

Transmission de données multimédia

TP3 FLUX VIDEO ET AUDIO

Guéron Marie / Favreau Jean-Dominique / Tanguy Arnaud

Table des matières

| | |
|-----------------------------------|----|
| Initialisation | 2 |
| 1. La salle 310..... | 2 |
| 2. Logiciels..... | 2 |
| Flux d'image | 3 |
| 1. Caméra..... | 3 |
| Avec un navigateur | 3 |
| Avec gstreamer..... | 4 |
| Décomposition du flux | 4 |
| Ce flux en réseau | 5 |
| 2. Un autre streaming..... | 6 |
| Avec decodebin | 6 |
| On décompose plus finement | 6 |
| Sauvegarde directe..... | 6 |
| Sauvegarde et visualisation | 7 |
| 3. Fichier | 8 |
| Sons | 9 |
| 1. Fichier | 9 |
| Lecture..... | 9 |
| Envoi UDP..... | 9 |
| Envoi RTP/UDP | 10 |
| Vidéo | 11 |

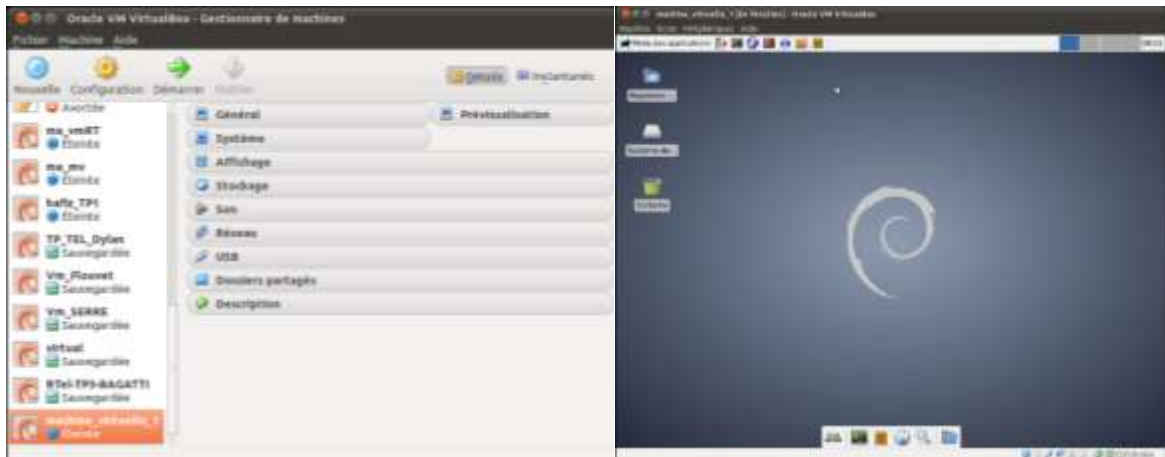
Initialisation

1. La salle 310

Grâce à la commande *creatvm* nous créons une machine virtuelle :

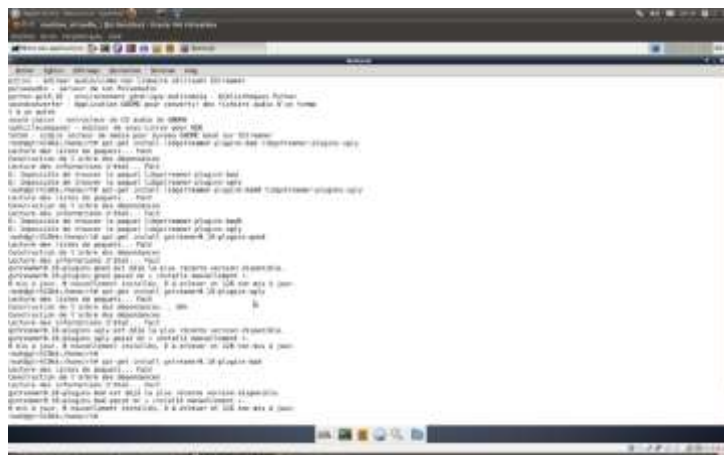


Puis nous lançons la machine virtuelle créée précédemment.



2. Logiciels

Sur la machine virtuelle fraîchement créée, les logiciels dont nous allons avoir besoin par la suite ne sont pas à jour. Nous avons donc dû les mettre à jour ainsi qu'installer les logiciels non présents, tels que *vlc*, *qstreamer*,...



Flux d'image

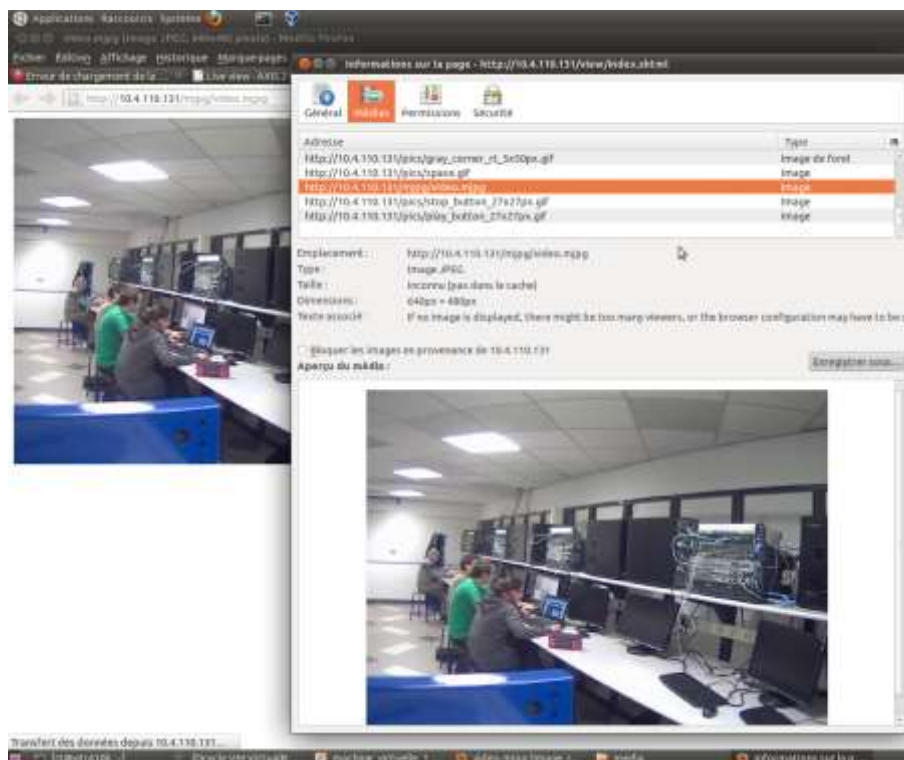
1. Caméra

Une fois la machine virtuelle lancée et à jour, nous avons eu à récupérer et afficher les images fournies en temps réel par la caméra Axis. Pour cela nous avons utilisé deux méthodes :

Avec un navigateur

Notre première méthode a été de récupérer les images envoyées par la caméra grâce un navigateur web. Pour cela, nous pouvons accéder à deux adresses différentes et récupérer deux flux de type différents.

- A l'adresse <http://10.4.110.131/mjpg/video.mjpg>, nous récupérons un flux encapsulé en MJPEG. C'est-à-dire une suite d'images statiques affichées et rafraichies suffisamment souvent pour que l'œil humain croie à des mouvements fluides.
- A l'adresse <https://10.4.110.131:554/mpeg4/media.amp>, nous récupérons un flux encapsulé en MPEG4. C'est-à-dire une vidéo.



Avec gstreamer

La deuxième méthode que nous avons utilisée consistait à utiliser la commande *gstreamer* pour construire une chaîne qui traite notre flux vidéo en agénçant les traitements les uns après les autres.



Décomposition du flux

Lorsque l'on décompose la commande *gstreamer*, on se rend compte que les plugins utilisés sont *jpegdec0*, *ffmpegCsp*, *autovideosink0*.

De plus, nous avons pu voir que les images envoyées par la caméra sont des images jpeg codées en YUV (luminance/chrominance). Ceci est lié au fait que ce sont les variations sur ces paramètres qui sont le mieux perçues par l'œil humain. Au contraire, les images sont reconverties pour être affichées en RGB, car c'est ce type d'affichage qui est le mieux géré par l'ordinateur.

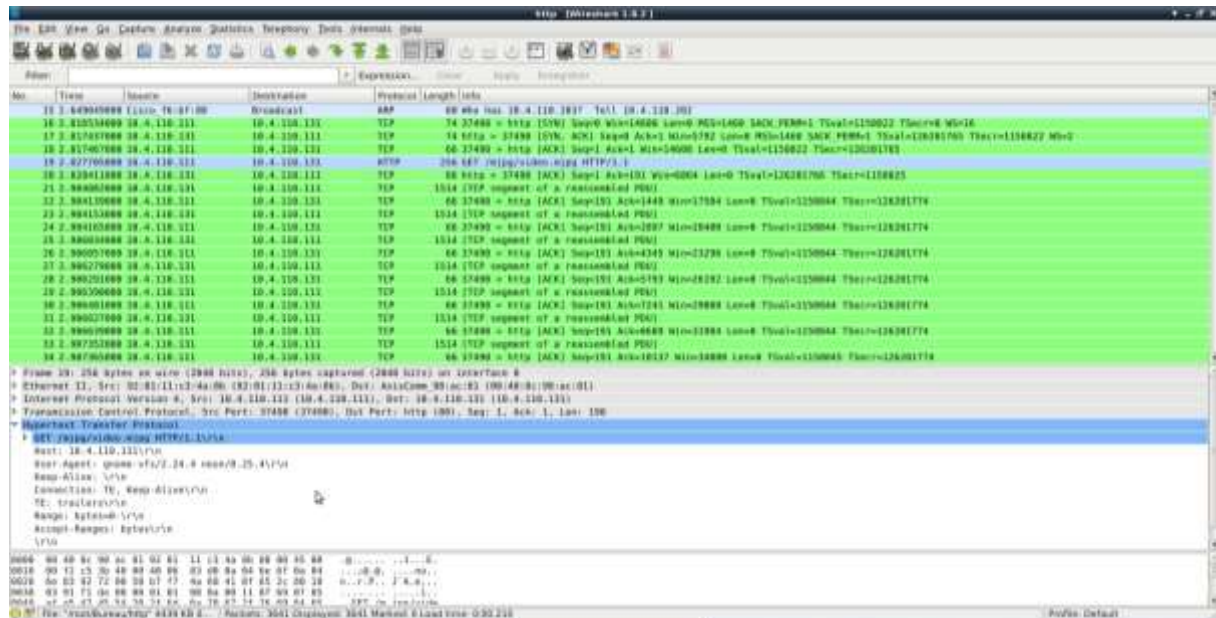
```
rtsp://10.4.110.131/mjpg/video.mjpg | decodebin | ffmpegcolorspace | autovideosink
Définition du pipeline à PAUSER...
Le pipeline est en phase de PREROLL...
Le pipeline est en phase de PREROLL...
Le pipeline a terminé la phase PREROLL...
Passage du pipeline à la phase PLAYING...
Now clock: GstSystemClock

/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstTypeFindElement:TypeFind.GstPad:src: caps = multipart/x-mixed-replace
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstMultiPartDemux:MultiPartDemux0.GstPad:sink: caps = multipart/x-mixed-replace
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstQueue:queue0.GstPad:src: caps = image/jpeg
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstQueue:queue0.GstPad:sink: caps = image/jpeg
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstIpegDec:jpegdec0.GstPad:sink: caps = video/x-raw-yuv, format=(fourcc)I420, width=(int)640, height=(int)480, framerate=(fraction)8/1
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstIpegDec:jpegdec0.GstPad:src: caps = video/x-raw-yuv, format=(fourcc)I420, width=(int)640, height=(int)480, framerate=(fraction)8/1
/GstPipeline:pipeline0/GstFFmpegCsp:ffmpegcsp0.GstPad:src: caps = video/x-raw-rgb, bpp=(int)32, depth=(int)32, endianess=(int)4321, red_mask=(int)16711680, green_mask=(int)16711680, blue_mask=(int)16777216, width=(int)640, height=(int)480, framerate=(fraction)8/1, pixel-aspect-ratio=(fraction)1/1
/GstPipeline:pipeline0/GstFFmpegCsp:ffmpegcsp0.GstPad:sink: caps = video/x-raw-yuv, format=(fourcc)I420, width=(int)640, height=(int)480, framerate=(fraction)8/1
/GstPipeline:pipeline0/GstDecodeBin:decodebin0/GstGstPad:src0.GstFracPad:proppad1: caps = video/x-raw-yuv, format=(fourcc)I420, width=(int)640, height=(int)480, framerate=(fraction)8/1
/GstPipeline:pipeline0/GstAutovideosink:autovideosink0/GstImageSink:autovideosink0-actual-sink-ximage.GstPad:sink: caps = video/x-raw-rgb, bpp=(int)32, depth=(int)24, endianess=(int)4321, red_mask=(int)16711680, green_mask=(int)16711680, blue_mask=(int)16777216, width=(int)640, height=(int)480, framerate=(fraction)8/1, pixel-aspect-ratio=(fraction)1/1
/GstPipeline:pipeline0/GstAutovideosink:autovideosink0/GstImageSink:autovideosink0.GstGstPad:sink0.GstFracPad:proppad1: caps = video/x-raw-rgb, bpp=(int)32, depth=(int)24, endianess=(int)4321, red_mask=(int)16711680, green_mask=(int)16711680, blue_mask=(int)16777216, width=(int)640, height=(int)480, framerate=(fraction)8/1, pixel-aspect-ratio=(fraction)1/1
Le pipeline a terminé la phase PREROLL...
Passage du pipeline à la phase PLAYING...
Now clock: GstSystemClock
```

Ce flux en réseau

Le but était ici d'analyser les protocoles utilisés pour envoyer la vidéo de la caméra à l'ordinateur.

Nous avons pu constater que la connexion est initialisée par une requête HTTP, après quoi les frames en jpeg sont envoyées par TCP.



The screenshot shows a Wireshark packet capture of an HTTP GET request and subsequent TCP segments. The packet list on the left shows packets 1 through 14. The packet details pane on the right shows the structure of the selected packet (packet 1), which is an HTTP GET request. The packet bytes pane at the bottom shows the raw data of the selected packet.

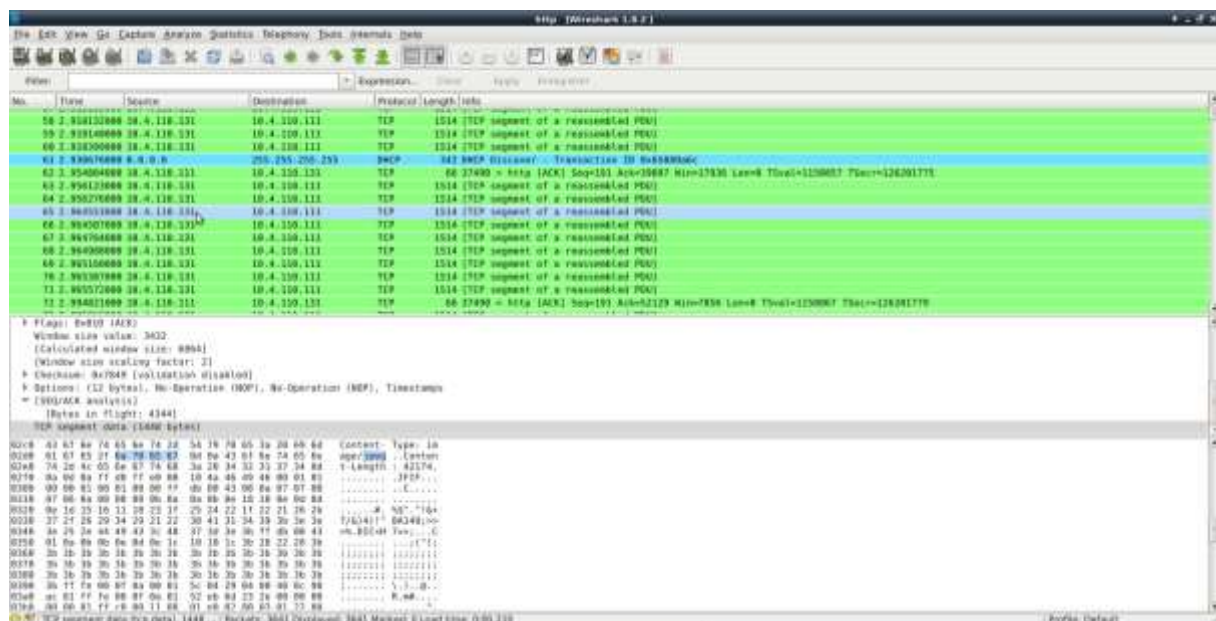
| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|----------|--------------|--------------|-----------|--------|---|
| 1 | 0.000000 | 10.4.110.101 | 10.4.110.101 | Broadcast | 60 | ARP Request 10.4.110.101 > 10.4.110.101 |
| 2 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 74 | 37488 > 37489 [SYN] Seq=0 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 3 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 74 | 37489 > 37488 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 4 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 5 | 0.000000 | 10.4.110.101 | 10.4.110.101 | HTTP | 284 | GET /img/000000.jpg HTTP/1.1 |
| 6 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 7 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 8 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 9 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 10 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 11 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 12 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |
| 13 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 14 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 60 | 37488 > 37489 [ACK] Seq=1448 Win=0 Len=0 MSS=1460 SACK_PERM=1 TSval=1150022 TSecr=0 Win=0 |

Packet 5 details:

```

GET /img/000000.jpg HTTP/1.1
Host: 10.4.110.101
User-Agent: curl/7.28.0 (macOS; arm64)
Accept: */*
Accept-Encoding: gzip, deflate
Range: bytes=0-1000000

```



The screenshot shows a Wireshark packet capture of a TCP segment and subsequent data segments. The packet list on the left shows packets 15 through 21. The packet details pane on the right shows the structure of the selected packet (packet 15), which is a TCP segment. The packet bytes pane at the bottom shows the raw data of the selected packet.

| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|----------|--------------|--------------|----------|--------|------------------------------------|
| 15 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 16 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 17 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 18 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 19 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 20 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |
| 21 | 0.000000 | 10.4.110.101 | 10.4.110.101 | TCP | 1514 | TCP segment of a retransmitted PDU |

Packet 15 details:

```

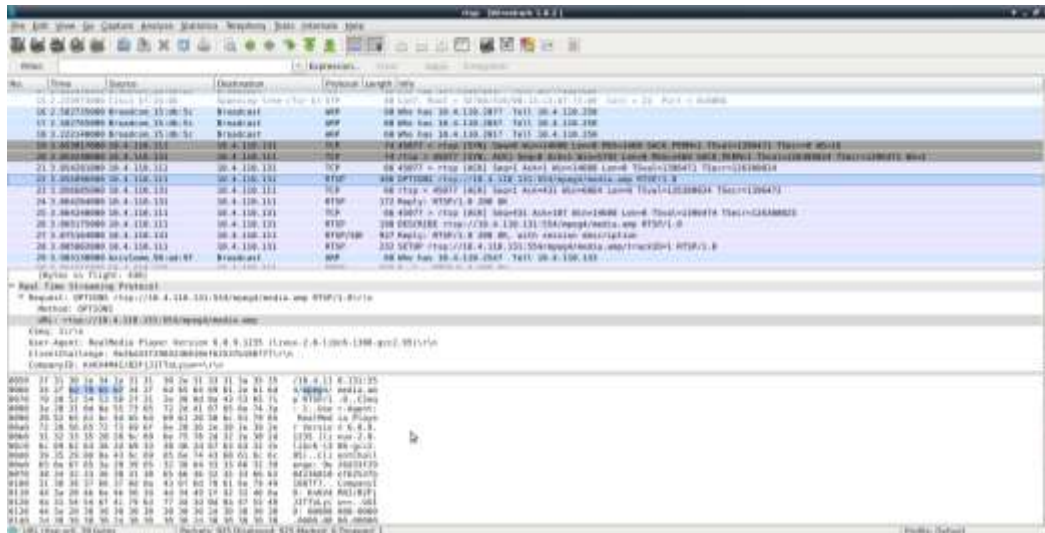
Flags: BWR (ACK)
Window size value: 3402
[Calculated window size: 3402]
[Window size scaling factor: 2]
Checksum: 0x7049 [validation failed]
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
= [RST/ACK analysis]
[Bytes in flight: 4344]
TCP segment data (1440 bytes)

```


2. Un autre streaming

Avec decodebin

Nous avons lancé la lecture de la vidéo avec *gststreamer* et avec le plugin *decodebin*, puis nous avons analysé les flux envoyés grâce à *wireshark*.

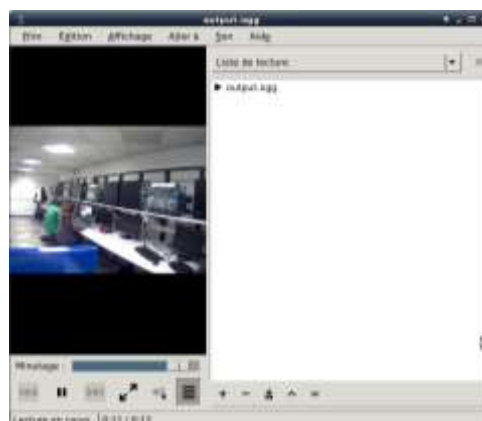


Nous pouvons par ailleurs constater que le transport utilisé ici est le protocole RTP.

On décompose plus finement

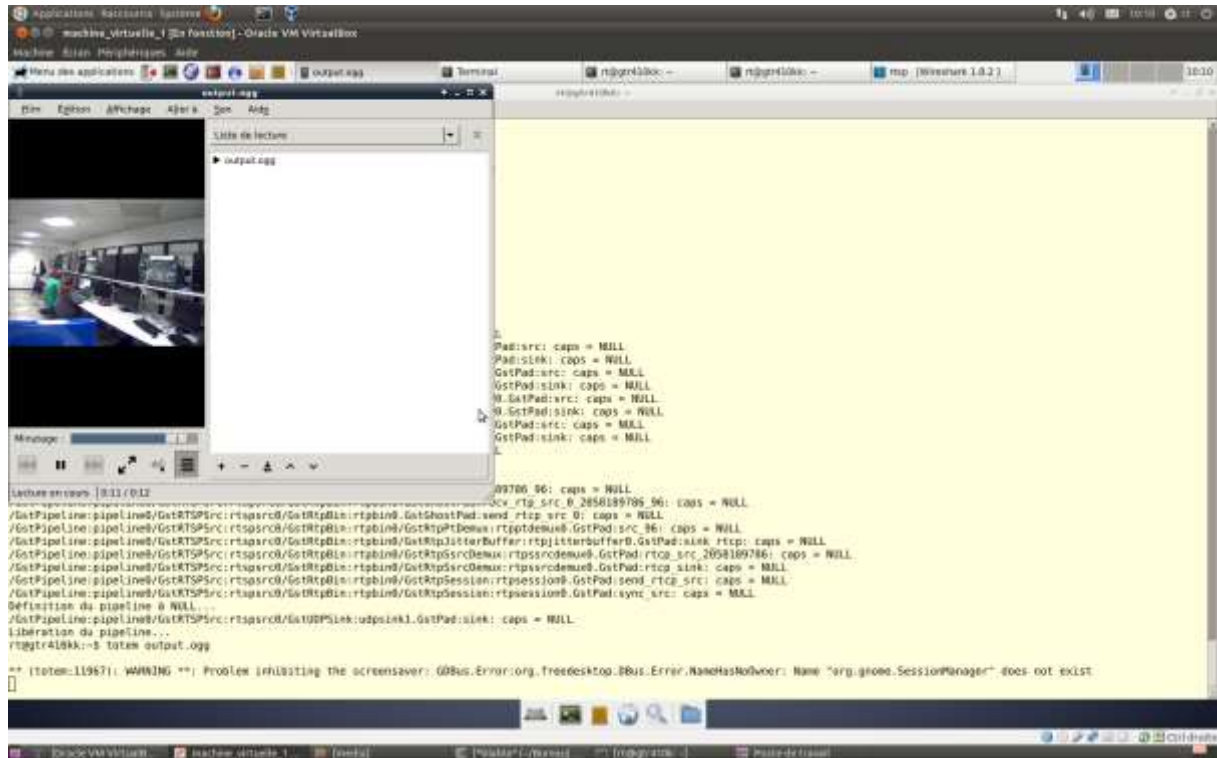
Sauvegarde directe

Dans cette partie, nous avons enregistré le flux de sortie de la caméra dans un fichier *output.ogg* grâce à la commande *gst-launch*. Il est important de noter dans ce cas que l'on ne voit pas directement le flux vidéo que l'on enregistre, mais l'on vérifie à posteriori l'efficacité de la méthode en lisant la vidéo avec un lecteur de film.



Sauvegarde et visualisation

Nous avons fait évoluer notre manière de sauvegarder le flux vidéo. En effet, maintenant nous voyons les images sortant de la caméra en même temps que la sauvegarde se fait.



Sons

1. Fichier

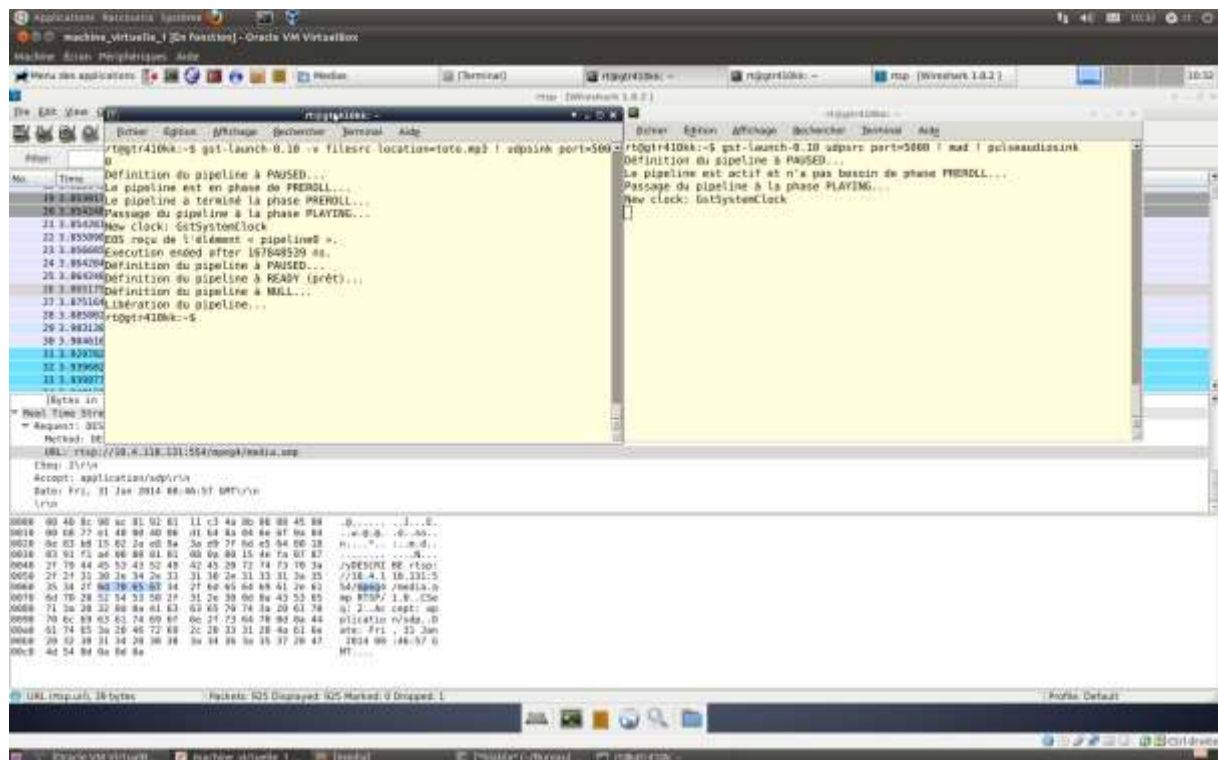
Lecture

La lecture du son faite avec *gststreamer* mais sans contrôle du flux (il lit les informations au fur et à mesure qu'elles arrivent et sans se préoccuper des pertes) nous donne un son de qualité médiocre.

Envoi UDP

Lors de l'envoi du son à un autre ordinateur, il faut faire attention à lancer la réception avant l'envoi. A l'inverse, le son n'a rien pour être reçu et n'est donc pas entendu.

Par ailleurs, le flux reçu n'étant toujours pas décodé par UDP/RTP, on constate toujours une mauvaise qualité audio.



Envoi RTP/UDP

Dans ce cas, nous avons ajouté une entête RTP à l'envoi du flux vidéo ainsi qu'à la réception. Ce qui signifie que les données audio ne sont pas lues au fur et à mesure de leur arrivée mais bien dans leur ordre de lecture normale. Nous avons constaté que le son entendu à la réception était de bien meilleure qualité bien qu'il y ait un phénomène de gigue (« trous » dans le son) perceptible.

Vidéo

En combinant toutes les commandes vues précédemment, nous avons lu la vidéo *big_bunny_720p_h264.mov* tout en ayant le son correspondant.

