projet de fin d'étude :

Vidéoconférence multicast sur satellite

et

adaptation des protocoles de routage multicast pour satellite

SOMMAIRE :

Introduction **4**

Le LaBri ***4***

Le Projet ***5***

Le CNES ***6***

1. Etat de l’Art **7**

1.1. Multicast sur satellite ***7***

1.2. Diffusion a la demande(Vod) et Télévision numérique ***9***

1.4. l architecture IMS ***9***

2. Routage multicast Satellite **12**

2.1 Architecture générale ***13***

2.2 Architecture cote GW ***15***

2.3 Implémentation de l agent SIP ***15***

2.3.1Parser XML *15*

2.3.2 écoute et interprétation des requêtes SIP *21*

2.3.3 Connexion au routeur TISS *24*

2.4 Architecture cote ST ***26***

2.6. compilation du ST. ***31***

2.7. Compilation et exécution de l agent SIP. ***32***

3. Vidéoconference multicast **33**

3.1. Etat de l art, choix et Principe ***33***

3.2. Architecture générale ***35***

3.2.1 mécanismes réseau *36*

3.2.2 Gestion des flux vidéos *39*

3.2.3 choix généraux d implémentation *39*

3.3 Implémentation coté client. ***42***

3.3.1 VLM ou VideoLanManager *42*

3.3.2 module d interface *44*

3.3.3 module de signalisation *50*

3.3.4 communication inter modules *58*

3.4. Implémentation coté concentrateur ***62***

3.4.1 Architecture général *62*

3.4.2 Le module sip focus *63*

3.4.3 intégration du module dans VLC *75*

3.5. Test ***75***

Conclusion **77**

A. Bibliographie **78**

Introduction

Le LaBri

Le LaBri est un laboratoire de recherche informatique public. Il est composé d'environ 250 personnes ; chercheurs, enseignants-chercheurs ou doctorants. Situé a l'intérieur de l'université Bordeaux 1,il fut crée il y a une vingtaine d'année en partenariat avec l'université Bordeaux 1 , l'ENSEIRB et l'université Bordeaux 2. Les chercheurs, doctorants et stagiaires sont regroupés par équipe. Chaque équipe possède un thème de recherche précis. Ce Laboratoire travaille sur la recherche fondamentale et la recherche appliqué dans le domaines :
_Combinaisons et Algorythmique : recherche fondamentale sur la manipulation des objets les plus courants (arbres, graphes...) en informatique
_Images et son : Cete branche travaille sur l'analyse des images sons et vidéos dans le but d'indexer ces supports. Elle participe aux recherches sur les images de synthèse ou la composition assisté par ordinateur.
_langages, Sytèmes et Réseaux : recherche sur la téléphonie IP, modélisation de modèles de communications, et la configuration et l'adaptation de mécanismes e qualité de service sur des réseaux IP.
_méthodes Formelles : recherche sur le Langage informatique afin de proposer de nouveaux modèles
_Modèles et algorithmes pour la Bioinformatique et la Visualisation d'informations :
_Supports et algorithmes pour les Applications Numériques hautes performances: développement de systèmes distribués et parallèles afin d'effectuer les calculs sur un très grand nombres de machine ou sur des machines possédant un grand nombre de processeurs.

figure 0.1 : vue du LaBri

Engagé au sein de l'équipe COMET, spécialisé dans la diffusion temps réel sur les réseaux, je travaille avec le stagiaire Heni Karaa sur l'implémentation d'un cahier des charges établi par toufik Ahmed et Ismaël Djama.

Le Projet

Le projet consiste en l'étude et l'implémentation de nouvelles architectures de communication de groupe pour les communications satellites. Ce projet se déroule sur une période d'un an. Ma tâche consistait en l'implémentation du projet. Le demandeur du projet est le CNES; Centre National d'Etudes Spatial.

Ce projet est décomposé en 3 éléments :
_ Un système permettant de gérer les abonnements de groupe à niveau IP tout en évitant les différents écueils liés au transport par satellite. Ce système d'abonnement se doit aussi d'apporter des services tel que l'authentification et le filtrage.
Ainsi que 2 services applicatifs permettant de vérifier la viabilité du système multicast notamment :
_ Un système de diffusion de la Télévision par satellites. Ce système était déjà développé par l'équipe.
_ Un logiciel de vidéoconférence utilisant les communications de groupes.

Le CNES

Le CNES participe en tant que client du projet. Cet organisme crée en 1961 a pour but l'exploration et la valorisation du domaine spatial. Le centre national d'études spatial travaille autour de 5 thèmes . Le premier thème est l'accès a l'espace, très connu grâce au fusées Arianeet au site de lancement de Kourou.
Le CNES participe au développement durable en fournissant les outils nécessaires permettant d'observer et d'analyser la Terre de manière global. On peut citer les programmes SPOT et Argos rentrant dans cette thématique.
Le CNES développe aussi les technologies spatiales pour le grand public notamment avec les satellites de télécommunication ou le programme de géo-localisation Galiléo.
Une partie des travaux du CNES consiste en l'usage du spatial militaire. C'est pour cette raison que le niveau de sécurité sur les sites du CNES, notamment celui de Toulouse, sont très élevé.
Enfin le CNES travaille sur la recherche et le développement de nouvelles applications spatiales. On peut citer par exemple le satellite Cassini-Huygens pour l'étude de notre système solaire.

Nous travaillons pour une équipe du centre de recherche de Toulouse. Afin d'effectuer les test sur place, nous avions à disposition 3 ordinateurs type PC sous linux disposé autour d'une plate-forme de simulation d'un lien satellite.

1. Etat de l'Art

1.1. Multicast sur satellite

Les protocoles pour le routage de flux multicast se décompose en 2 sous-ensembles : les protocoles tels M-OSPF ou PIM[1] de routage multicast inter-routeur ainsi que les protocoles d'abonnement des utilisateurs que sont MLD[2] et IGMP[3] (Internet Group Management Protocol) . Ces derniers ne sont utilisés que entre les terminaux et le premier routeur rencontré.

Le protocole IGMP permet la gestion des adresses multicast du protocole IPv4. Comme la norme IP, il est basé sur les principes du *best effort*, c'est-à-dire que le service est fourni sans offrir une qualité de service défini.
Avec le protocole IGMP, les terminaux peuvent s'abonner (requête *join*) et se désabonner (requêtes *leave*) à un groupe multicast auprès du routeur le plus proche. Le désabonnement est apparu avec la deuxième version du protocole.
Le routeur quant à lui répond à une requête d'abonnement avec un message *IGMP report* afin que chaque terminal présent sur le même lien connaisse l'existence de l'abonnement. En effet le système multicast ayant pour but d'optimiser l'usage des réseaux physiques multipoint, IGMP informe de cette manière chaque terminal des groupes qui sont diffusé sur son interface.

Le routeur a aussi la possibilité d'interroger les terminaux sur leur désirs de poursuivre l'abonnement avec la requête *IGMP Query*. les terminaux intéressés réponde alors par la requêtes IGMP report.
Afin d'éviter la saturation du routeur, des règles sont respectées pour l'envoi de cette réponse. Chaque client envoie sa réponse après un temps court aléatoire sur l'adresse multicast interrogé. Le fait d'envoyer la réponse en multicast permet ainsi aux autres terminaux de savoir si une requête *IGMP report* a déjà été envoyé. Dans ce cas alors, le terminal intéressé par l'adresse multicast ne vas pas répondre afin d'éviter l'engorgement. Dans le cas optimal une seule réponse sera envoyé a la requête *QUERY* du routeur.

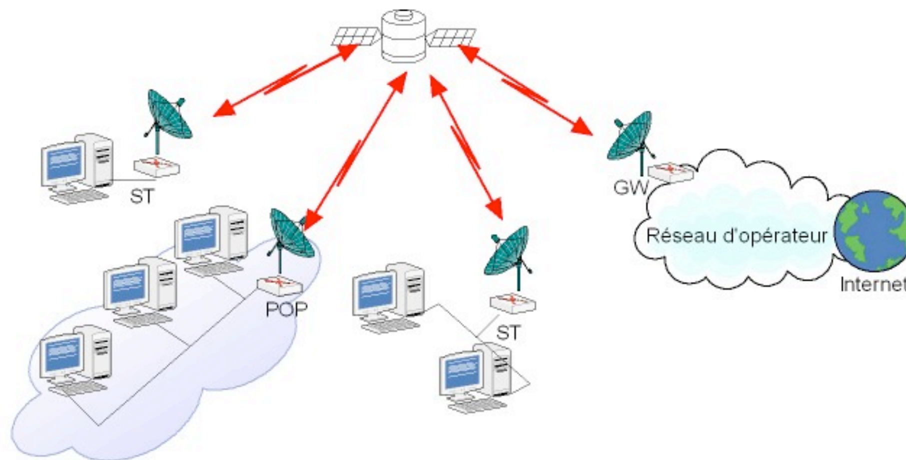

figure 1.1 : utilisation du satellite en tant que fournisseur d accès

Dans le cas du satellites en tant que fournisseur d'accès, comme montré figure 1.1, les techniques pour éviter la saturation sont inopérantes. en effet si le lien descendant (du satellite au client) est multipoint et à large diffusion, le lien montant est un lien point à point. La conséquence est que les clients ne peuvent s'écouter les uns les autres. Donc lorsque le routeur envoie une requête IGMP Query, tous les terminaux intéressés vont répondre sans pouvoir écouter ses voisins. Même si la réponse est décalé d'un temps aléatoire , le risque de saturation du lien satellite et du routeur est très grand. Afin de limiter ces problèmes, une version de IGMP a été modifié pour s'adapter au liens satellites et notamment pour résoudre ces problèmes de saturation.

La diffusion des adresses de groupes sur le lien descendant d'un satellite permet a des milliers de clients de potentiellement lire ces adresses. Afin d'empêcher que des utilisateurs non autorisé aient accès a la session multicast, il faut crypter les paquet au niveau réseau avec par exemple le protocole IPsec (Internet Protocol Security) ou au niveau transport comme le propose les protocole SSL (secure socket Layer) ou S-RTP. Ces protocoles se rajoutant au mécanisme d'abonnement aux flux multicast, ils ne sont pas forcement idéaux car ils rajoutent une couche d'en-tête supplémentaire.

L'idéal est un système pouvant prévenir les saturations serveurs et que le matériel permettant la réception satellite chez l'usager soit authentifié et fasse usage de filtre pour les adresses multicast non accordé à l'utilisateur.

1.2. Diffusion a la demande(Vod) et Télévision numérique

La diffusion numérique de contenu audio-visuel est un fer de lance du satellite de télécommunication civil. Depuis une quinzaine d'années, la télévision par satellite se diffuse numériquement sous le protocole DVB (Digital Video Broadcasting) . Ce protocole de transport de flux vidéo prévoit l'encodage du flux sous la norme MPEG2. Afin de combler les nouvelles attentes, le satellite est dorénavant équipé d'une voie de retour. Le protocole de transport a alors évolué de manière à pouvoir transiter d'autre flux que télévisuel, comme des paquets IP ou de la téléphonie mobile.
Depuis peu d'années, le protocole DVB a été porté sur le réseau hertzien. Implementé en France sous le nom T.N.T. (télévision numérique terrestre), il eu un grand succès.

Plus récemment, de nombreux fournisseurs de services liés à l'accès internet (Free, Neuf..) ou à la diffusion par satellite (Canal SAT) offre parmi leurs abonnements un système de VoD. Certaines de ces solutions sont basées sur des protocoles propriétaires (Neuf avec Microsoft TV) , d'autres non. On peut citer dans les solutions libres, celle adoptée par Free basée sur VLC[4] pour le client et sur des protocoles tel que RTSP[5], RTP/RTCP[6] recommandés par l'IETF.

Une autre méthode proche du VoD et qui connaît un grand succès actuellement, sont les sites web de diffusion de vidéo en ligne (YouTube, DailyMotion...). L'accès aux vidéo se fait par le protocole HTTP via un site web dynamique, le flux vidéo est quant à lui envoyé en streaming sur le protocole HTTP lui aussi.
On remarque la présence de buffer assez important sur ces systèmes afin de compenser le fait que HTTP ne soit pas un protocole adapté au temps-réel. De plus, la plupart de ces sites webs sont basés sur des clients propriétaires et des codecs vidéos fermés.

On remarque aussi l'apparition sur internet de système originaux de Vod basés sur le P2P (P2PTV du projet S4).

1.4. l'architecture IMS

L'architecture IMS (IP multimedia Subsystem) fut conçu par le groupe 3GPP. Elle a pour but d'offrir un système unique de transport pour les applications multimédia. De cette façon, les opérateurs n'auront pas à reconstruire à chaque fois les couches de transport et de contrôle pour les applications multimédia. De plus cette architecture a pour but de faciliter l'inter-opérabilité des services en assurant une base commune. Elle est basé sur IP et le protocole SIP.

SIP ou Session Initiation Protocol est un protocole permettant le gestion de session temps réel. Ce protocole écrit sous forme de texte est basé sur 6 requêtes :
_INVITE : Indique qu'un client souhaite entamer une conversation
_ACK : indique que le client a reçu la réponse de son message invite.

_BYE : indique que l'utilisateur souhaite quitter une conversation.
_CANCEL : annule une demande d'appel en cour.
_OPTIONS : demande les capacité du serveur.
_REGISTER : enregistre son adresse to auprès d'un serveur SIP.
Afin d'informer précisément la nature des ordres, les messages contiennent toujours les champs suivants:
_champ *to* le destinataire du message.
_champ *from* : l'expéditeur du message. C'est 2 champs sous forme d'URI a la manière du courriel "sip: *pseudonyme@serveur*".
_champ *Call-ID* : identifiant numérique permettant de discriminer une session. Ce numéro est défini par l'appelant et est conservé tant que la session existe.
_champ *Cseq* : numéro commençant à 1 au début d'une session. Il est incrémenté a chaque message SIP et permet d'ordonner les messages.
_champ *content-length* : spécifie la taille du corps du message.
_champ *content-Type* : spécifie le type du corps du message.

Le protocole est facilement extensible. Parmi les extensions proposées, on peut citer un gestionnaire de présence basé sur les ordres SUSCRIBE pour souscrire au gestionnaire de présence et NOTIFY pour informé les abonnés au gestionnaire de présence du changement de status dudit gestionnaire. Il existe aussi des extensions implémentant la messagerie instantané avec entre autre l'ordre MESSAGE. Cet ordre peut etre aussi utilisé "déconnecté" , sans l'initiation d'une session au préalable, à la manière d'un SMS en téléphonie portable.

Afin de transporter les flux , SIP se base sur de nombreux autres protocoles tel que SDP[7] pour la description de flux ou RTP pour le transport

Le système SIP propose 4 éléments réseau :
_L'agent utilisateur : c'est l'agent terminal du protocole, c'est par lui que l'utilisateur va entamer ou recevoir une session. Cela peut être une application ou un appareil téléphonique.
_Le serveur proxy : ce type de serveur permet de rediriger les paquets SIP et les flux qui en sont issu. C'est un point de passage obligé très utile pour passer les passerelles IP. Par exemple, il est indispensable pour accéder à un utilisateur situé sur un réseau IP privé.
_ Le registrar : permet aux utilisateurs d'enregistrer leurs positions afin qu'ils soient facilement accessibles ensuite. Un usager souhaitant appeler quelqu'un sans connaître sa position va d'abord interroger le registar de la personne, celui ci répondra par sa localisation. Il est évident que chaque utilisateur doit informer son registrar d'un changement de position afin de demeurer accessible. Si dans la pratique le registrar

n’est pas obligatoire , il est utilisé dans a plupart des déploiement de services basé sur SIP.

Aujourd’hui, SIP est principalement connu et utilisé pour la voix sur IP. Pourtant son usage ne se limite pas a ce champ d’action. Ainsi il peut etre utilisé pour tous transport de flux en temps réel, allant jusqu'à des applications pour la maîtrise de champ de calculateurs.

Pour l’usage qui nous intéresse, la conférence, SIP propose 3 modèles :

_Le modèle **Faiblement couplé** ou tous les utilisateurs de la conférence s’accorde autour d’une adresse multicast pour envoyer leurs flux. La liste des participants est connu grâce au protocole d’information tel que SAP/SDP (Session Annoucement Protocol). Aucune entité spécialisé ne gère la conférence. C’est la méthode la plus simple a mettre en oeuvre, par contre tous le monde y a accès et aucune fiabilité ne peut être assuré.

_ le modèle **Fortement Couplé** ou une entité spécialisé, nommé focus ou concentrateur, va gérer la conférence. Toute la signalisation et les flux vont passer par cette entité. l’URI de la conférence va correspondre a l’URI du concentrateur.Cette organisation permet une authentification des utilisateurs ainsi qu’un moyen de rediriger les flux de manière efficace. Comme le focus est un point central, il peut saturer assez rapidement rendant la conférence inutilisable. Enfin plusieurs conférences peuvent êtres attaché a un focus.

_Le modèle **totalement distribué** ou chaque participant maintient une signalisation direct avec les autres participants de la conférence. Chaque utilisateur se charge aussi de distribuer le flux autres participants. Ce modèle est très souple puisque chaque participant est indépendant et aucune entité ne menace la suite de la conférence si elle venait à tomber en panne.

2. Routage multicast Satellite

Afin d'adapter l'usage d'un lien satellite aux réseaux de nouvelle génération et d'offrir un système de qualité pour les communications audio et vidéo sur IP, il est proposé un système de niveau applicatif pour le routage multicast sur satellites. L'avantage d'utiliser un niveau applicatif est de pouvoir ajouter un système AAA[8].

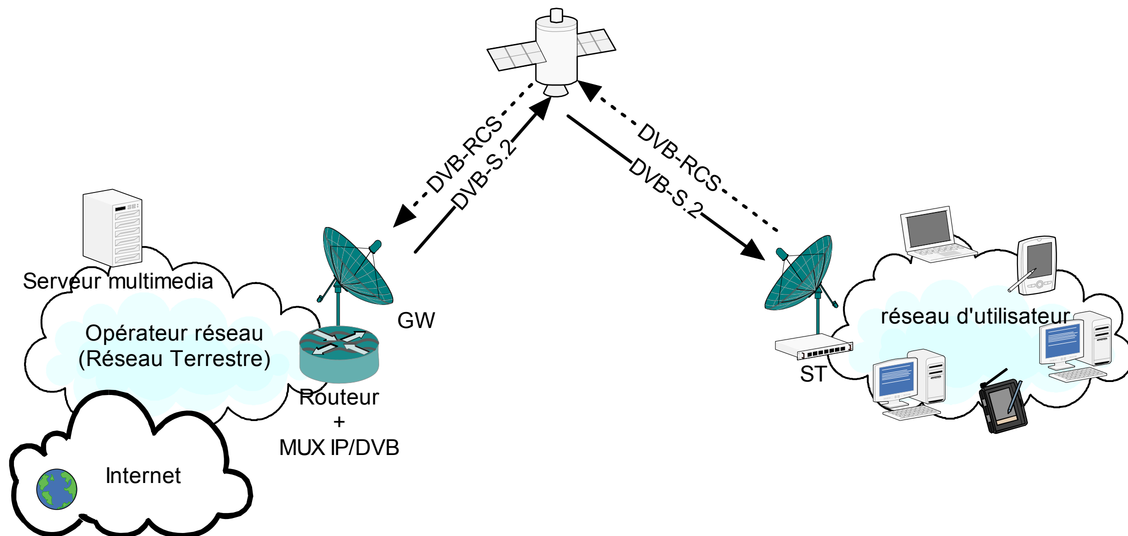

figure 2.1 : architecture d un réseau bidirectionnel classique .

Le système applicatif choisi pour gérer les connections multicast est SIP afin de respecter l'architecture IMS.

2.1 Architecture générale

le flux multicast provenant d'opérateur de contenu arrive coté GW de façon permanente. Il existe aussi des flux multicast arrivant de façon temporaire, typiquement les services de vidéoconférence.
ainsi le routeur récupère les requêtes IGMP ou PIM coté gateway afin de connaître quels flux multicast sont présents et prêts a être diffusés sur le lien satellite.

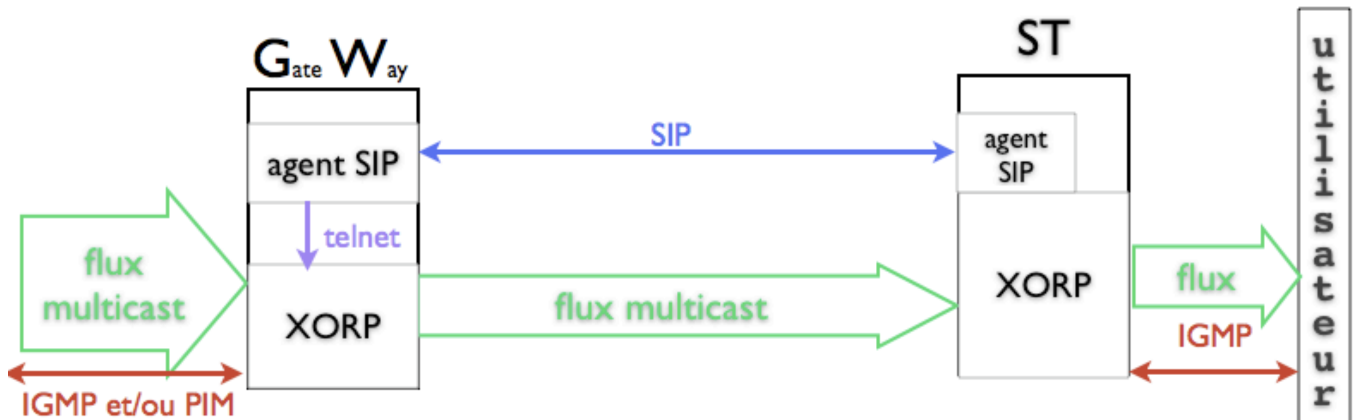

figure 2.2 : structure générale de IGMP2SIP

Sur le lien satellite la demande de flux multicast se fait non pas pour des requêtes de niveau 3 tel IGMP ou PIM, mais par un protocole applicatif en l'occurrence SIP comme le montre la figure 2.2.
Ainsi chaque ST envoie un paquet SIP INVITE pour une demande de flux multicast. Le groupe multicast correspond au nom d'utilisateur SIP invoqué par le ST.

Si l'agent SIP détermine que cette dite requête est acceptable il va alors renvoyer une réponse au ST pour lui demander de router le flux multicast. Si le flux n'est pas présent sur le lien satellite alors l'agent SIP coté GW va informer le routeur au moyen d'une route statique afin d'envoyer le flux sur le lien satellite.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.17.2	172.17.0.10	SIP	Request: INVITE sip:224.2.127.254@172.17.0.10
2	0.000823	172.17.0.10	192.168.17.2	SIP	Status: 180 Ringing
3	0.010373	172.17.0.10	192.168.17.2	SIP	Status: 404 Not Found in multicast Table

```
▷ Frame 3 (463 bytes on wire, 463 bytes captured)
▷ Ethernet II, Src: Netgear_cb:49:a0 (00:14:6c:cb:49:a0), Dst: Netgear_cb:3c:50 (00:14:6c:cb:3c:50)
▷ Internet Protocol, Src: 172.17.0.10 (172.17.0.10), Dst: 192.168.17.2 (192.168.17.2)
▷ User Datagram Protocol, Src Port: sip (5060), Dst Port: 33349 (33349)
▽ Session Initiation Protocol
  ▽ Status-Line: SIP/2.0 404 Not Found in multicast Table
      Status-Code: 404
      [Resent Packet: False]
  ▷ Message Header
```

figure 2.3 : demande refusé pour l ouverture du groupe 224.2.127.254

```
 9 0.025727   192.168.17.2   172.17.0.10    SIP   Request: INVITE sip:239.255.255.255@172.17.0.10
10 0.025942   172.17.0.10    192.168.17.2   SIP   Status: 180 Ringing
11 0.026356   172.17.0.10    192.168.17.2   SIP   Status: 200 OK

▷ Frame 9 (464 bytes on wire, 464 bytes captured)
▷ Ethernet II, Src: Netgear_cb:3c:50 (00:14:6c:cb:3c:50), Dst: Netgear_cb:49:a0 (00:14:6c:cb:49:a0)
▷ Internet Protocol, Src: 192.168.17.2 (192.168.17.2), Dst: 172.17.0.10 (172.17.0.10)
▷ User Datagram Protocol, Src Port: 33351 (33351), Dst Port: sip (5060)
▽ Session Initiation Protocol
  ▽ Request-Line: INVITE sip:239.255.255.255@172.17.0.10 SIP/2.0
      Method: INVITE
      [Resent Packet: False]
  ▽ Message Header
    ▷ From: 239.255.255.255 <sip:239.255.255.255@192.168.17.2>;tag=1846094669
      Via: SIP/2.0/UDP 192.168.17.2:33351
    ▷ To: sip:239.255.255.255@172.17.0.10
    ▷ Contact: sip:239.255.255.255@192.168.17.2:33351
      Call-ID: 1119763092@192.168.17.2
      CSeq: 1 INVITE
      Content-Type: application/sdp
      User-Agent: LIVE555 Streaming Media v2007.02.20
```

figure 2.4: demande accepté pour l ouverture du groupe 239.255.255.255

Les figures 2.3 et 2.4 montrent les captures des paquets SIP transmis dans les 2 cas les plus fréquents, a savoir l'acceptation ou le refus de la demande d'abonnement à un une adresse multicast.

2.2 Architecture cote GW

coté GW@IMS le service SIP2IGMP est en fait un agent SIP reposant sur les librairies libXml2 et sofia-SIP. LibXml2 est utilisé pour parser le PDF (policy decison function) , document écrit avec le méta-langage XML. Afin de limiter les fautes de frappe des utilisateurs, une DTD a été conçu spécialement pour le PDF. La librairie SOFIA SIP quant à elle sert à écouter les requêtes SIP provenant des différents *ST@IMS*. Enfin le routeur TISS est configuré via des routes statiques que l'agent va lui paramétrer grâce a des commandes Telnet. L'ensemble suit alors le schéma 2.5.

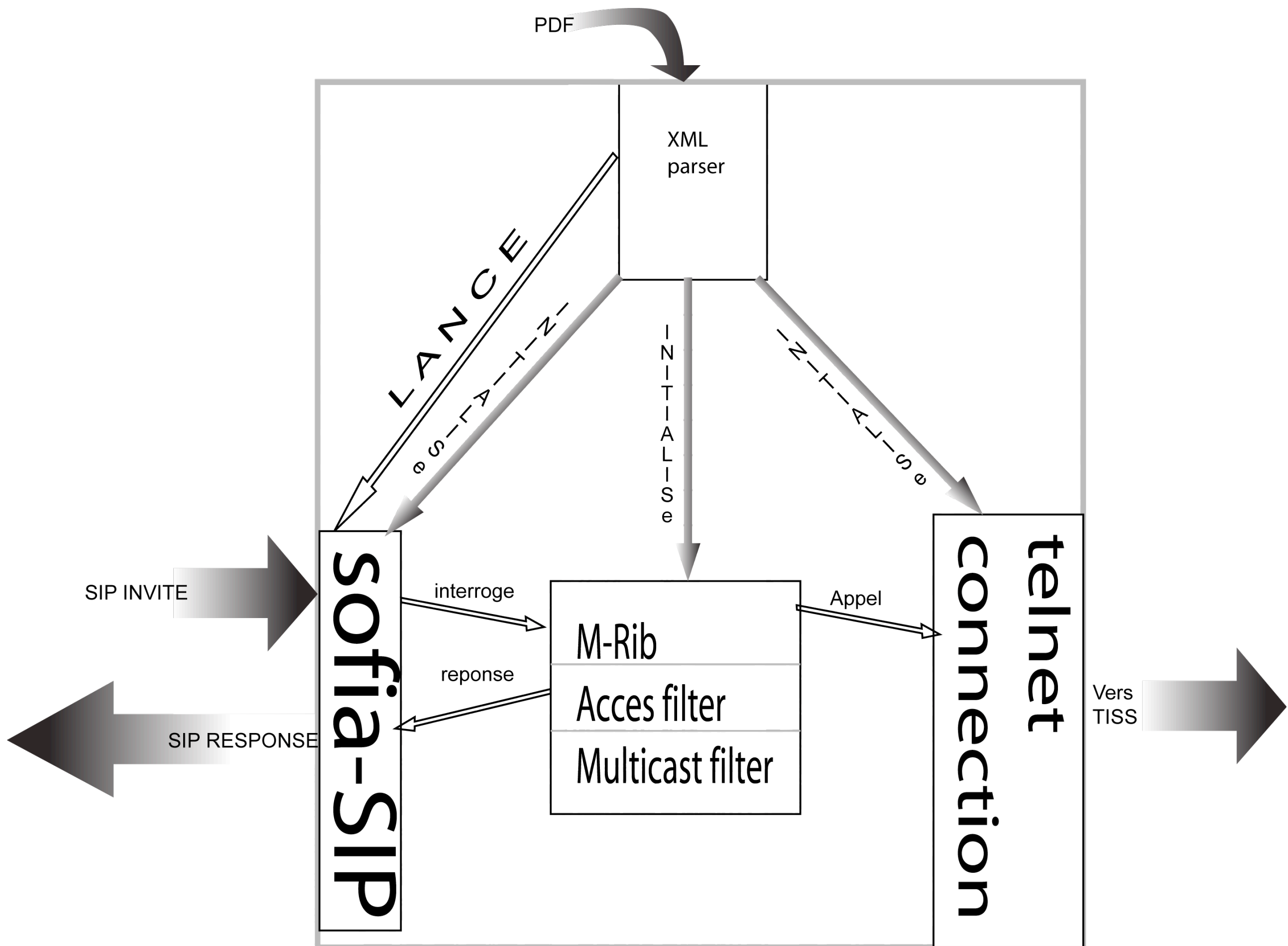

figure2.5 : structure de l agent SIP

2.3 Implémentation de l'agent SIP

• 2.3.1Parser XML

la libXml se charge de parser tout document XML avec ou non sa DTD et le représente ensuite sous forme de DOM (Document Object Model) .

```
  //XML
xmlDoc *doc =NULL;
```

```
 xmlNode * root_element=NULL;
 xmlParserCtxtPtr ctxt;
...
//on parse avec validation DTD
 xmlInitParser();

 //contexte (validation)
  ctxt= xmlNewParserCtxt();
  if (ctxt==NULL) {
       printf("Problème dans le Contexte de validation xml \n");
       return EXIT_FAILURE;
  };

 doc = xmlCtxtReadFile(ctxt,argv[1],NULL,XML_PARSE_DTDVALID);
 if (doc==NULL) {
      printf("LE fichier XML n'a pas pu etre parsé correctement\n");
      return EXIT_FAILURE;
 };
 root_element = xmlDocGetRootElement(doc);
```

LibXml se charge de la gestion des erreurs qui sont affichés sur la sortie d'erreur (stderr). En cas d'erreur lors du parsing le DOM renvoyé est Null.

une fois l'objet DOM obtenu , il ne reste plus qu'à extraire les informations de cet arbre. Le fait d'avoir vérifié ce document par une DTD nous assure de la structure de cet arbre.

```
<!ELEMENT agentSIP (User_Agent, SIP, Telnet, Client_filter, Multicast_filter)>

<!ELEMENT User_Agent EMPTY>
<!ATTLIST User_Agent    name NMTOKEN #REQUIRED>
....

<!ELEMENT Telnet (general_passwd? , conf_passwd? , interface? , add, del)>
<!ATTLIST Telnet    port NMTOKEN #REQUIRED
              method (CISCO | XORP) #IMPLIED
              addr NMTOKEN #REQUIRED >

...
<!ELEMENT Client_filter (all | net | addr)+ >
<!ELEMENT Multicast_filter (all | net | addr)+ >
<!ATTLIST Multicast_filter   verbose NMTOKEN #IMPLIED>

<!ELEMENT all EMPTY>
<!ELEMENT net EMPTY>
<!ATTLIST net ip NMTOKEN #REQUIRED
         masque (8 | 16 |24) #REQUIRED>
<!ELEMENT addr EMPTY>
<!ATTLIST addrip NMTOKEN #REQUIRED>
```

Il ne reste donc plus qu'à initialiser les informations relatives à chaque paramètre de l'agent en fonction du PDF.

```
 //reconnaissance des different √©l√©ments
 for(noeud = root_element->children; noeud; noeud = noeud->next) {
      if(! strcmp((char*)noeud->name,"User-Agent")) {
           //Action pour USer_Agent
          nua_set_params(nua,
                    NUTAG_USER_AGENT((char*)xmlGetProp(noeud, (unsigned char
*)"name")),
                TAG_END());
      }else if(! strcmp((char*)noeud->name,"SIP")) {
           // action pour SIP SIP
           SIP_parse(noeud,nua);

      }else if(! strcmp((char*)noeud->name,"Multicast_filter")) {
           //ACtion de filtre multicast
           MULTICAST_parse(noeud,table);

      }else if(! strcmp((char*)noeud->name,"Client_filter")) {
           //ACtion de filtre client
           CLIENT_parse(noeud,table);

      }else if(! strcmp((char*)noeud->name,"Telnet")) {
           //ACtion de filtre client
           TELNET_parse(noeud,connexion);
      };
 };
```

Le PDF est organisé en chapitre pour chaque élément ; ces chapitres correspondent aux fils du noeud racine du DOM. Une fois ces chapitres séparés, il suffit d'extraire les informations de l'arbre XML pour initialiser chaque élément de l' agent SIP.

Pour l'initialisation de la pile SIP, seul le port et la quantité d'information sur la console de débuggage sont paramètrables.

```
void SIP_parse( xmlNode * noeud,nua_t * nua){
   char * host = malloc(50);
   sprintf(host,"sip:0.0.0.0:%d",atoi((char*)xmlGetProp(noeud, (unsigned char
*)"port")));
   nua_set_params(nua,
            NUTAG_URL(host),
            TAG_END()
           );
   if (!strcmp((char*)xmlGetProp(noeud, (unsigned char *)"verbose"),"true"))
        verbose = 1;
};
```

Un autre chapitre du PDF est consacré au filtre Multicast. C’est dans ce chapitre que se seront inscrits quels flux multicast le *ST@IMS* a le droit de recevoir.

```
void MULTICAST_parse( xmlNode * noeud,table_multi table) {
    xmlNode * regle;
    char * addr;
    int masque;
    for(regle = noeud->children; regle; regle = regle->next) {
        ...
        if(! strcmp( (char * ) regle->name, "addr")) {
            addr = (char *) xmlGetProp( regle,  (unsigned char *)"ip");
            if( table_add_multicast_filter( table, addr, 32) != OK)
                printf(" failed to add multicast rule : allow %s\n", addr);

        }
        ...
    };
};
```

et dans “table_multicast.c”

```
struct addr {
    struct addr * suivant;
    char * addr;
    int nb_connection;
    };
struct Mrib {
    table_mode mode;
    table_storage type;
    p_addr liste;
    p_client_rules clients;
    int enable_client_rules;
    connec connexion;
};

...
table_return_code table_add_multicast_filter(table_multi table, char * adresse, int
masque/*=8,16,24,32*/) {
    p_addr liste;
    int i;
    int ip[4]={0,0,0,0};
    char * message = malloc(50);
    if (table ==NULL)
        return ERROR;
    int tour = masque /8;
    if (tour<=0) tour=1;
    if(tour>=4) tour=4;
    if (tour==4) {//on ajoute une adresse
        liste= table->liste;
        table->liste =malloc(sizeof(struct addr));
```

```
        if(table->liste ==NULL)
            return ERROR;
        table->liste->suivant = liste;
        table->liste->addr = strdup(adresse);
        table->liste->nb_connection = 0;
    }else {
        for(i=1; i<=tour; i++) {
            ip[i-1]=extract_number_of_ip_adress(adresse,i);
        };
        for(i=0; i<=255; i++){
            ip[tour+1-1]=i;
            sprintf(message,"%d.%d.%d.%d",ip[0],ip[1],ip[2],ip[3]);
            table_add_multicast_filter(table,message,(tour+1)*8);
        };
    };
    return OK;
};
```

Le PDF permet d'enregistrer les groupes multicast soit par adresse soit par réseau. Dans les 2 cas la fonction *table_add_multicast_filter(...)* va être appelée. Seul la variable masque va changer.

Ainsi pour chaque groupe multicast que l'on autorise a transiter sur le lien satellite, il va être créé dans une liste un compteur et l'adresse stockée sous forme de chaînes de caractères. Lorsque l'entrée du PDF (policy decison function) décrit un sous-réseau d'adresse, l'utilisation de la récursivité permet de facilement compléter la liste.

Un chapitre de PDF est réservé pour les filtres clients. Ce filtre est basé sur l'adresse IP du client.

```
void CLIENT_parse( xmlNode * noeud,table_multi table){
    xmlNode * regle;
    char * addr;
    int masque;
    for(regle = noeud->children; regle; regle = regle->next) {
       if(! strcmp( (char *) regle->name, "net")) {
        addr = (char *) xmlGetProp(regle, (unsigned char *) "ip");
        masque = atoi( (char *) xmlGetProp(regle,  (unsigned char *)"masque"));

        if( table_add_client_filter(table, addr,masque) != OK)
            printf(" failed to add client rule : allow %s/%d\n", addr,masque);
       }
       ...
    };
};
```

et dans "table_multicast.c"

```
table_return_code table_add_client_filter(table_multi table, char * adresse, int
masque/*=8,16,24,32*/){
	p_client_rules liste = table->clients;
	int i;

	table->clients = malloc(sizeof(*(table->clients)));
	if (table->clients ==NULL )
		return ERROR;
	for (i=1 ; i<=4 ; i++)
		table->clients->ip[i-1] = extract_number_of_ip_adress(adresse,i);

	switch (masque) {
		case 8 :
		case 16 :
		case 24 :
			table->clients->masque = masque;
			break;
		default :
			table->clients->masque = 32;
	};
	table->clients->suivant = liste;
	return OK;
};
```

Le système de filtre client est différent de celui du groupe multicast. Pour le filtre client, nous enregistrons les adresses ainsi que les sous réseau autorisés à se connecter sous forme d'un tableau de 4 éléments allant de 1 a 255 ainsi qu'un entier pour le masque . Lors d'une demande d'un client, nous allons voir si son adresse correspond à un réseau ou à une adresse déjà enregistré par comparaisons successives .

Il ne reste plus qu'à configurer le module TELNET afin qu'il prenne en compte les bons paramètres pour se connecter au routeur TISS.

```
void TELNET_parse( xmlNode *noeud,connec connexion){
	int port = atoi( (char *)xmlGetProp(noeud, (unsigned char *) "port"));
	if (port) add_port(connexion, port);

	char * addr = (char *)xmlGetProp(noeud,  (unsigned char *)"addr");
	if (addr) add_addr(connexion, addr);

	addr = (char *)xmlGetProp(noeud,  (unsigned char *)"method");
	if (addr) add_methode(connexion, addr);

	xmlNode * regle;
	for(regle = noeud->children; regle; regle = regle->next) {

		if(! strcmp((char*)regle->name,"add")) {
```

```
                addr =(char*)xmlGetProp(regle,  (unsigned char *)"order");
                if(!add_order_add(connexion,addr))
                    printf("order add non interpr√©t√© : erreur d'ecriture\n");

            }else if(! strcmp((char*)regle->name,"del")) {
                addr =(char*)xmlGetProp(regle, (unsigned char *)"order");
                if(!add_order_del(connexion,addr))
                    printf("order del non interpr√©t√© : erreur d'ecriture\n");

            }else if(! strcmp((char*)regle->name,"general_passwd")) {
                addr =(char*)xmlGetProp(regle, (unsigned char *)"value");
                if(!add_gen_passwd(connexion,addr))
                    printf("argument g√©n√©ral_password non interpr√©t√© : erreur 
d'ecriture\n");

            }else if(! strcmp((char*)regle->name,"conf_passwd")) {
                addr =(char*)xmlGetProp(regle, (unsigned char *)"value");
                if(!add_conf_passwd(connexion,addr))
                    printf("argument conf_password non interpr√©t√© : erreur 
d'ecriture\n");

            }else if(! strcmp((char*)regle->name,"interface")) {
                addr =(char*)xmlGetProp(regle, (unsigned char *)"name");
                if(!add_interface(connexion,addr))
                    printf("argument interface non interpr√©t√© : erreur d'ecriture
\n");

            };
        };
 };
```

Il y a beaucoup de paramètres pour accéder au routeur TISS. Notamment les mots de passe à inscrire, ainsi que les ordres à lui donner. Ces informations sont extraites de l'arbre DOM puis vérifier dans la forme et écrites dans l'objet connexion.

• 2.3.2 écoute et interprétation des requêtes SIP

La libraires Sofia-SIP fonctionnent à partir d'un CallBack et de l'initialisation de la pile SIP grâce aux fonctions *Nua_create()* , *nua_set_params()* , ainsi qu'aux systèmes de TAG. Un TAG correspond à l'initialisation ou la modification d'un paramètre de la Pile SIP. Ils peuvent être intégrés avec la fonction *nua_set_params(...)* et ce à n'importe quel moment.

```
 main.c function main() ligne 132 :
nua_set_params(nua,
         NUTAG_URL("sip:0.0.0.0:5060"),
         TAG_END()
           );
```

Une fois cette initialisation faite on peut appeler la boucle d'écoute grâce la fonction *su_root_run()*. La boucle d'écoute va appeler la fonction callback à chaque événement de la pile SIP. Un événement peut être un nouveau message SIP reçu ou envoyé ainsi qu'une erreur interne.

```
void event_callback(nua_event_t event, int status, char const  *phrase ,
                   nua_t * nua, nua_magic_t  * magic, nua_handle_t *nh,
               nua_hmagic_t *hmagic , sip_t const *sip,tagi_t tags[]) {

   char * multi_addr;
   char * pointeur;
   char * query_addr;

   switch(event) {
     case nua_i_invite:
          if (nh ==NULL)
               nh = nua_handle(nua,magic, TAG_END());
          //extract adresse
          multi_addr =strdup(url_as_string(NULL, sip->sip_from->a_url));
          multi_addr = multi_addr +4 ;//saute "sip:"
          pointeur =multi_addr;
          while (*pointeur!='@') pointeur++;
          *pointeur ='\0';
          query_addr = pointeur +1;
          switch (table_add(table,multi_addr,query_addr) ){
               case ERROR :
                    nua_respond(nh,500,"INTERNAL ERROR",TAG_END());
                    break;
               case NOT_FOUND:
                    nua_respond(nh,404,"Not Found in multicast Table",TAG_END());
                    break;
               case UNALLOWED:
                    nua_respond(nh,403,"Forbidden Access",TAG_END());
                    break;
               case OK:
                    nua_respond(nh,200,"OK",TAG_END());
                    break;
          };

     break;;
     case nua_i_bye:
          multi_addr =strdup(url_as_string(NULL, sip->sip_from->a_url));
          multi_addr = multi_addr +4 ;//saute sip:
          pointeur = multi_addr;
          while (*pointeur!='@') pointeur++;
          *pointeur ='\0';
          query_addr = pointeur +1;
          table_del(table,multi_addr,query_addr);
          if (nh ==NULL) nh = nua_handle(nua,magic, TAG_END());
```

```
		nua_handle_destroy(nh);
	break;;
	default:;
	};
};
```

Dans le cas de l'agent SIP seuls les événements *nua_i_invite* (réception d'un paquet SIP INVITE) et *nua_i_bye* (réception d'un paquet SIP BYE correspondant à une connexion) sont traités.

Pour l'acceptation d'un groupe multicast, le callback va appeler la fonction *table_add(...)* avec l'adresse du groupe et l'adresse du client. Cela va informer la M-RIb de la demande d'un ST, il va appliquer les règle de filtrages et incrémenter ou non le compteur du groupe multicast correspondant. Suivant la réponse de cette fonction on informe par une réponse différente le ST de l'acceptation ou non de sa requête.

```
table_return_code table_add(table_multi table  , char *multi_addr, char
*client_addr) {
	//verifie que le client est le bon
	p_client_rules liste = table->clients;
	int ip[4];
	int i;
	table_return_code code;
	if(table->enable_client_rules) {
		for (i=1 ; i<=4 ; i++)
			ip[i-1] = extract_number_of_ip_adress(client_addr,i);
	   code =UNALLOWED;
		while (liste!=NULL) {//iterative mode

			for(i=0 ; i<(liste->masque/8) ; i++) {
				if(liste->ip[i]!=ip[i])
					break;
				if(i==(liste->masque/8 -1) )code =OK;
			}
			if(code ==OK) break;
			liste = liste->suivant;
		};
		if(code !=OK ) return code;
	};

	//enfin on intègre à la MRIB
	code = add_int(table->liste,multi_addr,table);
...
	return code;
};
table_return_code add_int(p_addr liste, char *multi_addr,table_multi table) {
	if( liste==NULL) return NOT_FOUND;//pas trouv√© dans la liste
	if(!strcmp(liste->addr,multi_addr)) {
```

```
        if(liste->nb_connection==0)
            connec2add(table->connexion,multi_addr);
        liste->nb_connection++;
        return OK;
};
    return add_int(liste->suivant,multi_addr, table);
};
```

Lorsqu'un ST se désabonne d'un groupe multicast, la fonction callback informe la Mrib du départ d'un client par la fonction *table_del(...)* .

• 2.3.3 Connexion au routeur TISS

Lorsque la M-RIB de l'agent SIP en a besoin, elle va appeler l'objet connexion afin de transmettre par telnet l'ordre d'ouvrir ou de fermer une route pour le routeur TISS.
Cette connexion au routeur a été initialisée précédemment par le parser XML.
Ainsi à chaque demande de la M-RIB, l'objet connexion va initier une socket et amorcer la connexion afin de donner l'ordre au routeur TISS .

```
int connec2add(connec connection , char * adresse) {
    int sock = socket (AF_INET,SOCK_STREAM,0);
    char * message;
    const char * interface_base = "interface %s \r\n\0";
    const char * pass_base = "%s\r\n\0";
    char * order_base = strdup("%s %s     \r\n");//ctrl-Z a rajouter (5octets);
    char * ptr = order_base + strlen(order_base) - 5;
    int taille;

    printf(order_base,connection->order_add,adresse);
    if (connect(sock, &(connection->adresse), sizeof(connection->adresse))==-1)
        return 0;

   *ptr=0xff; //CTRL-Z pour telnet
   *(ptr+1)=0xed;

    *(ptr+2)=0xff;//fin d'instruction
    *(ptr+3)=0xfd;
    *(ptr+4)=0x06;
                            /*envoie du passwd*/
    taille = strlen(connection->general_passwd)+strlen(pass_base);
   message = malloc(taille);
    snprintf(message,taille,pass_base,connection->general_passwd);
    if(!send_and_wait(sock,message) )
        return 0;

            /* mode conf*/
    if(!send_and_wait(sock,"enable \r\n") )
        return 0;
```

```
    taille = strlen(connection->conf_passwd)+strlen(pass_base);
    message = realloc(message, taille);
    snprintf(message,taille,pass_base,connection->conf_passwd);
    if(!send_and_wait(sock,message) )
        return 0;

            /* configuration mode*/
    if(!send_and_wait(sock,"conf t\r\n") )
        return 0;
    taille = strlen(connection->interface)+strlen(interface_base);
    message = realloc(message, taille);
    snprintf(message,taille,interface_base,connection->interface);
    if(!send_and_wait(sock,message) )
        return 0;

            /*enfin l'ordre*/
    taille = strlen(connection->order_add)+strlen(adresse)+strlen(order_base);
    message = realloc(message, taille);
    snprintf(message,taille,order_base,connection->order_add,adresse);
   if(!send_and_wait(sock,message) )
                return 0;

            /* validation*/
    if(!send_and_wait(sock,"write-mem\r\n") )
         return 0;
    close(sock);
    return 1;
};
```

une fois la connexion établie, la procédure pour envoyer l'ordre est longue car le routeur fait plusieurs vérifications avant d'autoriser quelqu'un à lui envoyer ses ordres.

La procédure pour supprimer une route est très similaire à celle pour ajouter puisque seul l'ordre final est légèrement modifié.

2.4 Architecture cote ST

Coté Terminal Satellite, le service IGMP2SIP est composé du routeur XORP auquel est associé la librairie livemedia pour le protocole SIP. Le routeur XORP est un routeur logiciel basé sur le noyau Linux, il reprend une partie du noyau linux quant au routage unicast. Dans ce cas, il se comporte comme un simple ajout d'interprétation de protocole par rapport au noyau linux.

XORP est aussi un routeur pour paquets multicast. Il articule l'ensemble de son système de routage autour d'une M-RIB (Multicast Routing Information database) et des classes annexes pour chaque protocole de routage multicast (tel PIM ou IGMP) .

Ainsi lorsqu'un utilisateur souhaitera se connecter à un service multicast (type IPTV) , son client enverra au ST un paquet IGMP comme demande de flux multicast. Cette requête IGMP sera captée et interprétée par le routeur XORP. La décision de routage sera ensuite transmise au module PIM du routeur.
C'est dans ce module que va s'effectuer l'envoi de paquets SIP. Le code source du module est dans le dossier "*pim/*" .

Il faut tout d'abord couper l'émission de paquets PIM puisqu'il deviennent inutiles dans le lien satellite. Ceci est fait dans la fonction *PImVif::pim_send()* du fichier *pimvif.cc*.

```
int
PimVif::pim_send(const IPvX& src, const IPvX& dst,
          uint8_t message_type, buffer_t *buffer,
          string& error_msg)
{
...
    //
    // If necessary, send first a Hello message
    //
    if (should_send_pim_hello()) {
     switch (message_type) {
     case PIM_JOIN_PRUNE:
     case PIM_BOOTSTRAP:         // XXX: not in the spec yet
     case PIM_ASSERT:
         //pim_hello_first_send();
         break;
     default:
         break;
     }
    }

  ...
    //
    // Send the message
```

```
    //
    ret_value = 0;//pim_node().pim_send(vif_index(), src, dst, ttl, ip_tos,
              //       is_router_alert, buffer, error_msg);
    error_msg ="";//eviter erreur de compilation
    ...
}
```

l' arrêt de l'émission de paquet PIM se résume à commenter les 2 lignes qui envoient les paquets générés par la fonction.

Une fois l' émission de paquet PIM coupée , il reste à envoyer nos messages SIP. le module se chargeant de vérifier la véracité des routes à notre place, il suffit donc d'envoyer le message dans les bonnes fonctions et cela suffira. Ces fonctions sont dans le fichier "*pim_node.cc*" . La fonction pour disposer du flux multicast est *PimNode::add_membership(...)* et celle pour quitter une adresse de groupe est *PimNode::delete_membership(...)* .

```
int
PimNode::add_membership(uint32_t vif_index, const IPvX& source,
                const IPvX& group)
{...
        //traitement sip : FAB
     XLOG_TRACE(is_log_trace(),"SIP INVITE  to %s@192.168.100.51",cstring(group));

     client_sip = SIPClient::createNew(*_env, _desiredAudioRTPPayloadFormat);
     char * sip_message  = (char *) malloc(4+strlen(cstring(group))+1+11);
     sprintf(sip_message, "sip:%s@%s",cstring(group),"192.168.100.51");

     sip_response_code = client_sip->invite(sip_message); //anycase (error aren't
traited)
     if(sip_response_code)
        XLOG_TRACE(is_log_trace(), "sip response code is %d",
     sip_response_code);
     client_sip->sendACK();
     free(sip_message);

     if(sip_response_code==200) {
     //s'ajoute a liste des connections multicast en cours
          _connections.push_back(pair<char *,SIPClient *>(strdup(cstring
(group)),client_sip));
          join_multicast_group(1,IPvX::IPvX(cstring(group)));
          //free(sip_response_code);
          return (XORP_OK);
     }else{
          //free(sip_response_code);
          return (XORP_ERROR);
     };
}
```

La classe SIPClient est fournie par liveMedia. Elle enregistre tous les paramètres d'une session SIP tel que l'expéditeur, le destinataire, le SESSION-id ...

l'adresse de groupe est demandée par un message SIP INVITE `“sip:(@multicast)@(@GW)”` . La valeur de retour de la fonction invite correspond au code de retour envoyé par la Gateway.

Si la valeur de retour est ok , on conserve la classe dans une liste ainsi que l'adresse de groupe correspondante. Cette liste contient l'ensemble des flux multicast auquel on s'est abonné. enfin on demande à la M-RIB de router ce flux .

Afin de quitter une adresse de groupe , le routeur appelle la fonction *PimNode::delete_membership(...)* .

```
int
PimNode::delete_membership(uint32_t vif_index, const IPvX& source,
                 const IPvX& group)
{
   ...
         //traitement sip : FAB
     XLOG_TRACE(is_log_trace(),"SIP BYE to %s@192.168.100.51",cstring(group));

     std::list< std::pair<char *,SIPClient *> >::iterator it;
     std::list< std::pair<char *,SIPClient *> >::iterator a_supprimer;
     //on cherche notre petit dans la liste des connections en cours
     for(it = _connections.begin(); it != _connections.end(); it++) {
        if (!strcmp((*it).first,cstring(group))) {
          //on l'a enfin
          client_sip = (*it).second;
          a_supprimer = it;
          break;
        };
     };
     if(client_sip) {
          if(! client_sip->sendBYE()) {
               XLOG_ERROR("probleme envoi du paquet SIP");
          };
          //et ne pas oublier de l'enlever
          free( a_supprimer->first);
          _connections.erase(a_supprimer);
          leave_multicast_group(1,IPvX::IPvX(cstring(group)));
     };
    return (XORP_OK);
}
```

Pour quitter un groupe , il suffit de retrouver l'objet *SIPClient* correspondant et invoquer la fonction *sendBYE()* . Ainsi une requête va être émise à la gateway . Il ne reste donc plus qu'a supprimer l'objet de la liste.

L'envoi de requête SIP nécessite donc une liste d'objet ainsi qu'un lien vers la librairie LiveMedia. ces objets sont donc définis dans le fichier *pim_node.hh* .

```
#include "BasicUsageEnvironment.hh"
#include "GroupsockHelper.hh"
#include "liveMedia.hh"

...

class PimNode : public ProtoNode<PimVif>, public ServiceChangeObserverBase {
public:
   ...

private:

...
   //FAB SIP variables
     std::list< std::pair<char *,SIPClient *> >  _connections;
     unsigned char _desiredAudioRTPPayloadFormat;
     TaskScheduler    *_scheduler;
     UsageEnvironment *_env ;
};
```

Afin de gérer l'envoi et la réception de message la classe *SIPClient*, la librairie *LiveMedia* possède sa propre gestion de tache et d'environnement. Il faut donc stocker ces objets de *Livemedia* afin de pouvoir les réutiliser lors des créations des objets *SIPClients*.

Afin de faciliter l'intégration de *Livemedia* au routeur XORP, une modification de la valeur de retour de la fonction *SIPClient::invite(...)* a été effectuée. Au lieu de renvoyer le descriptif SDP de la session , il retourne désormais le code de retour du message INVITE. Ces modification ont eu lieu dans les fichiers *liveMedia/SIPClient.cpp* et *liveMedia/include/SIPClient.hh*.

```
int SIPClient::invite1(Authenticator* authenticator) {
...

    // NOTE: We return the SDP description that we used in the "INVITE",
    // not the one that we got from the server.
    // ##### Later: match the codecs in the response (offer, answer) #####
    if (fInviteSDPDescription != NULL)
      return atoi(fInviteSDPDescription);
```

```
  } while (0);

  fInviteStatusCode = 2;
  return 0;
}

void SIPClient::doInviteStateTerminated(unsigned responseCode) {
  fInviteClientState = Terminated; // FWIW...
  if (responseCode < 200 || responseCode > 299) {
    // We failed, so return NULL;
    delete[] fInviteSDPDescription; fInviteSDPDescription = NULL;
  }
  //fab retourne le code reponse
  delete[] fInviteSDPDescription;
  fInviteSDPDescription =new char[5];
  sprintf(fInviteSDPDescription,"%u",responseCode);
  // Unblock the event loop:
  fEventLoopStopFlag = ~0;
}
```

Ainsi la fonction *invite1(...)* correspond à la partie privée de la méthode Invite de la classe *SIPClient*. ainsi la variable dédié à la description SDP a été réutilisée pour y stocker le code de retour. Cette variable est modifiée par la méthode *SIPClient::doInviteStateTerminated(...)* qui correspond à la dernière fonction de la machine à état de la pile SIP.

Cette session SIP étant dans un usage particulier de SIP, la description SDP est donc inutile. Afin de gagner en légèreté sur le lien satellite, la transmission d'un descriptif SDP n'est plus effectué. cela se remarque à la synthèse du paquet SIP INVITE.

```
 int SIPClient::invite1(Authenticator* authenticator) {
    ...
    char* const cmdFmt =
      "INVITE %s SIP/2.0\r\n"
      "From: %s <sip:%s@%s>;tag=%u\r\n"
      "Via: SIP/2.0/UDP %s:%u\r\n"
      "To: %s\r\n"
      "Contact: sip:%s@%s:%u\r\n"
      "Call-ID: %u@%s\r\n"
      "CSeq: %d INVITE\r\n"
      "Content-Type: application/sdp\r\n"
      "%s" /* Proxy-Authorization: line (if any) */
      "%s" /* User-Agent: line */
      "Session-Expires : %u\n"
      "Content-length:0\r\n\r\n";
    unsigned inviteCmdSize = strlen(cmdFmt)
      + fURLSize
      + 2*fUserNameSize + fOurAddressStrSize + 20 /* max int len */
      + fOurAddressStrSize + 5 /* max port len */
      + fURLSize
```

```
      + fUserNameSize + fOurAddressStrSize + 5
      + 20 + fOurAddressStrSize
      + 20
      + strlen(authenticatorStr)
      + fUserAgentHeaderStrSize
     +20;
    delete[] fInviteCmd; fInviteCmd = new char[inviteCmdSize];
    sprintf(fInviteCmd, cmdFmt,
        fURL,
        fUserName, fUserName, fOurAddressStr, fFromTag,
        fOurAddressStr, fOurPortNum,
        fURL,
        fUserName, fOurAddressStr, fOurPortNum,
        fCallId, fOurAddressStr,
        ++fCSeq,
        authenticatorStr,
        fUserAgentHeaderStr,
        session_expire_timer );

    fInviteCmdSize = strlen(fInviteCmd);
```

Une dernière modification de la classe SIP a été effectué afin qu'elle interagisse sans problèmes avec le routeur XORP. En effet XORP ne permet pas à un module de rester bloqué plus d'une vingtaine de secondes , et la classe SIPClient de Livemedia est prévu pour attendre une minute la réponse à un message SIP avant de retourner TIMEOUT. Du coup si le temps de réponse est trop long , le routeur XORP va tuer le module PIM qui attend la réponse SIP. Afin de supprimer cette réaction, le temps d'attente de la classe SIPClient a été diminué à 10 secondes.

```
int SIPClient::invite1(Authenticator* authenticator) {
     ...
    fTimerB = sched.scheduleDelayedTask(20*fT1, timerBHandler, this);//fT1=500ms
```

2.6. compilation du ST.

Il faut tout d'abord compiler la dépendance de XORP qui est Live.

Une fois l'archive de Live décomprésse et patché (cmd : *patch -p1 <lieu/du/patch*) , la procédure de compilation est assez classique . (cmd : *./genMakefile linux && make*)
La librairie n'étant pas installée dans l'arborescence linux, il faudra donc indiquer à XORP où se situe la librairie Live.

Une fois l'archive de XORP décompressée , nous allons préparer la compilation par la commande *./configure*.
Nous pouvons alors patcher XORP (cmd : *patch -p1 <lieu/du/patch*). Il faut alors vérifier dans le fichier *pim/Makefile* si les chemins pour la librairie Live sont corrects. Les chemins déjà inscrits sont dans les variables CPPFLAGS et LIBS.

```
CPPFLAGS =  -I/home/comet/live/liveMedia/include -I/home/comet/live/groupsock/
include -I/home/comet/live/BasicUsageEnvironment/include -I/home/comet/live/
UsageEnvironment/include
LIBS = -lcrypto  -L/home/comet/live/liveMedia/ -lliveMedia -L/home/comet/live/
groupsock/  -lgroupsock -L/home/comet/live/BasicUsageEnvironment/  -
lBasicUsageEnvironment -L/home/comet/live/UsageEnvironment/ -lUsageEnvironment
```

Voilà le chemin par défaut */home/comet/live/* est à remplacer par l'emplacement où vous avez déposé les sources de Live.
La compilation de Xorp peut enfin commencer.

Pour exécuter le programme il suffit de lancer le routeur_Manager avec un fichier de configuration. (cmd : *./rtrmgr/xorp_rtrmgr -b chemin/vers/fichier/de/config.boot*)

2.7. Compilation et exécution de l'agent SIP.

Il faut d'abord installer les dépendances de l'agent SIP que sont LibXml2 et SOFIA SIP. Il faut installer l sinon il est toujours possible de compiler les sources, la version de sofia-sip utilisé est 1.12.5, celle de libXml2 est 2.6.20

Une fois les sources de l'agent SIP décompressées, la compilation se fait par un simple "make". Il faut toutefois vérifier si la variable d'environnement *PKG_CONFIG_PATH* soit bien positionné.
Pour exécuter l'agent SIP, il faut le lancer dans une console avec en argument le chemin vers le fichier de config. (cmd *./agent2sip config/cisco.xml*).

L'agentSIP et les différents *ST@IMS* peuvent être démarrés dans n'importe quel ordre. Il peut y avoir des problèmes si la *GW@IMS* est arrêtée alors que certains *ST@IMS* fonctionnent encore. En effet les flux auxquels seront abonnées ces *ST@IMS* ne seront pas coupés dans le réseau, même si ces *ST@IMS* se désabonnent du flux.

3. Vidéoconference multicast

3.1. Etat de l'art, choix et Principe

Afin de démontrer la viabilité de la plateforme de transport multicast par satellite, il a été choisi de générer un service de vidéo conférence dont les flux vidéos seraient diffusés par multicast.
Il existe actuellement de nombreux services et protocoles permettant la vidéoconférence , les plus connus d'entre eux sont LiveMessenger(Microsoft) , Skype, et la VOIP par SIP possédant de nombreuses implémentations notamment OpenWengo et Ekiga. Malheureusement ces produits ne supportent pas la diffusion multicast rendant leur utilisation impossible pour notre contexte.
Il existe des outils de vidéoconférence permettant de diffuser directement des flux vidéos en multicast comme VLC, vic (videoconférence client) .

Afin de s'approcher au mieux de conditions normales d'utilisation d'un satellite , il a été choisi d'utiliser un modèle centralisé autour d'un concentrateur ayant pour rôle de gérer les conférences des utilisateurs.
Le cahier des charges indiquait que le services devait s'intégrer dans un environnement multimédia de type IMS, afin de pouvoir assurer une bonne convergence avec d'autres services multimédia au besoin tel que la télévision numérique ou la voix par IP .
Ceci impliquait donc un gestionnaire de signalisation basé sur le protocole SIP qui le protocole choisi pour la signalisation d'un réseau de type IMS.
D'autres contraintes liés aux difficultés d'usage du satellite ainsi qu'aux habitudes des opérateurs indiquaient une structure centralisé pour le service de vidéoconférence. En effet la ressource étant plutôt rare au niveau d'un réseau satellite, les opérateurs souhaitent maîtriser parfaitement les flux qui y transitent. Ainsi il est très rare que ces opérateurs commerciaux acceptent qu'un de leurs clients diffuse un flux multicast ou broadcast et ceci d'autant plus que les voies retour d'un satellite (du client vers l'opérateur) ont des plus faibles débits que les voies aller. De plus les satellites ont très rarement la capacité de router les flux au niveau IP rendant ainsi le multicast aussi gourmand en bande passante que l'unicast.

Le modèle centralisé est le seul a même capable de contrôler les flux des différents usagers et d'offrir à l'opérateur des méthodes simples de tarification. Il en résulte donc une architecture telle que le montre la figure 3.1 :

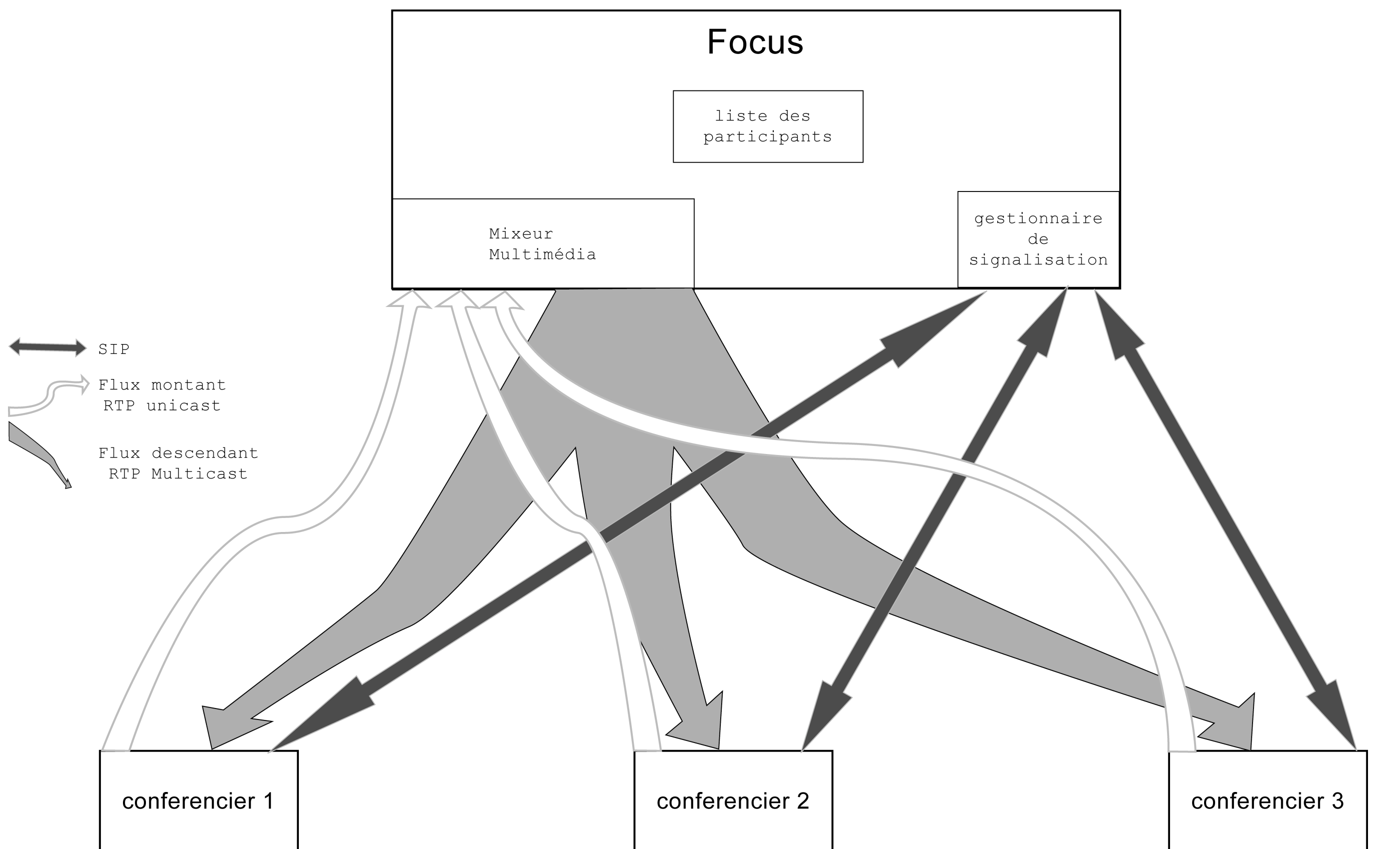

Figure 3.1 : structure du service de vidéoconférence.

Une fois chaque utilisateur identifié auprès du concentrateur, il envoie son flux de manière unicast et le concentrateur se chargera de mixer le flux de l'ensemble des utilisateurs et de le retransmettre en multicast. Le concentrateur est chargé de répartir les flux, de prévenir les autres utilisateurs du départ ou de l'arrivé de l'un d'eux grâce à la signalisation mise en place.

3.2. Architecture générale

Après les différentes contraintes qui nous étaient imposées, un modèle de conférences couplées centralisées nous était imposé. L'IETF propose une standardisation d'un système de conférence pouvant correspondre à nos attentes. Il s'agit d'un complément au protocole SIP décrit par la RFC 4353[9] permettant de mettre en place un système de conférence centralisé . Le fait que cet RFC n'utilise que le protocole SIP pour sa signalisation permet de rester totalement conforme avec un développement dans le cadre d'un réseau IMS.

Dans ce modèle, chaque participant appelle le serveur de conférence avec un nom de conférence commun. Pour bien les distinguer chaque conférencier devra porter un nom différent et se verra refuser l'accès si le nom choisi est déjà présent dans la conférence. Pour éviter des attaques DOS (Deny Of Services) et afin que le serveur puisse correctement fournir le service au utilisateur, un nombre maximal de conférences et d'usagers ont été définis. Ces limites dépendent fortement de la machine supportant le concentrateur. Ainsi l'accès au service de vidéoconférence se résume au schéma Figure 3.2 :

_ L'utilisateur va entrer sur son logiciel le nom de la conférence qu'il souhaite rejoindre ainsi que son pseudonyme.

_le concentrateur (dont l'adresse est connue à l'avance par tous les clients et de fait inscrite dans le logiciel) va vérifier l'existence de la conférence et la créer au besoin.

_le concentrateur vérifie ensuite que le nom d'utilisateur n'est pas déjà présent dans la conférence auquel cas cela entraînera un refus.

_enfin le concentrateur accorde l'accès à la conférence et délivre les informations nécessaires pour y participer.

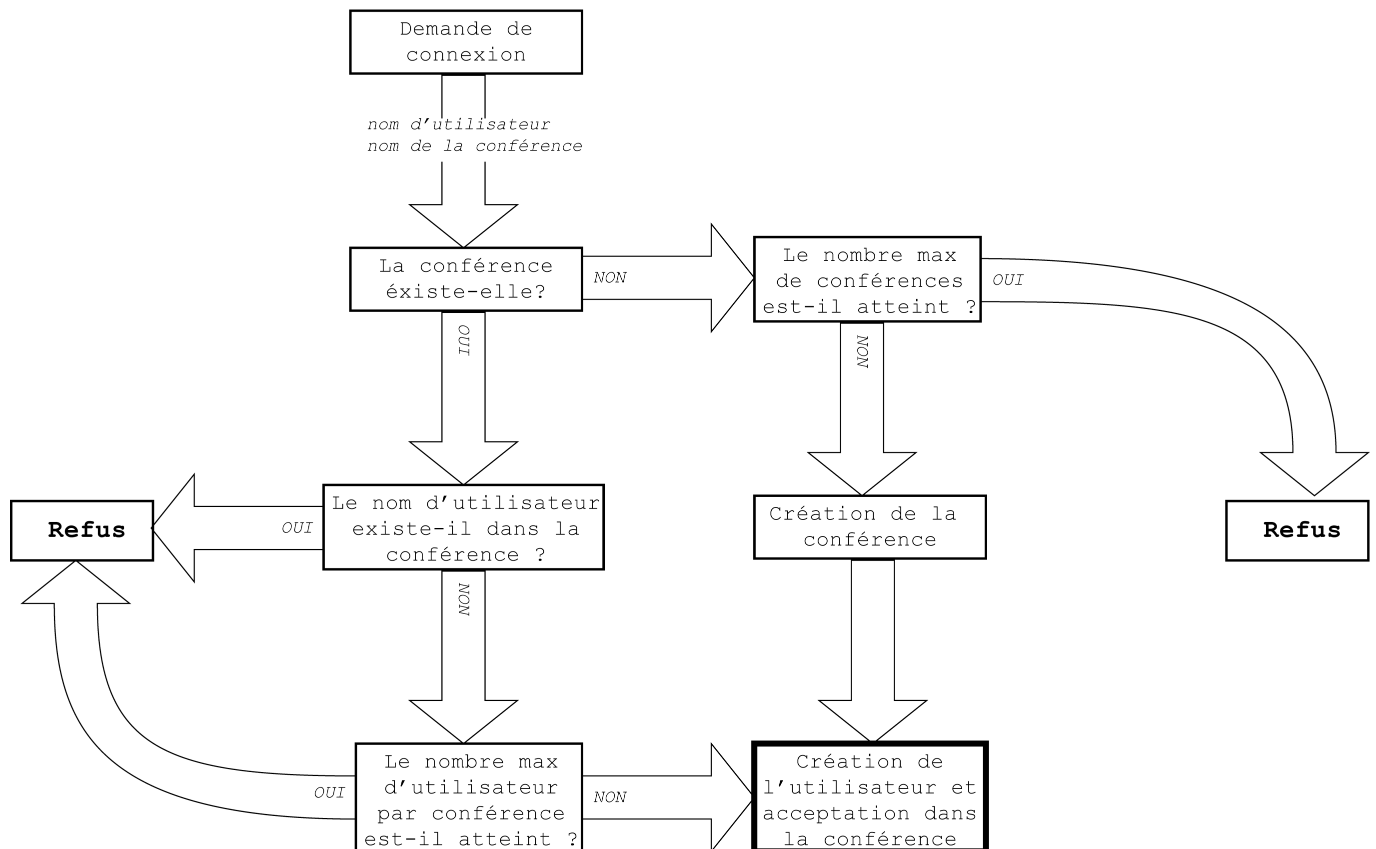

Figure 3.2 : schéma fonctionnel de connexion pour l utilisateur

• 3.2.1 mécanismes réseau

Tout ce mécanisme de négociation se fait à partir du protocole de signalisation SIP et des normes décrites dans la RFC 4353. La norme est basé sur la gestion d'événements de présence pour gérer le départ et l'arrivée des utilisateurs ainsi que sur le schéma classique d'un appel Voix sur IP proposé par le protocole SIP non étendu (RFC3261) pour l'accès à la conférence.

L'accès à la conférence se fait par l'envoi d'un message INVITE à l'adresse *Nom_de_la_conférence@Adresse_du_concentrateur*. Le concentrateur répond par un message contenant un code de retour. Ces codes sont numérotés et regroupés par groupes de centaines. Ainsi tout code de retour compris entre 400 et 500 signifie un refus de connexion. Le code signifiant l'acceptation est le numéro 200. Il est généralement suivi d'un document SDP contenant les informations nécessaires pour diffuser son flux vidéo aux membres de la conférence.

Le départ de la conférence se fait par l'envoi d'un message BYE au concentrateur par l'usager.

Le concentrateur va créer un gestionnaire d'événement auquel vont s'abonner chaque membre de la conférence. Les communications pour ce service sont gérés par 2 types de messages SIP :

_SUSCRIBE pour la souscription des usagers au gestionnaire d'événement.

_NOTIFY pour prévenir tout utilisateur d'un changement de status d'un autre conférencier.

La seule particularité par rapport à un gestionnaire d'événement décrit par la RFC3265 correspond au contenu du message NOTIFY. le contenu XML décrivant l'état de l'utilisateur ne suffisait pas pour connaître l'accès au flux des autres utilisateurs, nous avons donc remplacé le XML par un fichier texte permettant de décrire l'état d'un utilisateur et ses paramètres pour accéder aux flux. Ce fichier ressemble à un fichier SDP dans lequel sont notés le nom de l'utilisateur, le status, l'adresse multicast ainsi que le numéro de port.
L'ensemble de ces messages permet le déroulement d'une conférence comme le montre la figure 3.3 . Le scénario proposé est une conférence avec 3 participants ou le 3 ème participant quitte la conférence sitôt après y être entré.

USER 1
je demande la conférence.
jz suis connecté
User 2 vient d'arriver
j'envoie un message que je reçois
User 3 vient d'arriver
User 3 quitte la conférence

Focus

USER 3
je demande la conférence.
jz suis connecté
User 1 et User 2 sont déjà present.
je quitte la conférence.

USER 2
je demande la conférence.
jz suis connecté
User 1 est déjà present.
Je reçois un message de User 1
User 3 vient d'arriver
User 3 quitte la conférence

Mesage INVITE
Mesage SUSCRIBE
Message NOTIFY
Message BYE

Figure 3.3 : diagramme temporel des paquets émis sur le réseau

• <u>3.2.2 Gestion des flux vidéos</u>

En effet la gestion des flux vidéo est primordiale pour obtenir un système efficient. Le choix du multicast permet ainsi aux réseaux et de fait aux satellites de minimiser la charge. Il est à noter que le concentrateur est seul chargé de la gestion des flux vidéos, les différents clients n'ont qu'à suivre les indications fournies de différentes manières par le concentrateur.

Un utilisateur est supposé intéressé par tous les flux de sa conférence donc tous les flux d'une conférence sont regroupés sur une même adresse multicast. Chaque conférence possède ensuite une plage de port sur cette adresse multicast. Cette plage a une taille déterminée à l'avance correspondant aux capacités réseaux du concentrateur.

Ainsi chaque nouveau client va se voir offrir un numéro de port parmi cette plage pour diffuser son flux vidéo, il reçoit aussi un numéro de port afin de diffuser en unicast son flux vers le concentrateur.

Lorsqu'un utilisateur quitte la conférence, les ports qui lui étaient attribués sont libérés et mis a disposition des futurs utilisateurs. Si cet utilisateur est le dernier de la conférence celle-ci est détruite, et la plage entière de port est libéré.

• <u>3.2.3 choix généraux d'implémentation</u>

L'implémentation du service de vidéoconférence a été basé sur le logiciel VLC. VLC est un logiciel libre écrit en C/C++ capable de lire les vidéos sur la plupart des formats d'encapsulation et la plupart des supports tel qu'un fichier, un flux DVD, un flux UDP unicast ou multicast, ou autre flux réseau. Il permet aussi de diffuser sur le réseau n'importe quel flux qu'il est capable de lire au préalable.

Une particularité de VLC est d'être un logiciel modulaire (voir figure 3.4) , facilitant ainsi le développement. Pour ajouter des fonctionnalités à VLC , il suffit donc de modifier ou d'ajouter un ou plusieurs modules. Ainsi chaque action pour la lecture d'une vidéo est effectuée par un module spécifique, le coeur de VLC n'a plus qu'à choisir les modules pour construire la chaîne de décodage d'un flux vidéo.

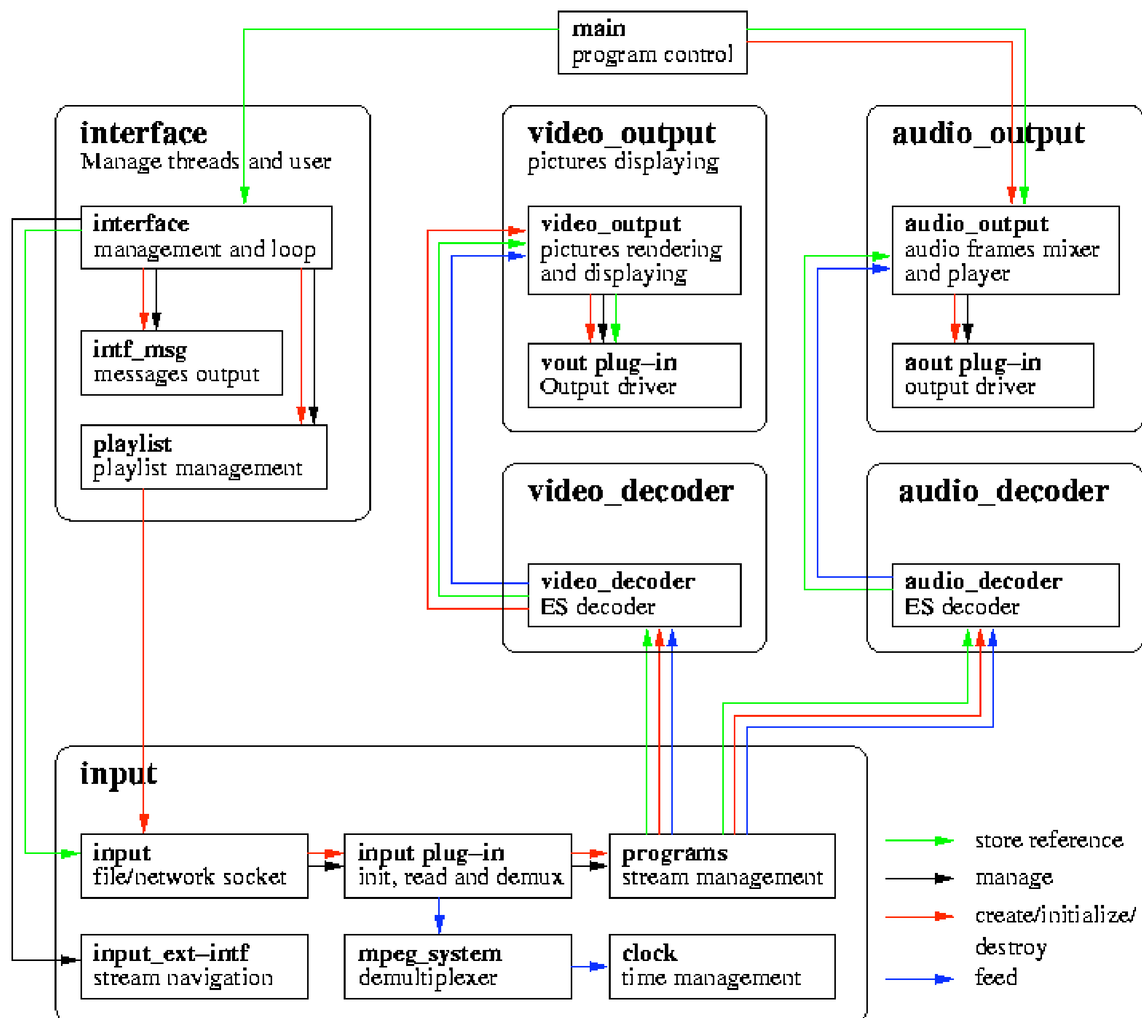

figure 3.4 : schéma bloc du logiciel VLC (source : www.videolan.org)

Par exemple la lecture d'un flux UDP encodé en MP4-MP3 et encapsulé en MPEG_TS nécessite le module d'accès UDP puis le module de désencapsulation MPEG_TS. Celui ci va diffuser 2 flux :

_ un flux audio décodé par le module de décodage MP3 et lu ensuite par le module chargé de l'interface avec la carte son.

_ un flux vidéo décodé par le module de décodage MP4 et lu ensuite par le module chargé de l'affichage.

Afin de préserver la synchronisation, les modules sont organisés sous forme de "callback" ou appel en retour. Ainsi c'est le dernier module de la chaîne de traitement (affichage et/ou sonorisation) qui va appeler les modules précèdent dans la chaîne de traitement. Et ainsi de suite chaque module va appeler le modules précèdent dans la chaîne de traitement.

Là encore des pertes peuvent survenir avec ce système, donc un système de tampon a été mis en place entre chaque module.

Pour la partie cliente, nous avons utilisé un fork nommé VLVC[10]. VLVC est une version modifié de VLC permettant la vidéoconférence. Le modèle de conférence choisi par VLVC est un modèle centralisé où la signalisation est séparé des flux vidéos. Bien que la signalisation proposé par VLVC ne soit pas compatible SIP, elle reste une signalisation texte facilement modifiable.

Afin de faciliter l'implémentation du protocole SIP nous avons utilisé une librairie développé par NOKIA nommé Sofia-SIP[11]. Cette libraire supporte les principales normes autour du protocole SIP, notamment la gestion des événements de présence. Cette librairie a une phase d'initialisation et fonctionne ensuite avec un appel en retour facilitant ainsi l'intégration en tant que module de VLC.

3.3 Implémentation coté client.

Pour le déploiement du service coté client, nous avons pris pour base le logiciel VLVC auquel nous avons seulement conservé le module d'interface graphique. Un module d'accès a été créé pour gérer la signalisation SIP. La gestion des flux vidéos étant faites par le VLM : VideoLan Manager. C'est l'interface de configuration des actions de VLC, elle est accessible soit par appel de fonctions , soit par connexion telnet. L'ensemble de cette structure est représenté figure 3.5

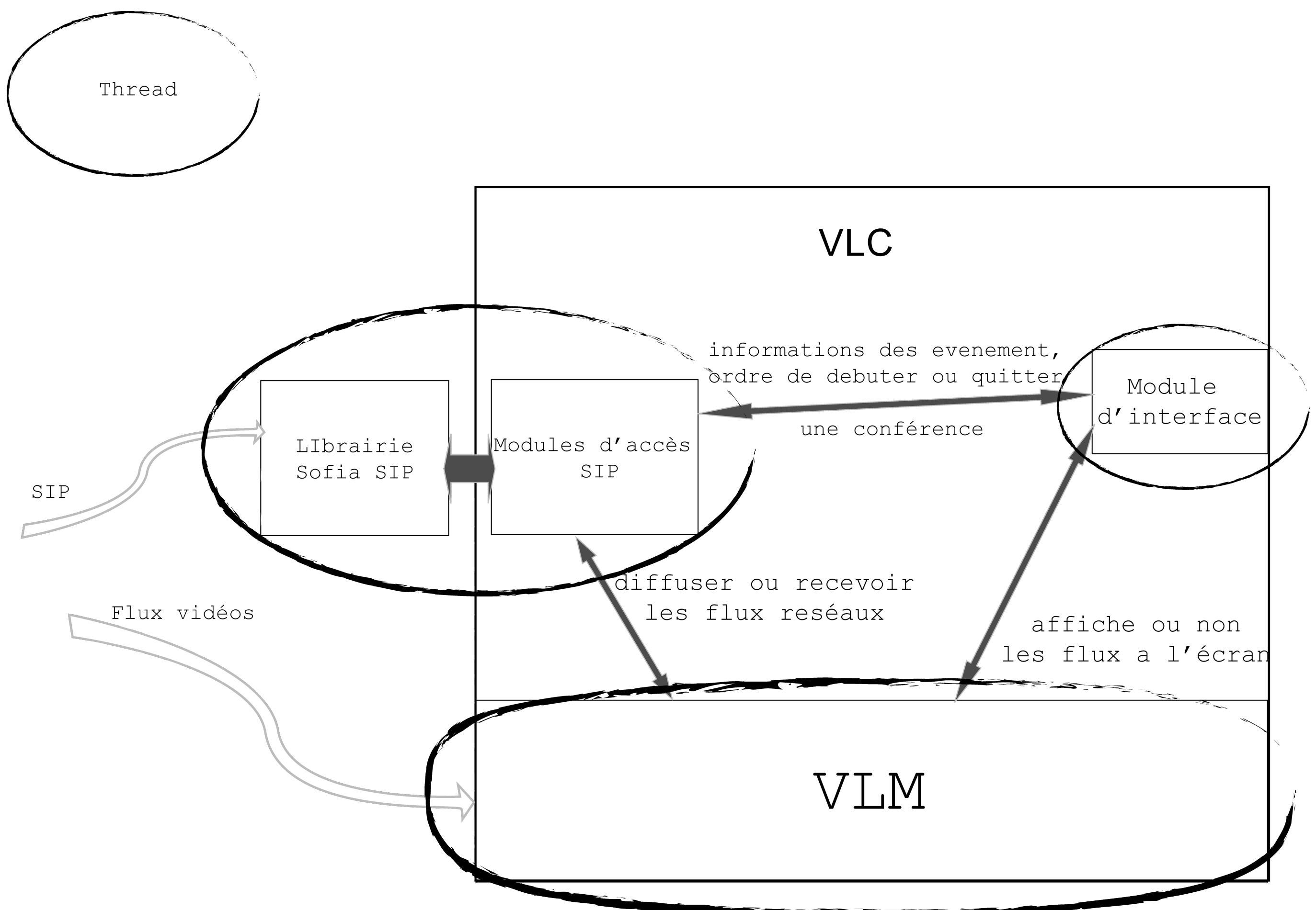

Figure 3.5 : organisation des modules concerné pour la vidéoconférence coté client.

Afin que chaque module effectue sa tache convenablement, ils sont exécutés sur des threads séparés. Le 1er module exécuté étant VLM, il est chargé de la création des threads pour l'interface et la signalisation. La synchronisation s'effectue ensuite en fonction du module de signalisation.

• <u>3.3.1 VLM ou VideoLanManager</u>

C'est aussi le coeur du logiciel VLC, programmable par telnet ou appel de fonction. Il permet de pouvoir gérer plusieurs flux vidéos avec une même instance de VLC. IL ne connaît que 2 types d'objets mais leurs possibilités sont quasiment infinies.

Il existe en premier lieu l'objet "Broadcast" : c'est l'objet permettant la diffusion d'un flux vidéo d'une source vers une destination.

La source peut être un périphérique vidéo (WebCam, carte d'acquisition DVB, carte d'acquisition analogique), un fichier ou un flux réseau. elle est précisée grâce au mot clé "input".

La destination peut être un flux réseau HTTP, UDP, RTP, multicast-UDP ou multicast-RTP ou un fichier. Il est possible de préciser le format d'encaspulation de la vidéo et de l'audio parmi les formats suivant OGG, MPEG2-TS, AVI, MPEG4. Il est aussi possible de transcoder l'audio et/ou la vidéo à la volée. Toutefois certains codecs sont limités à certains format d'encapsulation. La commande pour préciser les options de sorties est "output".

l'objet "Broadcast" est donc très souple d'utilisation comme le montre le tableau 3.6 :

diffusion simple d'un fichier vers l'ecran	new test broadcast enabled setup test input file://ma_video.mp4 setup test output #duplicate{dst=display} control test play
diffusion simple d'un fichier sur un groupe multicast	new test broadcast enabled setup test input file://ma_video.mp4 setup test output #duplicate{dst=std {access=rtp,mux=ts,dst=239.0.1.5:2002}} control test play
diffusion d'un flux unicast vers un flux multicast	new test broadcast enabled setup test input udp://@:2002 setup test output #duplicate{dst=std {access=rtp,mux=ts,dst=239.0.1.5:2002}} control test play
capture d'une carte DVB vers un fichier.	new France4 broadcast enabled setup France4 input "dvb/ts:adapter=0:frequency=490000000:bandwidth=8" setup France4 option :program=259 setup France4 output #duplicate{dst=std{access=file,mux=mp4,dst=/ ma_video.mp4}} control France4 play

exemple de transcodage	new test broadcast enabled setup test input file://video_brut.mp4 setup test output #transcode {vcodec=mp4v,vb=768,acodec=mp4a,ab=64}:standard {mux=mp4,dst=/video_reencode.mp4,access=file} control test play

figure 3.6 : exemple d objet broadcast

Le principal défaut de l’objet “Broadcast” est que seul le serveur (c’est a dire VLM) a le choix de diffuser ou non le flux. Afin que le client puisse interagir sur un flux , VLM propose un autre type d’objet : VOD (Video On Demand).

Cet objet peut prendre les mêmes sources en entrée qu’un objet “broadcast”. La sortie est forcement en RTP unicast, le système de contrôle du flux est RTSP.
L’objet Vod est donc spécialisé dans la diffusion en réseau, de plus chaque client est indépendant vis-à-vis des autres clients lisant le même flux. Ainsi, lors d’une demande de lecture, la vidéo commence à son début lorsque c’est possible. Il peut aussi stopper le flux , le mettre en pause sans que cela n’interfère la lecture des autres usagers.
De plus le protocole de contrôle du flux permet des négociations particulières telles que la transmission de clés de décryptage si la vidéo est crypté.

• 3.3.2 module d’interface

La librairie graphique utilisé par VLC pour wxWidget[12] ; elle a l’avantage d’être écrit en C/C++ et d’avoir des rendus similaires sous MacOS X, Linux et windows. L’interface est écrit sous forme d’un module VLC, placé dans le répertoire source *modules/gui/wxwidgets*. Il est décomposé en 2 sous-ensemble, l’interface lié au lancement de l’application et celle lié au déroulement de la conférence.

Au lancement du logiciel , l’interface est similaire à VLC, à l’exception d’un panneau CONFERENCE qu’on a rajouté dans le menu ouverture comme le montre la figure 3.7.

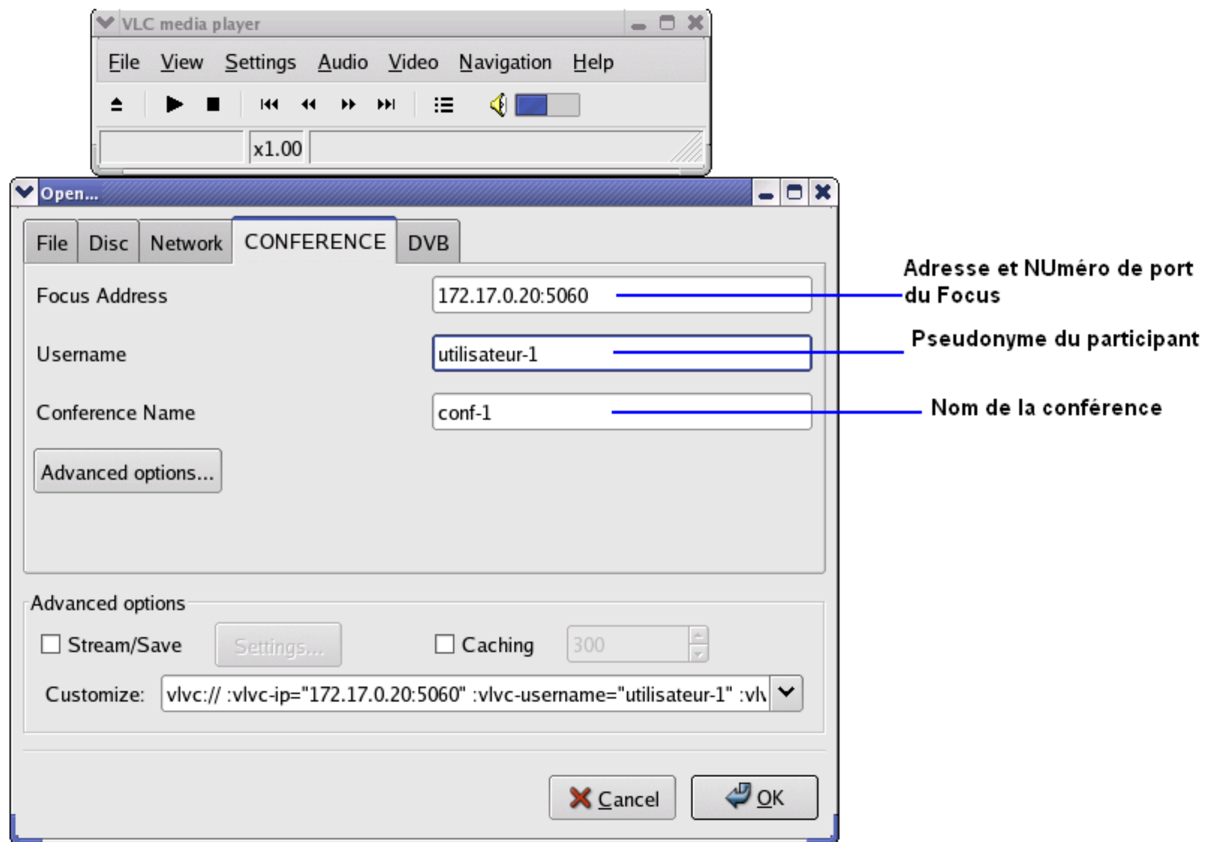

figure 3.7 : interface de choix de la conférence et du pseudonyme

Le code correspondant à la fenêtre d'ouverture de flux se situe dans les fichiers *modules/gui/wxwidgets/dialogs/open.cpp* et *modules/gui/wxwidgets/dialogs/open.hpp*. Pour chaque type de flux , un panneau est prévu dans le code. Ce panneau n'est affiché que si le module permettant d'ouvrir le type de flux est présent à l'exécution.

```
p_module = config_FindModule( VLC_OBJECT(p_intf), "vlvc" );
   if( p_module )
   {
       AutoBuiltPanel *autopanel =
           new AutoBuiltPanel( notebook, this, p_intf, p_module );
       input_tab_array.Add( autopanel );
       notebook->AddPage( autopanel, wxU( p_module->psz_shortname ?
                       p_module->psz_shortname : p_module->psz_object_name ),
                          i_access_method == CAPTURE_ACCESS );
   }
```

Un pseudo-module "vlvc" a donc été crée pour l'accès à la conférence. Il est uniquement là pour distribuer les informations issues de l'utilisateur au module de communication SIP ainsi qu'à l'interface graphique. Ce module est un vestige de notre point de départ , c'est à dire le logiciel VLVC.

Une fois que l'utilisateur a entré le nom de la conférence et son pseudonyme , il peut presser le bouton OK. Il faut remarquer que l'adresse du focus/concentrateur est disponible à la modification si l'utilisateur le souhaite.

Une fois le bouton OK pressé, le mécanisme de connexion au serveur s'enclenche :

```
 void OpenDialog::OnOk( wxCommandEvent& WXUNUSED(event) )
{
    bool b_is_vlvc = false;
    mrl = SeparateEntries( mrl_combo->GetValue() );

    if ( !strcmp( "vlvc://", wxFromLocale( mrl[0] ) ) )
    {
        b_is_vlvc = true;
    }

    .....

    vlc_object_release( p_playlist );

    if ( b_is_vlvc == true )
    {
        new VlvcFrame( p_intf, this );
    }

    Hide();

    if( IsModal() ) EndModal( wxID_OK );
}
```

L'interface graphique du déroulement de la conférence est crée puis controlé par la classe *VlvcFrame*. Cette classe est écrite dans les fichiers *modules/gui/wxwidgets/dialogs/vlvc/vlvc_frame.cpp* et *modules/gui/wxwidgets/dialogs/vlvc/vlvc_frame.hpp* . la classe *VlvcFrame* hérite de la classe servant à la création de fenêtre de la librairie Wxwidgets. Elle contient tous les objets nécessaires à la création de l'interface utilisateur, ainsi que des pointeurs vers les objets/modules de VLC. Ceux-ci sont nécessaires au bon déroulement de la conférence comme la définition ci dessous :

```
 class VlvcFrame: public wxFrame
    {
    public :
        vlvc_t          *p_vlvc;
     void             *handle;

        VlvcFrame( intf_thread_t *p_intf, wxWindow *p_parent);
        virtual ~VlvcFrame();
```

```
         nua_t              *nua;
        su_root_t          *root;
     client             new_client;
     int                index;
        char               my_name[VLVC_MAX_USER_NAME_LENGTH];

  protected:
      intf_thread_t            *p_intf;
      wxWindow                 *p_parent;
      char                     p_tab_status[VLVC_MAX_USERS];
      bool                     b_first_added;

      /* Widgets */
      wxListBox                *p_user_list;
      wxTextCtrl               *p_chat_textbox;
      wxTextCtrl               *p_entry_textbox;

      /* Widget callbacks functions */

      void                     OnButtonSendPress( wxCommandEvent& event );
      void                     OnButtonPlayPress( wxCommandEvent& event );
      void                     OnButtonStopPress( wxCommandEvent& event );

  public :

      void                     InitControls();
      wxPanel                  *GetMainPanel( wxFrame *parent );
      wxPanel                  *GetTopPanel( wxPanel *parent );
      wxPanel                  *GetChatPanel( wxPanel *parent );
      wxPanel                  *GetActionPanel( wxPanel *parent );
      wxPanel                  *GetUserPanel( wxPanel *parent );
      wxPanel                  *GetBtnPanel( wxPanel *parent );
      wxPanel                  *GetSendPanel( wxPanel *parent );

      void                     ChatMessage( wxString &message, wxColour &color,
bool bShowTimeStamp = true );

      void                     OnMessage( wxCommandEvent& );
      void                     OnAddUser(wxCommandEvent &evt);
      void                     OnDelUser(wxCommandEvent &evt);
      void                     OnChatMessage(wxCommandEvent &evt);

      MessageEventData         messageData;
      ChatMessageEventData     chatMessageData;
      vlc_mutex_t              messageLock;
      int                      b_messageLocked;
      vlc_mutex_t              chatMessageLock;
      int                      b_chatMessageLocked;


      DECLARE_EVENT_TABLE();
```

```
};
```

Lors de la création de l'objet correspondant à la classe, le constructeur va être chargé de se connecter au serveur via le module de communication.
A la réponse négative du serveur, il devra générer un message d'erreur.
A la réponse positive du serveur, il devra générer l'interface graphique tel que l'on peut la voir figure 3.8.

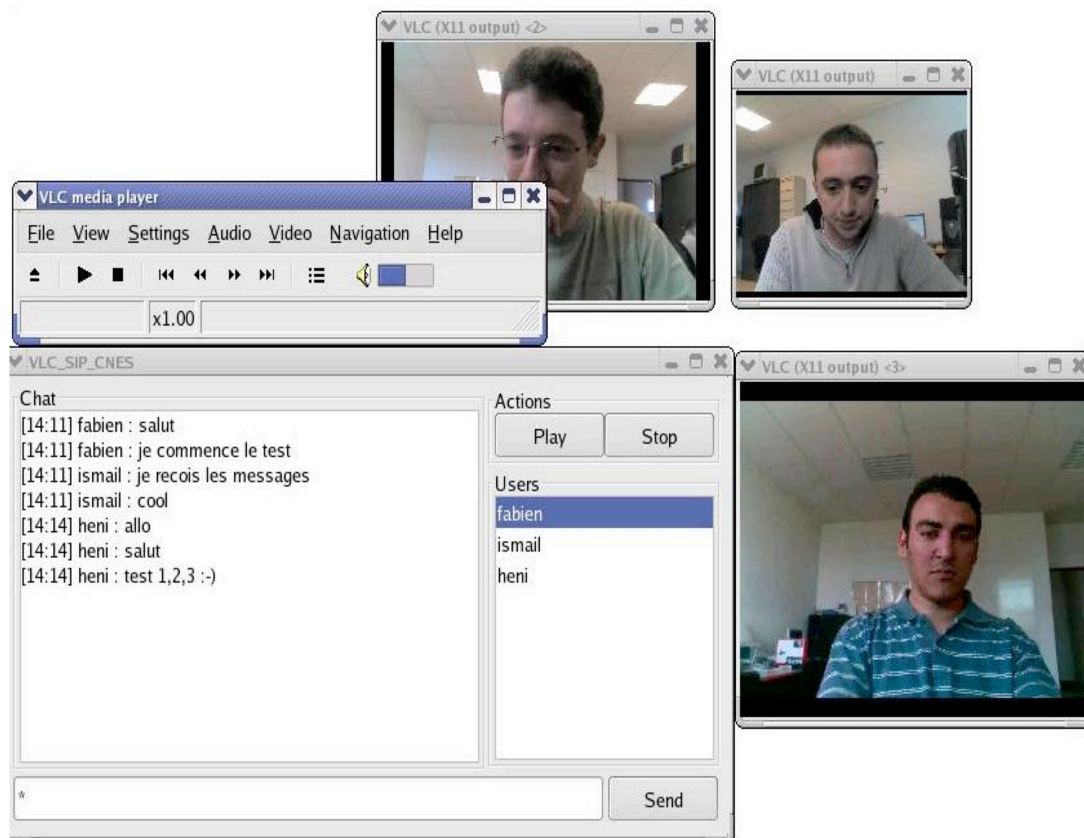

Figure 3.8 : Vidéoconférence en action; interface de l usager fabien.

Ainsi la construction de l'objet *Vlvcframe* se fait en plusieurs étapes. Il faut d'abord récupérer les informations issues du lancement de VLC ainsi qu'initialiser les différents Mutex :

```
VlvcFrame::VlvcFrame( intf_thread_t *p_intf, wxWindow *p_parent ) :
    wxFrame( p_parent, -1, wxT(VLVC_WINDOW_NAME), wxDefaultPosition,
    wxDefaultSize, wxCAPTION | wxMINIMIZE_BOX | wxRESIZE_BORDER )
{
memset( (void*)p_tab_status, VLVC_USER_STATUS_OFF, VLVC_MAX_USERS );
    p_user_list = NULL;

    vlc_mutex_init( p_intf->p_vlc, &this->messageLock );
    vlc_mutex_init( p_intf->p_vlc, &this->chatMessageLock );
    this->b_chatMessageLocked = VLC_FALSE;
    this->b_messageLocked = VLC_FALSE;
```

```
    this->p_intf = p_intf;
    this->p_parent = p_parent;

    p_vlvc = vlvc_New( p_intf );
    b_first_added = VLC_FALSE;
    InitControls();

    while ( p_vlvc->b_initialized == VLC_FALSE ) msleep( 10000 );
```

Le module vlvc activé à l'ouverture de la conférence et décrit précédemment ne fait que stocker les informations sur la structure *p_vlvc* puis place le marqueur *p_vlvc->b_initialized* a VLC_TRUE lorsqu'il a fini.
La fonction *InitControls()* permet de remplir la fenêtre de l'interface. Cette fonction va construire un tableau invisible (objet *wxBoxSizer*) et ensuite appeler les fonctions qui vont remplir les cases du tableau notamment *GetMainPanel(...)* et *GetTopPanel(...)* .

```
void VlvcFrame::InitControls()
{
    wxBoxSizer *p_rootsizer = new wxBoxSizer( wxHORIZONTAL );
    p_rootsizer->Add( GetMainPanel( this ), 1, wxEXPAND | wxALL, 5 );
    this->SetAutoLayout( TRUE );
    this->SetSizerAndFit( p_rootsizer );
    this->Show();
}

wxPanel *VlvcFrame::GetMainPanel( wxFrame *parent )
{
    wxBoxSizer *p_mainsizer;
    wxPanel    *p_mainpanel;

    p_mainpanel = new wxPanel( parent, -1 );
    p_mainsizer = new wxBoxSizer( wxVERTICAL );
    p_mainsizer->Add( GetTopPanel( p_mainpanel ), 1, wxEXPAND );
    p_mainsizer->Add( GetSendPanel( p_mainpanel ), 0, wxEXPAND );
    p_mainpanel->SetSizerAndFit( p_mainsizer );
    p_mainpanel->SetMinSize( wxSize( VLVC_WINDOW_WIDTH, VLVC_WINDOW_HEIGHT ) );
    return p_mainpanel;
}

wxPanel *VlvcFrame::GetTopPanel( wxPanel *parent )
{
    wxBoxSizer *p_top_sizer;
    wxPanel    *p_top_panel;

    p_top_panel = new wxPanel( parent, -1 );
    p_top_sizer = new wxBoxSizer( wxHORIZONTAL );
    p_top_sizer->Add( GetChatPanel( p_top_panel ), 1, wxEXPAND );
    p_top_sizer->Add( GetActionPanel( p_top_panel ), 0, wxEXPAND );
```

```
    p_top_panel->SetSizerAndFit( p_top_sizer );
    return p_top_panel;
}
...
```

Ensuite le constructeur va créer le module de signalisation "*sip client*" puis attendre que le module de signalisation soit bien lancé grâce à la condition *sip_agent->init_cond*.

```
this->handle=NULL;
     vlc_object_t  * p_sip =vlc_object_find(p_vlvc, VLC_OBJECT_SIP, FIND_ANYWHERE);
          if (p_sip==NULL)
            {    (p_vlvc->p_vlm)->sip = vlc_object_create( p_vlvc, VLC_OBJECT_SIP );
          vlc_object_attach((p_vlvc->p_vlm)->sip, p_vlvc );
          ((p_vlvc->p_vlm)->sip)->p_module = module_Need((p_vlvc->p_vlm)->sip, "sip
client", 0, 0 );
            }

...

   //wait for the initialisation of NUA
    vlc_mutex_lock(& sip_agent->init_lock);
    vlc_cond_wait( & sip_agent->init_cond, & sip_agent->init_lock);
    vlc_mutex_unlock(& sip_agent->init_lock);
```

enfin le constructeur va tenter de se connecter au serveur.

```
this->nua=sip_agent->nua;
   sip_agent->handle= this->handle = nua_handle(this->nua,NULL,
                   SIPTAG_FROM_STR(client),
                   SIPTAG_TO_STR(conf_uri),
                   TAG_END());
    nua_invite (this->handle,
              SOATAG_USER_SDP_STR(sdp_msg),
          SOATAG_RTP_SORT(SOA_RTP_SORT_REMOTE),
          SOATAG_RTP_SELECT(SOA_RTP_SELECT_ALL),
              NUTAG_EARLY_MEDIA (1),
              TAG_END ());
```

La réponse de ce message invite est reçu et traité par le module de signalisation.

• 3.3.3 module de signalisation

Autrement appelé "*sip client*", le module de signalisation est chargé de capter tous les messages SIP en direction du client. Il est aussi chargé de transmettre au concentrateur les messages fournis par l'utilisateur au travers de l'interface graphique.

Comme tous modules de VLC, le module "*sip client*" possède une fonction *Open()* et *Close()* ainsi qu'une structure pour y stocker ses données. Ces fonctions sont appelées respectivement a l'ouverture et a la fermeture de la conférence.

Au commencement de la conférence le module “*vlc sip*” est demandé par l’interface graphique. VLC va alors rechercher le code du module “*vlc sip*” et lancer la fonction *Open()* .

```
static int Open( vlc_object_t *p_this )
{
     sipagent_t *p_sip = (sipagent_t *)p_this;
     sipagent_sys_t * p_sys;

     int mut_result = vlc_mutex_init(p_sip, & p_sip->init_lock);
     int cond_result = vlc_cond_init(p_sip, & p_sip->init_cond);

     char *psz_url = 0;
     vlc_url_t url;

     psz_url = config_GetPsz( p_sip, "sip-host" );
     vlc_UrlParse( &url, psz_url, 0 );
     if( psz_url ) FREE( psz_url );

     p_sip->p_sys = p_sys = malloc( sizeof( sipagent_sys_t ) );
     if( !p_sys ) goto error;

     p_sys->i_port = url.i_port;
     p_sip->active_connection = 0;

     p_sys->host = malloc(4+strlen(url.psz_host)+1+5);
     sprintf(p_sys->host,"sip:%s:%d",url.psz_host, p_sys->i_port);


     if( vlc_thread_create( p_sip, "sip host thread", Sip_Host_New,
         VLC_THREAD_PRIORITY_HIGHEST, VLC_FALSE ) )
     {
          msg_Err( p_this, "cannot spawn sip host thread" );
          goto error;
     }

     vlc_UrlClean( &url );

    return VLC_SUCCESS;

error:
    if( p_sys ) free( p_sys );
    vlc_UrlClean( &url );
    return VLC_EGENERIC;
}
```

La fonction Open va d’abord initialiser la condition *cond-result* . Cette condition sera libéré ensuite par le module lorsqu‘il sera prêt a fonctionner. Les modules attendant la

libération de la condition (nottament l'interface graphique) sauront ensuite qu'il pourront utiliser le gestionnaire de signalisation.

Ensuite, l'adresse sip de la conférence est reconstruite sous la forme *sip:nom_de_la_conférence@IP_du_concentrateur*. Une fois effectué , un nouveau thread est crée avant de rendre la main. Ce thread lance la fonction *Sip_Host_New()*.

```
static void Sip_Host_New(sipagent_t * sip){
     vlc_thread_ready(sip);

     su_init();
     if(sip->p_sys) {
     sipagent_sys_t *  p_sys = sip->p_sys;

        p_sys->root = su_root_create(NULL);
     p_sys->nua = nua_create(p_sys->root,event_callback ,
                  (nua_magic_t *) sip,
            //on passe p_sip en tant que nua_magic (envirronnement)
             //seul lien entre VLC et SOFIA-SIP (dans le call back)
                  NUTAG_URL(p_sys->host),
                  TAG_END());//fin des tag
     sip->nua=p_sys->nua;
     nua_set_params(p_sys->nua,
             NUTAG_AUTOACK(1),
             NUTAG_UPDATE_REFRESH(1),
             NUTAG_SESSION_TIMER(18000),//30 min
     TAG_END());

     vlc_mutex_lock(& sip->init_lock);
     vlc_cond_signal(& sip->init_cond);
     vlc_mutex_unlock(& sip->init_lock);
          su_root_run(p_sys->root);
     }
};
```

La fonction *Sip_Host_New(...)* prépare la libraire SofiaSIP pour la conférence. Afin de créer une instance de la librairie representé par l'objet *p_sys->nua*, nous appelons la fonction *nua_create(...)*. Cette fonction nécessite au moins 4 paramètres :
_un pointeur sur l'environnement , en l'occurrence *p_sys->root* , qui peut être *NULL*.
_Un pointeur de fonction. Cette fonction, ici *event_callback(...)* , aura pour charge d'effectuer les actions liés à la réception d'un message SIP, ainsi que de traiter les erreurs de transmission.
_un pointeur vers un objet. Cet objet , en l'occurrence la structure de type *sipagent_t* nommé *sip*, sera passé en paramètre de la fonction *event_callback(...)*. La communication entre la fonction *event_callback(...)* et le reste du programme passe par le partage de cette objet.

_un TAG indiquant le nom de l'agent sip sur le réseau. Dans notre cas , ce tag a la valeur "sip:@IP_de_la_machine:5060". Cette valeur permet a la plupart des services basé sur SIP de fonctionner.

On peut à tout moment rajouter des options a l'instance de SofiaSIP par la fonction *nua_set_params()*. Dans notre cas nous avons ajouté le *TAG AUTOACK* qui permet d'acquitter automatiquement les messages reçus. elle permet aussi de gérer de manière automatique la plupart des erreurs provenant du réseau.

Enfin la fonction *Sip_Host_New()* libère la condition *sip->init_cond* , ce qui va avertir l'interface graphique que le module est prêt.

Ainsi lors du déroulement de la conférence seule la fonction *event_callback(...)* est utilisé :

```
void event_callback(nua_event_t event, int status, char const  *phrase ,
          nua_t * nua, nua_magic_t  * magic, nua_handle_t *nh,
          nua_hmagic_t *hmagic , sip_t const *sip,tagi_t tags[]) {
...

  switch(event) {

                   /**********************************/
  case nua_r_invite:
       if(status==403)
             ...;
       if(status==416)
             ...;

       if(status==600)
            ...;
       if(status==200 ) {
         ...
    break;;
                            /**********************************/
   case nua_i_message:
      ...
        break;;
                           /**********************************/
    case nua_i_notify:
     ...
     break;;
                       /**********************************/
     case nua_i_active:
     ...
         break;;
                       /**********************************/
    default:;
```

```
    printf("evenement non trait√©\n");
   };
};
```

Lorsque Sofia SIP reçoit un message SIP, il appelle la fonction *event_callback(...)* avec, pour argument *event* le type du message reçu. l'entier *status* correspond au code de réponse reçu ou attendu du message reçu. Quant au message en lui même , il nous est apporté par l'argument *sip* de type *sip_t **.
Ainsi Pour connaître le type de message il suffit de deux structures conditionnelles, une sur le paramètre *event* puis une sur le paramètre *status*.

Au lancement de la conférence, l'interface graphique demande à sofiaSIP d'envoyer le message. A la réception de la réponse du serveur la fonction *event_callback(...)* est appelé avec le paramètre *event* positionné a *nua_r_invite*. Ainsi le code est éxécuté :

```
       if(status==403)
       sipagent->show_error(sipagent->pointeur,p_vlvc_error_messages
[VLVC_CORE_MODULE_ERROR_USERNAME_INDEX],10);
       if(status==416)
       sipagent->show_error(sipagent->pointeur,p_vlvc_error_messages
[VLVC_CORE_MODULE_ERROR_CONFNAME_INDEX],10);

       if(status==600)
        sipagent->show_error(sipagent->pointeur,p_vlvc_error_messages
[VLVC_CORE_MODULE_ERROR_MAXIMUM_REACHED_INDEX],10);
```

Le status correspond au code de la réponse, le serveur ne pouvant répondre que par les codes 403, 416 ou 600 pour les messages d'erreurs, seul ces cas sont traités. L'erreur est alors remonté vers l'interface graphique grâce a la fonction *show_error(...).*

Ensuite est traité le cas ou le serveur renvoie une réponse positive, c'est a dire une réponse avec le status 200. Le serveur renvoie avec la réponse , les informations nécessaires pour envoyer son flux au serveur sous forme de fichier SDP. La première étape va donc constituer a parser le fichier SDP grâce a la fonction *sdp_parse(...)* . Le but est de récupérer le numéro de port et l'adresse IP sur lequel envoyer le flux vidéo.

```
       if(status==200 )
       {
            parser = sdp_parse(p_sys->root,sip->sip_payload->pl_data, sip-
>sip_payload->pl_len, 0);
            sdp = sdp_session(parser);
```

Une fois, le fichier SDP parsé, l'information qui nous intéresse se trouve dans la structure *sdp* . Il ne reste donc plus qu'a construire le flux grâce au module VLM. Le

paramètrage du module VLM se fait avec la fonction *vlm_ExecuteCommand(...)* . On va d'abord créer un objet *Broadcast* qui prendra en entrée la webcam de l'utilisateur.
La sortie de l'objet *broadcast* est un peu plus compliqué car le flux est transcodé avant d'être expédié sur le réseau (*#transcode{vcodec=DIV3, vb=96, scale=0.75, acodec=MP3, ab=32, channels=2}*) . Le transcodage chez le client permet de ne pas surcharger le concentrateur ni la voie montante du satellite. Il suffit alors de récupérer les informations de la structure *sdp* pour envoyer le flux tout juste transcodé sur le bon port du concentrateur (*sdp->sdp_media->m_port*). Enfin l'ordre *control vlc-%s play* est donné au VLM pour activer l'objet broadcast.

```
	sprintf(psz_vlm_command,"new vlc-%s broadcast enabled",(p_vlvc->p_parameters)->psz_username);
	vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

	msg_Dbg( p_vlvc, "vlm command : %s", psz_vlm_command );
	vlm_MessageDelete( message );

	sprintf(psz_vlm_command,
		"setup vlc-%s input %s",p_vlvc->p_parameters->psz_username,p_vlvc->p_parameters->psz_input_mrl);
	vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );
	msg_Dbg( p_vlvc, "vlm command : %s", psz_vlm_command );
	vlm_MessageDelete( message );

	sprintf(server_mrl,"#transcode{vcodec=%s,vb=%d,scale=0.75,acodec=%s,ab=%d,channels=2}:duplicate{dst=display,dst=std{access=%s,mux=ts,dst=%s:%d}}", p_vlvc->p_parameters->psz_video_codec, p_vlvc->p_parameters->i_video_bitrate, p_vlvc->p_parameters->psz_audio_codec, p_vlvc->p_parameters->i_audio_bitrate, p_vlvc->p_parameters->psz_network_protocol, sdp->sdp_connection->c_address, sdp->sdp_media->m_port);

	sprintf(psz_vlm_command,
		"setup vlc-%s output %s", p_vlvc->p_parameters->psz_username,server_mrl);
	vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );
	msg_Dbg( p_vlvc, "vlm command : %s", psz_vlm_command );
	vlm_MessageDelete( message );
	sprintf(psz_vlm_command,
		"control vlc-%s play", p_vlvc->p_parameters->psz_username);
	vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

	msg_Dbg( p_vlvc, "vlm command : %s", psz_vlm_command );
	vlm_MessageDelete( message );

	}
```

Une fois la session d'appel activé, la fonction *event_callback(...)* est rappelé avec le paramètre *event* positionné a *nua_i_active*. Il faut alors s'inscrire au gestionnaire de

présence du concentrateur afin de connaître l'identité et la location des autres participants à la conférence.

```
case nua_i_active:
    if (sipagent->active_connection!=0)
        return;
    sipagent->active_connection = 1;

    if (nh==NULL)
        nh =p_vlvc->handle;
    nua_subscribe(nh,
            SIPTAG_EVENT_STR("presence"),
            SIPTAG_ACCEPT_STR("application/pidf-info+xml"),
            TAG_END());
 break;;
```

l'événement *nua_i_active* correspond aussi à la réception du message SIP Update. Ce message permet de vérifier la continuité de la session. Il faut donc repérer le premier événement *nua_i_active* d'ou la présence du marqueur *sipagent->active_connection.*

Après la souscription a la conférence, elle peut alors commencer. Durant son déroulement, 2 type de messages peuvent survenir chez l'usager.

Le type SIP MESSAGE permet de recevoir un message texte issu d'un participant . Ce message est forcement transmit par l'intermédiaire du concentrateur.

```
case nua_i_message:
      sipagent->show_message(sipagent->pointeur,(sip->sip_payload)->pl_data,0);
break;;
```

Le module de signalisation va transmettre le contenu du message a l'interface graphique grâce à la fonction *show_message(...)* .

Le type SIP NOTIFY prévient quant a lui d'un changement de status d'un utilisateur. Il contient aussi les informations nécessaire pour capter son flux vidéo.
l'usager est informé quand un nouvel utilisateur s'inscrit à la conférence, ou lorsqu'un autre utilisateur quitte la conférence.

```
case nua_i_notify:

        //parser le sdp qui contient les adresses multicast des flux//

        sdp_message=(sip->sip_payload)->pl_data;
        ...
```

Une fois le contenu du message parsé, son interprétation va consister en un premier temps à informer l'interface graphique avec les fonctions *add_user(..)* ou *del_user(..)* . Ensuite Il faut afficher le flux correspondant à l'écran (créer l'objet *Broadcast*) ou supprimer ce flux de l'écran (supprimer l'objet *Broadcast* correspondant). Afin de ne pas

confondre les objets entre eux, la nomenclature des objets suit la règle "vod-*nom_de_l_utilisateur*" .

```
   /*************  Ajout d'un flux *************/

 if(sdp_state[1]=='N') //cherche ON
 {
     sipagent->add_user(sipagent->pointeur,sdp_host,sdp_add,sdp_port);

 sprintf(psz_vlm_command,"new vod-%s broadcast enabled",sdp_host);
 vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

 sprintf(psz_vlm_command,
     "setup vod-%s input udp://@%s:%d",sdp_host,sdp_add,sdp_port);
 vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

 sprintf(psz_vlm_command,"setup vod-%s output #duplicate{dst=display}",sdp_host);
 vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

  if(strcmp(sdp_host,(p_vlvc->p_parameters)->psz_username)!=0)
   {
    sprintf(psz_vlm_command,"control vod-%s play",sdp_host);
    vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );
   }
  }

  /*************** Départ d'un utilisateur*****************/

    if(sdp_state[1]=='F') //cherche OFF
 {
     sipagent->del_user(sipagent->pointeur,sdp_host);
     sprintf(psz_vlm_command,"control vod-%s stop",sdp_host);
     vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

     sprintf(psz_vlm_command,"del vod-%s",sdp_host);
     vlm_ExecuteCommand( p_vlvc->p_vlm, psz_vlm_command, &message );

 }
 break;;
```

• <u>3.3.4 communication inter modules</u>

En effet , pour transmettre les messages issu du réseau et les ordres de l'utilisateur , des mécanismes de synchronisation sont nécessaire entre le module de signalisation et l'interface graphique.

Les deux modules ayant une exécution concurrentielle, la première chose à faire est de démarrer les modules en même temps. Ainsi comme le constructeur de l'interface graphique est lancé en premier, il va appeler le module *sip client*. Ensuite le déroulement du programme s'effectue comme le montre la figure 3.9.

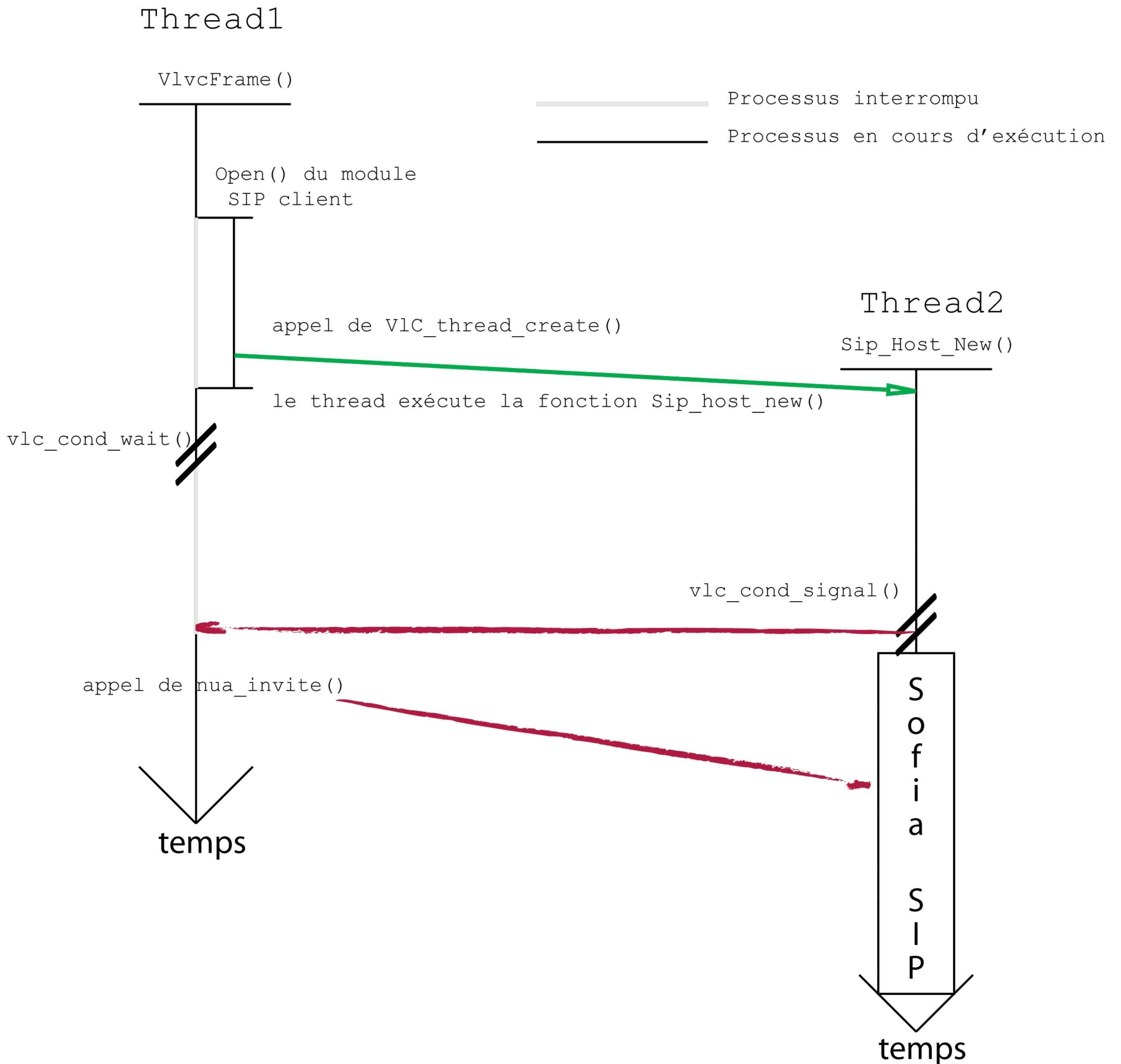

figure 3.9 : diagramme temporel au lancement des modules et synchronisation

Lors du déroulement de la conférence, le module de signalisation remonte les informations par les pointeurs de fonctions *sipagent->add_user(...)* , *sipagent->del_user()* , *sipagent->show_message(...)* , *sipagent->show_error(...)*.

Ces pointeurs sont initialisé par la fonction *VlvcFrame()* une fois la synchronisation entre les 2 modules effectué.

```
sipagent_t* sip_agent=(sipagent_t*)vlc_object_find
(p_vlvc,VLC_OBJECT_SIP,FIND_ANYWHERE);

   //wait for the initialisation of NUA
    vlc_mutex_lock(& sip_agent->init_lock);
    vlc_cond_wait( & sip_agent->init_cond, & sip_agent->init_lock);
    vlc_mutex_unlock(& sip_agent->init_lock);

   ...
    sip_agent->add_user = &AddUser;
    sip_agent->del_user = &DelUser;
    sip_agent->show_message = &IncomingChatMessage;
    sip_agent->show_error = &ShowMessage;
    sip_agent->pointeur = this;
```

Ces fonctions ont un usage particulier puisque leurs seul but est d'empaqueter les données dans un événement wxWidgets. Cet événement est ensuite lancé par l'objet *wxPostEvent(...)* .

```
void AddUser( VlvcFrame *pointeur, char *user_name, char* group, int port )
{
     pointeur->new_client.user_name = strdup(user_name);
     pointeur->new_client.group = strdup(group);
     pointeur->new_client.port = port;

     wxCommandEvent evt( wxEVT_COMMAND_ENTER, VLVC_ADD_USER_EVENT );
     evt.SetClientData( &pointeur->new_client );
     wxPostEvent( pointeur, evt );
}
```

Dans la fonction *AddUser(...)*, les données empaquetés sont le nom d'utilisateur, l'adresse multicast et le numéro de port du flux de cet utilisateur. l'événement wxWidgets prend le type *VLVC_ADD_USER_EVENT*.

```
void DelUser( VlvcFrame *pointeur, char *user_name )
{
    wxCommandEvent evt( wxEVT_COMMAND_ENTER, VLVC_DEL_USER_EVENT );
    evt.SetClientData( (void *)user_name );
    wxPostEvent( pointeur, evt );
```

```
}
```

Dans la fonction *DelUser(...)*, seul le nom d'utilisateur est empaqueté dans l'événement. l'événement wxWidgets prend le type *VLVC_DEL_USER_EVENT*.

```
void    ShowMessage( VlvcFrame *pointeur,char *error, int i_critical )
{
     vlc_mutex_lock( &pointeur->messageLock );
     while( pointeur->b_messageLocked == VLC_TRUE )
          msleep( 10000 );
     pointeur->b_messageLocked = VLC_TRUE;
     vlc_mutex_unlock( &pointeur->messageLock );

     wxCommandEvent evt( wxEVT_COMMAND_ENTER, VLVC_SHOW_MESSAGE_EVENT );
     pointeur->messageData.i_critical = i_critical;
     strcpy( pointeur->messageData.psz_message, error );
     evt.SetClientData( &pointeur->messageData );
     wxPostEvent( pointeur, evt );
}
```

Dans la fonction *ShowMessage(...)*, qui est chargé de transmettre les messages d'erreurs, l'événement *VLVC_SHOW_MESSAGE_EVENT* contient le texte décrivant l'erreur.

Enfin il reste la fonction *IncomingChatMessage(...)* chargé de retransmettre les messages texte que peuvent s'envoyer les utilisateurs. Elle va lancer l'événement *VLVC_CHAT_MESSAGE_EVENT* contenant le texte du message.

```
void IncomingChatMessage( VlvcFrame *pointeur, const char *psz_message,
                           int i_message_type )
{
    // TODO :implement message type
    vlc_mutex_lock( &pointeur->chatMessageLock );
    while ( pointeur->b_chatMessageLocked == VLC_TRUE )
        msleep( 10000 );
    pointeur->b_chatMessageLocked = VLC_TRUE;
    vlc_mutex_unlock( &pointeur->chatMessageLock );
    wxCommandEvent evt( wxEVT_COMMAND_ENTER, VLVC_CHAT_MESSAGE_EVENT );

    memset( pointeur->chatMessageData.psz_message, STRING_TERMINATOR,
        sizeof( pointeur->chatMessageData.psz_message ) );
    memcpy( pointeur->chatMessageData.psz_message, psz_message,
        strlen( psz_message ) * sizeof( *psz_message ) );
    pointeur->chatMessageData.i_type = i_message_type;

    evt.SetClientData( &pointeur->chatMessageData );
    wxPostEvent( pointeur, evt );
}
```

La bibliothèque wxWidget gère les événements au travers d’une table spécifique à chaque logiciel. cette table définit les types d'événements utilisé pour le logiciel et précise à chaque type la fonction capable de les traiter.

```
BEGIN_EVENT_TABLE( VlvcFrame, wxFrame )
...
    EVT_COMMAND_ENTER( VLVC_SHOW_MESSAGE_EVENT, VlvcFrame::OnMessage )
    EVT_COMMAND_ENTER( VLVC_ADD_USER_EVENT, VlvcFrame::OnAddUser )
    EVT_COMMAND_ENTER( VLVC_DEL_USER_EVENT, VlvcFrame::OnDelUser )
    EVT_COMMAND_ENTER( VLVC_CHAT_MESSAGE_EVENT, VlvcFrame::OnChatMessage )

END_EVENT_TABLE()
```

l’avantage de ce système est qu’il permet de facilement transiter les informations d’un thread à un autre sans perturber leur déroulement.

Les ordres issus de l’interface graphique pour le module de signalisation (envoyer un message ou quitter la conférence) ou pour VLM (afficher ou non un flux a l’écran) se font par un classique appel de fonction.

3.4. Implémentation coté concentrateur

• 3.4.1 Architecture général

L'implémentation coté concentrateur est basé sur le logiciel VLC fonctionnant en mode console (au lancement : *vlc -I telnet*). Afin que VLC puisse implémenter la fonction de concentrateur pour vidéoconférence, nous avons développé un module, nommé "*SIP focus*", respectant la RFC 4353. La librairie sofia SIP a été utilisé de manière similaire à l'implémentation coté client.

Le module nouvellement créé a pour rôle de gérer la signalisation de plusieurs conférences, de gérer la liste des ports disponible pour la réception des flux vidéos des clients, et d'attribuer des adresses multicast ainsi que les numéros de ports pour la redistribution des flux.
C'est le module VLM qui a la charge de redistribuer les flux vidéos issu des clients en fonction des instructions fourni par le module *sip focus*.

Afin de faciliter la découverte du service de vidéoconférence, le module *sip focus* fournit un mini-serveur web. Ce serveur n'affiche qu'une page réactualisée en temps réél contenant la liste des conférence et des clients par conférence. L'ensemble de l'implémentation est montré par le schéma figure 3.10.

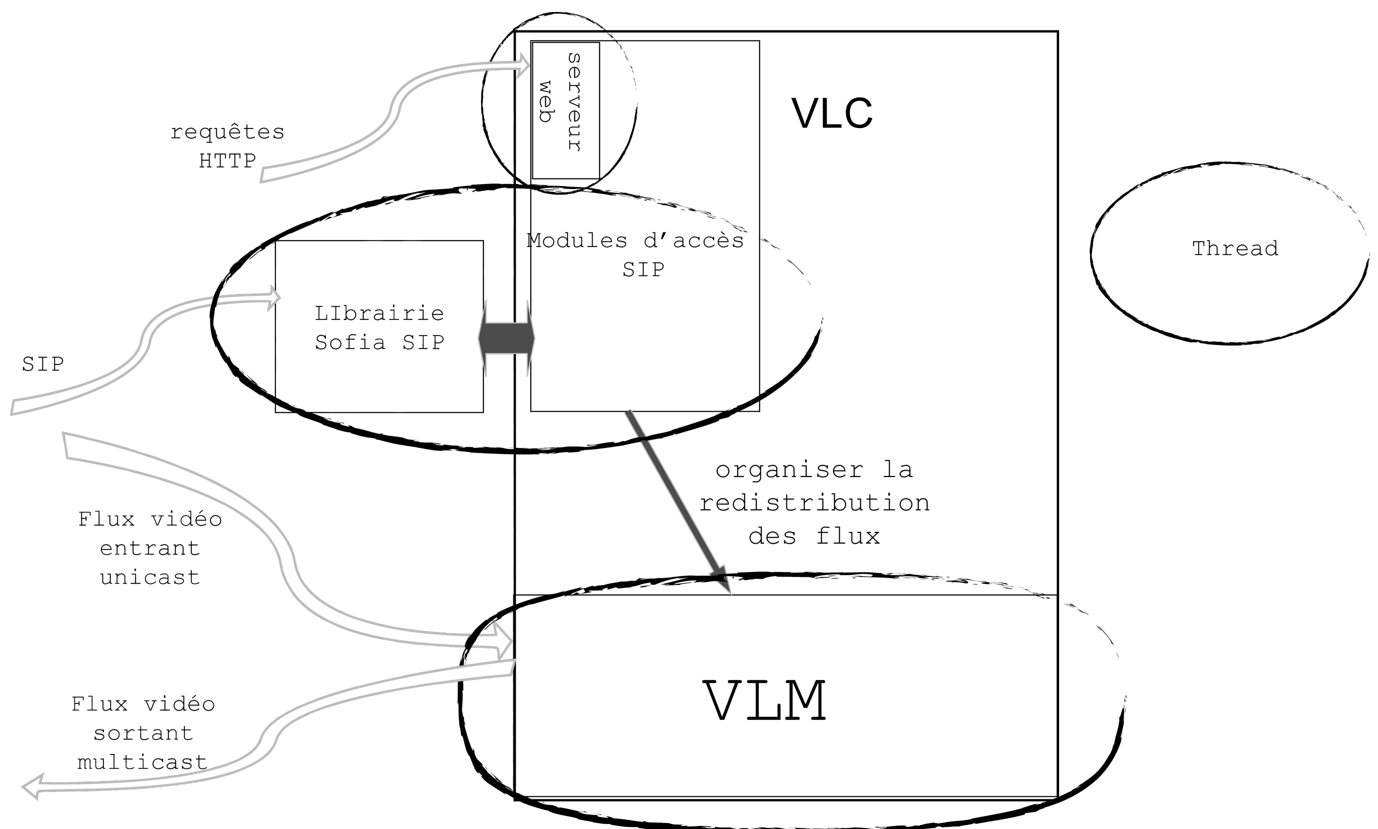

figure 3.10 : schéma global coté concentrateur

• 3.4.2 Le module sip focus

Le module *sip focus* possède son code dans le fichier “*modules/misc/sip_focus.c*” . Il a été créé pour gérer plusieurs conférence à la fois. Pour cela une structure de donnée particulière a été mis en place. Il s'agit d'une double liste pouvant stocker les informations relatives aux conférences et aux clients de manière ordonné, comme le montre la figure 3.11.

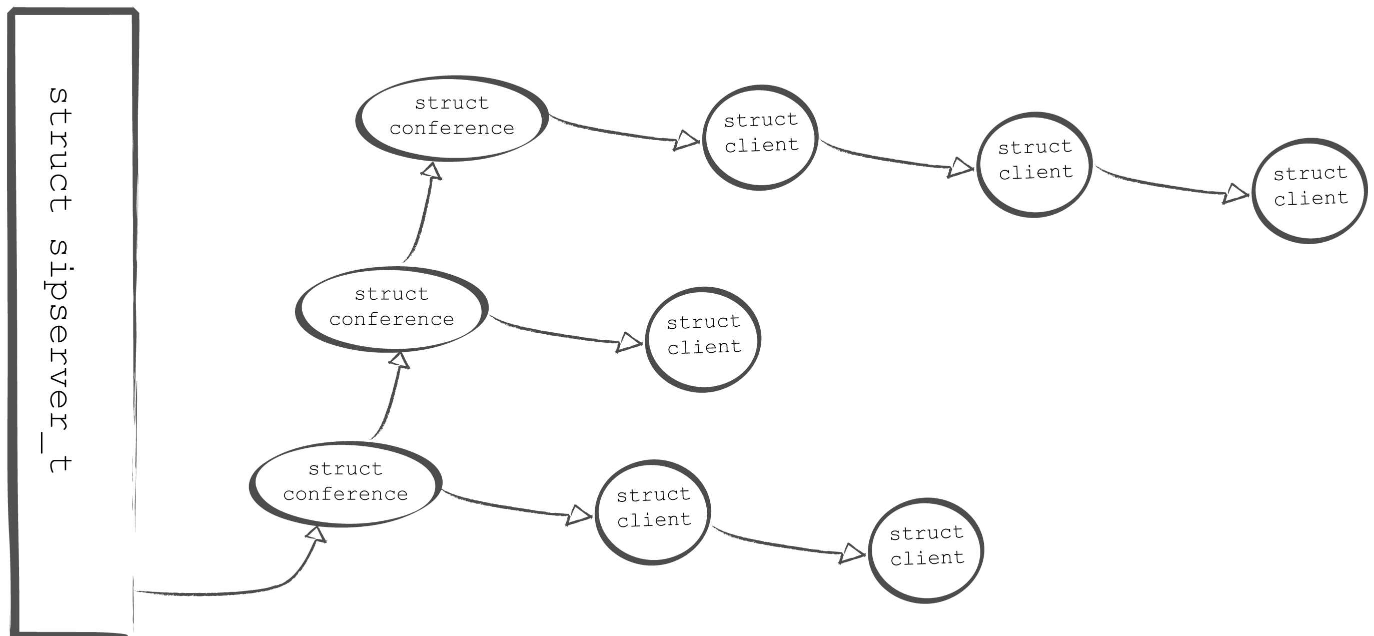

figure 3.11 : structure de donné du module sip focus

Le module *sip focus* stocke ses donnés dans la structure *sipserver_t*. Afin d'avoir une grande souplesse d'utilisation, les donnés liés aux conférences sont stockées sous forme de liste, chaque élément de la liste correspondant à une conférence. comme on peut le voir ci-dessous, une conférence se résume à un nom, une adresse multicast, une plage d'adresse (*int * pool*) , une liste de clients ainsi que certains compteurs.

```
struct conference {
    struct conference * next;
    char * name;
    char * group;
    int plage_min;
    int plage_max;
    int nb_clients;
    struct client * clients;
    int * pool;
};
```

Pour chaque conférence, les clients sont stocké sous forme de liste dont chaque élément représente un client. Un client est composé d'un nom, de 2 références vers sa

session géré par sofia SIP (nommées *Handle*) et d'un port par lequel le flux vidéo arrive au concentrateur.

```
struct client {
     struct client * next;
     char * name;
     void * nh;
     void * nh_notify;
     int incoming_port;
};
```

Afin de paramétrer facilement les modules , VLC propose un système de paramètres modifiable durant l'exécution. Le module *sip focus* utilise ce système pour 5 paramètres. Ces paramètres sont déclarés en fonction de leur type par la macro *add_string (...)* , *add_integer(...)* , *add_integer_with_range(...)* . Le 2eme argument de ces macros correspond à la valeur par défaut si l'administrateur ne précise pas ces paramètres a l'exécution. Nos 5 paramètres sont :

_*sip-host* qui précise la socket d'écoute pour Sofia SIP.

_ *sip-multicast* qui précise l'adresse de groupe des flux issus du concentrateur.

_ *sip-port-by-conf* qui précise la taille de la plage de port attribué par conférence.

_ *sip-max-confs* qui précise le nombre maximum de conférences en simultané.

_*sip-first-conf-port* qui précise la valeur minimal du numéro de port utilisé. En effet les numéros de port faible sont souvent attribué à d'autres services qui sont indispensable pour le bon fonctionnement de l'appareil dans le réseau.

```
add_string ( "sip-host", "0.0.0.0:5060", NULL, HOST_TEXT, HOST_LONGTEXT, VLC_TRUE );
add_string ( "sip-multicast", "224.0.0.8", NULL, HOST_TEXT, HOST_LONGTEXT,
VLC_TRUE );
add_integer( "sip-port-by-conf", 500, NULL, SIP_PORT_BY_CONF_TEXT,
SIP_PORT_BY_CONF_LONGTEXT, VLC_TRUE );
add_integer( "sip-max-confs", 10, NULL, SIP_MAX_CONF_TEXT, SIP_MAX_CONF_LONGTEXT,
VLC_TRUE );
add_integer_with_range( "sip-first-conf-port", 5000, 1024, 65535, NULL,
SIP_FIRST_TEXT, SIP_FIRST_LONGTEXT, VLC_TRUE );
```

Comme tous module de VLC, il possède une fonction *open(...)* qui sera appelé en premier. Le module étant basé sur la librairie SOFIA SIP, il est construit de manière similaire au module *vlc sip* utilisé pour l'implémentation du client.

```
static int Open( vlc_object_t *p_this )
{
    sipserver_t *p_sip = (sipserver_t *)p_this;
    sipserver_sys_t * p_sys;
    char *psz_url = NULL;
    p_sip->p_sys = p_sys = malloc( sizeof( sipserver_sys_t ) );
    if( !p_sys ) goto error;
```

```
vlc_url_t url;

//recuperation de l'adresse bind pour la signalistaion
psz_url = config_GetPsz( p_sip, "sip-host" );
vlc_UrlParse( &url, psz_url, 0 );
if( psz_url ) FREE( psz_url );

p_sys->i_port = url.i_port;

p_sys->host = malloc(4+strlen(url.psz_host)+1+5);
sprintf(p_sys->host,"sip:%s:%d",url.psz_host, p_sys->i_port);
vlc_UrlClean( &url );

//recuperation de l'adresse multicast
psz_url = config_GetPsz( p_sip, "sip-multicast" );
vlc_UrlParse( &url, psz_url, 0 );
if( psz_url ) FREE( psz_url );

p_sip->group_min = strdup(url.psz_host);

//recuperation des flux
p_sip->max_port_by_conf = config_GetInt( p_sip, "sip-port-by-conf" );
p_sip->first_conf_port = config_GetInt( p_sip, "sip-first-conf-port" );
p_sip->max_confs = config_GetInt( p_sip, "sip-max-confs" );
```

On récupère les variables paramètrables pour les stocker dans l'instance *p_sip* de la structure *sipserver_t*.

```
    if( vlc_thread_create( p_sip, "sip host thread", Sip_Host_New,
    VLC_THREAD_PRIORITY_LOW, VLC_FALSE ) )
    {
        msg_Err( p_this, "cannot spawn sip host thread" );
        goto error;
    }

    if( vlc_thread_create( p_sip, "http host thread", HTTPD_Host_New,
    VLC_THREAD_PRIORITY_LOW, VLC_FALSE ) )
    {
        msg_Dbg( p_this, "cannot spawn http host thread for discover conference
system" );
    }
    vlc_UrlClean( &url );
```

Ensuite, il faut créer 2 threads avec la fonction *vlc_thread_create(...)* ; un pour la signalisation SIP avec l'appel à la fonction *Sip_Host_New(...)* et l'autre pour le mini-serveur web avec l'appel à la fonction *HTTPD_Host_New(..)* .

Comme pour l’implémentation du client, la fonction *Sip_Host_New()* prépare la librairie Sofia-SIP et notamment avec la fonction *event_callback(...)* qui se charge de traiter les événements lorsque des messages SIP sont reçus. Par contre le traitement est très différent de celui de l'implémentation coté client.

```
void event_callback(nua_event_t event, int status, char const  *phrase ,nua_t * nua,
            nua_magic_t  * magic, nua_handle_t *nh, nua_hmagic_t *hmagic ,
            sip_t const *sip,tagi_t tags[]) {


   sipserver_t * sipserver = (sipserver_t *)magic;
   struct conference * conf=sipserver->confs;
   struct client  *cl = NULL;
...
 switch (event) {
    case nua_i_invite:
    case nua_i_subscribe:
    case nua_i_bye:
    case nua_i_message:
     if(!sip || !sip->sip_to ||!sip->sip_to->a_url || !sip->sip_to->a_url-
>url_user ) return;
     conf = conf_search(sip->sip_to->a_url->url_user,sipserver->confs);
     printf("conf_determin√©\n");

 /*determination du client */

    if( conf!=NULL) {
        if(!sip || !sip->sip_from ||!sip->sip_from->a_url || !sip->sip_from->a_url-
>url_user ) return;
        cl = client_search(sip->sip_from->a_url->url_user,conf->clients);
    }else {
       cl=NULL;
    };
    printf("client_determin√©\n");
    break;
```

La 1ere étape consiste a déterminer pour quelle conférence et par quel client ce message est adressé. Pour cela nous comparons la conférence demandé par le message au conférence déjà présente grâce a la fonction *conf_search(...)* .

```
 struct conference * conf_search(const char * name, struct conference * liste_conf)
{
    if (liste_conf == NULL) return NULL;
    if ( ! strcmp(liste_conf->name, name)) return liste_conf;
    return conf_search(name,liste_conf->next);
};
```

La fonction *conf_search(...)* est récursive et renvoie le pointeur *NULL* si elle ne trouve pas la conférence demandé, sinon elle renvoie le pointeur vers la conférence.

Ensuite il reste à déterminer quel client de la conférence effectue la requête. Pour cela nous utilisons la fonction *client_search(...)* .

```
struct client * client_search(const char * name, struct client * liste_cl) {
    if (liste_cl == NULL) return NULL;
    if ( ! strcmp(liste_cl->name, name)) return liste_cl;
    return client_search(name,liste_cl->next);
};
```

Cette fonction est aussi récursive et renvoie NULL si elle ne trouve pas le nom demandé dans la liste. Si elle trouve , elle renvoie un pointeur vers l'objet client qui nous intéresse.

Il reste maintenant à effectuer le traitement en fonction du type de message reçu. Tout d'abord l'utilisateur commencera par s'inviter a la conférence.

```
case nua_i_invite:
    if ( status ==200) return; //cas du UPDATE
       if (nh ==NULL)
        nh = nua_handle(nua,magic, TAG_END());

/*politique : on n'accepte que sipserver->max_confs conférence à la fois
 * si il y a des conférence disponible,
 * l'utilisateur crée une conf par un simple INVITE
 *
 * Une conf est detruite quand il n'y a plus d'utilisateur*/
    if (conf==NULL && nb_conf>=sipserver->max_confs) {
        nua_respond(nh,416,"Unsupported URI scheme",
            TAG_END());
        return;
        };
    if (conf==NULL && nb_conf<sipserver->max_confs) {
    conf = conf_new(sip->sip_to->a_url->url_user, sipserver);
    };
/*verification des doublons */
    if(cl !=NULL) {
        nua_respond(nh,403,"Not allowed : already connected",
            TAG_END());
        return;
    };
    cl=client_new(sip->sip_from->a_url->url_user, nh,conf,p_vlm);
...
```

On commence par traiter le cas ou l'utilisateur appelle une conférence qui n'existe pas encore. Si il y a pas trop de conférence au moment de la demande (plus de *sipserver->max_confs*) alors une nouvelle conférence est crée avec la fonction *conf_new(...)* . Sinon le focus répond par une réponse négative avec le code réponse 416.

On vérifie ensuite si la recherche du dit client a été fructueuse (*if cl!=NULL*) . Si oui ,

cela signifie que le pseudonyme est déjà utilisé dans la conférence , la connexion au service est donc refusé. Le serveur répond négativement avec le code 403.

Les principaux cas d'erreurs sont désormais traités, il font donc créer le client grâce a la fonction *client_new(...)* . Elle prend en argument le pseudonyme sous forme d'un char *, la reference de la session fourni par SOFIA SIP, la conférence demandé ainsi qu'un pointeur vers l'objet VLM.

```
 struct client * client_new(const char * name, nua_handle_t * nh, struct conference
* conf, vlm_t * p_vlm) {
     //creation de l'objet client
         if(conf->nb_clients >=conf->plage_max - conf->plage_min)
           return NULL;
     struct client * cl;
     MALLOC(cl , sizeof(struct client));
     struct client * clients=conf->clients;
     cl->name= strdup(name);
     cl->nh=nh;

     int i =0;
     while(conf->pool[i]!=0) i++;
     cl->incoming_port = conf->plage_min + i;
     conf->pool[i] = 1;

     cl->next=NULL;
     conf->nb_clients++;
     //ajoute a la fin de la liste
     if (conf->clients == NULL) {
          conf->clients = cl;
     }else {
          while(clients->next) clients = clients->next;
          clients->next  = cl;
     };

     //creation du broadcast associé
#define NOM_SIZE 200
#define CMD_SIZE 400
     vlm_message_t *p_message = NULL;
     char  nom[NOM_SIZE]; char cmd[CMD_SIZE];
     /*redirection du flux unicast vers le multicast*/
     snprintf( nom,NOM_SIZE, "%s%s" , conf->name, cl->name);
     snprintf( cmd,CMD_SIZE, "new %s broadcast enabled" , nom);
     vlm_ExecuteCommand( p_vlm, cmd , &p_message );

     snprintf( cmd,CMD_SIZE, "setup %s input udp://@:%d" , nom , cl->incoming_port);
     vlm_ExecuteCommand( p_vlm, cmd , &p_message );

     snprintf( cmd,CMD_SIZE, "setup %s output #standard{access=rtp,mux=ts,dst=%s:%
d}", nom, conf->group, cl->incoming_port );
```

```
    vlm_ExecuteCommand( p_vlm, cmd , &p_message );

    snprintf( cmd,CMD_SIZE, "control %s play" , nom );
    vlm_ExecuteCommand( p_vlm, cmd , &p_message );
    msleep(1000);//laisser le temps a vlm de faire sa tache


return cl;
};
```

la construction du nouveau client ne se fait que si il y a pas trop de client , sinon la première instruction conditionnelle renvoie le pointeur NULL sans avoir créer l'objet client.
Ensuite il faut sélectionner le numéro de port que va utiliser le client. Afin d'utiliser au mieux les ressources disponible, l'ensemble des ports réservés par la conférence est représenté sous forme d'un tableau (*conf->pool*) . le premier élément du tableau correspond au port au plus petit numéro et le dernier élement du tableau au port au plus grand numéro. si le port est disponible , alors la case correspondante du tableau aura le marqueur 0 ; sinon ce sera le marqueur 1. Les numéros de port en entré et en sortie sont identique afin d'en faciliter la gestion.
le client nouvellement crée est accroché en fin de liste. cette fin de liste est trouvé de manière itérative en parcourant toute la liste.
Il ne reste plus qu'a ordonner au module VLM la redirection du flux. Comme pour l'implémentation client, elle se fait par appel de la fonction *vlm_ExecuteCommand(...)* .

```
   if(cl==NULL) {
       nua_respond(nh,600,"Busy everywhere",
           SIPTAG_CONTACT(sip->sip_contact),
           TAG_END());
       return;
   };
      parser = sdp_parse(p_sys->root,sip->sip_payload->pl_data, sip->sip_payload-
>pl_len, 0);
      sdp = sdp_session(parser);


           /********************* Negociation **************************/

       (sdp->sdp_media)->m_port=cl->incoming_port; // numero de port
       (sdp->sdp_connection)->c_address=HOST_NAME;      // addresse du groupe
       (sdp->sdp_media)->m_type_name="audio";           // type du flux
       (sdp->sdp_media)->m_proto_name="UDP";
   nua_respond(nh,200,"OK",
           SOATAG_USER_SDP(sdp),
           TAG_END());
   break;;
```

Une fois le client crée, la fonction *event_callback(...)* n'a plus qu'a renvoyer une réponse positive (code de retour 200) auquel sera adjoint un fichier au format SDP contenant le numéros de port que devra utiliser le client.

La première phase de l'inscription au service de conférence est alors terminé. Le client va alors souscrire au gestionnaire de présence afin de connaître les autres participants. Le client va donc envoyer un message SIP de type SUSCRIBE. La librairie SOFIA SIP va en conséquence appeler la fonction *event_callback(...)* avec le paramètre *event* positionné à *nua_i_suscribe* .

```
case nua_i_subscribe:
     if (status == 200) return; //ne pas prendre en compte l'update du suscribe
      if (nh ==NULL)
       nh = nua_handle(nua,magic, TAG_END());
      if(conf == NULL || cl==NULL) {
     nua_respond(nh,404,"Not Found",
              NUTAG_WITH_THIS(p_sys->nua),TAG_END());
      return;
    };

      cl->nh_notify = nh;
//repond que tout va bien
    nua_respond(nh,200,"OK", NUTAG_WITH_THIS(p_sys->nua),TAG_END());
     send_notify_all_to_one(cl, "ON",conf,sipserver);
     send_notify_one_to_all(cl, "ON",conf,sipserver);
        break;;
```

SI la conférence et/ou le client ne sont pas retrouvé alors la réponse à la souscription est négative et donc le focus renvoie un message avec un code de retour 404.
Dans le cas normal, le concentrateur acquitte la souscription par un message avec un code de retour 200. Il prévient alors le nouveau venu de tous les utilisateurs présents lors de la conférence avec la fonction *send_notify_all_to_one(...)* . Puis les utilisateurs connaissent l'identité du nouveau venu grâce à la fonction *send_notify_one_to_all(...)* .

```
void send_notify_one_to_all(struct client * cl, char * status,struct conference *
conf,sipserver_t * sipserver) {
    const char * info_base = "h=%s\ns=%s\ng=%s\np=%d\n";//creation des info pour le
nouveau
    int taille= strlen(info_base) + strlen(cl->name) + strlen(status) +strlen(conf-
>group) +5;
    char * info_new;
    MALLOC(info_new, taille);
    sprintf(info_new, info_base,
         cl->name, status,
         conf->group, cl->incoming_port+sipserver->max_port_by_conf/2);

      struct client *  client = conf->clients;
        while(client != NULL){
```

```
          if (cl!=client)//nottament pour le cas OFF
               nua_notify(client->nh_notify,
                    SIPTAG_PAYLOAD_STR(info_new),
                    SIPTAG_CONTENT_TYPE_STR("text/plain"),
                    TAG_END());
          client = client->next;
         };
    FREE(info_new);
};
```

Cette fonction a pour but d'envoyer un message SIP NOTIFY a chaque participant de la conférence concernant le client *cl*. Ainsi chaque participant connaîtra le nom du client , son status (il entre ou il quitte la conférence) ainsi que l'adresse multicast et le numéro de port nécessaire a lecture de son flux vidéo. Ces informations sont inscrit dans le corp du message sous la forme suivante :

***h=**pseudonyme*

***s=**ON/OFF*

***g=**adresse_IP*

***p=**n°_de_port*

ensuite la liste des clients est parcouru pour envoyer ce message à chacun. le message est envoyé grâce à la commande *nua_notify(...)* .

Lorsqu'un utilisateur quitte la conférence , il envoie un message SIP BYE au focus. ce message est capté par la libraire SOFIA SIP au niveau du concentrateur qui va appeler la fonction *event_callback(...)* avec le paramètre *event* positioné à *nua_i_bye* .

```
case nua_i_bye:
    if(conf == NULL || cl==NULL) return;
//supprimer le flux multicast
    MALLOC(cmd , strlen(conf->name) + strlen(cl->name) + strlen("del %s ")+5);
    sprintf( cmd, "del %s%s" , conf->name, cl->name );
    vlm_ExecuteCommand( p_vlm, cmd , &p_message );
    msleep(1000);
    FREE(cmd);

    send_notify_one_to_all(cl, "OFF",conf,sipserver);
    client_del(cl,conf);

    if (conf->nb_clients== 0 )conf_del(conf,sipserver);
        break;;
```

Une fois le client retrouvé dans le liste, il faut supprimer l'objet correspondant à la redistribution du flux. L'ordre “del nomdeconferencpseudonyme” est adressé au module VLM via l'appel de fonction *vlmExecuteCommand(...)* . La fonction *send_notify_one_to_all(...)* prévient les autres participants du départ de l'utilisateur. Il ne reste plus qu'a supprimer le client de la liste des participants à la conférence. La fonction *client_del(...)* est là pour ca.

```
void client_del(struct client * client, struct conference * conf) {
//libération de sa place dans le pool
    conf->pool[client->incoming_port - conf->plage_min] = 0;
    struct client* cl=conf->clients;
     if (client == conf->clients) { //on supprime le premier
    conf->clients = client->next;
    }else {
    int i =0;
    while (cl != NULL && cl->next != client)  {
        cl = cl->next;
        i++;};
    if (cl == NULL) return;

    cl->next = client->next;
    };
    FREE(client->name);
    FREE(client);
conf->nb_clients --;
};
```

Durant le déroulement de la conférence , seul des messages SIP MESSAGE ont un intérêt pour le concentrateur. En effet pour la messagerie texte , le concentrateur a pour rôle de redistribuer le message issu d'un utilisateur a tous les participants de la conférence. Pendant la redistribution, le nom de l'auteur du message est rajuté au message.

```
case nua_i_message:
    if(conf == NULL || cl==NULL) return;
    MALLOC(info ,strlen(cl->name) + sip->sip_payload->pl_len +5);
    sprintf(info,"%s : %s",cl->name,sip->sip_payload->pl_data);
    cl2 = conf->clients;
    while(cl2 !=NULL) {
        nua_message(cl2->nh,
             SIPTAG_CONTENT_TYPE_STR("text/plain"),
             SIPTAG_PAYLOAD_STR(info),
             TAG_END());
       cl2 = cl2->next;
    };
    FREE(info);
    break;;
```

Une fois le nom de l'auteur rajouté, le message est réexpédié a tous les utilisateur de la conférence y compris l'auteur du message.

La conférence peut maintenant se dérouler normalement. Afin de repérer quelque anomalies , notamment quand un client a quitté la conférence sans prévenir, par exemple lors d'une coupure brutale du réseau, un traitement est effectué quand la librairie SOFIA SIP détecte un retour anormal sur le message NOTIFY.

```
case nua_r_notify:
    if (status !=503) return;//le client est parti sans prevenir
    send_notify_one_to_all(cl, "OFF",conf,sipserver);
    client_del( cl, conf);
    break;;
```

En fait, le traitement se résume à une suppression du client incriminé.

Afin de faciliter l’accès a la conférence un système de mini-serveur web a été mis en place. Comme nous l’avons a l’ouverture du module la fonction *HTTPD_Host_new()* est lancé dans un thread séparé afin de fournir ce service.

```
static void HTTPD_Host_New(sipserver_t * sip){
  vlc_thread_ready(sip);

        struct sockaddr_in soc_in;
    int val;
    int ss;//socket server
    int cs;//socket client
    struct sockaddr_in caddr;
    socklen_t caddr_size;
    char tampon[TAMPON_MAX];

    ss =  socket (AF_INET, SOCK_STREAM,0);
    if ( ss ==-1)
         return;

    /* Force la reutilisation de l'adresse si non allouee */
    val = 1;
    if (setsockopt(ss,SOL_SOCKET,SO_REUSEADDR,(char*)&val,sizeof(val)) == -1)
         return;
   /*
    * Nomme localement le socket :
    * socket inet, port local PORT, adresse IP locale quelconque
   */
    soc_in.sin_addr.s_addr = htonl(INADDR_ANY);
    soc_in.sin_family = AF_INET;
    soc_in.sin_port = htons(HTTPD_PORT); //port HHTP
    if(bind(ss, (struct sockaddr *) &(soc_in),sizeof(soc_in)) < 0)
         pthread_exit(NULL);

    if(listen(ss, 1) < 0) /* max 1 clients en attente */
         return;

    while (1) {
       cs = accept(ss,(struct sockaddr *) &(caddr),&(caddr_size));
       if(cs <0)
         continue;//on passe le traitement

       if( read(cs,tampon,TAMPON_MAX) <4) {
```

```
          close(cs);
          continue;
        };

//printf("recu une requete  : %s , reponse a construire\n",tampon);
        if(!strncmp(tampon,"GET / ",6)) {

          char *tmp =conflist_rec(sip->confs);
          char result[10300];
          sprintf(result,"HTTP/1.0 200 OK\r\n\r\n"
              "<HTML><HEAD><style></style><TITLE>Liste des conf&eacute;rences en
cours</TITLE></HEAD><BODY><H1>Liste des conferences :</H1> <BR>%s</BODY></
HTML>",tmp);
          FREE(tmp);
          send(cs,result,strlen(result),0);
          close(cs);
        }else {
          close(cs);
        };
     };
 };
```

La fonction *HTTPD_Host_New()* prépare une socket TCP écoutant sur le port 80. Le serveur répond à l'url[13] http ://*adresse_du_concentrateur/* . Il va alors appeler la fonction *conflist_rec(...)* . Cette fonction va construire de manière itérative la liste des conférences en cours de déroulement. Elle va ajouter a chaque conférence la liste des clients qui y participe. On peut voir le résultat figure 3.12.

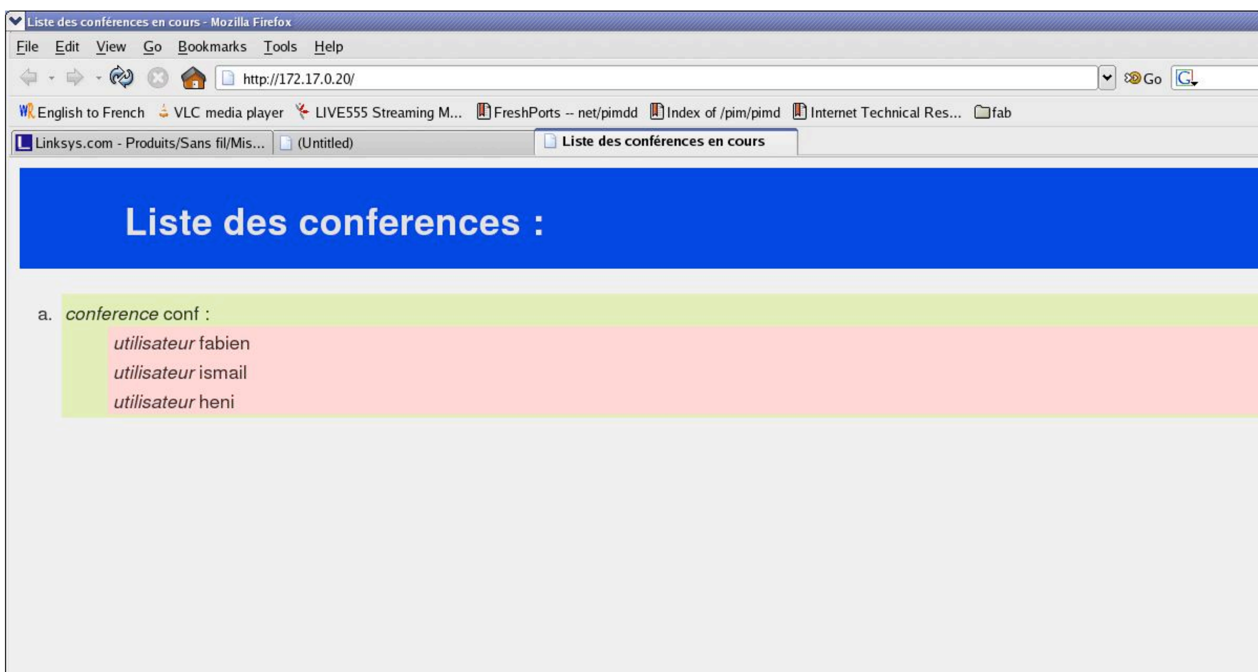

figure 3.12 : page web de la découverte de service

• <u>3.4.3 intégration du module dans VLC</u>

Lors du démarrage de VLC en mode console, le module VLM prend une importance toute particulière puisqu'il devient l'unique moyen de paramètrer l'application. On est donc sur que le module VLM est démarré lorsque VLC est actif.
Ainsi est venu l'idée d'appeler le module *sip focus* lors de la création de VLM , précisément dans la fonction *__vlm_New(...)* .

```
vlm_t *__vlm_New ( vlc_object_t *p_this )
{
  ...

    p_vlm->sip = vlc_object_create( p_vlm, VLC_OBJECT_SIP );
        vlc_object_attach( p_vlm->sip, p_vlm );
        p_vlm->sip->p_module = module_Need( p_vlm->sip, "sip focus", 0, 0 );
        if( !p_vlm->sip->p_module )
        {
            msg_Err( p_vlm, "cannot find sip focus" );
            vlc_object_detach( p_vlm->sip );
            vlc_object_destroy( p_vlm->sip );
            p_vlm->sip = 0;
            return NULL;
        }


   return p_vlm;
}
```

Le module *sip focus* va être appelé par la fonction *vlc_object_create(...)* . La fonction *Open (...)* du module va être appelé par la fonction *module_Need(...)*. C'est à ce moment là que les threads seront créés et que le module commencera à fonctionner.

3.5. Test

Les tests se sont déroulés sous Linux, avec des PC installés sous la distribution Fedora Core 3[14] , Fedora Core 4 ou Mandriva 2007[15]. Il faut donc préparer ces distributions afin qu'elle puisse compiler les services de vidéoconférence. En effet le liste de dépendances pour compiler VLC est très longue.
La compilation standard de VLC se fait avec le trio habituel des commandes de compilation : *./configure* , *make* et *make install* . Afin que la compilation se passe se la même manière avec l'implémentation du service de vidéoconférence, le fichier *configure.ac* a été modifié pour que les dépendances des modules *sip focus* et *sip client* soient intégré comme on peut le voir figure 3.13.

<table>
<tr><td>client</td><td><pre>
 VLC_ADD_PLUGINS([sipclient])
    VLC_ADD_LDFLAGS([sipclient],[`pkg-config sofia-sip-ua --libs`])
    VLC_ADD_CFLAGS([sipclient],[`pkg-config sofia-sip-ua --cflags`])
...

 VLC_ADD_LDFLAGS([wxwidgets],[`${WX_CONFIG} --libs` `pkg-config sofia-sip-ua --libs`])
 VLC_ADD_CXXFLAGS([wxwidgets],[`${WX_CONFIG} --cxxflags` `pkg-config sofia-sip-ua --
cflags`])
</pre></td></tr>
<tr><td>focus</td><td><pre>
dnl
dnl SIP focus module (sout package)
dnl
    VLC_ADD_PLUGINS([sipfocus])
    VLC_ADD_LDFLAGS([sipfocus],[`pkg-config sofia-sip-ua --libs`])
    VLC_ADD_CFLAGS([sipfocus],[`pkg-config sofia-sip-ua --cflags`])
</pre></td></tr>
</table>

figure 3.13 : code pour la configuration de la compilation

Les tests au Bureau d'étude ou au CNES ont été fait avec, au minimum, 2 clients et un focus. Un troisième client monté sur un ordinateur portable est de temps en temps ajouté au réseau.

Afin d'effectuer des optimisations , les programmes ont été exécuté dans une console.
la commande minimale pour lancer le client est *./vlc*, celle pour le focus est *./vlc --sip-host 0.0.0.0:5063 --sip-multicast 224.0.0.8 -I telnet*.
Avec ces réglages le système fonctionne mais le temps de latence est d'environ 3 s sur un réseau ethernet , 4 s si l'on compte le temps de transmission d'un satellite.
Ce temps de transfert rend impossible son usage pour une communication en vidéoconférence. Ce temps de transfert peut être amélioré en supprimant les caches que construisent les modules pour ce faciliter la tache. ce gain augmente le risque de perte dans le réseau et diminue les performances du logiciel. Comme le montre la figure 3.14 , le temps de latence peut être réduit a 400 ms sur un réseau ethernet, soit une seconde sur réseau satellite, temps suffisamment court pour rendre une conversation par vidéoconférence possible.

latence	coté client	coté focus
400 ms	*./vlc --sout-udp-caching 20 --udp-caching 1 --sout-ts-dts-delay 1 --rtp-late 1*	*./vlc -I telnet --sip-host 0.0.0.0:5063 --sip-multicast 224.0.0.8 -I telnet --sout-udp-caching 20 --udp-caching 1 --sout-ts-dts-delay 1*
1 200 ms	*./vlc --sout-udp-caching 20 --udp-caching 1 --sout-ts-dts-delay 200 --rtp-late 1*	*./vlc -I telnet --sip-host 0.0.0.0:5063 --sip-multicast 224.0.0.8 -I telnet --sout-udp-caching 20 --udp-caching 1 --sout-ts-dts-delay 200*
2 000 ms	*./vlc --sout-udp-caching 150 --udp-caching 150 --rtp-late 50*	*./vlc -I telnet --sip-host 0.0.0.0:5063 --sip-multicast 224.0.0.8 -I telnet --sout-udp-caching 50 --udp-caching 50*

figure 3.14 : temps de latence des différentes optimisation

Conclusion

Le développement de l'agent SIP permet démontrer la possibilité d'utiliser un protocole applicatif, en l'occurrence SIP, pour la gestion de réseau bas-niveau. Un tel usage permet d'apporter une qualité de service supérieure, en assurant nottament une authentification plus ou moins lourde du client. En effet si nous avons utilisé une authentification qu'avec l'IPfixe du client, il est tous a fait possible d'utiliser une clé de cryptage asymétrique.
Le protocole applicatif permet aussi la tarification de ces demandes, aspect non négligeable pour les opérateurs. Le principal inconvénient du système réside en ce que le récepteur satellite contient de l'informatique embarqué et donc devient une boite au lieu d'être implementé sur une carte électronique comme actuellement.

Les deux démonstrateurs que sont le service de télévision par IP et le service de vidéoconférence permettent de valider l'usage du routage multicast par SIP dans 2 cas communs différents.
La télévision par IP nécessite beaucoup de ressources et fait donc subir au routeur une forte charge. De plus ce service permet de facilement détecter les erreurs de transmission car l'image va sauter a ces moments là.
Le service de vidéoconférence quant a lui demande une plus faible chargé réseau mais augmente le nombre d'abonnements et désabonnements au services multicast. Ce service permet aussi de donner une forte charge au routeur, notamment sur les tables de routage.
Enfin, le service de vidéoconférence est construit dans l'esprit du modèle IMS. Ce modèle étant jeune, il a permis au CNES de se faire une idée sur les possibilités de ce modèle. Ce service reste tous de même assez simpliste. Ainsi il est évoqué la possibilité de crypter les flux ainsi que de pouvoir insérer les services de télévision sur IP dans une conférence.

Le projet devait être implementé en 5 mois. Malgré ce laps de temps, les timing furent difficilement respectés. Ce fut aussi mes premières installations chez le client. Ces 3 visites au Centre Spatial de Toulouse ont souvent mal commencés, mon inexpérience n'aidant en rien à la tâche. Pour autant j'ai toujours réussi à finaliser les objectifs des visites en temps et en heures, le tout au prix de poussés d'adrénaline. J'en ai retenu que, quelque soit la préparation de ces visites, l'imprévu est toujours présent.

Ce stage a permit de découvrir le métier d'ingénieur en recherche et développement, branche qui me convient bien. Le projet quant à lui est repris au CNES, son développement continue pour y inclure de nouveaux services tel le cryptage des flux et l'échange des clés par SIP et optimiser l'existant.

A. Bibliographie

[1] PIM : Protocol Independant Multicast - Sparse Mode RFC 2362 juin 1998

[2] MLD : Multicast Listener Discovery RFC 2710 octobre 1999

[3] IGMP : Intenet Group Management Protocol
version 1 : RFC 1112 aout 1989
version 2 : RFC 2236 november 1997
version 3 : RFC 3376 october 2002

[4] VLC : VideoLanClient http://www.videolan.org

[5] RTSP : Real Time Streaming Protocol (RFC 2326) April 1998

[6] RTP : Real Time Protocol (RFC 1889) Janvier 1996
RTCP : Real Time Control Protocol (RFC 1889)

[7] SDP : Session Description porotocol RFC2327 April 1998

[8] AAA : Authentication, Authorisation, Accounting

[9] RFC 4353 : J Rosenberg February 2006

[10] VLVC : Videolan Conference Client www.vlvc.net : projet de l'école EPITECH

[11] SOFIA-SIP: librairie implementant SIP http://opensource.nokia.com/projects/sofia-sip/

[12] Wxwidgets : librairie graphique multi plateforme www.wxwidget.org

[13] url : Uniform Ressource Locator

[14] Fedora : http://www.fedora-fr.org/
téléchargement de la distribution : http://mirrors.fedoraproject.org/publiclist/Fedora/7/

[15] Mandriva : http://www.mandriva.com/fr/
telechargement de la distribution : http://www.mandriva.com/fr/download/mandrivaone