Soluții Java pentru transmisie vocală în timp real utilizând protocolul UDP

Furtună Titus Felix
Academia de Studii Economice
Bucureşti – S1, Piaţa Romană nr. 6
titus@ase.ro

Dârdală Marian
Academia de Studii Economice
Bucureşti – S1, Piaţa Romană nr. 6
dardala@ase.ro

REZUMAT
Tehnologiile Java acoperă la momentul actual toate aspectele ce țin de dezvoltare software. Prezentul articol propune o soluție software pentru transmiterea în timp real a vocii la distanță utilizând tehnologii Java. Soluția propusă poate fi o alternativă de comunicare directă, simplă şi utilă în cazul persoanelor cu dizabilității de vedere, fiind capabilă să suplinească, în anumite situații, tehnologii mai complicate şi mult mai costisitoare cum ar fi de exemplu sinteza vocală.

Lucrarea cuprinde o parte teoretică în care sunt prezentate tehnologiile Java folosite - biblioteca Java Sound API şi elemente de programare în rețea utilizând protocolul UDP- şi o parte aplicativă în care este sunt prezentate elemente ale soluției software propuse – arhitectura aplicației şi cod semnificativ.

Cuvinte cheie
Dispozitiv audio, captura audio, playback, protocol.

Clasificare ACM
H5.2. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCERE

Java Sound este interfața de programare de nivel scăzut care asigură circulația fluxurilor audio în cadrul sistemului. Java Sound API se referă atât la aspectele legate de procesarea vocii (captura şi redarea audio) cât şi la interfața digitală muzicală (Musical Instrument Digital Interface – MIDI). În consecință, în Java Sound API au fost dezvoltate două pachete:

- *javax.sound.sampled*, pentru procesare voce,
- *javax.sound.midi*, pentru sinteza muzicală (MIDI).

Aplicațiile Java folosesc în general două protocoale pentru transportul datelor în rețea:

- protocolul TCP (Transmission Control Protocol), protocol orientat pe conexiune, legat de paradigma client-server;
- protocolul UDP (User Datagram Protocol), protocol neorientat pe conexiune care permite cea mai rapidă trimitere a datelor prin entități numite pachete.

Pentru transmiterea datelor audio prin rețea cele mai potrivite soluții sunt cele oferite de protocoalele neorientate pe conexiune. Pentru aplicațiile în timp real, cum sunt şi aplicațiile audio, garantarea unei transmisii sigure nu este neapărat o necesitate.

Folosind elemente de gestiune a fluxurilor audio prin Java Sound API şi elemente de programare în rețea pe baza protocolului UDP, pot fi create componente software reutilizabile pentru transmisie în timp real a vocii la distanță. Aceste componente pot intra apoi în componența oricărei aplicații Java adaptate la situații particulare.

ASPECTE GENERALE PRIVIND DATELE AUDIO

Noțiunea centrală în modelarea semnalului vocal este cea de ***eşantion*** sau de mostră (sampled audio)[1][2]. Eşantioanele sunt elemente succesive ale unui semnal. Semnalul audio este o undă de sunet. Un microfon preia semnalul acustic ca semnal analog, iar un convertor analog-digital transformă acest semnal (analog) în formă digitală. În figura 1 este prezentat un moment scurt din inregistrarea unui sunet.

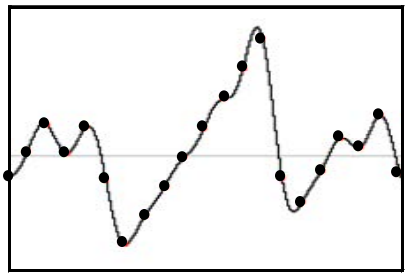

Figura 1. Eşantioane într-o subsecvență de semnal

În acest grafic este evidențiată variația amplitudinii presiunii aerului în funcție de timp. Eşantioanele sunt aşadar valori succesive ale amplitudinii sunetului măsurate la anumite intervale de timp. Rata de eşantionare reprezintă numărul de valori înregistrate pe unitatea de timp şi determină exactitatea aproximării digitale a semnalului analog[1][2]. În general, eşantioanele provin dintr-o înregistrare de sunet, dar pot rezulta şi dintr-un proces de generare sintetică a sunetului. Termenul *sampled audio* se referă la tipul de dată şi nu la originea ei.

Java Sound API are o structură flexibilă care se poate adapta la o configuratie hardware specifică. Biblioteca este astfel construită încât să permită instalarea diferitelor componente audio care să fie apoi accesate de API-ul Java[4].

PROCESAREA VOCALĂ UTILIZÂND PACHETUL *SAMPLED*

Pachetul *javax.sound.sampled*[4] este în mod esențial focalizat pe transportul audio: captura audio şi redarea audio. Sarcina principală a biblotecii Java Sound API constă în gestionarea fluxurilor audio în şi înafara sistemului. Această sarcina implică deschiderea dispozitivelor de intrare-ieşire, gestionarea bufferelor de date în timp real şi mixarea mai multor fluxuri audio într-un singur flux (intrare sau ieşire). Java Sound API furnizează metode de conversie între diferite formate de date audio şi pentru citirea/scrierea din/în fişiere audio cu

formate specifice. Transportul semnalului audio poate fi realizat atât în mod flux (*streaming*) utilizând buffere cât şi în modul *in-memory*, fără buffere (*unbuffered*). Un flux audio este un şir de bytes care ajung la o destinaţie mai mult sau mai puţin cu aceeaşi rată cu care ei urmează a fi utilizaţi (redaţi). În modul *streaming*, atât în cazul intrărilor cât şi în cazul ieşirilor audio, nu este neapărat necesar să fie cunoscută aprioric lungimea sunetului. Java Sound API permite transportul fără buffere doar în cazul în care datele audio nu sunt prea mari, adică pot încape în memorie în totalitate.

Pentru redarea sau captura audio folosind Java Sound API sunt necesare cel puţin trei elemente: un format pentru date, un mixer audio şi o linie de sunet.

Formatul pentru datele audio include atât formatele pentru datele audio propriu-zise (*Data Formats*) cât şi formatele pentru fişierele audio (*File Formats*). Formatele pentru date arată modul în care sunt interpretaţi o serie de bytes ai unei secvenţe audio neprelucrate, preluate dintr-un fişier sau înregistrate la microfon[4].

Formatul audio este modelat prin clasa *AudioFormat*. Un obiect *AudioFormat* include următoarele atribute:

- Tehnica de codare. De regulă se utilizează modulaţia în impulsuri codificate (PCM - Pulse Code Modulation)
- Număr de canale - 1 pentru mono, 2 pentru stereo;
- Rata de eşantionare - numărul de eşantione pe secundă, pe canal;
- Numărul de biţi pe eşantion;
- Rata cadrelor (numărul de cadre);
- Mărimea cadrelor în octeţi. Un cadru conţine datele pentru toate canalele la un anumit moment de timp. Pentru date codate PCM, cadrul este echivalent cu un set de esantioane simultane, în toate canalele. În acest caz, numărul de cadre este egal cu numărul de eşantione iar mărimea cadrului in octeţi este egală cu mărimea eşantionului în octeţi înmulţită cu numărul de canale. Pentru alte tipuri de codări, numărul de cadre poate fi diferit de numărul de eşantioane.
- Ordinea de memorare a octeţilor atunci când numărul de biţi pe eşantion este 16: big-endian sau litle-endian.

Un format de fişier specifică structura fişierului de sunet, incluzând datele audio brute (şirul de octeţi reprezentând eşantionale) şi informaţii prin care se specifică structura fişierului. Fişierele de sunet sunt construite după diferite standarde precum: WAVE (Waveform Audio Format), AIFF (Audio Interchange File Format) sau AU (AUdio). Aceste tipuri de fişiere de sunet au diferite structuri. Antetul fişierului care conţine informatii descriptive este în general urmat de zona de date propriu-zise. Există şi formate audio în care alternează informaţiile descriptive cu "porţiuni" de eşantioane.

În Java Sound API formatul de fişier e modelat prin clasa *AudioFileFormat*. Un obiect *AudioFileFormat* conţine:

- Tipul fişierului;
- Lungimea fişierului în octeţi;
- Lungimea datelor audio in fişier (număr de cadre);
- Un obiect *AudioFormat* care specifică formatul datelor audio propriu-zise conţinute în fişier.

Clasa utilitară *AudioSystem* furnizează metode pentru citirea şi scrierea sunetelor în diferite formate de fişiere şi pentru conversia dintre diferite formate de date. Unele metode permit accesarea conţinutul fişierului la nivel de flux prin obiecte de tip *AudioInput Stream*. Clasa *AudioInput Stream* este o subclasă a clasei generice *InputStream* care încapsuleaza un şir de octeţi ce pot fi citiţi secvenţial dintr-un fişier. La facilităţile superclasei, clasa *AudioInputStream* adaugă informaţii despre formatul datelor audio prin obiectul *AudioFormat*. Prin citirea unui fişier audio cu un flux *AudioInputStream* se obţine accesul imediat la eşantion sărind peste antet.

Un dispozitiv audio este o interfaţă software scrisă pentru un dispozitiv fizic de intrare/ieşire pentru sunet[4]. În Java Sound API dispozitivele sunt reprezentate de obiectele *Mixer*. Scopul unui mixer este de a gestiona unul sau mai multe fluxuri audio de intrare sau ieşire. În sensul cel mai general mixerul amestecă mai multe fluxuri de intrare într-un singur flux de iesire. Intrările şi ieşirile dispozitivelor audio sunt implementate prin obiectele de tip *Port*. Exemple obişnuite de porturi sunt intrarea de la microfon sau ieşirea prin speaker. Atât obiectele de tip *Mixer* cât şi cele de tip *Port* sunt derivate ale conceptului mai larg de linie (*Line*). O linie este un element al unei „conducte" digitale audio, adică o modalitate prin care se realizează transportul audio în/din sistem. De regulă linia este o cale de intrare/ieşire într-un/dintr-un mixer. Toate aceste elemente sunt implementate în Java Sound API printr-o ierarhie de interfeţe având la bază interfaţa *Line*:

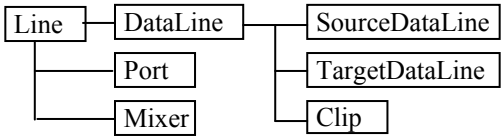

Figura 2. Ierarhia interfeţelor *Line*

Interfaţa de bază, *Line*, descrie funcţionalităţile minimale pentru toate tipurile de linii:

- controlul asupra unor aspecte ale sunetului precum volumul în decibeli (*gain*), balansul (stânga-dreapta) , ecoul sau alte efecte acustice (*reverb*) şi viteza de redare;
- informaţii de stare (închis sau deschis);
- răspunsul la anumite evenimente.

Interfaţa *DataLine* adaugă funcţionalităţi noi interfeţei *Line*, necesare procesului de captură sau redare, reflectate prin metodele:

```
void start();
```
// Permite startarea sau reluarea operaţiilor de I/O asupra datelor din buffer

```
void stop();
```
// Permite oprirea operaţiilor de I/O

```
void drain();
```
// Eliberează bufferul intern blocând firul de execuţie curent

```
boolean isRunning();
```
// Arată dacă linia este în lucru. O linie este în stare *running* de la prima operaţie de I/O

```
boolean isActive();
```
// Arată dacă linia este activă, adică se află în timpul unei operaţiuni de I/O

Un obiect *TargetDataLine* primeşte date de la mixer. Interfaţa *TargetDataLine* furnizează metode pentru deschiderea liniei şi pentru citirea datelor din bufferul liniei:

```
void open(AudioFormat format)throws
LineUnavailableException
void open(AudioFormat format, int bufferSize)
throws LineUnavailableException
int read(byte[] b,int off, int len)
```

Obiectele *SourceDataLine* primesc date pentru redare. Interfaţa *SourceDataLine* furnizează metode pentru deschiderea liniei, identice celor de la interfaţa *TargetDataLine* şi pentru scrierea datelor în bufferul intern al liniei pentru redare:

```
int write(byte[] b,int off, int len)
```

În Java Sound API există o ierarhie de clase statice care oferă informaţii cu privire la linii. Această ierarhie este suprapusă ierarhiei interfeţelor *Line* prezentate în figura 2. De exemplu, clasa *Mixer.Info* oferă detalii despre un mixer instalat în sistem (producător, nume, descriere, versiune).

Java Sound API pune la dispoziţie o clasă utilitară care acţionează ca intermediar în obţinerea accesului la resusele audio ale sistemului. Aceasta este clasa *AudioSystem*. De exemplu, clasa *AudioSystem* furnizează informaţii cu privire la mixerele instalate în sistem sau metode care permit obţinerea unei linii directe, fără o legătură explicită cu mixerul. Prin urmare obţinerea unei linii pentru captură sau redare se poate face în două moduri:

- obţinerea mixerului şi apoi a liniei;
- obţinerea directă a liniei prin metode *AudioSystem*.

Metodele care permit obţinerea unui mixer sunt:

```
public static Mixer.Info[] getMixerInfo(); //
```

Întoarce un masiv cu obiectele de tip Mixer.Info asociate mixerelor instalate în sistem

```
public static Mixer getMixer(Mixer.Info info); //
```

Întoarce obiectul mixer specificat

Metodele care permit obţinerea liniilor sau a informaţiilor despre linii, dintr-un obiect *Mixer* sunt:

```
Line[] getSourceLines();
Line[] getTargetLines();
Line.Info[] getSourceLineInfo();
Line.Info[] getTargetLineInfo();
```

Pentru obţinerea directă a unei linii este invocată metoda clasei *AudioSystem*:

```
public static Line getLine(Line.Info info) throws
LineUnavailableException;
```

TRANSMITEREA DATELOR UTILIZÂND PROTOCOLUL UDP

Transmiterea datelor prin reţea se realizează cu ajutorul unui canal de comunicaţie numit ***socket***[5]. În cazul utilizării protocolului UDP *socket*-urile sunt implementate prin clasa *DatagramSocket*. Datele sunt transmise sub forma unor pachete numite datagrame. Acestea specifică adresa destinatarului. Pentru datagrame este dezvoltată clasa *DatagramPacket*. Există două tipuri de constructori pentru clasa *DatagramPacket*, după scopul utilizării pachetelor: pentru transmitere sau pentru recepţionare[5].

Pentru trimiterea de pachete, o aplicaţie se poate limita la următoarele operaţiuni:

- crearea unei obiect de tip *DagramSocket* invocând un constructor fără parametrii;
- crearea unui obiect de tip *DatagramPacket* de tip transmitere cu datele transmise;
- trimiterea pachetului prin invocarea metodei *send* a obiectului *socket*.

Pentru recepţionarea de pachete etapele minimale sunt:

- crearea unui obiect *DatagramSocket* conectat la un anumit port;
- crearea unui obiect *DatagramPacket* pentru recepţia unei datagrame;
- recepţia datagramei şi preluarea datelor din datagramă.

ARHITECTURA APLICAŢIEI PENTRU TRANSMISIE VOCE

O aplicaţie pentru transmisie vocală în timp real preia semnalul audio de la o staţie din reţea (staţie emiţător) şi îl distribuie spre alte staţii în care este redat (staţii receptor). Într-o arhitectură simplificată aplicaţia include două module: un modul de captură audio şi transmisie pentru staţia emiţător şi un modul de recepţie şi redare (palyback) pentru staţia receptor. Ambele module sunt realizate pe fire de execuţie independente. Pentru o funcţionalitate mai mare se poate introduce o staţie intermediară, un fel de server de voce care să preia semnalul de la mai multe staţii emiţător şi să-l distribuie spre toate staţiile care solicită acest lucru (staţii receptor). Rezultă o arhitectură de tip stea precum cea din figura 3.

O astfel de arhitectură are însă dezavantajul unor pierderi de timp provocate de preluarea şi retransmiterea pachetelor de către serverul de voce.

În tabelul 1 este prezentată o metodă a unei clase Java cu crearea şi startarea unui fir de execuţie care efectuează recepţia pachetelor audio din reţea şi redarea lor. Metoda prezentată în tabelul 2 crează şi startează un fir de execuţie pentru captura audio, constituirea pachetelor de date audio şi transmiterea acestora prin reţea. Firele de execuţie rulează pâna la schimbarea stării unei variabile booleene cu rol de semafor.

Tabelul 1. Playback

```
private void playback(){
  Thread fir=new Thread() {
  public void run() { sfarsit=false;
    try{
      DataLine.Info dataLineInfo = new DataLine.Info( SourceDataLine.class,audioFormat);
      if(!AudioSystem.isLineSupported(dataLineInfo)){
        JOptionPane.showMessageDialog(null, "Format incompatibil!");return; }
      sourceDataLine = (SourceDataLine) AudioSystem.getLine( dataLineInfo);
      sourceDataLine.open(audioFormat); sourceDataLine.start();
      while(!sfarsit){ s.receive(d);
        if(d.getLength()>0){ byte a[]=d.getData(); sourceDataLine.write(a,0,a.length); }
        }
        sourceDataLine.stop();sourceDataLine.close(); }
      catch(Exception e){JOptionPane.showMessageDialog(null,"Eroare receptie!!");}
    }
  };
  fir.start();
}
```

Tabelul 2. Captura audio

```
private void captureAudio() {
  try { audioFormat = getAudioFormat();
    DataLine.Info dataLineInfo = new DataLine.Info( TargetDataLine.class, audioFormat);
    targetDataLine = (TargetDataLine) AudioSystem.getLine( dataLineInfo);
    targetDataLine.open(audioFormat); targetDataLine.start();
    Thread captura=new Thread() { byte tempBuffer[] = new byte[1000];
      public void run() { stopCapture = false;
        try {
          while(!stopCapture) {
            int cnt = targetDataLine.read( tempBuffer, 0, tempBuffer.length);
              if(cnt > 0) {
                DatagramPacket rasp=new DatagramPacket(tempBuffer,tempBuffer.length,
                  InetAddress.getByName(jt_server.getText().trim()), 2000);
                s.send(rasp); }
              }
         }
         catch (Exception e) { JOptionPane.showMessageDialog(null,"Eroare transmisie!"); }
       }
     };
     captura.start();
  }
  catch (Exception e) { JOptionPane.showMessageDialog(null,"Eroare captura!"); }
}
```

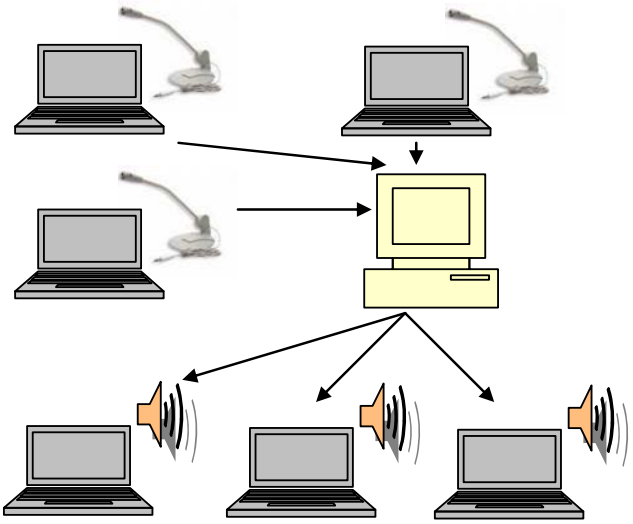

Figura 3. Arhitectura aplicaţiei

CONCLUZII

Transmisia la distanţă a semnalului vocal oferă oamenilor un mijloc direct de comunicare, iar prin intermediul sistemului de calcul acest mijloc este unul ieftin pentru că utilizează reţele de calculatore. Eficienţa comunicării, din punct de vedere al costului de transmisie este semnificativ mai mare în cazul în care utilizatorii se află în ţări sau pe continente diferite, datorită accesului ieftin la Internet.

Pe de altă parte, există biblioteci şi componente sofware care pot mixa transmisia vocală între diferite tipuri de reţele (reţeaua de telefonie pentru voce şi reţelele de calculatoare pentru date), ceea ce lărgeşte sfera de comunicare vocală între oameni.

REFERINŢE

[1] Vijay M., Douglas W., *Digital Signal Processing Handbook*, Chapman & Hall, 2005 ;

[2] Stein., J., *Digital Signal Processing: A Computer Science Perspective*, John Wiley & Sons, 2003

[3] ***, *Java Reference Documentation*, http://java.sun.com/reference/docs/

[4] ***, *Java Sound API*, http://java.sun.com/products/java-media/sound/

[5] ***, *Java Networking Features*, http://java.sun.com/javase/6/docs/technotes/guides/net/index.html