IDIAP COMMUNICATION

Institut Dalle Molle
d'Intelligence Artificielle
Perceptive • CP 592 •
Martigny • Valais • Suisse

téléphone +41−27−721 77 11
télécopieur +41−27−721 77 12
adr.él. secretariat@idiap.ch
internet http://www.idiap.ch

IDIAP

Martigny - Valais - Suisse

DATAPUMP FULL-DUPLEX

Florian Salamin [a] François Corthay [a]
Olivier Bornet [b] Jean-Luc Cochard [b]

IDIAP–COM 96-02

DÉCEMBRE 1996

a Ecole d'Ingénieur du Valais rue du Rawyl 47 1950 Sion
b Institut Dalle Molle d'Intelligence Artificielle Perceptive Case Postale 592 1920 Martigny

École d'Ingénieurs du Valais (EIV)  Ingenieurschule Wallis (ISW)

Département électrotechnique

Travail de semestre 1996

Florian Salamin

Datapump Full-Duplex

Professeur : François Corthay
Mandant : IDIAP, Martigny

Sion, août 96

Table des matières

1. Introduction

1.1 Les serveurs vocaux interactifs

De nos jours, l'accès rapide et efficace à l'information est non seulement un facteur de réussite pour les entreprises, mais aussi pour tout un chacun. De plus, la souplesse de consultation de l'information est primordiale.

Jusqu'à ces dernières années, lorsque l'on appelait un service de renseignements téléphonique tel que le 111 en Suisse, un opérateur nous répondait. On lui formulait une requête, et celui-ci, à l'aide de son ordinateur, nous donnait dans la mesure du possible l'information souhaitée. Ces services étaient en général efficaces mais pas assez rapides. C'est pour cette raison que des recherches sont entreprises dans le domaine des serveurs vocaux interactifs (SVI).

Les SVI présentent plusieurs avantages vis-à-vis de la technique traditionnelle de diffusion des informations. L'appelant peut dialoguer directement avec la machine et ainsi la piloter par choix successifs jusqu'à l'obtention de l'information désirée. Ces dernières sont en outre disponibles 24 heures sur 24 et 7 jours sur 7. Le coût de l'information est considérablement réduit du fait que seuls un ordinateur et un média de communication sont nécessaires. On peut ainsi réaliser des services à valeur ajoutée élevée.

La mise en oeuvre d'un SVI ne demande que peu de matériel et est relativement simple (fig. 1). Un serveur vocal peut se présenter sous la forme d'un ordinateur relié au réseau téléphonique par une ligne numérique de type RNIS (Réseau Numérique à Intégration de Services). N'importe que abonné du réseau peut appeler le serveur depuis un téléphone analogique ou numérique.

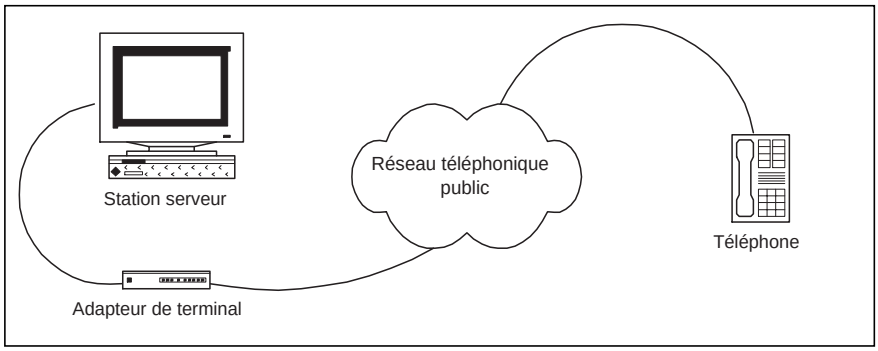

FIGURE 1. Principe de fonctionnement

La souplesse d'utilisation des SVI est aussi un énorme avantage au niveau de la mise sur pied d'un service vocal d'information. L'implantation de serveurs vocaux se réalise en douceur car cet outil ne demande quasiment aucun apprentissage de la part de l'utilisateur. Il est aussi sim-

ple d’appeler un serveur vocal que de converser au téléphone avec un ami. Le SVI se greffe de manière “naturelle”.

1.2 Développement des SVI en Suisse

L’avenir et le développement des serveurs vocaux interactifs en Suisse se présente de manière favorable. Des entreprises importantes acquièrent (PTT, UBS, SBS, Crédit Suisse) ou s’intéressent de près (CFF, Veillon) à cette technique. Ce n’est guère qu’un marché naissant, mais le fait que les leaders de l’économie suisse y soient présents est un signe encourageant.

Les SVI apportent à ces entreprises des avantages financiers indéniables ainsi qu’une augmentation de la productivité due à l’amélioration de la communication.

1.3 Les SVI au niveau mondial

Le marché mondial est en pleine expansion. Aux États-Unis et en France, le taux de croissance du marché lié aux serveurs vocaux interactifs est de l’ordre de 20 à 30% par an. Tant en France qu’aux États-Unis, on acquiert des SVI dans de nombreux secteurs d’activités économiques. Des banques, des sociétés de télécommunication, des sociétés de vente par correspondance, des offices du tourisme, des assurances, des services gouvernementaux, des instituts de météorologie et bien d’autres encore se lancent dans l’achat de serveurs vocaux interactifs.

En France, le marché du vocal interactif a généré un chiffre d’affaire de près de 400 millions de FS en 1992 et en Grande-Bretagne 500 millions de FS. Ce marché est considéré comme étant propice à une expansion importante à moyen terme.

1.4 Cahier des charges

Sun Microsystems commercialise un environnement de développement de SVI appelé SunXTL. Le but de ce projet est d’étendre les possibilités de la librairie datapump fournie avec l’environnement de développement SunXTL. Il faut lui ajouter la capacité de full-duplex, c’est-à-dire qu’elle doit être capable de recevoir et de diffuser de l’information simultanément. La capacité full-duplex doit permettre d’analyser les signaux émis et reçus pour mettre en évidence d’éventuels phénomènes d’écho sur une ligne téléphonique numérique.

Le projet est conduit sur station Sun. Il me faut donc aussi acquérir un minimum de connaissance du système d’exploitation UNIX pour pouvoir mener à bien ce projet.

2. Contexte

2.1 L'institut IDIAP à Martigny

L'Institut Dalle Molle d'Intelligence Artificielle Perspective (IDIAP) à Martigny dirige ses efforts dans les domaines des réseaux de neurones artificielles, de la vision artificielle et du traitement automatique de la parole. Dans ce dernier axe de recherche, l'IDIAP travaille aussi sur le développement de serveurs vocaux interactifs. L'un des objectifs est de mettre au point un serveur vocal que l'on peut piloter par voix humaine. Il serait ainsi possible de s'affranchir de l'utilisation du clavier de téléphone pour commander le serveur. Mon travail de semestre s'inscrit dans ce projet d'envergure mené par l'IDIAP. Il marque la première collaboration entre l'École d'Ingénieurs et l'IDIAP.

2.2 Station de travail UNIX

Le domaine des serveurs vocaux, bien qu'en forte croissance, est encore marginal. De ce fait, bien peu de constructeurs d'informatique et de fournisseurs de logiciels disposent d'outils de développement adéquats. Sun Microsystems est l'un des fabricants qui propose à la fois le hardware et le software nécessaire. Pour la mise en oeuvre de ce projet, il était donc indispensable de s'équiper en hardware de ce constructeur. Sur les conseils avertis de M. Olivier Bornet, ingénieur de recherche à l'IDIAP, l'École a acheté une station de travail Sun de type Ultra1 équipée de 64 méga-bytes de RAM (Random Access Memory), de 5 giga-bytes de disque dur, et d'un écran de 17 pouces. De plus, l'École à fait l'acquisition d'un carte d'interface RNIS, du système d'exploitation Solaris 2.5, et du compilateur C++ SPARCWorks de Sun. L'environnement de développement SunXTL est mis à disposition de l'École par l'IDIAP pour la durée du projet.

L'une de mes tâches importantes dans ce projet est d'acquérir la base de connaissances nécessaires avec le système d'exploitation UNIX pour pouvoir utiliser correctement la station de travail. J'ai aussi eu l'occasion de me familiariser quelque peu avec le système d'exploitation Solaris 2.5 et l'environnement graphique X Window.

2.3 Programmation C++

Le langage de programmation utilisé pour ce projet est le C++. Ce choix à été dicté par le fait que le logiciel de développement d'applications vocales proposé par Sun, ainsi que les exemples d'utilisation, ont été écrits en C++. Mes connaissances en programmation orientée objet m'ont permis de saisir relativement rapidement le fonctionnement des méthodes de base. J'ai dû, de plus, m'adapter au style de programmation

C++ propre au monde UNIX. Celui-ci diffère surtout quant à la surveillance du bon déroulement des différentes procédures.

2.4 Le RNIS

Le Réseau Numérique à Intégration de Services permet le transfert de données sur des lignes téléphoniques sous forme digitale. Un accès de base fournit à l'abonné 2x64kbit/s (canaux B) plus 1x16kbit/s (canal D) en full-duplex. L'utilisation d'une ligne numérique est judicieuse. En effet, les données sur la ligne sont déjà sous forme numérique. Elle sont donc directement lisibles par l'ordinateur. On évite ainsi, du côté du serveur, la numérisation du signal vocal.

L'interface RNIS installée dans la station de travail permet de relier l'ordinateur directement à la ligne numérique. Il devient ainsi possible de générer directement des données numériques sur la ligne. L'acquisition du flot d'entrée se fait de la même manière et devient ainsi stockable dans un fichier sur le disque dur de la station. L'environnement de développement fournit des outils pour piloter la carte au travers des drivers de celle-ci. Il est donc possible de prendre en charge un appel, de le mettre en attente, de le transférer, ou de raccrocher de manière logicielle.

2.5 Origine de l'écho

Le problème de l'écho est primordial dans un serveur vocal. En effet, il ne faut pas que l'ordinateur interprète l'écho du message diffusé sur la ligne téléphonique comme un ordre en provenance du locuteur.

Si l'on utilise des lignes numériques sur tout le trajet entre l'interlocuteur et le serveur, il n'y a pas d'écho possible. Par contre, si le locuteur possède une ligne analogique, la non adaptation parfaite du téléphone à la ligne risque de générer un écho (fig. 2). Celui-ci est numérisé dans le signal digital au central téléphonique public. Il est ainsi contenu dans le flot de bits qui partent en direction du serveur vocal.

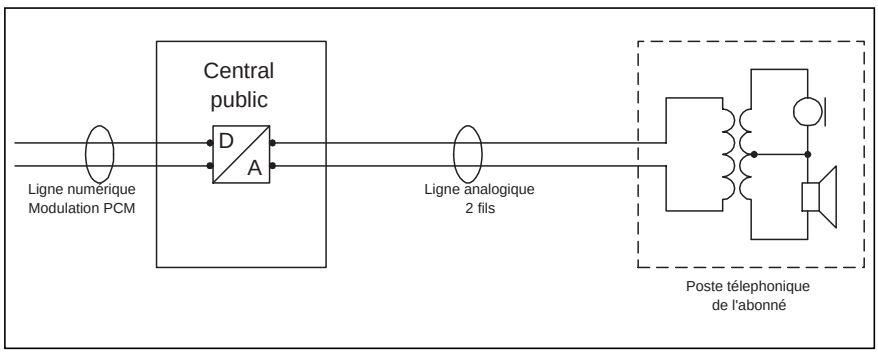

FIGURE 2. Provenance de l'écho

La réalisation de ce projet doit permettre de mettre en évidence ces phénomènes d’écho lors de liaisons avec un abonné raccordé par une ligne analogique. En fonction de leur importance, la décision sera prise d’appliquer ou non un algorithme de suppression de l’écho au niveau du signal sonore numérisé provenant du locuteur.

3. Objectifs du projet

3.1 Implémentation full-duplex

Le premier but est de me familiariser avec l'environnement de développement d'applications vocales SunXTL de SunSoft. Cet environnement fournit toute une série de classe C++ de relativement bas niveau qui permettent de gérer différents types de média. Toutes ces classes offrent au programmeur la possibilité de créer des applications vocales indépendantes du hardware de la machine utilisée (fig. 4).

Dans les exemples de code fournis avec l'environnement de développement SunXTL, il y a une application de répondeur téléphonique simple dénommée MailVox. Cet exemple permet de simuler un répondeur classique, c'est-à-dire que dans un premier temps, il diffuse un message à l'attention de l'appelant. Puis, dans un deuxième temps, il se met en mode d'enregistrement et stocke le message de l'appelant sur le disque dur ou tout autre périphérique connecté à l'ordinateur.

La modification du code de cet exemple doit permettre de lancer simultanément une opération de diffusion et une opération d'enregistrement (fig. 3). Ces deux tâches doivent s'appliquer au même flux de données et ne pas se gêner ou s'interrompre l'une l'autre.

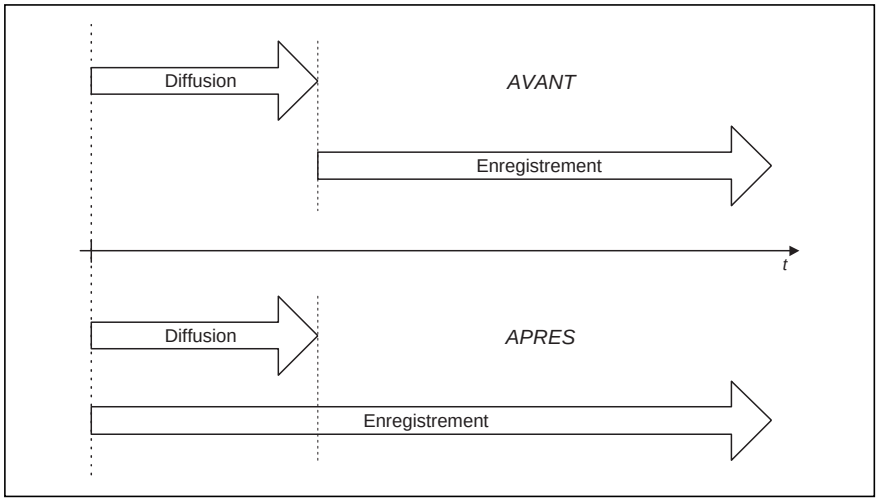

FIGURE 3. Modification full-duplex

De plus, il faut quelque peu remanier le code pour permettre de passer sous forme de paramètre le nom du fichier sonore à jouer sur la ligne.

Ces deux modifications majeures du programme doivent ainsi rendre possible la gestion en full-duplex du milieu de transmission.

Les différentes modifications du code sont présentées dans le chapitre 5 concernant les adaptations du logiciel.

4. SunXTL en bref

4.1 Buts de l'API SunXTL

Le design et l'architecture de SunXTL sont prévus pour atteindre plusieurs buts principaux:

- Fournir une API (Application Programming Interface) au programmeur pour le développement d'applications personnelles.
- Fournir une portabilité transparente entre les technologies analogiques, ISDN (Integrated Services Digital Network), ATM (Asynchronous Transfer Mode) et autres. L'application ne doit pas être recompilée lorsque l'on change de média ou de technologie.
- Proposer un accès facile aux services vocaux classiques. Elle fournit par exemple un outil de génération/détection DTMF (Dual Tone Multi Frequency).

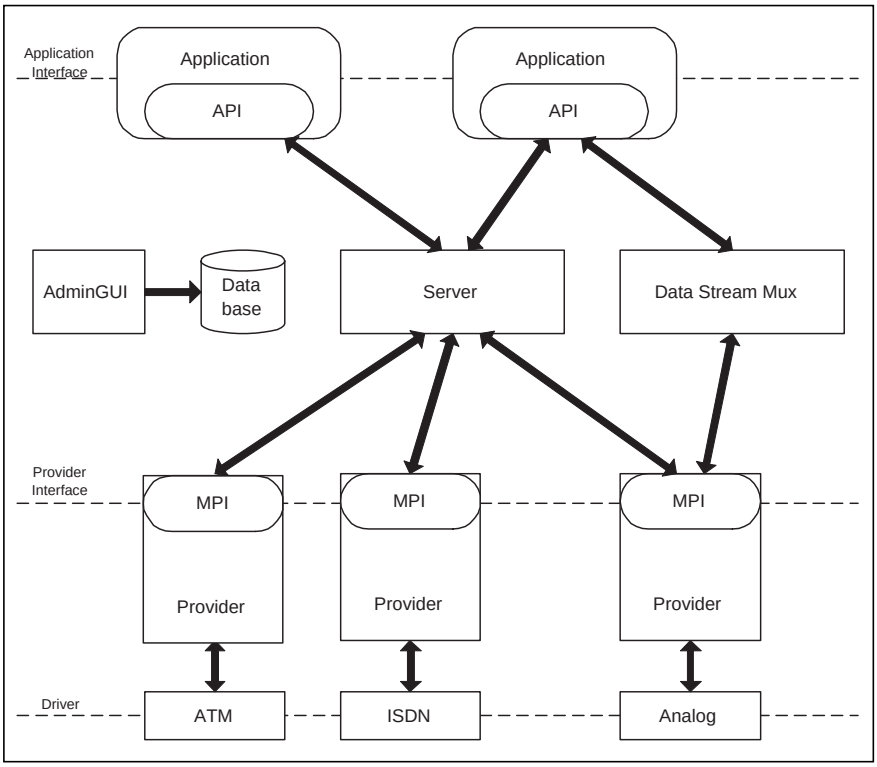

FIGURE 4. L'environnement SunXTL

L'API SunXTL est une interface orientée objet accessible en C++. Elle met à disposition du programmeur trois types de méthodes: requêtes asynchrones, commandes asynchrones et événements asynchrones. Différentes méthodes permettent la réponse à un appel, le transfert d'appel entre plusieurs applications, l'enregistrement ou la diffusion de messages. D'autres méthodes permettent de contrôler l'état d'un appel en cours et,

le cas échéant, de modifier cet état. L’environnement s’appuie, pour la gestion des appels, sur la notion de Provider qui est en quelque sorte un driver hardware fourni par le fabricant de la carte utilisée.

Le système SunXTL se comporte alors comme une couche intermédiaire entre le hardware et l’application. Le serveur SunXTL, de conserve avec les librairies de l’application et du provider, prend en charge le passage de message entre processus, la création et l’identification des objets, ainsi que l’appartenance d’un appel, la sécurité et la notification des événements asynchrones.

D’autre part, l’environnement met à la disposition du programmeur plusieurs méthodes annexes et classes utilitaires. Elles permettent de traiter des tableaux de bytes, des chaînes de caractères ainsi que des listes chaînées. Deux autres classes, dpDispatcher et dpIOHandlre, servent à traiter les événements produits par les entrées/sorties. A l’aide de tous ces outils, il est plus ou moins aisé de programmer une application vocale.

5. Adaptations du logiciel

5.1 Faculté de full-duplex

L'application de démonstration MailVox permet de simuler le comportement d'un répondeur téléphonique classique. C'est-à-dire qu'elle permet de diffuser un message puis, ensuite, d'enregistrer ce que l'interlocuteur dit.

Dans le cadre des serveurs vocaux interactifs, il faut autoriser l'interlocuteur à couper la parole au serveur vocal. Pour ce faire, il faut que l'enregistrement débute simultanément avec la diffusion du message. Le fichier généré par l'enregistrement est scanné en temps réel par un module de reconnaissance vocale qui, dès qu'il détecte un mot clé autorisé, génère un signal pour l'application MailVox. Ce signal est utilisé pour interrompre la diffusion et l'enregistrement en cours, et lancer la diffusion d'un nouveau message adapté à la requête de l'interlocuteur. De plus, il lance l'enregistrement d'un nouveau fichier pour permettre la reconnaissance de l'ordre suivant.

La plus importante modification du code se situe au niveau de la création et de la destruction des objets de gestion des flots d'entrée/sortie. En effet, il n'y a plus qu'un seul objet qui est créé pour toute la durée d'une communication. Cet objet prend en charge simultanément l'acheminement des données entrantes et sortantes. Ceci se fait en créant simultanément les tâches de diffusion d'annonce et d'enregistrement de message.

5.2 Passage de paramètre

Afin de pouvoir répondre correctement à la requête de l'interlocuteur, il faut pouvoir transmettre à l'application MailVox le nom du prochain fichier de son à diffuser sur la ligne. Il faut aussi transmettre au module de reconnaissance le nom du fichier d'enregistrement à traiter. Pour ce faire, il faut étendre le programme pour permettre le passage des paramètres des fichiers concernés.

6. Analyse des solutions

6.1 Le full-duplex

Le principal objectif du projet, à savoir doter l'application MailVox de la faculté de full-duplex, est atteint. Le serveur est ainsi capable de diffuser un message et d'enregistrer les informations provenant de la ligne RNIS simultanément.

Ceci a été rendu possible par les retouches apportées au code de MailVox. Les modifications sont les suivantes :

- La création de l'objet supportant un appel à été déplacé dans la méthode `configuration_ind` de la classe MsgCall. Ceci permet à l'objet de rester actif durant toute la durée de l'appel. Il est détruit seulement lorsque l'appel se termine. Cette opération de destruction à été implémentée dans le destructeur de la classe MsgCall.
- Le lancement des deux tâches de lecture et d'écriture sur le média de communication se fait dans la méthode `configuration_ind` de MsgCall. La tâche d'enregistrement à une légère priorité sur celle de diffusion pour s'assurer que l'on ne perd pas un ordre du locuteur.
- La plus grande partie des tests d'accès au média sont devenus caducs du fait que l'objet est créé et détruit en même temps que l'appel. Pour des questions de test, je les ai mis en commentaire.

Toutes les modifications apportées au code sont situées dans deux fichiers nommés msgcall.cc (annexe en page 12) et mailvox.cc (annexe en page 21). Les modifications de code apparaissent en caractères gras.

Il me faut noter une petite anomalie détectée dans le fonctionnement du logiciel modifié. Lorsque l'interlocuteur raccroche avant la fin de la diffusion de l'annonce, le serveur donne la tonalité d'occupé jusqu'à la fin de la diffusion du message. Il est donc inaccessible durant un certain temps qui dépend de la taille de l'annonce. Ceci semble provenir de la destruction de l'objet, mais je n'ai pas encore pu le vérifier.

6.2 Passage de paramètre

Le passage de paramètre pour la transmission du nom de fichier à diffuser n'est pas encore totalement implémenté. Je n'ai pas non plus mis en place de procédure de test pour m'assurer du bon fonctionnement de cette extension du programme.

7. Perspectives

Les modifications apportées au code de l’application MailVox vont me permettre de continuer ce travail au niveau du diplôme. En effet, l’application actuelle autorise la diffusion d’une annonce en simultané à l’enregistrement d’un message. L’analyse approfondie des messages émis et reçus vont m’aider à mettre en évidence la présence ou non d’un écho dans les transmissions du serveur vers un abonné sur ligne analogique. En fonction de ces résultats, une décision sera prise quant à la nécessité d’implémenter un module logiciel d’annulation de l’écho dans le cadre du serveur vocal interactif.

8. Annexes

8.1 Code msgcall.cc

```
////////////////////////////////////////////////////////////////////////////
//
// $Id: msgcall.cc,v 1.3 1996/05/28 16:00:00 fsalamin Exp $
//
// $Log: msgcall.cc,v $
// Revision 1.3  1996/05/28 16:00:00  fsalamin
// Change the place of DataPump object's creation and dstruction.
// Removed software control of the media access to allow read and write
// at the same time.
//
// Revision 1.2  1996/04/09 09:36:17  ljaccard
// Removed absolute path to 'mimencode'.
// Changed naming scheme for temporary mail file.
// Added system command to remove temp mail file.
//
// Revision 1.1.1.1  1996/04/02 15:37:04  ljaccard
//
////////////////////////////////////////////////////////////////////////////
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <strings.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <multimedia/libaudio.h>

#include <xtl/xtl.h>

#include "msgcall.h"

//
// FilePlayer takes a DataPump and "plays" data from an open file to
// the DataPump.  FilePlayer uses the bytes_secs (sample rate) and the
// length of the file to set a timer.  When the timer goes off the
// play *should* be finished.  FilePlayer does not verify that all the data
// was written successfully.  It just calls donePlaying() when the timer
// goes off.
//
FilePlayer::FilePlayer(
DataPump& input,
int source_fd,
int bytes_secs,
int bufsize)
: CopyWriter(input, source_fd, bufsize)
{
        fprintf(stderr, "FilePlayer created\n");
        dpDispatcher& d = dpDispatcher::instance();

// get length of file
int err;
struct stat source_stat;
if ((err = fstat(source_fd, &source_stat)) < 0) {
perror("FilePlayer:: could not stat source file descriptor");
}

// calculate the expected play time using the file length and sample rate
u_long play_time_usecs;
u_long play_time_secs;
double usecs_byte = (1000000 / bytes_secs);
play_time_secs = (source_stat.st_size / bytes_secs);
// original code, June 96 fsalamin
//play_time_secs = (source_stat.st_size / bytes_secs)/2;
play_time_usecs = (source_stat.st_size  % bytes_secs) * usecs_byte;
fprintf(stderr, "%d\n",source_stat.st_size);
fprintf(stderr, "%d\n",play_time_secs);
fprintf(stderr, "%d\n",play_time_usecs);
//d.startTimer(play_time_secs/2, play_time_usecs, this);
d.startTimer(5, 0, this);
}

FilePlayer::~FilePlayer()
{
dpDispatcher& d = dpDispatcher::instance();
```

```
// Stop the timer
d.stopTimer(this);
}


void
FilePlayer::donePlaying()
{
}

void
FilePlayer::timerExpired(long, long)
{
// Note: does *not* verify successful writing of data
this->donePlaying();
}


//
// AudioWriter is used to play a Sun audio file to a Sun audio device.
//
//audio = DataPump attatched to a Sun audio STREAMS device
//audio_file = file descriptor to a Sun audio file.
//bufsize = size of buffer to use when copying from file to device
//
//NOTE:AudioWriter is designed to work on the Sun /dev/audio
//(audio (4)) interface.  Some providers configurations
//may not support the ioctls used by AudioWriter.  This
//example is only known to work on the xtlp_sun_5e5
//configured as INPUT = STREAM, and OUTPUT = STREAM.
//
//The AudioWriter client_data is a pointer to a composite object that
//is using the AudioWriter.  It is used in donePlaying() to notify the
//composite object that the play has finished.  By default the
//AudioWriter assumes the composite object is MsgCall, but AudioWriter
//could be subclassed, and donePlaying could be overriden to call
//a different composite object.
//
AudioWriter::AudioWriter(
DataPump& audio,
int audio_file,
void* data,
int bufsize)
: CopyWriter(audio, audio_file, bufsize),
  audio_device(audio.fd()),
  client_data(data)// a MsgCall
{
dpDispatcher& d = dpDispatcher::instance();

// clear eof flag of audio device, so we can detect end of play
audio_info_t            audioinfo;
AUDIO_INITINFO(&audioinfo);// init info struct
audioinfo.play.eof = 0;// clear eof flag
audio_setinfo(audio.fd(), &audioinfo);

//d.startTimer(0, 100, this);
}

AudioWriter::~AudioWriter()
{
dpDispatcher& d = dpDispatcher::instance();

// Stop the timer
// d.stopTimer(this);
}


void* AudioWriter::clientData() { return client_data; }

//
// The outgoing message is done playing.
//
void
AudioWriter::donePlaying()
{
// Specific to using MsgCall, subclass and override this function
// to call a different composite object
MsgCall* msgcall = (MsgCall*) clientData();
msgcall->stopPlayAnnc();
msgcall->playDone();
}

//
// AudioWriter determines that the outgoing message has been completely
```

```
// played if the audioinfo.play.eof bit associated with the fd
// is 1.
//
boolean_t
AudioWriter::playedZeroLengthBuffer(int audfd)
{
        audio_info_t            audioinfo;
        int                     err;


        err = audio_getinfo(audfd, &audioinfo);
        if ((err == 0) && (audioinfo.play.eof >= 1)) {
                /* done playing */
return(B_TRUE);
        }

        return(B_FALSE);
}

//
// Interrupt handler for the timer set in the AudioWriter constructor.
// Check to see if the outgoing message is done playing.  If it hasn't
// reset the timer.
//
void
AudioWriter::timerExpired(long, long)
{
if (playedZeroLengthBuffer(audio_device) == B_TRUE)
donePlaying();
else {
// reset timer
dpDispatcher& d = dpDispatcher::instance();
// d.startTimer(0, 100, this);
}
}


// MsgCall will play a message (from a file referenced by the announce
// paramter. or record a message (to a file referenced by the message
// paramter) over a call's data channel.  It plays a message when
// MsgCall::playAnnc is called and record a message when MsgCall::recordMsg
//is called.
//
MsgCall::MsgCall(
Xtl::Exception e,
XtlCallState&cs,
char* announcement,
char*message,
char*recipient)
: XtlCall(e, cs), _audioFd(-1), _anncFd(0), _msgFd(0),audioPump(0),
player(0), recorder(0), record_requested(B_FALSE),
play_requested(B_FALSE), dtmf(0), _anncAudioFile(announcement),
_msgAudioFile(message), _recipient(recipient)
{
}

MsgCall::~MsgCall()
{
fprintf(stderr, "MsgCall: cleaning up\n");

// stop any records or plays that may be going on
stopPlayAnnc();
stopRecordMsg();

// modification 7 May 96, fsalamin
// Avoid to send e-mail message during the tests.
//sendMailMsg();
// end modification

// modification 28 May 96, fsalamin
// Destuct audioPump object when the call is finished.
delete audioPump;
audioPump = NULL;
fprintf(stderr, "audioPump destruct\n");
// end modification

close(_audioFd);
fprintf(stderr, "close audioFd\n");
close(_anncFd);
fprintf(stderr, "close anncFd\n");
close(_msgFd);
fprintf(stderr, "close msgFd\n");

// modification 28 May 96, fsalamin
```

```
/*
// delete temporary file
// Removed during the tests
char cmd[256];
sprintf(cmd, "rm -f %s", _msgAudioFile());
system(cmd);
*/
// end modification
}

void
MsgCall::error_ind(Request req, Xtl::Error err, XtlKVList&)
{
XtlStringrequest = string(req);
XtlStringerror = Xtl::string(err);

fprintf(stderr, "Request %s failed: %s\n", request(), error());
}


//
// find_kv_string_pair is an auxiliary function used to search an XtlKVList
// for a kv pair whose key is <key> and whose value (an XtlKVString)
// is <value>.
//
boolean_t
find_kv_string_pair(
XtlKVList kv_list,
const XtlString &key,
const XtlString &value)
{
XtlStringthis_value;

kv_list.reset();
while (kv_list.next(key) == B_TRUE) {
if (kv_list.get(this_value) == B_FALSE) {
continue;
}
if (this_value == value) {
return(B_TRUE);
}
}
return(B_FALSE);
}



//
// When the call's data stream is reconfigured to include INPUT=STREAM and
// OUTPUT=STREAM then play the greeting message or record an
// imcoming message depending on whether record_requested or play_requested
// is set (set during playAnnc and recordMsg, respectively).
//
// If the data stream configuration is not INPUT=STREAM and OUTPUT=STREAM,
// there is trouble.  The record or play machinery should be cleaned up.
// But for simplicity sake, we just exit instead.
//
void
MsgCall::configuration_ind(XtlKVList& config)
{
// ignore null configuration event
if (!config.first())
return;

if ((find_kv_string_pair(config, XtlConfigInputK, XtlConfigStreamC) != B_TRUE) &&
    (find_kv_string_pair(config, XtlConfigOutputK, XtlConfigStreamC) != B_TRUE)) {
fprintf(stderr, "Warning: Unsupported configuration:\n");
config.print(STDERR_FILENO);
fprintf(stderr, "Exiting\n");
exit(1);
}

// pull input and output fds out of congfiguration, and save them
config.first("INPUT_STREAM_FD");
u_longinput_fd;
config.get(input_fd);
config.first("OUTPUT_STREAM_FD");
u_longoutput_fd;
config.get(output_fd);

// MsgCall code assumes we have a bidirectional stream.  If
// we don't get a bidirectional stream back, exit
//
if (output_fd != input_fd) {
```

```
fprintf(stderr,
    "detected unidirectional stream, exiting.\n");
exit(1);
}

// save valid audio fd
_audioFd = int(output_fd);

// modification 21 May 96, fsalamin
// Creates object DataPump for this particular call. It will be destructed
// only when the call is finished.
audioPump = new DataPump(_audioFd);
ioctl(_audioFd, I_FLUSH, FLUSHRW);
fprintf(stderr, "audioPump created\n");
// end modification

// modification 21 May 96, fsalamin
/*
       // Sanitiy test
       // Removed to allow playing and recording on the media at the same time.
if ((record_requested == B_TRUE) && (play_requested == B_TRUE)) {
fprintf(stderr,
    "internal error: recording and playing simultaneously\n");
exit(1);
}

// proceed with existing request
if (record_requested) {
start_record();
} else if (play_requested){
start_play();
}
*/
// end modification

// modification 7 May 96, fsalamin
// proceed with existing request
// Call start_play and start_record methods at the same time.
if (record_requested) start_record();
if (play_requested) start_play("./Messages/absent.au");
// end modification
}



//
// Start recording (or prepare the data configuration to record) the
// callers message.
//
// RETURNs B_FALSE if object is already busy, B_TRUE on success.
//
boolean_t
MsgCall::recordMsg()
{
XtlKVListread_write_stream_config;

// modification 21 May 96, fsalamin
/*
       // Removed to allow playing and recording on the media at the same time.
// If we are already doing something return an error
Exception excp;
if ((play_requested) || (record_requested) ||
    !call_state(excp).media_channel_available()) {
return B_FALSE;
}
*/
// end modification

// modification 7 May 96, fsalamin
// Just checks if the media is available.
// If we are already doing something return an error
Exception excp;
if (!call_state(excp).media_channel_available()) return B_FALSE;
// end modification

record_requested = B_TRUE;

// _audioFd is a file descritor to the call's data channel.
// If it is not a valid fd, configure the data stream
// to INPUT=STREAM and OUTPUT=STREAM mode.  configuration_ind
// will set _audioFd approriately.
// Otherwise start recording.
if (_audioFd < 0) {
read_write_stream_config.add(XtlConfigInputK, XtlConfigStreamC);
```

```
read_write_stream_config.add(XtlConfigOutputK, XtlConfigStreamC);

configuration_req(read_write_stream_config);
} else {
start_record();
}

return B_TRUE;
}

//
// Start playing (or prepare the data configuration to play) the outgoing
// (greeting) message
//
// RETURNs B_FALSE if object is already busy, B_TRUE on success.
//
boolean_t
MsgCall::playAnnc(XtlString _playAudioFile)
{
XtlKVListread_write_stream_config;

// modification 21 May 96, fsalamin
/*
// If we are already doing something return an error
        // Removed to allow playing and recording on the media at the same time.
Exception excp;
if ((play_requested) || (record_requested) ||
    !call_state(excp).media_channel_available()) {
return B_FALSE;
}
*/
// end modification

// modification 7 May 96, fsalamin
// Just checks if the media is available.
// If we are already doing something return an error
Exception excp;
if (!call_state(excp).media_channel_available()) return B_FALSE;
// end modification

play_requested = B_TRUE;

// _audioFd is a file descritor to the call's data channel.
// If it is not a valid fd, configure the data stream
// to INPUT=STREAM and OUTPUT=STREAM mode.  configuration_ind
// will set _audioFd approriately.
// Otherwise start playing.
if (_audioFd < 0) {
read_write_stream_config.add(XtlConfigInputK, XtlConfigStreamC);
read_write_stream_config.add(XtlConfigOutputK, XtlConfigStreamC);

configuration_req(read_write_stream_config);
} else {
start_play(_playAudioFile);
    //("./Messages/absent.au");
}

return B_TRUE;
}

//
// Start recording an incoming message to _msgAudioFile (as initialized
// in MsgCall constructor).
//
void
MsgCall::start_record()
{
if ((_msgFd =
    open(_msgAudioFile(), O_WRONLY|O_TRUNC|O_CREAT, 0664)) < 0) {
perror("MsgCall:: could not open message file");
disconnect_req();
}

//
// Need to put proper audio header on file.
//
Audio_hdr audheader;
char *callinfo = "";// It would be more interesting if the calling
// party info was contained in this string.

// Everything except the encoding is fixed.
audheader.sample_rate = 8000;// XXX - Sloppy, not always ISDN
audheader.samples_per_unit = 1;// XXX - Sloppy, not always ISDN
audheader.bytes_per_unit = 1;// XXX - Sloppy, not always ISDN
```

```
audheader.channels = 1;// XXX - Sloppy, not always ISDN
audheader.endian = AUDIO_ENDIAN_BIG;
audheader.data_size = AUDIO_UNKNOWN_SIZE;

// Determine audio encoding
Xtl::Exception exc;
XtlString enc;
this->call_state(exc).format().first(XtlFormatEncodingK);
if (exc == EXCEPTION_SUCCESS)
this->call_state(exc).format().get(enc);
if ((exc == EXCEPTION_SUCCESS) && (enc == XtlFormatALawC)) {
audheader.encoding = AUDIO_ENCODING_ALAW;
} else if ((exc == EXCEPTION_SUCCESS) && (enc == XtlFormatULawC)) {
audheader.encoding = AUDIO_ENCODING_ULAW;
} else {
// XXX - This should never happen if A-law and mu-law are
// XXX   the only encodings ever used.
fprintf(stderr,
    "Cannot determine audio encoding from call format\n");
this->call_state(exc).format().print(fileno(stderr));
// XXX -- changed default to Alaw, for Europe.
audheader.encoding = AUDIO_ENCODING_ALAW;
}

(void)audio_write_filehdr(_msgFd, &audheader, 0, 0);

// Ready to record incoming voice data
fprintf(stderr, "Recording message ...\n");

// Use a DataPump and CopyReader to do the work

// modification 21 May 96, fsalamin
// audioPump is created only once when the configuration_ind method of
// MsgCall in called.
// audioPump = new DataPump(_audioFd);
// ioctl(_audioFd, I_FLUSH, FLUSHRW);
// end modification

       recorder = new CopyReader(*audioPump, _msgFd);

// add software dtmf filter
Exception excp;
dtmf = new DTMFHandler(*audioPump, AudFormat(call_state(excp).format()));
}

//
// Subclass FilePlay to create UlawPlayer which sets the sample rate
// and buffer size for Ulaw audio and calls MsgCall::playDone()
// methods when the play finishes
//
class UlawPlayer : public FilePlayer {
public:
UlawPlayer(
    DataPump& input,
    int source_fd,
    MsgCall& msgcall,
    int bytes_secs = 8000,
    int bufsize = 1024)
    : FilePlayer(input, source_fd, bytes_secs, bufsize),
      _msgcall(msgcall) {}
virtual ~UlawPlayer() {}
virtual void donePlaying();
private:
MsgCall&_msgcall;
};
void
UlawPlayer::donePlaying()
{
_msgcall.stopPlayAnnc();
_msgcall.playDone();
}


//
// Start playing the greeting message _anncAudioFile
//
void
 MsgCall::start_play(XtlString _playFile)
{
// open outgoing announcement file
if ((_anncFd <= 0)
    && (_anncFd = open(_playFile(), O_RDONLY)) < 0) {
perror("could not open announcement file");
disconnect_req();
```

```
}

// NOTE:
//We assume that the file is a Sun audio format file, and
//the device is a Sun audio device interface.  This is
//not very portable.  For example, it would break if
//the device was alaw and the file was ulaw.  Ideally,
//we would get the format of the data, and verify the
//device can handle the format.  Then if there were
//an incompatiblity we could either convert the file format
//to something the device could handle or change
//the device's format mode.
//

// seek past audio file header
Audio_hdraudio_header;
if (AUDIO_SUCCESS !=
    audio_read_filehdr(_anncFd, &audio_header, NULL, 0)) {
fprintf(stderr, "WARNING: not a Sun audio file\n");
}

fprintf(stderr, "Playing message ...\n");

// create Xtl data stream pump, and flush Xtl data stream

// modification 21May 96, fsalamin
// audioPump is created only once when the configuration_ind method of
// MsgCall in called.
// audioPump = new DataPump(_audioFd);
// ioctl(_audioFd, I_FLUSH, FLUSHRW);
// end modification

//audioPump->flushWrite();

player = new UlawPlayer(*audioPump, _anncFd, *this);

// add software dtmf filter
Exception excp;
dtmf = new DTMFHandler(*audioPump, AudFormat(call_state(excp).format()));
}


void
MsgCall::sendMailMsg()
{
    char header[256];
    sprintf(header, "Subject: Audio Message\n\
From: MailVox\n\
Mime-Version: 1.0\n\
Content-Type: audio/basic\n\
Content-Description: message.au\n\
Content-Transfer-Encoding: base64\n");

    char temp_mail_msg[256];
    sprintf( temp_mail_msg, "%s.msg", _msgAudioFile() );
    FILE *message = fopen(temp_mail_msg, "w");
    fprintf(message, "%s", header);
    fclose(message);

    char cmd[256];
    sprintf(cmd,
    "mimencode %s >> %s",
    _msgAudioFile(),
    temp_mail_msg);
    system(cmd);

    sprintf(cmd,
    "mail %s < %s",
    _recipient(),
    temp_mail_msg);
    system(cmd);
    sprintf(cmd, "rm -f %s", temp_mail_msg);
    system(cmd);
}

//
// When done recording the incoming message, add an audio header
// and clean up the recording machinery.
//
void
MsgCall::stopRecordMsg()
{
// clean up must be done in this order
close(_msgFd);
```

```
_msgFd = -1;

#if 0
// Audio file is already in place
if (recorder) {
// add header to raw audio file
charcmd[256];
sprintf(cmd,
    "audioconvert -f voice -p -i rate=8k,channels=mono,encoding=alaw %s",
    _msgAudioFile());
if (system(cmd) < 0) {
perror("could not convert message file");
}
}
#endif

delete dtmf;
dtmf = NULL;

delete recorder;
recorder = NULL;

// modification 28 May 96, fsalamin
// audioPump is destroyed only when the destructor of MsgCall
// is called, i.e. at the end of a call.
// delete audioPump;
// audioPump = NULL;
// end modification

        record_requested = B_FALSE;
}

//
// Clean up machinery used to play the greeting
//
void
MsgCall::stopPlayAnnc()
{
        // fprintf(stderr, "MsgCall stopPlayAnnc\n");
// clean up must be done in this order
close(_anncFd);
_anncFd = -1;

delete dtmf;
dtmf = NULL;

delete player;
player = NULL;

// modification 28 May 96, fsalamin
// audioPump is destroyed only when the destructor of MsgCall
// is called, i.e. at the end of a call.
// delete audioPump;
// audioPump = NULL;
// end modification

play_requested = B_FALSE;
}

//
// Detect when DTMF goes off (is released)
//
void
DTMFHandler::detectedToneUp(char c)
{
fprintf(stderr, "detected up %c\n", c);
}

//
// Detect a string of DTMF tones (called when the "#" is detected)
//
void
DTMFHandler::detectedToneString(XtlString& tones)
{
fprintf(stderr, "detected tone string %s\n", tones());
}

//
// Detect when DTMF goes on (is pressed)
//
void
DTMFHandler::detectedToneDown(char c)
{
fprintf(stderr, "detected down %c ...", c);
```

```
}
```

8.2 Code mailvox.cc

```
////////////////////////////////////////////////////////////////////////////
//
// $Id: mailvox.cc,v 1.3 1996/05/28 16:00:00 fsalamin Exp $
//
// $Log: mailvox.cc,v $
// Revision 1.3  1996/05/28 16:00:00  fsalamin
// Start methods playAnnc and recordMsg at the same time. This allows
// to store the data stream from channel B and play a message simultaneously.
//
// Revision 1.2  1996/04/03 09:22:25  ljaccard
// A call object is now removed from the list when the provider
// receives a DISCONNECT_EVENT with a call reference.
// Conceptually cleaner and need for a ‘removeCall’ method.
//
// Revision 1.1.1.1  1996/04/02 15:37:04  ljaccard
//
//////////////////////////// DESCRIPTION ////////////////////////////////////
//
// Machine immediately answers all incoming calls.
// If the called number is found in the database, the corresponding
// greeting message will be played, followed by the recording of a
// vocal message. The latter is then sent to the associated recipient,
// via E-mail.
//
//////////////////////////// INCLUDES ///////////////////////////////////////
#include <xtl/xtl.h>
#include <xtl/xtlprovider.h>
#include <xtl/xtlcall.h>
#include <xtl/xtlcallstate.h>
#include <Dispatch/sldispatcher.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#include “msgcall.h”

//////////////////////////// GLOBAL /////////////////////////////////////////
//
// The database file must have the following format:
// <Called#><TAB><e-mail_address><TAB><filename><NEWLINE>
// which means that <filename> is to be played into the B-Channel,
// whenever <Called#> has been reached, and then send the recorded
// message to <e-mail_address>.
//
char*db_filename = “./db_answer”;

//////////////////////////// INTERFACE //////////////////////////////////////

class MyProvider : public XtlProvider {
  public:
    MyProvider(Xtl::Exception& err, XtlString pname) : XtlProvider(err, pname) {}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
    virtual void offer_ind(XtlCallState& cs, XtlKVList&);
    virtual void call_event_ind(XtlCallState&, CallEvent, Cause, XtlKVList&);
  private:
    XtlKVListlist_of_calls;
};

class MyCall : public MsgCall {
  public:
    MyCall(
Xtl::Exception& e,
XtlCallState& cs,
char* announcement,
char* message,
char* recipient)
: MsgCall(e, cs, announcement, message, recipient)
    {
_provider = (MyProvider*)(cs.provider());
    }
    virtual void playDone();
    virtual void callDone();
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
    virtual void event_ind(CallEvent ev, Cause, XtlKVList&);
  private:
    MyProvider* _provider;
```

```
};

//////////////////////////// IMPLEMENTATION ////////////////////////////////

void
MyProvider::offer_ind(XtlCallState& call, XtlKVList&)
{
    // A call has been offered for ownership and the MyProvider object will
    // attempt to claim it by creating a new call object using the call state
    // of the offered call.

    Xtl::Exception e;

    FILE *db_file;
    char db_line[256];// a whole line of the database
    char db_nbr[15];// phone number
    char db_addr[32];// e-mail address
    char db_msg[128];// name of the file to play
    char record_msg[128];// name of the file to record a message

    // open the database file
    if ((db_file = fopen(db_filename, "r")) == NULL)
perror("MyProvider::offer_ind");

    // pick up incoming call
    if (call.state() == INCOMING) {

// Print local and remote addresses (i.e. called and calling #s)
fprintf(stderr,"Incoming call from %s to %s.\n",
call.remote_address()(), call.local_address()());

boolean_t found = B_FALSE;

while ((fgets(db_line, 256, db_file) != NULL) && !found) {
    sscanf(db_line, "%s\t%s\t%s", db_nbr, db_addr, db_msg);
    found = (XtlString(db_nbr) == call.local_address());
}

if (found) { // the call is in the DB, and must be answered

    // prepare temporary filename, to store recorded message
    sprintf(record_msg, "./msg.%ld.au", (time_t)time(0));

    // Create and store the call in a KV list:
    // KEY = call reference; VALUE = address of the call object
    fprintf(stderr, "Create call.\n");
    list_of_calls.add
(call.call_reference(),
 (u_long)(new MyCall(e, call, db_msg, record_msg, db_addr)));

    fprintf(stderr, "Added call '%s'\n", call.call_reference()());

    // print out the list
    fprintf(stderr, "List of calls:\n");
    list_of_calls.print(2);
    fprintf(stderr, "-------- end --------\n");
}
else // not in the database
    fprintf(stderr, "Ignore call.\n");
    }
}

void
MyProvider::activated_ind(XtlKVList&)
{
    // This notification is called when the provider object has finished
    // its internal setup and is ready for commands or further notifications.
    Exception excp;
    fprintf(stderr, "Provider process started (%s).\n", (name(excp))());

    // register for offer events
    enable_offer_event_req(B_TRUE);

    // listen for any call being disconnected
    listen_req(DISCONNECT_EVENT);
}

void
MyProvider::deactivated_ind(XtlKVList&)
{
    // This notification is called when the provider process
    // associated with this provider object dies.
    fprintf(stderr, "Provider process died.\n");
    exit(1);
```

```
}

void
MyProvider::call_event_ind(XtlCallState& cs, CallEvent ev, Cause, XtlKVList&)
{
    XtlCallReference ref = cs.call_reference();

    // if a call gets disconnected, remove it from the calls' list
    if (ev == DISCONNECT_EVENT) {
list_of_calls.first(ref);
list_of_calls.remove();

fprintf(stderr, "Removed call '%s'\n", ref());

fprintf(stderr, "List of calls:\n");
list_of_calls.print(2);
fprintf(stderr, "-------- end --------\n");
    }
    else
fprintf(stderr,
"MyProvider:: Ignoring %s provider event\n", Xtl::string(ev)());
}

////////////////////////////////////////////////////////////////////////////

void
MyCall::activated_ind(XtlKVList&)
{
    // The call object now has ownership of the call.

    Exception e;
    XtlKVList format;

    format.add(XtlFormatClassK, XtlFormatVoiceC);
    format.add(XtlFormatEncodingK, XtlFormatALawC);
    format.add(XtlFormatSampleSizeK, 8);
    format.add(XtlFormatSampleRateK, 8000);

    switch (call_state(e).state()) {
      case INCOMING:
answer_req();
break;

      case UNKNOWN:
      case IDLE:
      case DISCONNECTED:
      case INVALID:
// Do nothing ...
break;

      default:
fprintf(stderr, "MyCall: given call in unsupported state\n");
    }

    // The phone call may have been previously answered by another
    // call object but then had its ownership offered.  This call
    // object now has claimed ownership of the call and will play
    // the greeting.  It is not necessary to call "answer_req()" the
    // call because it is not incoming or in one of the inactive states.
    // Just check if the data channel is available, if so start play.
    if (call_state(e).media_channel_available())
      {

// modification 7 May 96, fsalamin
// Starts playing and recording message at the same time.
        playAnnc("./Message/laisser.au");
        recordMsg();
// end modification

      }

}

void
MyCall::deactivated_ind(XtlKVList&)
{
    callDone();
}

void
MyCall::callDone()
{
    Exception e;
```

```
    // Stop recording the call's data stream
    stopRecordMsg();

    // This code only manages one call at a time, delete
    // only call and reset global pointer
    delete this;
}

void
MyCall::playDone()
{
  // modification 7 May 96, fsalamin
  // Don't start recording a new message when the play is done.
  // recordMsg();
fprintf(stderr, "playDone\n");
playAnnc("./Message/interne.au");
  // end modification
}

void
MyCall::event_ind(CallEvent ev, Cause, XtlKVList&)
{
    switch(ev) {
      case CHANNEL_AVAILABLE_EVENT:
// Now attempt to acquire a data stream and play a greeting
// fprintf(stderr, "Numero 4 mailvox\n");
playAnnc("./Message/laisser.au");
break;

      case CONNECT_EVENT:
      case ALERTING_EVENT:
      case PROCEEDING_EVENT:
// Ignored
break;

      case DISCONNECT_EVENT:
callDone();
break;

      case CHANNEL_UNAVAILABLE_EVENT:
      default:
// Unsupported
fprintf(stderr, "MyCall: unsupported %s event\n", Xtl::string(ev)());
    }
}

////////////////////////////////////////////////////////////////////////////
int
main(int argc, char* argv[])
{
    MyProvider::Exceptionerr;
    MyProvider*Pv;
    dpDispatcher::instance(new dpSLDispatcher);
    dpDispatcher&     d = dpDispatcher::instance();

    // Parse arguments
    if (argc > 2)  {
fprintf(stderr, "usage: %s [provider]", argv[0]);
exit(1);
    }

    // Assume that the second argument (if it exists) to the program
    // is the provider name.
    char*pvname;
    if (argc == 2) {
pvname = argv[1];
    } else {
pvname = NULL;
    }

    // Connect to the provider specified by pvname.  If pvname is NULL
    // then XTL will attempt to start up the default system provider.
    Pv = new MyProvider(err, pvname);
    if (err != MyProvider::EXCEPTION_SUCCESS) {
fprintf(stderr, "could not connect to provider\n");
exit(1);
    }

    // Sit in the dispatcher loop
    while(B_TRUE)
d.dispatch();

    return 1;
}
```

8.3 Procédure de connexion Telnet

La possibilité de faire du télétravail sur les ordinateurs de l'IDIAP depuis l'École d'Ingénieurs à été mise à profit lors de ce travail de semestre. Voici décrite succinctement la manière d'établir une connexion Telnet sous un environnement X Window depuis une station Unix HP du laboratoire de télécommunication.

Tout d'abord, sur la station locale, il faut autoriser l'affichage de la station distante sur l'écran local. Ceci se fait par la commande xhost. Puis il faut spécifier à la station distante le nom de la machine sur laquelle les informations doivent être affichées.

Pour illustrer les opérations de connections, la station locale a pour nom *stlabo8.eiv.vsnet.ch* et la station distante *nendaz.idiap.ch*.

Voici point par point les opérations requises pour la connection:

1. Sur la station locale: *xterm &*
2. Sur la station locale: *xhost + nendaz.idiap.ch*
3. Sur la station locale: cliquer dans la nouvelle fenêtre xterm et taper *telnet nendaz.idiap.ch*
4. À l'invite de la session Telnet, entrer le login et le password
5. Dans la session Telnet: *setenv DISPLAY stlabo8.eiv.vsnet.ch:0.0*
6. Lancer les applications X distantes de la même manière que les applications locales

Lors de la déconnection, il faut quitter la session Telnet et annuler l'autorisation d'affichage sur la station locale.

Voici point par point les opérations requises pour la déconnection:

7. Dans la session Telnet: taper exit
8. Sur la station locale: fermer le fenêtre xterm
9. Sur la station locale: *xhost - nendaz.idiap.ch*

Le texte en italique représente ce que l'utilisateur doit taper et voir apparaître à l'écran.

Table des figures