# Internet et Multimedia Streaming avec les exercices sur Rock PI 5

## Contenu

# Internet et Multimedia Streaming
# avec les exercices sur Rock PI 5

## Introduction

Les laboratoires M1 (Internet et Multimédia) sont basées sur les cartes **Rock Pi 5 (Radxa)**. Cette plateforme embarquée avec le SoC **ARM/TPU – RK3588** est une de plus puissantes architectures embarquées pour le développement des systèmes multimédia et du streaming sur Internet.
Dans le premier laboratoire - **Lab 0** (**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 **Rock Pi 5 B (A)**.

**Lab 0 :** 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 1**. Les entrées/sorties multimédia et la capture et la compression des flux vidéo/audio, le « playout » et le transcodage des formats multimédia.
**Lab 2**. Le streaming vidéo, audio, vidéo/audio simple avec **UDP**, **TCP**, et **RTP et RTCP**
**Lab 3**. 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 Lab1 et Lab2.

Comme source d'information et des codes nous allons utiliser une préparation installée sur la carte et la documentation Gstreamer sur Rock Pi 5. Les codes de départ sont fournis dans un répertoire installé sur la carte.

## Carte Rock Pi 5 (B et A)

ROCK5 est la cinquième génération de SBC(Single Board Computer) designed by Radxa. Le coeur de la carte est le SoC de RockChip - **RK3588** . La carte peut exécuter Linux, Android, BSD et autres distributions.

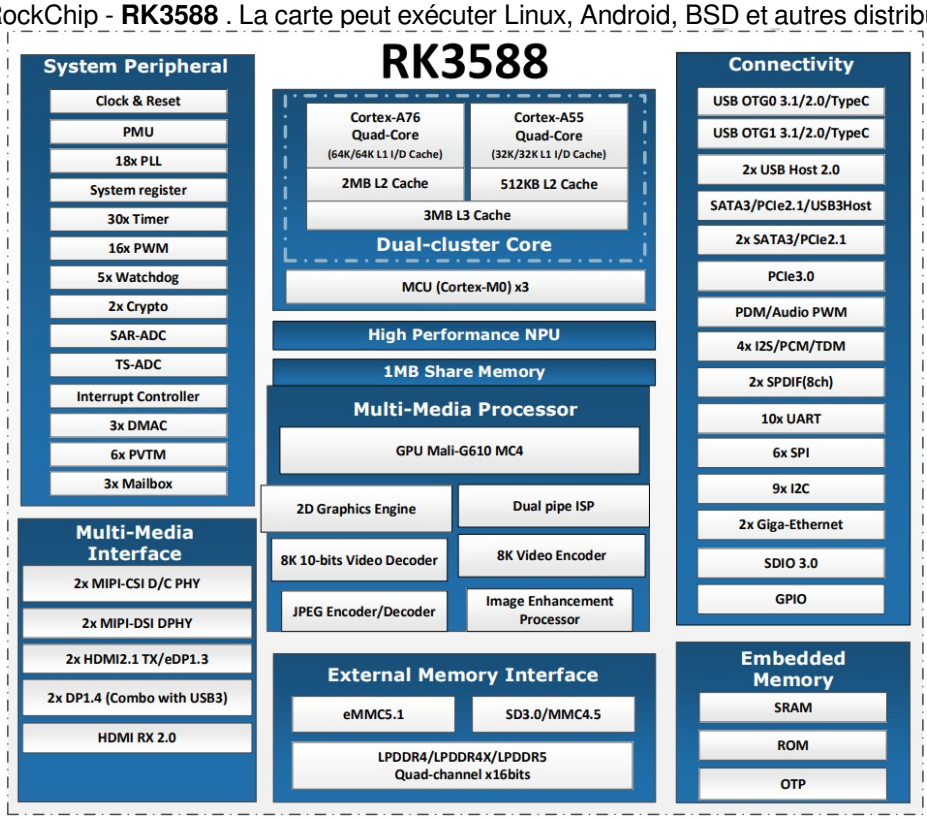



**Fig 0.1** Architecture de SoC RK3588

Rockchip RK3588(s) SoC est un processeur octa-core ARM (4x Cortex-A76 + 4x Cortex-A55), avec max 32GB 64bit LPDDR4X RAM, les interfaces 8K@60 HDMI, MIPI DSI, MIPI CSI, 3.5mm jack vec mic, ports USB, GbE LAN, PCIe 3.0 x4, PCIe 2.0, 40-pin GPIO header, RTC.

**Fig 0.2** Cartes – SBC : Rock Pi5A et RockPi5B

Les système d'exploitation installé est à la base de **Debian**.
Le **nom d'utilisateur** est **`rock`** et le **mot de passe** est **`rock`**.
La commande de base pour configurer le clavier en azerty est : **`setxkbmap fr`**

La carte Rock Pi5 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`** .

## Connexion WiFi

Avec l'utilitaire sur le Bureau/Desktop ou par les commandes.

```
sudo iwconfig
```

```
rock@rock-5b:~$ sudo iwconfig
lo        no wireless extensions.

enP4p65s0  no wireless extensions.

wlan0     unassociated  Nickname:"<WIFI@REALTEK>"
          Mode:Managed  Frequency=5.18 GHz  Access Point: Not-Associated
          Sensitivity:0/0
          Retry:off   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off
          Link Quality:0  Signal level:0  Noise level:0
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0

wlP2p33s0  IEEE 802.11AC  ESSID:"Livebox-08B0"  Nickname:"<WIFI@REALTEK>"
          Mode:Managed  Frequency:5.5 GHz  Access Point: 78:81:02:31:08:B5
          Bit Rate:867 Mb/s   Sensitivity:0/0
          Retry:off   RTS thr:off   Fragment thr:off
```

```
          Encryption key:****-****-****-****-****-****-****-****   Security mode:open
          Power Management:off
          Link Quality=57/100  Signal level=57/100  Noise level=0/100
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

```
rock@rock-5b:~$ sudo iwlist wlan0 scan | grep ESSID

                    ESSID:"VAIO-MQ35AL"
                    ESSID:"Livebox-08B0"
                    ESSID:"Livebox-08B0"
                    ESSID:"PIX-LINK-2.4G"
                    ESSID:"DIRECT-G8M2070 Series"
                    ESSID:"freebox_DZHECV"
                    ESSID:"PIX-LINK-5.8G"
                    ESSID:""
```

Après le choix du point d'accès, connectez vous par la commande – **n**etwork **m**anager **cli**ent :

```
nmcli d wifi connect votre_AP password votre_password
```

**La documentation et les laboratoires (codes) peuvent être téléchargées à partir de :**

**github.com/smartcomputerlab**

**Après le téléchargement placez le fichier `.zip` dans le répertoire de destination, par exemple `internet_et_multimedia` crée par la séquence :**

```
cd
mkdir  internet_et_multimedia
cd internet_et_multimedia
cp ~/Downloads/fichier.zip .
unzip fichier.zip
```

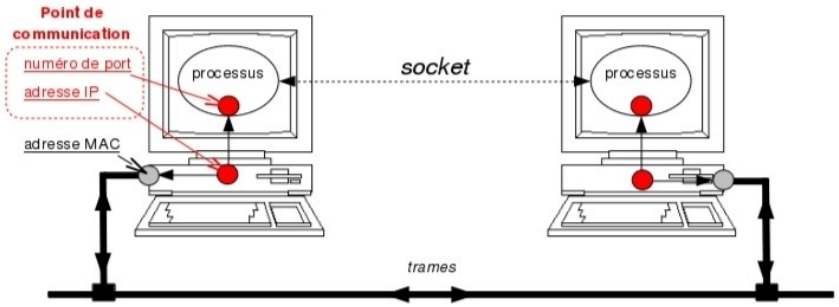
# Lab 0

## 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`**

## 0.1 Les fonctionnalités de base de l'API « socket »

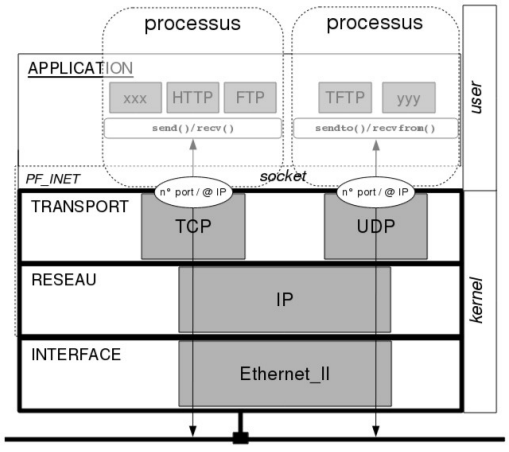



Une socket est communément représentée comme un point d'entrée initial au niveau **transport** du modèle à couches TCP/IP.
La couche **transport** est responsable du transport des messages complets de bout en bout au travers du réseau. En programmation, si on utilise comme point d'entrée initial le niveau **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



## 0.2 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** ).

### 0.2.1 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;
}
```

### 0.2.2 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()`**.

#### 0.2.2.1 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;
}
```

#### 0.2.2.2 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).

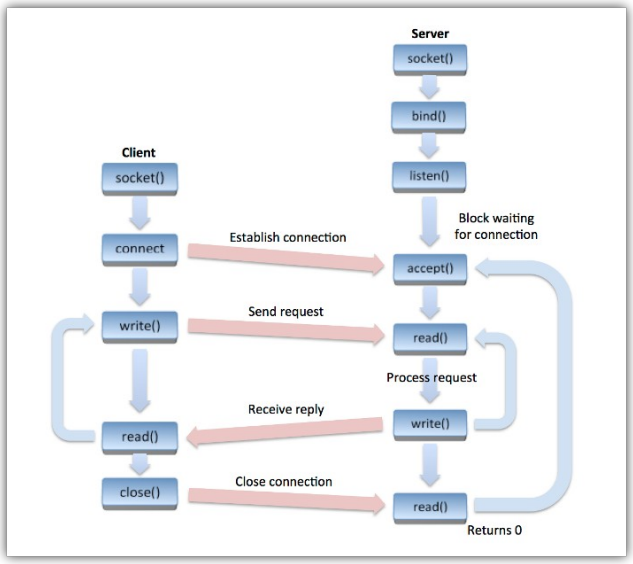
**Rappel :**

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
```

## 0.3 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()`**.




### 0.3.1 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);
}
```

### 0.3.2 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 1

## 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/vidéo.
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 !**

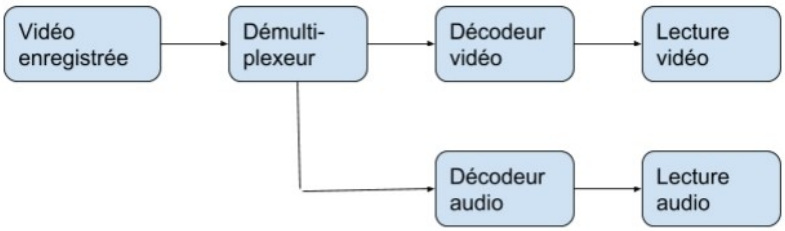
## 1.1 Compression, décompression vidéo

### 1.1.1 Fonctionnement de la compression et de la décompression vidéo

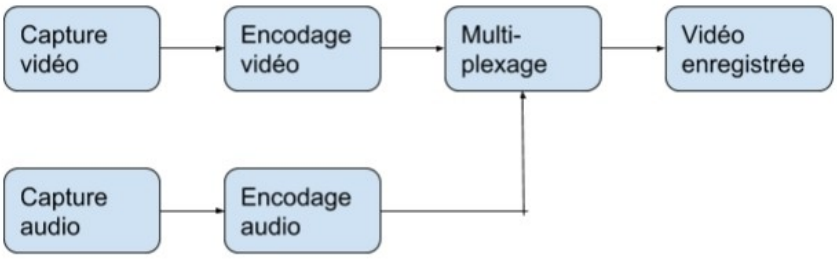
Avant de pouvoir utiliser les commandes Gstreamer, il faut d'abord se pencher sur les étapes de la compression et décompression vidéo qu'il faudra détailler lors de l'utilisation de Gstreamer.
Deux cas sont mis en avant car utilisés lors du projet technique.

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**.

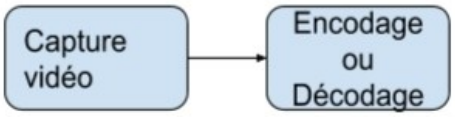



Le deuxième cas est l'utilisation de la vidéo uniquement en sortie d'une **webcam**. La webcam peut fournir

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.




### 1.1.2 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 vidéo 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. (exemple: **`/dev/video1`**)
- **`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. (exemples : **`device="default"`** ou **`device=hv:4`** .

### 1.1.3 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`**.

### 1.1.4 Encodeurs et décodeurs

Les encodeurs et décodeurs tels que **`x264enc`** ou **`avdec_h264`** peuvent être utilisés mais ils fonctionnent moins vite.

Le décodeur vidéo accéléré est **`avdec_264_omx`** 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.

### 1.1.5 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

### Attention Gstreamer est déjà installé sur votre carte !

### 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
```

## 1.3 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 uniquement la partie **audio**. Commencez 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 samples/tom.mp4
tom.mp4 - video/quicktime, variant=(string)iso

gst-typefind-1.0 samples/music.mp3
samples/music.mp3 - audio/mpeg, mpegversion=(int)1, layer=(int)3, parsed=(boolean)false
```

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

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

rock@rock-5b:~$  gst-inspect-1.0 | grep mp4
isomp4:  3gppmux: 3GPP Muxer
isomp4:  ismlmux: ISML Muxer
isomp4:  mj2mux: MJ2 Muxer
isomp4:  mp4mux: MP4 Muxer
isomp4:  qtdemux: QuickTime demuxer
isomp4:  qtmoovrecover: QT Moov Recover
isomp4:  qtmux: QuickTime Muxer
isomp4:  rtpxqtdepay: RTP packet depayloader
libav:  avmux_mp4: libav MP4 (MPEG-4 Part 14) muxer (not recommended, use mp4mux instead)
rtp:  rtpmp4adepay: RTP MPEG4 audio depayloader
rtp:  rtpmp4apay: RTP MPEG4 audio payloader
rtp:  rtpmp4gdepay: RTP MPEG4 ES depayloader
rtp:  rtpmp4gpay: RTP MPEG4 ES payloader
rtp:  rtpmp4vdepay: RTP MPEG4 video depayloader
rtp:  rtpmp4vpay: RTP MPEG4 Video payloader
typefindfunctions: video/quicktime: mov, mp4
```

```
$ gst-inspect-1.0 | grep h264

libav:  avdec_h264: libav H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10 decoder
libav:  avenc_h264_omx: libav OpenMAX IL H.264 video encoder encoder
openh264:  openh264dec: OpenH264 video decoder
openh264:  openh264enc: OpenH264 video encoder
rtp:  rtph264depay: RTP H264 depayloader
rtp:  rtph264pay: RTP H264 payloader
typefindfunctions: video/x-h264: h264, x264, 264
uvch264:  uvch264deviceprovider (GstDeviceProviderFactory)
uvch264:  uvch264mjpgdemux: UVC H264 MJPG Demuxer
uvch264:  uvch264src: UVC H264 Source
videoparsersbad:  h264parse: H.264 parser
```

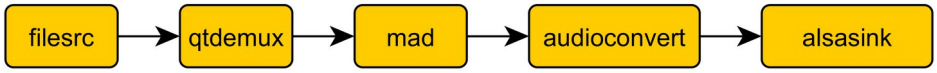
```
rock@rock-5b:~$ gst-inspect-1.0 | grep mp3

lame:  lamemp3enc: L.A.M.E. mp3 encoder
libav:  avdec_mp3: libav MP3 (MPEG audio layer 3) decoder
libav:  avdec_mp3adu: libav ADU (Application Data Unit) MP3 (MPEG audio layer 3) decoder
libav:  avdec_mp3adufloat: libav ADU (Application Data Unit) MP3 (MPEG audio layer 3) decoder
libav:  avdec_mp3float: libav MP3 (MPEG audio layer 3) decoder
libav:  avdec_mp3on4: libav MP3onMP4 decoder
libav:  avdec_mp3on4float: libav MP3onMP4 decoder
libav:  avmux_mp3: libav MP3 (MPEG audio layer 3) formatter (not recommended, use id3v2mux instead)
mpg123:  mpg123audiodec: mpg123 mp3 decoder
typefindfunctions: application/x-apetag: mp3, ape, mpc, wv
typefindfunctions: application/x-id3v1: mp3, mp2, mp1, mpga, ogg, flac, tta
typefindfunctions: application/x-id3v2: mp3, mp2, mp1, mpga, ogg, flac, tta
typefindfunctions: audio/mpeg: mp3, mp2, mp1, mpga
```

### A faire:

Lancer les commandes ci-dessus et analyser les réponses.

### 1.3.1 Quelques exemples de pipelines : avec `avdec_mp3` et `avdec_h264`

Voici quelques exemples (à tester) de ligne de commandes (pipelines Gstreamer) pour la lecture d'une audio/vidéo **`mp3`**/**`mp4`**.
**Attention** : Branchez un écouteur sur la **sortie audio de la carte** (prise jack).




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

WARNING: erroneous pipeline: no element "mad"
gst-launch-1.0 filesrc location=../samples/tom.mp4 ! qtdemux name=demux demux.audio_0 ! queue !
mpegaudioparse !  avdec_mp3 ! audioconvert ! alsasink
```

**A faire :** ajouter un composant **`mpegaudioparse`** et changer le composant – decoder → **`avdec_mp3`**

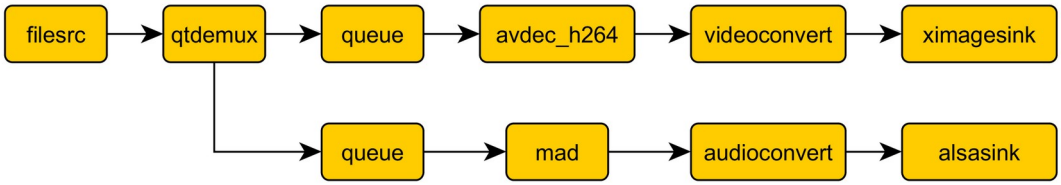
```
gst-launch-1.0 filesrc location=../samples/tom.mp4 ! qtdemux name=demux demux.audio_0 ! queue !
mpegaudioparse !  avdec_mp3 ! audioconvert ! alsasink
```


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

WARNING: erroneous pipeline: no element "omxh264dec"
```

**A faire**
Changer le composant – decoder → **`omxh264dec → avdec_h264`**

```
gst-launch-1.0 filesrc location=../samples/tom.mp4 ! qtdemux name=demux demux.video_0 ! queue !
h264parse ! avdec_h264 ! videoconvert ! ximagesink
```

Les noms probables des **pistes** pour le **`qtdemux`** sont : **`audio_0`** et **`video_0`**, nous pouvons essayer de les traiter séparément. Voici le pipeline (double) , avec les **modifications** à faire.




```
gst-launch-1.0 filesrc location=../samples/tom.mp4 ! qtdemux name=demux \
        demux.audio_0 ! queue ! mpegaudioparse !  avdec_mp3 ! audioconvert ! alsasink \
        demux.video_0 ! queue ! h264parse ! avdec_h264 ! videoconvert ! ximagesink
```

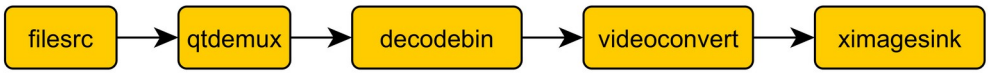
### Remarque :

Dans le pipeline ci dessus nous avons inséré le composant **`queue`** , il est nécessaire pour séparer deux processus avec les même priorités d'exécution ; un pour la partie audio et un pour la partie vidéo.

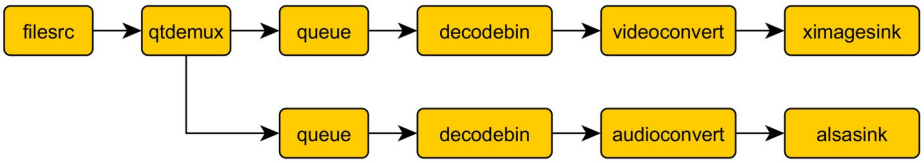
### 1.3.2 Un moyen simple de décoder les flux vidéo et audio avec `decodebin`

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=../samples/tom.mp4 ! decodebin ! videoconvert !
ximagesink
```




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

### 1.3.3 Lire et encoder un fichier audio/vidéo dans un autre format

Pour commencer lions un fichier audio simple type mp3 :Notons que terme **mp3** dénote à la fois le **codec** et le **conteneur.**

```
gst-launch-1.0 location=../samples/$.mp3 ! mpegaudioparse ! avdec_mp3 ! audioconvert ! autoaudiosink
```

Vous pouvez le lancer comme suit :

```
$audioplay_mp3.sh music
```

**A faire :** éditez le même pipeline avec **decodebin**

Testez également l'exemple suivant permet de transcoder la source vidéo/audio de test en fichier mp4.

```
gst-launch-1.0 -v -e videotestsrc \
  ! openh264enc \
  ! h264parse \
  ! mp4mux name=mux \
  ! filesink location="../samples/bla.mp4" audiotestsrc \
  ! lamemp3enc \
  ! mux.
```

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=../samples/music.ogg
```

Notez les étapes du pipeline:

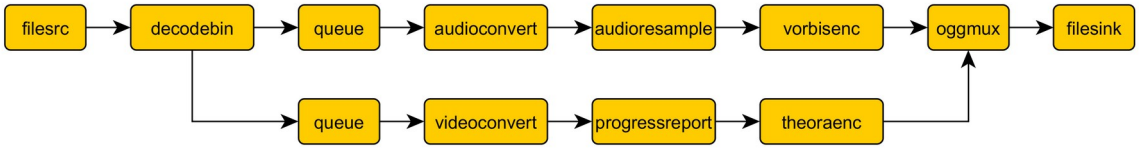
- **décodage** automatique par **decodebin** au format brut
- **audioconvert** - convertit les tampons audio bruts entre différents formats possibles. Il prend en charge la conversion d'entier en flottant, la conversion de largeur/profondeur, la conversion de signature et les transformations de canal.
- **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

## A faire

Utilisez votre pipeline précédent avec **decodebin** pour jouer le fichier **music.ogg**

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=../samples/tom.mp4 ! decodebin name=foo  foo. ! queue !
audioconvert ! audioresample ! vorbisenc ! oggmux name=bar  foo. ! queue ! videoconvert !
theoraenc ! bar. bar. ! filesink location=../samples/tom.ogv
```

L'exemple suivant montre un pipeline qui transcode le fichier .mp4 en format .mkv avec une datarate de 1Mb/s :

```
gst-launch-1.0 -v filesrc location="../samples/tom.mp4" ! qtdemux name=demux  demux.audio_0 !
queue ! decodebin ! audioconvert ! audioresample ! lamemp3enc ! matroskamux name=bar
demux.video_0 ! queue ! decodebin ! videoconvert ! progressreport ! openh264enc bitrate=1000000 !
h264parse ! bar. bar. ! filesink location="../samples/tom.mkv"
```

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 %)
..
```

## A faire

Testez les exemples ci dessus en changeant le taux de compression.

## 1.4 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 **`video1`**, est monté et répertorié dans le dossier **`/dev`**.
Notez que dans certains cas, comme dans un système utilisant de nombreuses webcams (**C902**), il peut être monté sur un numéro de périphérique différent (**`video1`**, **`video2`**, ..).
Testez l'installation avec:

```
rock@rock-5b:~/gstreamer/lab1$ v4l2-ctl --list-devices
rk_hdmirx (fdee0000.hdmirx-controller):
        /dev/video0
HD Pro Webcam C920 (usb-fc880000.usb-1):
        /dev/video1
        /dev/video2
        /dev/media0
```

Testez les formats (codecs) disponibles:

```
rock@rock-5b:~$ v4l2-ctl --device=/dev/video1 --list-formats
ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture
        [0]: 'YUYV' (YUYV 4:2:2)
        [1]: 'H264' (H.264, compressed)
        [2]: 'MJPG' (Motion-JPEG, compressed)
```

### 1.4.1 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 C920** fournissant à la fois un flux vidéo brut (**`raw`**), un flux vidéo **`mjpeg`** et un flux **`h264`**.essayez d'exécuter les pipelines simple comme suit:


```
gst-launch-1.0 v4l2src device=/dev/video1 ! image/jpeg, width=1280,height=720, framerate=10/1 !
jpegdec ! videoconvert ! xvimagesink
```

```
gst-launch-1.0 v4l2src device=/dev/video1 ! video/x-h264, width=1280,height=720, framerate=30/1 !
h264parse ! openh264dec ! videoconvert ! xvimagesink
```

Pour le format **`raw`**:

```
gst-launch-1.0 v4l2src device=/dev/video1 ! videoconvert ! videoscale ! video/x-raw, hight=480,
width=640, format=RGB ! queue ! videoconvert ! ximagesink
```

### 1.4.2 Lecture des flux vidéo et leur enregistrement

Le flux vidéo de la webcam peut être enregistré dans le fichier.
Par exemple, une lecture en mode raw avec la compression en **`jpeg`** et enregistrement en format **`.avi`**.


```
gst-launch-1.0 v4l2src device=/dev/video1 num-buffers=300 ! video/x-raw,
width=320,height=240,framerate=15/1 ! videoconvert ! progressreport ! jpegenc ! avimux ! filesink
location=../samples/webcamconvjpeg.avi
```

L'exemple suivant va lire directement le flux **`h264`** et l'enregistrer sans compression dans un fichier en format **`.mkv`**.

```
gst-launch-1.0 -v v4l2src device=/dev/video1 ! queue ! video/x-h264,width=1280,height=720,
framerate=30/1 ! h264parse ! queue ! matroskamux ! filesink
location='../samples/webcam.video_h264.mkv'
```

Pour lire le fichier multimedia utilisez le pipeline : **`gstplay.sh :`**

```
echo "gstplay.sh file_name"
gst-play-1.0 $1
```

### 1.4.3 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
rock@rock-5b:~/gstreamer/lab1$ cat /proc/asound/cards
 0 [rockchiphdmi0  ]: rockchip-hdmi0 - rockchip-hdmi0
                      rockchip-hdmi0
 1 [rockchiphdmi1  ]: rockchip-hdmi1 - rockchip-hdmi1
                      rockchip-hdmi1
 2 [rockchipes8316 ]: rockchip-es8316 - rockchip-es8316
                      rockchip-es8316
 3 [rockchiphdmiin ]: rockchip_hdmiin - rockchip,hdmiin
                      rockchip,hdmiin
 4 [C920           ]: USB-Audio - HD Pro Webcam C920
                      HD Pro Webcam C920 at usb-fc880000.usb-1, high speed
```

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

```
arecord -l

**** List of CAPTURE Hardware Devices ****
card 2: rockchipes8316 [rockchip-es8316], device 0: fe470000.i2s-ES8316 HiFi es8316.7-0011-0
[fe470000.i2s-ES8316 HiFi es8316.7-0011-0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 3: rockchiphdmiin [rockchip,hdmiin], device 0: fddf8000.i2s-dummy_codec hdmiin-dc-0
[fddf8000.i2s-dummy_codec hdmiin-dc-0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 4: C920 [HD Pro Webcam C920], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

**Attention** – pour la WebCam device :
**`device="default"`** ou **`device=hw:4`**


Lecture en mono du flux audio **`raw`** et son enregistrement en format **`wav`**.

```
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=../samples/webcam.wav
```

Lecture en stéréo du flux audio **`raw`** et son enregistrement en format **`mp3`**.

```
gst-launch-1.0 -v alsasrc device="default" num-buffers=1000 ! audio/x-raw,channels=2 ,rate=16000 !
audioconvert ! lamemp3enc ! filesink location='../samples/webcam.output_mp3.mp3'
```

### 1.4.4 Lecture audio/vidéo sur la webcam dans un fichier

Voici un pipeline pour lecture de la vidéo en mode **`x-h264`** et audio `x-raw` avec compression mp3 dans un fichier **`.mkv`** :

```
gst-launch-1.0 -v v4l2src device=/dev/video1 ! queue ! video/x-
h264,width=1280,height=720,framerate=30/1 ! h264parse ! queue ! mux. \
        alsasrc device="default" ! audio/x-raw,channels=2,rate=16000 ! audioconvert ! queue !
lamemp3enc ! queue ! mux. matroskamux name=mux ! queue ! filesink
location='../samples/webcammux.mkv'
```

## A faire

1. Enregistrez la **vidéo** de Webcam avec les codecs **theoraenc** et **openh264enc** en format de conteneur **`.ogg`** et **`.mkv`**, et **`mp4`**.
2. Changez la **résolution** en **1280x720** puis en **680x480** (VGA).

# Lab 2

## Gstreamer-1.0 , streaming

## 2.0 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
```

## 2.1 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) :

### 2.1.1 Emission/réception vidéo à partir d'une Webcam

#### 2.1.1.1 Emission

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video1 ! video/x-h264, width=1280, height=720, framerate=30/1 !
udpsink host=$CLIENT port=$PORT_UDPTCP_VIDEO
```

#### 2.2.1.2 Réception :

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_UDPTCP_VIDEO ! queue ! h264parse ! openh264dec ! videoconvert !
xvimagesink
```

**Attention** :
Ces pipelines peuvent fonctionner localement sur le support « rapide » type **ethernet**.

## 2.2 TCP

**TCP** est un protocole sûr et il 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**

### Attention

Le streaming sur TCP est faisable seulement sur un réseau local sans la perte des données – sans la retransmission):

## 2.2.1 Emission/réception vidéo à partir d'une Webcam

### 2.2.1.1a Emission (serveur à lancer en premier!)

Emission à partir d'une source en mode **raw** avec une compression par **jpegenc** :

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video1 ! videoconvert ! videoscale ! video/x-raw, width=640,
height=480, format=RGB ! queue ! videoconvert ! jpegenc ! gdppay ! tcpserversink host=192.168.1.52
port=9009 sync=false
```

### 2.2.1.1b Emission avec la compression dans Webcam

```
source config.sh
gst-launch-1.0 v4l2src device=/dev/video1 ! image/jpeg, width=640, height=480, framerate=10/1 !
gdppay ! tcpserversink host=192.168.1.52 port=9009 sync=false
```

### 2.2.1.2 Réception (client)

```
source config.sh
gst-launch-1.0 tcpclientsrc host=192.168.1.52 port=9009 ! gdpdepay ! jpegdec ! videoconvert !
xvimagesink
```

**Attention** :
TCP utilise de **grands *buffers*** et le temps de réception est largement retardé, par exemple **15 secondes**.
La communication fonctionne avec une connexion à la fois !

En conséquence ce type de streaming ne peut pas être utilisé en communication **temps-réel**, mais il peut servir pour la visualisation d'un contenu multimédia enregistré (par exemple un film).

## 2.3 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- cap )**
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.

### 2.3.1 Emission/réception vidéo à partir d'une Webcam (h264)

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

#### 2.3.1.1a Emission d'un flux h264 à partir d'une Webcam

```
source config.sh
CLIENT_IP="192.168.1.62"
gst-launch-1.0 -v v4l2src device=/dev/video1 ! video/x-h264, width=1920,height=1080,framerate=30/1 !
\
h264parse ! rtph264pay mtu=1400 ! udpsink host=$CLIENT_IP port=8050 sync=false async=false
```

#### 2.3.1.1b Emission du flux vidéo – de l'écran (h264)

```
gst-launch-1.0 -v ximagesrc ! video/x-raw,framerate=20/1 ! videoscale ! videoconvert ! openh264enc !
rtph264pay ! udpsink host=192.168.1.27  port=8050
```

#### 2.3.1.2 Réception d'un flux h264

```
source config.sh
gst-launch-1.0 udpsrc port=8050 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000,
encoding-name=(string)H264, encoding-params=(string)1" ! \
rtph264depay ! h264parse ! avdec_h264 ! queue ! videoconvert ! xvimagesink
```

**Remarque :** Le flux et mis en **queue** avant le **depay** (déchargement) et décodage.

### 2.3.2 Emission/réception vidéo à partir d'une Webcam (jpeg)

#### 2.3.2.1 Emission du flux compressé par la Webcam (jpeg)

```
CLIENT_IP="192.168.1.62"
gst-launch-1.0 -v v4l2src device=/dev/video1 ! image/jpeg,width=640,height=480,framerate=30/1 ! \
rtpjpegpay mtu=1400 ! udpsink host=$CLIENT_IP port=8050 sync=false async=false
```

#### 2.3.2.2 Réception (jpeg)

```
gst-launch-1.0 udpsrc port=8050 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000,
encoding-name=(string)jpeg, encoding-params=(string)1" ! \
rtpjpegdepay ! jpegdec ! queue ! videoconvert ! xvimagesink
```

### 2.3.3 Emission/réception vidéo à partir d'un fichier (h264)

#### 2.3.3.1 Emission

Le fichier **$1 (tom.mp4, tgtbtu.mkv)** est décodé , puis encodé pour l'émission. Notez que le flux est synchrone : **sync=true**.

```
CLIENT_IP="192.168.1.62"
gst-launch-1.0 -v filesrc location=../samples/$1 ! decodebin ! openh264enc ! rtph264pay ! udpsink
host=$CLIENT_IP port=8050 sync=true
```

#### 2.3.3.2 Réception

```
gst-launch-1.0 -v udpsrc port=8050 caps = "application/x-rtp, media=(string)video, clock-
rate=(int)90000, encoding-name=(string)H264, payload=(int)96" ! rtph264depay ! decodebin !
videoconvert ! autovideosink
```

## 2.3.4 Emission et réception d'un flux audio (SPEEX)

### 2.3.4.1 Emission

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

### 2.3.4.2 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
```

## 2.3.5 Emission et réception d'un flux audio (ac3)

### 2.3.5.1 Emission

```
gst-launch-1.0 -v alsasrc device="default" ! audio/x-raw,rate=16000,channels=2 ! audioconvert !
audioresample ! avenc_ac3 ! rtpac3pay  ! udpsink host=192.168.1.62 port=8060
```

### 2.3.5.2 Réception

```
gst-launch-1.0 -v udpsrc port=8060 caps='application/x-rtp' ! rtpac3depay ! decodebin ! audioconvert
! alsasink sync=false
```

## 2.3.6 Emission et réception d'un flux video (h264) et audio (speex)

Finalement nous avons un mix du flux **vidéo et audio.**

### 2.3.6.1 Emission

```
CLIENT="192.168.1.62"
gst-launch-1.0  v4l2src device=/dev/video1 ! video/x-h264, width=1280, height=720, framerate=30/1 !
rtph264pay ! udpsink host=$CLIENT port=9002 \
  alsasrc device="default" ! audio/x-raw,rate=16000,channels=2 ! audioconvert ! speexenc ! queue !
rtpspeexpay ! udpsink host=$CLIENT port=9004
```

### 2.3.6.2 Réception

```
gst-launch-1.0 udpsrc port=9002 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000,
encoding-name=(string)H264, encoding-params=(string)1" ! \
rtph264depay ! h264parse ! avdec_h264 ! queue ! videoconvert ! xvimagesink \
udpsrc caps="application/x-rtp,media=(string)audio,clock-rate=(int)16000, encoding-
name=(string)SPEEX, encoding-params=(string)1, payload=(int)110"  port=9004 ! queue !
rtpspeexdepay ! decodebin ! audioconvert ! Alsasink
```

## 2.3.7 Emission et réception d'un flux vidéo (h264) Webcam+écran

### 2.3.7.1 Emission avec incrustation de la capture Webcam

```
HOST=192.168.1.27
gst-launch-1.0 v4l2src device=/dev/video1 ! image/jpeg, width=320,height=240, framerate=10/1 ! queue
! jpegdec ! videoconvert ! xvimagesink \
ximagesrc ! video/x-raw,framerate=20/1 ! queue ! videoscale ! videoconvert ! openh264enc !
rtph264pay ! udpsink host=$HOST port=5005
```

### 2.3.7.2 Réception du flux incrusté

```
gst-launch-1.0 udpsrc port=5005 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000,
encoding-name=(string)H264, encoding-params=(string)1" ! \
rtph264depay ! h264parse ! avdec_h264 ! queue ! videoconvert ! xvimagesink
```

### A faire

1. **Tester** les pipelines avec **UDP-RTP**.
2. **Modifier** le dernier exemple en ajoutant un flux audio - speex
3. **Ecrire** un pipeline pour le streaming du vidéo et de l'audio d'un fichier type **.mp4** ou **.mkv**

## 2.4 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**. 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 **8050** pour **RTP** et **8051** 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.**

### 2.4.1 Emission/réception vidéo avec une rtpsession à partir d'une Webcam (h264)

#### 2.4.1.1 Emission

```
gst-launch-1.0 v4l2src device=/dev/video1 ! video/x-h264, width=1920, height=1080, framerate=30/1 !
rtph264pay !  .send_rtp_sink rtpsession name=session .send_rtp_src ! udpsink host=192.168.1.27
port=8050 session.send_rtcp_src ! udpsink host=192.168.1.27 port=8051
```

#### 2.4.1.2 Réception

```
source config.sh
gst-launch-1.0 udpsrc port=$PORT_RTP_VIDEO caps="application/x-rtp, media=(string)video,clock-
rate=(int)90000, encoding-name=(string)H264, encoding-params=(string)1" ! .recv_rtp_sink rtpsession
name=session .recv_rtp_src ! rtph264depay ! h264parse ! openh264dec ! queue ! videoconvert !
xvimagesink udpsrc port=$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**)

### 2.4.2 Vidéo et audio en RTCP/RTP/UDP avec rtpbin

**rtpbin combine les fonctions** de **GstRtpSession**, **GstRtpSsrcDemux**, **GstRtpJitterBuffer** et **GstRtpPtDemux** en un seul élément.

**Rtpbin permet la création** de **plusieurs sessions RTP** , audio et vidéo, qui seront **synchronisées ensemble** à l'aide de paquets **RTCP_SR**. (**send report**)

**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**.

### 2.4.2.1 Emission avec h264 et speex

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

```
CLIENT=192.168.1.27
PORT_RTP_VIDEO=5000
PORT_RTCP_VIDEO=5001
PORT_RTP_AUDIO=5002
PORT_RTCP_AUDIO=5003
PORT_RTCP_VIDEO_RET=5005
PORT_RTCP_AUDIO_RET=5007

gst-launch-1.0 rtpbin name=rtpbin \
v4l2src device=/dev/video1 ! video/x-h264, width=1920, height=1080, framerate=30/1 ! rtph264pay !
rtpbin.send_rtp_sink_0 \
rtpbin.send_rtp_src_0 ! udpsink host=192.168.1.27 port=$PORT_RTP_VIDEO \
rtpbin.send_rtcp_src_0 ! udpsink host=192.168.1.27 port=$PORT_RTCP_VIDEO \
sync=false async=false \
udpsrc port=$PORT_RTCP_VIDEO_RET ! rtpbin.recv_rtcp_sink_0 \
alsasrc device="default" ! queue ! audioconvert ! audioresample ! audio/x-raw, rate=16000, width=16,
channels=1 ! speexenc ! rtpspeexpay ! rtpbin.send_rtp_sink_1 \
rtpbin.send_rtp_src_1 ! udpsink host=192.168.1.27 port=$PORT_RTP_AUDIO \
rtpbin.send_rtcp_src_1 ! udpsink host=192.168.1.27 port=$PORT_RTCP_AUDIO \
sync=false async=false \
udpsrc port=$PORT_RTCP_AUDIO_RET ! rtpbin.recv_rtcp_sink_1
```

### 2.4.2.2 Réception

```
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



CLIENT=192.168.1.72
PORT_RTP_VIDEO=5000
PORT_RTCP_VIDEO=5001
PORT_RTP_AUDIO=5002
PORT_RTCP_AUDIO=5003
PORT_RTCP_VIDEO_RET=5005
PORT_RTCP_AUDIO_RET=5007
```

### 2.4.2.3 Emission avec `jpeg`

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

```
CLIENT=192.168.1.27
PORT_RTP_VIDEO=5000
PORT_RTCP_VIDEO=5001
PORT_RTP_AUDIO=5002
PORT_RTCP_AUDIO=5003
PORT_RTCP_VIDEO_RET=5005
PORT_RTCP_AUDIO_RET=5007

gst-launch-1.0 rtpbin name=rtpbin \
v4l2src device=/dev/video1 ! image/jpeg, width=640, height=480, framerate=30/1 ! rtpjpegpay !
rtpbin.send_rtp_sink_0 \
rtpbin.send_rtp_src_0 ! udpsink host=192.168.1.27 port=$PORT_RTP_VIDEO \
rtpbin.send_rtcp_src_0 ! udpsink host=192.168.1.27 port=$PORT_RTCP_VIDEO \
sync=false async=false \
udpsrc port=$PORT_RTCP_VIDEO_RET ! rtpbin.recv_rtcp_sink_0 \
alsasrc device="default" ! queue ! audioconvert ! audioresample ! audio/x-raw, rate=16000, width=16,
channels=1 ! speexenc ! rtpspeexpay ! rtpbin.send_rtp_sink_1 \
rtpbin.send_rtp_src_1 ! udpsink host=192.168.1.27 port=$PORT_RTP_AUDIO \
rtpbin.send_rtcp_src_1 ! udpsink host=192.168.1.27 port=$PORT_RTCP_AUDIO \
sync=false async=false \
udpsrc port=$PORT_RTCP_AUDIO_RET ! rtpbin.recv_rtcp_sink_1
```

### 2.4.2.4 Réception

```
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=5000 ! rtpbin.recv_rtp_sink_0  \
     rtpbin. ! queue ! rtpjpegdepay ! jpegdec ! 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

Analyser et tester les pipelines ci-dessus – combien de numéros de ports utilise-t-on et pourquoi ?
Préparer et tester le même pipeline avec un **fichier de configuration type :**
Effectuer un streaming d'une source enregistrée - un fichier de type **`tom.mp4`** ou **`tgtbtu.mkv`**.

## 2.5 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 **`smartcomputerlab.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/video1 ! video/x-raw! videoconvert ! videoscale ! video/x-raw,
hight=480, width=640, format=RGB ! queue ! gdkpixbufoverlay location=../samples/smartcomputerlab.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/video1 ! video/x-raw! videoconvert ! videoscale ! video/x-raw,
hight=480, width=640, format=RGB ! queue ! gdkpixbufoverlay location=../samples/smartcomputerlab.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.

### A faire

Envoyer une vidéo avec une image incrustée en **jpeg** (640x480) puis en h264.
Ajouter une piste audio en speex.

# Lab 3

## Gstreamer-1.0 , applications

Dans ce laboratoire nous allons développer quelques applications **gstreamer** avec le code en langage C. Selon le besoin on peut créer de applications avec les composants liés de la façon **statique** ou **dynamique**.

## 3.1 Applications «dynamiques»

Nous allons créer une première application simple, un simple lecteur audio en ligne de commande Ogg/Vorbis. Pour cela, nous utiliserons uniquement les composants standards de GStreamer. Le joueur lira un fichier spécifié sur la ligne de commande. Commençons!
La première chose à faire dans votre application est d'initialiser GStreamer en appelant **gst_init()**. Assurez-vous également que l'application inclut **gst/gst.h** afin que tous les noms de fonctions et objets soient correctement définis.
Utilisez **#include <gst/gst.h>** pour ce faire.

Ensuite, vous souhaiterez créer les différents éléments en utilisant **gst_element_factory_make ()**. Pour un lecteur audio **ogg/vorbis**, nous aurons besoin d'un élément source qui lit les fichiers depuis un disque. GStreamer inclut cet élément sous le nom « **filesrc** ».
Ensuite, nous aurons besoin de quelque chose pour analyser le fichier et le décoder en audio brut. GStreamer comporte deux éléments pour cela : le premier analyse les flux **ogg** en flux élémentaires (vidéo, audio) et est appelé **oggdemux**. Le second est un décodeur audio **vorbis**, commodément appelé « **vorbisdec** .

Puisque **oggdemux** crée des **pads dynamiques** pour chaque flux élémentaire, vous devrez définir un gestionnaire d'événements **pad-added** sur l'élément **oggdemux**, comme vous l'avez appris dans **Dynamic** (ou parfois) **pads**, pour lier l'**ogg** le démultiplexeur et les éléments du décodeur **vorbis** ensemble.

Enfin, nous aurons également besoin d'un élément de sortie audio, nous utiliserons **autoaudiosink** , qui détecte automatiquement votre périphérique audio.

La dernière chose à faire est d'ajouter tous les éléments dans un élément **conteneur**, un **GstPipeline**, et d'attendre que nous ayons joué la chanson entière. Nous avons déjà appris comment ajouter des éléments à un conteneur dans **Bins**, et nous avons découvert les états des éléments dans **Element States**.
Nous allons également attacher un gestionnaire de messages au **bus pipeline** afin de pouvoir récupérer les **erreurs** et détecter la fin du flux.

Ajoutons maintenant tout le code pour obtenir notre tout premier lecteur audio :

```
#include <gst/gst.h>
#include <glib.h>

static gboolean
bus_call (GstBus     *bus,
          GstMessage *msg,
          gpointer    data)
{
  GMainLoop *loop = (GMainLoop *) data;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
      g_print ("End of stream\n");
      g_main_loop_quit (loop);
      break;
    case GST_MESSAGE_ERROR: {
      gchar  *debug;
      GError *error;
      gst_message_parse_error (msg, &error, &debug);
      g_free (debug);
      g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      g_main_loop_quit (loop);
      break;
    }
    default:
      break;
  }
  return TRUE;
}
```

```
static void on_pad_added (GstElement *element,
              GstPad     *pad,
              gpointer    data)
{
  GstPad *sinkpad;
  GstElement *decoder = (GstElement *) data;
  /* We can now link this pad with the vorbis-decoder sink pad */
  g_print ("Dynamic pad created, linking demuxer/decoder\n");
  sinkpad = gst_element_get_static_pad (decoder, "sink");
  gst_pad_link (pad, sinkpad);
  gst_object_unref (sinkpad);
}

int main (int   argc, char *argv[])
{
  GMainLoop *loop;
  GstElement *pipeline, *source, *demuxer, *decoder, *conv, *sink;
  GstBus *bus;
  guint bus_watch_id;
  /* Initialisation */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  /* Check input arguments */
  if (argc != 2) {
    g_printerr ("Usage: %s <Ogg/Vorbis filename>\n", argv[0]);
    return -1;
  }
  /* Create gstreamer elements */
  pipeline = gst_pipeline_new ("audio-player");
  source   = gst_element_factory_make ("filesrc",       "file-source");
  demuxer  = gst_element_factory_make ("oggdemux",      "ogg-demuxer");
  decoder  = gst_element_factory_make ("vorbisdec",     "vorbis-decoder");
  conv     = gst_element_factory_make ("audioconvert",  "converter");
  sink     = gst_element_factory_make ("autoaudiosink", "audio-output");
  if (!pipeline || !source || !demuxer || !decoder || !conv || !sink) {
    g_printerr ("One element could not be created. Exiting.\n");
    return -1;
  }
  /* Set up the pipeline */
  /* we set the input filename to the source element */
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);
  /* we add a message handler */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  bus_watch_id = gst_bus_add_watch (bus, bus_call, loop);
  gst_object_unref (bus);
  /* we add all elements into the pipeline */
  /* file-source | ogg-demuxer | vorbis-decoder | converter | alsa-output */
  gst_bin_add_many (GST_BIN (pipeline),
                    source, demuxer, decoder, conv, sink, NULL);
  /* we link the elements together */
  /* file-source -> ogg-demuxer ~> vorbis-decoder -> converter -> alsa-output */
  gst_element_link (source, demuxer);
  gst_element_link_many (decoder, conv, sink, NULL);
  g_signal_connect (demuxer, "pad-added", G_CALLBACK (on_pad_added), decoder);
  /* note that the demuxer will be linked to the decoder dynamically.
     The reason is that Ogg may contain various streams (for example
     audio and video). The source pad(s) will be created at run time,
     by the demuxer when it detects the amount and nature of streams.
     Therefore we connect a callback function which will be executed
     when the "pad-added" is emitted.*/
  /* Set the pipeline to "playing" state*/
  g_print ("Now playing: %s\n", argv[1]);
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  /* Iterate */
  g_print ("Running...\n");
  g_main_loop_run (loop);
  /* Out of the main loop, clean up nicely */
  g_print ("Returned, stopping playback\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);
  g_print ("Deleting pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));
  g_source_remove (bus_watch_id);
  g_main_loop_unref (loop);
  return 0;
}
```

Nous avons préparé un simple script pour la compilation du code (**`comp.sh`**)avec les bibliothèques de **gstreamer**.

```
g++ -Wall $1.c -o $1 $(pkg-config --cflags --libs gstreamer-1.0)
```

```
./comp.sh dynapipe.audio_vorbis
```

Exécution :

```
./dynapipe.audio_vorbis ../samples/music.ogg
```

Le même modèle du programme peut être utilisé pour lire les fichiers vidéo. Voici le fragment du code pour la création du pipeline avec le décodage **mp4**.

```
  /* Create gstreamer elements */
  pipeline = gst_pipeline_new ("video-player");
  source   = gst_element_factory_make ("filesrc",       "file-source");
  demuxer  = gst_element_factory_make ("qtdemux",      "qt-demuxer");
  decoder  = gst_element_factory_make ("avdec_h264",     "mp4-decoder");
  conv     = gst_element_factory_make ("videoconvert",  "converter");
  sink     = gst_element_factory_make ("autovideosink", "video-output");
```

Après les modifications vous pouvez compiler l'application par :

```
./comp.sh dynapipe.video_mp4
```

Exécution :

```
./dynapipe.video_mp4 ../samples/tom.mp4
```

## 3.2 Application «statique»

Le lancement d'un pipeline statique nécessite une simple ligne du code :

```
pipeline = gst_parse_launch("playbin
uri=https://www.freedesktop.org/software/gstreamer-sdk/data/media/sintel_trailer-480p.webm",
nullptr);
```

ou

```
pipeline = gst_parse_launch("playbin uri=file:///home/rock/gstreamer/samples/tom.mp4", nullptr);
```

### 3.2.1 Application statique – code complet

```
#include <gst/gst.h>

int main(int arg, char *argv[]) {
    GstElement *pipeline = nullptr;
    GstBus *bus = nullptr;
    GstMessage *msg = nullptr;
    // gstreamer initialization
    gst_init(&arg, &argv);
    // building pipeline
    pipeline = gst_parse_launch( "playbin
uri=https://www.freedesktop.org/software/gstreamer-sdk/data/media/sintel_trailer-480p.webm",
nullptr);
    // start playing
    gst_element_set_state(pipeline, GST_STATE_PLAYING);
    //wait until error or EOS ( End Of Stream )
    bus = gst_element_get_bus(pipeline);
    msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE,
                                     static_cast<GstMessageType>(GST_MESSAGE_ERROR |
GST_MESSAGE_EOS));

    // free memory
    if (msg != nullptr)
        gst_message_unref(msg);
    gst_object_unref(bus);
    gst_element_set_state(pipeline, GST_STATE_NULL);
    gst_object_unref(pipeline);

    return 0;
}
```

### A faire :

1. Tester les exemples ci-dessus
2. Créer une application dynamique permettant de decoder et rendre plusieurs types de média audio et vidéo.

# Table of Contents