
rtp:  rtpmp4vdepay: RTP MPEG4 video depayloader
rtp:  rtpmp4apay: RTP MPEG4 audio payloader
rtp:  rtpmp4adepay: RTP MPEG4 audio depayloader
rtp:  rtpmp4gdepay: RTP MPEG4 ES depayloader
rtp:  rtpmp4gpay: RTP MPEG4 ES payloader
```

3.3.1 Quelques exemples de pipelines pour commencer

Voici quelques exemples (à tester) de ligne de commandes (pipelines Gstreamer) pour la lecture d'une audio/vidéo mp3/**`mp4`**.

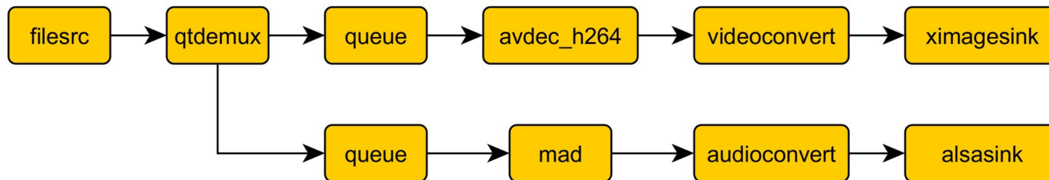



```
gst-launch-1.0 filesrc location=tom.mp4 ! qtdemux ! mad ! audioconvert !
alsasink
```




```
gst-launch-1.0 filesrc location=tom.mp4 ! qtdemux ! h264parse ! omxh264dec !
videoconvert ! Ximagesink
```

Maintenant que nous connaissons les noms probables des pistes: **`audio_0`** et **`video_0`**, nous pouvons essayer de les traiter séparément. Voici le pipeline (double) requis:




```
gst-launch-1.0 -v filesrc location="tom.mp4" ! qtdemux name=q q.video_0 !
queue ! avdec_h264 ! videoconvert ! ximagesink q.audio_0 ! queue ! mad !
audioconvert ! alsasink
```

3.3.2 Un moyen simple de décoder les flux vidéo et audio avec GStreamer

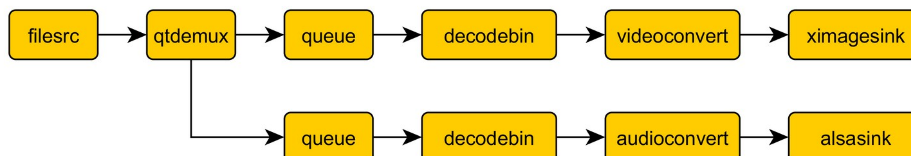
Les pipelines présentés ci-dessus utilisent des décodeurs **explicitement** spécifiés pour décoder la vidéo et les flux audio.

GStreamer propose ici un moyen simple de décoder via le plugin `decodebin` qui détecte automatiquement le **type** de flux/piste et applique le **codec** correspondant.

Décodage et lecture de la vidéo:

```
gst-launch-1.0 filesrc location=tom.mp4 ! decodebin ! videoconvert ! ximagesink
```

Decoding and playing video and audio :




```
gst-launch-1.0 filesrc location=tom.mp4 ! qtdemux name=foo foo.video_0 ! queue !
decodebin ! videoconvert ! ximagesink foo.audio_0 ! queue ! decodebin !
audioconvert ! alsasink
```

3.3.3 Lire et encoder un fichier audio/vidéo dans un autre format

Prenons maintenant un fichier audio **mp3** et transcodons le dans un autre format (et code), par exemple **ogg** et **vorbis.**

- **ogg** est un format de conteneur ouvert et gratuit. Il n'est pas limité par les brevets logiciels et est conçu pour fournir une diffusion et une manipulation efficaces du multimédia numérique de haute qualité.
- **vorbis** est une spécification de format audio et un codec pour la compression audio avec perte. Il est le plus couramment utilisé en conjonction avec le conteneur `.ogg`.

Tous les navigateurs Web modernes peuvent utiliser le conteneur `.ogg` et le décodeur **vorbis**.

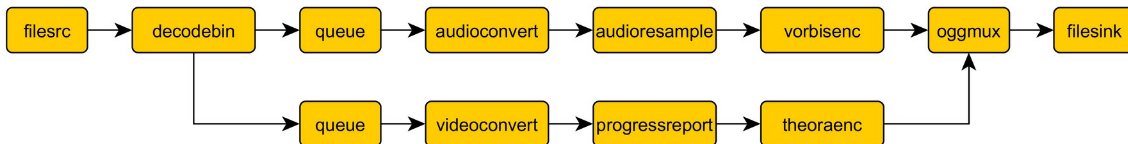
Le pipeline de transcodage à tester est:

```
gst-launch-1.0 -v filesrc location="../samples/music.mp3" ! decodebin !
audioconvert ! audioresample ! vorbisenc ! oggmux ! filesink location=music.ogg
```

Notez les étapes du pipeline:

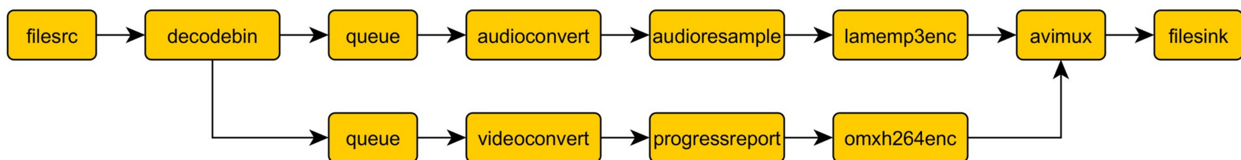
- **décodage** automatique par **`decodebin`** au format brut
- **`audioconvert`** - convertit les tampons audio bruts entre différents formats possibles. Il prend en charge la conversion d'entier en flottant, la conversion de largeur/profondeur, la conversion de signature et les transformations de canal.
- **`audioresample`** - vous pouvez fournir des paramètres de ré-échantillonnage - par exemple **`audio/x-raw, rate = 8000`**
- **`vorbisenc`** - codec vorbis ou lamemp3enc pour mp3
- **`oggmux`** - pour mettre la piste audio dans le conteneur

Dans le pipeline suivant, nous transcodons à la fois la piste audio et la piste vidéo. Nous utilisons le conteneur **`.ogv`**.
`.ogv` est un conteneur vidéo open-source pour la vidéo avec ou sans son:




```
gst-launch-1.0 -v filesrc location=tom.mp4" ! decodebin name=foo  foo. ! queue !
audioconvert ! audioresample ! vorbisenc ! oggmux name=bar  foo. ! queue !
videoconvert ! theoraenc ! bar. bar. ! filesink location="tom.ogv"
```

3.3.4 Trans-codage avec accélérateur matériel - VPU




```
gst-launch-1.0 -v filesrc location="../samples/tom.mp4" ! decodebin name=foo
foo. ! queue ! audioconvert ! audioresample ! lamemp3enc ! avimux name=bar  foo.
! queue ! videoconvert ! progressreport ! omxh264enc target-bitrate=1000000
control-rate=variable ! bar. bar. ! filesink location="tom.avi"
```

Affichage de **`progressreport`**:

```
New clock: GstSystemClock
progressreport0 (00:00:05): 14 / 596 seconds ( 2.3 %)
progressreport0 (00:00:10): 33 / 596 seconds ( 5.5 %)
progressreport0 (00:00:15): 52 / 596 seconds ( 8.7 %)
progressreport0 (00:00:20): 71 / 596 seconds (11.9 %)
..
```

3.4 Lecture des flux vidéo/audio de la webcam

Essayons maintenant de capturer les flux vidéo et audio d'une webcam. Lorsque nous connectons la webcam à un port USB, un nouveau périphérique vidéo, par défaut **`video0`**, est monté et répertorié dans le dossier **`/dev`**.
Notez que dans certains cas, comme dans un système utilisant de nombreuses webcams, il peut être monté sur un numéro de périphérique différent (**`video1`**, **`video2`**, ..).

Testez l'installation avec:

```
v4l2-ctl --list-devices
UVC Camera (046d:0819) (usb-0000:03:00.0-2):
      /dev/video0

v4l2-ctl --list-formats-ext
```

Testez les formats (codecs) disponibles:

```
bako@bako:~$ v4l2-ctl --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
      Index       : 0
      Type        : Video Capture
      Pixel Format: 'MJPG' (compressed)
      Name        : Motion-JPEG
            Size: Discrete 640x480
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)
            Size: Discrete 1280x720
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)
            Size: Discrete 1920x1080
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)

      Index       : 1
      Type        : Video Capture
      Pixel Format: 'YUYV'
      Name        : YUYV 4:2:2
            Size: Discrete 640x480
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)
            Size: Discrete 1280x720
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)

      Index       : 2
      Type        : Video Capture
      Pixel Format: 'H264' (compressed)
      Name        : H.264
            Size: Discrete 640x480
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)
            Size: Discrete 1280x720
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)
            Size: Discrete 1920x1080
                  Interval: Discrete 0.033s (30.000 fps)
                  Interval: Discrete 0.067s (15.000 fps)
                  Interval: Discrete 0.100s (10.000 fps)

      Index       : 3
      Type        : Video Capture
```

```
Pixel Format: '' (compressed)
Name          : 35363248-0000-0010-8000-00aa003
      Size: Discrete 640x480
            Interval: Discrete 0.033s (30.000 fps)
            Interval: Discrete 0.067s (15.000 fps)
            Interval: Discrete 0.100s (10.000 fps)
      Size: Discrete 1280x720
            Interval: Discrete 0.033s (30.000 fps)
            Interval: Discrete 0.067s (15.000 fps)
            Interval: Discrete 0.100s (10.000 fps)
      Size: Discrete 1920x1080
            Interval: Discrete 0.033s (30.000 fps)
            Interval: Discrete 0.067s (15.000 fps)
            Interval: Discrete 0.100s (10.000 fps)
```

3.4.1 Lecture des flux vidéo et leur affichage

Connaissant les paramètres de votre webcam, vous pouvez utiliser la capture **vidéo brute** ou la capture vidéo compressée au format `mjpeg` **ou** `h264`. Si vous travaillez avec **Logitech C270** fournissant à la fois un flux vidéo brut et un flux vidéo `mjpeg`. essayez d'exécuter un script simple comme celui-ci:

```
gst-launch-1.0 v4l2src device="/dev/video0" ! "video/x-raw, width=640,
height=480, format=(string)I420" ! xvimagesink -e

gst-launch-1.0 v4l2src ! image/jpeg,width=320,height=240,framerate=30/1 !
jpegdec ! videoconvert ! ximagesink

Si vous travaillez avec Logitech C920 qui compresse la vidéo également en h264
vous pouvez lancer ce pipeline:

gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264,
width=1920,height=1080, framerate=30/1 ! h264parse ! omxh264dec ! videoconvert !
xvimagesink
```

Pour le format `raw`:

```
gst-launch-1.0 v4l2src device=/dev/video0 ! videoconvert ! videoscale ! video/x-
raw,format=RGB ! queue ! videoconvert ! ximagesink
```

3.4.2 Lecture des flux vidéo et leur enregistrement

Le flux vidéo de la webcam peut être enregistré dans le fichier.
Par exemple:

```
gst-launch-1.0 v4l2src num-buffers=300 ! video/x-raw,
width=320,height=240,framerate=15/1 ! videoconvert ! progressreport ! jpegenc !
avimux ! filesink location=webcamconvjpeg.avi
```

3.4.3 Capture audio de la webcam avec microphone

Nous pouvons détecter la présence d'appareils de capture en regardant dans le fichier des **cards**:

```
cat /proc/asound/cards
```

La commande `arecord -l` nous permet de trouver les identifiants des appareils.

```
arecord -l
```

```
gst-launch-1.0 -v alsasrc device="default" num-buffers=1000 ! audio/x-raw,rate=16000,channels=1,width=16 ! queue  ! progressreport ! audioconvert ! audioamplify amplification=2.0 ! wavenc ! filesink location=webcam.wav
```

3.4.4 Lecture des flux vidéo à partir de la camera intégrée

Sur la carte Jetson X1 nous avons une **camera vidéo 4K** intégrée. Vous pouvez donc capter la vidéo avec un pipeline si-dessous :

Encodage en `h264` :

```
gst-launch-1.0 nvarguscamerasrc ! \
'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, \
format=(string)NV12, framerate=(fraction)30/1' ! nvv4l2h264enc \
maxperf-enable=1 bitrate=8000000 ! h264parse ! qtmux ! filesink \
location=mon_fichier_h264.mp4 -e
```

Encodage en `h265` :

```
gst-launch-1.0 nvarguscamerasrc ! \
'video/x-raw(memory:NVMM), width=(int)1920, height=(int)1080, \
format=(string)NV12, framerate=(fraction)30/1' ! nvv4l2h265enc \
bitrate=8000000 ! h265parse ! qtmux ! filesink \
location= mon_fichier_h265.mp4 -e
```

A tester !

Pour finir ce Lab essayez de construire quelques pipelines qui transcodent les contenus audio et vidéo en autres formats d'encodage, essayez d'exploiter au maximum les **codecs accélérés**.

Dans le Lab suivant - **Lab4** vous allez expérimenter avec le streaming vidéo (et audio) simple et un streaming enrichie par les transformations et incrustations des données.

Lab 4

Streaming avec Gstreamer-1.0

4.1 Protocoles utilisés

Pour effectuer le streaming, il est nécessaire d'envoyer des données par un premier terminal et de les recevoir par un deuxième. Pour ce faire il faut utiliser un protocole de communication mais pour un streaming de qualité il faut pouvoir s'assurer de perdre le moins de paquets de données que possible et de réduire le retard au maximum.

Configuration

Avant de commencer le développement des pipelines il est important de préparer un fichier de configuration dans lequel nous allons mettre l'adresse IP du `CLIENT`(récepteur)et les numéros de port **UDP/TCP**, **RTP-UDP**, **RTCP** et **RTCP_RET**.

```
CLIENT="adresse_IP_client"
PORT_UDPTCP_VIDEO=5000   #flux UDP/TCP video
PORT_RTP_VIDEO=5000   #flux RTP-UDP video
PORT_RTCP_VIDEO=5001  #flux RTCP video
PORT_UDPTCP_AUDIO=5002   #flux UDP/TCP   audio
PORT_RTP_AUDIO=5002   #flux RTP-UDP  audio
PORT_RTCP_AUDIO=5003  #flux RTCP audio
PORT_RTCP_VIDEO_RET=5005 #flux RTCP de retour video
PORT_RTCP_AUDIO_RET=5007 #flux RTCP de retour audio
```

Ce ficher , par exemple – `config.sh` doit être inséré dans nos pipelines par la commande source.

```
# fichier à insérer
source config.sh
# notre pipeline
```

4.1.1 UDP

L'UDP est le premier protocole utilisé, il est assez simple et permet d'envoyer des fichiers audios ou vidéos en IP ciblé, c'est à dire pour un utilisateur bien déterminé, en **multicast** pour plusieurs utilisateurs ou encore en **broadcast** ce qui permet à n'importe quel utilisateur de récupérer les données envoyées.

L'inconvénient principal est que la réception de tous les paquets de données n'est pas assurée. Il peut donc y avoir une perte importante de données et donc le streaming vidéo serait de mauvaise qualité.

Le protocole UDP s'utilise avec la fonction `udpsink` à laquelle, il faut ajouter une **adresse IP** de destination et un **port** lors de l'émission.
Le récepteur utilise `udpsrc` et ne déclare qu'un **port** sur lequel chercher les paquets, cependant ce port **doit correspondre au port de l'émetteur**.

Voici un exemple de lignes de commandes qui utilisent la transmission **UDP** (émission et réception) :

Emission :

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264, width=1920,
height=1080, framerate=30/1 ! udpsink host=$CLIENT port=$PORT_UDPTCP_VIDEO
```

Réception :

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_UDPTCP_VIDEO ! queue ! h264parse ! omxh264dec !
Videoconvert ! xvimagesink
```

4.1.2 TCP

Ensuite le protocole **TCP** a été testé, il est sûr et permet donc de récupérer toutes les données envoyés cependant il a pour défaut de créer des délais de retransmission.
Le protocole TCP, à l'émission utilise la fonction **`tcpserversink`** où il faut déclarer une **adresse IP** du serveur et un **port**.
Le récepteur utilise **`tcpclientsrc`** et doit déclarer l'**adresse IP** du serveur et un **port identique** à celui d l'émetteur (serveur).
Voici un exemple de lignes de commandes qui utilisent la **streaming sur TCP** :

Emission :

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264, width=1920,
height=1080, framerate=30/1 ! tcpserversink host=$CLIENT port=$PORT_UDPTCP_VIDEO
```

Réception :

```
source config.sh
gst-launch-1.0 tcpclientsrc host=$CLIENT port=$PORT_UDPTCP_VIDEO ! h264parse !
queue !omxh264dec ! videoconvert ! xvimagesink
```

4.1.3 RTP/UDP

Dans cette section nous allons utiliser le protocole **RTP**.

RTP ajoute un entête aux paquets UDP contenant des informations et notamment le numéro des paquets pour gérer la perte de données. Il a l'avantage par rapport au protocole TCP d'être en **temps réel** et donc de ne pas créer du délai supplémentaire.

En fonctionnant avec le format **H264** pour les échanges, le protocole doit s'adapter au format. A l'émission, on utilise la fonction **`rtph264pay`**, elle est **spécifique au format H264**, et définit les **caractéristiques du flux (capacités).**

Puis on utilise **`udpsink`** qui fonctionne comme en **UDP** car les paquets sont portés par les **datagrammes UDP**. Ensuite à la réception, on utilise **`udpsrc`** qui fonctionne comme en **UDP**.

Puis il faut ajouter les **caractéristiques du flux** que l'on souhaite recevoir avec **`caps`** et enfin il faut utiliser **`rtph264depay`** pour **faire correspondre** les caractéristiques avec le flux en entrée, on obtient alors la vidéo à la réception.

4.1.3.1 Emission et réception d'un flux vidéo (`h264`)

Voici un exemple de lignes de commandes (émetteur et récepteur) qui utilisent la transmission **RTP** :

Emission :

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264, width=1920,
height=1080, framerate=30/1 !rtph264pay ! udpsink host=$CLIENT
port=$PORT_RTP_VIDEO
```

Réception avec décodage hardware avec omx (accéléré) :

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_RTP_VIDEO caps="application/x-rtp,
media=(string)video, clock-rate=(int)90000, encoding-name=(string)H264,
encoding-params=(string)1" ! queue ! rtph264depay ! h264parse ! omxh264dec !
Queue ! videoconvert ! xvimagesink
```

Réception avec décodage software :

```
source config.sh
gst-launch-1.0 -v udpsrc port=$PORT_RTP_VIDEO caps = "application/x-rtp,
media=(string)video, clock-rate=(int)90000, encoding-name=(string)H264,
payload=(int)96" ! queue ! rtph264depay ! decodebin ! videoconvert ! ximagesink
```

Remarque : dans les deux cas le flux et mis en queue avant le **`depay`** (déchargement) et décodage.

4.1.3.2 Emission et réception d'un flux audio (SPEEX)

Emission :

```
source config.sh
gst-launch-1.0 -v alsasrc device=hw:2 ! audio/x-raw,rate=16000,channels=2 !
audioconvert ! speexenc ! queue ! rtpspeexpay ! udpsink host=$CLIENT
port=$PORT_RTP_AUDIO
```

Réception :

```
source config.sh
gst-launch-1.0 -v udpsrc caps="application/x-rtp,media=(string)audio,clock-
rate=(int)16000, encoding-name=(string)SPEEX, encoding-params=(string)1,
payload=(int)110"  port=$PORT_RTP_AUDIO ! queue ! rtpspeexdepay ! decodebin !
audioconvert ! alsasink
```

A faire :
Tester les pipelines ci-dessus avec vos adresses IP (avec connexion **Ethernet** puis **WiFi**)

4.1.4 RTCP/RTP/UDP

Enfin le dernier protocole à utiliser (ajouter) est le **RTCP**. **RTCP** s'ajoute au protocole **RTP** et permet un **retour sur les transferts des paquets**. s du **streaming**.
Le protocole **RTCP** possède des caractéristiques très proche du **RTP**. Le format du flux envoyé impacte les fonctions à utiliser en **RTCP** comme en **RTP**.

A l'émission, il faut commencer avec `rtph264pay`. Puis il faut créer une **session RTP** avec un lien sur un **envoi RTP**, utilisant `udpsink` et un deuxième lien sur un **envoi RTCP**, utilisant également `udpsink`, avec une adresse **IP de destination** qui **est la même** que pour l'envoi **RTP** et un port dont le numéro doit suivre celui de RTP. (par exemple **8000** pour **RTP** et **8001** pour **RTCP**)
A la réception, le fonctionnement est le même que pour **RTP** au départ puis on utilise `udpsrc` comme en UDP, on ajoute **les caractéristiques du flux** avec `caps` et enfin on termine avec une **session** pour la réception des **données RTCP**.

Voici un exemple de lignes de commandes qui utilisent la transmission **UDP**/**RTP**/**RTCP.**

Emission :

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-h264, width=1920,
height=1080, framerate=30/1 ! rtph264pay ! .send_rtp_sink rtpsession
name=session .send_rtp_src ! udpsink host=$CLIENT port=$PORT_RTP_VIDEO
session.send_rtcp_src ! udpsink host=$CLIENT port=$PORT_RTCP_VIDEO
```

Réception :

Décodage hardware avec **omx** (accéléré – **décodage hard**) :

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_RTP_VIDEO caps="application/x-rtp,
media=(string)video,clock-rate=(int)90000, encoding-name=(string)H264, encoding-
params=(string)1" ! .recv_rtp_sink rtpsession name=session .recv_rtp_src !
rtph264depay ! H264parse ! omxh264dec ! videoconvert ! xvimagesink udpsrc
port=$PORT_RTCP_VIDEO caps="application/x-rtcp" ! session.recv_rtcp_sink
```

et **décodage software** sans accélération (**décodage soft**):

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_RTP_VIDEO caps="application/x-rtp,
media=(string)video,clock-rate=(int)90000, encoding-name=(string)H264, encoding-
params=(string)1" ! .recv_rtp_sink rtpsession name=session .recv_rtp_src !
rtph264depay ! h264parse ! avdec_h264 ! queue ! videoconvert ! xvimagesink
udpsrc port=$PORT_RTCP_VIDEO caps="application/x-rtcp" ! session.recv_rtcp_sink
```

A faire :
Tester les pipelines ci-dessus avec vos adresses IP (tester avec une connexion **Ethernet** et **WiFi**)

4.1.5 Vidéo et audio en RTCP/RTP/UDP avec `rtpbin`

`rtpbin` **combine les fonctions** de **`GstRtpSession`**, **`GstRtpSsrcDemux`**, **`GstRtpJitterBuffer`** et **`GstRtpPtDemux`** en un seul élément. Il permet **plusieurs sessions RTP** , audio et vidéo, qui seront synchronisées ensemble à l'aide de paquets **`RTCP_SR`**.
`rtpbin` est configuré avec un certain nombre de champs de requête qui définissent la fonctionnalité activée, similaire à l'élément **`GstRtpSession`**.

Pour utiliser **`rtpbin`** comme récepteur **RTP**, demandez un **pad** **`recv_rtp_sink_%u`**. Le **numéro de session** (**%**) doit être spécifié dans le nom du **pad**.
Les données reçues sur le pad **`recv_rtp_sink_%u`** seront traitées dans le gestionnaire **`GstRtpSession`** et après avoir été validées transmises sur l'élément **`GstRtpSsrcDemux`**.

Chaque flux **RTP** est démultiplexé en fonction du **SSRC** et envoyé à un **`GstRtpJitterBuffer`**.
Une fois que les paquets sont libérés du **`jitterbuffer`**, ils seront transmis à un élément **`GstRtpPtDemux`**.
L'élément **`GstRtpPtDemux`** démultiplexera les paquets en fonction du type de charge utile et créera un **pad unique** **`recv_rtp_src_%u _%u`** sur **`rtpbin`** avec le numéro de session, le **SSRC** et le type de charge utile respectivement comme nom de pad.

Pour utiliser également **`rtpbin`** comme **récepteur RTCP**, demandez un **pad** **`recv_rtcp_sink_%u`**.
Le numéro de session doit être spécifié dans le nom du pad.

Si vous souhaitez que le gestionnaire de session génère et envoie des paquets RTCP, demandez le pad **`send_rtcp_src_% u`** avec le numéro de session dans le nom du pad.
Le paquet dirigé sur ce pad contient des rapports **SR**/**RR RTCP** qui doivent être envoyés à tous les participants à la session.

Pour utiliser **`GstRtpBin`** comme expéditeur, demandez un pad **`send_rtp_sink_%u`**, qui créera automatiquement un pad **`send_rtp_src_%u`**.

Si le numéro de session n'est pas fourni, le pad de la session disponible la plus basse sera retourné.
Le gestionnaire de session modifiera le **SSRC** des paquets **RTP** vers son propre **SSRC** et transmettra les paquets sur le pad **`send_rtp_src_%u`** après avoir mis à jour son état interne.

Dans l'exemple à suivre, pour une vidéo accompagnée du son, on reprend toute la partie vidéo à laquelle s'ajoute l'audio avec un schéma similaire. Il s'agit donc au final de **2 sessions rtcp distinctes** mises l'une après l'autre.

Chacune a une source, un encodeur, deux ports pour la transmission **RTCP** puis un décodeur et un **sink** pour la lecture. Le codec utilisé pour le **son** est **`speex`** avec l'encodeur **`speexenc`**.

On récupère la **source audio** de la webcam avec **`alsasrc`** avant l'émission et on utilise **`alsasink`** après la réception pour la lecture audio. La fonction **`rtpspeexpay`** est spécifique au format **`speex`** et permet de définir les caractéristiques du flux audio, il faut ensuite les récupérer et vérifier la correspondance avec **`rtpspeexdepay`**.

Emission avec `h264`:

La ligne de commandes suivante représente la transmission d'une vidéo contenant le flux vidéo (**`h264`**) et le son (**`speex`**).

```
source config.sh
gst-launch-1.0 rtpbin name=rtpbin \
v4l2src device=/dev/video2 ! video/x-h264, width=1920, height=1080,
framerate=30/1 ! rtph264pay ! rtpbin.send_rtp_sink_0 \
    rtpbin.send_rtp_src_0 ! udpsink host=$CLIENT port=$PORT_RTP_VIDEO  \
    rtpbin.send_rtcp_src_0 ! udpsink host=$CLIENT port=$PORT_RTCP_VIDEO \
    sync=false async=false    \
    udpsrc port=$PORT_RTCP_VIDEO_RET ! rtpbin.recv_rtcp_sink_0          \

alsasrc device=hw:2 ! audio/x-raw,rate=16000,channels=2 ! audioconvert !
speexenc ! queue !rtpspeexpay ! rtpbin.send_rtp_sink_1                 \
    rtpbin.send_rtp_src_1 ! udpsink host=$CLIENT port=$PORT_RTP_AUDIO   \
    rtpbin.send_rtcp_src_1 ! udpsink host=$CLIENT port=$PORT_RTCP_AUDIO \
    sync=false async=false    \
    udpsrc port=$PORT_RTCP_AUDIO_RET ! rtpbin.recv_rtcp_sink_1
```

Réception :
Avec le **décodage (`h264`) par software** :

```
gst-launch-1.0 -v rtpbin name=rtpbin  \
     udpsrc caps="application/x-rtp, media=(string)video,clock-rate=(int)90000,
encoding-name=(string)H264, encoding-params=(string)1" \
     port=5000 ! rtpbin.recv_rtp_sink_0  \
     rtpbin. ! queue ! rtph264depay ! h264parse ! avdec_h264 ! queue !
videoconvert ! xvimagesink \
      udpsrc port=5001 ! rtpbin.recv_rtcp_sink_0   \
      rtpbin.send_rtcp_src_0 ! udpsink port=5005 sync=false async=false   \
      udpsrc caps="application/x-rtp,media=(string)audio,clock-rate=(int)16000,
encoding-name=(string)SPEEX, encoding-params=(string)1, payload=(int)110" \
      port=5002 ! rtpbin.recv_rtp_sink_1  \
      rtpbin. !  queue ! rtpspeexdepay ! decodebin ! audioconvert ! alsasink \
      udpsrc port=5003 ! rtpbin.recv_rtcp_sink_1  \
      rtpbin.send_rtcp_src_1 ! udpsink port=5007 sync=false async=false
```

A faire

Préparer et tester le même pipeline avec un décodage accéléré (hard) par **`omxh264dec`**

Emission avec `jpeg`

La ligne de commandes suivante représente la transmission d'une vidéo contenant le flux vidéo (**jpeg**) et le son.

```
source config.sh
gst-launch-1.0 rtpbin name=rtpbin \
v4l2src device=/dev/video0 ! image/jpeg, width=640, height=480, framerate=30/1 !
  rtpjpegpay ! rtpbin.send_rtp_sink_0 \
  rtpbin.send_rtp_src_0 ! udpsink host=$CLIENT port=$PORT_RTP_VIDEO  \
  rtpbin.send_rtcp_src_0 ! udpsink host=$CLIENT port=$PORT_RTCP_VIDEO
    sync=false async=false    \
udpsrc port=$PORT_RTCP_VIDEO_RET ! rtpbin.recv_rtcp_sink_0           \

alsasrc device=device=hw:2  ! queue ! audioconvert ! Audioresample !
  audio/x-raw,  rate=16000, width=16, channels=1 ! speexenc !\
  rtpspeexpay ! rtpbin.send_rtp_sink_1                   \
  rtpbin.send_rtp_src_1 ! udpsink host=$CLIENT port=$PORT_RTP_AUDIO   \
  rtpbin.send_rtcp_src_1 ! udpsink host=$CLIENT port=$PORT_RTCP_AUDIO sync=false
  async=false    \
udpsrc port=$PORT_RTCP_AUDIO_RET ! rtpbin.recv_rtcp_sink_1
```

Réception avec décodage soft de jpeg

```
source config.sh
gst-launch-1.0 -v rtpbin name=rtpbin \
  udpsrc caps="application/x-rtp, media=(string)video,clock-rate=(int)90000,
             encoding-name=(string)JPEG, encoding-params=(string)1" \
    port=$PORT_RTP_VIDEO ! rtpbin.recv_rtp_sin  \
    rtpbin. !  rtpjpegdepay ! decodebin ! queue ! videoconvert ! Xvimagesink \
  udpsrc port=5001 ! rtpbin.recv_rtcp_sink_0  \
    rtpbin.send_rtcp_src_0 ! udpsink port=5005 sync=false async=false  \
  udpsrc caps="application/x-rtp, media=(string)audio, clock-rate=(int)16000,
      encoding-name=(string)SPEEX, encoding-params=(string)1, payload=(int)110"
    port=5002 ! rtpbin.recv_rtp_sink_1  \
    rtpbin. !  rtpspeexdepay ! decodebin ! audioconvert ! alsasink \
  udpsrc port=5003 ! rtpbin.recv_rtcp_sink_1  \
    rtpbin.send_rtcp_src_1 ! udpsink port=5007 sync=false async=false
```

A faire :

1. Analyser et tester les pipelines ci-dessus – combien de numéros de ports utilise-t-on et pourquoi ?
2. Effectuer un streaming d'une source enregistrée - un fichier de type **`tom.mp4/bunny.mp4`** ou **`tom.mkv/bunny.mkv`** ou **`tgtbtu.mkv`**.

4.2 Streaming simultané de plusieurs vidéos

La carte Jetson Nano est capable d'utiliser plusieurs caméras simultanément, il est donc possible de filmer et d'envoyer les différents flux vidéos séparément sur des canaux distincts. Cela permet ainsi de capturer plusieurs vidéos en même temps et d'envoyer chacun des flux vidéos à leurs destinataires bien précis.

L'objectif de l'exercice suivant est de filmer et d'envoyer simultanément 2 vidéos. Les sources vidéos sont les webcams Logitech C920 et une camera RPI V2 dont il faut préciser la localisation, ici ce sera **`/dev/video0`** et **`/dev/video1`**, elles fournissent directement un flux vidéo en H264 pour ne pas avoir à faire d'encodage pour la transmission, avec un format de 1920 x 1080 pixels et 30 images par seconde.

La première vidéo est envoyée par les ports 5000 et 5001 respectivement pour les protocoles RTP et RTCP et le retour RTCP s'effectue au port 5005. Pour la seconde vidéo, ce sont les ports 5002 et 5003 qui envoient la vidéo respectivement pour les protocoles RTP et RTCP et le retour RTCP s'effectue au port 5007. L'adresse du récepteur des 2 vidéos est la même, il s'agit du 172.19.65.68.

Maintenant, intéressons nous aux lignes de commandes. Dans un premier temps, un ensemble rtpbin est créé, la création de cet ensemble permet d'utiliser plusieurs sessions RTP en même temps. La caméra positionnée en **`video0`** est lue directement avec le codec H264. Puis on configure l'envoie RTP comme utilisant le codec H264 avec **`rtph264pay`**.
Après cela, on indique avec **`rtpbin.send_rtp_sink_0`** que les paquets à envoyer le seront sur la **session numéro 0**.
Puis, l'adresse de l'hôte et le port sont définis pour le RTP avec **`rtpbin.send_rtp_src_0`**.
Ensuite vient la configuration du protocole RTCP, l'adresse et le port sont définis pour l'envoi des informations avec **`rtpbin.send_rtcp_src_0`**.
Un second port est également déclaré pour réceptionner les paquets RTCP, cela fonctionne avec la fonction **`rtpbin.recv_rtcp_sink_0`** . Le même principe est utilisé pour la **session numéro 1** avec la camera en **`video1`**.
La ligne de commandes suivante représente la transmission simultanée de deux vidéos extrait de deux webcams sur deux ports différents.

La ligne de commandes suivante représente la transmission simultanée de deux vidéos extrait de deux webcams/camera sur deux ports différents

```
gst-launch-1.0 rtpbin name=rtpbin \
v4l2src device=/dev/video0 ! video/x-h264, width=1920, height=1080,
framerate=30/1 ! h264parse ! rtph264pay pt=96 ! rtpbin.send_rtp_sink_0 \
rtpbin.send_rtp_src_0 ! udpsink host=172.19.65.68 port=5000 \
rtpbin.send_rtcp_src_0 ! udpsink host=172.19.65.68 port=5001 sync=false
async=false \
udpsrc port=5005 ! rtpbin.recv_rtcp_sink_0 \
v4l2src device=/dev/video1 ! video/x-h264, width=1920,height=1080,
framerate=30/1 ! h264parse ! rtph264pay pt=96 ! rtpbin.send_rtp_sink_1 \
rtpbin.send_rtp_src_1 ! udpsink host=172.19.65.68 port=5002 \
rtpbin.send_rtcp_src_1 ! udpsink host=172.19.65.68 port=5003 sync=false
async=false \
udpsrc port=5007 ! rtpbin.recv_rtcp_sink_1
```

Après l'envoi, il faut s'occuper de la réception des deux vidéos. Les vidéos seront lues sur **deux ports différents**. Les protocoles devront être gérés pour correspondre avec les mêmes configurations qu'à l'envoi. Les vidéos seront ensuite décodées et lues.

Au niveau des lignes de commandes, elles commencent par la création d'un ensemble **`rtpbin`** pour la gestion de la réception de plusieurs sessions **RTP**. Puis on indique à la source **UDP** le type de fichier à recevoir, notamment le type de média et de format.
On définit également le port sur lequel les paquets transitent et on termine en indiquant que c'est une réception **RTP** sur la **session numéro 0** avec **`port=5000 ! rtpbin.recv_rtp_sink_0`**.

Ensuite la vidéo reçue est décodée et affichée. Enfin les paramètres du protocole RTCP sont déterminés pour correspondre aux paramètres d'envoi, comme les numéros de ports et l'adresse de retour à laquelle envoyer les paquets RTCP.

La ligne de commandes suivante représente la réception de deux vidéos simultanément

```
gst-launch-1.0 -v rtpbin name=rtpbin \
udpsrc caps="application/x-rtp,media=(string)video,clock-
rate=(int)90000,encoding-name=
(string)H264" \
port=5000 ! rtpbin.recv_rtp_sink_0 \
rtpbin. ! rtph264depay ! h264parse ! omxh264dec ! nvvidconv ! xvimagesink \
udpsrc port=5001 ! rtpbin.recv_rtcp_sink_0 \
rtpbin.send_rtcp_src_0 ! udpsink host=172.19.65.68 port=5005 sync=false
async=false \
udpsrc caps="application/x-
rtp,media=(string)video,clockrate=(int)90000,encoding-name=(string)H264" \
port=5002 ! rtpbin.recv_rtp_sink_1 \
rtpbin. ! rtph264depay ! h264parse ! omxh264dec ! nvvidconv ! xvimagesink \
udpsrc port=5003 ! rtpbin.recv_rtcp_sink_1 \
rtpbin.send_rtcp_src_1 ! udpsink host=172.19.65.68 port=5007 sync=false
async=false
```

4.3 Insertion de données dans le flux vidéo

L’une des problématiques de ce Laboratoire est de pouvoir insérer des images fixes dans le flux vidéo. C’est une pratique très répandue qui permet notamment d’afficher le logo de la chaîne qui diffuse le programme ou encore d’afficher de la publicité.
Pour insérer une image dans une vidéo, il est possible d’utiliser la fonction **gdkpixbufoverlay**.
Cette fonction travaille sur une **vidéo étant au format brut**, il est donc important de décoder les vidéos ou la sortie de la webcam. Cette fonction nécessite au minimum le paramètre **location** qui permet d’avoir le chemin relatif menant à la localisation de l’image à insérer dans la vidéo.
Des options sont disponibles pour améliorer cette fonction comme par exemple ajouter un offset sur x et y pour placer l’image à une position différente de la position par défaut sur la vidéo, ou encore modifier la taille de l’image sur la vidéo. L’insertion d’une image dans une vidéo au format brut ne l'empêche pas de pouvoir être encodée.

Pour commencer, il faut insérer une image dans une vidéo déjà existante. Il faut d’abord chercher la vidéo “tom.mp4” et la décoder avec decodebin pour obtenir la vidéo en format brut, ensuite on y insère l’image nommée “**scl.jpg**” avec **gdkpixbufoverlay** dans le coin en haut à gauche correspondant aux coordonnées **(x,y)=(0,0)**.
Enfin **videoconvert ! xvimagesink** va afficher la vidéo finale avec le lecteur adapté.

```
gst-launch-1.0 filesrc location=tom.mp4 ! decodebin ! queue ! gdkpixbufoverlay
location=scl.jpg ! videoconvert ! xvimagesink
```

Il faut maintenant pouvoir insérer une image dans une vidéo qui est filmée en direct. On va également utiliser d’autres attributs facultatifs de la fonction **gdkpixbufoverlay**, notamment “**offset-x**” et “**offset-y**” qui permettent de déplacer l’image respectivement sur les axes **x** et **y**.

Les attributs “**relative-x**” et “**relative-y**” quant à eux permettent également de déplacer l’image selon les axes x et y mais d’un pas égal aux dimensions de l’image.
Les attributs **overlay-height** et **overlay-width** servent à modifier la longueur et la largeur de l’image insérée.
L’attribut “**alpha**” permet de régler la **transparence** de l’image. Les valeurs sont comprises entre 0 et 1, à 1 l’image n’est pas du tout transparente et à 0 elle l’est complètement.

Il existe d’autres attributs disponibles dans des versions plus récentes de GStreamer.

Il faut maintenant filmer avec une webcam et insérer dans la vidéo l’image nommée **scl.jpg** avec **gdkpixbufoverlay**.
Ensuite la vidéo sera convertie à un format adapté au lecteur par **videoconvert** et affichée grâce au lecteur **xvimagesink**.

Si le format par défaut de la webcam est utilisé, ce sera un format brut, il n’est donc pas nécessaire de décoder le flux vidéo. Cependant, comme il est possible d’obtenir là un format différent du format brut, il faudra décoder le flux vidéo si c’est le cas.

```
gst-launch-1.0 v4l2src device=/dev/video0 ! gdkpixbufoverlay location=scl.jpg !
videoconvert ! xvimagesink
```

Voici un exemple de ligne de commandes utilisant des options de la fonction **gdkpixbufoverlay** sur le flux d’une webcam. La position de l’image est modifiée avec **offset-x** et **offset-y**, la taille de l’image sur la vidéo est modifiée avec **overlay-height** et **overlay-width** et la position de l’image est également déplacée d’une fois sa taille suivant **x** et **y** avec **relative-x=1 relative-y=1**.

Un autre exemple :

```
gst-launch-1.0 v4l2src device=/dev/video0 ! gdkpixbufoverlay location=scl1.jpg
offset-x=20 offset-y=20 ! videoconvert ! xvimagesink
```

Maintenant, il est intéressant de se pencher sur le **streaming d’une vidéo avec une image incrustée**.
Le principe de l’envoi est simple. La vidéo est récupérée et doit être au format brut, si ce n’est pas le cas elle doit être décodée. Ensuite sur la vidéo brut on vient insérer l’image désirée sur la vidéo avec les options choisies.
Pour l’instant, les lignes de commandes ne sont pas très différentes de celle vues précédemment. Maintenant, comme la vidéo doit être envoyée, il est préférable de l’encoder, ici le format choisi est **H264** avec la fonction hardware **omxh264enc**.

Une fois cela fait, il ne reste plus qu'à envoyer la vidéo à l'aide d'un protocole de transmission, ici le RTP et RTCP.
Pour la réception, il suffit simplement de faire un récepteur de vidéo au format H264 avec le protocole adapté avec celui utilisé pour l'envoi.
L'incrustation d'une image dans une vidéo ne change pas la perception de la vidéo par le système. C'est pourquoi la lecture et l'encodage ne sont pas impactés par l'insertion d'image.

La ligne de commandes suivante permet **l'envoi d'une vidéo** en format H264 avec une image incrustée.

```
gst-launch-1.0 -tv v4l2src device=/dev/video1 ! gdkpixbufoverlay
location=image.png offset-x=20 offset-y=20 ! omxh264enc bitrate=3000000
control-rate=2 ! h264parse ! queue ! rtph264pay config-interval=1 pt=96 !
.send_rtp_sink rtpsession name=session .send_rtp_src ! udpsink port=5000
host=172.19.65.68 session.send_rtcp_src ! udpsink port=5001 host=172.19.65.68
```

La ligne de commandes suivante permet la **réception vidéo** en format H264.

```
gst-launch-1.0 -tv udpsrc port=5000 caps="application/x-rtp,
media=(string)video,
clock-rate=(int)90000, encoding-name=(string)H264, encoding-params=(string)1,
payload=(int)26" ! .recv_rtp_sink rtpsession name=session .recv_rtp_src !
rtph264depay ! decodebin ! videoconvert ! ximagesink udpsrc port=5001
caps="application/x-rtcp" ! session.recv_rtcp_sink
```

Insérer une image fixe est bien mais il peut arriver qu'il soit nécessaire d'afficher simultanément plusieurs images fixes. En reprenant les deux exemples précédents, le logo d'une chaîne en principe reste tout au long de la diffusion du programme et la publicité s'ajoute à cela. Il faut donc maintenant pouvoir insérer plusieurs images dans le même flux vidéo.

Le code suivant permet de filmer avec une webcam et de mettre les images nommées **`image1.png`** et **`image2.png`** dans la vidéo respectivement aux positions x=20, y=20 et x=2180, y=20 avec une modification de taille de l'image 2.